DAVIDE SPATARO

SEAMLESS ACCELERATION OF NUMERICAL REGULAR GRID METHODS ON MANYCORE SYSTEMS

SEAMLESS ACCELERATION OF NUMERICAL REGULAR GRID METHODS ON MANYCORE SYSTEMS

A dissertation submitted to attain the degree of

DOCTOR OF PHILOSOPHY of UNIVERSITÀ DELLA CALABRIA
(Ph.D. Università della Calabria)

presented by

DAVIDE SPATARO

born on 14 February 1990

Supervisors

Prof. William Spataro
Prof. Donato D'Ambrosio

2017

DOI:

Alla Mia Famiglia e a K.

ABSTRACT

Over the last two decades, a lot has changed regarding the way modern scientific applications are designed, written and executed, especially in the field of data-analytics, scientific computing and visualization. Dedicated computing machines are nowadays large, powerful agglomerates of hundreds or thousands of multi-core computing nodes interconnected via network each coupled with multiple accelerators. Those kinds of parallel machines are very complex and their efficient programming is hard, bug-prone and time-consuming. In the field of scientific computing, and of modeling and simulation especially, parallel machines are used to obtain approximate numerical solutions to differential equations for which the classical approach often fails to solve them analytically making a numerical computer-based approach absolutely necessary. An approximate numerical solution of a partial differential equation can be obtained by applying a number of methods, as the finite element or finite difference method which yields approximate values of the unknowns at a discrete number of points over the domain. When large domains are considered, big parallel machines are required in order to process the resulting huge amount of mesh nodes. Parallel programming is notoriously complex, often requiring great programming efforts in order to obtain efficient solvers targeting large computing cluster. This is especially true since heterogeneous hardware and GPGPU has become mainstream. The main thrust of this work is the creation of a programming abstraction and a runtime library for seamless implementation of numerical methods on regular grids targeting different computer architecture: from commodity single-core laptops to large clusters of heterogeneous accelerators. A framework, `OpenCAL` had been developed, which exposes a domain specific language for the definition of a large class of numerical models and their subsequent deployment on the targeted machines. Architecture programming details are abstracted from the programmer that with little or no intervention at all can obtain a *serial, multi-core, single-GPU, multi-GPUs and cluster of GPUs* `OpenCAL` application. Results show that the framework is effective in reducing programmer effort in producing efficient parallel numerical solvers.

SOMMARIO

Durante l'ultimo ventennio il modo in cui le moderne applicazioni scientifiche sono scritte e progettate è cambiato radicalmente, specialmente in campi come la data-analytics, il calcolo scientifico e la visualizzazione. Systemi di calcolo dedicate sono oggigiorno grandi e potenti agglomerati di centinaia o migliaia di nodi di calcolo interconnessi l'uno all'altro tramite reti ad alta velocità ed ognuno dotato di uno o più acceleratori. Questa macchine parallela sono complesse e la loro programmazione efficiente è difficile, bug-prone e richiede tempo e denaro. Nel campo del calcolo scientifico e della modellazione e simulazione specialmente, macchine parallele sono usate per ottenere soluzioni numeriche approssimate a equazioni differenziali per cui gli approcci classici, basati sul calcolo differenziale, falliscono nel risolverle analiticamente rendendo le soluzioni numeriche calcolate tramite sistemi computerizzati assolutamente indispensabili. Le soluzioni numeriche per equazioni differenziali possono essere ottenute attraverso l'utilizzo di una serie di metodi, tra cui il metodo degli elementi o delle differenze finite, quest'ultimo ad esempio, fornisce valori approssimati della incognite in un numero discreto e finito di punti nel dominio. Al crescere delle dimensioni dei domini, crescono le dimensioni delle macchine parallele che sono necessarie a processare l'incredibile numero di punti che costituisce la griglia di nodi che discretizza il dominio. La programmazione parallela è notoriamente difficile, e richiede uno sforzo da parte del programmatore per ottenere solvers efficienti e che siano in grado di essere eseguiti su grandi cluster di calcolo. Questo è diventato un problema ancora più centrale da quando la GPGPU è diventata mainstream. Il contributo principale di questa tesi è la creazione di una programming abstraction e una libreria runtime per l'implementazione seamless di modelli numerici su griglia regolare che possano essere eseguiti su svariate architetture, a partire da personal computers o laptops fino a grandi cluster di calcolo eterogenei dotati di acceleratori. Il framework `OpenCAL` è il risultato di questo lavoro, ed è sostanzialmente composto da un domain specific language per la definizione e l'implementazione di una famiglia di modelli numerici e il loro successivo deployment sulla macchina target. Dettagli architetturali sono totalmente astratti dal programmatore che con pochissimo sforzo di programmazione può ottenere diverse versioni della stessa applicazione `OpenCAL`: *seriale, multi-core, single-GPU, multi-GPU, distributed memory-multi-GPU*. I risultati mostrano che il framework è realmente in grado di ridurre lo sforzo di programmazione per lo sviluppo di solvers numerici paralleli efficienti.

ACKNOWLEDGEMENTS

ACKNOWLEDGMENTS

I would like to thank my supervisors **William Spataro** and **Donato D'Ambrosio** for their invaluable support and encouragement and for believing in me from the beginning of this Ph.D. I really wish every Ph.D. student in the world have supervisors like them. I would also like to thank **Alessio De Rango** and **Paola Arcuri** who were of invaluable help and for being such a great study partners.

I would like to thank NVIDIA corporation for providing the *GTX980 GPU* and *Tesla K40* coprocessor I have used to carry out my research.

I would also like to thank also to **Prof. Filippo Melascina** from the Institute of BioEngineering of the University of Edinburgh (UK) who kindly gave access to the microscopy equipment and to **Alice Mari** who was of great help in providing the experimental raw data and subsequently in analyzing the results obtained in chapter 7.

I would also like to thank **Prof. Jin Ooi** for giving the opportunity to being a research visiting student at the University of Edinburgh for 10 months (October 2016 - July 2017) working on the VELaSSCO EU project and, **Prof. Gihan R. Mudalige**, who supervised me during a three months visiting period (June- September 2017) at University of Warwick, while I was investigating a novel MPI implementation of an algorithm for solving tridiagonal linear systems within the context of the OPS framework.

I would also like to thanks all the people who I have met during these years and from which I have learned and, with whom I have enjoyed together, so many things.

A special thank to all friends, especially to **Vincenzo Curti** with whom I have shared so many important moments during this journey.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Over the last two decades, a lot has changed regarding the way modern scientific applications are designed, written and executed, especially in the field of data-analytics, scientific computing and visualization. The main reasons behind these changes are that the size of the problems that scientists try to tackle is nowadays much bigger and the amount of available raw data that can be analyzed has widened the spectrum of computing applications. Data analytics and big-data techniques are applied in pretty much every field of science and have been exploited effectively also by governments and corporate organizations.

Traditionally, performance improvements in computer architecture have come from cramming more functional units onto silicon, increasing clock speeds and transistors number. Coupled with increasing clock speeds, CPU performance has until recently doubled every two years. But it is important to acknowledge that this trend cannot be sustained indefinitely or forever. Increased clock speed and transistor number require more power and consequently generate more heat, at the point that the heat emitted from a modern processor, measured in power density, rivals the heat emitted by a nuclear reactor core! But the demand of speed did not stop in over the years and is not going to stop in the near future, and thus, from these reasons comes the necessity of relying heavily on parallel architectures. Multi-core CPUs (2, 4, 8, 12, up to 40) are ubiquitous at the point that even smart-phones are proper multi-core machines. Dedicated computing machines are nowadays large, powerful agglomerates of hundreds or thousands of multi-core computing nodes interconnected via network each coupled with multiple accelerators. Those kinds of parallel machines are very complex and their efficient programming is hard, bug-prone and time-consuming.

In the field of scientific computing, and of modeling and simulation especially, parallel machines are used to obtain approximate numerical solutions to differential equations which describe a physical system rigorously, as for example for the *Maxwell's* equations at the foundation of classical electromagnetism or the *Navier-Stokes* for fluid dynamics. The classical approach, based on calculus, often fails to solve these kinds of equations analytically, making a numerical computer-based approach absolutely necessary. An approximate numerical solution of a partial differential equation can be obtained by applying a number of methods, as the finite element or finite difference method which yields approximate values of the unknowns at a discrete number of points over the domain. When large domains are considered, large parallel machines are required in order to process the resulting huge amount of mesh nodes. Parallel programming is notoriously complex, often requiring great programming efforts in order to obtain efficient solvers targeting large computing cluster. This is especially true since heterogeneous hardware and GPGPU has become mainstream.

The main thrust of this work is the creation of a programming abstraction and a runtime library for seamless implementation of numerical methods on regular grids targeting different computer architecture: from commodity single-core laptops to large clusters of heterogeneous accelerators. A framework, `OpenCAL` had been developed, which exposes a domain specific language for the definition of a large class of numerical models and their subsequent deployment on the targeted machines. Architecture programming details are abstracted from the programmer that with little or no intervention at all can obtain a *serial, multi-core, single-GPU, multi-GPUs and cluster of GPUs* `OpenCAL` application. Results show that the framework is effective in reducing programmer effort in producing efficient parallel numerical solvers.

The rest of the thesis is organized as follows: Chapters 3 and 4 introduce the main targeted numerical models and parallel architectures, respectively. Chapters 5 describes `OpenCAL`, its implementation and different versions, usage and performance on a number of benchmarks, while Chapter 6 introduces the multi-GPU and distributed memory version of `OpenCAL` and evaluates its performance on three kinds of applications, each with different computational and memory requirements. Eventually, Chapters 7 and 8 introduce other HPC numerical modeling and simulation applications that have been investigated. In particular, Chapter 7 introduces a specialized framework based on `OpenCAL` for tracking particle-like objects from a time-lapse video which has been applied to analyze the motility of the *B. subtilis* bacterium, while Chapter 8 investigates multi-agent collective system acceleration on GPU. Appendix A refers to an ad-hoc stream-compaction algorithm specifically targeting NVIDIA newest hardware that was investigated during the work on `OpenCAL`.

Part I

Uniform Grid Numerical Methods and Parallel Computing

2 CELLULAR AUTOMATA

There are many interesting phenomena … which involve a mixture of physical phenomena and physiological processes, and the full appreciation of natural phenomena, as we see them, must go beyond physics in the usual sense. We make no apologies for making these excursions into other fields, because the separation of fields, as we have emphasized, is merely a human convenience, and an unnatural thing. Nature is not interested in our separations, and many of the interesting phenomena bridge the gaps between fields.

— Richard Feynman

Physical system are usually composed by many components that interact in a complex net of causes and consequences that is often hard or impossible to describe in its entirety analytically. Even if each single components is simple, extremely complex behaviors emerge naturally due to the resulting effect of their cooperative interaction. Much has been discovered about the nature of the components in natural systems, but little is known about the way those components interact as the overall complexity is observed. Such systems are often described by partial differential equations, which are hard to solve analytically especially when they are non-linear and consequently require alternative, or approximate solutions (see Chapter 3). This Chapter introduces *Cellular Automata*, *CA*, that have been proved to be suitable for the modellation and simulation of a wide class of complex physical systems, in particular those ones constructed from **many identical** components, each (ideally and relatively) simple, but when together, capable of complex behaviour [1, 2]. As can be seen from the number of the published papers on the topic, cellular automata have been applied in a wide range of classes of problems from gas [3] and fluid turbulence [4] simulation to macroscopic phenomena [5] like epidemic spread [6], snowflakes and lava flow [7–9]. Cellular Automata were first investigated by *S. Ulam* when he was trying to understand the growth of crystals using a lattice network and at the same time by *John von Neumann* who adopted CA in order to study self-reproduction [10]; CA were pretty much unknown until the 1970 when the famous *Conway's* game of life [11] appeared, and since then they have been widely studied from a theoretical view point until they were proved capable of computational universality[1] [12]. They have been mainly adopted, after 1980's, as a computational parallel model due to their intrinsically parallel nature [13].

2.1 INFORMAL DEFINITION

A *cellular automata* (CA) is a mathematical model that consists of a discrete lattice of sites and a value, the state, that is updated in a sequence of subsequents discrete timestamps (steps) according to some rules that depend on a neighbor sites of the cell. Hence CA describe systems whose the overall behavior and evolution may be exclusively described

1 Logical gates can be simulated using the simple rules of the *Game of Life* combining special patterns as *gliders* and *guns* that appears naturally and frequently in the Conway's CA.

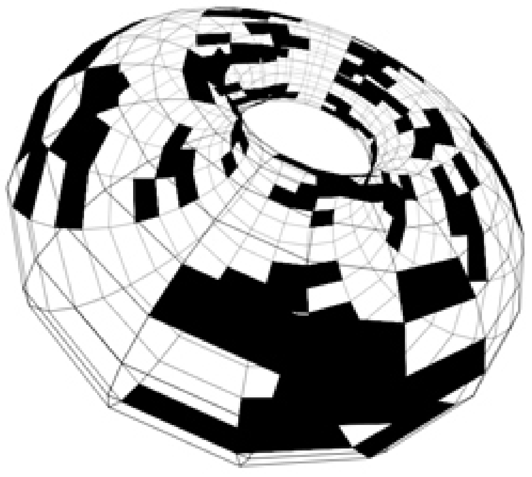


FIGURE 2.1: 3D cellular automaton with toroidal cellular space.

on the basis of local interactions [14], property also called *centrism*. The most stringent and typical characteristic of the CA model is the restriction that the local function does not depend on the time t or the cell site i: a cellular automaton has homogeneous space and time behavior. It is for this reason that CA are sometimes referred to as *shift-dynamical* or *translation invariant* systems. From another point of view we can say that in each lattice site resides a deterministic finite (state) automaton (DFA) [15] that takes as input only the states of the cells in its neighborhood (see Figure 2.4 and Section 2.2.1).

2.1.1 *Cellular space dimension and geometry*

The cellular space is a *discrete* d-dimensional lattice of sites (see figure 2.2). For $1D$ automaton the only way to discretize the space is in a one-dimensional grid. For automaton with dimensionality higher than 1 the shape of each cell can be different than squared. In $2D$ tessellation for example, each cell can be hexagonal or triangular, and each tessellation presents its own advantages and disadvantages. For instance the squared can be easily visualized on a screen as each cell is easily mapped onto a pixel, but may present problems of anisotropy for some kind of fluid simulations as in the case of the *HPP* model for fluid simulation [3]. An Hexagonal tessellation can solve the anisotropy problem [16] but presents obvious graphical challenges. Often, to avoid complications, periodic boundary conditions are used, so that for instance, a two-dimensional grid is the surface of a torus as shown in Figure 2.1.

2.1.2 *Neighborhood*

The evolution of a cell's state is function of the states of the neighborhood's cells. The geometry and the number of cells that are part of the neighborhood depends on the tessellation type, but it has to have three fundamental properties:

1. **Locality**: it should involve only a *limited* or *finite* finite number of cells.

2. **Invariance**: it should not be changed during the evolution.

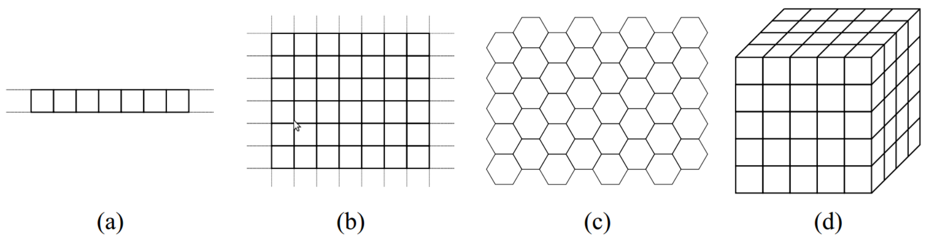


FIGURE 2.2: Examples of cellular spaces. (a) 1-D, (b) 2-D squared cells, (c) 2-D hexagonal cells, (d) 3-D cubic cells.

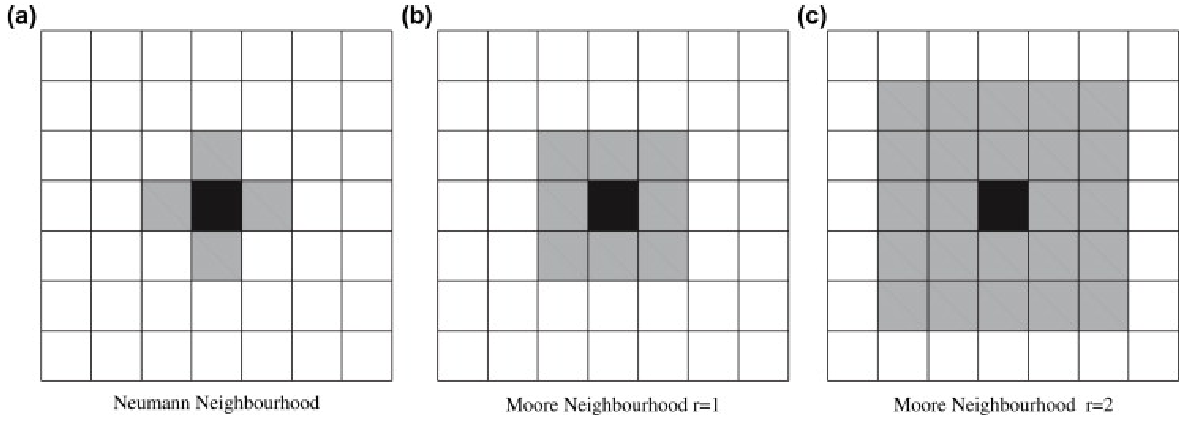


FIGURE 2.3: Examples of different kinds of neighborhood with different radius values.

3. **Homogeneity**: it has to be the same for each cell of the automaton.

Typically, the neighborhood of a cell *"surrounds"* the cell itself. For $1D$ cellular automata the neighborhood is identified by a number r called *radius* [17]. A $r = 2$ identifies $n = 2r + 1$ cells in a 1D lattice: the central cell plus the right and left cells. Typical $2D$ cellular space neighborhood are the those of *Moore* and *von Neumann*. The number of cells in the Moore neighborhood of range r is the odd squares $(2r + 1)^2$, the first few of which are 1, 9, 25, 49, 81, and so on as r is increased. von Neumann's one consists of the central cell plus the cell at north, south, east, and west of the central cell itself. Moore's ($r = 1$) one add the farther cells at north-east, south-east, south-west and north-west (see figure 2.3).

2.1.3 *Transition Function*

The evolution of the cell's state is specified in the so called *transition function* that is applied at the same time and on each cell. Usually the transition function is *deterministic* and usually defined by a look-up table only when the total number of state for each cell is small, otherwise the resulting table would have enormous size because the number of possible state transition is exponential in the number of states. Alternatively, a transition function is defined by an algorithmic procedure that may be probabilistic, in the case of *stochastic cellular automata* [18].

δ	a	b	c	d	e
q_0	q_0	q_0	q_2	q_1	q_1
q_1	q_1	q_3	q_1	q_1	q_1
q_2	q_3	q_2	q_2	q_0	q_1
q_3	q_0	q_1	q_1	q_0	q_1

Table 2.1: An example of tabular representation of DFM transition function.

2.2 FORMAL DEFINITION

Cellular automata are dynamic discrete in time, space and state models, defined by a lattice of cells each containing a finite state automaton.

2.2.1 *Finite State Automaton*

Also known as deterministic finite automata (DFAs) or as deterministic finite state machines, DFAs are among the simplest and better studied computational models. A DFA is a theoretical model of computation with limited capability. It can only recognize regular languages (the family of languages in the third category of the *Chomsky* classification and that can be obtained by regular expressions) and can only be in one of the finite number of states at a time, namely the *current state*. Its state can change in response of the input taken by a transition function, describe all possible state changes, and of the current state. It is a much more restrictive model in its computational capabilities than the one of the Turing machines and, for example, it is possible to prove that it is impossible for a DFA to determine whether its input consists of a prime number of symbols, but, they are still powerful enough to solve simpler problems, and hence to recognize simpler languages, as for example the following: $L = \{w \in \{0,1\}^*\}$, the language composed by strings with an even number of 0 and 1; they are only capable to recognize languages in the class 3 of the *Chomsky* classification [19]. In fact it can be proven that for each language L accepted by a DFA exists a grammar L_G s.t. $L = L_G$ i.e. L_G generates L, but they fail for example, in accepting *context-free* languages.

Formally, a DFA is defined as a 5-tuple:

$$M =< Q, \Sigma, \delta, q_0, F >$$

where:

- Q is a finite, nonempty, set of states.
- Σ is the alphabet
- $\delta : Q \times \Sigma \longmapsto Q$ is the transition function, also called next-state function, and can be represented in tabular form as in 2.1
- q_0 is the initial (or starting) state : $q_0 \in Q$
- F is the set, possibly empty, of final states: $F \subseteq Q$

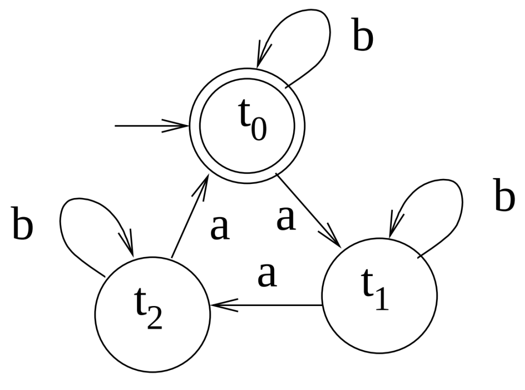


FIGURE 2.4: Graph representation of a DFA that accepts the following language: $\{b^*ab^*ab^*a\}$.

Note that we can also assume that F is composed by a single state i.e. $|F| = 1$ because a DFA can always transformed into another one that accepts exactly the same set of strings and has only one final state. The transformation consists in adding an additional state q_f, and for each of the previous final states $q_i \in F$ a new rule of the type $\delta(q_i, *) = q_f, * \in I$ is added to the transition function.

A run of DFA on a input string $u = a_0, a_1, \dots, a_n$ is a sequence of states $q_0, q_1, \dots, q_n$ s.t. $q_i \overset{a_i}{\longmapsto} q_{i+1}$, $0 \leq i < n$. For each pair of two states and a input the transition function deterministically returns the next DFA's state i.e. $q_i = \delta(q_{i-1}, a_i)$. For a given string $w \in \Sigma^*$, the DFA has a unique run (because of its deterministic nature), and is it said that it **accepts** w if the last state $q_n \in F$. A DFA recognizes the language $L(M)$ consisting of *all* strings it accepts.

Figure 2.4 shows an example of a DFA represented graphically as a graph where nodes are the states and the labeled edges are the possible state transitions from a state u to a state v. Note that, because the automaton is deterministic, it is not possible for two edges with the same label to point to two different nodes. In this example, $\Sigma = \{a, b\}$ is the alphabet, $Q = \{t_0, t_1, t_2\}$ is the set of states, $q_0 = t_0$ is the initial state and $F = \{t_0\}$, is the set of final or accepting states and the transition function is s.t. accepts the language generated by regular expression $E = \{b^*ab^*ab^*a\}$. When the automaton is executed on the input string $s = \{aaabba\}$ at the beginning of the execution, at time $t = 0$ the DFA is in the initial state t_0 and the first symbol of s, a is read. The transition function is applied once per each symbol of s i.e. $|s|$ times. The only rule that matches the current state t_0 and the current input a is $\delta = (t_0, a) = t_1$ hence the new state of the DFA becomes t_1. The DFA accepts the string only if the current state is in the set of final states F when it has consumed the input in its entirety. s is not accepted by the DFA described in the Figure2.4 because at the end of the computation the reached state is t_1 that is not a final state, as shown in the following execution trace:

$$t_0 \overset{\delta(t_0,a)}{\longmapsto} t_1 \overset{\delta(t_1,a)}{\longmapsto} t_2 \overset{\delta(t_2,a)}{\longmapsto} t_0 \overset{\delta(t_0,b)}{\longmapsto} t_0 \overset{\delta(t_0,b)}{\longmapsto} t_0 \overset{\delta(t_0,a)}{\longmapsto} t_1$$

On the input $S^1 = \{abababb\}$, instead, the DFA accepts as shows the following execution trace (see Equation 2.1):

$$t_0 \overset{\delta(t_0,a)}{\longmapsto} t_1 \overset{\delta(t_1,b)}{\longmapsto} t_1 \overset{\delta(t_1,a)}{\longmapsto} t_2 \overset{\delta(t_2,b)}{\longmapsto} t_2 \overset{\delta(t_2,a)}{\longmapsto} t_0 \overset{\delta(t_0,b)}{\longmapsto} t_0 \overset{\delta(t_0,b)}{\longmapsto} \mathbf{t_0} \tag{2.1}$$

2.3 HOMOGENEOUS CELLULAR AUTOMATA

Formally a CA A, is defined as quadruple $A =< Z^d, X, Q, \sigma >$ where:

- $\mathbb{Z}^d = \{i = (i_1, i_1, \ldots, i_d) \mid i_k \in \mathbb{Z}, \forall k = 1, 2, \ldots, d\}$ is the set of cells of the d-dimensional Euclidean space.
- X is the neighborhood, or neighborhood template; a set of m d-dimensional vectors (one for each neighbor)

  $$\xi_j = \{\xi_{j1}, \xi_{j2}, \ldots, \xi_{jd}\}\ , 1 \leq j \leq m$$

  that defines the set of the neighbors cells of a generic cell $i = (i_1, i_1, \ldots, i_d)$

  $$N(X, i) = \{i + \xi_0, i + \xi_2, \ldots, i + \xi_d\}$$

  where ξ_0 is the *null vector*. ξ_0 ensures that the cell i is always in its own neighborhood. Cell i is referred as to *central cell*.
- Q is the finite set of states of the elementary automaton EA.
- $\sigma = Q^m \to Q$ is the transition function of the EA. σ must specify $q_k \in Q$ as successor state of the central cell. If there are m cells in the neighborhood of the central cell including itself, then there are $|Q|^m$ possible neighborhood's state configurations. It means that there are $|Q|^{|Q|^m}$ possible transition functions. Plus we can see that the tabular definition of the next-state function is unsuitable for practical purpose. It should have $|\sigma| = |Q|^m$ entries, an exceedingly large number.
- $\tau = C \longrightarrow C \longmapsto \sigma(c(N(X, i)))$ where $C = \{c \colon Z^d \to Q\}$ is called the set of the possible configurations and $C(N(X, i)))$ is the set of states of the neighborhood of i.

As an example, consider a 2D cellular automaton, a generic cell $c = (10, 10)$ in its cellular space Z^2, 5 cell states $|Q| = 5$ with Moore's neighborhood X defined as follows:

$$X = \{\xi_0, \xi_1, \xi_2, \xi_3, \xi_4, \xi_5, \xi_6, \xi_7, \xi_8\} =$$
$$\{(0,0), (-1,0), (0,-1), (1,0), (0,1), (-1,-1), (1,-1), (1,1), (-1,1)\}$$

The set of cells belonging to the neighborhood of $c = (10, 10)$ is:

$$V(X, c) = \{(0,0) + c, (-1,0) + c, (0,-1) + c, (1,0) + c, (0,1) + c, (-1,-1) + c, (1,-1)$$
$$+c, (1,1) + c, (-1,1) + c = \{(10,10), (9,10), (10,9), (11,10), (10,11), (9,9), (11,9), (11,11), (9,11)\}$$

The total number of entries of a tabular definition of a transition function, taken from the set of all the $|Q|^{|Q|^{|X|}} = 5^{5^9} = 5^{1953125}$ possible transition functions, is $|Q|^{|X|} = 5^9 = 1953125$.

2.4 ELEMENTARY CELLULAR AUTOMATA

The simplest kind of CA are the so called *elementary* cellular automata, widely studied by Wolfram in [17]. They are defined as a 1-dimensional periodic array $\{C_i \mid 1 \leq i \leq N, C_i \in \{0, 1\}\}$ where N is the size of the automata. Each cell can only take one of two possible states, i.e. zero (0) or one (1). The transition function depends on the nearest

$F(1,1,1) = \{0,1\}$		$F(1,1,1) = 0$
$F(1,1,0) = \{0,1\}$		$F(1,1,0) = 1$
$F(1,0,1) = \{0,1\}$		$F(1,0,1) = 1$
$F(1,0,0) = \{0,1\}$	$\xrightarrow{instance}$	$F(1,0,0) = 0$
$F(0,1,1) = \{0,1\}$		$F(0,1,1) = 1$
$F(0,1,0) = \{0,1\}$		$F(0,1,0) = 1$
$F(0,0,1) = \{0,1\}$		$F(0,0,1) = 1$
$F(0,0,0) = \{0,1\}$		$F(0,0,0) = 0$

TABLE 2.2: Encoding of a transition function for a generic elementary CA to a 8-bit binary number. The transition function rules are firstly ordered according to the neighborhood pattern. Then each value of the right hand side of the i-th rule is interpreted as the i-bit of a binary number. The instance 110 of the Wolfram's elementary CA is shown in the right side of the table.

neighbors of, i.e. on cells within a radius $r = 1$ from, the central cell, thus involving a total of $2r + 1 = 2 \times 1 + 1 = 3$ cells (central, right and left ones). Since there are only $2 \times 2 \times 2 \times = 2^{2r+1} = 2^3 = 8$ possible state configurations for the aforementioned neighborhood, there can only be a total of $2^{2^3} = 2^8 = 256$ possible elementary automata, each of which may be uniquely mapped and to a 8-bit binary number [20], as shown in Table 2.2 and Section 2.4.1.

2.4.1 *Wolfram's code*

The generic transition function $F(C_{i-1}, C_i, C_{i+1})$ is defined by a look-up table of the form stated in Table 2.2 which also shows an example of an instance of a particular function the rule 110 which is the most important one. Cook proved universal computational power of the 110-th elementary CA, as conjuctured in 1985 by Wolfram himself, which is arguably the simplest Turing complete system [20].

Wolfram's code [17, 20] is easy to determine given that it is possible to sort neighborhoods patters in non-decreasing order (if interpreted as 3-bits) i.e. $(111 = 7), (110 = 6), (101 = 5)$ etc. using the following procedure:

1. Rules are sorted according to the neighborohood
2. For each configuration, the state which the given cell will take in the subsequent iteration, is specified
3. The next-iteration state of the i-th rule is interpreted as the i-th bit of a binary number, which is then converted in base 10.

2.4.2 *Wolfram's classification*

Despite their simple definition the mathematical analysis of elementary CA is not straightforward. A first attempt to classify CA was attempted by Wolfram [20]. He proposed a set of four classes for their classification that is still the most popular method of CA classification even if they suffer from a degree of subjectivity. Classification is based only on *visual valuations*, which are obviously subjective. A more rigorous definition of these classes is given in [22] where Karel Culik proves that deciding whether an automaton lies in a specific one out of four Wolfram's classes is an undecidable problem.

Wolfram's classes are defined as follows:

I these CA have the simplest behavior; almost all initial conditions result in the same uniform initial state (**homogeneous state**).

II different initial conditions yield different final patterns, but these different patterns consist of an arrangement of a certain set of structures, which stay the same forever or repeat themselves within a few steps (**periodic structures**).

III the observed behavior is more complicated and appears random, but patterns are still present (often in the form of *triangles*)(**chaotic pattern**).

IV in some respects these are the most complicated class; these behave in a manner somewhere in between class II and III, exhibiting sections of both predictable patterns and randomness (**complex structures**).

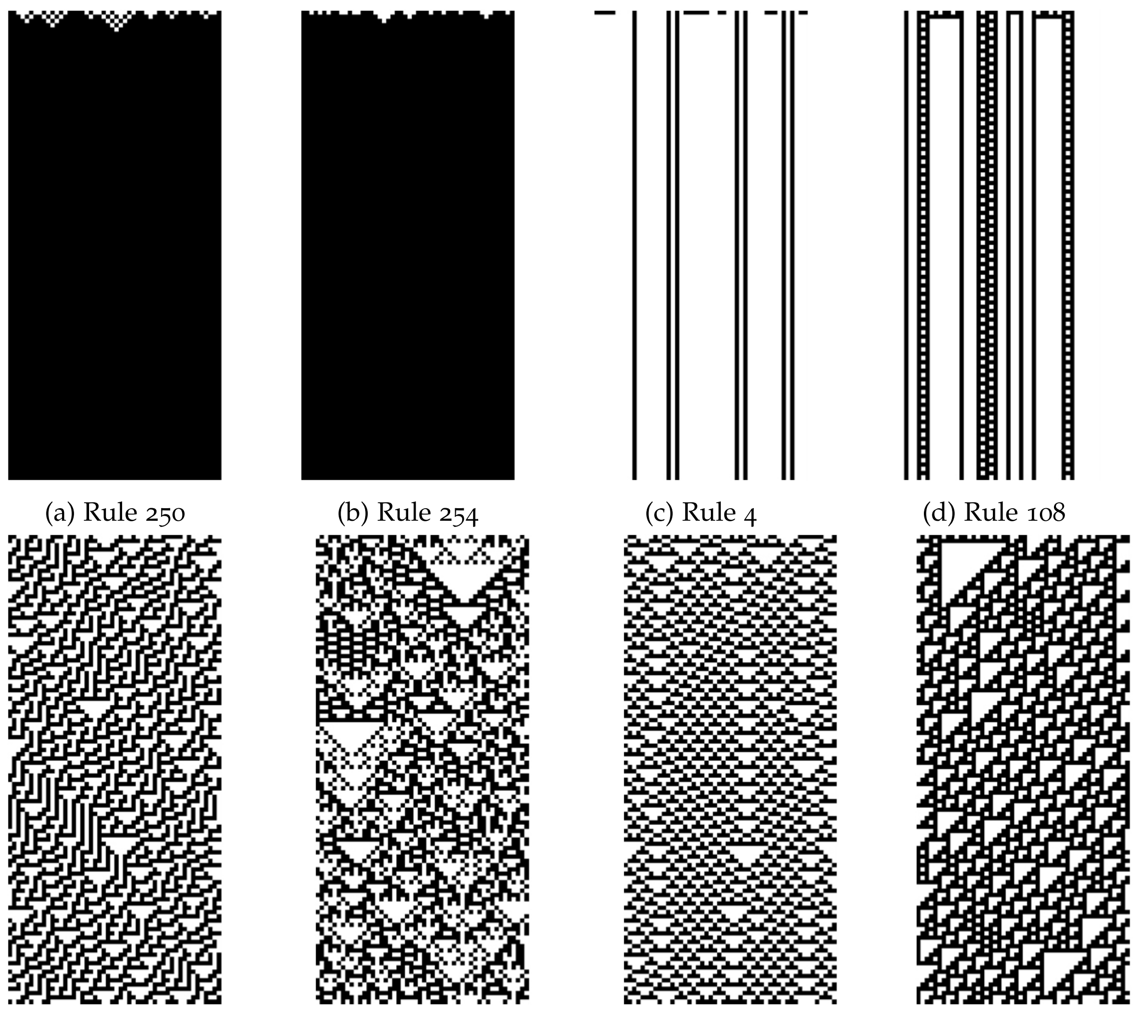
Wolfram observed that the behavior of a meaningful class of Cellular Automata by performing computer simulations of the evolution of the automata starting from random configurations. He suggested that the different behaviors of automata in his classes seems to be related to the presence of different types of attractors. In Figure 2.5 some elementary automata are divided in their respective classes. It is clear from these examples that automata from class 1 end up very quickly having the same value i.e. in a *homogeneous state* while automata from class 2 in a simple final periodic patterns. Class 3 appear to be completly chaotic and non-periodic while automata from class 4 have a mixed behaviour where complex-chaotic structures are locally propagated.

2.4.3 *At the edge of Chaos*

Class 4 automata are at *the edge of chaos* and give a good metaphor for the idea that the *interesting* complexity like the one exhibit by biological entities and their interactions or analogous to the phase transition between solid and fluid state of the matter, is in equilibrium between stability and chaos [23].

> *Perhaps the most exciting implication (of CA representation of biological phenomena) is the possibility that life had its origin in the vicinity of a phase transition and that evolution reflects the process by which life has gained local control over a successively greater number of environmental parameters affecting its ability to maintain itself at a critical balance point between order and chaos.*
> *(**Chris Langton** - Computation at the edge of chaos. Phase transition and emergent computation - pag.13).*

Langton in his famous paper, *Computation at the edge of chaos: phase transition and emergent computation* [23], was able to identify, simply parameterizing the rule space, the various CA classes, the relation between them and to "couple" them with the classical complexity classes. He introduced the parameter λ [24] that, informally, is simply the fraction of the



(a) Rule 250 (b) Rule 254 (c) Rule 4 (d) Rule 108

(e) Rule 30 (f) Rule 90 (g) Rule 54 (h) Rule 110

FIGURE 2.5: Examples of Wolfram's class 1 (a,b), 2 (c,d), 3 (e,f) and 4 (g,h) elementary cellular automata

entries in the transition rule table that are mapped to the not-quiescent state. The definition of the λ parameter is as follows:

$$\lambda = \frac{K^N - n_q}{K^N}$$

where:

- K is the number of the cell states
- N the arity of the neighborhood
- n_q the number of rules mapped to the quiescent state q_q

Langton's major finding was that a simple measure such as it correlates with the system behavior: as it goes from 0 to $1 - \frac{1}{K}$, the most homogeneous and the most heterogeneous rules table scenario, respectively, the average behavior of the system goes from freezing to periodic patterns to chaos and functions with an average value of λ are being *on the edge* [23](see Figure 2.7). Langton studied a entire family of totalistic CA with $k = 4$ and $N = 5$ and having λ varying in $[0, 0.75]$. He was able to determine that values of $\lambda \approx 0.45$

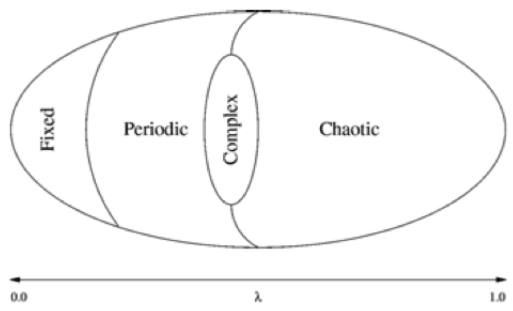


Figure 2.7: Relation between lambda parameter and the CA behaviors-Wolfram's classes.

raise up to class 4 cellular automata. Computational system must to provide fundamental properties if it is to support computation. Only CA *on the edge* show these properties on manipulating and store information data. The properties that a computational system must provide are:

STORAGE
: the ability of the system of preserving information for arbitrarily long times

TRANSMISSION
: the propagation of the information in the form of signals over arbitrarily long distance

MODIFICATION
: the modification of one or more signals.

Storage is coupled with less entropy of the system, but transmission and modification are not. A little entropy is associated with CA of class 1 and 2 while higher entropy with class 3. Class 4 is something in between, the cells cooperate and are correlated to each other, but not too much otherwise they would be overly dependent with one mimicking the other. Moreover they are very dependent to the initial configuration opening to the possibility to encode programs in it.

2.5 GAME OF LIFE

CA are suitable for representing many physical, biological, social and other natural phenomena. But they have proved to be a good tool to study under which condition a physical system expose the basic operations to support computation in all its aspects and requirements. Game of life is a famous 2D cellular automaton of the '70s well studied for its universal computation capacity, which has been proved indeed.

2.5.1 *Definition*

The Game of Life (GOL) is totalistic CA. A totalistic cellular automaton is a one in which the rules depend only on the total, or equivalently, the average, of the values of cells in the neighborhood. GOL can be thought as an infinite two-dimensional orthogonal and regular grid of square cells, each taking one of two possible states, *dead* or *alive*. Every cell interacts with the nine adjacent neighbors belonging to the Moore neighborhood. At each time step, one of the following transitions occur:

- *Birth*: if the cell is in the state ***dead*** and the number of alive neighbors is 3, then the cell state becomes alive (1) .
- *Survival*: if the cell is in the state ***alive*** and the number of alive neighbors is ***2 or 3***, then the cell state is still alive (1) .
- *Death*: If the cell is in the state ***alive*** and the number of alive neighbors is ***less than 2 or higher than 3***, then the cell state becomes dead (0).

The initial configuration of the system specifies the state (dead or alive) of each cell in the cellular space. The evolution of the system is thus obtained by applying the aforementioned transition function rules **simultaneously** to every cell in the cellular space, so that each new configuration depends on the one at the current step. The rules continue to be applied repeatedly to create further generations.

Formally the Game of Life automaton is defined as follows:

$$Life =< R, X, Q, \sigma >$$

where:

- R is the set of cells, forming a two-dimensional toroidal cellular space. A generic cell in R is individuated by means of a pair of integer coordinates (i, j) s.t. $0 \leq i < i_{max}$ and $0 \leq j < j_{max}$ where i_{max}, j_{max} are the sizes of R in the horizontal and vertical directions, respectively. The first coordinate, i, represents the row, while the second, j, the column. The cell at coordinates $(0, 0)$ is located at the top-left corner of the computational grid (cf. Figure 2.2).
- $X = \{(0,0), (-1,0), (0,-1), (0,1), (1,0), (-1,-1), (1,-1), (1,1), (-1,1)\}$ is the Moore neighborhood pattern. The coordinates of neighbors of (i, j) are given by:

  $$\begin{gathered} N(X,(i,j)) = \\ = \{(i,j) + (0,0), (i,j) + (-1,0), \dots, (i,j) + (-1,1)\} = \\ = \{(i,j), (i-1,j), \dots, (i-1,j+1)\} \end{gathered}$$

  A subscript can be used to index cells belonging to the neighborhood. Let $|X|$ be the number of elements in X, and $n \in \mathbb{N}$, $0 \leq n < |X|$; the notation

  $$N(X,(i,j),n)$$

  represents the coordinates of the n-th neighbor of (i, j). Thereby, $N(X,(i,j),0) = (i,j)$, i.e. the central cell, $N(X,(i,j),1) = (i-1,j)$, i.e. the first neighbor, and so on (cf. Figure 2.3b).
- $Q = \{0, 1\}$ is the set of cell states, 0 representing the dead state, 1 the alive one.
- $\sigma : Q^9 \rightarrow Q$ is the deterministic cell transition function. It is composed by a single elementary process, which implements the aforementioned evolution rules.

The Game of Life belongs to the class 4 of the Wolfram's taxonomy, because (quoting the words of Wolfram) "rich and complex structures, stable blocks and moving patterns come into existence even starting from a completely random configuration". Among many patterns and blocks appearing in the GOL, one of the most common is the so called *glider* (see Figure 2.9) that is a periodic pattern with a period of 5 steps that is capable of moving into the cellular space and thus of transmitting information.

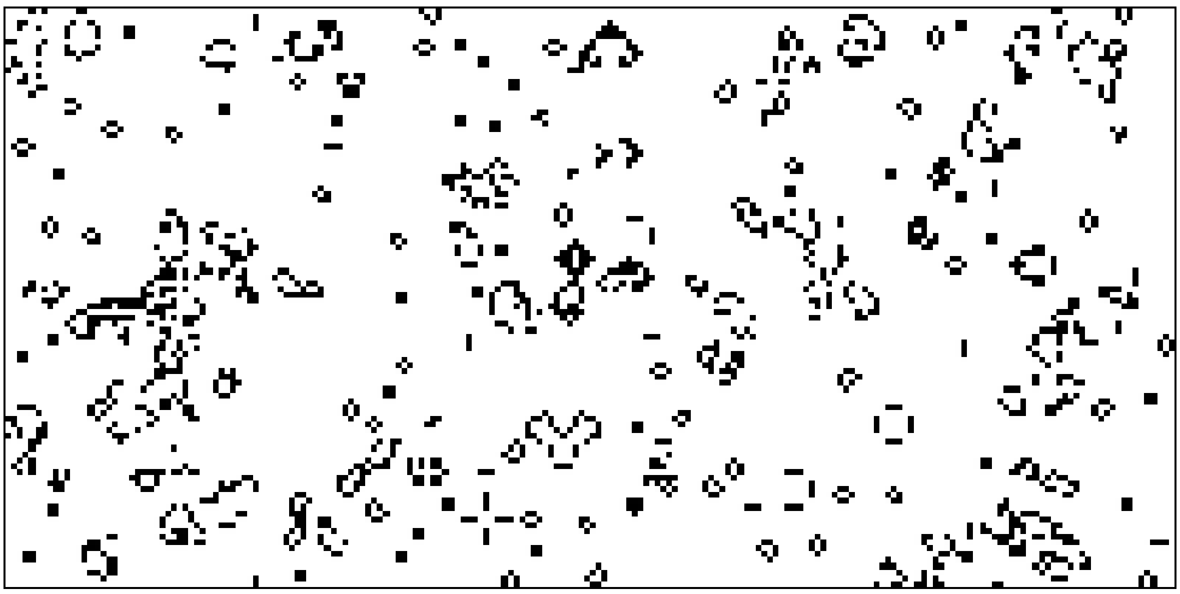


FIGURE 2.8: An execution of the Game of Life. Many common patterns are visible as blikers, gliders and beacons.

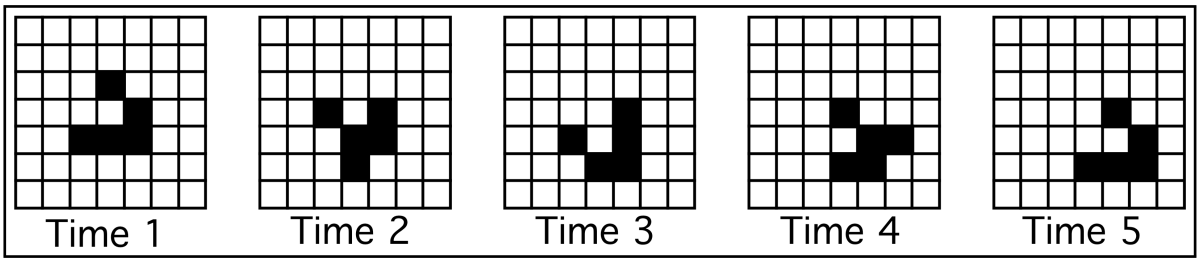


FIGURE 2.9: Glider in Conway's *Game of Life*.

2.5.1.1 *Game of life as Turing machine*

Every CA can be considered as a device capable of supporting computation where the initial configuration encodes an input string (the source code of a program). At some point in time, the current configuration can be interpreted as the result of the computation and decoded into an output string. As mentioned in Section 2.2.1, not all the computational devices have the same (computational) power. So which is the one of the *Game of Life*? GOL is proved to be capable of universal computation. Therefore the Game of life is computationally equivalent to a Turing machine [25]. This result raises a interesting issue; since the *Halting Theorem* is undecidable i.e. no algorithm can ever decide whether a Turing Machine will accept a certain input or not, the evolution of the GOL is **unpredictable** (as all the universal computational systems) practically meaning that is not possible to use any algorithmic shortcuts to anticipate the resulting configuration given an initial input. The most efficient way to know the outcome of an execution of GOL, is to let the system run.

Life, like all computationally universal systems, defines the most efficient simulation of its own behavior [26]

2.6 extended cellular automata model (xca)

The Extended Cellular Model (XCA) was introduced by *G.M. Crisci et al.* in order to overcome the limitation of the classical CA method (monolithic cell state and transition function as a look-up table) regarding the simulation of macroscopic natural systems and processes as for instance debris and lava flow, [27, 28], forest fire [29], soil erosion/degradation by rainfall [30] and water flux in unsaturated soils [31]. The main difference of XCA compared to CA are:

- the state of the cell Q is subdivided in r smaller components, the substates, each representing a particular feature of the phenomenon to be modeled (e.g. for lava flow models, cell temperature and lava quantity) and relevant to the evolution of the system. The overall global state of the cell is then defined as the Cartesian product of all substates $Q = Q_1 \times Q_2 \times \ldots \times Q_r$.
- the transition function $\tau = \{\tau_1, \ldots, \tau_s\}$ is also decomposed into *elementary processes*, in turn further splitted into *local interaction* which accounts for the interactions with neighboring cells and *internal transformation* which model the changes in cell state that are not consequence of any external interaction with neighboring cells.
- A set of parameters, $P = \{p_1, p_2, \ldots, p_p\}$ is furthemore considere, which allow to *tune* the the CA for reproducing different dynamical behaviors of the phenomenon of interest (e.g. lava solidification threshold and density, or the *Stephen-Boltzmann* constant).
- a subset $E = E_1 \cup E_2 \cup \ldots \cup E_l \subseteq R$ of the cellular space R that is subject to *external influences* (e.g. lava craters) is specified by a supplementary function $\gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_t\}$ External influences model those kind of features that are not easily described in term of internal transformation or local interactions.

2.7 probabilistic ca

If some of the assumptions of the ordinary CA characterization are relaxed interesting results are obtained. As an example, the following are extensions that have proved to be useful in many cases: *asynchronous update*, *non-homogenous cellular space and neighborhood* or transition function that depends on the coordinates of a cell. An interesting class of CA is obtained when the transition function is based on some stocastic process. Probabilistic CA [32] have a single key difference w.t.r. to ordinary CA i.e. the transition function σ is a stochastic-function that decides the next-state according to some probability distributions. They are used in a wide class of problems like in modelling ferromagnetism, statistical mechanics [33] or the cellular Potts model[2] [34]

Probabilistic CA (PCA) evolution can be studied if interpreted as of Markov processes. A Markov process, is a stochastic process that exhibits memorylessness, also called Markov property, which means that the future state of a system is conditionally independent from the past. Two events A and B are independent if $P(AB) = P(A)P(B)$ or in other words, that the probability of the event A to occurs does not depends on the event B i.e. $P(A|B) = P(A)$. The property of this kind of processes allows future probabilities of an event to

2 Lattice-based model adopted for the simulation of the collective behavior of cellular structures.

be determined from the probabilities of events at the *current time* only. In PCA analysis, homogeneous Markov chains are adopted because each cell has a discrete set of possible values for the status variable. In terms of such type of chain a CA is a process that starts in one of these states and moves successively from one state to another. If the chain is currently in state s_i, than it evolve to state s_j at the next step with probability p_{ij}.The changes of state of the system are called transitions, and the probabilities associated with various state changes are named transition probabilities and are usually represented in the *Markov chain transition matrix* of the form shown below:

$$M = \begin{pmatrix} p_{11} & p_{12} & p_{13} & \cdots \\ p_{21} & p_{12} & p_{23} & \cdots \\ p_{31} & p_{32} & p_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Markov chain transition matrix are useful in analyzing very small PCA but they prove to be impractical for larger sizes as for instance, a CA on a small celular space of size 10×10 has $2^{10\times10}$ possible states and the resulting chain transition matrix has size of $2^{10\times10} \times 2^{10\times10}$ that is a huge number!

3 THE FINITE DIFFERENCE METHOD

All exact science is dominated by the idea of approximation. When a man tells you that he knows the exact truth about anything, you are safe in infering that he is an inexact man.
— Bertrand Russel

Truth ... is much too complicated to allow anything but approximations.
— John von Neumann

Most of the Partial Differential Equations (PDE) arising from the mathematical formulation of physical systems are often very hard if not impossible to solve analytically, thus requiring approximate numerical solutions. An important part of handling and solving PDEs is to be able to use local, accurate and stable algebraic expressions as an approximation of the derivatives appearing in the equations while retaining, at the same time, most of the global and continuous information of the original formulation. During the last half century several approximation methods have been developed and studied, such as the finite volume (FV), finite element (FE), and finite difference (FD) methods (FDM), each with its specific approach to discretization and strength.

This chapter briefly describes the concept of differential equations and introduces their numerical solution using the finite difference method. For a rigorous and complete description of the topic of this chapter please refer to [35–37].

3.1 DIFFERENTIAL EQUATION

A differential equation [38, 39] is an equation where the unknown is a function itself and where derivatives ot the unknown appears in the equation. Differential Equations can be divided into two main classes:

ORDINARY DIFFERENTIAL EQUATION (ODE):
: where the unknown function contains only derivatives with the respect of a single variable. As example of ODE is the following equation,

$$\frac{dT}{dx} = \alpha T(x) + b$$

where a and b are real constants.

PARTIAL DIFFERENTIAL EQUATIONS (PDE):
: a class that is extremely large and rich in functions, each of them with different behaviors and properties. Examples of classes of this kind of differential equations are *parabolic*, *elliptic* and *hyperbolic* equations. PDEs search for a multidimensional function of several variables, and this means that partial derivatives may now appear in the equation. The following equations are among the most famous and popular PDEs:

1D TRANSPORT EQUATION:

$$\frac{\partial T}{\partial t} + \frac{\partial T}{\partial x} = 0 \tag{3.1}$$

1D DIFFUSION EQUATION:

$$\frac{\partial T}{\partial t} - \frac{\partial^2 T}{\partial x^2} = 0 \tag{3.2}$$

1D WAVE EQUATION:

$$\frac{1}{c^2}\frac{\partial^2 T}{\partial t^2} - \frac{\partial^2 T}{\partial x^2} = 0 \tag{3.3}$$

where c is the speed of propagation. It can be applied to solve

- *transverse string vibration* problem with T representing the transverse displacement of the string, $c = \sqrt{\frac{K}{\rho A}}$ where K, ρ and A are the tension, density of the material and the cross area of the string.
- *acoustic* with T representing the pressure or the velocity of the considered fluid where c is the speed of sound in the medium.
- *mambrane vibration* with T representing the transverse displacement of a membrane e.g. a drum head, $c = \sqrt{\frac{K}{m}}$ where K, m are the tension and the mass per unit of area.

LAPLACE'S EQUATION:

$$\nabla^2 T = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2}{\partial z^2} = 0 \tag{3.4}$$

where ∇^2 is refeered to as the Laplacian operator and given by

$$\nabla^2 = \nabla \cdot \nabla = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$$

It often arises in fluid flow (where T is the velocity potential), gravitational and bar torsion problems.

HEAT EQUATION:

$$\rho c \frac{\partial T}{\partial t} - \kappa \nabla^2 T + Q = 0 \tag{3.5}$$

where k is the thermal conductivity (following from Fourier's law of heat conduction $q_x = -k\frac{dT}{dx}$ measured in $\frac{J}{s\,m^2}$, Q is the internal heat, c, rho are the material specific heat conduction parameter and density.

Note that there is a missing piece that would allow all these equations to be solved unequivocally, namely the initial condition and/or boundary. For example, regarding the 1D wave equation (Equation 3.3), it is not clear what the reflection coefficient at the ends of the string is, or with regards to the heat equation (Equation 3.5) what the initial temperature at time $t = 0$ is. *Initial* conditions must be provided whenever the differential equation is time dependent and, *boundary* conditions must be specified whenever spacial dependency occurs. The latter, in particular specifies the behavior of the equation at the boundary of

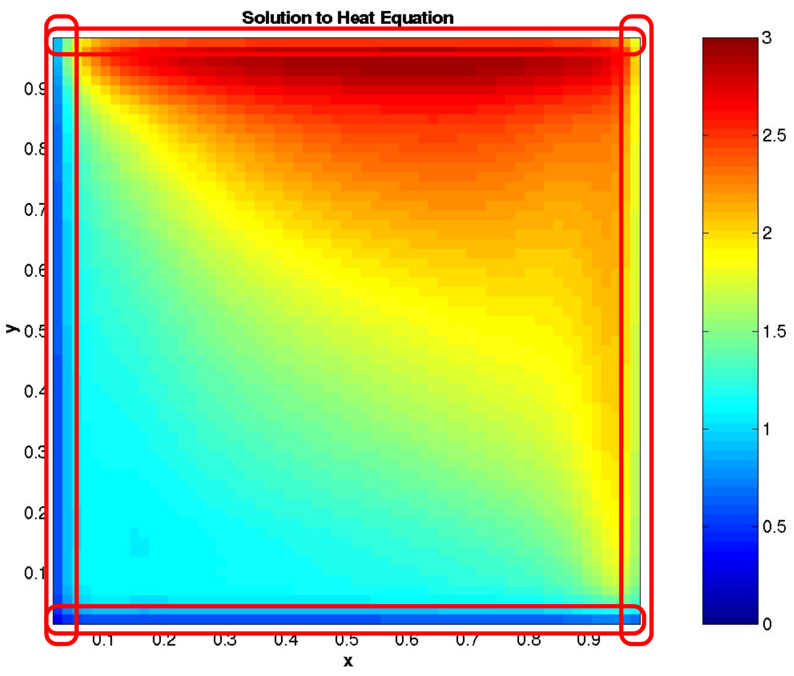


FIGURE 3.1: Heat Equation (see Equation 3.5) Dirichlet Boundary Conditions. The perimeter of the square domain (highlighted in red) has fixed temperature, i.e. the solution T is known at the boundary. In this particular case, $T(x, 0) = T(0, y) = 0$, $T(1, x) = 3$ and $T(1, y) = 2$ where $0 \leq x, y \leq 1$.

the domain $\partial\Gamma$ (which has to be compact). The most commonly used boundary conditions defining a PDE are of two kinds:

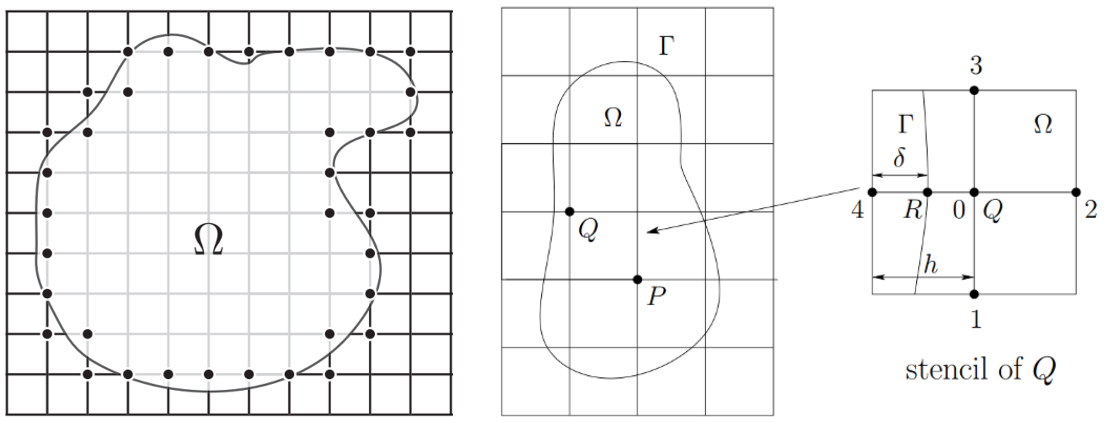
DIRICHLET:
in which the values of the functions at $\partial\Gamma$ are hard-coded, i.e. $T(\partial\Gamma)$ is known.

NEUMANN:
specifies values of the derivatives at $\partial\Gamma$.

Other kinds of boundary conditions are possible, such as, for instance, the Robin's boundary conditions [40] which is a mix of Dirichlet and Neumann ones. See Figure 3.1 for an example of Dirichlet boundary conditions for the equation 3.5.

3.2 FINITE DIFFERENCE METHOD

The finite difference approximation for derivatives is one of the simplest oldest methods adopted in solving differential equations numerically. It is used since 1768, when *L. Euler* discovered it while he was trying to solve one dimensional problems and subsequently extended by *C. Runge* to two dimensions in 1910. Since the advent of computers in 1950, FDM popularity skyrocketed also thanks to the theoretical results that have been obtained regarding stability, convergence and other of its properties.



(a) Discretization of a curvilinear domain.

(b) Interpolation of boundary values on a non rectangular geometry domain.

FIGURE 3.2: Approximation of curvilinear geometries on a square uniform computational grid.

The general idea behind FDM is that the differential operator is approximated by replacing the derivatives using difference quotients. The differential operator is approximated constituting the field equation locally, among a number of finite function values. Therefore, the space and time domain are *partitioned* in a grid-like fashion in order to store the local field quantities, and approximated solutions are computed only for those discrete grid points. The numerical solution is known only at a finite number of points in the physical domain. A difference quotient is a linear combination of function values at neighboring grid points. The number of different points appearing in the quotient directly dictates the order of the differential operator. We can always assume that the domain is discretized via a rectangular grid since we can always specify boundary conditions for the grid points such that they mimic the real shape of the boundary at hand as depicted in Figure 3.2a. Complex geometries and curvilinear boundaries can be treated by computing the value of such points that lie on the boundary as a linear interpolation of neighboring boundary **grid points** as shown in Figures 3.2 and 3.2b and, Equation 3.6,

$$T(R) = \frac{T_4(h-\delta) + T_0\delta}{h} = g_0(R) \tag{3.6}$$

where g_0 specifies the values at the boundaries.

Figure 3.3 is a schematic representation of how FDM is used to obtain a numerical solution. First thing, the continuous differential operators and the domain are discretized and then the approximation is computed by solving the difference formulas on grid points using. The error between the numerical and the exact solution is determined by the employed difference formula and it is commonly refereed to as *truncation*[1] or *discretization*

1 The term truncation comes from the fact that a finite difference quotient is a truncation of the Taylor expansion.

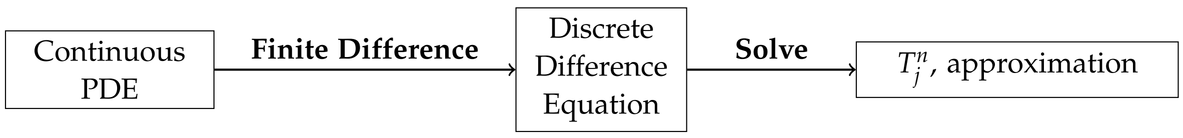


FIGURE 3.3: Relationship between continuous and discrete problems.

error. Increasing the resolution of the grid, in turn, increases the accuracy of the numerical solution since the error associated with the finite difference formulas directly depends on the distance between grid points.

Depending on how the derivatives are approximated, explicit or implicit FDM schemes are obtained. When forward difference formulas are considered, the resulting difference equation is generally expressed in terms of an explicit recurrence formula, while backward difference formulas generally lead to implicit recurrence formulas involving unknown values, and therefore require the solution of a linear system of equations to obtain the new state of the physical system (i.e. values of the unknown variables in the PDEs) at each grid point.

3.2.1 *Finite Difference Formulas*

The differential operators appearing in a PDE problem can be approximated at a given point by a difference formula which is a linear function of its neighboring grid points. Finite difference formula can be defined and derived in a number of ways but some of them are more widely and commonly used then others. The rest of this Section shows how these common formulas are derived along with their basic properties.

For the sake of simplicity the formulas are refereed to a one-dimensional space and time domain since the generalization to several dimensions is obvious. Both space and time domains are partitioned into a finite discrete mesh as follows:

$$t_n = n\Delta t,\; n = 0, 1, \ldots, L,\; \Delta t = \frac{1}{L} \tag{3.7}$$

$$x_j = j\Delta x,\; j = 0, 1, \ldots, M,\; \Delta x = \frac{1}{M} \tag{3.8}$$

For the rest of the chapter, it can be assumed that grid points are identified by two indices, j, n, and T_j^n is the value the function at time n at grid point j (see Figure 3.5).

3.2.1.1 *Forward Scheme*

The *forward* scheme is probably the most common FD formula and can be derived from Taylor's expansion of T_{j+1}^n in terms of T_j^n and its derivatives as:

$$T_{j+1}^n = T_j^n + \left.\frac{\partial T}{\partial x}\right|_j^n \Delta x + \frac{1}{2!}\left.\frac{\partial^2 T}{\partial x^2}\right|_j \Delta x^2 + \ldots + \frac{1}{k!}\left.\frac{\partial^k T}{\partial x^k}\right|_j \Delta x^k + \ldots \tag{3.9}$$

If series is truncated after the second term ($k = 1$) and solving for $\frac{\partial T}{\partial x}$ the following is obtained:

$$\frac{\partial T(t, x)}{\partial x} = \frac{T_{j+1}^n - T_j^n}{\Delta x} + \mathcal{O}(\Delta x) \tag{3.10}$$

Equation 3.10 is called **first order forward** finite difference approximation. Other approximations are possible and are easily obtainable by expanding different and/or more points of the grids in the Taylor's expression.

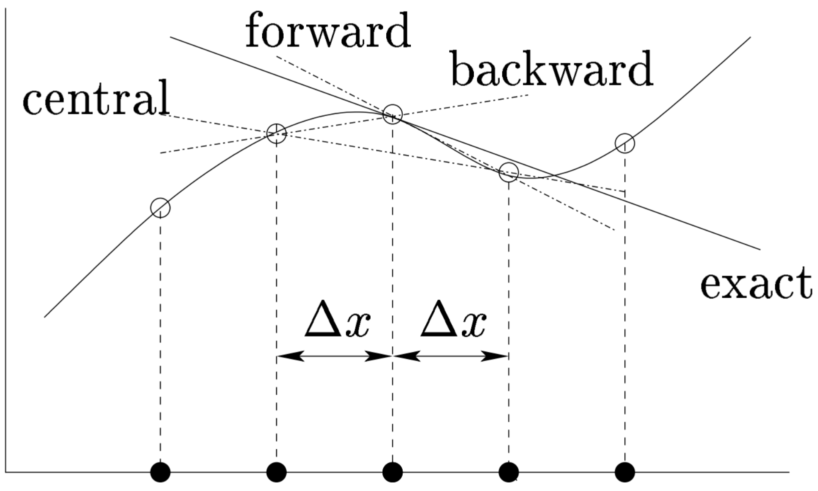


FIGURE 3.4: Explicit FDM discretization for the 1D heat conduction problem

3.2.1.2 *Backward Scheme*

The **Backward** finite difference quotient can be obtained from the Taylor's expansions of T^n_{j-1} in terms of T^n_j and its derivatives:

$$T^n_{j-1} = T^n_j - \left.\frac{\partial T}{\partial x}\right|_j \Delta x + \frac{1}{2!}\left.\frac{\partial^2 T}{\partial x^2}\right|_j \Delta x^2 + \ldots + \frac{1}{k!}\left.\frac{\partial^k T}{\partial x^k}\right|_j \Delta x^k + \ldots \tag{3.11}$$

which can be rearranged in the following manner

$$\left.\frac{\partial T}{\partial x}\right|_j = \frac{T^n_j - T^n_{j-1}}{\Delta x} + \mathcal{O}(\Delta x) \tag{3.12}$$

The same approach can be used to derive approximation for higher order derivatives. For example equation 3.13, known as **central difference formula**, is an approximation for the second order derivative and can be obtained retaining the first four terms in both equations 3.9 and 3.11 and adding the resulting expression:

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_j = \frac{T^n_{j+1} - 2T^n_j + T^n_{j-1}}{\Delta x^2} + \mathcal{O}(\Delta x^2) \tag{3.13}$$

Figure 3.4 shows how finite difference formulas can be interpreted geometrically. Note that this approach in deriving FD formulas can be generalized in order to obtain FD approximation for derivatives of any order.

3.2.1.3 *Mixed Derivatives*

Mixed derivatives can also be approximated using FDM, e.g. for two dimensions by means of the following property of mixed derivatives:

$$\frac{\partial^2 T}{\partial x \partial y} = \frac{\partial}{\partial x}\left(\frac{\partial T}{\partial y}\right) = \frac{\partial}{\partial y}\left(\frac{\partial T}{\partial x}\right) \tag{3.14}$$

and considering the following approximations:

$$\begin{cases} \frac{\partial^2 T}{\partial x \partial y} = \frac{\left(\frac{\partial T}{\partial y}\right)_{i+1,j} - \left(\frac{\partial T}{\partial y}\right)_{i-1,j}}{2\Delta x} + \mathcal{O}(\Delta x)^2 \\ \left(\frac{\partial T}{\partial y}\right)_{i+1,j} = \frac{T_{i+1,j+1} - T_{i+1,j-1}}{2\Delta y} + \mathcal{O}(\Delta y)^2 \\ \left(\frac{\partial T}{\partial y}\right)_{i-1,j} = \frac{T_{i-1,j+1} - T_{i-1,j-1}}{2\Delta y} + \mathcal{O}(\Delta y)^2 \end{cases} \tag{3.15}$$

A second order 2 variables finite difference approximation for the mixed derivative is the following:

$$\left(\frac{\partial^2 T}{\partial x \partial y}\right)_{i,j} = \frac{T_{i+1,j+1} - T_{i+1,j-1} - T_{i-1,j+1} - T_{i-1,j-1}}{4\Delta xy} + \mathcal{O}((\Delta x)^2, (\Delta y)^2) \tag{3.16}$$

Extending the former method to higher dimensional mixed derivatives is straightforward.

3.3 HEAT EQUATION

As an example, a simple FDM scheme for the heat conduction initial-boundary value problem shown in Equation 3.17 is derived.

$$\frac{\partial T(t,x)}{\partial t} = \kappa \frac{\partial^2 T(t,x)}{\partial x^2} \tag{3.17}$$

where $0 \leq t \leq L$ and $0 \leq x \leq M$.

In order to construct a FD approximation for equation 3.17 it is necessary to:

1. discretize the domain into a finite uniform and regular mesh where each point x_j is identified with a unique index j.
2. first and second order derivative appearing in Equation 3.17 are substituted by forward and central difference formulas, respectively, leading to Equation 3.18

$$\frac{T_j^{n+1} - T_j^n}{\Delta t} = \kappa \frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{\Delta x^2} \tag{3.18}$$

3. Equation 3.17 is evaluated at grid point $(n\Delta t, j\Delta x)$

Figure 3.5 depicts the set of points that play a role in the difference formula 3.18, commonly refeered to as the *stencil*. Each point of the stencil is used at time m in order to compute grid values at time $m+1$ as shown in Figure 3.6. Solution to equation 3.17 using the discretization 3.18 is called *forward time, centered space or FTCS* approximation and requires the specification of initial conditions at $t = 0$ and boundary condition at $x = 0$ and $x = M$ (see Figure 3.1).

It can be shown that in order to the solution to be stable Δt must not be too large and in particular the following condition must hold to ensure a stable solution [41–43]:

$$r = k\frac{\Delta t}{\Delta x^2} < \frac{1}{2}$$

This scheme is also called *explicit* because values at the subsequent time step are explicitly computable from the values at the current time as it is shown in the equation 3.19.

$$T_j^{n+1} = T_j^n + \frac{k\Delta t}{\Delta x^2}(T_{j+1}^n + T_{j-1}^n - 2T_j^n) \tag{3.19}$$

When the backward difference formula is used to approximate the time derivative the following approximation is obtained:

$$\frac{T_j^n - T^{n-1}}{\Delta x} = k\frac{T_{j+1}^n - 2T_j^n + T_{j-1}^n}{\Delta x^2} \tag{3.20}$$

This stepping scheme is called *implicit* because values at time n are given implicitly as can be seen if equation 3.20 is rearranged to obtain the following:

$$T_j^n - \frac{k\Delta t}{\Delta x^2}(T_{j+1}^n + T_{j-1}^n - 2T_j^n) = T_j^{n-1} \tag{3.21}$$

In order to obtain values of T at time n a system of non trivial algebraic equations has to be solved. It can be rewritten in matrix form yielding to a linear tridiagonal system as shown in Figure 3.7. where $\lambda = \frac{k\Delta t}{\Delta x^2}$ for which an efficient algorithm exists, the *Thomas's algorithm* [44, 45], which solves it in $\Theta(n)$ where n is the number of unknowns.

3.4 SOLVING FINITE DIFFERENCE PROBLEMS FDM WITH EXTENDED CELLULAR AUTOMATA

XCA can be employed to formally represent FDM models for both explicit and implicit schemes. In fact, with reference to the definitions given in Chapter 2, in case of an explicit scheme, the computational domain can be represented by means of the R cellular space and the coordinates of the grid points involved in the recurrence formula defined by means of the X neighborhood relationship. Moreover, the values of the involved variables can be represented in terms of substates and the explicit recurrence formula easily expressed in terms of elementary processes. On the other hand, when dealing with a linear system resulting from an implicit FDM scheme, a steering function can be employed in association with an external linear algebra solver e.g. BLAS [46] or other dedicated libraries [47].

It is worth to recall that physically-based models laying on a XCA direct discrete approach (i.e., not going through the discretization of differential equations) can lead to the same discrete formulations achieved with the FDM, making these latter formulations a specific case of the general XCA approach. As an example, the work of Mendicino *et al.* proved that their direct discrete formulation applied to the *Darcy's* equation for modeling

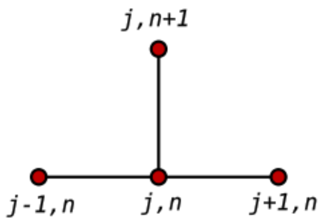


FIGURE 3.5: Explicit FDM stencil for the considered discretization of the 1D heat conduction problem.

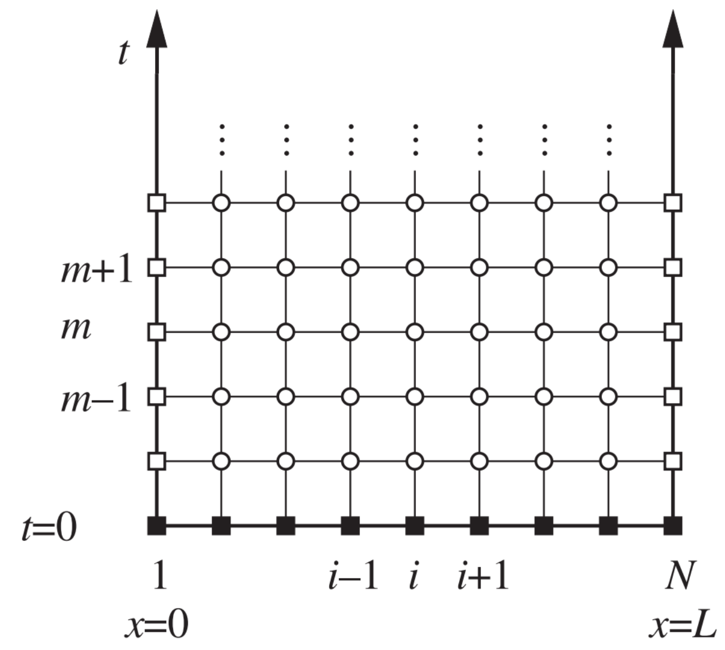


FIGURE 3.6: 1D-heat equation FDM space and time partitioning. The solution for grid point $(i, m+1)$ is explicitely computed by considering points $(i, m), (i-1, m), (i+1, m)$

$$\begin{bmatrix} (1+2\lambda) & -\lambda & & \\ -\lambda & (1+2\lambda) & -\lambda & \\ & \ddots & \ddots & \ddots \\ & -\lambda & (1+2\lambda) & -\lambda \\ & & -\lambda & (1+2\lambda) \end{bmatrix} \begin{bmatrix} T_1^n \\ T_2^n \\ \vdots \\ T_M^n \end{bmatrix} = \begin{bmatrix} T_1^{n-1} \\ T_2^{n-1} \\ \vdots \\ T_M^{n-1} \end{bmatrix}$$

FIGURE 3.7: Tridiagonal Matrix.

unsaturated flow in a three-dimensional cubic cell system is similar to the one achieved using an explicit FDM or a finite volume scheme [31]. However, the same discrete governing equation system would allow a greater level of convergence with respect to traditional methods if an irregular mesh were used and a not linear (e.g., quadratic) interpolation of the hydraulic head on the cells were adopted (e.g., Tonti proved it for the Finite Element Method [48]).

4 PARALLEL COMPUTING

Divide ut regnes.

— Julius Caesar

This chapter briefly introduces some of the main concepts and technologies of parallel computing used thoughout this work. It also contains a description of the Flynn's categorization of parallel architectures and a description of OpenMP, OpenCL together with examples of their usage and applications.

4.1 INTRODUCTION AND MOTIVATIONS

Traditionally performance improvements in computer architectures have come from cramming ever more functional units onto silicon, increasing clock speeds and transistors number. Moore's law, shown in Figure 4.1, states that the number of transistors that can be placed inexpensively on an integrated circuit will double approximately every two years. Coupled with increasing clock speeds, CPU performance has until recently scaled likewise. But it is important to acknowledge that this trend cannot be sustained indefinitely or forever. Increased clock speed and transistors number require more power and consequently, generate more heat. Although the trend for transistor densities has continued to increase steadily, clock speeds began to slow circa 2003 at about 3GHz. If Moore's law type of thinking is applied to clock-speed performance, it should be able to buy at least 10GHz CPUs. However, the fastest CPU available at the time of writing is $\approx 4.0GHz$. At same point the gain in performance in terms of clock speeds fails to increase proportionally with the additional efforts needed to overcome heat dissipation problems that in turn, become more and more important and challenging. The heat emitted from the modern processor, measured in power density rivals the heat of a nuclear reactor core (see Figures 4.2 and 4.3)! Additionally, the transistor resolution on the wafer is not far from from its physical limit (at the atomic scale), preventing further improvements. But the power demand did not stop in these year, and is not going to stop in the near future, and from these reason comes the necessity of relying heavily on parallel architectures. Today, the dominating trend in commodity CPU architectures is multiple processing cores mounted on a single die operating at "reduced" clock speeds sharing resources and memory with each other. Multi-core (2,4,8,12, up to 40) CPUs on a desktop PC at home or at the office are ubiquitous at the point that is even hard to be able to buy a single core device. Even smart-phones are proper multi-core machines; for instance, the popular mobile CPU, the *Snapdragon 835*, manufactured by *Qualcomm* is a $8\times$ cores, each of them with a clock speed up to 2.45 GHz. Lot of efforts have been put during these years in order to mitigate and overcome the limits that the sequential computer architecture has which its three main components impose:

1. Processor (Cores, branch prediction etc.)
2. Memory (Ram, Caches, Registers etc.)
3. Communication system i.e. datapaths,usually buses (PCI or SCSI)

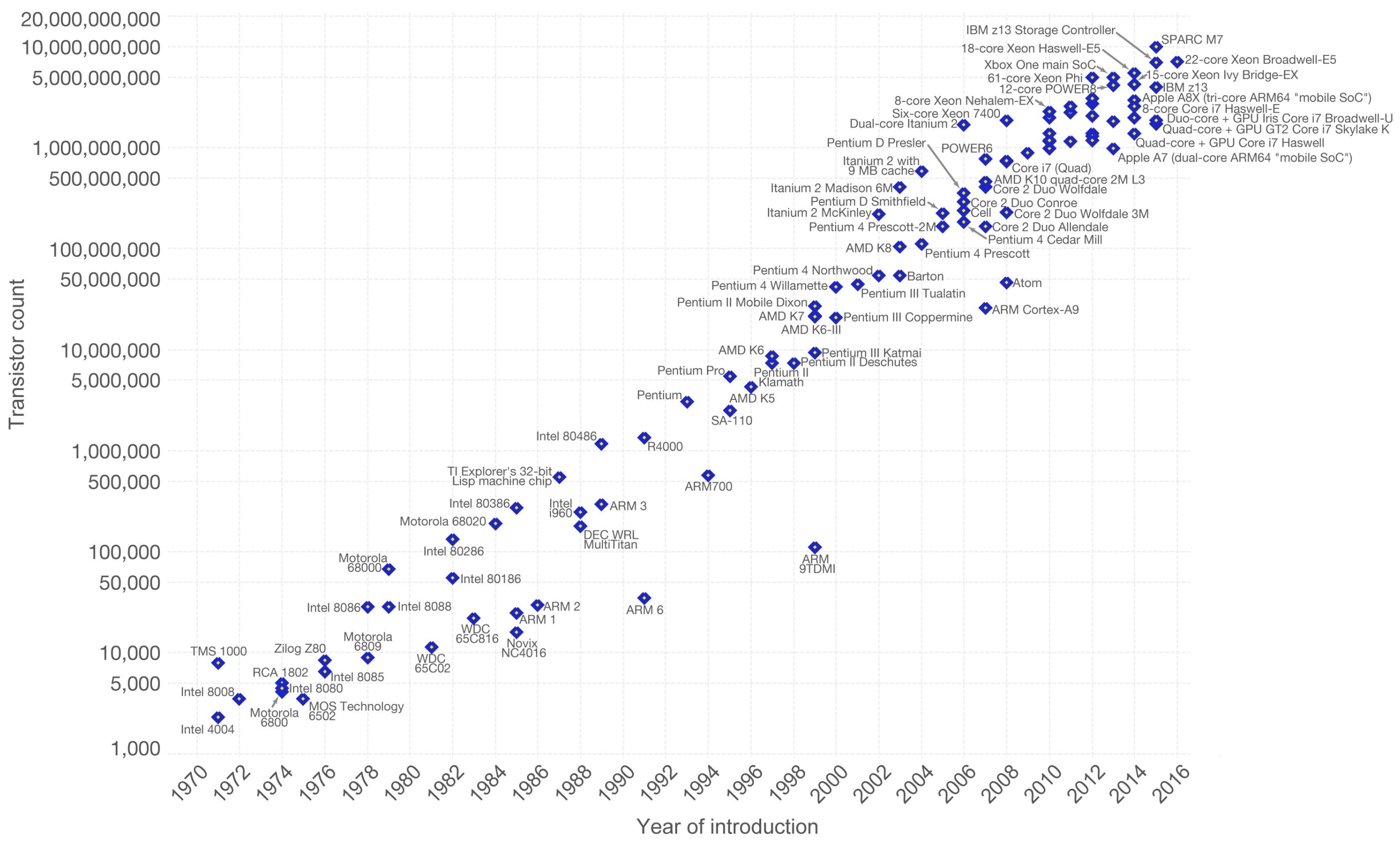


FIGURE 4.1: Moore's Law regarding CPU transistors number. INTEL co-founder, Gordon Moore in 1965 observed that number of transistors in integrated circuits doubled each year. Although the pace has slowed in recent times, the number of transistors per square centimeter has since doubled approximately every 18 months. This is used as the current definition of the law. It is *expected* to hold true until 2020-2025. Note the logarithmic vertical scale. The almost linear trend correspond to an exponential growth.

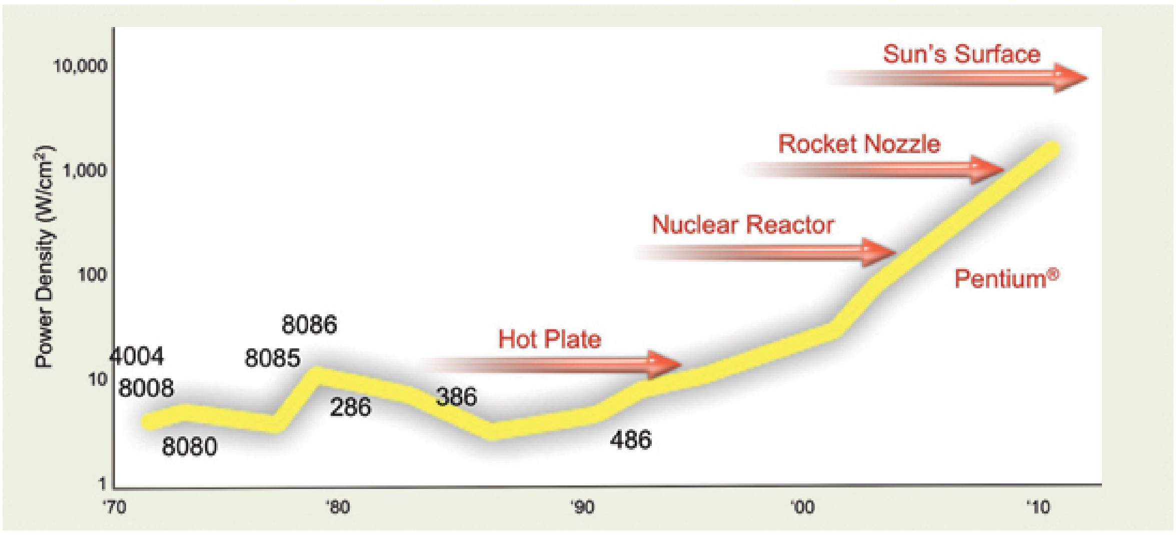


FIGURE 4.2: Integrated circuits power density over the years. Note that since 2000s power-density is higher in CPUs than in nuclear reactors. This clearly shows that this trend of growth is not sustainable.

All three components present bottlenecks that limit the overall computing performance capability of a system. Caches, low-latency high bandwidth and small capacity high speed memories, for example can hide latency of DRAM storing the fetched data and serving subsequent requests of the same memory (or neighboring) locations[1]. But one of the most important innovation that addresses these bottlenecks is **multiplicity** (in processors, memories and datapaths) that allows to improve the overall performance of a system and thus extending the size of the problems that a computer can solve. Hardware multiplicity has been organized in several manners during the years, giving birth to a variety of parallel computer architectures.

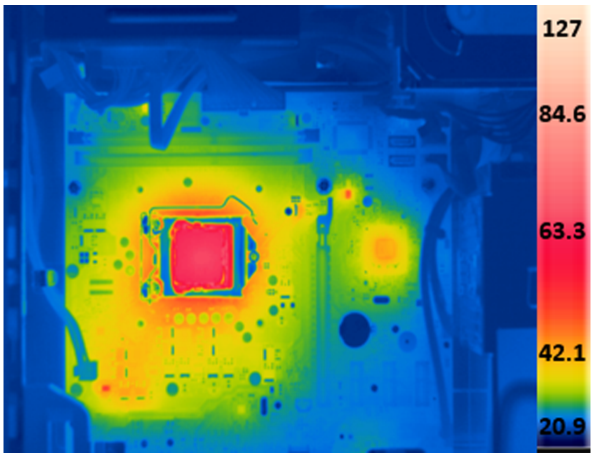


Figure 4.3: Thermal camera image of a modern CPU showing that the whole CPU heat is concentrated in a small part of the wafer i.e. power density is. is very high in some areas.

4.2 parallel architectures - flynn's taxonomy

A number of definitions and classifications have been proposed over the years in order to categorize parallel systems, the majority of them are mostly based on the adopted hardware configuration or on the logical approach in handling and implementing the parallelism. Among all, *Flynn*'s taxonomy [49, 50] is probably the most famous and it is also well accepted by the scientific community, therefore introduced briefly in this section.

Flynn's classification is based on the notion of *stream of information*. Two types of information flow into the processor: **instructions** and **data**. Conceptually they can be separated into two independent streams. A coarse classification can be made taking in account only the number of streams of both instructions and data that an architecture can manage concurrently (see Figure 4.1). Flynn's taxonomy classifies machines according to whether they have one or more streams of each type. Flynn's classifies architectures into four main categories:

sisd: ***Single*** instruction ***Single*** data.
No parallelism in either instruction or data streams. Each arithmetic instruction initiates an operation on a data item taken from a single stream of data elements. A single control unit fetches a single instruction from the memory. Mainframes belong to this category.

1 The fraction of the data served by the various caches is commonly refeered to as ***hit rate***.

Single Instruction		Multiple Instructions	
Single Data	Multiple Data	Single Data	Multiple Data
SISD	MISD	MISD	MIMD

TABLE 4.1: Flynn's Parallel architecture taxonomy.

SIMD: ***Single*** instruction ***Multiple*** data.

Data parallelism. The same instruction is executed on a batch of different data. The control unit is responsible for fetching and interpreting one instruction at a time. When it encounters an arithmetic or an other data ALU instruction, it broadcasts the instruction to all processing elements (PE), which then all perform the same operation in unison. For example, the instruction might be `add R3,R0`. Each PE would add the contents of its own internal register R3 to its own R0. This is how stream processors work, for example, data elements are distributed across all available data memories and, the same instruction executed on each PE. `SSE` and `AVX` extensions to the $x86$ processors family is an example of such parallelism. One single instruction can operate on up to 512 byte of data (see Figure 4.4 and Listing 4.1). SIMD exploits **data and spatial parallelism** in a synchronous manner.

```
//adds 8 floats in a serial fashion. One at the time in 8 different steps
void multiply_and_add(const float* a, const float* b, const float* c, float* d) {
  for(int i=0; i<8; i++) {
    d[i] = a[i] * b[i];
    d[i] = d[i] + c[i];
  }
}
//A single instruction adds 8 floats in a SIMD fashion
__m256 multiply_and_add(__m256 a, __m256 b, __m256 c) {
  return _mm256_fmadd_ps(a, b, c);
}
```

LISTING 4.1: Multiplying eight floats of one array by eight floats of a second array and add the result to a third array. Serial and SIMD (AVX2) code examples. Note that the AVX2 intrinsic function `__mm256_fmadd_ps` processes twentyfour floats, but it does not map to a single instruction (see Figure 4.4). Instead, it executes three instructions: `vfmadd132ps`, `vfmadd213ps`, and `vfmadd231ps`. Despite this, it executes quickly and it is much faster than looping through the individual elements.

MISD: ***Multiple*** instruction ***Single*** data.

Multiple instruction operating on the same single data stream (ee Figure 4.5). It is a class of system very unusual. No machines in this category have been commercially successful or had any impact on computational science. A type of computer that fits the MISD description is the so called *systolic array* [51, 52] which consists of a network of pipelined primitive computing nodes or processors.

MIMD: ***Multiple*** instruction ***Multiple*** data.

Multiple instructions operating independently on multiple data streams. Most modern computers belogn to this family. A MIMD machine is an example of a true multiprocessor and are often employed to perform the so called *Single Program Multiple*

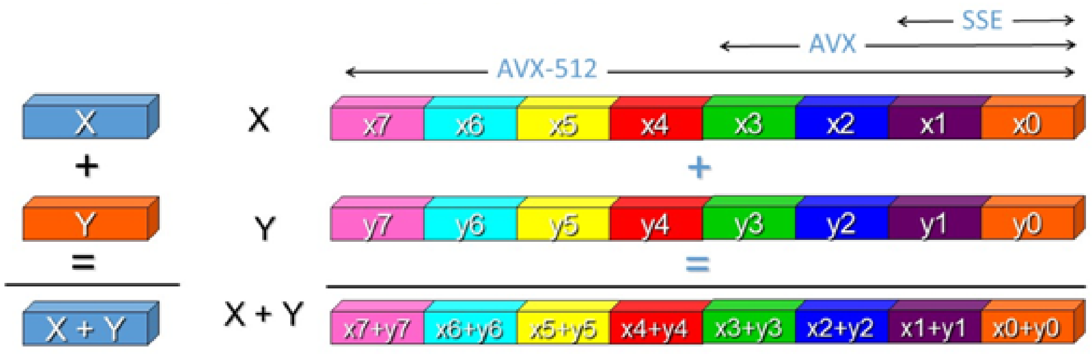


Figure 4.4: Example of vectorization with Intel SSE and extension.

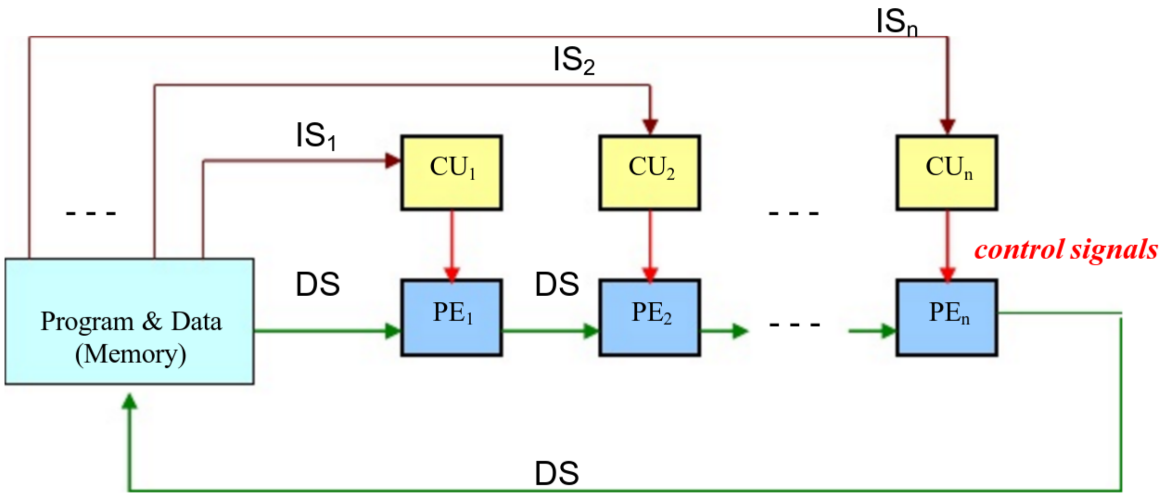


Figure 4.5: MISD machine schematization. Each processor CU_i has its own instruction flow and all operates on the same data.

Data computation where each independent processor executes the same program. Note that modern machine also exposes SIMD capability within each instruction stream. MIMD architecture can be further divided by considering the layout organization of memories:

shared memory (modern cpus) where each processors shares the same memory address space and are interconnected by shared buses. Shared memory architecture are usually shipped as *Symmetric Multi Processors* (SMP) since each of them is usually identical in computational and access to resources capabilities, and the OS kernel can run on any of them in contrast to *Asymmetric Multiprocessors* where there is a master-slave relationship among the group of processors. According to whether subsets of processors have a dedicated/private memory module SM architectures can be categorized into:

- UMA (Uniform Memory Access) : Identical processors with equal access time to memory (see figure 4.6). Also known as *Cache Coherent UMA* (CC-UMA), because the hardware ensures that all the processor can see a modification in memory performed by one of them.

- NUMA (Non Uniform Memory Access): Usually different several *Symmetric multiprocessors (SMP)*, group of usually not more than 32 processors communicating via buses, are inter-connected together, and processors belonging to different SMP can access the memory spaces of each others. The time required for a memory access is not uniform because the cost for a communication among SMP is higher than a intra-SMP one. As for UMA, if a cache coherence mechanism is present, then this architecture is called *CC-NUMA*.

SM architecture provides an easy perspective to memory, data sharing across processors and parallelism since no explicit communication is involved. Memory accesses and communications are fast due to the proximity of memory to CPUs, but it is not scalable because adding more CPUs to the pool can geometrically increases the traffic on the bus and makes cache management harder. Additionally, ensuring the correctness of accesses to global memory, in order to avoid race-conditions, is up to the programmer. As an example of real world SM modern processor, Figure 4.9 shows the Intel *Knights Landing* architecture for the Intel Phi processor family which is armed with 72 cores, 8 billions transistors at 14 nm, AVX-512 and is able to executes 240 threads simultaneously. Many software models (usually tightly coupled with this architecture) can be used to program SM machines. Among all, the most used are:

- Threads; Lightweight processes but with same PID (e.g. pthreads)
- Compiler preprocessor directives; A standard language with preprocessor directives to the compiler that is capable of converting the serial program in a parallel one without any (or very few) interventions or hints by the programmer (e.g. `OpenMP`, introduced in Section 4.3).

DISTRIBUTED MEMORY Different systems, and hence, different multiprocessors connected via some kind of network (see Figure 4.8b), usually high speed networks such as gigabit ETHERNET [53], INFINIBAND [54] and MYRINET [55], where the memory space in one processor does not map to another's one. Each of them operate independently on its memory address, so changes are not reflected on memory spaces of the others. Explicit communication is required between processors with synchronization under the programmer responsibility. This archi-

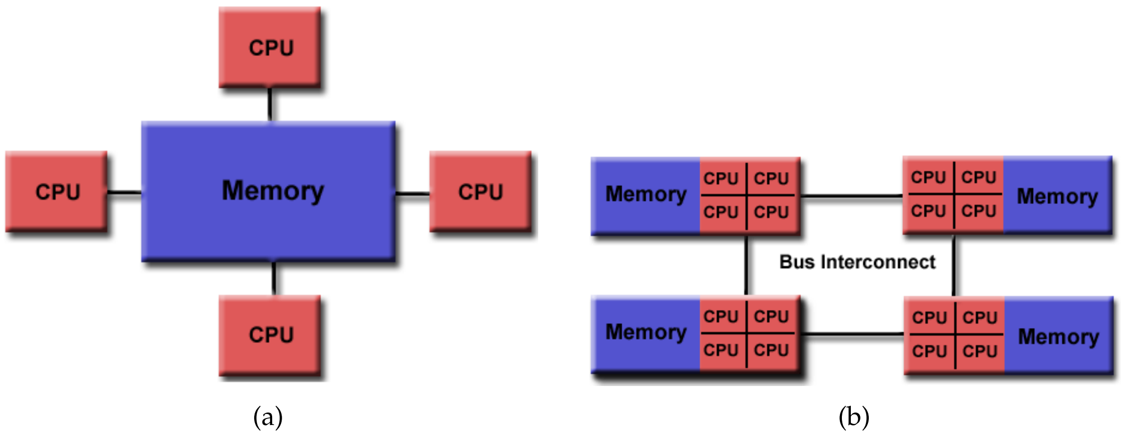


FIGURE 4.6: UMA and NUMA shared memory architecture.

(a) Hybrid memory architecture (each processor is milti-core)

(b) Distributed memory architecture.

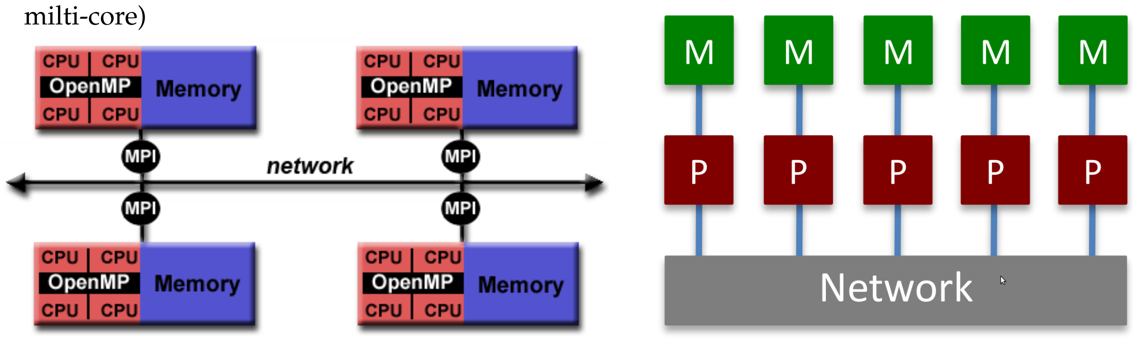


FIGURE 4.8: Hybrid (4.8a) and distributed memory (4.8b) architectures.

tecture is very scalable and there is no overhead in maintaining cache coherency. The most used paradigm for programming distributed memory machines is the message passing for which the *Message Passing Interface* (`MPI`[2]) [56, 57] is the *de facto* industrial and academic standard.

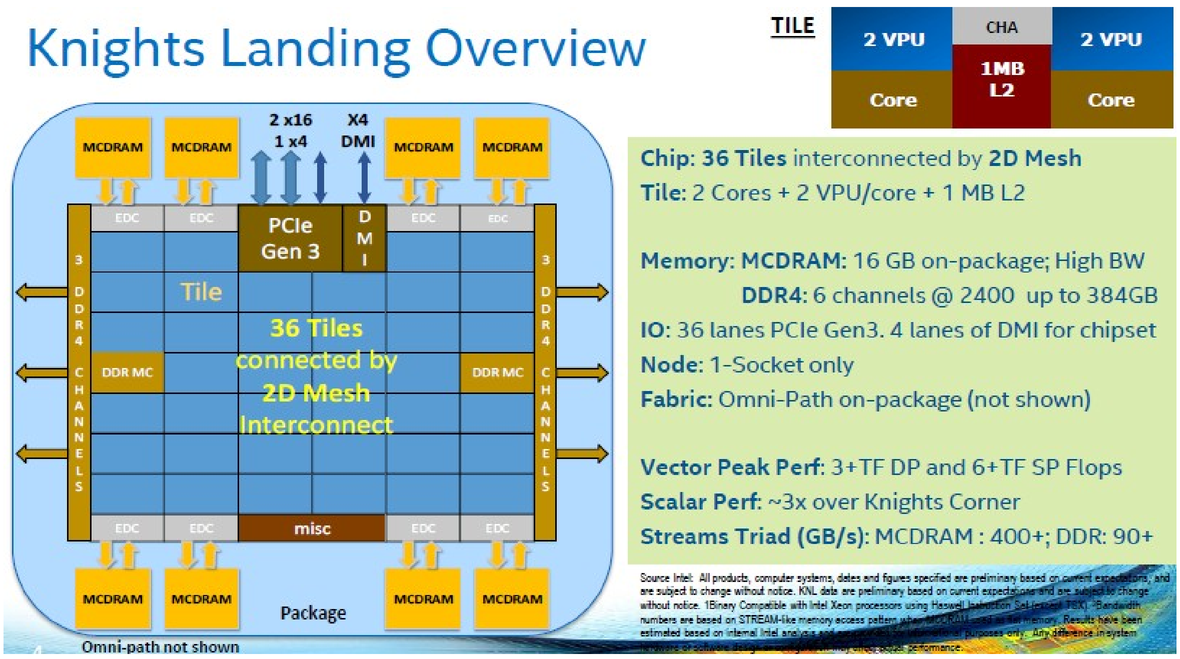
HYBRID SYSTEMS As the name suggests a system belonging to this category, is a mix of architectures. Only a limited number of processors, say N, have access to a common amount of shared memory. They are inter-connected to the others groups via network. Each group usually is an agglomerate of many computing cores (SMP). Hybrid systems of the kind described in this section are usually programmed using a combination of the message passing model (`MPI`) with the threads model (`OpenMP`) in which:

- threads perform computationally intensive task, using local **on-node** memory space, taking advantage of spatial locality of data (via vectorization/AVX for instance) and
- communications between processes on different nodes occur over network using `MPI` (see figure 4.8a).

4.3 THE OPENMP PARALLEL COMPUTING PARADIGM FOR SHARED MEMORY COMPUTERS

`OpenMP` is a portable API providing compiler directives and library functions for shared memory parallel programming in `C/C++` and Fortran [58] designed on top of `pthread` [59]. It implements the multi-threaded *fork-join* programming model (see Figure 4.10), where an initial (or master) thread forks a given number of new threads (team of threads), which share the resources of the parent process and run concurrently on the available processing cores. Threads created during the *fork* phase can therefore rejoin to the master thread (join phase), and more *fork-join* stages can occur in a typical execution of an `OpenMP` executable. The size of the teams can be controlled by an environment variable `OMP_NUM_THREADS`, set at runtime by using the `omp_set_num_threads(n)` function or specified for each parallel region using `num_thread(n)` in the `#pragma omp parallel` directive (see Listing 4.3).

2 `http://www.mpi-forum.org/`



Chip: **36 Tiles** interconnected by **2D Mesh**
Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: **MCDRAM:** 16 GB on-package; High BW
DDR4: 6 channels @ 2400 up to 384GB
IO: 36 lanes PCIe Gen3. 4 lanes of DMI for chipset
Node: 1-Socket only
Fabric: Omni-Path on-package (not shown)

Vector Peak Perf: 3+TF DP and 6+TF SP Flops
Scalar Perf: ~3x over Knights Corner
Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

Source Intel: All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. KNL data are preliminary based on current expectations and are subject to change without notice. 1Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX). 2Bandwidth numbers are based on STREAM-like memory access pattern when MCDRAM used as flat memory. Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

FIGURE 4.9: Xeon Phi, Intel Knight Landing architecture and technical specifications.

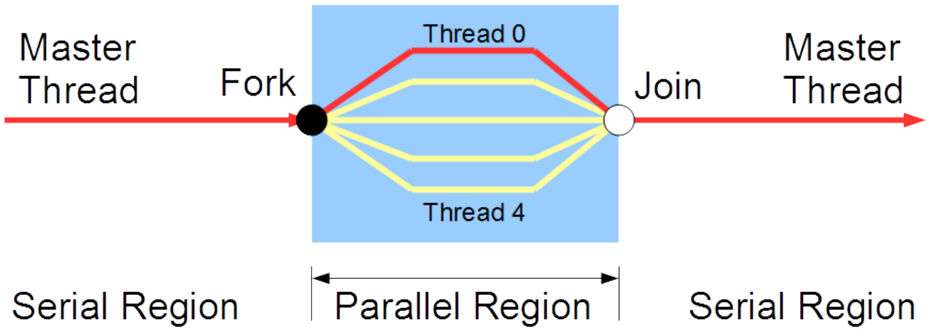


FIGURE 4.10: `OpenMP` fork-join execution model. 1. OpenMP programs start with a single thread; the *master thread* (*Thread* #0). 2. At the starting point of a parallel region, *master* creates team of parallel worker threads (**FORK**). 3. Statements in the parallel block are executed in concurrently by every thread (master as well). 4. At end of parallel region, all threads **synchronize**, and join *master* thread (**JOIN**). Note that parallel regions can be nested.

The *fork-join* model allows for the selective parallelization of the original source code (e.g. loops), by leaving portions that are difficult to be parallelized or that would lead to negligible (or even worsening) improvements, unchanged with respect to the serial implementation. Numerical applications usually accomplish most of their work in a relatively small portion of their codebase, known as *hotspots*. Those are the solely parts of an application that are worth parallelizing. Profilers and performance analysis tools are funamental and employed extensively during in order to identify such hotspots. OpenMP can therefore effectively utilized to share the iterations of a loop within a pool of threads.

By default, static scheduling is adopted where iterations are equally subdivided in chunks and statically allotted to threads in a round-robin policy. However, iterations can also be assigned to threads on demand, by using the *dynamic* scheduling clause. In this manner, when a thread terminates processing its current chunk, it requests for a new one, usually resulting in better performance in the case where load is not well balanced across chunks. Scheduling can be specified using the `schedule` keyword in the `#pragma omp for` directive. Static scheduling can be non-optimal in the case when, for instance, different iterations take a different amount of time to be executed. As an example consider the program in Listing 4.2 in which each loop iteration causes the executing thread to sleep for a number of seconds equals to the number of the iteration.

```
#define CHUNK_SIZE (1)
int main ( ) {
  #pragma omp parallel for schedule(static,CHUNK_SIZE) num_threads(4)
  for (int i = 0; i < N; i++) {
    /* wait for i seconds */
    sleep(i);
    printf("Thread %d has completed iteration %d.\n",
        omp_get_thread_num( ), i);
  }
  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

LISTING 4.2: Scheduling in OpenMP example. Note that specifying **static scheduling** is not needed since it is the default setting.

Among the threads spawned by code in Listing 4.2, there is a great imbalance in the number of seconds they will wait, because the last thread executes the last chunk of iterations (which translates to longer sleep time). *Dynamic* scheduling can be applied in a case like this to improve the overall execution time. `OpenMP` assigns one iteration to each thread and when a thread completes its work, it is assigned the next iteration that has not been executed yet reducing execution time effectively. See table 4.3 for a complete list description of all kind of scheduling in OpenMP.

Moreover, `OpenMP` provides locking mechanism to serialize the access to shared variables by defining critical sections. A lock must be firstly initialized and then can be acquired or released. When a thread attempts to acquire a lock that is already be set by another thread, its execution is suspended until the lock is released, giving rise to performance degradation (or even to possible deadlock situations). However, a lock can also be queried in order to evaluate its state, without blocking the thread execution. In thisway, if the lock is already set, the querying thread can do perform other trasks,

Scheduling Kind	Description
`static`	The loop is divided into equal-sized chunks of iterations. By default, chunk size is $\frac{iterations}{number\_of\_threads}$. If the chunk size is set to 1, iterations are executed by the threads in a interleaved fashion.
`dynamic`	chunk-sized block of iterations are internally queued. When a thread is ready to execute some work, it retrieves the next block from the top of the queue. Note that by default chunk size is 1. Managing the queue and assigning work to threads comes with an arrached overhead.
`guided`	similar execution policy to `dynamic` but the chunk size decreases over time to better handle load imbalance between iterations. In this case the optional parameter of the schedule construct specifies the minimum chunk size. By default it is equal to $\frac{iterations}{number\_of\_threads}$
`auto`	The compiler is free to decide any possible mapping of iteration to threads.
`runtime`	the scheduling policy is choosen at runtime and changes according to the environment variable `OMP_schedule`

TABLE 4.3: `#pragma omp parallel for schedule(kind [,chunk size])` OpenMP scheduling kinds. The optional parameter (chunk size), when specified, must be a positive integer.

thus minimizing idle time. See Listing 4.3 for an example of lock usage in OpenMP.

```
1  omp_lock_t writelock;
2  omp_init_lock(&writelock);
3  #pragma omp parallel for num_threads(5)
4  for ( i = 0; i < x; i++ ){
5  // some stuff in a concurrent fashion
6  omp_set_lock(&writelock);
7  // one thread at a time stuff
8  omp_unset_lock(&writelock);
9  // some more  stuff in a concurrent fashion
10 }
11 omp_destroy_lock(&writelock);
```

LISTING 4.3: `OpenMP` lock acquisition, usage and destruction example. Note that the region of code between lines 6 and 8 is serialized.

In addition to the data-type parallelization provided by the fork-join model, `OpenMP` aslo supports the functional-type parallelization, where different portions (called *regions*) of code to be processed are assigned to different threads. In both cases, `OpenMP` parallelization of a code is straightforward, by hiding most low-level implementation details. Moreover, by using `OpenMP` it is possible to build the same source code to produce both parallel or sequential executable. In the latter case, the compiler simply ignores the `#pragma omp` directives.

4.3.1 *General Purpose GPU Computing - GPGPU*

The concept of many processors working together in concert is not new in computer graphics. Since the demand generated by entertainment started to grow, multi-core hardware emerged in order to take advantage of the high amount of parallel work available in the process of generating and rendering and manipulation of 3*D* images. The main goal of the computer graphics is to render and then display 3*D* images onto the screen, which translates to refreshing pixels at rate of 60 or more Hz [60]. Each pixel has to be processed goes through a number of stages, and this process is commonly referred as to the *graphic processing pipeline*. The peculiarity of this task is that the computation of each pixel is independent of the other's. This specific task is thus suitable for distribution over parallel processing elements as it can be categorized as a **embarassingly, perfect** or **pleasingly** *parallel* problem [61] (see Section 6.3 and Listing 6.8 at pages 113 and 115 respectively, for an example of a perfectly parallel problem). To support extremely fast processing of large graphics data sets (which mainly consist of vertices and fragments), modern GPUs employ a stream processing model with parallelism. The game industry boosted the development of GPUs, that offer today greater performance than CPUs and are improving faster too as shown in Figures 4.12 and 4.11. The reason behind the discrepancy in floating-point capabilities between CPU and GPU is that GPUs are designed such that more transistors are devoted to data crunching and processing rather than caching, flow control, branch prediction, etc. Also note that memory bandwith is still one of the main advantages of GPU and Xeon Phi over CPUs architectures. The gap is getting wider thanks also to the introduction of High Bandwidth Memories (HBM), stacked memory by NVIDIA and MCDRAM for Xeon Phi [62, 63]. Nowadays, GPU are widely used for general purpose computing. The Top 500 supercomputers ranking [3] [64] is dominated by massively parallel computer, built on top of superfast networks and millions of sequential CPUs working in concert. But as the industry is developing even more powerful, programmable and capable GPUs in term of GFLOPS, they begin to offer advantages over traditional cluster of computers in terms of economicity and scalability as depicted in Figure 4.13.

4.3.2 *From Graphics Graphics to General Purpose HW*

A graphics task such as rendering a 3*D* scene on the screen involves a sequence of processing stages inside the GPU, i.e. shaders, that run in parallel and in a prefixed order, known as the graphics hardware pipeline [65] which is the most common form of 3D computer rendering, distinct from, for instance, *raytracing* or *raycasting* for which the concept of pipeline is not even defined.

3 `http://www.top500.org/statistics/list/`

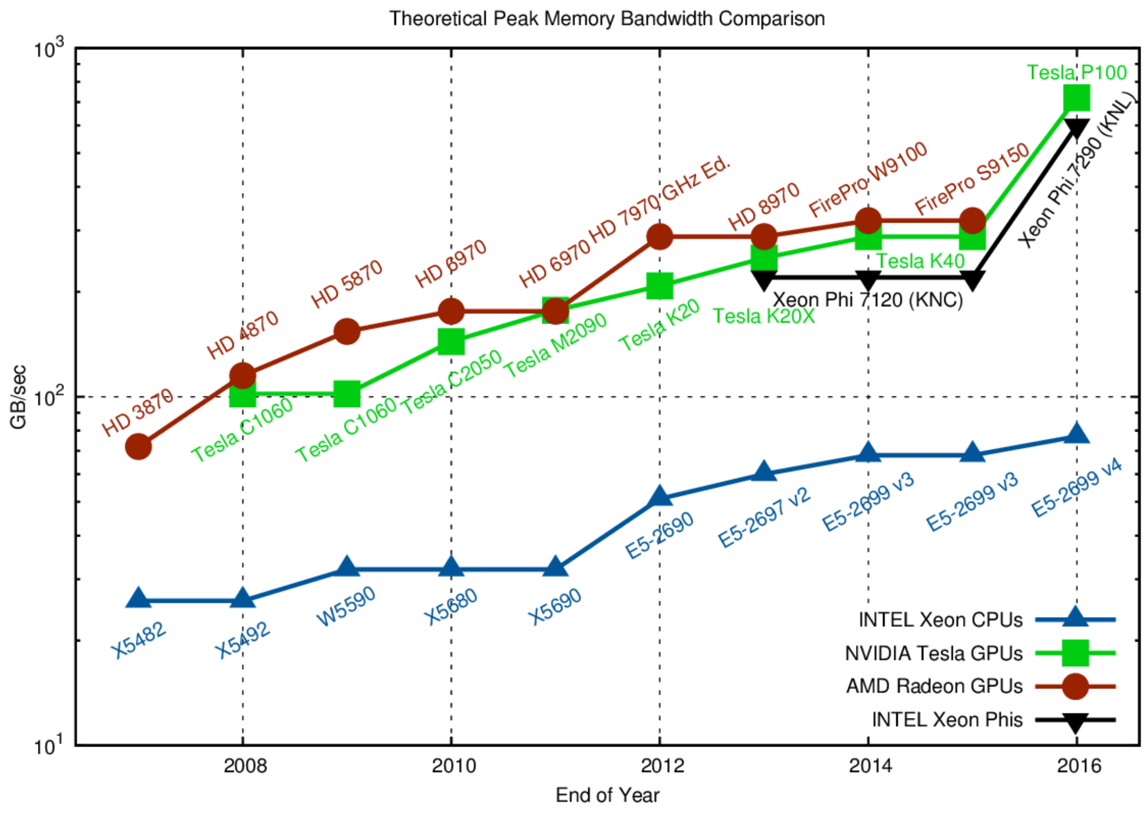


Figure 4.11: Memory Bandwidth comparisong between Intel CPUs and NVIDIA chips over time. Higher is better.

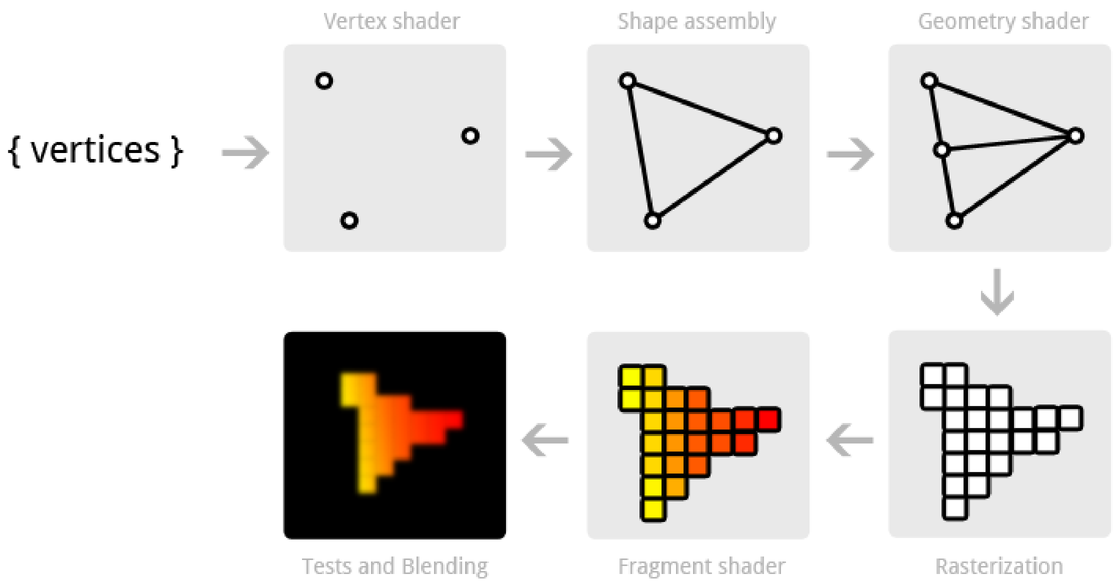


Figure 4.14: Typical computer graphics pipeline.

Figure 4.14 shows the important key steps that make up the graphic pipeline and for which the GPU hardware has been specialized for years. The pipeline works taking as input graphics primitives, each stage forward its results on to the next stage.

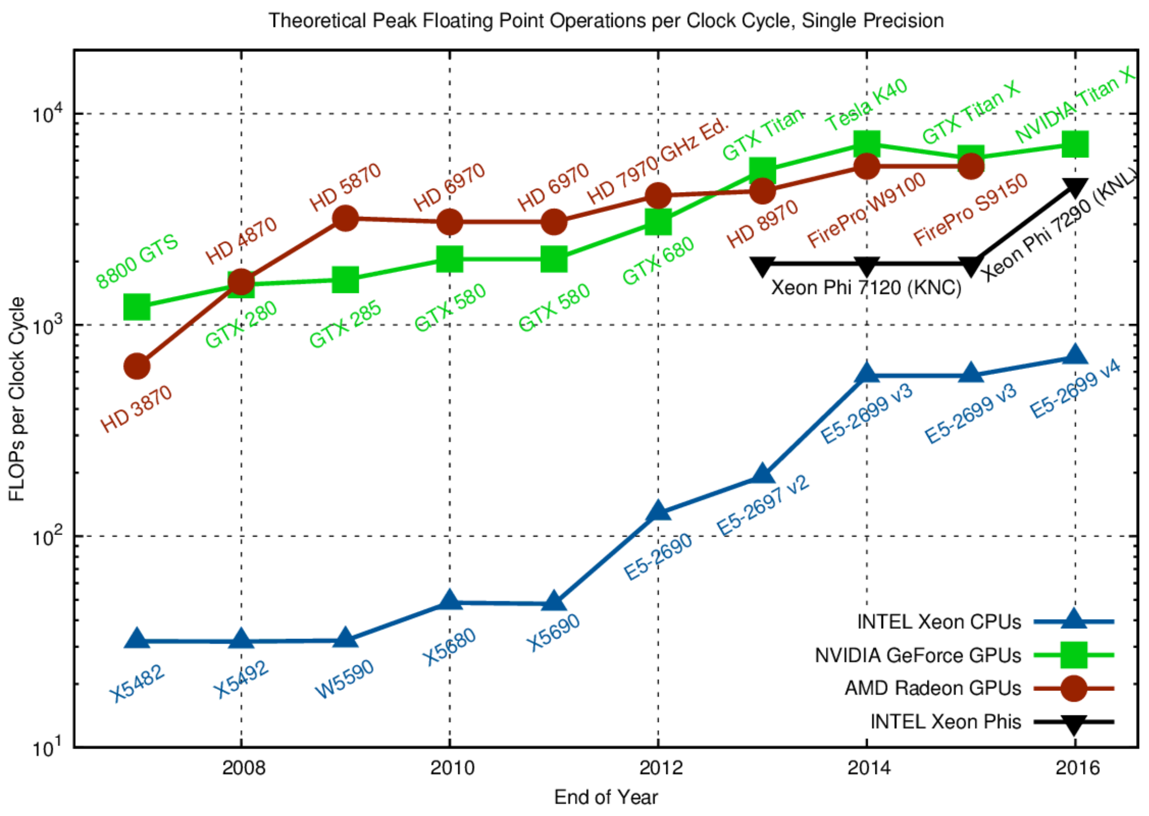


FIGURE 4.12: Performance comparison (FLOPs) between CPU and modern accelerators (NVIDIA, INTEL and AMD) over time. When Double precision is considered, relative performance between the considered hardware does not change. Higher is better.

- The first stage of the pipeline is the *vertex shader*. The input to this phase is a list of vertices in object space coordinates which are then converted to world coordinates (applying the *model and view matrices*).
- Shape assembly is performed where vertices are grouped together by forming graphics primitives, i.e. lines, point, polygons, triangles, tringles strips etc. If enabled, lighting calculation is also performed for each vertex.
- The following step, the *geometry shader* is optional in the `openGL` pipeline and can be used to produce or delete primitives. One of the most common use of this shader is in reducing the communication between the GPU and the CPU. For instance one can only pass to the graphic pipeline a list of vertices for drawing cubes, and produce the primitives for them only when the geometry shader is executed, reducing the amount of information exchanged between the CPU and the GPU.
- Each of the (from the now final list of) primitives is scan-converted or rasterized generating a set of fragments in screen space for the visible only part of the shapes. Each fragment stores the state information needed to update a pixel and are obtained by interpolating per vertex attributes coming from the geometry shader. For instance, each vertex of a triangle end up having its color changed based on the color of its neighbors.

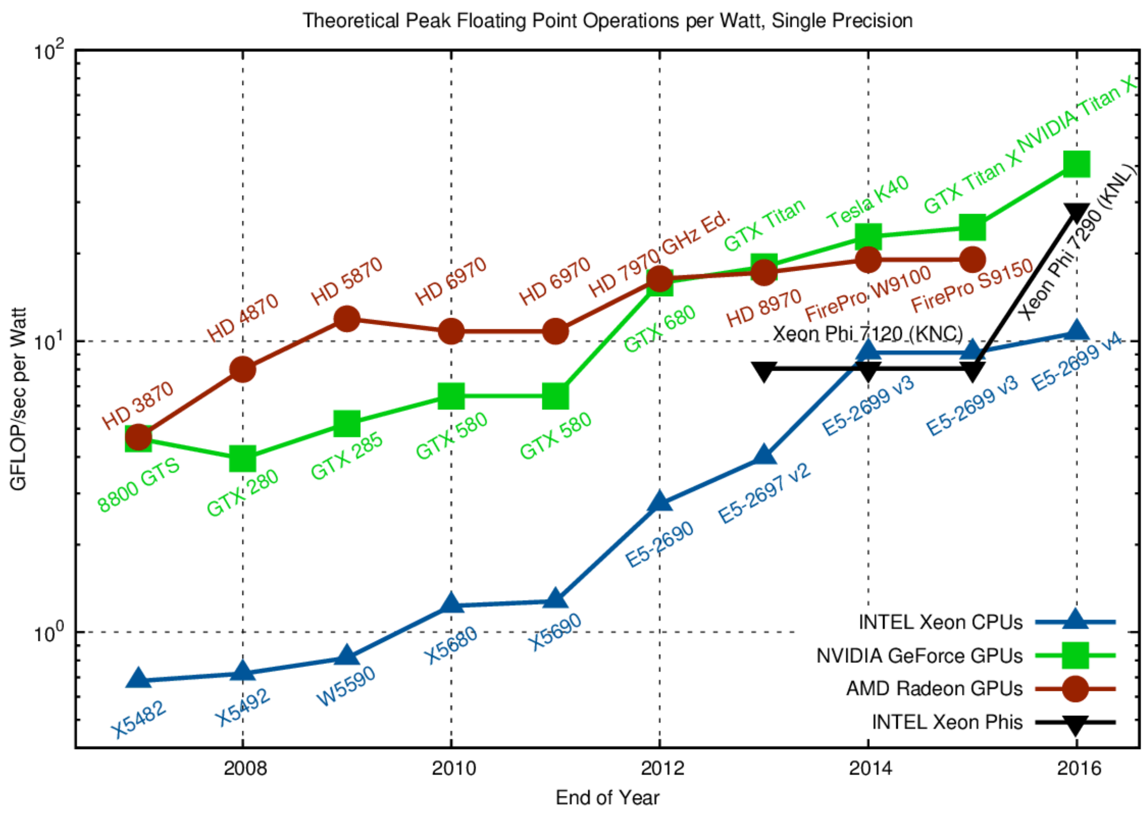


FIGURE 4.13: GFLOPs per Watt, Higher is better. GPUs offer higher performance per Watt utilised especially for those fine-grained massively parallel problems such as dense matrix-matrix multiplication. Note that the Figure also shows that the CPU is getting smaller, indicating that CPU are introducing more GPU-like capabilities into their transistors. *Intel* added wider vector processing units (up to 64 byte) to their latest processors.

- The *fragment shader* is used to calculate the final color of each individual fragment. Texture coordinates of each fragment are used to fetch colors of the appropriate texels (texture pixels) from one or more textures. Further interpolation may also be performed to determine the ultimate color for the fragment.
- Finally, various tests (e.g., *depth* and *alpha*, etc.) are conducted to determine whether or how the fragment should be used to update a pixel in the frame buffer. Each shader in the pipeline performs a basic but specialised operation on the vertices as it passes.

In a shader based architecture the individual shader processors exhibit very limited capabilities beyond their specific purpose. Before the advent of CUDA in 2006 most of the techniques for non-graphics computation on the GPU took advantages of the programmable fragment processing stage. The steps involved in mapping a computation on the GPU are as follows:

1. The data are laid out as texel colors in textures;
2. Each computation step is implemented with a user-defined fragment program. The results are encoded as pixel colors and rendered into a pixel-buffer (stored into GPU main memory, similar to a frame-buffer);

3. Results that are to be used in subsequent calculations are copied to textures for temporary storage and the process could start over again for another iteration.

The year 2006 marked a significant turning point in GPU architecture. The `G80` was the first NVIDIA GPU to have a unified architecture whereby the different shader processors were combined into unified stream processors (see Figures A.2 and A.3 at pages 174 174 respectively). The resulting stream processors had to be more complex so as to provide all of the functionality of the shader processors they replace. Although research had been carried out into general purpose programming for GPUs previously, this architectural change opened the door to a far wider range of applications and practitioners. GPUs are nowadays, well suited for data-parallel problems because they are very effective at executing the same code on many data elements at the same time in a *Single Instruction, Multiple Threads* (SIMT) fashion or using a more general definition as *Parallel Random-Access Machine in which each thread Can Read or Write a memory location* (`CRCW PRAM`).

4.4 THE OPENCL PARALLEL COMPUTING PARADIGM ON HETEROGENEOUS DEVICES

Released on December 2008 by the Kronos Group, `OpenCL` [66–68] is an open standard for programming heterogeneous computers built from CPUs, GPUs and other processors. It allows to define the intended computation using the *platform* abstraction. A platform is composed by an host and one or more compute devices. A C-like language is used to program and orchestrate the various components of the platform (see Figure 4.15).

One of the advantages of `OpenCL` is that it is not restricted, as in the case of CUDA [69], to the use of GPUs only but it takes each computing resource in the system as computational peer unit, interposing a uniform set of API between them and the programmer, easing the process of interfacing with them. Another big advantage is that it is open, free, and cross-compatible across vendors since is supported by all major hardware producers.

A typical `OpenCL` application is subdivided in two parts, one running on the CPU (*host* application) and one or more running on a compliant device (*device* application), where the actual parallel computation generally takes place. The host application defines the tasks to be executed in parallel. Each parallel task is implemented as an `OpenCL` *kernel*, which is a special C function, which is compiled at runtime for each specifically for and deployed to a compliant device, or to different ones, for execution. The execution model is similar to the one of CUDA (CUDA and OpenCL share a similar programming model and underlying hardware architecture, even if a quite different terminology is adopted), where each kernel is executed by **threads**, the smallest execution entity, also called **work-items**, which are grouped into **work-groups**. A work-item is executed by one or more processing elements as part of a work-group executing on a compute unit. A work-item has to be considered a thread in terms of its control flow and memory model, but the hardward and the compiler can run multiple work-items on a single thread. As an example one can imagine that work-items computation can be carried out on lanes of a SSE vector. A work-group is a collection of related work-items that are executed on a single compute unit. A work

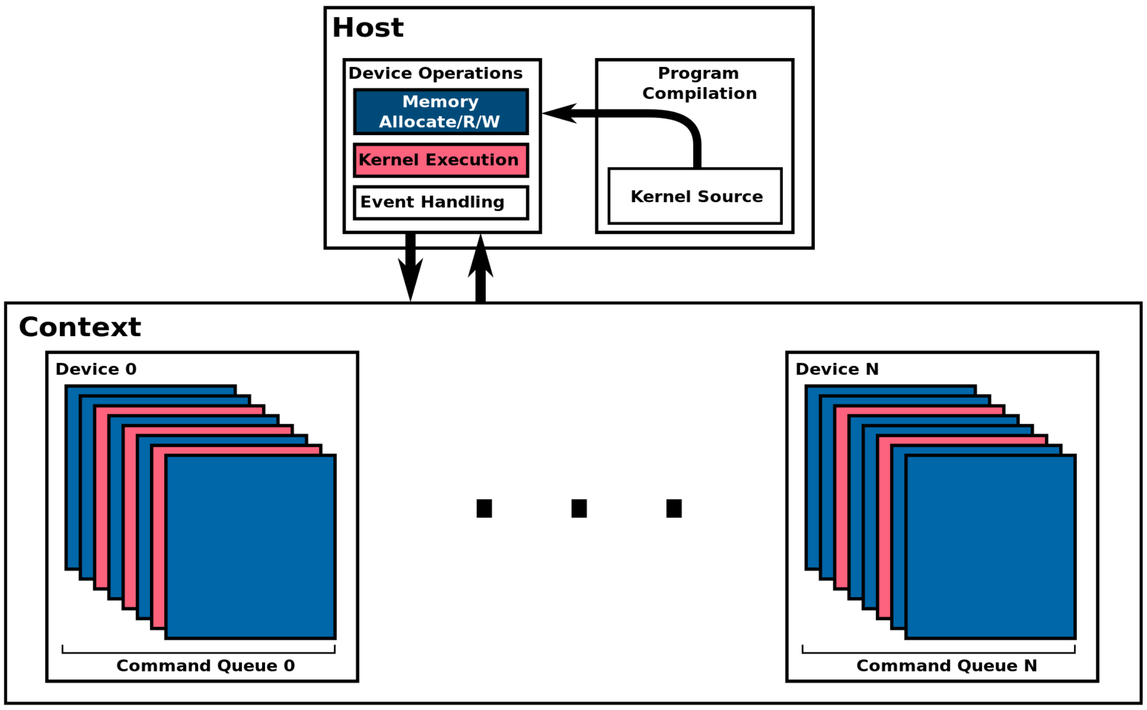


FIGURE 4.15: `OpenCL` platform abstraction. A device is any supported device including GPU, CPU, FPGAs, etc. Command queues are executed concurrently for each device and can be synchronized by means of API calls.

group must map to a single compute unit (a core on a CPU, or using CUDA terminology a streaming multiprocessor). Work-groups can synchronize internally between work-items using local or global memory or barriers but they cannot synchronize with each other, making impossible building locking and synchronization primitives (among work-groups). The locality of execution of work-items leads to more efficient synchronization, and makes possible to have access to user managed local fast memories and caches (similar to CPU L1 caches) in order to makes communication fast. A work-item is distinguished from other executions within the collection by its **global ID** and **local ID** (relative to the parent worl-group).

Work-groups can:

- **Share data** between the work-group's work-items using local memory
- **Synchronize** between work-items using barriers and memory fences mechanism
- **Use special built-in functions** such as `work_group_copy`

Work-groups and work-items are arranged in a indexed grid-like structure. When launching the kernel for execution, the host code defines the grid dimensions, or the global work size. The host code can also define the partitioning to work-groups, or leave it to the implementation. During the execution, the implementation runs a single work item for each point on the grid (a kernel per work-item). It also groups the execution on compute units according to the work-group size. The order of execution of work items within a work-group, as well as the order of work-groups, is implementation-specific (see Figure 4.17). Data to be processed has to be explicitly partitioned and assigned to compute units because each work-item runs the same kernel on different portions of data in a *SIMD/SIMT* fashion. For example, in case of

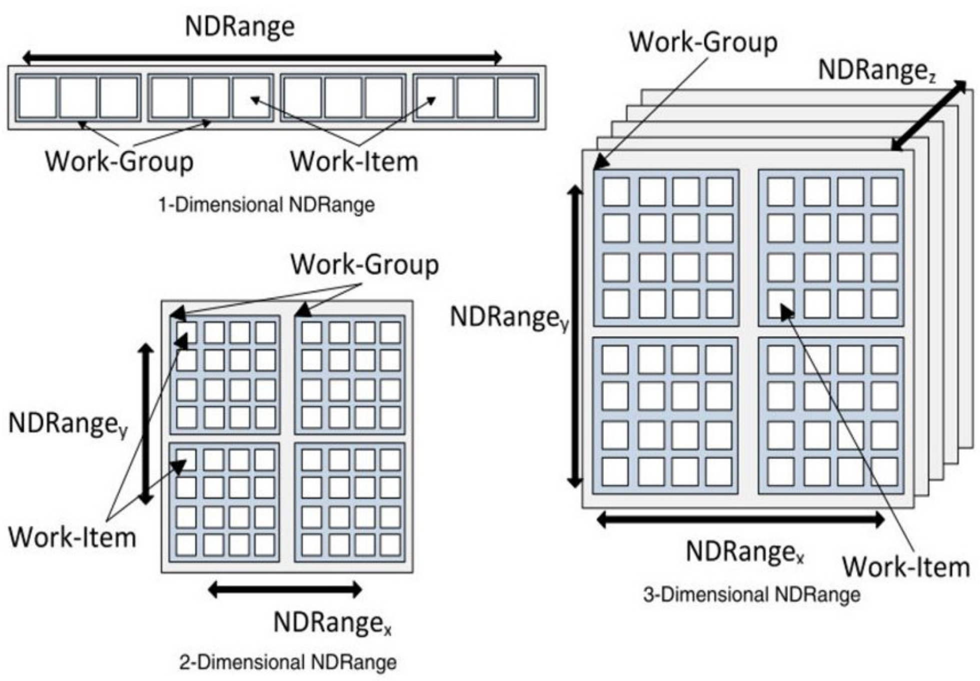


FIGURE 4.16: `OpenCL` 1D,2D,3D work-items and work-groups partitioning.

an array of n elements and n work-items, data can be partitioned by associating each work item to the array element with index corresponding to the work-item global ID. Figure 4.16 depicts how items and groups can be arranged when partitioned in $1D$, $2D$ and $3D$. Figure 4.18 shows a $2D$ decomposition with details on global ID computation from local group and thread indices.

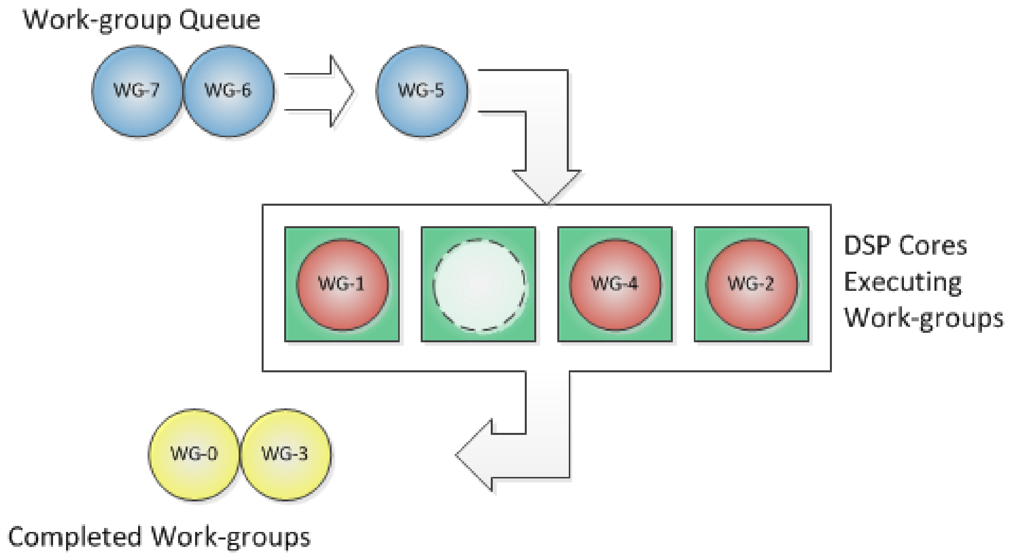


FIGURE 4.17: `OpenCL` work-groups scheduling. The green boxes represent the computing unit. The circles represent the work-groups. Blue work-groups are waiting to be executed, pink work-groups are currently executing and yellow work-groups have been completed. Each work group is queued for execution and executes on a single computing unit (a GPU multiprocessor, CPU core, etc.) Note that execution order is not guaranteed by the standard.

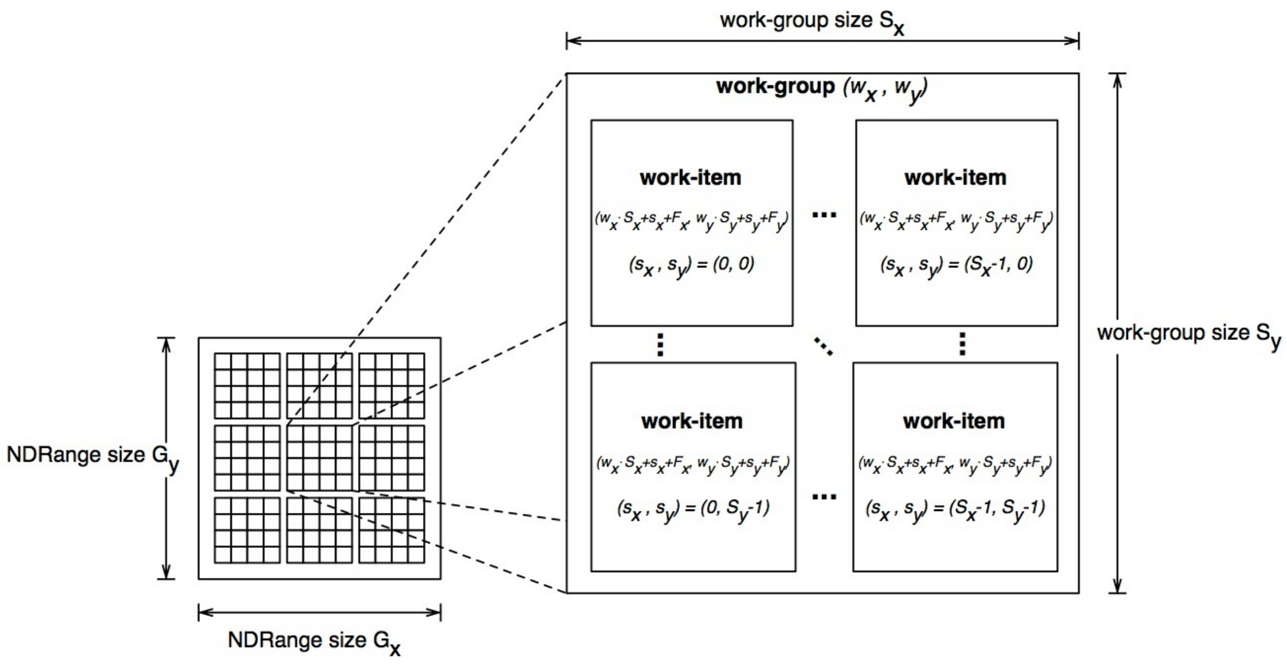


FIGURE 4.18: `OpenCL` 2D work-items and work-groups detailed partitioning. The computation of global index from local item and group index is also shown.

When the kernel execution terminates, work-items globally synchronize, and the control returns to the host application. Similarly to the OpenMP fork-joins stages, different kernel executions and synchronization stages can take place in a typical `OpenCL` application.

Data can be shared by all the running work-items by means of the device global memory, which is generally the largest among all the different memory levels available on the device (this is especially true for modern GPUs) though being the slowest. A read-only memory, equivalent to the global one in terms of latency and dimension, called constant memory, is also available. Some devices have an appropriate portion of this memory, while in other cases the constant memory space coincides with that of the global memory. Threads within a work-group are executed by a specific compute unit and therefore can share data on the local memory and also synchronize each other. Local memory is generally smaller with respect to the global one, but allows for faster access (about 100× faster on modern GPUs). Eventually, each work-item has its own private memory, which is at the same time the fastest and the smallest one.

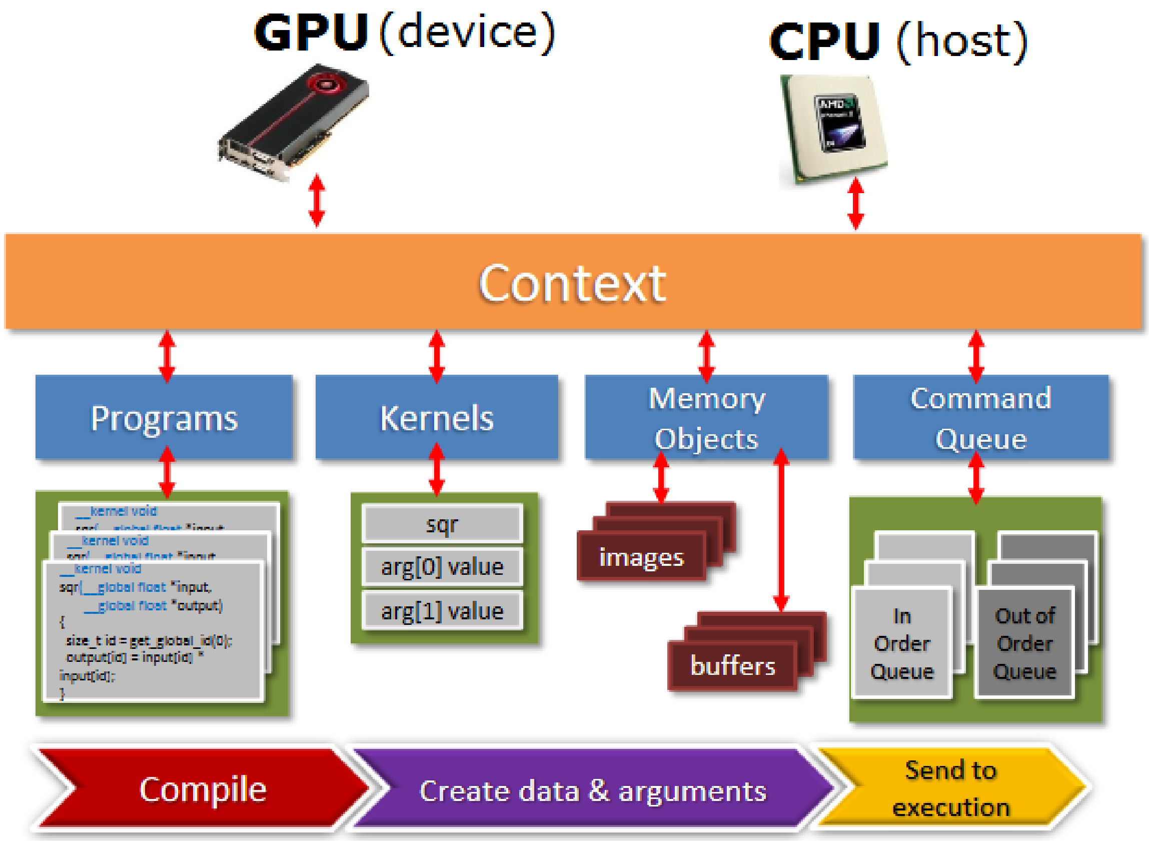
The memory in which a given data is stored must be initially defined and allocated by the host using the appropriate API calls (e.g. see Listing 4.4).

```
clCreateBuffer(cl_context context,  cl_mem_flags flags,
        size_t size,void *host_ptr,cl_int *errcode_ret)
```

LISTING 4.4: Allocate buffer API call in OpenCL.

Nevertheless, data can move among different memory levels during kernel execution (from global, to local, to private or the other way round).

Data exchange and kernels execution are managed by the host thanks to an `OpenCL` context. In particular, the host application links kernels into one or more containers,



```
__kernel void
sqr(__global float *input,
    __global float *output)
{
 size_t id = get_global_id(0);
 output[id] = input[id] *
input[id];
}
```

FIGURE 4.19: OpenCL program flow and interaction between memory, device and host.

called *programs*. The program therefore connects kernels with the data to be processed and dispatches them to a special OpenCL structure called *command queue* (see Figure 4.15). This is necessary because only enqueued kernels are actually executed. The context contains all the devices, command queues and kernels, whereas each device has its own command queue each containing the kernels to be executed on the corresponding device. Moreover, an OpenCL application can configure different devices to perform different tasks, and each task can operate on different data. OpenCL is thus capable of full task-parallelism. Command queues are also used for host-device and device-device data transfer operations, synchronization between different kernels, and profiling operations.

Kernels are usually listed in separate files the OpenCL runtime use to create kernel object that can be first decorated with the parameters on which it is going to be executed and then effectively enqueued for execution onto device.

The following is a brief description of the typical flow of an OpenCL application (See Figure 4.19).

CONTEXTS CREATION:

The first step in every OpenCL application is to create a context and associate to it a number of devices, an available OpenCL platform (there might be present more than one implementation). Each subsequent operation (memory management, kernel compiling and running), is performed within *this* context. In the example

4.5 a context associated with the CPU device and the first platform returned by OpenCL is created.

```
cl_int err;
cl::vector<cl::Platform> platformList;
//Gets a list of available platforms.
cl::Platform::get(&platformList);
checkErr(platformList.size()!=0 ?CL_SUCCESS:-1,"cl::Platform::
    get");
cl_context_properties cprops[3] ={CL_CONTEXT_PLATFORM,(
    cl_context_properties)(platformList[0])(), 0};
//create a context based on the first platform from the list
//Constructs a context including a list of specified devices
cl::Context context(CL_DEVICE_TYPE_CPU,cprops,NULL,NULL,&err);
check_error(err, "Conext::Context()");
```

LISTING 4.5: `OpenCL` context creation.

MEMORY BUFFERS CREATION:

`OpenCL` buffer objects on which kernels operate are created at this point using an available and valid context object as shown in Listing 4.6.

```
memobj = clCreateBuffer(context, CL_MEM_READ_WRITE,MEM_SIZE * sizeof(char),
    NULL, &ret);
check_error(err, "Buffer::Buffer()");
```

LISTING 4.6: `OpenCL` context creation

BUILD A PROGRAM:

The actual code that runs on the devices has to be compiled first. The ***cl::Program*** object takes care of building the device code for the devices listed during the context creation. See Listing 4.7 for an example of how ***cl::Program*** are created.

```
std::ifstream file("pathToSourceCode.cl");
check_error(file.is_open() ? CL_SUCCESS:-1, "pathToSourceCode.cl");std::string
prog( std::istreambuf_iterator<char>(file),
(std::istreambuf_iterator<char>()));
cl::Program::Sources source(1,std::make_pair(prog.c_str(), prog.length()+1));
cl::Program program(context, source);
err = program.build(devices,"");
check_error(err, "Program::build()");
cl_kernel kernel = clCreateKernel(program, "nameofthekernel", &ret);
```

LISTING 4.7: `OpenCL` program load and build

KERNEL LAUNCH:

In order a kernel to be executed a *kernel object* must be created. For a given *Program* there would exists more than one entry point, identified by the keyword *__kernel*. One of them is choosen for execution specifying its name in the kernel object constructor. The kernel is eventually effectively executed by putting it into a *cl::CommandQueue*. Given a *cl::CommandQueue* queue, kernels can be queued using *queue.enqueuNDRangeKernel* that queues a kernel on the associated device. Launching a kernel requires some parameters (similar to launch configuration in CUDA) to specify the work distribution among work-groups and their dimensionality and size of each dimension (see listing 4.8). The status of the execution can be tested by querying the associated *event*.

```
      cl::CommandQueue queue(context, devices[0], 0, &err);
      /*cl_kernel kernel = clCreateKernel(program, "nameofthekernel", &ret);*/
      kernel.setArg(0,memobj);
      //let the work-group size be choosen by the implementation
      ret = clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
      &MEM_SIZE, NULL, 0, NULL, NULL)
```

LISTING 4.8: OpenCL command queue definition and kernel enqueuing.

Part II

Seamless Acceleration of Numerical Simulation: The `OpenCAL` framework

5 THE OPEN COMPUTING ABSTRACTION LAYER FOR EXTENDED CELLULAR AUTOMATA AND THE FINITE DIFFERENCES METHOD

There are two kinds of truths: those of reasoning and those of fact. The truths of reasoning are necessary and their opposite is impossible; the truths of fact are contingent and their opposites are possible.

— Gottfried Leibniz

This chapter introduces `OpenCAL`, an open source computing abstraction layer defining a domain specific language for Extended Cellular Automata and the Finite Differences method (see chapters 2 and 3 at page 5 and 19 repectively). Different implementations have been developed, which allow for transparent parallelism and are able to exploit multicore CPUs and manycore devices like GPUs, thanks to the adoption of `OpenMP` and `OpenCL`, respectively, as well as distributed memory architectures and/or multiple GPUs concurrently. System software architecture is presented and the underlying adopted data structures and algorithms are described in detail. Numerical correctness and efficiency have been assessed by considering the well known *SciddicaT* Computational Fluid Dynamics landslide simulation model as reference example. Moreover, a comprehensive study has been performed to device the best platform for execution as a function of numerical complexity and computational domain extent. Obtained results have highlighted the `OpenCAL` suitability for numerical models development and their execution on the most suitable high-performance parallel computational device.

5.1 INTRODUCTION

Scientific Computing [70] is a broad and constantly growing multidisciplinary research field that uses formal paradigms to study complex problems and solve them through simulation by using advanced computing techniques and capabilities.

Different formal paradigms have been proposed to provide the abstraction context in which problems are formalized. Partial Differential Equations (PDEs) were probably the first to be largely employed for describing a wide variety of phenomena. Unfortunately, PDEs can be analytically solved only for a small set of simplified problems [71] and Numerical Methods have to be employed to obtain approximate solutions for real situations. Among them, the Finite Differences Method (FDM) was one of the first considered, still currently employed, to address a wide variety of phenomena such as acoustics [72, 73], heat [74, 75], computational fluid dynamics (CFD) [76, 77], and quantum mechanics [78, 79].

Besides other solutions proposed for numerically approximating PDEs like, for instance, Finite Elements [80] and Finite Volume Methods [81], further formal paradigms were more recently proposed for modeling complex systems. Among them, Cellular Automata (CA) [82] are Turing-equivalent [21, 83] parallel computational models. CA are widely studied from a theoretical point of view [20, 84–86], and their application domains vary from Artificial Life [87, 88] to Computational Fluid Dynamics [89–92], besides many others. In the 80s, an extension of the original CA formalism was proposed to better model and simulate a specific set of complex phenomena [5]. Such an extension is known as Complex or Multi-Component Cellular Automata and was applied to the simulation of debris flows [93, 94], lava flows [95–98], pyroclastic flows [99, 100], forest fires spreading [101, 102], hydrologic and eco-hydrologic modeling [103–105], soil erosion [30], crowd dynamics [106–108], urban dynamics [109], besides others. Please refers to Chapter 2 for an extensive introduction on Cellular Automata and XCA.

Independently from the adopted formal paradigm, the simulation of complex systems often requires Parallel Computing. OpenMP is the most widely adopted solution for parallel programming on shared memory computers [58]. It fully supports parallel execution on multi-core CPUs and, starting from the 4.0 specification, also includes support for accelerators like graphic processing units (GPUs) or Xeon Phi co-processors. Unfortunately, compilers like gcc currently do not fully support the OpenMP most recent specifications and, in practice, OpenMP-based applications still mainly run on CPUs [97, 110, 111]. However, in recent years, general purpose computing on graphic processing units (GPGPU), which exploits GPUs and many-core co-processors for general purpose computation, has gained wide acceptance as an alternative solution for high-performance computing, resulting in a rapid spread of applications in many scientific and engineering fields [112]. Most implementations are currently based on Nvidia CUDA (see e.g. [113–116]), one of the first platforms proposed to exploit GPUs computational power on NVIDIA hardware. An open alternative to CUDA is OpenCL [117], an Application Program Interface (API) originally proposed by Apple and currently managed by Khronos Group for parallel programming on heterogeneous devices like CPUs, GPUs, Digital Signal Processors (DSPs), and Field-Programmable Gate Arrays (FPGAs). Interest in OpenCL is continuously growing and many applications can already be found in literature [118–121]. However, an OpenCL parallelization of a scientific application is often a non-trivial task and, in many cases, requires a thorough refactorization of the source code. For this reason, many computational layers were proposed, which make many-core co-processors computational power easier to be exploited. For instance, ArrayFire [122] is a mathematical library for matrix-based computation such as linear algebra, reductions, and Fast Fourier transform; clSpMV [123] is a sparse matrix vector multiplication library; clBlas [124] is an OpenCL parallelization of the Blas linear algebra library. Examples of higher level computational layers, which provide the abstraction of formal computational paradigms, are: OPS [125, 126] and OP2 [127, 128], which are open-source frameworks for the execution of structured and unstructured grid applications, respectively, on clusters of GPUs or multi-core CPUs; AQUAgpusph [129], which is a smoothed-particle hydrodynamics solver; ASL [130], an accelerated multi-physics simulation software based, among others, on the Lattice Boltzmann Method; CAMELot [28, 131] and libAuToti [132], which are a proprietary simulation environment and an efficient parallel library for XCA model development, respectively.

Among the above cited softwares, OPS, ASL and CAMELot probably are the most similar to OpenCAL in terms of modeling and development approach, and could be considered as possible alternatives to the library proposed in this thesis. In particular, OPS provides a straightforward Domain Specific Language for structured grid-based modeling, even if it does not refer to any specific abstract computational formalism. Its main characteristic consists in allowing to obtain different parallel versions of a computational model starting from its serial implementation, thanks to a seamless code-generator approach. MPI-based distributed memory, as well as CUDA and OpenCL many-core versions can be obtained in this way, with a minimal effort by the developer. Conversely, ASL provides different higher level modeling abstractions among which the Lattice Boltzmann Method, that is eventually a Cellular Automata-based paradigm. Nevertheless, it currently does not allow for parallel execution on distributed memory systems, which can be a great limitation in some cases. Eventually, CAMELot offers an integrated simulation environment for XCA development and allows for parallel execution on both shared and distributed memory systems thanks to the message passing paradigm, not permitting however the exploitation of modern many-core devices.

This chapter is devoted to the description of `OpenCAL` which aims to be a portable parallel computing abstraction layer for scientific computing. It provides the Extended Cellular Automata general formalism as a Domain Specific Language, allowing for the straightforward parallel implementation of a wide range of complex systems. Cellular Automata, Finite Differences and, in general, other structured grid-based methods are therefore supported. Different versions of the library allow to exploit both multi- and many-core shared memory devices, as well as distributed memory systems. Specifically, OpenMP- and OpenCL-based implementations have been developed, both of them providing optimized data structures and algorithms to speed-up the execution and allowing for a transparent parallelism to the user. A MPI-based implementation is also currently under development and allows to exploit many-core accelerators on interconnected systems. With respect to the above cited softwares, OpenCAL therefore provides both the higher CAMELot modeling approach and, similarly to OP2, allows for the execution on a wide range of shared and distributed parallel platforms (even if by adopting a classic library approach). In addition, OpenCAL provides different embedded strategies and optimization algorithms which allow to progressively improve the computational performance of different kinds of models and simulations. `OpenCAL` is released under Lesser GNU Public License (LGPL) version 3 and is freely downloadable from GitHub at the following link: `https://github.com/OpenCALTeam/opencal`. `OpenCAL` allows for the definition of computational models based on CA, XCA and FDM. It is designed to be easily extended and applied to all computational methods based on regular and uniform grids. The implementation described in this chapter targets shared multi-core, distributed memory and GPUs and is designed to make the parallelism transparent to the user addressing and hiding many aspects of the underlying formal computational model and parallel execution platform.

In the following, the OpenCAL architecture is presented and the OpenMP- and OpenCL-based parallel implementations described. The implementation of a first simple example of application for multi- and many-core devices is also presented and discussed in order to show how straightforward the OpenCAL-based model development is. Therefore, the *SciddicaT* XCA landslide simulation model [133] is then considered as a more complex reference example for correctness and computational performance evaluation on multi-core CPUs, many-core GPUs. In particular, three different versions of *SciddicaT* are refeered to in this chapter, which progressively exploit OpenCAL built-in features and, for each of them, different implementations based on the serial and parallel versions of the library are proposed. Eventually, results of a further study performed to devise the best platform for execution, depending on the model's computational intensity and the domain extent, is presented. A general discussion concerning OpenCAL and future outcomes concludes the chapter.

5.2 OPENCAL SOFTWARE ARCHITECTURE AND THE CONSIDERED PARALLEL COMPUTING PARADIGMS

The OpenCAL architecture is depicted in Figure 5.1. At the higher level of abstraction, the Scientist conceptually designs the computational model, by referring the Extended Cellular Automata general formalism. Structured grid-based models whose evolution is determined by local rules, as well as by global laws or even by a combination of local and global operations, are therefore fully supported. At this level, domain topology and extent, boundary conditions, substates (each of them representing the set of admissible values of a given characteristic assumed to be relevant for the modeled system and its evolution), neighborhood (defining the pattern over which local rules are applied), and elementary processes (defining the local rules of evolution), are formalized. The simulation process is also designed at this level, by specifying the initial conditions of the system, optional global operations (e.g. steering or global reductions), and a termination criterion to stop the system evolution. Boundary conditions can be implemented by the user by creating ad-hoc code within the transition function which treats the boundary cells (which can be identified to belonging to the boundary, within the code) s.t. the boundary condition is enforced. For instance if implementing a heat transfer model, adiabatic walls can be enforced programmatically by setting heat transfer to 0 only for those cells that make up the walls. Any other boundary condition can be implemented similarly. Note that, being supported by OpenCAL, some specific optimizations can be accounted at design time. Specifically, the explicit updating feature allows to both redefine the elementary processes application order and to selectively update substates after the application of each elementary process, while the *active cells optimization*, also known as *quantization*, allows to restrict the computation to a subset of the whole computational domain, by excluding stationary cells.

OpenCAL can be found in the implementation abstraction level. As it can be seen, four different versions can be considered for implementing the previously designed computational model, namely OpenCAL, OpenCAL-OMP and OpenCAL-CL and OpenCAL-MPI. The first one refers to the serial implementation of the library, while second and the third are OpenMP and OpenCL-based parallelizations, respectively, as pointed out by the language/low-level library level. The fourth one is a cluster ready implementation that allows to execution on distributed memory cluster where each node can exploit multiple GPUs. All implementations are written in C for the maximum efficiency and provide high-level data types and functions that match the XCA formal components, allowing for a straightforward implementation of the designed computational model, by also allowing to ignore low-level issues like memory management and I/O operations. In this respect, OpenCAL can be considered as a domain-specific language (DSL) for the CA, XCA and FDM computational methods. Finally, at the hardware level, depending on the adopted version of the library, execution can be performed on single- and multi-core CPUs, or on many-core accelerators like GPUs, almost transparently to the user. Figure 5.1 also shows hybrid MPI+Open-MP and MPI+OpenCL parallel implementations of OpenCAL, which will allow to exploit clusters of CPUs and many-core accelerators (see Chapter 6).

Note that an OpenMP-based parallelization is generally more straightforward with respect to one based on OpenCL or MPI and, when compilers will fully support the 4.0/4.5 OpenMP specifications, it will be possible to execute OpenMP-based applications on both multi-core CPUs and many-core high-performance devices. On the other hand, an OpenCL-based parallelization allows to exploit a wide range of high-performance many-core devices straight

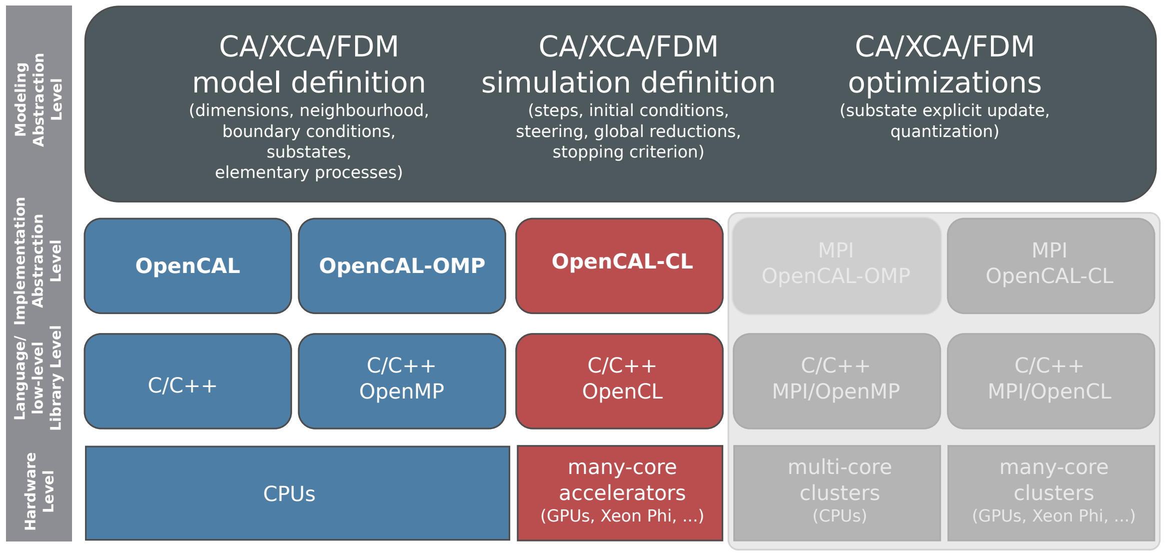

Modeling Abstraction Level
Implementation Abstraction Level
Language/ low-level Library Level
Hardware Level
CA/XCA/FDM model definition
(dimensions, neighbourhood, boundary conditions, substates, elementary processes)
CA/XCA/FDM simulation definition
(steps, initial conditions, steering, global reductions, stopping criterion)
CA/XCA/FDM optimizations
(substate explicit update, quantization)
OpenCAL
OpenCAL-OMP
OpenCAL-CL
MPI OpenCAL-OMP
MPI OpenCAL-CL
C/C++
C/C++ OpenMP
C/C++ OpenCL
C/C++ MPI/OpenMP
C/C++ MPI/OpenCL
CPUs
many-core accelerators (GPUs, Xeon Phi, ...)
multi-core clusters (CPUs)
many-core clusters (GPUs, Xeon Phi, ...)

FIGURE 5.1: `OpenCAL` architecture. At the higher level of abstraction, the models, as well as the simulation process, and possible optimizations are designed. `OpenCAL` can be found at the implementation abstraction layer, allowing for a straightforward implementation of the designed model. In fact, it can be considered as a domain-specific language for the CA, XCA and FDM computational methods, built on top of the C language and the `OpenMP` and OpenCL APIs. `OpenCAL` applications can be executed at the hardware level on both multi-core CPUs and many-core devices, while the execution on cluster is planned but currently not supported.

away and, with greater control on the underlying hardware and on the execution flow, allowing better exploitation of the hardware capabilities. For these reasons, both the `OpenMP` and OpenCL versions of `OpenCAL` have been developed and are maintained.

5.3 THE OPEN COMPUTING ABSTRACTION LAYER

The `OpenCAL` API was designed to be clear and easy to use. For this purpose, it follows some naming conventions, the most important of which are listed below:

- `CALbyte`, `CALint`, and `CALreal` redefine the `char`, `int` and `double` C native scalar types, respectively;
- Derived data types start with the `CAL` prefix (or `CALCL` for some specifc `OpenCAL-CL` data types), followed by a type identifier formed by one or more capitalised keywords, an optional suffix identifying the model dimension (e.g. `2D` or `3D`), and an eventual optional suffix specifying the basic scalar type, which can be `b`, `i`, or `r`, for `CALbyte`, `CALint` and `CALreal` derived types, respectivey (e.g. `CALSubstate3Dr` represents an example of three-dimensional double precision-based data type - cf. below);
- Constants and enumerals start with the `CAL_` prefix, followed by one or more uppercase keywords separated by the _ character (e.g. the `CAL_TRUE` and `CAL_FALSE` boolean enumerals);

- Functions are characterised by the `cal` prefix (or `calcl` for some specifc `OpenCAL-CL` functions), followed by at least one capitalized keyword, and end with a suffix specifying the model dimension and the basic datatype (e.g. `calSet2Di` represents an example of an API function acting on a bi-dimensional integer based data type).

In the following, the `{arg1|arg2|...|argn}` and `[arg1|arg2|...|argn]` conventions will be adopted: the first one identifies a list of n mutually exclusive arguments, where one of the arguments is needed; the second is used to identify a set of n non-mutually exclusive optional arguments. As an example, `calGet[X]{2D|3D}{b|i|r}()` function actually identifies a set API functions with one optional and two mandatory suffixes: the first one, if present, indicates that the fuction is able to access naighborhood data (`X` is the symbol commonly used in the XCA formalism to refer the neighborhood), while the last two ones indicate the domain dimension and the basic type of the data to be accessed, respectively.

Among derived data types, `CALParameter{b|i|r}` represents an alias for the corresponding basic `OpenCAL` scalar data type, and can be optionally used for defining model parameters.

5.3.1 *API*

An `OpenCAL` model is declared as a pointer to `CALModel{2D|3D}` and defined by means of the `calCADef{2D|3D}()` function. The model object stores the dimensions of the computational domain in terms of number of rows and columns (and also slices in case of a 3D model), the computational domain boundary topology (e.g., if a 2D computational domain has to be considered as a bounded or an unbounded torus), the neighbourhood pattern, and also registers pointers to substates and elementary processes composing the transition function. Moreover, it manages a sub-structure which allows to exploit the built-in quantization feature by means of which, based on user-specified criteria, the computation can be restricted to a subset of non-stationary cells (also called active cells) of the whole computational domain.

As regards neighborhoods, `OpenCAL` provides a set of predefined patterns. For instance, the `CAL_MOORE_NEIGHBORHOOD_{2D|3D}` enumeral refers to the Moore pattern (cf. Figures 5.2b and 5.3b). von Neumann 2D and 3D neighborhoods are also predefined, as well as 2D hexagonal patterns (cf. Figures 5.2c and 5.2d). Custom neighborhoods can also be defined in `OpenCAL` by considering the `CAL_CUSTOM_NEIGHBORHOOD_{2D|3D}` enumeral and the `calAddNeighbor{2D|3D}()` function, which adds a cell to the neighbourhood by means of its relative coordinates with respect to the central one. Note that, a zero-based index is assigned to the neigbouring cells in order to address them without the need to provide their relative coordinates, as shown in Figures 5.2 and 5.3. Elementary processes, both local interactions and internal transformations, are defined by means of callback functions and registered to a computational model by means of the `calAddElementaryProcess{2D|3D}()` function. Each elementary process callback must return `void` and takes a list of integer arguments, representing the coordinates of a generic cell in the computational domain. Elementary processes define the `OpenCAL` transition function, and can be implicitly applied by the simulation loop to the cells of the computational domain in the same order in which they were registered to the computational model, or in a user defined order (cf. below).

Substates are defined as pointers to `CALSubstate{2D|3D}{b|i|r}` and can be registered by means of the `calAddSubstate{2D|3D}{b|i|r}()` function. Substates are internally defined by means of two linearized arrays (also called computational layers), having the same di-

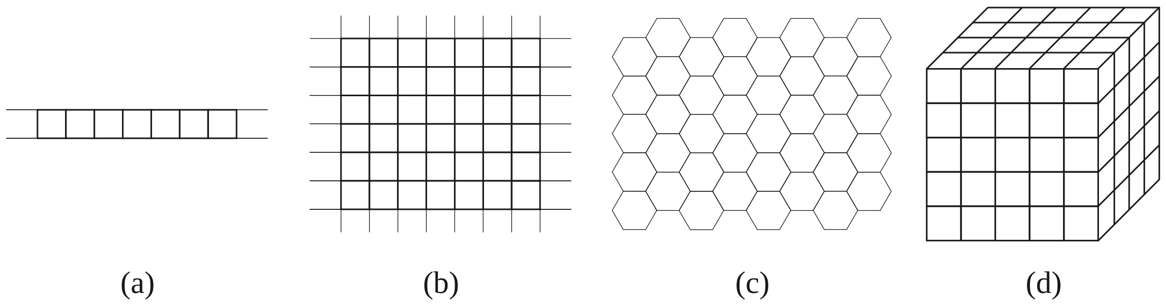


FIGURE 5.2: Examples of *Von Neumann* (a) and *Moore* (b) neighborhoods for two-dimensional CA with square cells. Examples of Moore neighborhoods are also shown for hexagonal CA, both for the cases of horizontal (c) and vertical (d) orientations. Central cell is represented in dark gray, while adjacent cells are in light gray. A reference system is here considered to evaluate cells coordinates in terms of row and column indices in a matrix-style representation, and a 0-based numerical identifier assigned to each cell in the neighborhood for straightforward access.

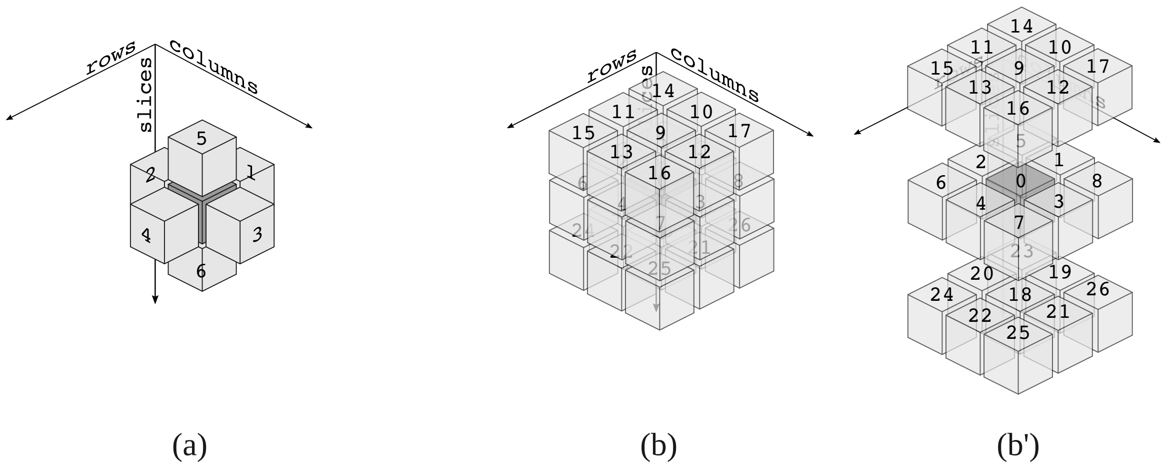


FIGURE 5.3: Examples of *Von Neumann* (a) and *Moore* (b, b′) neighborhoods for three-dimensional CA with cubic cells. Central cell is represented in dark gray, while adjacent cells are in light gray. A reference system is here considered to evaluate cells coordinates in terms of row, column and slice indices in a matrix-style representation, and a 0-based numerical identifier assigned to each cell in the neighborhood for straightforward access.

mensions of the computational domain. The *current* layer is used as a read-only memory for retrieving central and neighboring cells current states, while the *next* one for updating the new value for the central cell. This is a commonly adopted solution for obtaining the *implicit parallelism*, thanks to which cells appear to be simultaneously updated with respect to each other, even in the case of serial computation. Single layer substates can be also defined in `OpenCAL` by simply registering them through the `calAddSingleLayerSubstate{2D|3D}{b|i|r}()` function. In this case, they only consist of the current computational layer, and can be used for internal transformations processing. To retrieve the current value of a substate for a (central) cell by providing its coordinates within the computational domain, the `calGet{2D|3D}()` function can be adopted, while the `calGetX{2D|3D}()` function can be considered for obtaining the same information for a neighbouring cell, by providing an additional parameter specifying the index of the cell in the neighborhood (cf. Figures 5.2 and 5.3 for predefined neighborhoods). Eventually, the `calSet{2D|3D}()` function can be used to set the new value of a substate for the (central) cell to the next computational layer. In the case of a single layer substate, the `calSetCurrent{2D|3D}{b|i|r}()` function has to be employed for updating purposes. It acts like the `calSet{2D|3D}()` function, with the exception that the new value is written on the current computational layer. Note that, after the application of each elementary process, all the registered substates are implicitly updated, i.e. the next layer is copied into the current one. However, this behaviour can be overridden and substates explicitly, as well as selectively, updated by means of the `calUpdateSubstate{2D|3D}{b|i|r}` function (cf. below). Obviously, single layer substates do not need to be updated.

5.3.2 *The quantization strategy*

In many grid-based simulations, system's dynamics only affects a small region of the whole computational domain. For instance, this is the case of topologically connected phenomena, like debris or lava flows. In these cases, a naive approach where the overall domain is processed can lead to a considerable waste of computational resources, even in the case stationary cells (i.e. those cells that do not change their state to the next computational step) are only checked and the application of the evolution rules skipped.

Different approaches have been proposed to improve the efficiency of the naive approach. Among them, the hyper-rectangular bounding box (HRBB) optimization, consisting in surrounding the simulated phenomenon by means of a fitting rectangle (or a parallelepiped, in the case of a 3D model), by contextually restricting the computation to this specific subregion, proved to be a simple but effectiveg approach in different cases (see e.g. the work of **D'Ambrosio2013630** [**D'Ambrosio2013630**]). However, HRBB demonstrated its limit in the simulation of scattered phenomena, where the hyper-rectangle can easily grow up to the whole domain, by however embedding a considerable number of inactive cells.

A more effective approach, which is also able to optimally distribute the computational load in case of parallel execution, consists in maintaining a dynamic set of coordinates of the only active cells during the simulation, by restricting the computation only to this set (see e.g. [115]). The activation state for a cell generally depends on the specific system to be simulated. In many cases, e.g. in computational fluid-dynamics, a threshold-based criterion can be adopted. For this reason, this latter approach is commonly known as quantization. Even if more complex to be implemented, in many cases it outperforms the HRBB approach and, for this reason, it was chosen over HRBB in OpenCAL.

The `OpenCAL` quantization feature[1] is implemented by considering a dynamic array of active cells, A of the size of the grid-space, which is initially empty. An array of flags, F, having the same dimension of the computational domain is also considered, which is initially set to `CAL_FALSE` in each position. Eventually, an integer variable, *size*, initially set to zero, is used to evaluate the new dimension of A per effect of the add/remove operations. In order to add a cell to A, the `calAddActiveCell{2D|3D}()` function can be used, which sets the flag value to `CAL_TRUE` in the corresponding position of the array F and increases *size* by one. Similarly, the `calAddActiveCellX{2D|3D}()` function adds a neighbouring cell to A. Eventually, the `calRemoveActiveCell{2D|3D}()` function can be used to remove a cell from A, by contextually modifying the corresponding flag in F and decrementing *size* by one. When an add/remove stage is completed, e.g. after the execution of an elementary process, and the (scattered) array F is well defined, the set A must be updated. The update process deletes the current set A, allocates a new set of active cells of dimension *size*, and applies a straightforward serial stream compaction algorithm by processing the entire array of flags F, as shown in Figure 5.4. As evident, the algorithm has a $O(n)$ computational complexity, being n the number of cells of the cellular space R. In this way, only the loop updating A occurs on the whole computational domain (since the F array must be fully checked). In fact, when A is not empty, it is processed instead of the whole computational domain and both elementary processes computation and substates updating take place on the active cells. As substates, even A is implicitly updated at the end of each elementary process.

Using the quantization optimization is quite straightforward. Firstly, it must be enabled at model object definition time by means of the `calCADef{2D|3D}()` function. Subsequently, the `calAddActiveCell[X]{2D|3D}()` function can be used to mark the central cell and its neighbors (if the `X` version of the function is considered) to be added to A, while the `calRemoveActiveCell{2D|3D}()` to mark the central cell to be removed. All these functions essentially write a 8-bit long Boolean value to F and, for this reason, there is not any risk to obtain a corrupted value, even in the case of parallel execution (i.e. in the case two threads/work-items attempt to store their own value to the same memory word at the same time). Even in the case of OpenCAL-CL, if the same instruction is executed by more than one work-item (even belonging to different work-groups) to the same location in global memory (where F is stored), the access is serialized and at least one access is guarantied (even if which the actual thread performing the operation is undefined - cf. e.g. [134]). Eventually, in case of explicit update scheme, the `calUpdateActiveCells{2D|3D}` function must be explicitly invoked to update A after each add/remove phase is complete.

Note that, since the API allows to modify the neighboring cells activation state, the quantization optimization can give rise to race conditions. Nevertheless, to avoid them it is sufficient to keep the add and remove phases disjoint, i.e. performed by different elementary processes. In fact, if the same elementary process could both add and remove cells to/from A, two different (central) cells could update the same (neighboring) cell to different activation states, and the resulting value in F before the stream compaction execution would depend on the application order of the elementary process to the cells.

There are other ways to implementation the quantization strategy s.t. less time or space space is used and some of them have also been investigated as for instance a strategy that allows to use space proportional to the number of active cells, which consist of a dynamic

1 The term quantization has been chosen to refer to this optimization as when enabled, it treats the grid space as a group of quanta of computation instead of as a single massive computation.

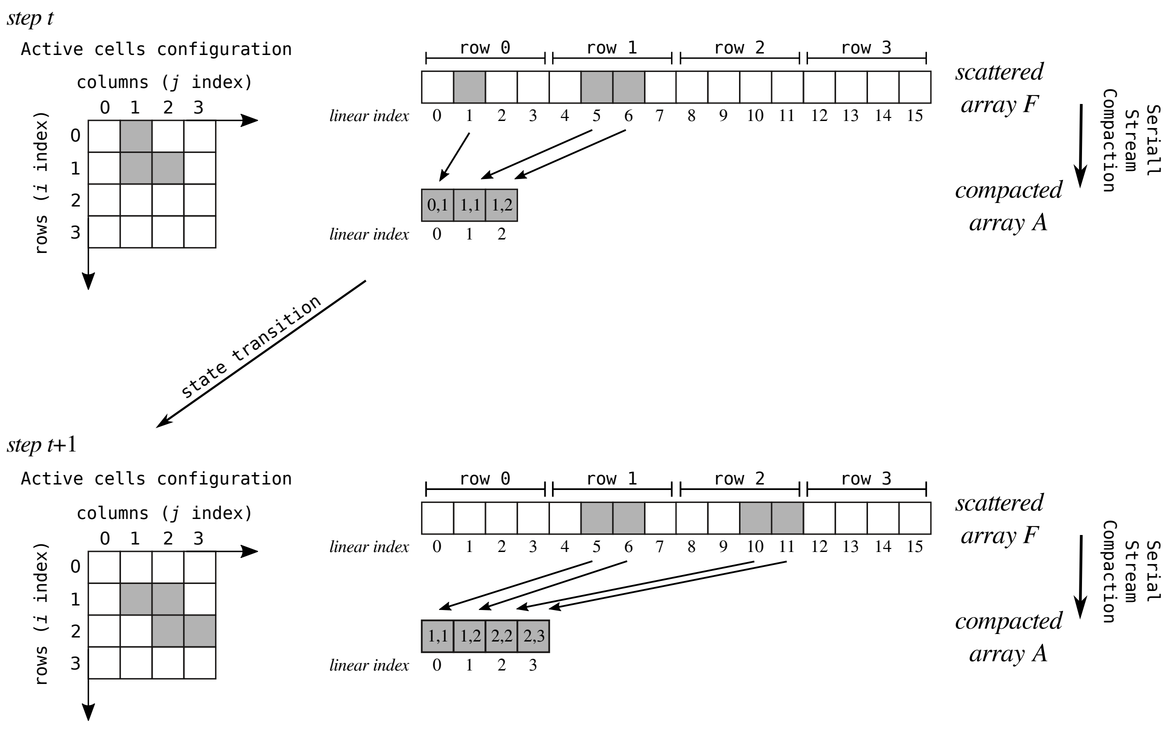


Figure 5.4: An example of application of the serial stream compaction algorithm adopted to update the set A of active cells in the serial implementation of OpenCAL. In the example, active cells are represented in gray within a two-dimensional 4x4 matrix of flags, implemented as a linearized array, F. The stream compaction algorithm simply processes F and produces the compacted array A as output, containing the coordinates of the active cells. The global state transition is therefore limited to the cell in A, and a new configuration of the system obtained. The process is therefore repeated at the next computational step.

list of active cells. The one shipped with this release of OpenCAL guarantees uniform performance on all platforms and is for this reason adopted. future versions will feature multiple implementation from which the user can choose from.

In order to perform a simulation, a pointer to `CALRun{2D|3D}` must be declared and defined by means of the `calRunDef{2D|3D}()` function. The simulation object stores a pointer to the OpenCAL model to be run, a step counter, the initial and final computational steps, an enumeral of type `CALUpdateMode`, which specifies the substates update policy (if implicit or explicit), and registers a set of four optional callback functions. These latter, described below, take a pointer to an OpenCAL model as argument and do not return any value, with the exception of the termination function, which returns a `CALbyte` (`CAL_TRUE` if the termination criterion is satisfied, `CAL_FALSE` in the other case). The optional callback functions are:

- `init()`: It can be used to set the initial conditions of an OpenCAL computational model. It can be registered to the simulation object by using the `calRunAddInitFunc{2D|3D}()` function. If defined, the `init()` function is executed once before the simulation loop.
- `globalTransition()`: It can be used to redefine the execution order of the registered elementary processes and to perform selective substate updating.

The `calApplyElementaryProcess2D|3D()` function can be used within the registered callback to apply a registered elementary process to each cell of the computational domain, while the `calUpdateSubstate{2D|3D}{b|i|r}` to update a registered substate. The `globalTransition()` function also allows to call functions that can perform global operations over the computational domain, e.g. reductions. It can be registered to the simulation object by using the `calRunAddGlobalTransitionFunc{2D|3D}()` function. If defined, the `globalTransition()` function overrides, i.e. is applied instead of, the implicit global transition function.

- `steering()`: It can be used to perform global operations, e.g. reductions, at the end of each computational step, that is after that all the elementary processes have been applied. Predefined reductions allow to compute global minimum, maximum, sum, product, as well as logical and bit-wise AND, OR and NOT operations on the registered substates. A steering can be registered to the simulation object by using the `calRunAddSteeringFunc{2D|3D}()` function. If defined, the `steering()` function is applied at the end of each computational step.
- `stopCondition()`: It can be used to define a stopping criterion for the simulation. Note that, if the last computational step was set to `CAL_RUN_LOOP`, the `stopCondition()` callback is mandatory to stop the simulation. It returns `CAL_TRUE` if the termination criterion is satisfied, `CAL_FALSE` in the other case. The termination callback can be registered by using the `calRunAddStopConditionFunc{2D|3D}()` function. If defined, the `stopCondition()` function is executed at the end of each computational step.

The simulation process can be executed by means of the `calRun{2D|3D}` function. Algorithm 1 outlines the `OpenCAL` implicit simulation process, which takes place in the case the enumeral `CAL_UPDATE_IMPLICIT` was used as last argument for the `calRunDef{2D|3D}()` function. The `init()` function, if defined, is called first and then active cells (if quantization is enabled) and substates are updated. Moreover, the `step` counter and the `halt` variable, that is used to check the simulation termination condition, are set to the initial step and to `CAL_FALSE`, respectively. The main simulation loop follows, which applies elementary processes in the order they were registered to each cell of the computational domain. After the execution of each elementary process, active cells and substates are updated. If defined, the `steering()` global function is therefore called and active cells and substates updated. The `stopCondition()` function is also called and the step counter increased. The simulation loop continues while the `halt` variable, whose value is set by the `stopCondition()` function, is `CAL_FALSE` or the final step of computation is met.

Explicit update can be set by using the `CAL_UPDATE_EXPLICIT` enumeral as last argument of the `calRunDef{2D|3D}()` function. In this case, the global transition function must be overridden and both active cells and substates explicitly updated. Eventually, instead of the `calRun{2D|3D}()`, which executes the whole simulation process, the `calRunCAStep{2D|3D}()` function can be used to apply a single step of the global transition function, including steering and stop condition checking. In this case, the initialization function must be called explicitly, as well as the simulation counter increased.

5.3.3 `OpenCAL` *Conway's Game of Life*

As a first illustrative example, we here present the `OpenCAL` implementation of the Turing complete Conway's Game of Life [135], one of the most simple, yet powerful examples of

Algorithm 1: OpenCAL main implicit simulation process.

```
1  init()                                       // Call the init() global function
2  if $quantization$ then
3      update ($A$)                              // Update the array of active cells
4  forall $q \in Q$ do
5      update ($q$)                              // Update the substate $q$
6  $step \leftarrow initial\_step$
7  $halt \leftarrow false$
8  while $\neg halt \wedge (step \leq final\_step \vee final\_step = CAL\_RUN\_LOOP)$ do
9      forall $e$ of $\sigma$ do
10         forall $(A \neq \emptyset \wedge i \in A) \vee i \in R$ do
11             $e(i)$                            // Apply the elementary process $e$ to the cell $i$
12         if $quantization$ then
13             update ($A$)                      // Update the array of active cells
14         forall $q \in Q$ do
15             update ($q$)                      // Update the substate $q$
16     steering()                               // Call the steering() global function
17     if $quantization$ then
18         update ($A$)                          // Update the array of active cells
19     forall $q \in Q$ do
20         update ($q$)                          // Update the substate $q$
21     $halt \leftarrow$stopCondition()          // Check the stop condition
22     $step \leftarrow step + 1$
23 return
```

CA, devised by mathematician John Horton Conway in 1970. See Section 2.5 at page 14 for a formal definition of *The Game of Life* and a description of its governing rules.

The program in Listing 5.1 provides a complete OpenCAL implementation of Game of Life in just few lines of code, by defining both the CA model and the simulation object, needed to let the CA evolve step by step.

```c
#include <OpenCAL/cal2D.h>
#include <OpenCAL/cal2DIO.h>
#include <OpenCAL/cal2DRun.h>
#include <stdlib.h>

struct CALModel2D* life;
struct CALSubstate2Di* Q;
struct CALRun2D* life_simulation;

void lifeTransitionFunction(struct CALModel2D* life, int i, int j)
{
  int sum = 0, n;
  for (n=1; n<life->sizeof_X; n++)
    sum += calGetX2Di(life, Q, i, j, n);

  if ((sum == 3) || (sum == 2 && calGet2Di(life, Q, i, j) == 1))
17     calSet2Di(life, Q, i, j, 1);
18   else
19     calSet2Di(life, Q, i, j, 0);
20 }
21
22 int main()
23 {
24   life = calCADef2D(8, 16, CAL_MOORE_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL, CAL_NO_OPT);
25   life_simulation = calRunDef2D(life, 1, 1, CAL_UPDATE_IMPLICIT);
26
27   Q = calAddSubstate2Di(life);
28
29   calAddElementaryProcess2D(life, lifeTransitionFunction);
30
31   calInitSubstate2Di(life, Q, 0);
32   calInit2Di(life, Q, 0, 2, 1);
33   calInit2Di(life, Q, 1, 0, 1);
34   calInit2Di(life, Q, 1, 2, 1);
35   calInit2Di(life, Q, 2, 1, 1);
36   calInit2Di(life, Q, 2, 2, 1);
37
38   calSaveSubstate2Di(life, Q, "./life_0000.txt");
39
40   calRun2D(life_simulation);
41
42   calSaveSubstate2Di(life, Q, "./life_LAST.txt");
43
44   calRunFinalize2D(life_simulation);
45   calFinalize2D(life);
46
47   return 0;
48 }
```

LISTING 5.1: An `OpenCAL` implementation of the Conway's Game of Life.

`OpenCAL` header files are included at lines 1-3. Specifically, `cal2D.h` allows to define 2D CA and substates, `cal2DRun.h` the simulation object, while `cal2DIO.h` provides some basic I/O functions. The CA object is declared at line 6, while lines 7 and 8 declare a substate and a simulation object, respectively. Objects declared at lines 6-8 are defined later in the `main` function. In particular, the `life` CA object is defined at line 24 by the `calCADef2D()` function. The first 2 parameters define the dimensions of the computational domain (in terms of number of rows and columns, respectively), while the third the Moore neighborhood. Furthermore, the fourth parameter sets a toroidal topology for the cellular space, while the last switches the active cells optimization to off.

The CA simulation object is defined at line 25 by the `calRunDef2D()` function, where the first parameter is a pointer to the `life` CA object, while the second and third parameters specify the initial and last simulation steps, respectively. Eventually, the last parameter sets the update policy to implicit.

Line 27 allocates memory and registers the integer-based Q substate to the CA by means of the `calAddSubstate2Di()` function, while line 29 registers an elementary process by means of the `calAddElementaryProcess2D()` function. Here, the `lifeTransitionFunction` parameter represents the name of a developer-defined function implementing the transition function rules (cf. lines 10-20). Within the elementary process, the `calGet[X]2Di()` and `calSet2Di()` functions are used for reading and updating purposes, respectively. The `calInitSubstate2Di()` function at line 31 sets the whole Q substate to the value 0 (for both

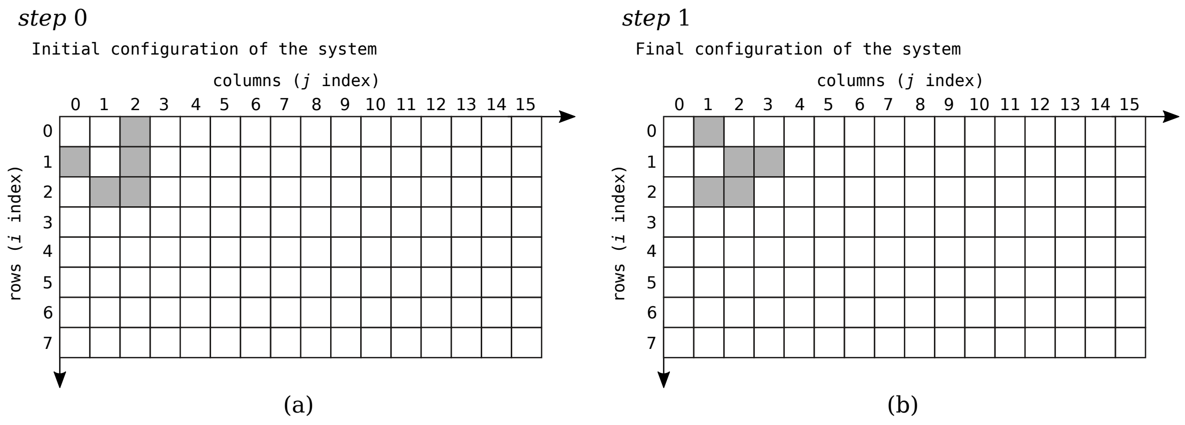


FIGURE 5.5: Graphical representation of one computational step of the Game of Life, showing the (a) initial and (b) final configurations of the system. Alive cells are represented in grey, dead cells in white.

the current and next layers). Lines 32-36 define a so called *glider* pattern (cf. Figure 5.5a) by means of the `calInit2Di()` function. The `calSaveSubstate2Di()` function at line 38 saves the *Q* substate to file, while the `calRun2D()` function at line 40 enters the simulation process (actually, only one computational step in this example), and returns to the `main` function when the simulation is terminated. The `calSaveSubstate2Di()` is called again at line 42 to save the new (last) configuration of the CA, while the last two functions at lines 44 and 45 release memory previously automatically allocated by `OpenCAL` for the CA, substates (actually, only *Q* in this case) and simulation object. The `return` statement at line 47 ends the program.

Figure 5.5 show a graphical representation of the initial and final configurations of Game of Life, respectively, as implemented in Listing 5.1. As expected, the glider initially defined has evolved into the new correct configuration.

5.4 THE `opencal` `openmp`-BASED PARALLEL IMPLEMENTATION

In this section we present `OpenCAL-OMP`, the `OpenMP`-based parallel implementation of `OpenCAL`, which allows for seamless parallel execution on shared memory computing systems in a SIMD fashion. For brevity, we only present and discuss differences with respect to `OpenCAL` in terms of API and internal algorithms. Since one of the main goal is to obtain transparent parallelism, the `OpenCAL-OMP` API has been designed to differ as less as possible to the serial one, leading to the adoption of the same naming conventions, interface, and programming model.

Double layer substates were also considered in `OpenCAL-OMP`, which permitted a straightforward lock-free `OpenMP` parallelization. In fact no race conditions can occur, since the current layer is accessed in read mode, and the update phase access to the next layer is limited to the memory location associated with the central cell. As a consequence, elementary processes and substates updating loops, as well as global reduction operations, were parallelized by simply considering lock-free `OpenMP` pool of threads, as shown in Figure 5.6. In the example, a pool of three threads were used to partition the computational domain in three subregions, which were therefore processed in parallel both during the application of an elementary process and the subsequent substates updating. Note that, in the

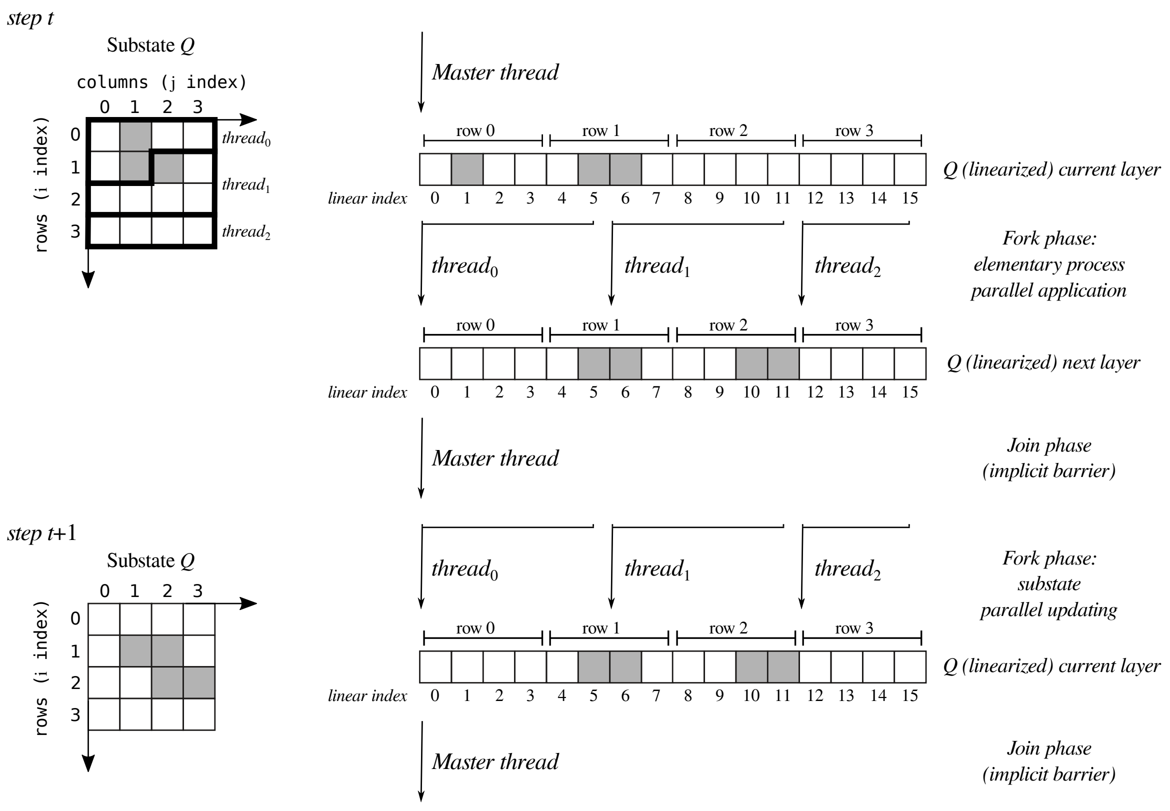

step t
Substate Q
columns (j index)
0 1 2 3
rows (i index)
0
1
2
3
thread0
thread1
thread2
Master thread
row 0
row 1
row 2
row 3
Q (linearized) current layer
linear index
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Fork phase: elementary process parallel application
Q (linearized) next layer
Join phase (implicit barrier)
step t+1
Fork phase: substate parallel updating

FIGURE 5.6: An example of OpenCAL-OMP parallel application of an elementary process to a substate Q and its subsequent parallel updating. The computational domain is initially partitioned by means of a pool of three threads (fork phase). These latter concurrently apply the elementary process by reading state values from the current layer and by updating new values to the next one. At the end of the elementary process application, threads implicitly synchronize by joining into the master one (join phase), and the parallel update phase starts. As before, a pool of threads concurrently copies the next layer into the current one and the new configuration of Q is obtained. A join phase eventually occurs, which ensures data consistency before the application of another elementary process.

case only a subset of cells are actually involved in computation, as in the example, load unbalance conditions can occur. In fact, the third thread is wasted, since it only applies the elementary process on a subset of stationary cells. A dynamic OpenMP scheduling is adopted in this case to mitigate the unbalance among chunks. As regards the quantization feature, it is still based on the dynamic array of active cells A, containing the coordinates of non stationary cells, and on the array of flags F, having the same dimension of the computational domain, used to register the cells activation state during the application of the transition function. F is initially set to CAL_FALSE in each position and, each time a cell has to be added to/removed from A, the corresponding position in F is updated by a CAL_TRUE/CAL_FALSE value. At the end of the add/remove stage, a lock-free parallel stream compaction is executed on the resulting scattered array F to obtain the new set of active cells A. For this purpose, F is partitioned over the N running threads. Each of them builds a private subset, A_p $(p = 0, 1, \dots N - 1)$, of cells to be added to/removed from A, by contextually evaluating its relative size, $size_p$. Eventually, the actual $size$ of A is obtained as

$size = \sum_{p=0}^{N-1} size_p$, and the subsets A_p assembled together to form the new set A, as shown in Figure 5.7. As for the case of the OpenCAL serial stream compaction, it is evident that also in this case the algorithm has a $O(n)$ computational efficiency, where n is the number of cells of the cellular space. Note that, being both the CAL_TRUE and CAL_FASE 8-bit long enumerals, no inconsistent values can be obtained even in the case more than one thread accesses the same location in F simultaneously and, therefore, a lock-free updating policy was also here considered for the quantization optimization. However, as for OpenCAL, the add and remove stages updating the values in F have to be performed separately, e.g. by two different elementary processes, in order to avoid possible race conditions. In fact, if the same elementary process could both add and remove cells (depending for instance on the current state of the central cell), two different threads applying the elementary process to two different cells could update the same (neighbouring) cell to different activation states. In this case, the resulting value in F would depend on which thread writes the value for last, giving rise to a possible logical error. Eventually, note that the quantization feature is able to optimally distribute the computational load over the running threads, since stationary cells are simply excluded by the computation (cf. Figure 5.7). A more efficient static OpenMP scheduling is therefore here adopted instead of the dynamic one.

5.4.1 *OpenCAL-OMP implementation of the Game of Life*

Conway's Game of Life can be straightforwardly implemented in parallel by using OpenCAL-OMP. The source code is almost identical to the one in Listing 5.1, with the exception of the first three lines, where the OpenCAL-OMP header files are included instead of the OpenCAL ones. For this purpose, it is sufficient to change the headers parent directory from OpenCAL to OpenCAL-OMP. For instance, the OpenCAL/cal2D.h header is replaced by OpenCAL-OMP/cal2D.h. The remaining lines of code are unchanged and therefore source code is omitted. As for the case of the OpenCAL implementation, Figure 5.5 shows the initial and final configuration of the system.

5.5 THE OpenCAL OPENCL-BASED PARALLEL IMPLEMENTATION

In this section, we present the OpenCL-based parallel implementation of OpenCAL, which allows for the parallel execution on both shared memory computing systems and many-core acceleration devices in a SIMD fashion, by highlighting the differences with respect to the serial and OpenMP implementations of OpenCAL in terms of API and internal algorithms.

The API is very similar to the serial one and adopts the same programming model and naming conventions, with the exception that the calcl, CALCL and CALCL_ prefixes are adopted for functions, data types and constants, respectively. The main difference with respect to OpenCAL and OpenCAL-OMP is that an OpenCAL-CL application is subdivided in two parts, one running on the CPU, the other in parallel on a compliant device. The host application defines the host-side computational model, registers the required substates to it, while elementary processes and other global function are implemented as OpenCL kernels and registered to a device-side computational model. When a device-side model is defined, data registered to the host-side model is implicitly copied to the compliant device global memory, transparently to the user. Within kernels, however, the user can transfer data to the local memory and then update the global memory when the local operations are

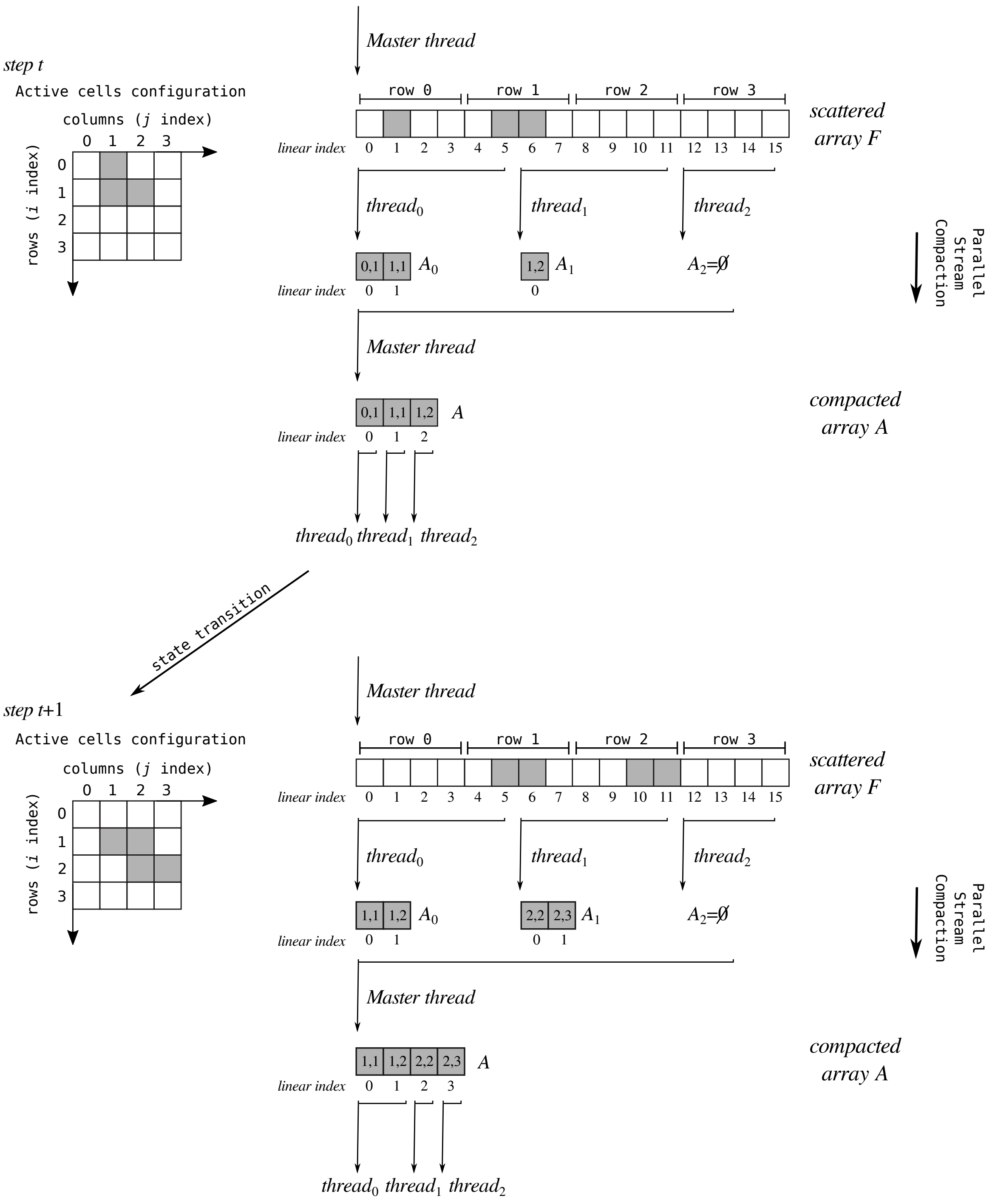


FIGURE 5.7: An example of application of the parallel stream compaction algorithm used in OpenCAL-OMP to update the set A of active cells. Active cells are represented in gray within a two-dimensional 4x4 matrix of flags, implemented as a linearized array, F. The parallel stream compaction algorithm processes F by means of a lock-free pool of threads, resulting in the A_p $(p = 0, 1, 2)$ partial arrays of active cells. These latter are eventually assembled together by the master thread, resulting in the compacted array A. A new pool of threads therefore applies the state transition function in parallel on A with an optimal load balancing, and a new configuration of the system is obtained. The process is therefore repeated at the next computational step.

complete, for better performance. Data stored in global memory is therefore copied back to the host at the end of the simulation process. This latter if equivalent to that of OpenCAL and `OpenCAL-OMP` (cf. Algorithm 1), with the exception that kernels are executed device-side and the simulation process can not be currently, in this first `OpenCAL` release, made explicit, because the *explicit* API is not entirely correctly implemented.

Grid of work-items can be two- or three-dimensional, depending on the dimension of computational model, if 2D or 3D, respectively. In the case the quantization feature is exploited, a one-dimensional grid is considered. The number of work-items is evaluated for each model dimension by preliminary querying OpenCL for the (device-dependent) preferred work-group size multiple (i.e. the warp/wavefront size in NVIDIA/AMD GPUs), w_s, and therefore by considering the smallest multiple of w_s which is greater than or equal to the model dimension. For instance, if $w_s = 32$ and the first dimension of the domain is 2000, the number of work-items in that dimension will be 2016, i.e. the first multiple of 32 which is greater than or equal to 2000, thus resulting in 16 redundant work-items. However, since redundant work-items do not map any cell of the computational domain, they immediately terminate their execution. Moreover, according to OpenCL, work-items are grouped in workgroups. The choice of the number of workgroups to be considered, and therefore the workgroup size, depends on the device architecture and is automatically determined by default, transparently to the user. These choices should allow to not waste resources and also permits the user to ignore low-level hardware details. In any case, the `calclSetWorkGroupDimensions{2D|3D}()` function can be used to explicitly set the workgroup size.

Double layer substates are also considered in OpenCAL-CL. That is, as for the serial and OpenMP-based versions of OpenCAL, the current layer is used for reading the states of the neighboring cells, while the next for updating the new value for the central one. A grid of OpenCL work-items can therefore be defined, each one executing the transition function on a different cell of the computational domain, independently to each other. In fact, no race conditions can occur, since access to the current layer is read-only, and each work-item updates a different memory location of the next layer. Being the data stored in global memory, work-items can be easily synchronized after the execution of each elementary process, since a global barrier is implicitly defined at the end of each kernel execution. The kernel-based transition function is therefore applied by considering the one-thread/one-cell parallel execution policy and, at the end of each computational step, substates are updated device side, avoiding the need to perform time consuming data transfer between host and device.

The quantization feature is also supported in OpenCAL-CL, where coordinates of the active cells are stored in the array A that, differently from the serial and OpenMP-based implementations, is static and has the same dimension of the computational domain. A variable, *size*, initially set to zero, is also considered to identify the actual number of active cells. A static array of flags, F, is also considered, which has the same dimension of the computational domain. It is initially set to `CAL_FALSE` in each position, to mark all cells as inactive. The add/remove operations, performed by work-items at the init stage or during the execution of the transition function, update the array of flags F as in the previous discussed implementations. In case of concurrent access to the same location of F, data integrity is guarantied only under the condition work-items execute the same instruction (e.g. all of them write the `CAL_TRUE` value). In fact, if the same instruction is executed by more than one work-item (even belonging to different workgroups) to the same location in global memory where F is stored, the access is serialized and at least one access is guarantied (even if which actual thread performs the operation is undefined - cf. e.g. [134]). Moreover, the same considerations discussed for the `OpenCAL` and OpenCAL-OMP implementations of the active cells optimization, even hold for the `OpenCAL-CL` one. As a consequence, to both guaranty data integrity and to avoid possible logical errors, the add and remove stages have to be performed separately, e.g. by two different elementary processes. At the end of each add or remove stage, cells to be considered active are marked by the `CAL_TRUE` value in the scattered array F. In order to update A, a parallel stream compaction algorithm is considered. F is partitioned in N chunks, being N the number of considered work-items, and the following three stages applied:

1. Each work-item counts the number of active cells in its chunk of the array F. As a result, an array S is obtained, where s_p ($p = 0, 1, \ldots, N-1$) is the number of the active cells counted by the p^{th} work-item;

2. A prefix-sum algorithm is executed to both evaluate the total number of active cells, *size*, and a further array, O, where o_p ($p = 0, 1, \ldots, N-1$) represents the offset to be considered by the p^{th} work-item from which it must start entering in A the coordinates of the cells that are marked as active in its chunk of the array F.

3. Each work-item processes its chunk of F and enters in A the coordinates of the cells marked as active, starting from the offset o_p ($p = 0, 1, \ldots, N-1$) computed in the previous stage.

For illustrative purposes, the above three stages are graphically represented in Figure 5.9. At step t, the configuration of the cells actually involved in the computation are represented in gray in a two-dimensional 4x4 matrix, corresponding to the linearized scattered array F. The parallel stream compaction algorithm processes the scattered array F to obtain the compacted array A, containing the coordinates of the three active cells in its first three positions. A two-block grid with two threads per block is considered in the example, which adopt the one thread/one active cell execution policy. Note that, at step t a total of four work-items are considered to process a set of three active cells. In this case, the thread that does not match any active cell immediately terminates. Among the parallel stream compaction stages, the prefix-sum algorithm at stage two, which evaluates the array O of offsets, is crucial for the overall efficiency of the algorithm. As shown in Figure 5.8, the algorithm takes as input the array S of partial sums and uses it to initialize the array O of offsets. This latter is considered as a balanced tree, where its elements are the nodes. In the first phase, the tree is crossed from the leaves to the root (up-sweep phase) by calculating, for each level, the partial sums of the nodes of the previous level (by a parallel reduction pattern). Here the total number of work-items is set to $N/2$, which means that a thread will elaborate two elements of the array. The total number of active cells, necessary to set the *size* variable, is obtained at the end of the up-sweep phase in the root node (i.e. in the last element of the array). In the second phase, the tree is traversed from the root to the leaves (down-sweep phase). At each iteration, each node sets the value of the right child to the sum of its value and the value of the left child. In addition, each node sets the value of its left child to its value. At the end of this phase, the array O contains at each location the position from which each work-item can write the coordinates of the active cells in its chunk of F to the array A. As known, the parallel prefix-sum algorithm has a $O(log_2 N)$ computational efficiency, and is well suitable for parallel execution, resulting

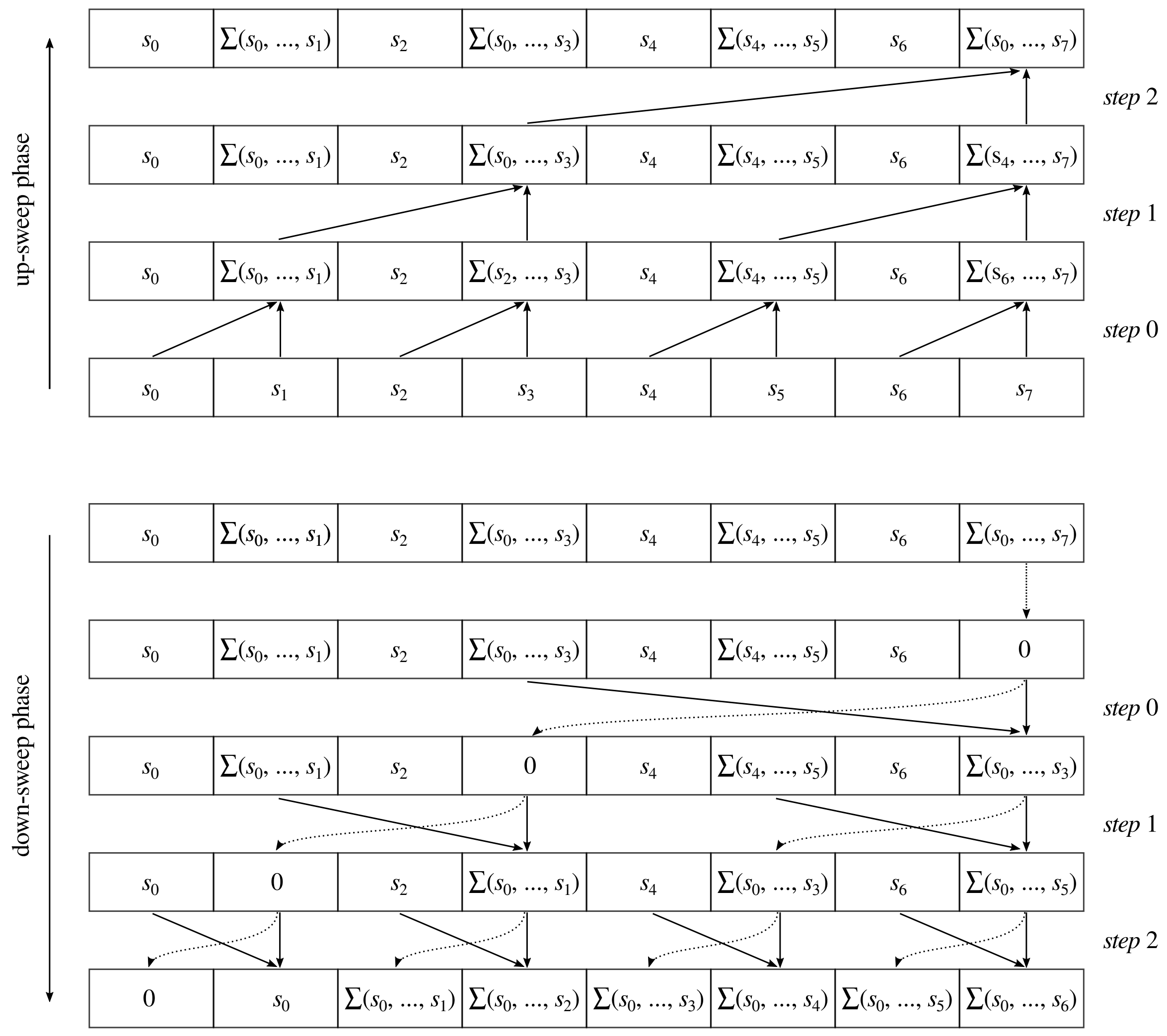


FIGURE 5.8: An example of application of the prefix-sum algorithm used to evaluate the array O of offsets, needed to build the array A of active cells. In the up-sweep phase, the array O is initialized to the values of the array S, where s_p is the number of cells to be added to A by the p^{th} wor-item. The array O is therefore efficiently processed by a set of work-items and, at the end of the phase, the last cell of the array contains the total number of active cells to be considered for the next computational step. The down-sweep phase follows, in which dotted arrows are used to set the pointed cell of the array O to 0, while continuous arrows to evaluate the sum of the source cells and then to write the computed value to the pointed position, as in the previous phase. Offsets are eventually obtained at the end of the down-sweep phase. The first work item will therefore start adding S_0 cells to A from the index 0, the second adding S_1 cells starting from the offset S_0, the third adding S_2 cells from the offset $S_0 + S_1$, up to the latter, which will add S_7 cells starting from the offset $S_0 + S_1 + \ldots + S_6$.

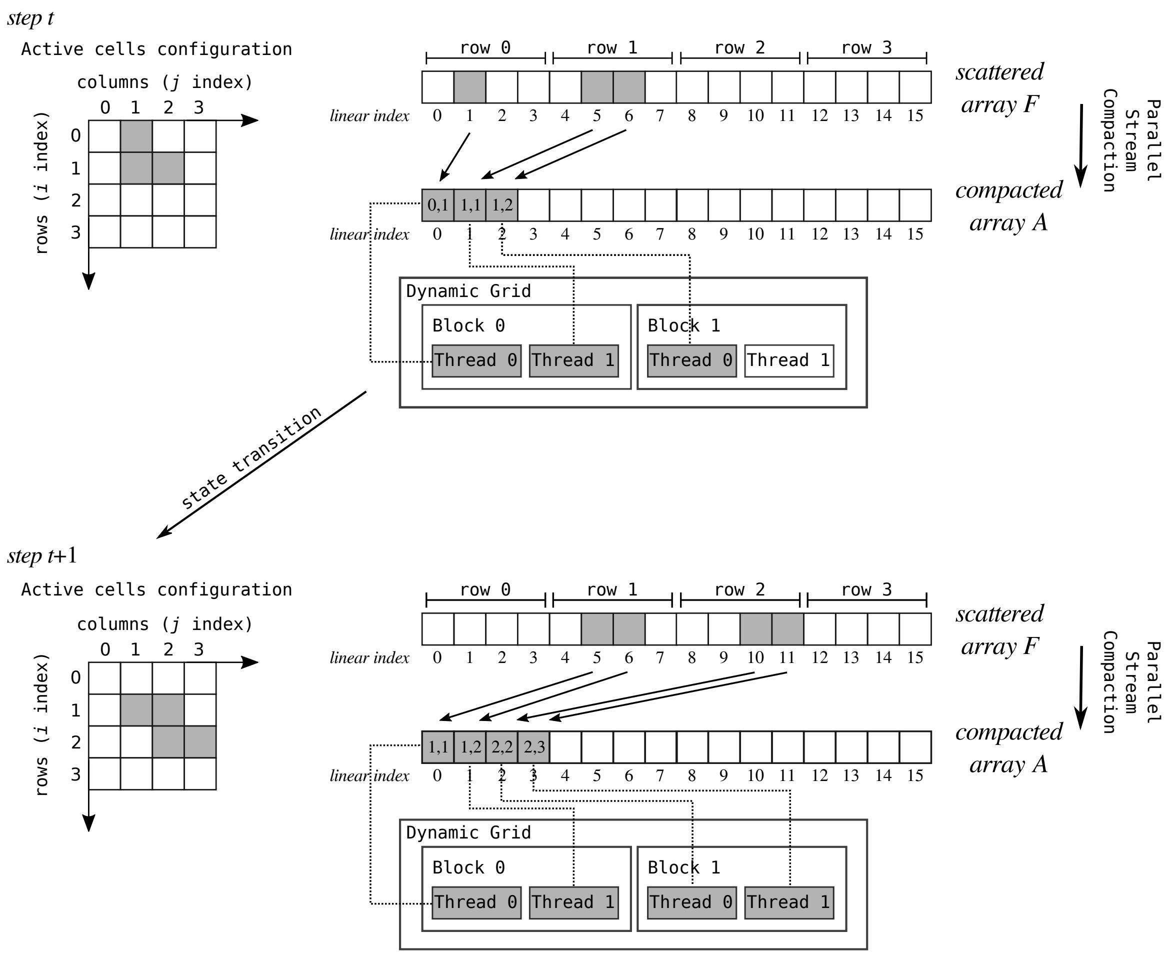


Figure 5.9: An example of application of the OpenCAL-CL parallel stream compaction algorithm. Active cells are represented in gray within a two-dimensional 4x4 matrix of flags, implemented as a linearized array, *F*. The parallel stream compaction algorithm processes *F* and produces the compacted array *A* as output, containing the coordinates of the active cells in its first part. A grid of work-items therefore processes data by adopting the one thread/one active cell policy. The process is therefore repeated at the next computational step.

in a fast solution for the second stage of the parallel stream compaction algorithm. The overall computational complexity of the OpenCAL-CL parallel stream compaction, as for the OpenCAL and OpenCAL-OMP implementations, is however $O(n)$, being n the number of cells of the computational domain. Due to its efficiency, the up-sweep phase of the parallel prefix-sum algorithm is also applied to implement parallel global reductions on substates (cf. Section 5.3).

5.5.1 *OpenCAL-CL device-side kernels*

Differently to OpenCL, where a kernel can have no parameters, OpenCAL-CL ones must have at least the `__CALCL_MODEL_2D` meta-parameter (cf. line 8 of Listing 5.2). Actually, this is a macro-like C object, defining a list of pre-fixed typed parameters, needed to let the kernel access the model data device-side. Moreover, the `calcl[Active]ThreadCheck{2D|3D}()`

function must be called within any elementary process implemented as a kernel to prevent the execution of a number of threads out of the computational domain (cf. line 11) or of the set of active cells. The `calclGlobal{Row|Column|Slice}()` function (cf. lines 15-16) have also to be used to get the global cell coordinates, which here do not appear in the kernel parameter list. The `calclGet[X]{2D|3D}{b|i|r}()` and `calclSet{2D|3D}{b|i|r}()` functions are used for reading and updating purposes. Differently to their host-side counterparts, they take the `MODEL_{2D|3D}` macro-like C meta-object, which implicitly defines a list of required prefixed parameters (cf. e.g. line 23). Moreover, substates are referred by means of numerical handles (cf. the second parameter of the `calclSet2Dr()` function at line 23), which have to be previously defined in the kernel (cf. line 6). The criterion to be adopted is very simple: handles are zero-based IDs, i.e. zero is used to refer the first substate registered to the host-side model, one to refer the second substate, and so on. Different zero-based handles must be defined for different typed substates.

```
 1 #include <OpenCAL-CL/calcl2D.h>
 2 
 3 // Define substates handles
 4 // omissis ...
 5 #define Z 4
 6 #define H 5
 7 
 8 __kernel void calcl_kernel_example(__CALCL_MODEL_2D)
 9 {
10 // Prevent the execution of more threads than the CA dimension
11 calclThreadCheck2D();
12 
13 // omissis ...
14 
15 // Get the cell coordinates back
16 CALint i = calclGlobalRow();
17 CALint j = calclGlobalColumn();
18 
19 // omissis ...
20 
21 // Set a new value for the substate
22 // whose handle is defined by H.
23 // Please, note the usage of the
24 // MODEL_2D macro-like object
25 calclSet2Dr(MODEL_2D, H, i, j, h_next);
26 
27 // omissis ...
28 }
```

Listing 5.2: Example of `OpenCAL-CL` kernel.

5.5.2 *OpenCAL-CL host-side Programming*

An `OpenCAL-CL` host application is typically subdivided in the following parts:

- Definition of the host-side computational model;
- Selection of the OpenCL compliant device;
- Kernels reading and program generation;

- Definition of the device-side computational model (which also embeds simulation facilities);
- Kernels enqueuing;
- Simulation execution (on the previously selected compliant device).

The `OpenCAL-CL` host-side model definition does not differ from the serial implementation of OpenCAL. Indeed, in Listing 5.3, a two-dimensional host-side model object is declared by using the `CALModel{2D|3D}` data type (line 4), and then initialized by means of the `calCADef{2D|3D}()` function (line 11). Note that the `calcl{2D|3D}.h` header file is included at line 1. This, in turn, includes the `cal{2D|3D}.h` header, so that it is possible to use `OpenCAL` data types and functions from an `OpenCAL-CL` host application.

```
1  #include <OpenCAL-CL/calcl2D.h>
2
3  // omissis ...
4
5  struct CALModel2D* hostCA;
6
7  // omissis ...
8
9  int main(int argc, char** argv)
10 {
11 // omissis ...
12
13 hostCA = calCADef2D(ROWS, COLS, CAL_VON_NEUMANN_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL,
       CAL_OPT_ACTIVE_CELLS);
14
15 // omissis ...
16
17 }
```

LISTING 5.3: An example of `OpenCAL-CL` host-side application.

`OpenCAL-CL` provides the `CALCLManager` structure which, together with other utility functions, considerably simplifies platform, device, and context management with respect to the native OpenCL API. Listing 5.4 shows how to select a compliant device in OpenCAL-CL. Line 7 declares a pointer to the `CALCLManager` `OpenCAL-CL` data type, and initializes it through the `calclCreateManager()` function. This object, `calcl_manager`, is then used as parameter for the `calclInitializePlatforms()` function (line 10), which fills the object with the platforms available on the machine. Line 13 calls the `calclInitializeDevices()` function, that initializes the available devices, while line 20 selects one of them for kernel execution. Specifically, an object of type `CALCLdevice` is declared and initialized by the function `calclGetDevice()`. This latter takes a pointer to a `CALCLManager` object as first parameter, while the second and third parameters specify the platform and device to be selected, respectively. Since both platforms and devices are identified by a 0-based index, statement at line 20 selects the first device belonging to the first platform (e.g., a GTX 980 belonging to the Nvidia CUDA platform). If system platforms and devices are unknown, the `calclGetPlatformAndDeviceFromStdIn()` function can be used alternatively to `calclGetDevice()`. It prints all the available platforms and devices to standard output and permits for their interactive selection from standard input. Eventually, line 23 creates an OpenCL context, based on the device previously selected. For this purpose, an object of

`CALCLcontext` type is declared and defined by means of the `calclCreateContext()` function.

```
#include <OpenCAL-CL/calcl2D.h>

// omissis ...

int main (int argc, char** argv)
{
// Initilize a pointer to the CALCLManager structure
CALCLManager* calcl_manager = calclCreateManager();

// get all available platforms
calclInitializePlatforms(calcl_manager);

// Initialize the devices
calclInitializeDevices(calcl_manager);

// Uncomment if platforms and devices are unknown
//calclGetPlatformAndDeviceFromStdIn();

// get the first device on the first platform
// this call is unnecessary if
// calclGetPlatformsAndDeviceFromStandardInput() is used
CALCLdevice device = calclGetDevice(calcl_manager, 0, 0);

// create a context CALCLcontext
context = calclCreateContext(&device);

// omissis ...
}
```

Listing 5.4: Example of `OpenCAL-CL` access to platform and devices.

Once the compliant device has been selected and functions to be executed in parallel implemented as kernels, these latter can automatically be read and compiled through the `calclLoadProgram{2D|3D}()` function. It takes both the context and device, and also the paths to the directory containing the user defined kernels and related headers (if any), and returns an OpenCL program. All the files in the kernel source directory are automatically loaded. Note that, since kernel headers are optional, the last parameter can be `NULL`.

The device-side counterpart of the host-side computational model can be declared as a pointer to `CALCLModel{2D|3D}` and, beside managing all the host model components device-side, also embeds simulation execution facilities. In this manner, the user can continue to deal with only two main structures, as in the serial and OpenMP-based implementations. The `calclCADef{2D|3D}()` function can be used to initialize the device-side model object. It takes a pointer to an host-side `CALModel{2D|3D}` serial model, an OpenCL context, an OpenCL program, and a compliant device as parameters.

Kernels can be extracted from an OpenCL program by means of the `calclGetKernelFromProgram()` function and then registered to the device-side model by means of the `calclAddElementaryProcess{2D|3D}()` function, which adds the kernel to the execution queue, in a transparent manner to the user. The function takes a pointer to a host, a device model and also a pointer to an OpenCL kernel. Global functions can also be registered to the device model. For instance, the `calclAddInitFunc{2D|3D}()` function registers a global initialization kernel, the `calclAddSteeringFunc{2D|3D}()` function registers a global kernel to be executed at the end of each computational step, while the `calclAddStopConditionFunc{2D|3D}()` function registers a global stop condition kernel callback.

The `calclRun{2D|3D}()` function runs the simulation by executing all the defined kernels on the selected compliant device. The first two parameters are pointers to a device and host models, respectively, while the last two are the initial and final step for the simulation execution. If the last parameter is set to `CAL_RUN_LOOP`, the simulation never ends. In this case, a stop condition criterion must defined by registering a termination kernel callback to halt the simulation.

5.5.3 *OpenCAL-CL implementation of the Game of Life*

According to OpenCAL-CL, the Game of Life implementation here described is subdivided in a device- and an host-side part. The device-side kernel implementing the Conway's Game of Life transition function is shown in Listing 5.5. The `calcl2D.h` is included at line 1, and a numeric handle defined at line 3 to refer the Q substate device-side. The transition rules are implemented as an elementary process kernel at lines 5-23. In particular, line 7 checks for redundant work-items, while lines 9-10 get the indices corresponding to the integer coordinates of the cell the kernel is going to process. Similarly, line 12 retrieves the neighborhood size by means of the `calclGetNeighborhoodSize()` function. Eventually, lines 16-22 implement the transition rules by using the `calclGet[X]2Di()` and `calclSet2Di()` functions for reading and updating purposes, respectively.

```
 1 #include <OpenCAL-CL/calcl2D.h>
 2 
 3 #define DEVICE_Q 0
 4 
 5 __kernel void lifeTransitionFunction(__CALCL_MODEL_2D)
 6 {
 7   calclThreadCheck2D();
 8 
 9   int i = calclGlobalRow();
10   int j = calclGlobalColumn();
11 
12   CALint sizeof_X = calclGetNeighborhoodSize();
13 
14   int sum = 0, n;
15 
16   for (n=1; n<sizeof_X; n++)
17     sum += calclGetX2Di(MODEL_2D, DEVICE_Q, i, j, n);
18 
19   if ((sum == 3) || (sum == 2 && calclGet2Di(MODEL_2D, DEVICE_Q, i, j) == 1))
20     calclSet2Di(MODEL_2D, DEVICE_Q, i, j, 1);
21   else
22     calclSet2Di(MODEL_2D, DEVICE_Q, i, j, 0);
23 }
```

LISTING 5.5: The `OpenCAL-CL` kernel implementing the Conway's Game of Life elementary process.

The host-side application, running on the CPU and controlling the computation on the compliant device (e.g. a GPU), is shown in Listing 5.8.1. The `calcl2D.h` header file is included, together with the `OpenCAL cal2DIO.h` header for I/O operations at lines 1-2. The kernel path is defined at line 4, while the name of the kernel considered in this example is defined at line 5. Lines 6-8 define the IDs of the OpenCL platform and device to be considered. For the sake of simplicity, in this example the first device belonging to the first platform is considered. Lines 12-15 are needed to select the compliant device and to create an OpenCL context. These statements widely simplify the device management and can be considered as a kind of template to be used in each OpenCAL-CL application. Line 16 reads kernels (actually, just one in this example) from file (contained in the directory specified at line 4), compile and groups them into an OpenCL program, to be used later to extract kernels for execution. As in the serial implementation of the Game of Life, the `host_CA` host-side object is defined at line 18 and the *Q* substate declared at line 19. This latter is therefore registered to the host-side CA at line 21. Eventually, the substate is set to zero in each cell and a glider is defined at lines 23-28. Line 30 defines the `device_CA` device-side object. The `calclCADef2D()` function initializes the device-side CA, by performing data transfer from host to device, in a transparent way to the user. Note that this function implicitly registers each host-side defined substate to the device object. In order to register an elementary process to the device-side CA, the elementary process (which actually is an OpenCL kernel) must be preliminarily extracted from the previously compiled program. This operation is done at line 32 by means of the `calclGetKernelFromProgram()`. It returns an OpenCL kernel, which is subsequently registered to the device CA by means of the `calclAddElementaryProcess2D()` function at line 33. Lines 35 and 39 are used to save the CA state before and after simulation execution, respectively. The CA simulation, for one step, is executed by means of the `calclRun2D()` function at line 37. In this example, the only defined elementary process is executed in parallel on the compliant device in a transparently way to the user. Eventually, lines 41-43 perform memory deallocation for the previously defined objects. The return statement at line 45 terminates the program.

```
1  #include <OpenCAL-CL/calcl2D.h>
2  #include <OpenCAL/cal2DIO.h>
3
4  #define KERNEL_SRC "./kernel"
5  #define KERNEL_LIFE_TRANSITION_FUNCTION "lifeTransitionFunction"
6  #define PLATFORM_NUM 0
7  #define DEVICE_NUM 0
8  #define DEVICE_Q 0
9
10 int main()
11 {
12   struct CALCLDeviceManager* calcl_device_manager = calclCreateManager();
13   calclPrintPlatformsAndDevices(calcl_device_manager);
14   CALCLdevice device = calclGetDevice(calcl_device_manager, PLATFORM_NUM,
       DEVICE_NUM);
15   CALCLcontext context = calclCreateContext(&device);
16   CALCLprogram program = calclLoadProgram2D(context, device, KERNEL_SRC, NULL);
17
18   struct CALModel2D* host_CA = calCADef2D(8, 16, CAL_MOORE_NEIGHBORHOOD_2D,
       CAL_SPACE_TOROIDAL, CAL_NO_OPT);
19   struct CALSubstate2Di* Q;
20
21   Q = calAddSubstate2Di(host_CA);
22
23   calInitSubstate2Di(host_CA, Q, 0);
24   calInit2Di(host_CA, Q, 0, 2, 1);
25   calInit2Di(host_CA, Q, 1, 0, 1);
26   calInit2Di(host_CA, Q, 1, 2, 1);
27   calInit2Di(host_CA, Q, 2, 1, 1);
28   calInit2Di(host_CA, Q, 2, 2, 1);
```

```

    struct CALCLModel2D* device_CA = calclCADef2D(host_CA, context, program, device
        );

    CALCLkernel kernel_life_transition_function = calclGetKernelFromProgram(&
        program, KERNEL_LIFE_TRANSITION_FUNCTION);
    calclAddElementaryProcess2D(device_CA, &kernel_life_transition_function);

    calSaveSubstate2Di(host_CA, Q, "./life_0000.txt");

    calclRun2D(device_CA, 1, 1);

    calSaveSubstate2Di(host_CA, Q, "./life_LAST.txt");

    calclFinalizeManager(calcl_device_manager);
    calclFinalize2D(device_CA);
    calFinalize2D(host_CA);

    return 0;
}
```

LISTING 5.6: An `OpenCAL-CL` host-side implementation of the Conway's Game of Life.

As for the case of the `OpenCAL` implementation of the Game of Life, Figure 5.5 shows the initial and final configuration of the system.

5.6 THE $SciddicaT_{naive}$ EXAMPLE OF APPLICATION

In this section we show the OpenCAL, `OpenCAL-OMP` and OpenCAL-CL implementations of a more complex example, namely the $SciddicaT_{naive}$ fluid-flow XCA computational model. This is a simplified version of the XCA model described in [133] and is able to simulate the dynamics of a generic non-inertial fluid-type flow over a real topographic surface. The model is formally defined and key implementation sections reported and commented. Eventually, the application to the simulation of a real case study, namely the *1992 Tessina (Italy)* landslide, is shown.

5.6.1 *The $SciddicaT_{naive}$ Formal Definition*

The $SciddicaT_{naive}$ fluid-flow XCA computational model is formally defined as:

$$SciddicaT_{naive} =< R, X, Q, P, \sigma >$$

where:

- R is the set of points, with integer coordinates, which defines the two-dimensional domain over which the phenomenon evolves. The generic cell in R is individuated by means of a couple of integer coordinates (i, j), where $0 \leq i < i_{max}$ and $0 \leq j < j_{max}$. The first coordinate, i, represents the row, while the second, j, the column. The cell at coordinates $(0, 0)$ is located at the top-left corner of the computational grid.
- $X = \{(0,0), (-1,0), (0,-1), (0,1), (1,0)\}$ is the von Neumann neighborhood relation (cf. Figure 5.2a), a geometrical pattern which identifies the cells influencing the state

transition of the central cell. The neighborhood of the generic cell of coordinate (i,j) is given by

$$N(X,(i,j)) =$$
$$= \{(i,j)+(0,0),(i,j)+(-1,0),(i,j)+(0,-1),(i,j)+(0,1),(i,j)+(1,0)\} =$$
$$= \{(i,j),(i-1,j),(i,j-1),(i,j+1),(i+1,j)\}$$

Here, a subscript operator can be used to index cells belonging to the neighborhood. Let $|X|$ be the number of elements in X, and $n \in \mathbb{N}$, $0 \leq n < |X|$; the notation

$$N(X,(i,j),n)$$

represents the n^{th} neighborhood of the cell (i,j).

- Q is the set of cell states. It is subdivided in the following substates:
  - Q_z is the set of values representing the topographic altitude (i.e. elevation a.s.l.);
  - Q_h is the set of values representing the thickness of the fluid;
  - Q_o^4 are the sets of values representing the outflows from the central cell to the neighboring ones.

  The Cartesian product of the substates defines the overall set of states Q:

  $$Q = Q_z \times Q_h \times Q_o^4$$

  so that the cell state is specified by the following sextuplet:

  $$q = (q_z, q_h, q_{o_0}, q_{o_1}, q_{o_2}, q_{o_3})$$

  In particular, q_{o_0} represents the outflows from the central cell towards the neighbor 1, q_{o_1} the outflow towards the neighbor 2, and so on.

- P is the set of parameters ruling the model dynamics where:
  - p_ϵ is the parameter which specifies the minimum thickness of fluid below which it cannot leave the cell due to the effect of adherence;
  - p_r is the relaxation rate parameter, which essentially is an outflow damping factor.

- $\sigma : Q^5 \rightarrow Q$ is the deterministic cell transition function. It is composed by two elementary processes, listed below in the same order they are applied:
  - $\sigma_1 : (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \rightarrow Q_o^4$ determines the outflows from the central cell to the neighboring ones by applying the *minimization algorithm of the differences* [5]. In its simplest form, here considered, the algorithm is able to lead the neighborhood to the hydrostatic equilibrium in a single computational step. In the σ_1 elementary process, a preliminary control avoids outflows computation for those cells in which the amount of fluid is smaller or equal to p_ϵ, acting as a simplification of the adherence effect. If $f(0,m)$ $(m = 0,\ldots,3)$ represent the outgoing flows towards the 4 adjacent cells, as computed by the minimization algorithm, the resulting outflows are given by $q_o(0,m) = f(0,m) \cdot p_r$, being $p_r \in]0,1]$ a

relaxation factor considered to damp outflows in order to obtain a smoother convergence to the global equilibrium of the system. The Q_o^4 substates are accordingly updated.

- $\sigma_2 : Q_h \times (Q_o^4)^4 \rightarrow Q_h$ determines the value of debris thickness inside the cell by considering mass exchange in the cell neighborhood: $h^{t+1}(0) = h^t(0) + \sum_{m=0}^{3}(q_o(0,m) - q_o(m,0))$. Here, $h^t(0)$ and $h^{t+1}(0)$ are the mass thickness inside the cell at t and $t+1$ computational step, respectively, while $q_o(m,0)$ represents the inflow from the $n = (m+1)^{th}$ neighboring cell. The Q_h substate is accordingly updated to account for the mass balance within the cell.

5.6.2 *The $SciddicaT_{naive}$ OpenCAL and OpenCAL-OMP implementations*

According to OpenCAL/ and OpenCAL-OMP, the XCA programming model, substates, parameters and the simulation object can be declared as:

```
struct CALModel2D* sciddicaT;

struct sciddicaTSubstates {
struct CALSubstate2Dr *z;
struct CALSubstate2Dr *h;
struct CALSubstate2Dr *f[NUMBER_OF_OUTFLOWS];
} Q;

struct sciddicaTParameters {
CALParameterr epsilon;
CALParameterr r;
} P;

struct CALRun2D* sciddicaT_simulation;
```

where `NUMBER_OF_OUTFLOWS` is equal to 4, since the σ_1 elementary process computes 4 outflows towards the adjacent cells belonging to the von Neumann neighborhood. Note that the real-based substates and parameters are grouped in C structures for convenience.

The model object is defined through the `calCADef2D()` function, by specifying the dimensions of the computational domain, the neighborhood pattern, the boundary topology and the optimization to be used:

```
sciddicaT = calCADef2D(ROWS,
  COLS,
  CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
  CAL_SPACE_TOROIDAL,
  CAL_NO_OPT
);
```

A toroidal domain is here defined even if, by considering the application described below, a bounded one could be equivalently adopted. Moreover, according to the model definition, the active cell optimization is not employed.

Substates are therefore registered to the XCA model by means of the `calAddSubstate2Dr()` function:

```
Q.z = calAddSubstate2Dr(sciddicaT);
Q.h = calAddSubstate2Dr(sciddicaT);
for (i = 0; i < NUMBER_OF_OUTFLOWS; i++)
  Q.f[i] = calAddSubstate2Dr(sciddicaT);
```

as well as elementary processes through the `calAddElementaryProcess2D()` function:

```
calAddElementaryProcess2D(sciddicaT, flowsComputation);
calAddElementaryProcess2D(sciddicaT, widthUpdate);
```

where `flowComputation()` and `widthUpdate()` are callback functions implementing the σ_1 and σ_2 elementary processes, respectively. A snippet of the σ_1 elementary process is shown below:

```
void flowsComputation(struct CALModel2D* model, int i, int j)
{
CALreal f[NUMBER_OF_OUTFLOWS];

if (calGet2Dr(model, Q.h, i, j) <= P.epsilon)
  return;

computeMinimizingOutflows(f);

for (n=1; n<model->sizeof_X; n++)
  calSet2Dr(model, Q.f[n-1], i, j, f[n]*P.r);
}
```

The `calGet2Dr()` function is here used to retrieve the thickness of the fluid in the central cell, comparing its value with the p_ϵ parameter for evaluating the adherence condition. In case the thickness overcomes the adherence threshold, the `computeMinimizingOutflows()` function is called, which applies the minimization algorithm of the differences (whose implementation is here omitted) and returns the array of outgoing flows, `f`. Such flows are eventually damped by considering the p_r factor and the resulting values used to update the corresponding substates by means of the `calSet2Dr()` function. The implementation of the σ_2 elementary processes is also shown below:

```
void widthUpdate(struct CALModel2D* model, int i, int j )
{
CALint n;
CALreal h_next = calGet2Dr(model, Q.h, i, j);

for(n=1; n<sciddicaT->sizeof_X; n++)
  h_next += calGetX2Dr(model, Q.f[NUMBER_OF_OUTFLOWS-n], i, j, n) - calGet2Dr(model, Q.f[n-1], i
      , j);

  calSet2Dr(model, Q.h, i, j, h_next);
}
```

Here, the `calGetX2Dr()` function is used to get the incoming flows from the neighbouring cells which, together with the outgoing flows, are used to evaluate the mass balance and the resulting value to update the Q_h substate.

The above OpenCAL/OpenCAL-OMP code snippets completely define the XCA model according to the $SciddicaT_{naive}$ formal definition. In order to perform a simulation, a simulation object must be defined, by means of the `calRunDef2D()` function, permitting the model evolve step by step:

```
sciddicaT_simulation = calRunDef2D(sciddicaT,
  1,
  STEPS,
  CAL_UPDATE_IMPLICIT
);
```

The function takes the computational model to be carried out as first parameter, the initial and final computational step and the substates update policy. In this case, the implicit scheme is considered, which demands the substates updating entirely to `OpenCAL/OpenCAL-OMP`, transparently to the user. The simulation object therefore registers two global callbacks for initialization and steering purposes through the `calRunAddInitFunc2D()` and `calRunAddSteeringFunc2D()` functions, respectively:

```
calRunAddInitFunc2D(sciddicaT_simulation, simulationInit);
calRunAddSteeringFunc2D(sciddicaT_simulation, steering);
```

In particular, the `simulationInit()` callback, listed below, is executed once before the simulation loop (cf. Algorithm 1) to define the initial condition of the system. By referring the application described in the next section, the callback simply subtracts the thickness of the mass, represented by the Q_h substate, from the surface over which it will flow down, represented by the Q_z substate:

```
void simulationInit(struct CALModel2D* sciddicaT)
{
CALreal z, h;
CALint i, j;

for (i=0; i<sciddicaT->rows; i++)
  for (j=0; j<sciddicaT->columns; j++){
    h = calGet2Dr(sciddicaT, Q.h, i, j);
    z = calGet2Dr(sciddicaT, Q.z, i, j);

    calSet2Dr(sciddicaT, Q.z, i, j, z-h);
  }
}
```

Similarly, the steering callback, shown below, is executed at the end of each computational step (cf. Algorithm 1) and is here simply used to reset the outflow substates, as needed, by means of the `calInitSubstate2Dr()` function:

```
void steering(struct CALModel2D* sciddicaT)
{
for(n=0; n<NUMBER_OF_OUTFLOWS; n++)
  calInitSubstate2Dr(sciddicaT, Q.f[n], 0);
}
```

Eventually, the simulation is performed for `STEPS` computational steps by simply calling the `calRun2D()` function:

```
calRun2D(sciddicaT_simulation);
```

5.6.3 *The SciddicaT*$_{naive}$ *OpenCAL-CL implementation*

According to OpenCL, the `OpenCAL-CL` implementation of *SciddicaT*$_{naive}$ differs from the one described in the previous section as it is subdivided in two parts, the first running on the CPU, the other on an OpenCL compliant device.

The computational model (here called `host_CA`), substates and parameters are defined host-side exactly as before, as well as the initial conditions of the system. A device-side model is therefore declared as an object of type `CALCLModel2D`:

```
struct CALCLModel2D* device_CA;
```

and then defined by means of the `calclCADef2D()` function:

```
device_CA = calclCADef2D(host_CA, context, program, device);
```

which takes a pointer to the host-side model as first parameter, needed for host-device data transfer purposes, while the other ones are the OpenCL context, program and device, respectively.

The code to be executed device-side is defined as kernels. In particular, the `OpenCAL-CL` implementation of $SciddicaT_{naive}$ define both the σ_1 and σ_2 elementary processes and the steering global functions as kernels. The kernel defining the σ_1 elementary process is defined as:

```
__kernel void flowsComputation(__CALCL_MODEL_2D, __global CALParameterr* Pepsilon, __global
    CALParameterr* Pr )
{
calclThreadCheck2D();

int i = calclGlobalRow();
int j = calclGlobalColumn();

CALint sizeof_X = calclGetNeighborhoodSize();

CALreal f[5];

if (calclGet2Dr(MODEL_2D, H, i, j) <= *Pepsilon)
  return;

computeMinimizingOutflows(f);

for (n = 1; n < sizeof_X; n++)
  calclSet2Dr(MODEL_2D, n-1, i, j,f[n]*(*Pr));
}
```

Besides the mandatory `__CALCL_MODEL_2D` meta-parameter, the kernel takes two further parameters, corresponding to the $SciddicaT_{naive}$ model parameters. Here, the two additional parameters are located in the device global memory, even if they could be stored in the (fast) local memory by simply using the `__local` qualifier instead of the `__global` one. The function `calclThreadCheck2D()` is called first to check if the work-item executing the kernel actually maps a cell of the computational domain, where in such case cell coordinates are retrieved by means of the `calclGlobalRow()` and `calclGlobalColumn()` (kernel execution immediately terminates in case of wrong mapping). The `calclGet2Dr()` function is used to retrieve the thickness of the fluid in the central cell, referred by the `H` numerical handle, which is therefore compared with the p_ϵ parameter for evaluating the adherence condition. In case the thickness overcomes the adherence threshold, the `computeMinimizingOutflows()` function is called, which applies the minimization algorithm of the differences and returns the array of outgoing flows, `f`. Such flows are eventually damped by considering the p_r factor and the resulting values used to update the values of the corresponding substates by means of the `calclSet2Dr()` function. As regards the optional kernels parameters, they can be defined host-side by means of the `calclSetKernelArg2D()` function:

```
calclSetKernelArg2D(&flow_computation_kernel,
  0,
  sizeof(CALParameterr),
  &P.epsilon
);

calclSetKernelArg2D(&flow_computation_kernel,
  1,
  sizeof(CALParameterr),
  &P.r
);
```

It takes the kernel as first argument, a 0-based handle identifying the position of the parameter within the kernel parameter list (the `__CALCL_MODEL_2D` pseudo-parameter excluded), and both the size and the host-side parameter.

Similarly to σ_1, the σ_2 elementary process is defined by an `OpenCAL-CL` kernel as:

```
__kernel void widthUpdate(__CALCL_MODEL_2D) {
calclThreadCheck2D();

int i = calclGlobalRow();
int j = calclGlobalColumn();

CALreal h_next;
CALint n;

h_next = calclGet2Dr(MODEL_2D, H, i, j);

for (n = 1; n < calclGetNeighborhoodSize(); n++)
  h_next += calclGetX2Dr(MODEL_2D, NUMBER_OF_OUTFLOWS-n, i, j, n) - calclGet2Dr(MODEL_2D, n-1, i
      , j);

  calclSet2Dr(MODEL_2D, H, i, j, h_next);
}
```

Here, the `calclGetX2Dr()` function is used to get the incoming flows from the neighbouring cells that, together with the outgoing flows, are used to evaluate the mass balance and therefore to update the Q_h substate.

Once defined as kenels, elementary processes are added to the device-side model by means of the `calclAddElementaryProcess2D()` function:

```
calclAddElementaryProcess2D(device_CA, &flow_computation_kernel);
calclAddElementaryProcess2D(device_CA, &width_update_kernel);
```

Eventually, a steering kernel, whose implementation is here omitted, can be added by means of the `calclAddSteeringFunc2D()` function:

```
calclAddSteeringFunc2D(device_CA, &steering_kernel);
```

and the simulation performed for `STEPS` computational steps by means of the `calclRun2D()` function:

```
calclRun2D(device_CA, 1, STEPS);
```

5.6.4 *The SciddicaT$_{naive}$ Simulation of the Tessina Landslide*

Here we show the application of *SciddicaT$_{naive}$* to the simulation of the Tessina landslide [133] that occurred in Northern Italy in 1992. The real case developed in the Tessina valley between altitudes of 1220 m and 625 m a.s.l., with a total longitudinal extension of nearly 3 km and a maximum width of about 500 m. The landslide skimmed over the town of Funes and stretched downhill as far as the village of Lamosano.

The topographic surface over which the landslide developed was discretized as a DEM (Digital Elevation Model) of 410 rows per 294 columns, with square cells of 10 m side, for

SciddicaT parameter	Value	Unit
p_ϵ	0.001	m
p_r	0.5	1

TABLE 5.1: *SciddicaT* parameters considered for the simulation of the 1992 Tessina (Italy) landslide.

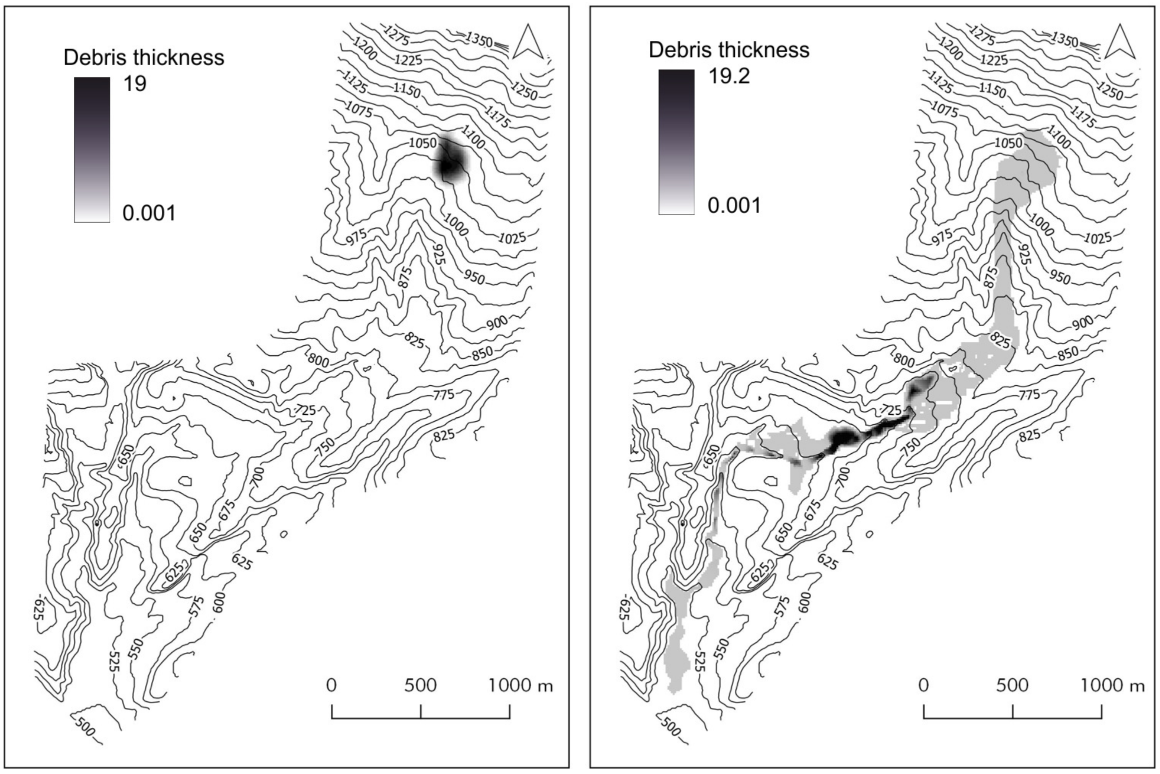


FIGURE 5.10: The *SciddicaT* simulation of the 1992 Tessina (Italy) landslide: (a) landslide source; (b) final landslide path. Topographic altitudes are represented in gray and vary between 1220 and 625 m a.s.l. Debris thickness is represented with colors ranging from red (lower values) to yellow (higher values).

a total of 102,540 cells. The landslide source, specifying the location and thickness of the detachment area, was also described by means of a raster map of the same dimensions.

The *SciddicaT*$_{naive}$ parameters were set to the values listed in table 5.1 and a total of 4000 computational steps considered in experiments. Simulation outcomes obtained by considering the serial and the two parallel implementations of *SciddicaT*$_{naive}$ did not differ, confirming the numerical correctness of the OpenMP- and OpenCL- based implementation of OpenCAL. Figure 5.10 show the initial and final configuration of the Tessina landslide, as obtained by the *SciddicaT*$_{naive}$ simulation. It is worth to note that, even if *SciddicaT*$_{naive}$ is a simplified model and parameters were preliminary evaluated, simulation outcome is in agreement with that of Avolio *et al.* [133].

5.7 THE *SciddicaT*$_{ac}$ EXAMPLE OF APPLICATION

In this section we show the OpenCAL, `OpenCAL-OMP` and OpenCAL-CL implementations of *SciddicaT*$_{ac}$, a computationally improved version of the *SciddicaT*$_{naive}$ fluid-flow XCA model which exploits the `OpenCAL` active cell optimization feature. The model is formally defined and key implementation sections reported and commented. The application to the simulation of the 1992 Tessina (Italy) landslide is here omitted since results are equivalent to those obtained by *SciddicaT*$_{naive}$.

5.7.1 *The SciddicaT*$_{ac}$ *Formal Definition*

In the case of a fluid-flow model, only cells involved in mass variation can be interested in a state change to the next computational step. On the basis of this simple observation, we can initialize the set of active cells to those cells containing mass. Moreover, if during the computation an outflow is evaluated from an active cell towards a neighboring non-active cell, this latter can be added to the set of active cells and then considered for subsequent state change. Similarly, if a given active cell looses a sufficient amount of debris, it can be eliminated from the set of active cells. In the case of *SciddicaT*$_{ac}$, this happens when its thickness becomes lower than or equal to the p_ϵ threshold.

In order to account for these processes, we have to slightly revise the formal definition of the XCA fluid-flow model, by adding the set of active cells, A. The optimized *SciddicaT*$_{ac}$ model is now defined as:

$$SciddicaT_{ac} =< R, A, X, Q, P, \sigma >$$

where $A \subseteq R$ is the set of active cells, while the other components are defined as in the formal definition of *SciddicaT*$_{naive}$. The transition function is now defined as:

$$\sigma : A \times Q^5 \to Q \times A$$

denoting that it is applied only to the cells in A and that it can add/remove active cells. More in detail, the σ_1 elementary process has to be modified, as it can activate new cells. Moreover, a new elementary process, σ_3, has to be added in order to remove cells that cannot produce outflows during the next computational step due to the fact that their debris thickness is negligible. The new sequence of elementary processes is listed below, in the same order they are applied.

- $\sigma_1 : A \times (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \to Q_o^4 \times A$ determines the outflows from the central cell to the neighboring ones, as in the case of *SciddicaT*$_{naive}$. In addition, each time an outflow is computed, the neighbor receiving the flow is added to the set of active cells.

- $\sigma_2 : A \times Q_h \times (Q_o^4)^4 \to Q_h$ determines the value of debris thickness inside the cell by considering mass exchange in the cell neighborhood. This elementary process does not differs with respect to that of the *SciddicaT*$_{naive}$ model.

- $\sigma_3 : A \times Q_h \times p_\epsilon \to A$ removes a cell from A if its debris thickness is lower than or equal to the p_ϵ threshold.

5.7.2 The *SciddicaT$_{ac}$* OpenCAL and OpenCAL-OMP implementations

Here we highlight the `OpenCAL` and `OpenCAL-OMP` differences between the *SciddicaT$_{ac}$* and *SciddicaT$_{naive}$* models. In particular, to properly exploit the active cells optimization, we have to change the definition of the CA object by using the `CAL_OPT_ACTIVE_CELLS` parameter in the model definition:

```
sciddicaT = calCADef2D (ROWS,
    COLS,
    CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
    CAL_SPACE_TOROIDAL,
    CAL_OPT_ACTIVE_CELLS
);
```

and by adding the σ_3 elementary process to the model, in addition to the σ_1 and σ_2 ones:

```
calAddElementaryProcess2D(sciddicaT, flowsComputation);
calAddElementaryProcess2D(sciddicaT, widthUpdate);
calAddElementaryProcess2D(sciddicaT, removeInactiveCells);
```

Preliminarly, at the `init` stage, cells belonging to the landslide source are set as active by means of the `calAddActiveCell2D()` function:

```
void simulationInit(struct CALModel2D* model){
CALreal z, h;
CALint i, j;

for (i=0; i<model->rows; i++)
  for (j=0; j<model->columns; j++){
    h = calGet2Dr(model, Q.h, i, j);
    z = calGet2Dr(model, Q.z, i, j);

    calSetCurrent2Dr(model, Q.z, i, j, z-h);
    calAddActiveCell2D(model, i, j);
  }
}
```

and, when a flow is computed from the central cell towards a neighbor by σ_1, the neighbor is added to A by means of the `calAddActiveCellX2D()` function:

```
void flowsComputation(struct CALModel2D* model, int i, int j )
{
// omissis...

for (n=1; n<model->sizeof_X; n++){
  calSet2Dr(model, Q.f[n-1], i, j, f[n]*P.r);
  calAddActiveCellX2D(sciddicaT, i, j, n);
  }
}
```

The σ_2 elementary process, here omitted, does not differ from the one of *SciddicaT$_{naive}$*, while σ_3 is new, and is responsible to remove a cell if its debris thickness is lower than or equal to the p_ϵ threshold:

```
void removeInactiveCells(struct CALModel2D* model, int i, int j){
if (calGet2Dr(model, Q.h, i, j) <= P.epsilon)
  calRemoveActiveCell2D(model, i, j);
}
```

No other changes with respect to the *SciddicaT$_{naive}$* implementation are needed.

Eventually, note that the active cells adding and remove stages were implemented by two different elementary processes, σ_1 and σ_3 respectively, according to the considerations discussed in Sections 5.3 and 5.4.

5.7.3 *The SciddicaT$_{ac}$ OpenCAL-CL implementation*

To properly exploit the active cells optimization in the `OpenCAL-CL` implementation of *SciddicaT$_{ac}$*, the host-side CA object, here called `host_CA`, was defined as in the OpenCAL/OpenCAL-OMP implementation, by using the `CAL_OPT_ACTIVE_CELLS` parameter in the definition. Moreover, the σ_3 elementary process was added to the device-side model, in addition to the σ_1 and σ_2 ones:

```
calclAddElementaryProcess2D(device_CA, &flow_computation_kernel);
calclAddElementaryProcess2D(device_CA, &width_update_kernel);
calclAddElementaryProcess2D(device_CA, &rm_active_cells_kernel);
```

As for the `OpenCAL` and `OpenCAL-OMP` implementations, cells belonging to the landslide source are set as active by means of the `calAddActiveCell2D()` at the init stage and when a flow is computed from the central cell towards a neighbor by σ_1, the neighbor is added to A by means of the `calclAddActiveCellX2D()` function:

```
__kernel void flowsComputation(__CALCL_MODEL_2D, __global CALParameterr* Pepsilon, __global
    CALParameterr* Pr )
{
// omissis ...

calclActiveThreadCheck2D();

// omissis ...

for (n = 1; n < calclGetNeighborhoodSize(); n++){
  calclSet2Dr(MODEL_2D, n-1, i, j,f[n]*(*Pr));
  calclAddActiveCellX2D(MODEL_2D, i, j, n);
  }
}
```

Here, note that the `calclActiveThreadCheck2D()` is called instead of the the `calclThreadCheck2D()` function to restrict the application of the elementary process to the cells actually belonging to A. The σ_2 elementary process, here omitted, does not differ from the one of *SciddicaT$_{naive}$* `OpenCAL-CL` implementation, while the σ_3 one is new, which is responsible to remove a cell if its debris thickness is lower than or equal to the p_ϵ threshold:

```
__kernel void removeInactiveCells(__CALCL_MODEL_2D, __global CALParameterr * Pepsilon )
{
// omissis ...
if (calclGet2Dr(MODEL_2D, H, i, j) <= *Pepsilon)
  calclRemoveActiveCell2D(MODEL_2D,i,j);
}
```

Eventually, as for the case of the OpenCAL/OpenCAL-OMP implementation, the active cells adding and remove stages were implemented by two different elementary processes, σ_1 and σ_3 respectively, according to the considerations discussed in Section 5.5. Moreover, it is worth to note that the number of active cells involved during the simulation of the Tessina landslide vary between 637, corresponding to the number of cells defining the landslide source, and 5,509. The resulting mean value of active cell processed per step is

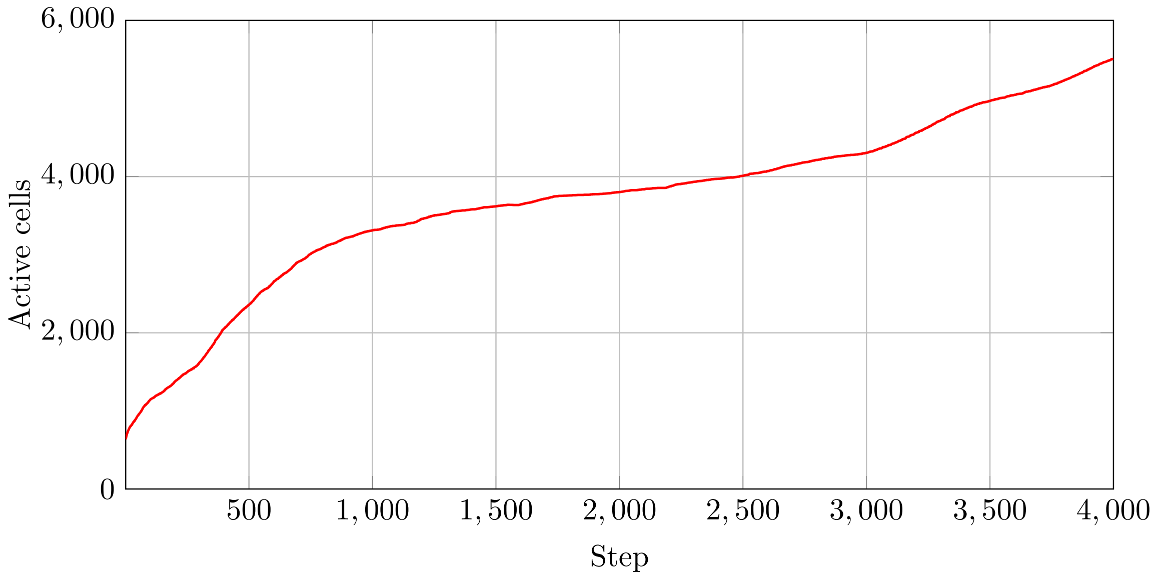


FIGURE 5.11: Number of active cells over time for the considered *SciddicaT*$_{ac}$ simulation model of the Tessina landslide shown in Figure 5.10.

3,277, corresponding to about the 3.2% of the whole computational domain. Figure 5.11 shows how the number of active cells varies when the *SciddicaT*$_{ac}$ computational step is increased.

5.8 THE $SciddicaT_{ac+esl}$ EXAMPLE OF APPLICATION

In this section we show the `OpenCAL` and `OpenCAL-OMP` implementations of the further computationally improved *SciddicaT*$_{ac+asl}$ fluid-flow XCA model, which exploit both the active cell optimization and the explicit simulation loop feature. The formal definition of the model does not differ from *SciddicaT*$_{ac}$, as well as the application to the simulation of the 1992 Tessina (Italy) landslide, and are not, therefore, reported in this section again. The key implementation sections are reported and commented in the following section.

5.8.1 *The SciddicaT*$_{ac+esl}$ *OpenCAL and OpenCAL-OMP implementations*

Here we highlight the OpenCAL/OpenCAL-OMP few implementation differences between the *SciddicaT*$_{ac+esl}$ and *SciddicaT*$_{ac}$ models. In particular, to properly exploit the explicit simulation loop feature, which is able to override the predefined OpenCAL/OpenCAL-OMP global transition function, by also allowing for the selective update of model substates, we have to change the definition of the CA simulation object by using the `CAL_UPDATE_EXPLICIT` parameter in its definition:

```
sciddicaT_simulation = calRunDef2D(sciddicaT,
  1,
  STEPS,
  CAL_UPDATE_EXPLICIT
);
```

and also register a callback function to the simulation object to implement the overridden global transition function:

```
calRunAddGlobalTransitionFunc2D(sciddicaT_simulation,
overridedGlobalTransitionFunction
);
```

The overridden transition function is defined as:

```
void overridedGlobalTransitionFunction( struct CALModel2D* model){
CALint i;

calApplyElementaryProcess2D(model, flowsComputation);
calUpdateActiveCells2D(model);
for (i=0; i<NUMBER_OF_OUTFLOWS; i++)
  calUpdateSubstate2Dr(model, Q.f[i]);

calApplyElementaryProcess2D(model, widthUpdate);
calUpdateSubstate2Dr(model, Q.h);

calApplyElementaryProcess2D(model, removeInactiveCells);
calUpdateActiveCells2D(model);
}
```

The `calApplyElementaryProcess2D()` is used to explicitly apply the elementary processes to the whole computational domain or, as in this case, to the set of active cells. Similarly, the active cells and substates updating must be explicitly performed. These operations can be performed by considering the `calUpdate2D()` function, which updates both the active cell structures and all the registered substates or, as reported in this example, by means of the `calUpdateActiveCells2D()` and `calUpdateSubstate2Dr()` functions, which allows to restrict the update phase to only data processed by the elementary processes. Indeed, the first elementary process only processes the active cells structure and the outflows substates, so that both the Q_z and Q_h substates do not need to be updated. Similarly, since the second elementary process only changes the debris thickness by evaluating the incoming and outcoming flows mass balance, only the Q_h substate is updated. Similarly, since the last elementary process simply removes cells that have become inactive from A, only the active cells structure is updated.

Note that, even if their implementations are here omitted, explicit updates have also to be performed after the execution of each global functions.

5.9 VALIDATION, AND PERFORMANCE RESULTS

In order to evaluate `OpenCAL` from a computational point of view, the different versions of *SciddicaT* presented in the previous Section were considered and the Tessina landslide taken into account as simulation reference case study for a first set of tests (*standard tests*). In particular, a total of ten benchmark simulations were executed for each of the nine *SciddicaT* implemented versions, and the speed-up evaluated with respect to the serial implementation of $SciddicaT_{naive}$, by considering the minimum recorded execution times. Furthermore, in order to better assess the impact of local memory usage in OpenCAL-CL, a further implementation based on $SciddicaT_{naive}$ was considered, namely $SciddicaT_{local}$. In this version, a 8×8 work-group size was considered and data, i.e., the substates values of the cells belonging to the neighborhood, transparently transferred from the global to the fast local device memory by using the `calclGlobaltoLocal[X]()` API function (cf. Section

Threads	$SciddicaT_{naive}$	$SciddicaT_{ac}$	$SciddicaT_{ac+esl}$
1	78.221	7.828	5.076
2	46.192	4.990	3.862
4	29.321	3.287	2.501
8	20.576	2.745	1.698
16	16.746	2.705	1.536

TABLE 5.2: Elapsed times (in seconds) registered for the simulation of the Tessina Landslide (cf. Figure 5.10) by different `OpenCAL` and `OpenCAL-OMP` versions of the *SciddicaT* fluid-flow model. The adopted CPU is an Intel Xeon 2.0GHz E5-2650.

5.3). In addition, due to the low transition function computational intensity of *SciddicaT* (i.e., the model is a more memory-bound rather than compute-bound application) and the data-set dimension, which are not adequate to take significant advantage of the adopted GPUs, two additional stress tests were carried out: the transition functions were fictitiously made computationally heavier by reapplying them, at each step, for a total of 200 times (*transition function stress tests*), and the landslide source replicated for a total of 100 times over a wider computational domain by considering a DEM of a total of 13,401,890 cells (*computational domain stress tests*). These latter tests were also considered to evaluate the preliminary OpenCAL-MPI versions of *SciddicaT*, both in terms of correctness and performance.

In all cases, `OpenCAL` and `OpenCAL-OMP` benchmarks were executed on a 8-core/16 threads Intel Xeon 2.0GHz E5-2650 CPU based workstation. One thread was considered for testing the different `OpenCAL` versions of *SciddicaT*, while 2, 4, 8 and 16 threads were employed for benchmark experiments concerning OpenCAL-OMP implementations. Moreover, two devices were adopted for testing the different versions of the `OpenCAL-CL` implementations of *SciddicaT*, namely a GTX 980 (Maxwell architecture) and a Tesla K40 (Kepler architecture) graphic processor. In particular, the former has 2048 CUDA cores, 4 GB global memory and 112 GB/s theoretical bandwidth communication for double precision data between CPU and GPU, while the latter device has 2880 cores, 12 GB global memory and 144 GB/s double precision high-bandwidth. Eventually, a Gigabit Ethernet interconnected dual node test system with a GTX 980 GPU per node, which is the configuration used for development purposes, was considered for preliminary evaluating the OpenCAL-MPI versions of *SciddicaT* .

5.9.1 *Standard Tests*

The speed-up and execution times of the Tessina landslide simulation related to the `OpenCAL` and `OpenCAL-OMP` different versions of *SciddicaT* are shown in Figure 5.12 and reported Table 5.2. Here, it is worth to note how the optimizations progressively introduced are effective and, as expected, execution times decrease steadily in all cases. In fact, even in the case of the serial OpenCAL-based implementations, the execution time decreases significantly from about 78 seconds, registered by $SciddicaT_{naive}$, to about 5 seconds, for the fully optimized $SciddicaT_{ac+esl}$ version. As expected, $SciddicaT_{ac+esl}$ is the version exhibiting the best performance, running about 51 times faster on 16 threads with respect to the reference simulation.

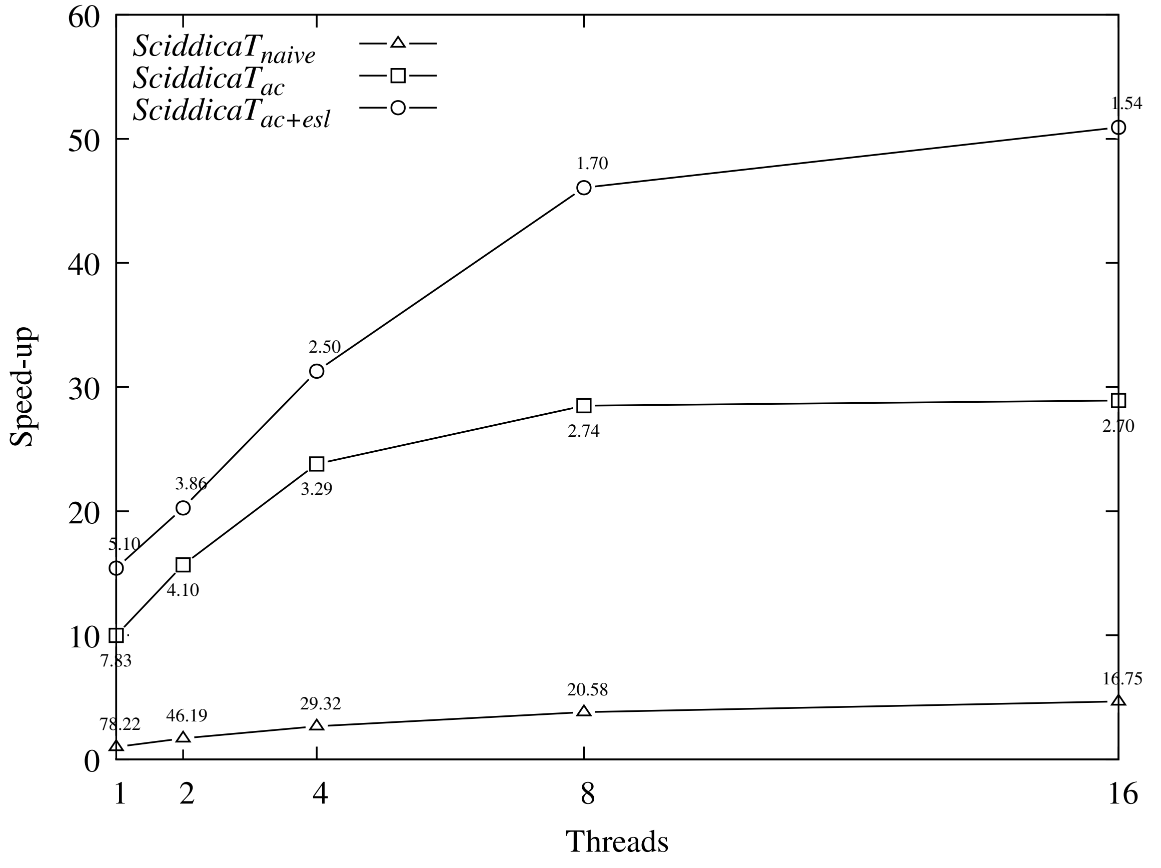


FIGURE 5.12: Speed-up obtained by the different OpenCAL-OMP versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown in correspondence of each speed-up vertex. The considered case study is the Tessina Landslide (cf. Figure 5.10). The adopted CPU was an Intel Xeon 2.0GHz E5-2650 CPU.

Device	$SciddicaT_{naive}$	$SciddicaT_{local}$	$SciddicaT_{ac}$
Tesla K40	11.108	10.605	3.960
GTX 980	5.063	5.453	2.981

TABLE 5.3: Elapsed times (in seconds) registered for the simulation of the Tessina Landslide (cf. Figure 5.10) by different `OpenCAL-CL` versions of the *SciddicaT* fluid-flow model. The adopted OpenCL compliant devices are a Nvidia Tesla K40 and a Nvidia GTX 980.

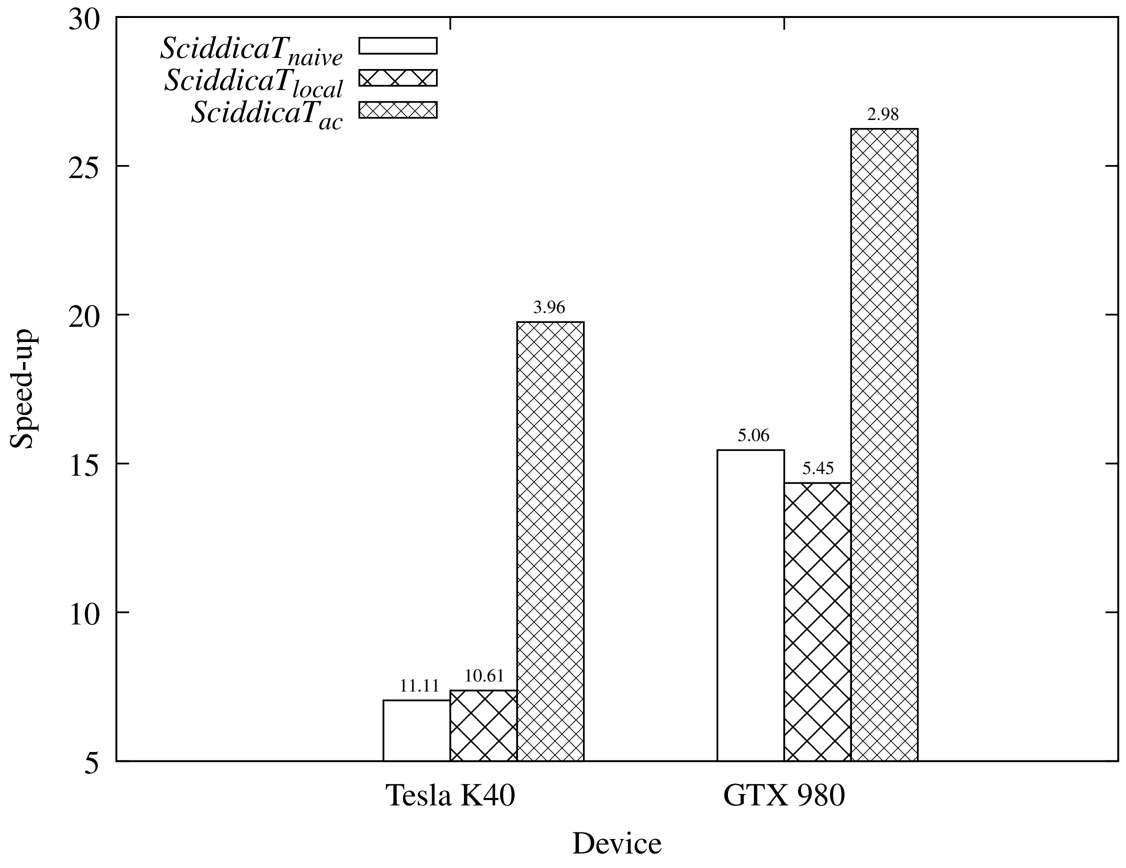


FIGURE 5.13: Speed-up obtained by the different OpenCAL-CL versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown on top of each speed-up bar. The considered case study is the Tessina Landslide (cf. Figure 5.10). The adopted OpenCL compliant devices were a Nvidia Tesla K40 and an Nvidia GTX 980.

The benchmark results of the `OpenCAL-CL` versions of *SciddicaT* are instead shown in Figure 5.13 and reported in Table 5.3. Here, as expected, *SciddicaT*$_{ac}$ resulted the more performing on both devices. Unexpectedly, however, all the experiments executed on the GTX 980 have outclassed simulations that were performed on the Tesla K40, notwithstanding the first one being a gaming oriented GPU, while the latter a HPC dedicated device. Even GPU hardware issues might be taken into account: for instance, the K40, though having more cores with respect to the GTX 980, has a lower CUDA core clock-rate (745MHz vs 1126MHz) and lower memory clock-rate (6008 MHz vs 7012 MHz). Also cache issues could justify the results, since the K40 has less of both L1 and L2 level cache memories than the GTX 980, this latter benefiting from Nvidia's hardware improvements carried out in the more recent Maxwell architectures with respect to the Kepler ones. Moreover, independently from the adopted device, the *SciddicaT* version exploiting the GPU local memory did not resulted faster than the corresponding global memory version. This can be justified by the low transition function computational intensity, whereby work-items do not access data in local memory a sufficient number of times to result in better trade-off and thus better performances.

In addition, in the case of *SciddicaT*$_{ac}$, it is worth to note that the CPU performs better than the considered GPUs: 2.70 seconds on 16 threads, against 2.98 and 3.96 seconds on

Threads	$SciddicaT_{naive}$	$SciddicaT_{ac}$	$SciddicaT_{ac+esl}$
1	8,665.033	303.904	308.735
2	6,240.054	221.756	217.155
4	3,366.411	111.761	113.116
8	1,945.686	58.656	57.947
16	1,385.546	31.279	30.077

TABLE 5.4: Elapsed times (in s) registered during the *transition function stress test* for the simulation of the Tessina Landslide (cf. Figure 5.10) by different `OpenCAL` and `OpenCAL-OMP` versions of the *SciddicaT* fluid-flow model. The adopted CPU is an Intel Xeon 2.0GHz E5-2650.

the GTX 980 and the Tesla K40, respectively. This can be explained by considering that the mesh generated by the quantization algorithm is too small to exploit the GPU latency thread hiding mechanism at best [136]. In fact, the mean number of cells processed per step is 3,277 (cf. Section 5.7), which is of the same order of magnitude of the number of cores of the adopted GPUs (cf. above in this Section). This also leads to a waste of bandwidth. In fact, while the other versions were able able to adequately exploit the available bandwidth (e.g. the $SciddicaT_{naive}$ version reached about 88 GB/s on the Tesla K40 GPU), the one exploiting the quantization optimization was not able to take advantage of it (achieving 10 GB/s only). Eventually, a further study performed on the most time consuming kernels has shown that the achieved bandwidth is significantly higher for the CPU. Particularly indicative is the value measured for the more time consuming kernel, i.e. the one implementing the stream compaction algorithm. This latter, which takes alone about the 55% of the overall execution time on both the adopted CPU and GPUs versions, exploits the bandwidth the 35% better on the CPU, while the other kernels are bandwidth equivalent or perform better on the GPUs, all having however in this latter case a negligible percentage of the overall execution time (about the 3%). In other words, the standard test case here considered is simply too small to make decent use of the considered GPUs and, consequently it not surprising that the CPU performs better in this specific case.

5.9.2 *Transition Function Stress Tests*

As anticipated, in order to evaluate performances when considering computationally intensive state transitions, further tests were carried out by fictitiously increasing the complexity of the *SciddicaT* transition function. This was done by reapplying the transition function σ for a total of 200 times during each simulation step, excluding data transfer (e.g. from global to local memory, in the case of $SciddicaT_{local}$).

Results of the benchmarks executed on the CPU are shown in Figure 5.14, both in terms of execution time and speed-up. Raw execution times for this benchmark are also reported in Table 5.4. A more pronounced timings decrease is here observed for each *SciddicaT* version as the number of threads is increased, with a maximum speed-up of about 289 for the $SciddicaT_{ac+esl}$ execution on 16 threads. Here, the implementations exploiting the active cells optimization outperform the naive one of two orders of magnitude.

Figure 5.15 shows instead, the benchmark results of the different `OpenCAL-CL` versions of *SciddicaT* on the considered graphic hardware for the transition function stress test. Raw execution times are also reported in Table 5.5. Here, conversely from the standard tests,

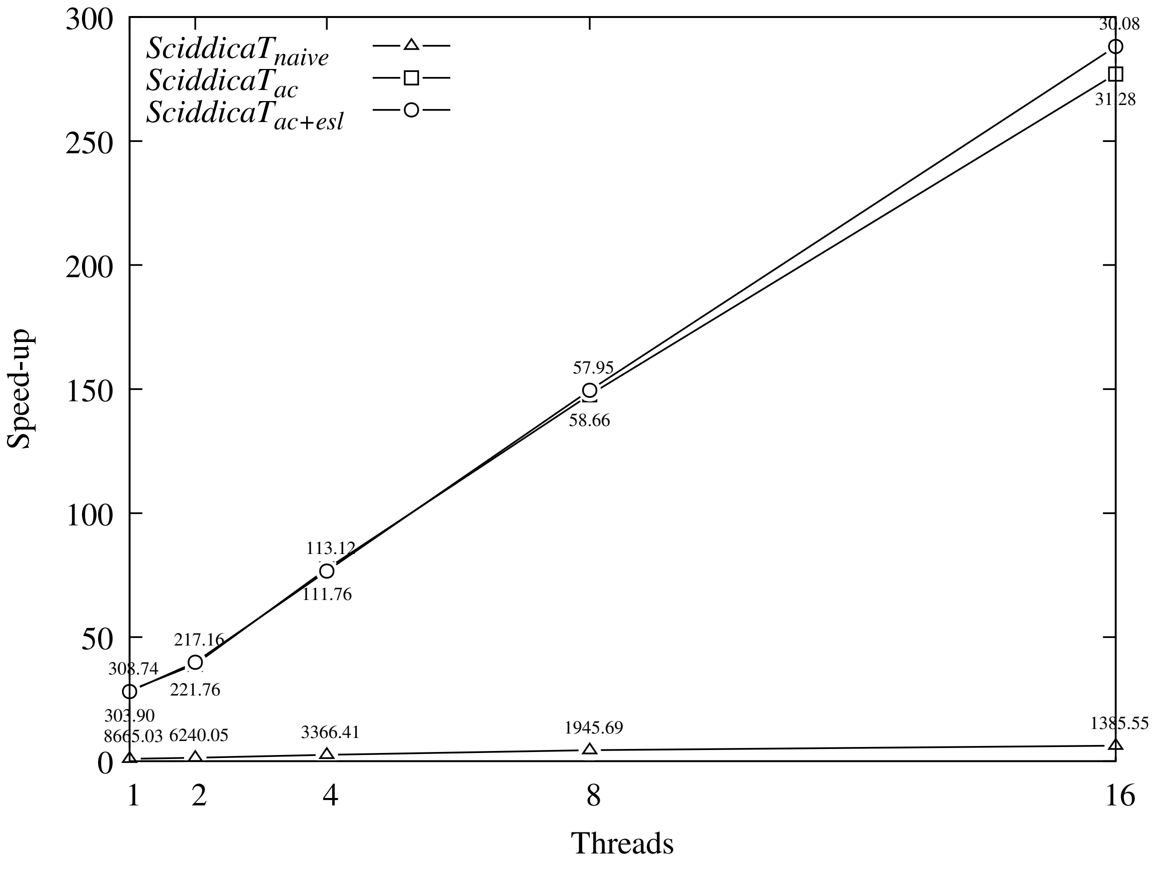


Figure 5.14: Speed-up obtained during the *transition function stress test* by the different OpenCAL-OMP versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown in correspondence of each speed-up vertex. The considered case study is the Tessina Landslide (cf. Figure 5.10). The adopted CPU was an Intel Xeon 2.0GHz E5-2650 CPU.

Device	$SciddicaT_{naive}$	$SciddicaT_{local}$	$SciddicaT_{ac}$
Tesla K40	11.108	10.605	3.960
GTX 980	5.063	5.453	2.981

Table 5.5: Elapsed times (in seconds s) registered during the *transition function stress test* for the simulation of the Tessina Landslide (cf. Figure 5.10) by different OpenCAL-CL versions of the *SciddicaT* fluid-flow model. The adopted OpenCL compliant devices are a NVIDIA Tesla K40 and a Nvidia GTX 980.

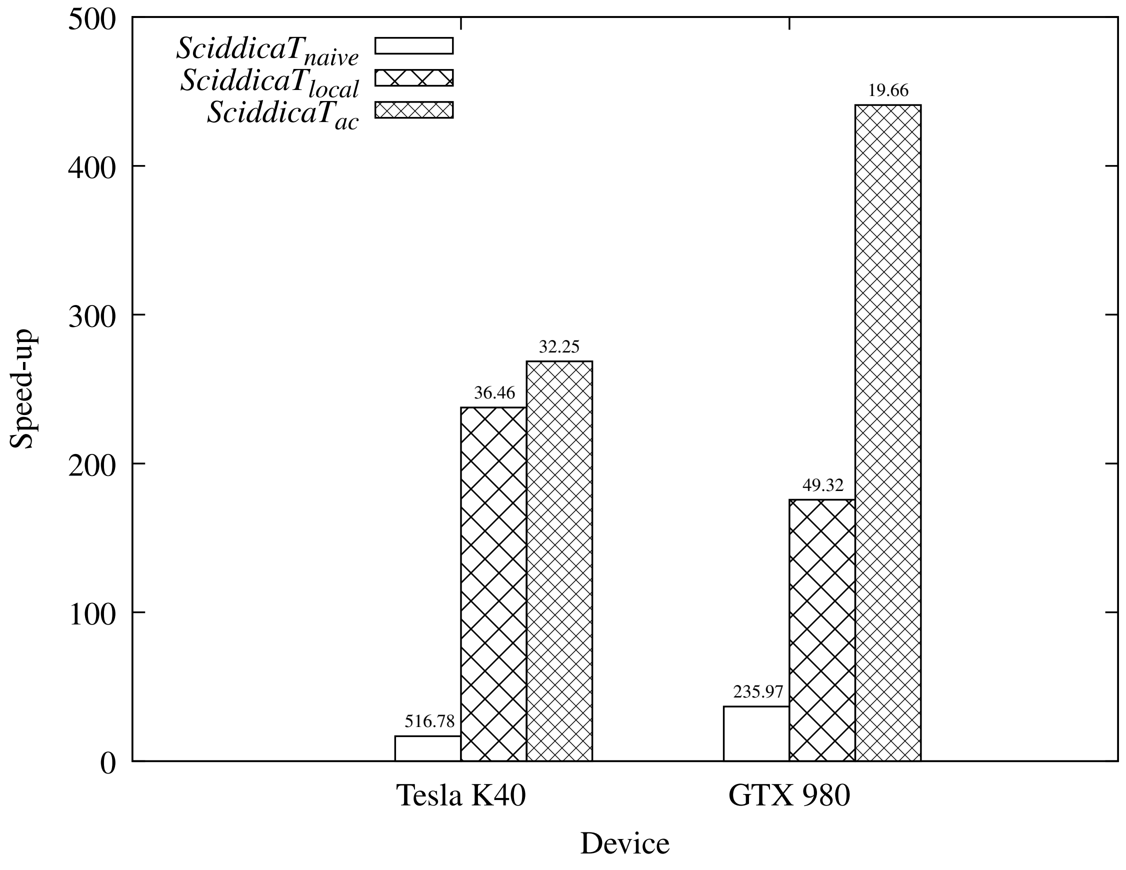


FIGURE 5.15: Speed-up obtained during the *transition function stress test* by the different `OpenCAL-CL` versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown on top of each speed-up bar. The considered case study is the Tessina Landslide (cf. Figure 5.10). The adopted OpenCL compliant devices were a Nvidia Tesla K40 and an Nvidia GTX 980.

the *SciddicaT* version exploiting the GPU local memory resulted significantly faster with respect to the corresponding global memory version on both the considered devices. In particular, in this case, the Tesla K40 reported the best result, evidencing a better local memory system (i.e., better tradeoff between local memory access/transfer) with respect to the GTX 980 GPU. Nevertheless, the *SciddicaT$_{ac}$* performances resulted always better than any CPU/GPU version (the GTX 980 performing better), demonstrating even in this case the validity of the active cells optimization. It is worth to note that this time the best GPU performance registered by the *SciddicaT* OpenCAL-CL versions significantly overcame the one registered on the CPU. In particular, the *SciddicaT$_{ac}$* ran about 441 times faster than the serial version of *SciddicaT$_{naive}$*, against the best 289 speed-up registered on the CPU, pointing out, as expected, the full suitability of GPGPU solutions in the case of sufficiently computational intense simulation models.

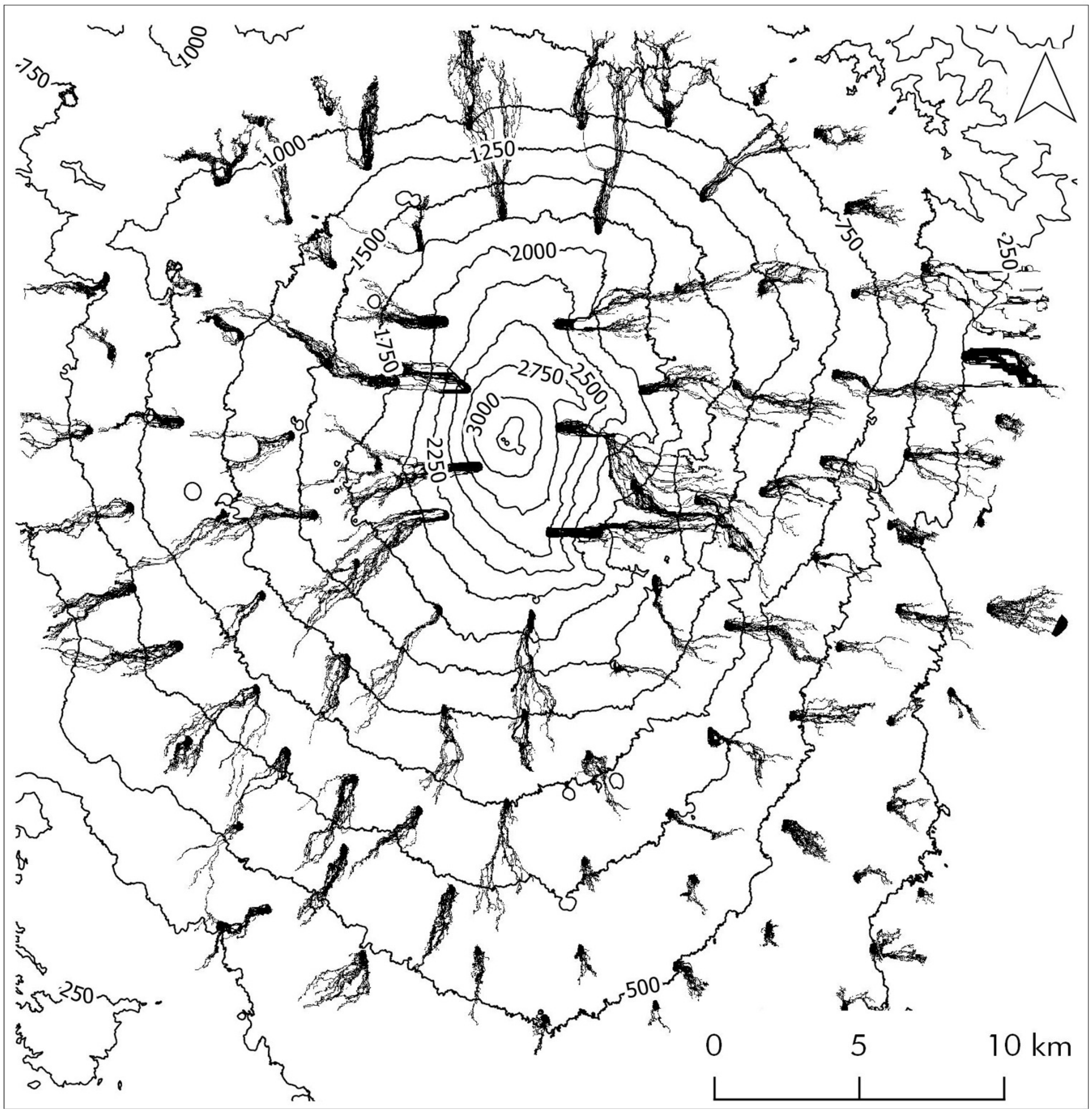


FIGURE 5.16: *SciddicaT* simulation stress test of 100 landslide sources distributed over a DEM of 3593 rows per 3730 columns, with square cells of 10 m side. Landslides paths are represented in black.

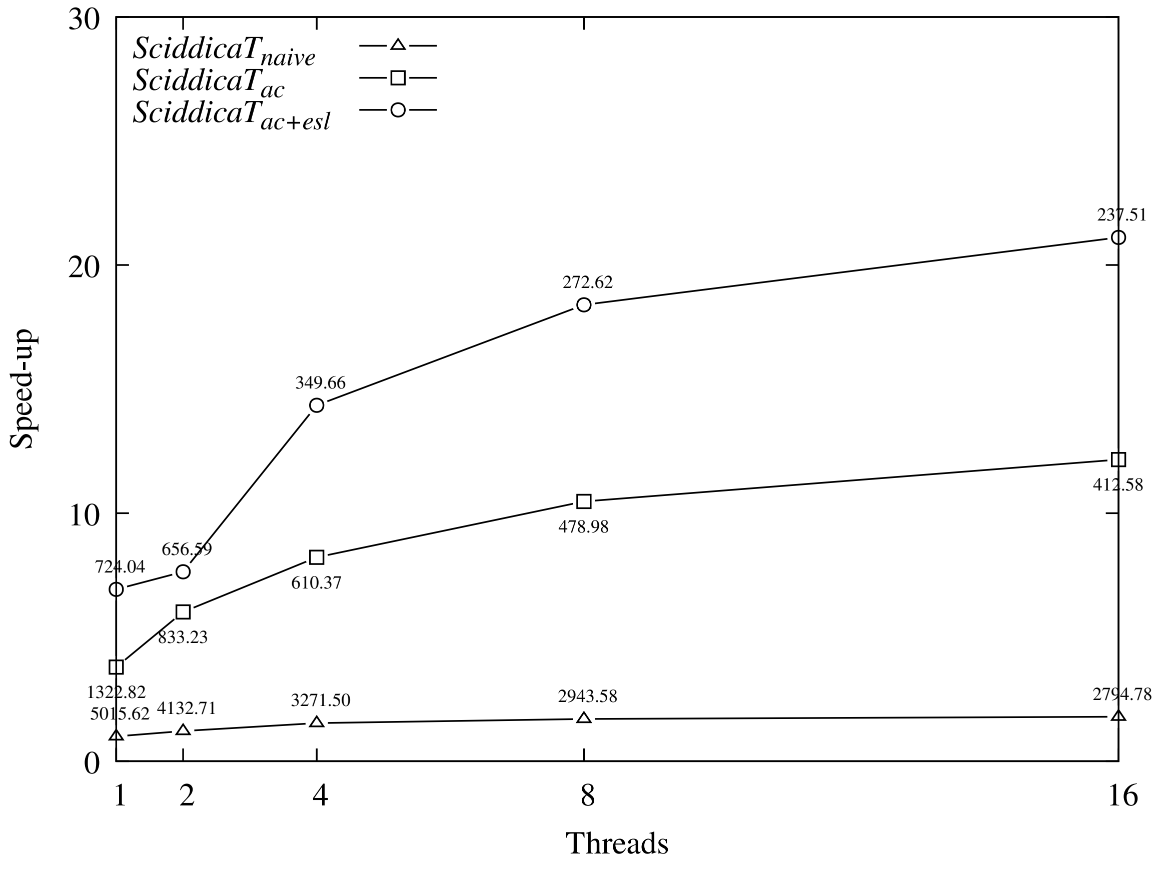


FIGURE 5.17: Speed-up obtained during the *computational domain stress test* by the different `OpenCAL-OMP` versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown in correspondence of each speed-up vertex. The considered case study is the simulation shown in Figure 5.16. The adopted CPU was an Intel Xeon 2.0GHz E5-2650 CPU.

5.9.3 *Computational Domain Stress Tests*

In order to evaluate performances when larger computational domains are taken into account, further tests were carried out by considering a DEM of 3,593 rows per 3,730 columns, with square cells of 10 m side. Moreover, the landslide source was uniformly replicated 100 times over the extended DEM and a simulation executed for each combination of *SciddicaT* versions and available devices. Figure 5.16 shows the simulation outcomes obtained by considering the wider DEM and the 100 landslide sources.

Computational results of the `OpenCAL-OMP` versions of *SciddicaT* are reported in Figure 5.17 and Table 5.6. Similarly to the standard tests, a slight timings decrease is observed for all cases as the number of threads is increased. Values increase accordingly to the adopted optimizations, resulting $Sciddica_{ac+esl}$ the fastest version with a value of about 21.

Benchmark results of the `OpenCAL-CL` different versions of *SciddicaT* on the computational domain stress tests are instead reported in Figure 5.18 and Table 5.7. Here, as for the standard tests, the *SciddicaT* version exploiting the GPU local memory did not result significantly faster than the corresponding global memory version on both the considered devices, by confirming that local memory has to be accessed an elevated number of times

Threads	$SciddicaT_{naive}$	$SciddicaT_{ac}$	$SciddicaT_{ac+esl}$
1	5,015.624	1,322.817	724.035
2	4,132.714	833.233	656.593
4	3,271.501	610.365	349.662
8	2,943.584	478.984	272.623
16	2,794.782	412584	237.513

TABLE 5.6: Elapsed times (in seconds) obtained for the *computational domain stress test* based on the simulation shown in Figure 5.16) by different `OpenCAL` and `OpenCAL-OMP` versions of the *SciddicaT* fluid-flow model. The adopted CPU is an Intel Xeon 2.0GHz E5-2650.

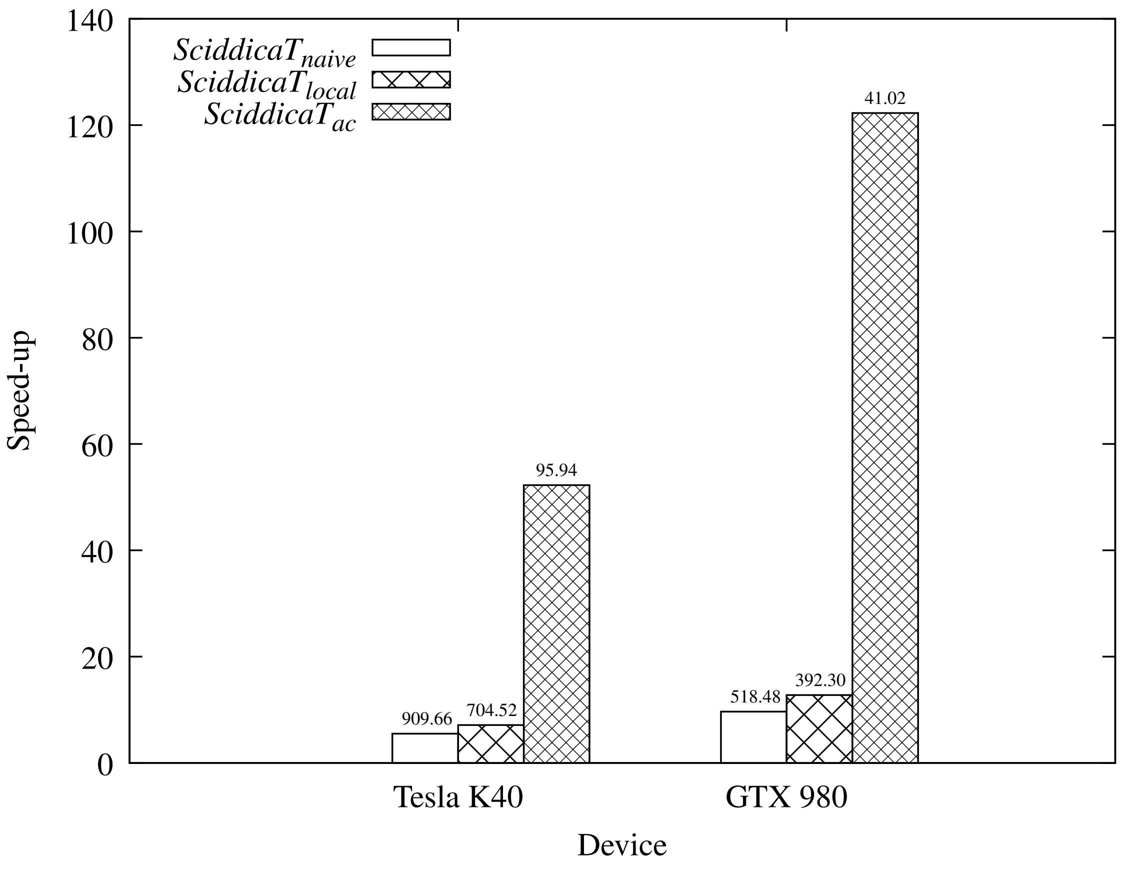


FIGURE 5.18: Speed-up obtained during the *computational domain stress test* by the different `OpenCAL-CL` versions of the *SciddicaT* fluid-flow model. Elapsed times in seconds are also shown on top of each speed-up bar. The considered case study is the simulation shown in Figure 5.16. The adopted OpenCL compliant devices were a Nvidia Tesla K40 and an Nvidia GTX 980.

Device	$SciddicaT_{naive}$	$SciddicaT_{local}$	$SciddicaT_{ac}$
Tesla K40	909.664	704.518	95.939
GTX 980	518.479	392.300	41.019

TABLE 5.7: Elapsed times (in seconds) obtained for the *computational domain stress test* based on the simulation shown in Figure 5.16) by different `OpenCAL-CL` versions of the *SciddicaT* fluid-flow model. The adopted OpenCL compliant devices are a Nvidia Tesla K40 and a Nvidia GTX 980.

to take an effective advantage compared to the global one (and thus can result more useful for higher computationally complex models). However, the $SciddicaT_{ac}$ versions performances resulted always better than any CPU/GPU version (the GTX 980 performing better), demonstrating even in this case the validity of the active cells optimization. Moreover, even in this case, the best GPU performance overcame the one registered on the CPU. In particular, the $SciddicaT_{ac}$ ran about 122 times faster on the GTX 980 than the serial version of $SciddicaT_{naive}$, against the best 21 absolute speed-up registered on the CPU pointing out, as expected, the usefulness of GPGPU solutions also in the case of extended computational domains. The result is justified by the same evaluations performed for the standard test case. In particular, the higher dimension of the computational domain stress test mesh permits the GPUs to always perform better than the CPU in terms of achiewed bandwidth for all the considered *SciddicaT* versions (the $SciddicaT_{ac}$ version included, which achieves about 77 GB/s of bandwidth on the Tesla K40, against the 10 GB/s achieved on the smaller mesh), by consequently allowing to hide the thread latency in all cases, and thus to better exploit the GPU computational power. Eventually, it is worth to note that, as in the standard tests, the GTX 980 outperformed the Tesla K40, confirming that a gaming-oriented device is a preferable solution in case of low-intense computational models.

5.10 DISCUSSION AND OUTLOOKS

In this chapter the first release of `OpenCAL` is presented, a new open source computing abstraction layer for Scientific Computing, currently supporting Cellular Automata, Extended Cellular Automata and the Finite Differences computational methods.

Besides the serial implementation, two different parallel versions were developed, namely `OpenCAL-OMP` and OpenCAL-CL, based on OpenMP and OpenCL, respectively. The first one allows to exploit multi-core CPUs on shared memory computers, while the second a wide range of heterogeneous devices like GPUs, FPGAs and other many-core coprocessors.

Each version was designed to be the most reliable and fast possible and, for this purpose, the C language was adopted and efficient data types and algorithms considered. In particular, also to permit a more straightforward OpenCL parallelization, linearized arrays were adopted to represent both one-dimensional and higher order structures like substates and neighbourhoods. Moreover, the quantization optimization, which allows to define the set A of non-stationary cells to which restrict the application of the transition function, was implemented in each version. Specifically, a straightforward stream compaction was considered in OpenCAL, which serially checks the state of the cells in the computational domain, by placing the coordinates of non-stationary cells in A. `OpenCAL-OMP` essentially implements the same strategy, even if a pool of threads preliminarily build a set of sub-arrays of active cells in parallel, which are eventually assembled together to form the final array A. A different algorithm was implemented in OpenCAL-CL, where the parallel stream compaction relies on a parallel prefix sum algorithm used to preliminary evaluate the offsets to be used by work-items to fill the array of active cells A in parallel. In addition to the quantization optimization, `OpenCAL` and `OpenCAL-OMP` were designed to allow for the explicitation of the global transition function, by also allowing selective updating of substates.

The SciddicaT XCA landslide simulation model was considered to show the straightforward implementation of a computational model and also to assess numerical correctness and computational efficiency of each `OpenCAL` implementation. Specifically, the `OpenCAL` and OpenCAL-OMP implementations of three different versions of *SciddicaT* were shown, from a naive one, $SciddicaT_{naive}$, to a version supporting the quantization optimization, $SciddicaT_{ac}$, up to a fully optimized version, $SciddicaT_{ac+esl}$, supporting both the quantization and the explicitation of the global transition function. The first two versions of *SciddicaT* were also implemented in OpenCAL-CL. In addition, a naive version of *SciddicaT* exploiting the local memory, namely $SciddicaT_{local}$, was implemented in OpenCL to evaluate the role of different GPUs memory levels.

For each *SciddicaT* version, the Tessina landslide was considered and a total of ten benchmarks (simulations) executed to evaluate correctness and timings on each considered hardware configuration, namely a 16 threads Intel Xeon CPU based workstation and two Nvidia GPUs. Numerical correctness was confirmed by all the simulation outcomes, which perfectly matched to the one of the OpenCAL implementation of $SciddicaT_{naive}$ that was selected for reference. Regarding computational performance, the different *SciddicaT* versions demonstrated to be able to efficiently exploit the computational power of the heterogeneous devices considered in this work, by reducing the execution time of all the performed benchmarks accordingly to the progressively adopted optimizations. However, the best result obtained by $SciddicaT_{ac+esl}$ using 16 threads on the CPU surprisingly doubled the best one obtained by $SciddicaT_{ac}$ on the GPU in terms of absolute speed-up (i.e. computed with respect to the timing of the reference simulation), probably due to the very low computational complexity of the transition function and the dimension of the computational domain. Nevertheless, subsequent stress tests performed by fictitiously complicating the transition function execution, and a further set of tests where the computational domain was considerably increased with respect to the one originally considered, overturned the results, and GPUs significantly resulted faster than the CPU, pointing out their usefulness in case of the simulation of computationally heavy models. Eventually, as regards GPU local memory, it showed to provide an actual advantage only in the case of the first set of stress tests, pointing out that data must be accessed an adequate number of times to be effective. Here, in particular, the Tesla K40 resulted more efficient with respect to the GTX 980, even if based on the previous Nvidia hardware architecture, probably due to a better management of the local memory.

Though preliminary, obtained results confirm correctness and efficiency of the different `OpenCAL` versions here presented, by highlighting their goodness for numerical model development of complex systems in the field of Scientific Computing and their execution on parallel heterogeneous devices. Moreover, since the implementations do not significantly differ from an OpenCAL implementation to another, it is easily possible to obtain two different CPU/GPU parallel versions of the same model with a minimum effort and, therefore, to test them on the available hardware to select the best platform for execution. In fact, as shown for the case of *SciddicaT*, the best choice can deeply depend on both the computational complexity of the transition function and on the extent of the computational domain, and the best solution can not be determined *a priori*.

Nevertheless, a fine tuning of underlying data structures and algorithms will be performed in order to make `OpenCAL` still more performing and MPI will be adopted to allow `OpenCAL` to exploit the computational power of distributed memory systems. As regard the OpenCL implementation, the seamless management of GPUs local memory will be introduced in the next releases, and Multi-GPU support added to intelligently scale the overall system performances. Subsequent releases will also progressively support further computational paradigms, like the Lattice Boltzmann, the Smoothed Particle Hydrodynamics (SPH), as well as other mesh-free numerical methods, with the aim to become a general software abstraction layer for computation.

The `OpenCAL` software libraries, together with a comprehensive installation and user manual accompanied by numerous examples, are currently freely available on GitHub, at `https://github.com/OpenCALTeam/opencal`.

6 OPENCAL-CLUST - THE DISTRIBUTED MEMORY IMPLEMENTATION OF OPENCAL

In reality the space in which we moved was all battlemented and perforated, with spires and pinnacles which spread out on every side, with cupolas and balustrades and peristyles, with rose windows, with double- and triplearched fenestrations, and while we felt we were plunging straight down, in reality we were racing along the edge of moldings and invisible friezes, like ants who, crossing a city, follow itineraries traced not on the street cobbles but along walls and ceilings and cornices and chandeliers. Now if I say city it amounts to suggesting figures that are, in some way, regular, with right angles and symmetrical proportions, whereas instead, we should always bear in mind how space breaks up around every cherry tree and every leaf of every bough that moves in the wind, and at every indentation of the edge of every leaf, and also it forms along every vein of the leaf, and on the network of veins inside the leaf, and on the piercings made every moment by the riddling arrows of light, all printed in negative in the dough of the void, so that there is nothing now that does not leave its print, every possible print of every possible thing, and together every transformation of these prints, instant by instant, so the pimple growing on a caliph's nose or the soap bubble resting on a laundress's bosom changes the general form of space in all its dimensions.

— Italo Calvino

For most problems in physics and engineering there is a huge demand for improving the time needed to solve a certain problem. For instance, some numerical meteorological models are so complex that even executed on powerful computers the execution time would be so long that by the time the results are available the prediction would be of no practical use. But speed is not the only important key aspect in scientific computing. Sometimes, the accuracy of a solution needs to be improved and that usually translates to a bigger and denser discretization of the problem. It is clear that in order to overcome these limitations more computing power needs to be employed.

Nowadays, computing systems are often equipped with more than one GPU. From high-end workstations, that are able to accommodate up to 8 GPUs or more on the same motherboards to large clusters, that are usually composed of several computing nodes (see Figure 4.8b and Section 4.2 at page 35 and 31 respectively) each of them equipped with one or more accelerators possibly made by different manufacturers and with different underlying architecture. Programming distributed memory machines is complex especially for non-HPC computer scientist because of the intrinsic complexity introduced by the parallelism and because of the hardware heterogeneousness, and this is especially true since large clusters are equipped with several accelerators. However it is important to fully and effectively utilize such computing power, especially since having multiple GPUs per node improves the ratio performance over Watt and price over Watt. For these reasons, multi-GPU programming is becoming more and more important.

Unfortunately, neither CUDA or OpenCL supports natively a multi-GPU model. The model they support is based on a *single core-single GPU* relationship and works really well for tasks that are independent one from the other. On the other hand, the aforementioned model makes things more difficult when a task needs to have several GPUs cooperate in some way in order to solve a problem instance. As an example of an application of the supported model, the BOINC application [137] allows a user to donate computing power and time to solve relevant problems. In a multi-GPU environment, it works spawning N independent tasks and each of them is scheduled on one of the N available GPUs. When a task is finished, the application simply requests another task to the central server, the *task dispatcher*. No cooperation or communication is required between the GPUs as the tasks to be solved is self-contained, meaning that it does not need any external input or information to be fully completed. An example of the unsupported model, consider the Lattice Boltzmann (LB) [90] [92] [91] method on a domain that is decomposed along one axis, let's say the x axis, as depicted in Figure 6.1, so that each GPU is responsible for a subset of the whole mesh. Computation for a grid point that lies on the boundary of the domain portion assigned to the GPU 2 needs information about neighboring points that are stored on different GPUs (1 and 3 in this case). As a consequence, communication of such boundary points between the two devices is required.

6.1 OPENCAL-CLUST

This chapter describes the distributed memory version of `OpenCAL`, `OpenCAL-CLUST`, which has been designed to take advantage of the modern multi-GPU capabilities of multi-node systems. This makes `OpenCAL` applications deployable on a variety of computer architectures, from a single CPU workstation to large heterogeneous clusters. Section 6.1.1 starts by describing the *configuration file* that is used to dispatch data and computation across the machine. Sections 6.1.2 and 6.2 outline the adopted parallelization and domain decomposition strategies and introduce a set API and number of examples that show how `OpenCAL` can be used to code distributed memory applications. Finally, Section 6.4 discusses performance and tests.

6.1.1 *Run Configuration*

Each `OpenCAL-CLUST` application is attached with a running configuration that is provided by the user or the programmer and describes both which computational resources are going to be used during the execution and how the domain is decomposed among them. The running configuration is provided as a plain text file whose syntax and structure are described in Listing 6.1.

The configuration file starts with two lines describing:

1. The size of the domain along each of the dimensions as a list of positive integers.
2. The number N of computational nodes, each described and identified uniquely by its IP address.

N descriptions of nodes follow. A node i, $1 \leq i \leq N$ is described by a line containing its IP address IP_i and the number of devices (installed and available on that node) $NUM\_GPU\_NODE_i$ to be utilized. $NUM\_GPU\_NODE_i$ lines follow, each containing the definition of the devices within node i and its workload.

DOMAIN SIZE

```
DIM_1 DIM_2 ... DIM_N
```

NUMBER OF NODES

```
NUMBER OF NODES
```

IP NODE 1

```
IP_NODE_1 NUM_GPU_NODE_1
```

Device List Node 1

```
PLATFORM_NUMBER_1 DEVICE_NUMBER_1 LOAD_1_1
PLATFORM_NUMBER_1 DEVICE_NUMBER_2 LOAD_1_2
         ...
PLATFORM_NUMBER_1 DEVICE_NUMBER_K1 LOAD_1_K1_1
PLATFORM_NUMBER_2 DEVICE_NUMBER_1  LOAD_2_1
         ...
PLATFORM_NUMBER_2 DEVICE_NUMBER_K2 LOAD_2_K1_2
         ...
PLATFORM_NUMBER_M1 DEVICE_NUMBER_KM LOAD_M1_K1_M1
```

IP NODE 2

```
IP_NODE_2 NUM_GPU_NODE_2
```

Device List Node 2

```
PLATFORM_NUMBER_1 DEVICE_NUMBER_1 LOAD_1_1
PLATFORM_NUMBER_1 DEVICE_NUMBER_2 LOAD_1_2
         ...
PLATFORM_NUMBER_1 DEVICE_NUMBER_K2_1 LOAD_1_K2_1
PLATFORM_NUMBER_2 DEVICE_NUMBER_1  LOAD_2_1
         ...
PLATFORM_NUMBER_2 DEVICE_NUMBER_K2 LOAD_2_K2_2
         ...
PLATFORM_NUMBER_M2 DEVICE_NUMBER_K2_M2 LOAD_M1_K2_M2
```

⋮ ⋮

IP NODE N

```
IP_NODE_N NUM_GPU_NODE_N
```

Device List Node N

```
PLATFORM_NUMBER_1 DEVICE_NUMBER_1 LOAD_1_1
PLATFORM_NUMBER_1 DEVICE_NUMBER_2 LOAD_1_2
         ...
PLATFORM_NUMBER_1 DEVICE_NUMBER_KN_1 LOAD_1_KN_1
PLATFORM_NUMBER_2 DEVICE_NUMBER_1  LOAD_2_1
         ...
PLATFORM_NUMBER_2 DEVICE_NUMBER_KN_2 LOAD_2_KN_2
         ...
PLATFORM_NUMBER_MN DEVICE_NUMBER_KN_MN LOAD_M1_KN_MN
```

Listing 6.1: File Format for the domain decomposition of a `OpenCAL-CLUST` application. It describes the size of the domain and the machine on which the model is executed. Note that for each node of the cluster, the IP address and a list of devices is listed. Each device is identified, *à la* OpenCL, by platform and device number (within the platform).

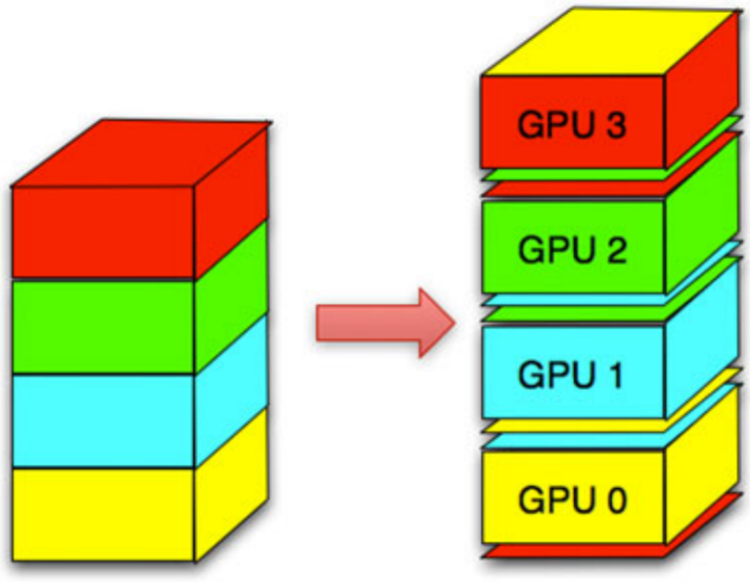


FIGURE 6.1: Domain decomposition along one axis.

A device is identified by its **platform number** $PLATFORM\_NUMBER_p$, $1 \leq p \leq M_i$, where M_i is the number of the platforms on node i, a **device number within the platform** $DEVICE\_NUMBER_l$ (l relative to the platform). A **load** parameter for the device d, $LOAD^i_{(p,d)}$ describing the amount of work assigned (portion of the domain) to that device.
Eventually, the list of devices within a node can be arbitrarily ordered.

6.1.2 *Domain Decomposition*

In this preliminary work, the general strategy for dividing work among the available nodes and devices is to decompose the domain along the first dimension listed in the run configuration. The configuration file has to correctly describe a $1D$ decomposition along the first dimension of the domain. This means that the following has to be always true:

$$\sum_{i}^{N}\sum_{p}^{M_i}\sum_{d}^{K_p} LOAD^i_{(p,d)} = DIM\_1$$

i.e. the sum of the loads has to match the size of the first dimension of the domain exactly. This ensures that the whole domain is assigned to some device on a node.

The decomposition follows the order in which nodes and devices are listed in the configuration file. Subsequent portions of not yet assigned portions of the domain are assigned to subsequent (with the respect to the order in which they appear in the file) devices. The size of such portion is described by the device load parameter.

In order to show how decomposition works, consider the configuration file in Listing 6.1.

```
16384 16384
2
192.168.1.111 2
0 0 4096
0 1 4099
192.168.1.222 3
0 0 1200
1 0 3200
1 1 3792
```

LISTING 6.1: Configuration file example. Size of the domain is 16384×16384, scattered along the first dimension among 2 nodes and 5 devices overall.

which defines a $2D$ domain of size $2^{14 \times 2^{14}} = 2^{28}$ points. The domain is scattered along the first dimension, x, among 2 nodes and 5 devices, in the following manner:

- $0 \le x < 4096 \longmapsto$ device $(0, 0)$ running at node 192.168.1.111.
- $4096 \le x < 4096 + 4099$ go to device $(0, 1)$ running at node 192.168.1.111.
- $4096 + 4096 \le x < 1200 + 4096 + 4096 \longmapsto$ device $(0, 0)$ running at node 192.168.1.222.
- $1200 + 4096 + 4096 \le x < 3200 + 1200 + 4096 + 4096 \longmapsto$ device $(1, 0)$ running at node 192.168.1.222.
- $3200 + 1200 + 4096 + 4096 \le x < 3792 + 3200 + 1200 + 4096 + 4096 \longmapsto$ device $(1, 1)$ running at node 192.168.1.222.

Generally speaking, using the decomposition described in section 6.1.2, when N devices are involved, GPU i needs to know, for the update of the grid points within the boundaries of its subdomain, the value of substates of the neighboring subdomains belonging to different devices (that can be possibly located on different nodes). At each iteration, it must perform all the operations depicted in Figures 6.2 and 6.3 (assuming periodic boundary conditions for the sake of simplicity).

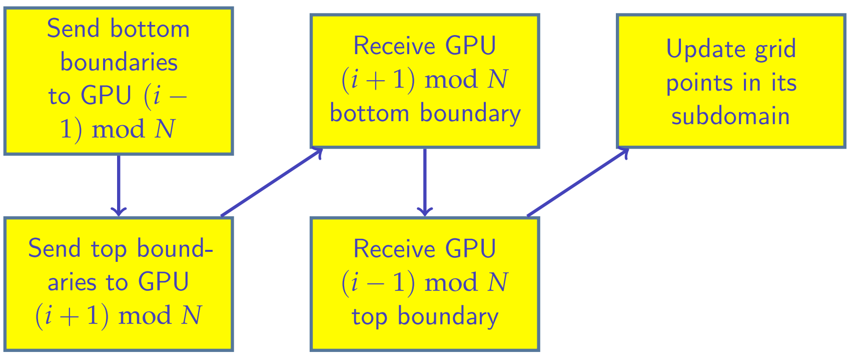

Send bottom boundaries to GPU $(i - 1) \bmod N$
Send top boundaries to GPU $(i + 1) \bmod N$
Receive GPU $(i + 1) \bmod N$ bottom boundary
Receive GPU $(i - 1) \bmod N$ top boundary
Update grid points in its subdomain

FIGURE 6.2: Communication scheme in a multi-GPU OpenCL application

Note that this approach always requires CPU intervention as the OpenCL device-device memory transfer feature in the current implementation only works between devices that are within the same OpenCL context. This implies that synchronization is also required during the boundary exchange, as depicted in figure 6.3b. For a communication step to take place, it is necessary to:

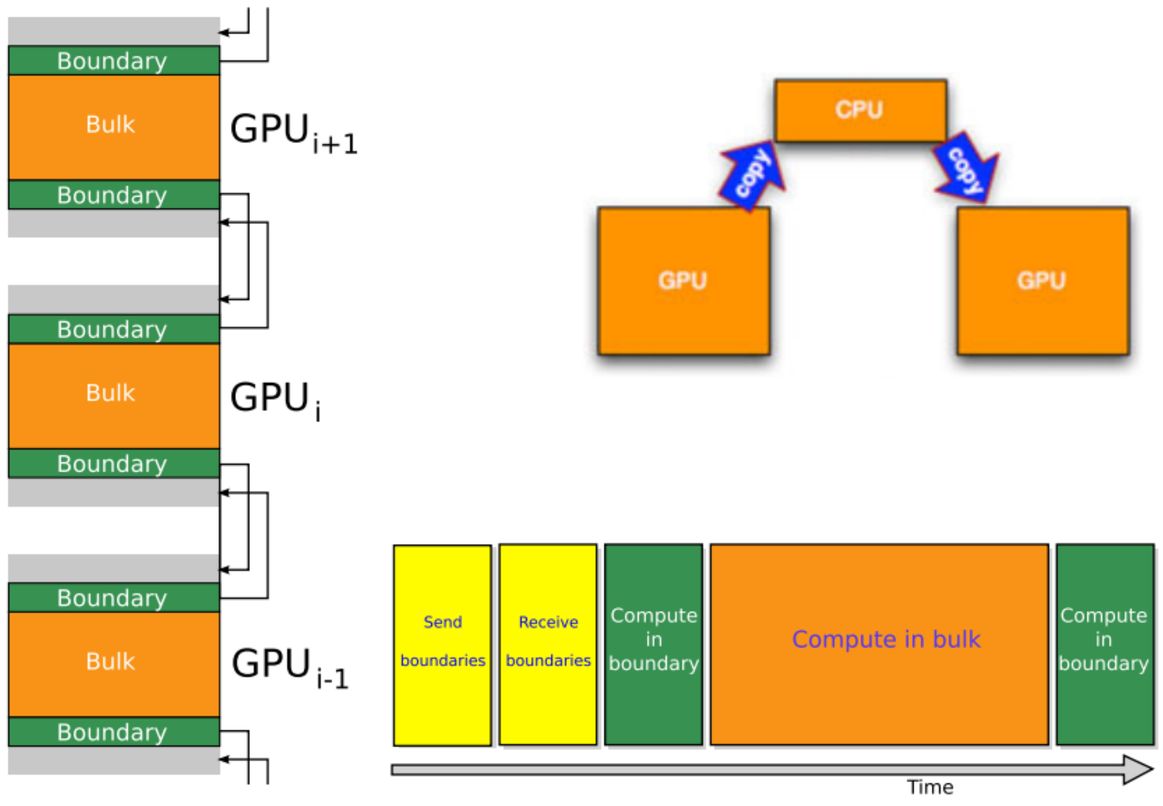


FIGURE 6.3: The adopted multi-GPU computation scheme.

1. Pack and upload boundary data to the CPU (device $\mapsto$ host memory transfer)
2. If the two GPUs involved in the communications are controlled by different nodes, an extra communication step over the network is performed between the two nodes. This phase is implemented via MPI [56] to ensure and guarantee portability and scalability.
3. Unpack and upload boundary data to the recipient GPU (host $\mapsto$ device memory transfer).

6.2 THE OPENCAL-CLUST PARALLEL IMPLEMENTATION

In this section, the OpenCL distributed memory and multi-GPU parallel implementation of `OpenCAL` is described, which allows for the parallel execution on one or more accelerators installed on computing nodes interconnected via network and MPI. This section describes the difference between OpenCAL-CL and `OpenCAL-CLUST` and discusses only the set of the additional API calls that the latter version exposes.

The programming model adopted by `OpenCAL-CLUST` is similar to the other versions of `OpenCAL` (see Section 5) but it introduces new concepts that mainly reflect the multi-GPU and multinode structure of the target machines.

The main difference is the addition of the `MultiNode` class (see Listing 6.2).

```
template <class Init_Functor,class Finalize_Functor>
class MultiNode{
public:
    Cluster c;
    Init_Functor *init;
    Finalize_Functor *finalize;
        ...
```

LISTING 6.2: `OpenCAL-CLUST` MultiNode Class Declaration

It manages communications and computations across nodes and across GPUs and must be constructed by all the processes/nodes.

In order to use `OpenCAL-CLUST`, a valid domain decomposition must be specified, see Section 6.1.2 and 6.1.1. The Domain Decomposition format described in section 6.1.2 is reflected in the `OpenCAL` implementation with the following classes:

`device` described by 4 non-negative integers, two of which identify the device within the node (using the OpenCL idiom of platform and device numbers) while the rest describe the portion of the subdomain assigned to the specific device.

`node` containing an IP address, a integer values describing the portion of subdomain assigned to it and finally, a list of `Devices` installed on the machine that are used for the computation.

`cluster` containing a list of `Nodes` that are concurrently used to execute an `OpenCAL` application.

Each `OpenCAL-CLUST` application necessitates of a `Cluster` object correctly initialized that can be conveniently constructed from a file using the `calFromClusterFile` function exposed. Given a configuration file, `calFromClusterFile` parses, validates and finally returns a valid `Cluster` instance.

The information stored in the `Cluster` object is then utilized by each MPI process to allocate and to initialize all the listed devices. The library is designed s.t. each MPI process runs several instances of OpenCAL-CL (see Section 5.5), each executing on a different device and on a different portion of the original subdomain. Note also that, `OpenCAL-CLUST` degenerates to OpenCAL-CL when only one none and one device are utilized, and that a multi-GPU single node configuration is obtainable specifying a single node with several devices in the configuration file.

Devices within a node are managed by a `CALCLMultiGPU` object, which coordinates them and can be created using the `calclMultiGPUDef2D` API call. `CALCLMultiGPU` takes care of storing hooks to a per node's devices and resources as the list of compiled kernels for each device. It also exposes a number of functions for adding or removing a device from the pool and for boundaries exchange between two devices.

6.2.1 *`Init` and `finalize` functors*

Listing 6.2 shows that the `MultiNode` object, among others fields, contains a `Cluster` object and two pointers to functors which type is shown in listing 6.3.

```
void finalize(struct CALCLMultiGPU*);
void init(struct CALCLMultiGPU*, const Cluster*);
```

LISTING 6.3: `OpenCAL-CLUST` init and finalize functor signature

`init` and `finalize` if defined, are executed by each MPI process (Node) at the initialization and finalization phases, respectively. Their definition is optional. They can be employed to perform operations that are hard to manage automatically on a per-node basis. For example, a certain node might need a particular initialization phase such as module loading for instance. The library can completely hide the initialization and allocation phases, but it is important to note that this behaviour can be changed and manual initialization can be enabled. When it is the case, the init functor can be employed to take care of making sure that each MPI process allocates all of its listed devices with the right resources in order to process the assigned domain.

Listing 6.4 shows an example of init function that is part of the `OpenCAL` Julia Set generator example application shown in section 6.3.1 and in listing 6.8. It shows how manual initialization can be performed from the configuration file. Each nodes access its own list of devices using its MPI rank (lines 3-4) and add them all to the pool using the `calclAddDevice` function (lines 7-11).

```
1  void init(struct CALCLMultiGPU* multigpu, const Cluster* c){
2      //add devices from the cluster configuration
3      Node mynode = c->nodes[rank];
4      auto devices = mynode.devices;
5      struct CALCLDeviceManager* calcl_device_manager = calclCreateManager();
6      calclSetNumDevice(multigpu, devices.size());
7      for (auto& d : devices) {
8          calclAddDevice(multigpu,
9              calclGetDevice(calcl_device_manager, d.num_platform, d.num_device),
10             d.workload);
11     }
12     //create the model
13     struct CALModel2D* host_CA =
14     calCADef2D(mynode.workload, mynode.columns, CAL_MOORE_NEIGHBORHOOD_2D, CAL_SPACE_TOROIDAL,
           CAL_NO_OPT);
15     //add the substate
16     Q_fractal = calAddSubstate2Di(host_CA);
17     //gosh  cells radius
18     int borderSize = 1;
19     calclMultiGPUDef2D(multigpu, host_CA, KERNEL_SRC, KERNEL_INC,
20                         borderSize, mynode.devices, c->is_full_exchange());
21     calclAddElementaryProcessMultiGPU2D(multigpu,     KERNEL_LIFE_TRANSITION_FUNCTION);
22 }
```

Listing 6.4: `OpenCAL-CLUST` finalize example code for the Julia Set generator application. It outputs the node's portion of a substate to a file.

Listing 6.4 shows also that cluster object is used to eventually configure a `calclMultiGPUDef2D` object (line 19). Note that the decomposition along the first dimension of the domain is clear here. A CALModel2D object is created using the node's workload (the number of rows in this case). The finalize functor is executed, if defined, at the end of the computation at the node level. Listing 6.5 shows an example in which the `finalize` is used to save a per node copy of the grid to a file.

```
void finalize(struct CALCLMultiGPU* multigpu){
//for each node, save the substate to a file
std::string fractal_str = "./fractal_portion" + std::to_string(rank)+".txt";
    calSaveSubstate2Di(multigpu->device_models[0]->host_CA, fractal_substate, (char*)fractal_str
        .c_str());
```

Listing 6.5: `OpenCAL-CLUST` init and finalize functor signature

Note that an implicit MPI barrier is present right before the execution of the `init` and `finalize` call.

A MultiNode is used in user code as shown in listing 6.6

```
//Construct a MultiNode object
MultiNode<decltype(init), decltype(finalize)> mn(cluster, mpi_world_rank, init, finalize);
//trigger allocation and init execution
mn.allocateAndInit();
```

Listing 6.6: `OpenCAL-CLUST` init and finalize functor signature

6.2.2 *Kernel Side*

Kernel side API and *build-in* variables are added. Those additional variables and functions can be used to access and manage boundary cells belonging to neighboring devices, the so call *ghost cells*. Each device keeps an updated copy of neighboring devices boundaries and offers to the programmer the abstraction of a single domain. A *built-in* variable, `border_radius` is exposed in the kernel side and can be used, from within the kernel, to retrieve the size of boundaries.

The following sections provides examples of usage of `OpenCAL-CLUST` . Section 6.3 provides a complete code that generates large `BMP` images of Julia Sets. Section 6.3.4 shows how to apply a Sobel's convolutional filter to a large 2D image. Section 6.3.5 shows an implementation of *sciddicaT*, described in Section 5.6.

6.3 opencal-clust high resolution julia set generation

As a first illustrative example of the usage of `OpenCAL-CLUST` this section shows an application which generates high resolution Julia Set images running on an heterogeneous cluster of GPUs.

6.3.1 *Julia Set*

Julia set fractals are normally generated by initializing a complex number $z = x + yi$ where $i2 = -1$ and x and y are image pixel coordinates. Then, z is repeatedly updated using:

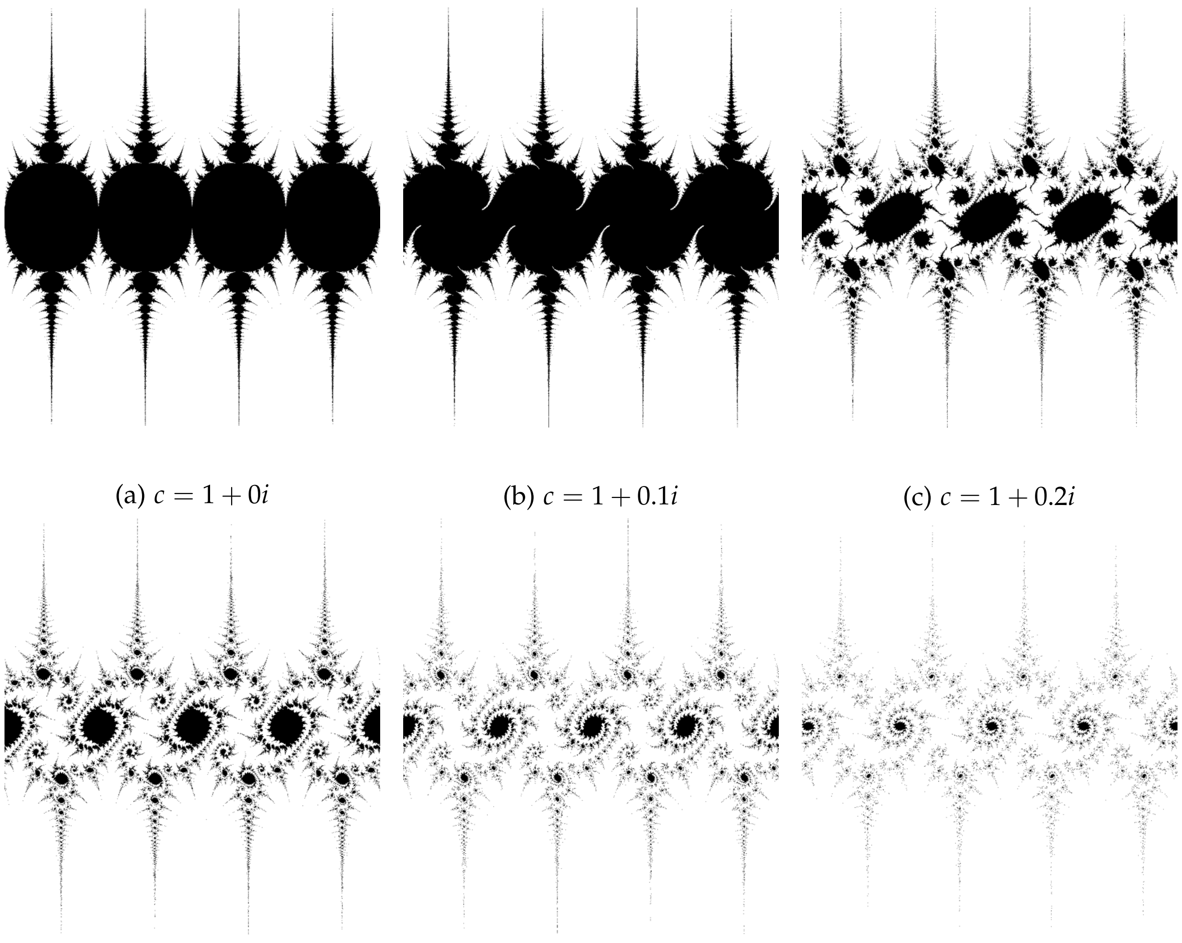
$$z_{n+1} = z_n^2 + c$$

where c is a complex constant that gives a specific Julia set (see Figure 6.4).

In the broader sense the exact form of the iterated function may be almost anything of the form $z_{n+1} = f(z_n)$. Interesting sets arises with non-linear functions. Commonly used ones include the following:

$$\begin{aligned} z_{n+1} &= c\,sin(z_n) & z_{n+1} &= c\,exp(z_n) \\ z_{n+1} &= i\,c\,cos(z_n) & z_{n+1} &= c\,z_n(1 - z_n) \end{aligned}$$

A point is said to be part of the set if after the repeated iteration it does not tend to infinity. The fractal is created by first mapping each pixel to a rectangular region of the complex plane. Each pixel represents the initial value of z_0. The series is computed for each pixel and if it does not diverge to infinity it is drawn in black while, if it doesn't, then a



(a) $c = 1 + 0i$ (b) $c = 1 + 0.1i$ (c) $c = 1 + 0.2i$

(d) $c = 1 + 0.3i$ (e) $c = 1 + 0.4i$ (f) $c = 1 + 0.5i$

FIGURE 6.4: 6 Examples of Julia sets obtained variating the constant c.

color is chosen depending on the number of iterations taken to diverge. Nevertheless, this convergence or otherwise isn't always obvious and it may take a large number of iterations to resolve and so a decision procedure is required to determine divergence. This typically involves assuming the series tends to infinity as soon as its value exceeds some threshold; if the series has not diverged after a certain number of terms it is similarly assigned to be part of the set.

6.3.2 *Julia Sets `OpenCAL-CLUST` implementation*

In order to generate the fractal, for each pixel, information regarding how many steps are necessary to diverge is stored in a single integral substate. In order to compute this value, the iterative process described in Section 6.3.1 is implemented in listing 6.8 and is executed once for each pixel of the final image.

As `OpenCAL` uses OpenCAL-CL, source code and execution is divided in *host* and *device* (kernels) sub-parts as seen in Section 5.5.

Host side code is shown in listing 6.7. Note that `init` and `finalize` functions are omitted since already shown in listings 6.4 and 6.5. The application takes as an input parameter a configuration file that describes the size and the partitioning of the domain among the nodes and the devices of the cluster. It constructs a `Cluster` objects out of it using the `fromClusterFile` function. The MultiNode class is then created (line 13), and used to allocate (line 14) and eventually starts the executions (line 17).

```
 1 #define KERNEL_SRC "~/fractal2D/kernel_fractal2D/source/"
 2 #define KERNEL_INC "~/fractal/kernel_fractal2D/include/"
 3 #define KERNEL_LIFE_TRANSITION_FUNCTION "fractal2D_transitionFunction"
 4 
 5 struct CALSubstate2Di *Q_fractal;
 6 int main(int argc, char** argv){
 7         //create the cluster file from input parameter path
 8         string clusterfile;
 9         clusterfile = parseCommandLineArgs(argc, argv);
10         Cluster cluster;
11         cluster.fromClusterFile(clusterfile);
12         //declare and initialize a multinode object
13         MultiNode<decltype(init), decltype(finalize)> mn(cluster, world_rank, init, finalize);
14         mn.allocateAndInit();
15         //start crunching numbers
16         MPI_Barrier(MPI_COMM_WORLD);
17         mn.run(STEPS);
18         //a barrier and finalize functor are implicitly called here
19     return 0;
20 }
```

Listing 6.7: `OpenCAL-CLUST` kernel for the generation of Julia Set.

The run function takes care of splitting the domain and handle communication among the devices transparently. This means that at each iteration the code shown in Listing 6.8 is executed on each device and on each point of the grid assigned to it. Note that in this example, boundaries communication is not required, since no neighboring values are needed in order to compute the value of a pixel.

```
typedef double2  cl_complex;

#define DEVICE_Q_fractal (0)
#define MAXITERATIONS (5000)
#define SIZE (16384)
#define moveX (0)
#define moveY (0)
// Maps and zoom a pixel (x,y) to the complex plane
cl_complex convertToComplex(const int x, const int y, const double zoom,
const int DIMX, const int DIMY) {
    double jx = 1.5 * (x - DIMX / 2.0) / (0.5 * zoom * DIMX) + moveX;
    double jy = (y - DIMY / 2.0) / (0.5 * zoom * DIMY) + moveY;
    return (cl_complex)(jx, jy);
}
cl_complex juliaFunctor(const cl_complex p, cl_complex c) {
    const cl_complex c_ipow =
        cl_complex_multiply(&p, &p);
    return cl_complex_add(&c_ipow, &c);
}
//Returns the number of iteration taken to diverge to infinity
int evolveComplexPoint(cl_complex p, cl_complex c) {
    int it = 1;
    while (it <= MAXITERATIONS && cl_complex_modulus(&p) <= 10) {
        p = juliaFunctor(p, c);
        it++;
    }
    return it;
}
__kernel void fractal2D_transitionFunction(__CALCL_MODEL_2D) {
    calclThreadCheck2D();
    int i = calclGlobalRow() + borderSize;
    int j = calclGlobalColumn();

    const double zoom = 1.0;
    const cl_complex c;    c.x = -0.391;c.y = -0.587;

    int global_i = i - borderSize + offset;
    cl_complex p = convertToComplex(global_i, j, zoom, SIZE, SIZE);
    calclSet2Di(MODEL_2D, DEVICE_Q_fractal, i, j, evolveComplexPoint(p, c));
}
```

LISTING 6.8: `OpenCAL-CLUST` kernel for the generation of Julia Set.

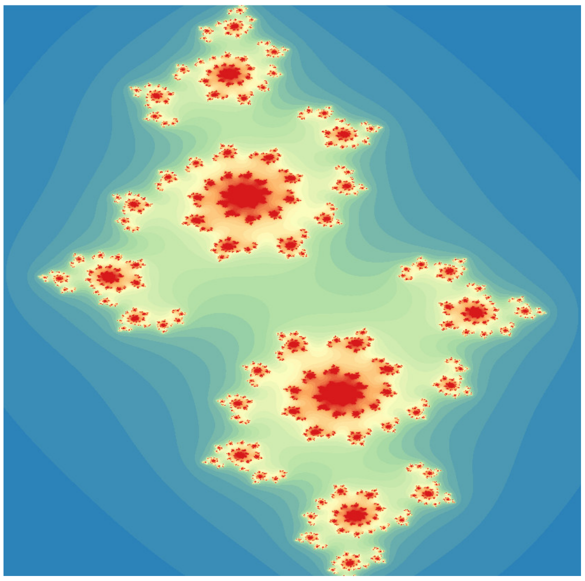


FIGURE 6.5: Julia set of size 1.07 GigaPixel, $\approx 3.22GB$ in the BMP uncompressed format. It is generated using `OpenCAL-CLUST` on two `NVIDIA GTX 980` and rendered on QGIS [138] interpreting each point value as color intensity in the spectral color map. Note that the Figure has been subsequently optimized and rescaled for book format.

6.3.3 *Convolutional Filters*

The example presented in this section is an application that applies the Sobel convolutional filter [139, 140] on a large 2D image. The code presented can be trivially extended to support any kind of convolutional filter also on a domain with more dimensions .

Convolution filtering is used to modify the spatial frequency characteristics of an image. Its name derives from the term *convolution* which is a general purpose filter. It is applied to each point of the domain and consists of determining the new value of the point by adding weighted values of all its neighbors together, as shown in Figure 6.7. Conovolution is performed multiplying the whole neighborhood of a point by a matrix, called the *kernel*

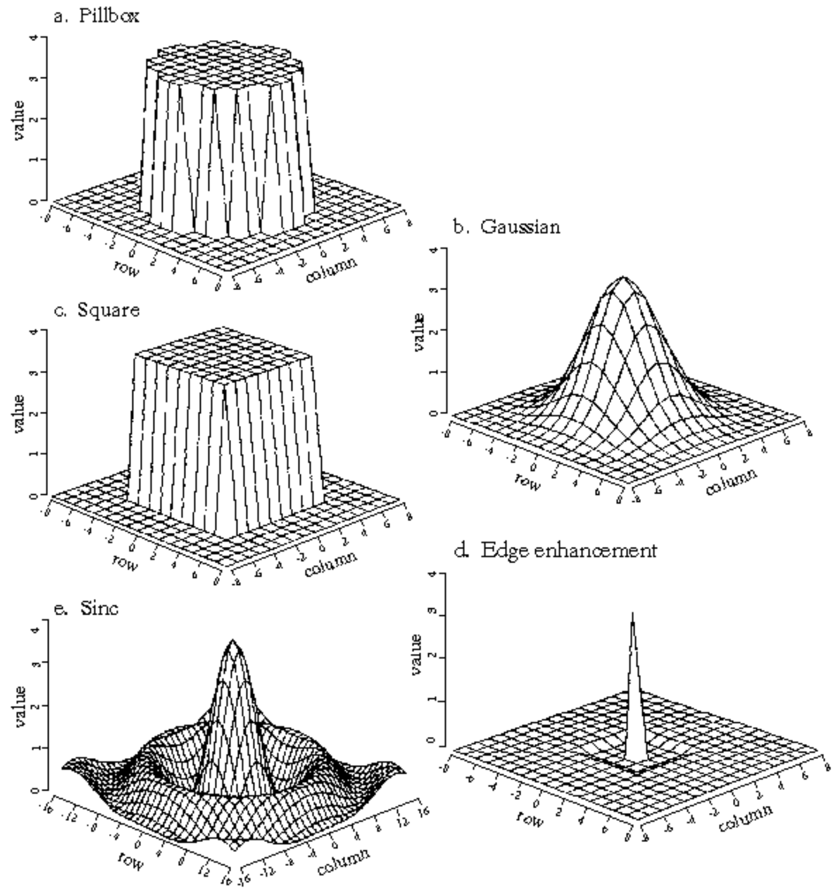


Figure 6.6: Common point spread functions. The Pillbox (a), Gaussian (b) and Square (c) are common smoothing, low-pass filters. Edge enhancement (d) is an example of high-pass filter.

of the convolution, which usually is a small square matrix of size r (the most common size for kernels is 3×3). Kernels coefficients correspond to point-wise values of an arbitrary fixed continuous function, called Point Spread Functions (PSF). Figure 6.6 depicts some of the most common PSF.

Convolution is very often used in image processing. Examples in this field are the Sobel's edge detection filter and the Gaussian Blur filters, that are shown in Figures 6.8 and 6.9.

Formally, convolution can be expressed by the following formula:

$$f'_{ij} = \sum_{i'=0}^{n} (\sum_{j'i'=0}^{m} f_{(i+i')(j+j')} \times d_{ij}) \tag{6.1}$$

where

- m, n are the vertical and horizontal size of the kernel,
- f_{ij} and f'_{ij} are the old and new value of the cell at coordinate (i, j),
- d_{ij} is the value of kernel at location (i, j)

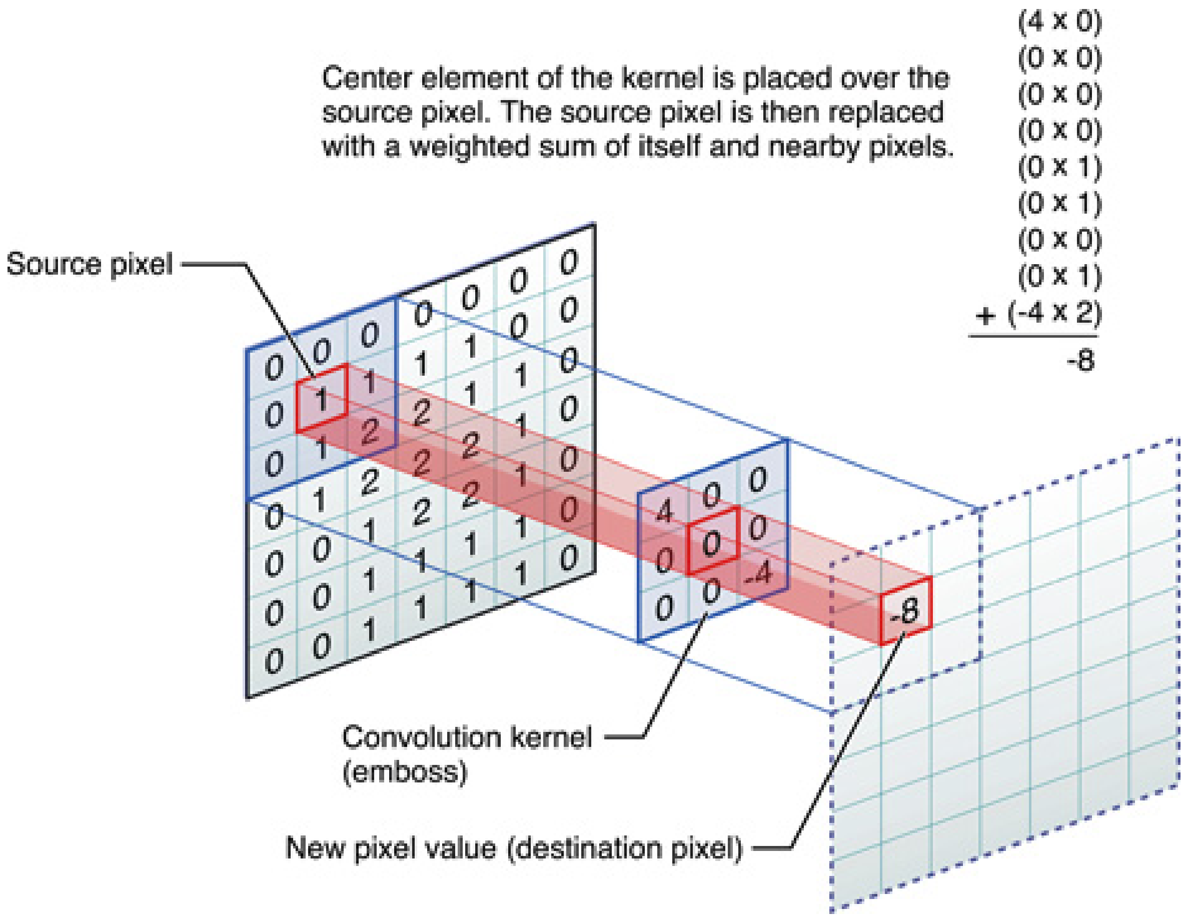
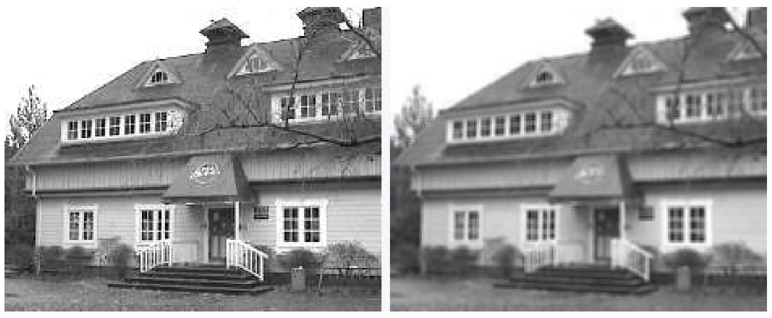
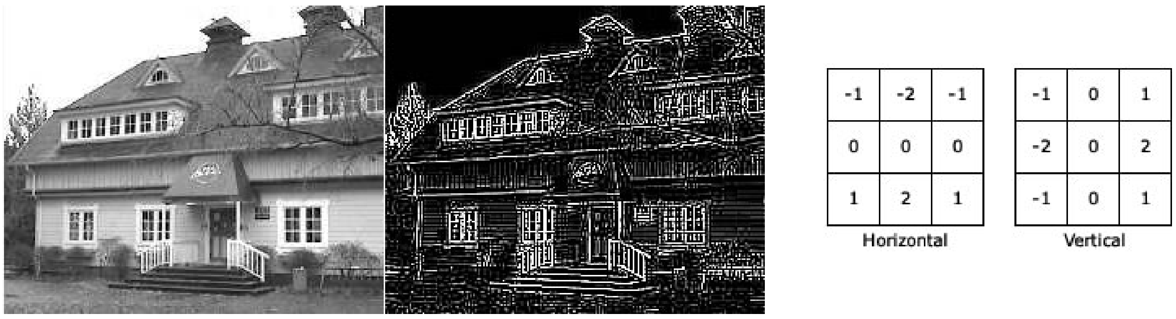

Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

FIGURE 6.7: The process of determining the new value of the central cell by applying a convolution matrix to its neighborhood.



0	0	0	5	0	0	0
0	5	18	32	18	5	0
0	18	64	100	64	18	0
5	32	100	100	100	32	5
0	18	64	100	64	18	0
0	5	18	32	18	5	0
0	0	0	5	0	0	0

FIGURE 6.8: Gaussian Convolution filter application. The emnployed Gaussian Kernel has radius 4.



-1	-2	-1
0	0	0
1	2	1

Horizontal

-1	0	1
-2	0	2
-1	0	1

Vertical

Figure 6.9: *Sobel* edge detection filter. The new pixel value is computed in two subsequent steps, horizontal and vertical, in order the final image to be less affected by noise.

6.3.3.1 *Edge Handling*

It is clear from Equation 6.1 that kernel convolution requires values from pixels outside the domain boundaries. There are a number of ways for handling these corner cases:

WRAP
: The image is conceptually treated as it was wrapped in a toroidal shape. `OpenCAL` natively deals with toroidal domain.

MIRROR
: The image boundaries are mirrored at the edges, meaning that if trying to read a pixel 2 units outside the edges, the returned value is the corresponding pixel 2 unit inside the edge instead.

CROP
: The final image does not contain pixel which would require values from beyond the edges. The output is smaller than the input because edges have been cropped out.

6.3.4 *Sobel Edge Detection `OpenCAL-CLUST` implementation*

Convolutional filtering is easily implemented in `OpenCAL-CLUST` in the following steps and has been applied to the image shown in Figure 6.10a:

1. Image channels are separately read by each `OpenCAL` process into `short` substates (using any image reading third part library, as `SOIL`[1], for instance).
2. A cluster file is defined for the image and shown in listing 6.10. A single node and 3 GPUs were employed in this example. Two `NVIDIA GTX980` and one `NVIDIA K40` each with an equal workload.
3. The kernels depicted in Figure 6.9 are applied to each pixel of the image. `OpenCAL` kernel code is shown in Listing 6.9. Boundaries are automatically handled and transferred between devices. When devices running on different nodes need to communicate then a MPI communication takes place.
4. The resulting image is written on disk and shown in Figure 6.10 (optimized for book format).

1 SOIL webpage: `http://www.lonesock.net/soil.html`

```
10800 21600
1
192.168.1.111 3
0 0 3600
0 1 3600
0 2 3600
```

Listing 6.10: Adopted cluster file for the Sobel filtering example. The image is decomposed equally among 3 devices.

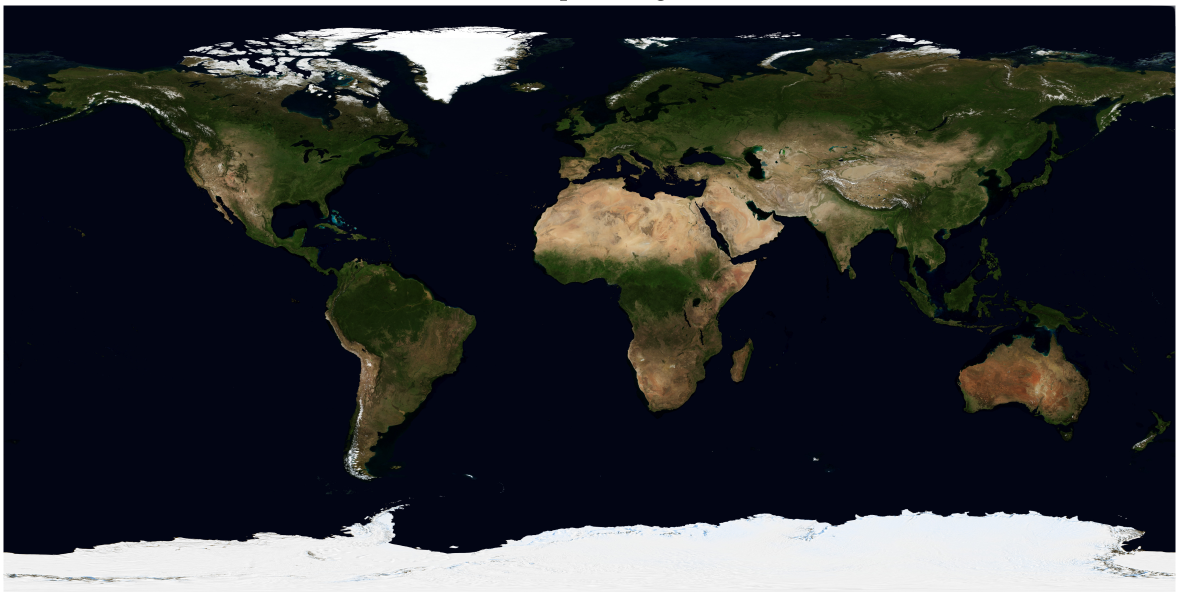
```
    #define DEVICE_Q_red (0)

    __kernel void sobel2D_transitionFunction(__CALCL_MODEL_2D) {

    calclThreadCheck2D();
    int i = calclGlobalRow() + borderSize;
    int j = calclGlobalColumn();
    int KX[3][3] = {
                    {-1, 0, 1},
                    {-2, 0, 2},
                    {-1, 0, 1}};

    int KY[3][3] = {
                    {1, 2, 1},
                    {0, 0, 0},
                    {-1, -2, -1} };

    int Gx,Gy,n,k,k1;
    Gx = Gy = n = 0;
    if (j > 0 && j < calclGetColumns() - 1)
        for (k = -1; k <= 1; k++)
                for (k1 = -1; k1 <= 1; k1++) {
                Gx += calclGet2Di(MODEL_2D, DEVICE_Q_red, i + k, j + k1) *
                                                        KX[k + 1][k1 + 1];
                Gy += calclGet2Di(MODEL_2D, DEVICE_Q_red, i + k, j + k1) *
                                                        KY[k + 1][k1 + 1];
            }
    const int P = sqrt(Gx * Gx + Gy * Gy);
    //set new pixel color for red channel
    calclSet2Di(MODEL_2D, DEVICE_Q_red, i, j, P);
    return;
}
```

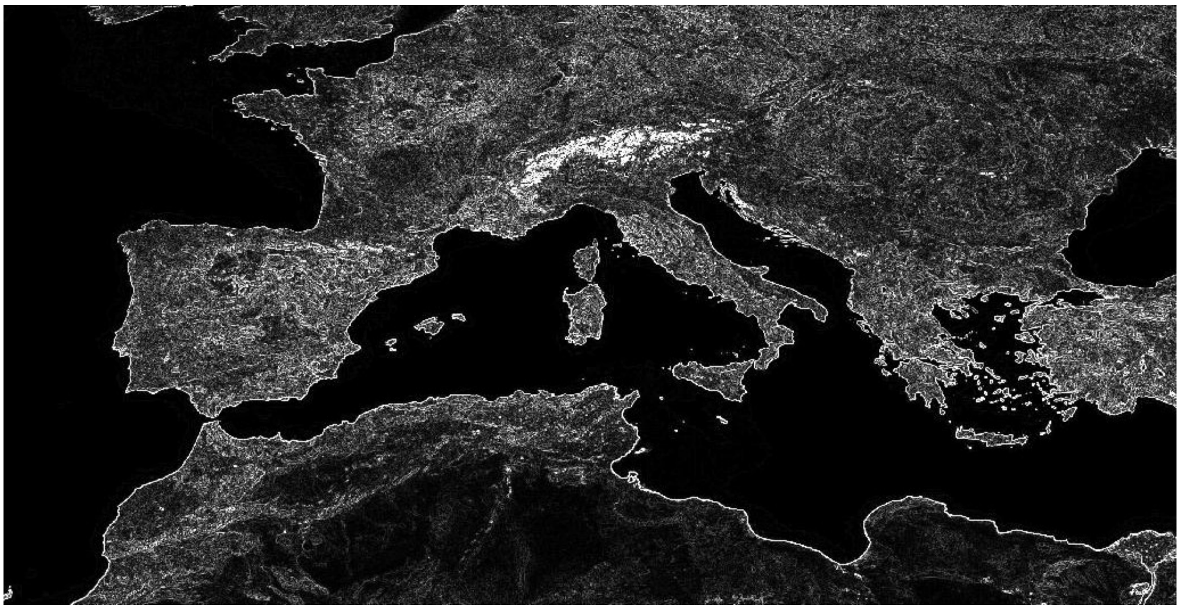
Listing 6.9: `OpenCAL` Sobel edge detection filter kernel. For the sake of simplicity

6.3.5 *SciddicaT*

This section briefly describes the `OpenCAL-CLUST` implementation of the *SciddicaT* landslide model introduced in Section 5.6.1. The aim of this section is to show that it is possible to deploy any model written in OpenCAL-CL to `OpenCAL-CLUST` on multiple nodes and accelerators very easily. The code for the implementation of sciddicaT shown in Section 5.6.3 is used in this section, with few lines added. The additional lines take care of the definition of a cluster object, as shown in Listing 6.11, during the initialization phase, as

(a) Input Image





(b) Zoomed cut on Europe and North Africa of the Output Image.

FIGURE 6.10: Input Image for the Sobel filter example shown in listing 6.9. Image size is 233 MegaPixel $\approx$ 700MB in BMP uncompressed format.

shown in Listing 6.12. Note that in this example no configuration file is used, as to show that is possible to set up a configuration launch programmatically.

```
void setUpParallelWork(Cluster& mn, const uint XDIM, const uint YDIM){
    //------node 1
    struct Node n1;
    struct Device d1_0 = {0,0,XDIM/4}; //NVIDIA  GTX980
    struct Device d1_1 = {0,1,XDIM/4}; //NVIDIA  K40
    struct Device d1_2 = {0,2,XDIM/4}; //NVIDIA  K40
    n1.devices.push_back(d1_0);
    n1.devices.push_back(d1_1);
    n1.devices.push_back(d1_2);
    //node workload's is the sum of its devices workloads
    n1.workload = d1_0.workload+d1_1.workload+d1_2.workload;
    n1.columns=C;
    n1.offset = 0;
    mn.nodes.push_back(n1);

    //------node 2
    struct Node n2;
    //remainder work to this device
    struct Device d2_0 = {0,0,XDIM/4+XDIM%4};//NVIDIA K20
    n2.devices.push_back(d2_0);
    n2.workload = d2_0.workload;
    n2.columns=C;
    //n2.workload starting from n1.workload
    n2.offset = n1.workload;

    mn.nodes.push_back(n2);
}
```

Listing 6.11: OpenCAL-CLUST sciddicaT Cluster defined programmatically during the init (see Section 6.2.1) phase.

```
void init( struct CALCLMultiGPU* multigpu , const Cluster* c){
Node mynode = c->nodes[rank];
auto devices = mynode.devices;
struct CALCLDeviceManager * calcl_device_manager = calclCreateManager();

calclSetNumDevice(multigpu,devices.size());
for(auto& d : devices){
    calclAddDevice(multigpu,calclGetDevice(calcl_device_manager, d.num_platform , d.num_device)
         ,  d.workload);
}

CALModel2D* host_CA;
host_CA = calCADef2D(mynode.workload, mynode.columns, CAL_VON_NEUMANN_NEIGHBORHOOD_2D,
     CAL_SPACE_TOROIDAL, CAL_OPT_ACTIVE_CELLS_NAIVE);

// Add substates
Q.f[0] = calAddSubstate2Dr(host_CA);
Q.f[1] = calAddSubstate2Dr(host_CA);
Q.f[2] = calAddSubstate2Dr(host_CA);
Q.f[3] = calAddSubstate2Dr(host_CA);
Q.z = calAddSubstate2Dr(host_CA);
Q.h = calAddSubstate2Dr(host_CA);

int my_readoffset, my_writeoffset=0;
my_readoffset = mynode.offset;
// Load configuration
calLoadSubstate2DrMulti(host_CA, Q.z, DEM_PATH,my_readoffset,my_writeoffset);
calLoadSubstate2DrMulti(host_CA, Q.h, SOURCE_PATH,my_readoffset,my_writeoffset);

// Initialization
sciddicaTSimulationInit(host_CA);
calUpdate2D(host_CA);


// Define a device-side CAs
calclMultiGPUDef2D(multigpu,host_CA,KERNEL_SRC,KERNEL_INC, 1, c->nodes.size() == 1);

// Extract kernels from program
calclAddElementaryProcessMultiGPU2D(multigpu, KERNEL_ELEM_PROC_FLOW_COMPUTATION);
calclAddElementaryProcessMultiGPU2D(multigpu, KERNEL_ELEM_PROC_WIDTH_UPDATE);

bufferEpsilonParameter = calclCreateBuffer(multigpu->context, &P.epsilon, sizeof(CALParameterr))
    ;
bufferRParameter = calclCreateBuffer(multigpu->context, &P.r, sizeof(CALParameterr));

calclAddSteeringFuncMultiGPU2D(multigpu,KERNEL_STEERING);
calclSetKernelArgMultiGPU2D(multigpu,KERNEL_ELEM_PROC_FLOW_COMPUTATION, 0,sizeof(CALCLmem), &
    bufferEpsilonParameter);
calclSetKernelArgMultiGPU2D(multigpu,KERNEL_ELEM_PROC_FLOW_COMPUTATION, 1, sizeof(CALCLmem), &
    bufferRParameter);
}
```

Listing 6.12: `OpenCAL-CLUST` sciddicaT model definition and launch configuration set up programmatically during the init (see Section 6.2.1) phase.

In this case, a configuration `Cluster` specifying two nodes and four devices in total is created. The four employed devices are:

- 2 NVIDIA K40
- 1 NVIDIA K20
- 1 NVIDIA GTX980

(see Appendix B for the technical specification of the accelerators).

The initialization phase is used here in order to allocate and initialize the library using the same approach shown in Listing 6.4 (lines 7-11).

The kernel side of the application remains also the same, with the exception that boundaries cells are not considered, and are only used in read-mode.

6.4 PERFORMANCE ANALYSIS

This section describes the computational results of `OpenCAL-CLUST` by considering the models presented in the previous sections tested on the following benchmarks:

1. **Julia Set** Generation, described in Section 6.3.1, representing a typical compute bound application (see Section 6.3.1).
2. **Convolutional Filter** application, described in Section 6.3.3, is an example of memory bound application (see Section 6.3.4).

3. Landslide numerical model **sciddicaT**, introduced and formally defined in Section 5.6.1 at page 79, and is used to benchmark the performance of OpenCAL-CLUST on a real life computational model. Its kernels are both memory and compute bounds (see Section 6.3.5).

Assessing the type of the limiting factor for performance in relatively short kernels as Julia Set generation and convolutional filter is relatively easy by considering the *instruction:byte* rate of the considered kernel. Each GPU is attached with a theoretical peak in memory and instruction throughputs [141]. Let I, in GInst/s, and M,in GB/s, be the peak instruction and memory throughput, respectively. The quantity $\frac{I}{M}$ is then called **balanced** *instruction:byte* ratio. This is the number of fp32 operations per byte that should be issued in order to obtain peak compute and bandwidth performance. For example, the GPU NVIDIA K40 has a theoretical instruction throughput of 715.2 GInstr/s and a theoretical bandwidth of 288 GB/s. Its theoretical instruction throughput is computed by considering the base clock of a single core, that is $F = 745$ MHz and their number, that is $N = 2880$. Assuming that a fp32 operation is completed with a latency of $L = 3$ cycles then the throughput is computed as follows:

$$\frac{\frac{F*2880}{1000}}{3} = \frac{745 * 2880}{3000} = 715.2$$

The balanced ratio for the K40 is then:

$$\frac{715.2}{288} = 2.83$$

If, compared to the *balanced* ratio for the device at hand, the *instruction:byte* ratio of a kernel is:

higher :
usually means that the kernel is **instruction/compute** bound, while if it is

lower :
usually mean that the kernel is **memory/bandwidth** bound.

The ratio for the Listing 6.8 has a value of roughly (assuming $MAXITERATION = 10000$, and cost for cl_complex_multiply and cl_complex_add is equal to 2 fp32 instructions): $\approx 40000 : 1 >> 2.48$. Therefore, Listing 6.8 is a compute bound kernel for the $K40$ GPU. For the same reasons we can conclude that kernel shown in Listing 6.9 is memory bound: its *instruction:byte* ratio for is $\approx 1 : 1$.

SciddicaT exposes both kind of bounds within its kernels. Some are memory bound, as for instance width_update while other are more compute intense as for instance flow_computation.

6.4.1 *Adopted Test Hardware*

All test are executed on two computing nodes (named *JPDM1* and *JPDM2*, respectively) interconnected via Gigabit Ethernet. Technical specification of both nodes can be found in Section B.0.2.

6.4.2 *Julia Set Generation*

Fractal Generation is an example of a perfectly parallelizable problem since the computation of each point of the grid does not require any communication and does not depend on

	GTX980	K40
Time(ms)	10908	2561
Speedup($\times$)	116	496

TABLE 6.2: Timings and speedups obtained on a single NVIDIA K40 and GTX980 for the small dataset.

the value of any other grid points other than itself. Moreover, the granularity of the work is small, making it a good candidate for GPU acceleration. The Julia generating function adopted is the following $z_{n+1} = z_n^2 + c$ where $c = -0.391 + -0.587i$. Each discrete point of the grid (x, y) $0 \leq x < S_x, 0 \leq y < S_y$ is mapped to the complex plane using the following mapping:

$$
\begin{aligned}
Re(z) =& \frac{3(x - \frac{S_x}{2})}{KS_x} \\
Im(z) =& \frac{2(y - \frac{S_y}{2})}{KS_y}
\end{aligned}
$$

where K is the zoom factor, S_x and S_y are the vertical and horizontal sizes, respectively, of the discrete computational grid.

Two domain sizes are considered:

- **small**, consisting of $12000 \times 12000 = 144 \times 10^6$ points, 10^3 iterations limit per step.
- **large**, consisting of $17000 \times 17000 = 289 \times 10^6$ points, 10^4 iterations limit per step.

It is worth to note that besides its much higher number of grid points, the *large* domain case performs $10\times$ more work (in a single step) per grid point than the *small* case, as the number of iteration limit is increased from 10^3 to 10^4.

Table 6.2 shows timings and speedup obtained on a single NVIDIA K40 and GTX980 on the *small* domain. As expected, speedup results are extremely positive, up to $\approx 500\times$, thanks to the great parallelizability of the problem on GPU architecture.

Table 6.3, Figures 6.12 and 6.11 show timings and speedups of the same application on two GPUs on the *small* domain. Note that since the two adopted devices have a substantial difference in hardware (see Table B.1) it is not easy to determine in advance the best workload for each of the GPU in order to obtain the best load balancing. For this reason, a number of experiments are performed in order to discover the optimal workloads for the considered GPUs and kernel. The best speedup is obtained when only 25% of the domain is assigned to the GTX 980. It can be seen that for this type of kernels, the $K40$ shows better performance, due to the fact that the K40 has a higher number of processing cores and shows a better divergence management than the GTX980. Moreover, it also worth to note that the Julia set kernel is highly divergent and that computational work is not homogeneously spread across the domain, as can be seen from Figure 6.5. Pixels colored in blue correspond to a small number of iterations of the loop in Listing 6.8, lines 23-26, while the ones colored in red to a higher number of iterations. Note that no communication whatsoever is performed during this application since the problem is embarrassingly parallel. The rightmost part of Figure 6.12a (from 9000 to 12000) shows a weird fluctuation in the

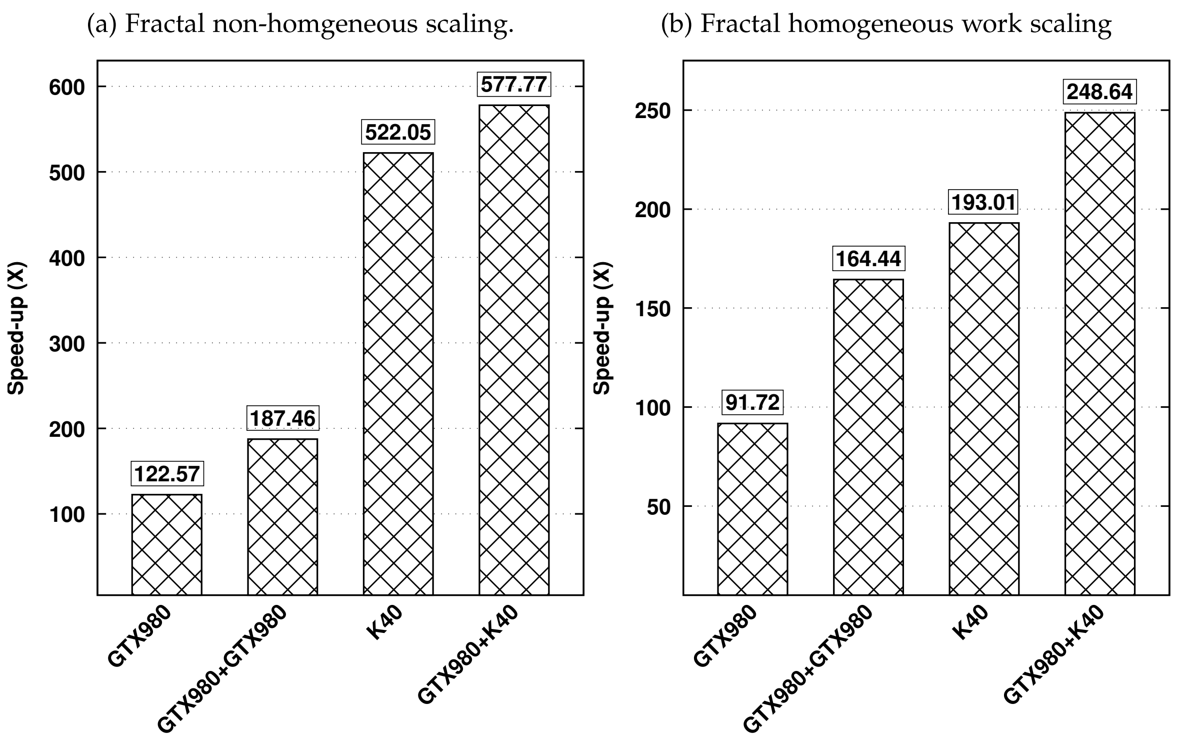


Figure 6.11: Fractal Generation speedups obtained on single-GPU execution (`K40`, `GTX980`) and multi-GPU execution on two devices (small dataset).

speedups, mainly due to the fact that pixels contained in rows from 1000 to 2000 correspond to *black* ones i.e. to pixels with almost no work attached to them. In order to show that in case of uniform work attached to each pixel the speedup curve behaves normally, a variation of Listing 6.8 is used and shown in Listing 6.13 where each execution of the kernel lasts for 1000 iterations. Figure 6.12b shows that fluctuation in this case are not present.

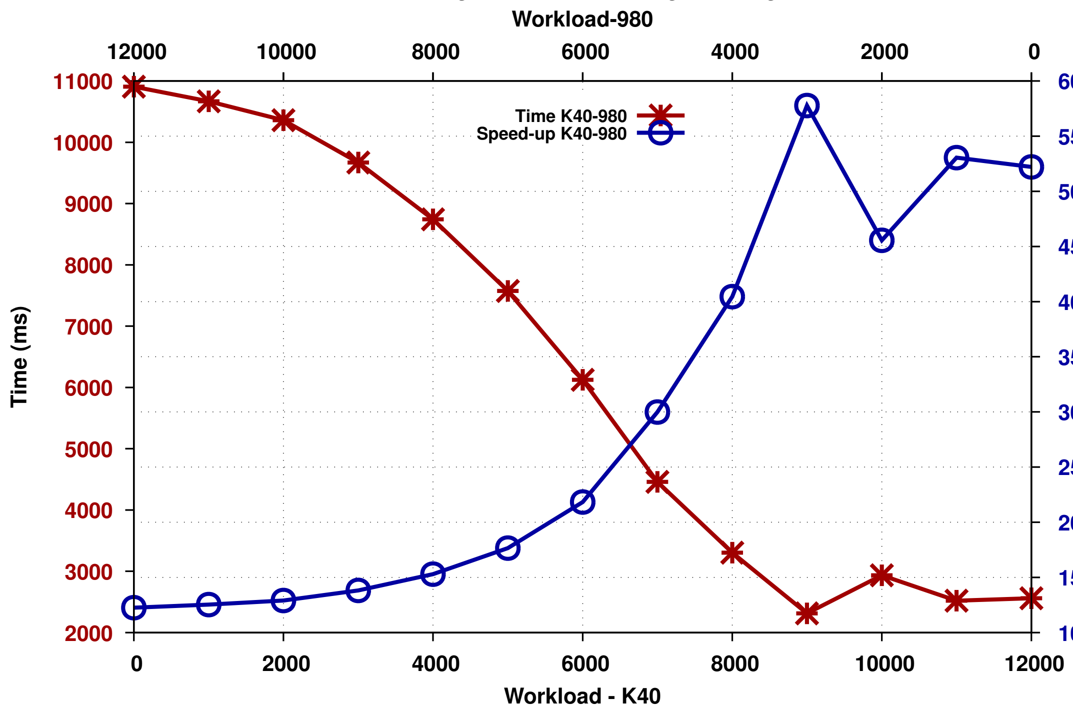
```
int evolveComplexPoint(cl_complex p,cl_complex c){
int it=1;
volatile cl_complex p1={p.x,p.y};
while(it++ <= 1000)
    p1=juliaFunctor(p1,c);
return 100;
}
```

Listing 6.13: `OpenCAL-CLUST` kernel for the generation of Julia Set. `volatile` keyword ensures that the object code for the while is emitted

Table 6.5 and Figure 6.13 show speedup and timings for the case where two identical GPUs `GTX980` are employed. It is not surprising that in this case best performance are achieved when the dataset is shared in an equal manner among the two devices. However, timing and speed-up are not perfectly symmetrical in this case, since one of the GPUs is attached with the (small) overhead of performing screen rendering.

Table 6.7 and Figure 6.14 show timing and speedup for the *large* dataset that is divided among the three employed devices. In this case, the workload not assigned to the *K*40 is equally divided among the two *GTX*980 as this is the case where best performance is achieved when two identical devices are adopted, as shown in Figure 6.13. As noted, achieved speedups are good, up to to $\approx$ 110×, when workload division is s.t. $\approx$ 20%

(a) Non Homogeneous work. High divergent code.



(b) Homogeneous work. No thread divergence.

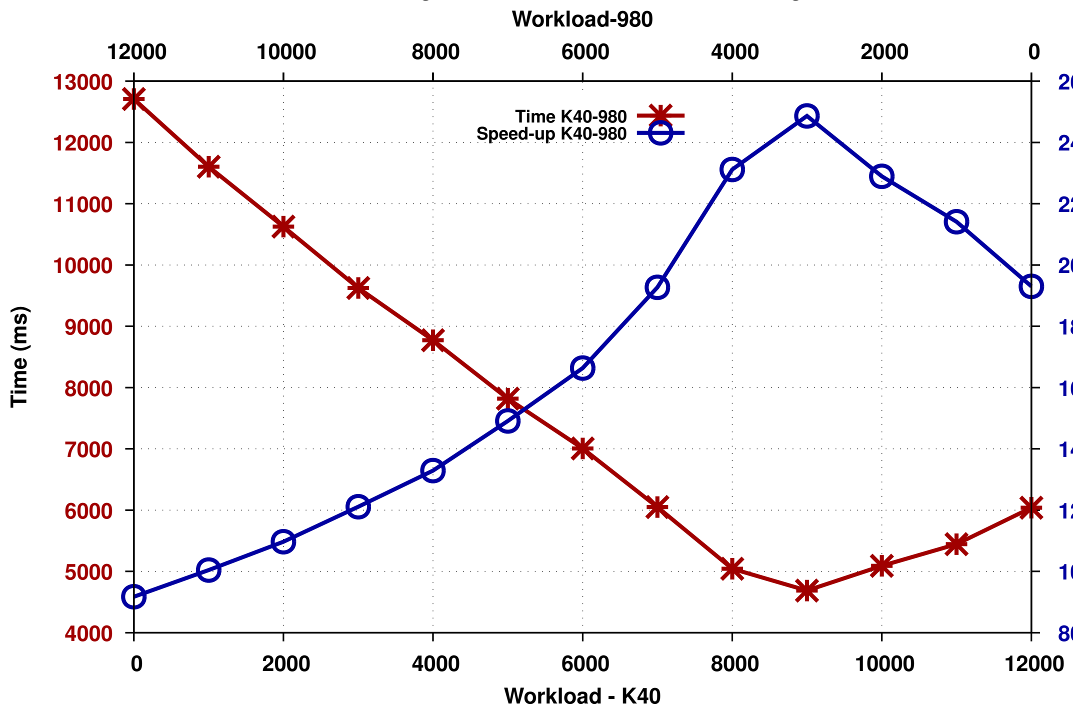
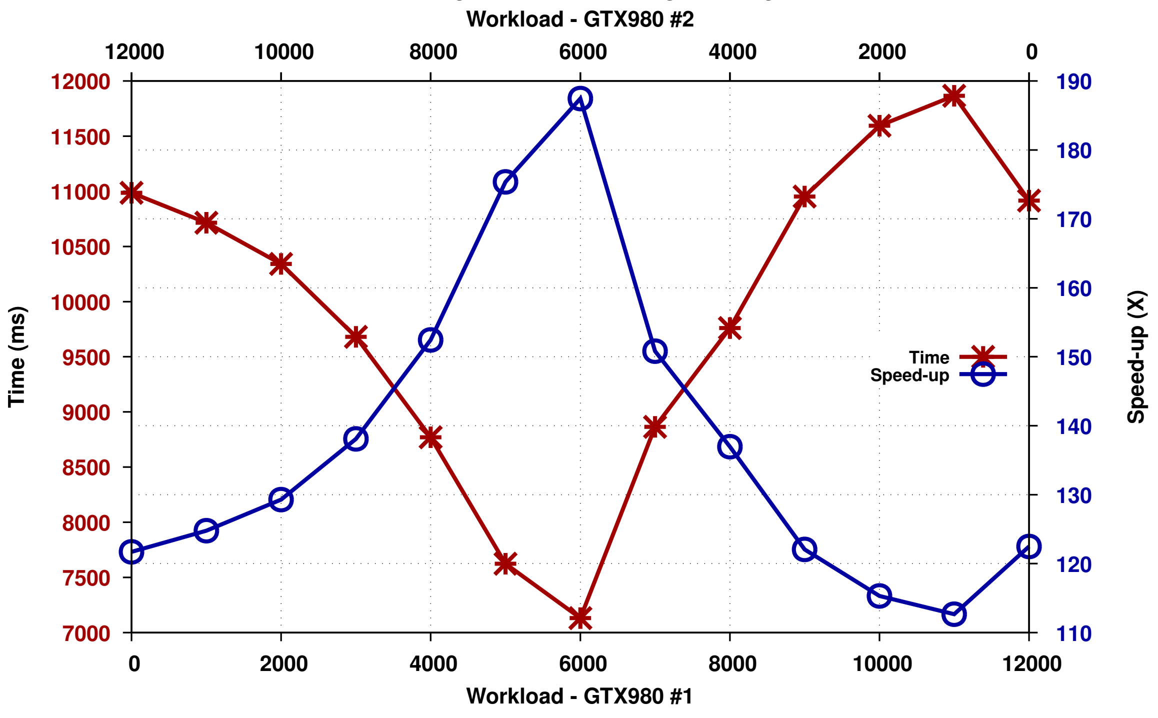


FIGURE 6.12: Time and Speed-up for the fractal generation *small* case on two different GPU: 1 GTX980 and 1 K40. The bottom and top horizontal axes indicate the amount of rows assigned to the K40 and GTX980, respectively. Note that Figure 6.12b depicts time and speedup values for the modified version of the kernel 6.8 that force homogeneous amount of work among all the grid points.

(a) Non Homogeneous work. High divergent code.



(b) Homogeneous work. No thread divergence.

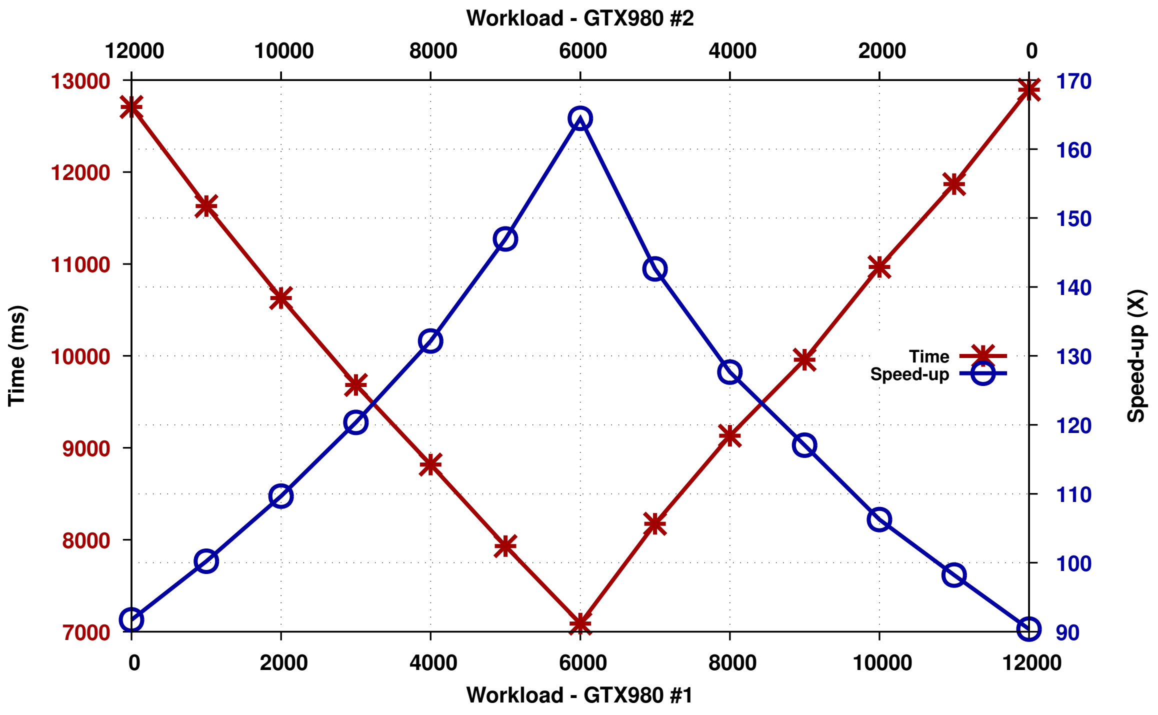
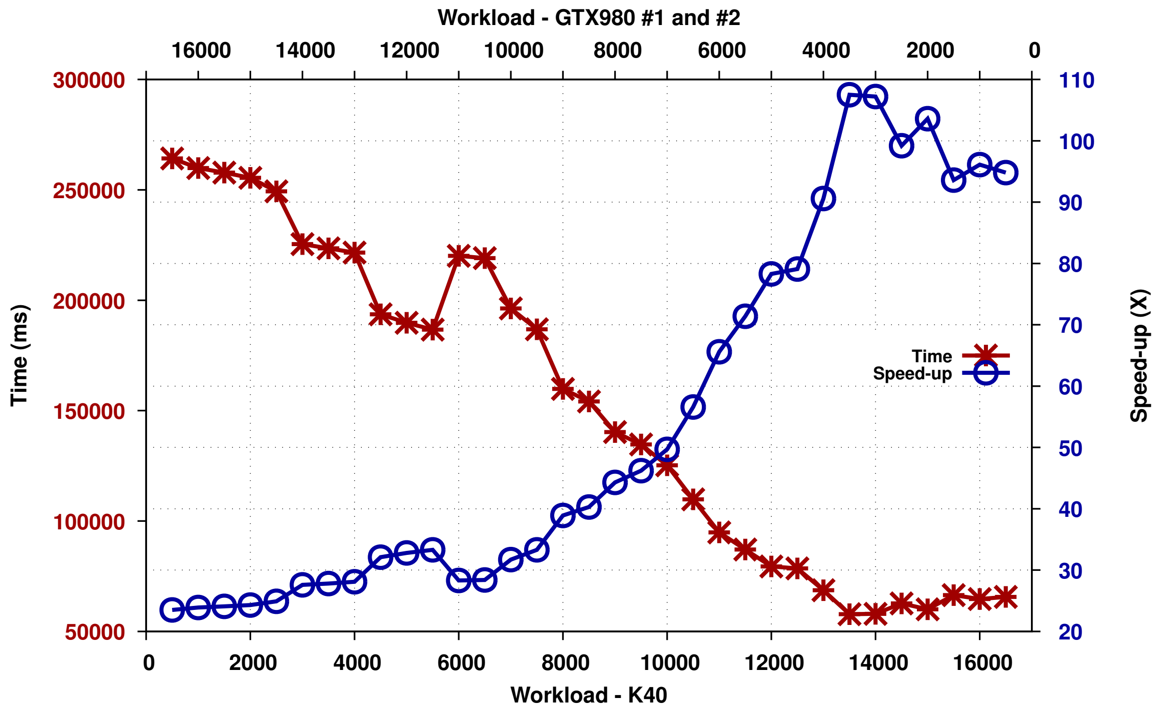
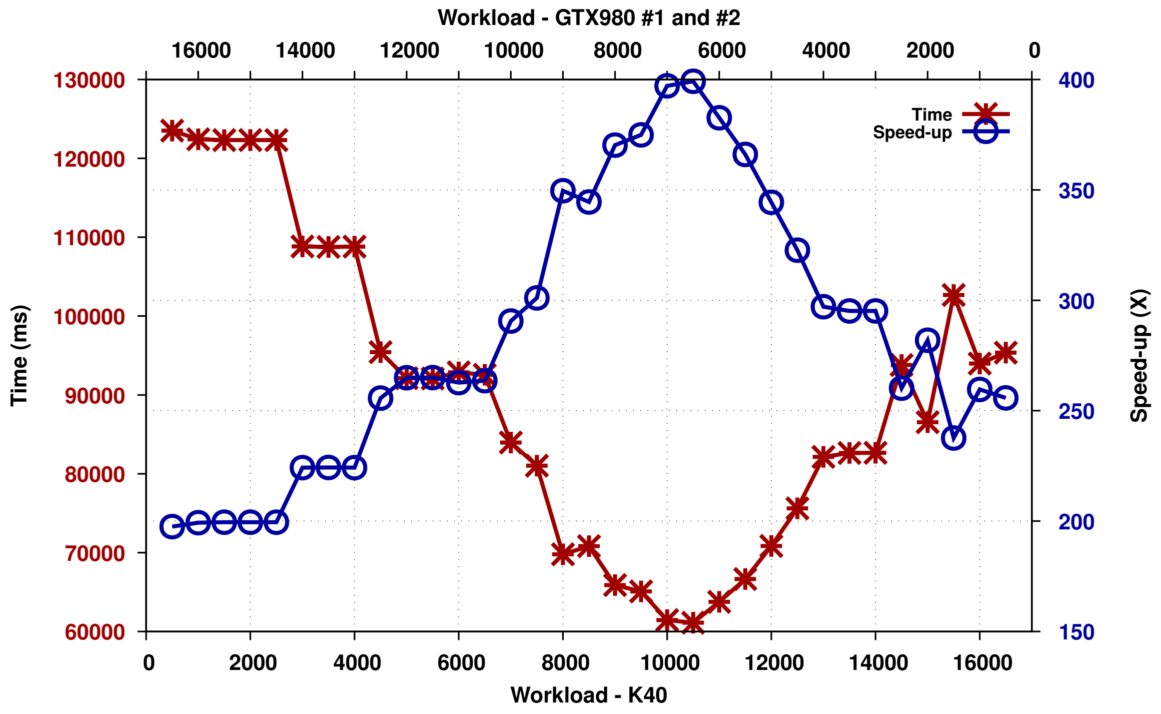


FIGURE 6.13: Time and Speed-up for the fractal generation *small* case on two GTX980. The bottom and top horizontal axes indicate the amount of rows assigned to the first and second `GTX980`, respectively. Figure 6.13a depicts time and speedup values for the modified version of the kernel reported in Listing 6.8 which forces homogeneous amount work among all the grid points. Note that in this case the graph is almost perfectly symmetrical. Time in red (lower is better), speed-up in blue (higher is better).

(a) Non homogeneous work case. Note that this is a code with an high value of thread divergence . A number 3 of GPUs are employed.





(b) Homogeneous work. No thread divergence.3 GPUs employed.

Figure 6.14: Timings and Speed-up for the fractal generation *large* case. (a) shows non-homogeneous work i.e. the real fractal, (b) the homogeneous one. Time in red (lower is better), speed-up in blue (higher is better).

(a) Real Fractal

Workload			
GTX980	K40	Time(ms)	Speedup(×)
0	12000	2561	522.05
1000	11000	2520	530.54
2000	10000	2935	455.52
3000	9000	2314	577.77
4000	8000	3305	404.53
5000	7000	4458	299.90
6000	6000	6123	218.35
7000	5000	7573	176.54
8000	4000	8744	152.90
9000	3000	9669	138.27
10000	2000	10359	129.06
11000	1000	10669	125.31
12000	0	10908	122.57

(b) Uniform number of iterations.

Workload			
GTX980	K40	Time(ms)	Speedup(×)
0	12000	6039	193.01
1000	11000	5444	214.10
2000	10000	5092	228.91
3000	9000	4688	248.63
4000	8000	5043	231.13
5000	7000	6049	192.69
6000	6000	7006	166.37
7000	5000	7819	149.07
8000	4000	8771	132.89
9000	3000	9625	121.10
10000	2000	10625	109.70
11000	1000	11603	100.45
12000	0	12708	91.72

Table 6.3: Timing and speedups for the fractal generation (small dataset) on aK40 and a GTX980. (a) and (b) are refereed to the real case and the uniform work scenarios, respectively. The Best speed-up case is highlighted in dark gray. The workload columns indicate the amount of rows assigned to each device.

(a) Real Fractal

Workload			
GTX980	GTX980	Time(ms)	Speedup(×)
0	12000	12896	90.38
1000	11000	11869	98.20
2000	10000	10968	106.27
3000	9000	9959	117.04
4000	8000	9131	127.65
5000	7000	8173	142.61
6000	6000	7088	164.44
7000	5000	7931	146.97
8000	4000	8819	132.17
9000	3000	9684	120.36
10000	2000	10629	109.66
11000	1000	11630	100.22
12000	0	12708	91.72

(b) Uniform number of iterations.

Workload			
GTX980	GTX980	Time(ms)	Speedup(×)
0	12000	10916	122.48
1000	11000	11866	112.67
2000	10000	11595	115.30
3000	9000	10953	122.06
4000	8000	9761	136.97
5000	7000	8865	150.81
6000	6000	7132	187.46
7000	5000	7624	175.36
8000	4000	8770	152.45
9000	3000	9681	138.10
10000	2000	10341	129.29
11000	1000	10715	124.77
12000	0	10986	121.70

Table 6.5: Timing and speedups for the fractal generation (small dataset) on two identical GTX980. (a) and (b) are refereed to the real case and the uniform work scenarios, respectively. The Best speed-up case is highlighted in dark gray.

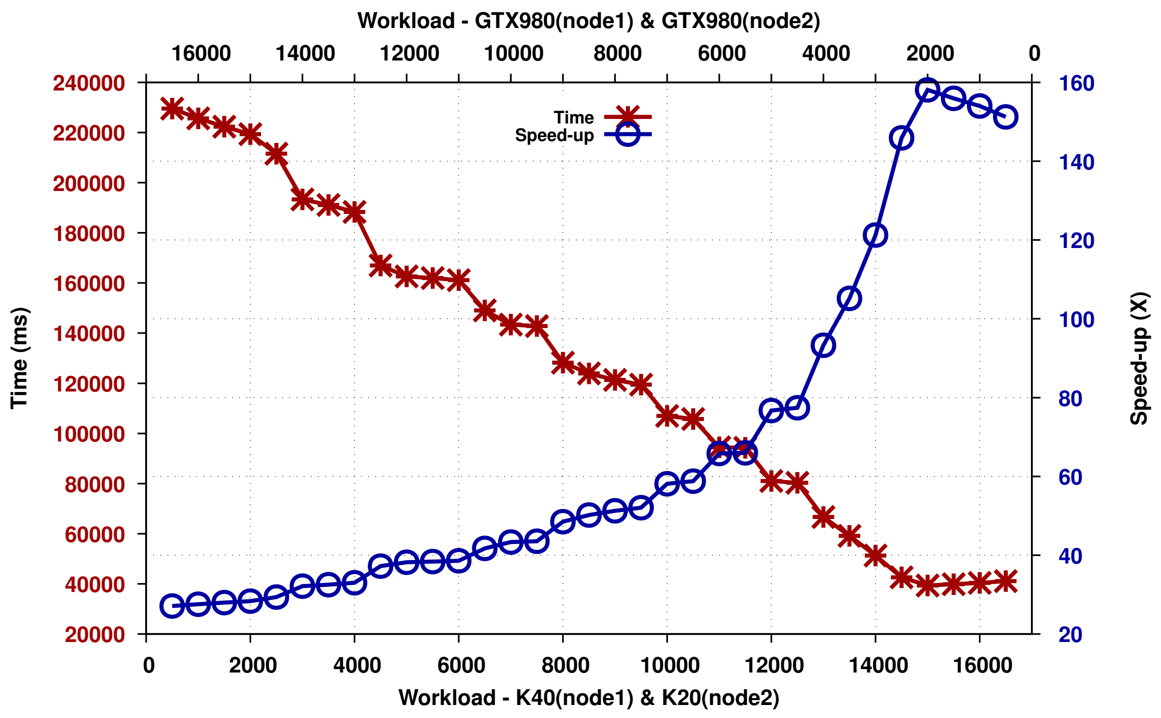


FIGURE 6.15: Timing and speedups for the multinode test of the fractal generation (large dataset, real case, non homogeneous work) on a total of 4 devices: a `GTX980` and a `K40` on *node 1* and a `GTX980` and a `K20` on *node 2*. The workload on the x dimension is shared among two devices: $K40$ and $K20$ while the remaining portion of the domain among the two $GTX980$. Time in red (lower is better), speed-up in blue (higher is better).

of the total work is shared among the two `GTX980` (see Figure 6.14a). This is due to the non-homogeneous distribution of the work along the domain and to the high number of divergent threads. In fact, when homogeneity of work if forced, i.e. all threads perform the same number of iterations, the best performance are obtained assigning $\approx 40\%$ of the domain to the two `GTX980` (see Figure 6.14b).

Finally, Figure 6.15 shows an example of execution of this benchmark (**large** dataset, non-homogeneous work) on an experimental cluster, which technical specification can be found in Appendix B, composed of two nodes interconnected via GigaBit Ethernet for a total of 4 accelerators (2 `GTX980`, 1 `K40` and 1 `K20`). It can be seen that this leads to a $\approx +50\times$ speed-up improvement w.r.t. the **3 GPU single node** execution depicted in Figure 6.14a, reaching a peak improvement of $158\times$ w.r.t. the **serial execution**. The workload specified on the label of the x-dimension of Figure 6.15 is shared among two devices i.e. the $K40$ and $K20$ while the remaining portion of the domain among the two $GTX980$s. This means that peak performance are achieved when a portion corresponding to 44% of the domain is assigned to each K-type GPU and only the 6% to each GTX.

6.4.3 *Convolutional Filtering*

In contrast to fractal generation, convolutional filtering is not a perfectly parallelizable problem since grid points on the boundaries requires values from grid points that reside on different GPUs. Here, the ratio of *instruction:byte* is usually low (it can vary depending on the convolutional kernel adopted), meaning that the problem is low compute-intensity

(a) Real Fractal.

WORKLOAD				
	GTX			
K40	#1	#2	TIME(ms)	SPEEDUP(×)
16500	250	250	65617	94.8
15500	750	750	66438	93.6
14500	1250	1250	62685	99.2
13500	1750	1750	57851	107.5
12500	2250	2250	78570	79.1
11500	2750	2750	87098	71.4
10500	3250	3250	109818	56.6
9500	3750	3750	134713	46.2
8500	4250	4250	154125	40.3
7500	4750	4750	186840	33.3
6500	5250	5250	219067	28.4
5500	5750	5750	186668	33.3
4500	6250	6250	193661	32.1
3500	6750	6750	223505	27.8
2500	7250	7250	249335	24.9
1500	7750	7750	257837	24.1
500	8250	8250	264234	23.5

(b) Uniform number of iterations.

WORKLOAD				
	GTX			
K40	#1	#2	TIME(ms)	SPEEDUP(×)
16500	250	250	95371	255.8
15500	750	750	102673	237.6
14500	1250	1250	93783	260.2
13500	1750	1750	82648	295.2
12500	2250	2250	75620	322.7
11500	2750	2750	66651	366.1
10500	3250	3250	61123	399.2
9500	3750	3750	65084	374.9
8500	4250	4250	70829	344.5
7500	4750	4750	81022	301.1
6500	5250	5250	92560	263.6
5500	5750	5750	92070	265.0
4500	6250	6250	95433	255.7
3500	6750	6750	108764	224.3
2500	7250	7250	122306	199.5
1500	7750	7750	122318	199.5
500	8250	8250	123519	197.5

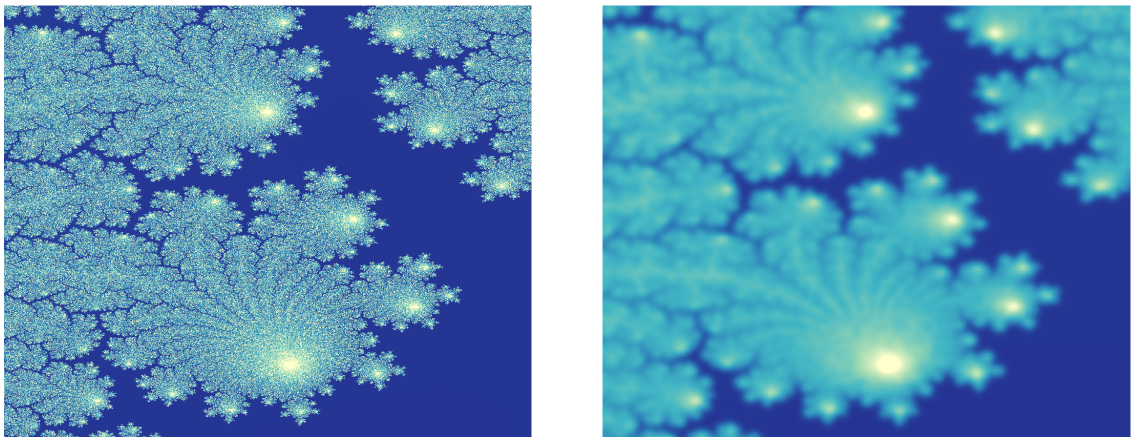
TABLE 6.7: Timing and speedups for the fractal generation (small dataset) on three devices: a pair of `GTX980` and a `K40`. (a) and (b) are refereed to the real case and the uniform work scenarios, respectively. The Best speed-up case is highlighted in dark gray.

and latency/bandwidth bound. Two filtering operations are taking in consideration for assessing the performance of `OpenCAL-CLUST` on this kind of algorithm:

sobel edge detection :
Described in Section 6.3.3 and which kernel is shown in Listing 6.9,

gaussian blur :
that consists of a uniform matrix with coefficients equal to $\frac{1}{9}$. At each step of the the application change the value of a grid point with the average of its 8 immediate neighboring points. It has a *instruction:byte* ratio ≈ 1. See Figures 6.16 and 6.8. Listing 6.14 shown the actual kernel code which implements this filter.

Both filters are applied on a computational grid of size $15000 \times 15000 = 225 \times 10^6$ cells for a number of 100 steps. The image upon which the filter are applied are generated using the `OpenCAL` fractal application described in Sections 6.3.1.



(a) Raw detail from a rendered image of a Julia Set.

(b) Gaussian Blur rendered version of Figure 6.16a

FIGURE 6.16: Raw and blurred detail of the of the image upon which the `OpenCAL-CLUST` convolutional filter application is ran upon. Rendering of fractals are obtained using QGIS software [138].

```
int convolutional_gaussian_blur(cl_complex p,cl_complex c){
    int sum=0,n=0;
    const int sizeOfX_ =calclGetNeighbor\right oodSize();
    for(; n < sizeOfX_; ++n)
        sum+= calclGetX2Di(MODEL_2D,DEVICE_Q_red, i, j, n);
    calclSet2Di(MODEL_2D, DEVICE_Q_red, i, j,sum/sizeOfX_);
}
```

LISTING 6.14: `OpenCAL` kernel implementing the Gaussian blur convolutional filter.

Table 6.9 and Figure 6.18 shows performances for the *sobel* filter on two (6.18a) and three (6.18b) devices respectively. Note that in this case, the disparity in performance between the K40 and the `GTX980` is not evident: in-fact, best performance behavior is obtained when the domain is shared in an equal manner among the devices. It is worth to note that the speedup scales linearly as the number of GPUs increases, as a $\approx 28\times$ speed-up is obtained with 1 GPU, $\approx 56\times$ with 2 GPUs and $\approx 82\times$ when 3 devices are employed as shown in Figure 6.17b.

Similar results are obtained for the *Gaussian blur* kernel as can be seen from Figure 6.19. The *blur* kernel is even more memory bound than *sobel*, which explains why performances are lower w.t.r to the Sobel experiments. Note that the speedup scales super-linearly as the number of GPUs increases as a $\approx 10\times$ speed-up is obtained with 1 GPU, $\approx 36\times$ with 2 GPUs and $\approx 53\times$ when 3 devices are employed as shown in Figure 6.17a.

Eventually, Figure 6.20 shows an execution of the *Sobel* benchmark on the test cluster specified in Appendix B. As can be seen, good speedups are achieved even in the presence of network communication between nodes. Figure 6.20a reports timing and speedup for an execution of two nodes where *node 1* and *node 2* are equipped with a `GTX980` and a `K40`, respectively. Obtained performance are comparable to the two GPU single node case depicted in Figure 6.18a. Figure 6.20b reports timing and speedup for an execution on two different nodes each equipped with two accelerators (a total of 4, adopting the same configuration adopted for the Julia Set Generation benchmark). In this case, $22\times$ and $\approx 103\times$ improvements are obtained w.r.t. **the single node** 3 **GPU** (reported in Figure 6.17b)

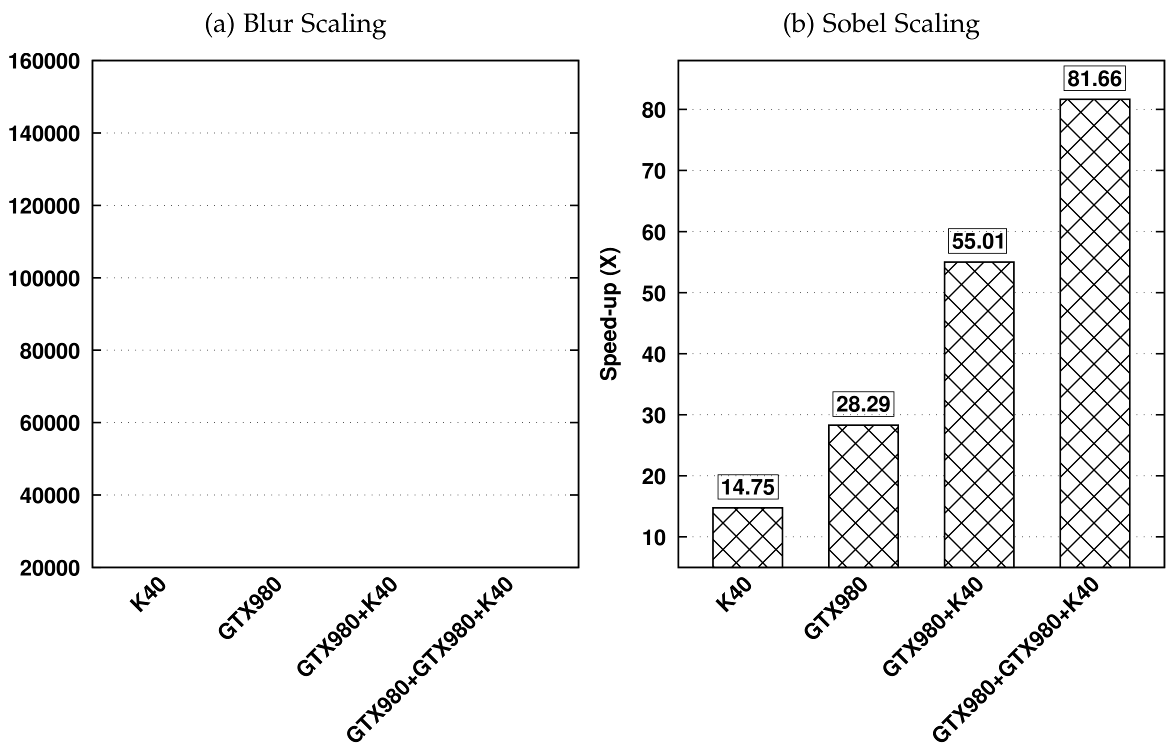


FIGURE 6.17: **Blur** and **Sobel** Speed-up benchmark scales linearly with the number of GPUs.

and the **serial** executions, respectively when ≈ 25% of the domain is assigned to each of the devices involved.

6.4.4 *The `sciddicaT` landslide model*

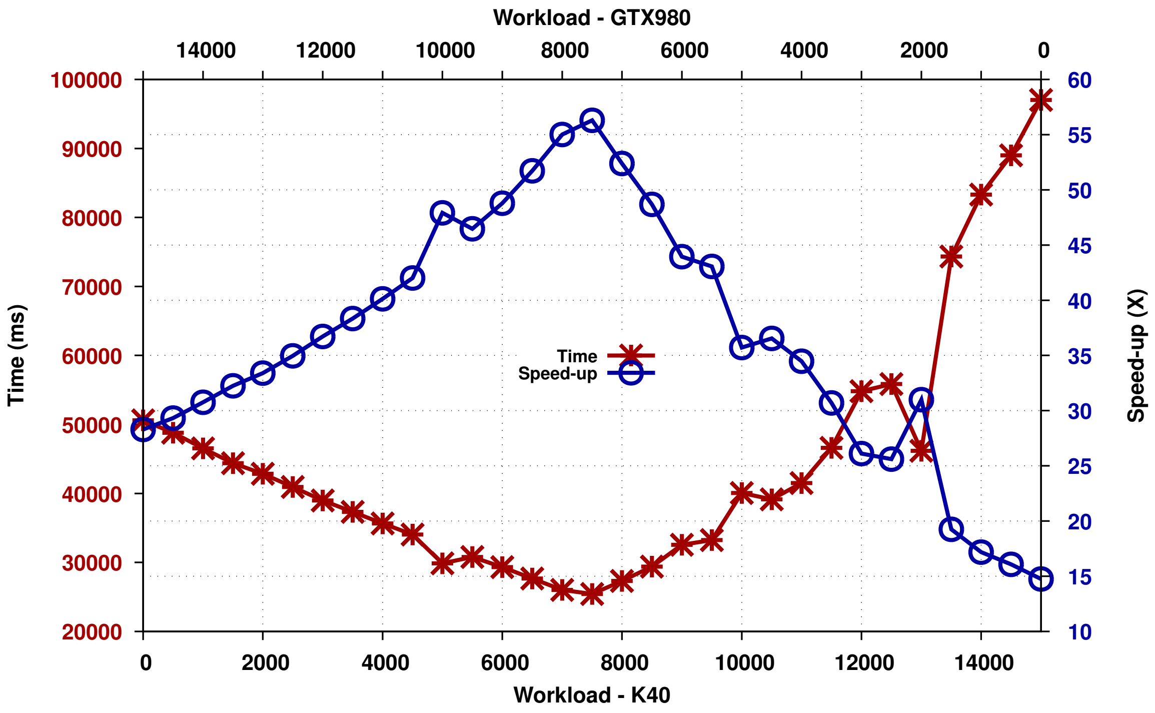
`sciddicaT` is an example of a real world model with both compute and memory bounded kernels (see Section 5.6.1). `sciddicaT` parallelization across several accelerators requires the communication of boundary cells residing on different GPUs. Despite three benchmarks are considered for performance evaluation of `OpenCAL` as shown in Section 5.9, only two benchmarks are here considered:

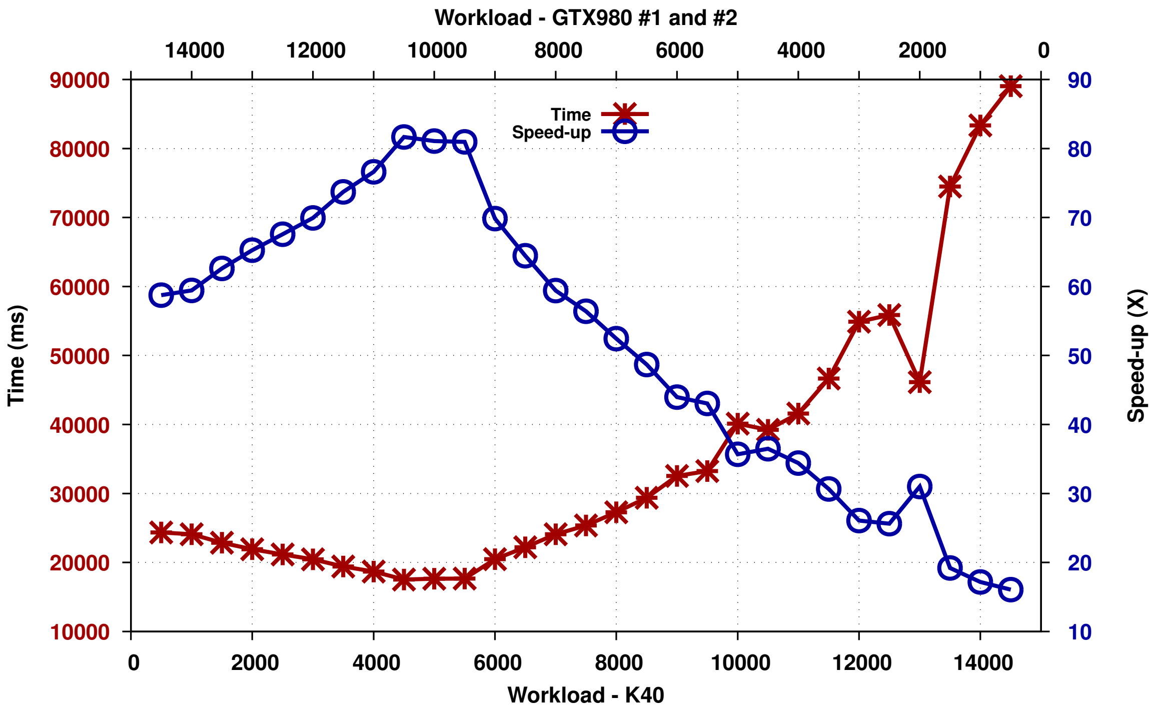
standard
the **small** standard domain of size 610 × 496 cells (see section 5.9.1)

computational domain stress test
the **large** domain of size 3593 × 3730 cells (see section 5.9.3)

Figure 6.21 shows that good absolute speed-up performance, for the large dataset, are obtained when multiple devices are utilized, up to ≈ 25×. It is worth to note that in this case the `GTX980` shows better performances than the *K*40 (9.7× vs 5.5× respectively) which explains also the reason why `sciddicaT` runs faster on a pair of `GTX980` than the case where one of them is replaced by a `K40`. This is mainly due to the fact that memory bounded kernels are in `sciddicaT` the ones that make up for most of the execution time, and it has been shown in Section 6.4.3 that the *GTX*980 performs better executing such latency bound kernels. Despite the good performance obtained in the **large** dataset, performance for the **standard** dataset are worse when adopting multiple devices than a single GPU

(a) Performance on 2 GPUs: `NVIDIA K40` and 1 `GTX980`.





(b) Performance on 3 GPUs: `NVIDIA K40` and a pair of `GTX980`.

Figure 6.18: *Sobel* benchmark performance metrics on two (a) and three (b) GPUs. Time in red (lower is better), speed-up in blue (higher is better).

(a) Performance on 2 GPUs: `NVIDIA K40` and 1 `GTX980`.

Workload			
K40	GTX980	Time(ms)	Speedup(×)
15000	0	97028	14.75
14000	1000	83304	17.18
13000	2000	46178	30.99
12000	3000	54814	26.11
11000	4000	41506	34.48
10000	5000	40067	35.72
9000	6000	32556	43.96
8000	7000	27314	52.39
7000	8000	26013	55.01
6000	9000	29330	48.79
5000	10000	29861	47.92
4000	11000	35657	40.13
3000	12000	38970	36.72
2000	13000	42851	33.39
1000	14000	46527	30.76
0	15000	50583	28.29

(b) Performance on 3 GPUs: `NVIDIA K40` and a pair of `GTX980`.

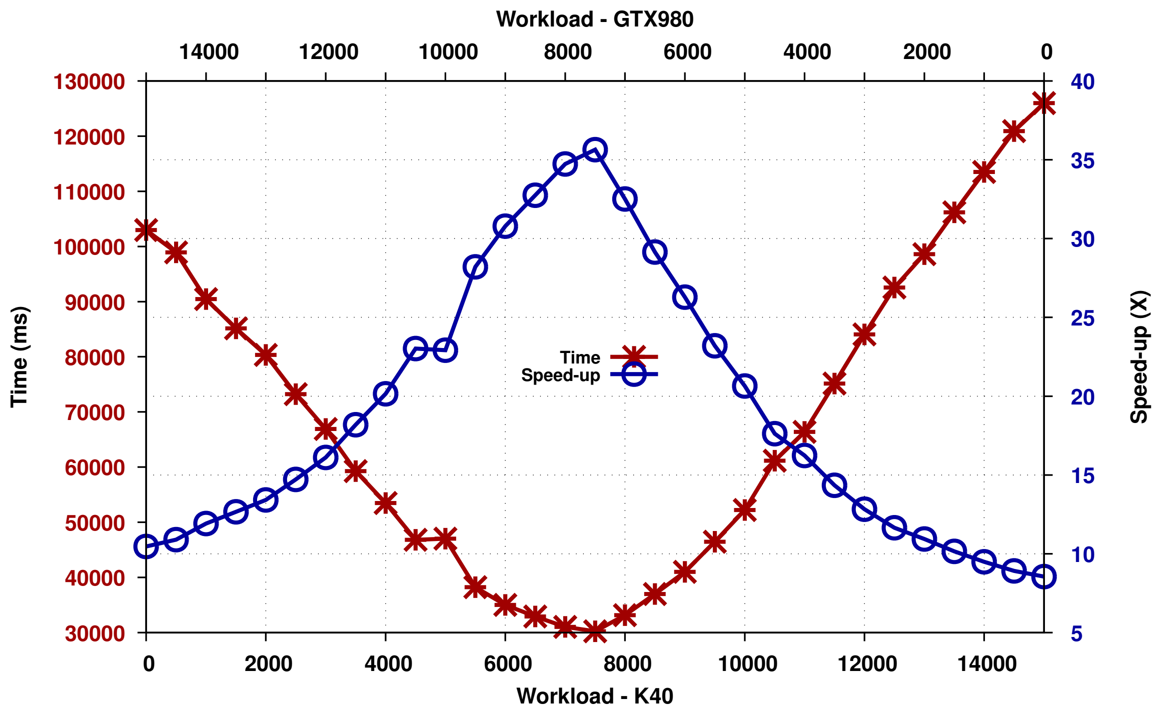
Workload				
	GTX			
K40	#1	#2	Time(ms)	Speedup(×)
14500	250	250	89033	16.07
13500	750	750	74482	19.21
12500	1250	1250	55861	25.61
11500	1750	1750	46666	30.66
10500	2250	2250	39233	36.47
9500	2750	2750	33240	43.05
8500	3250	3250	29388	48.69
7500	3750	3750	25364	56.41
6500	4250	4250	22197	64.46
5500	4750	4750	17678	80.94
4500	5250	5250	17524	81.65
3500	5750	5750	19415	73.70
2500	6250	6250	21177	67.57
1500	6750	6750	22856	62.60
500	7250	7250	24364	58.73

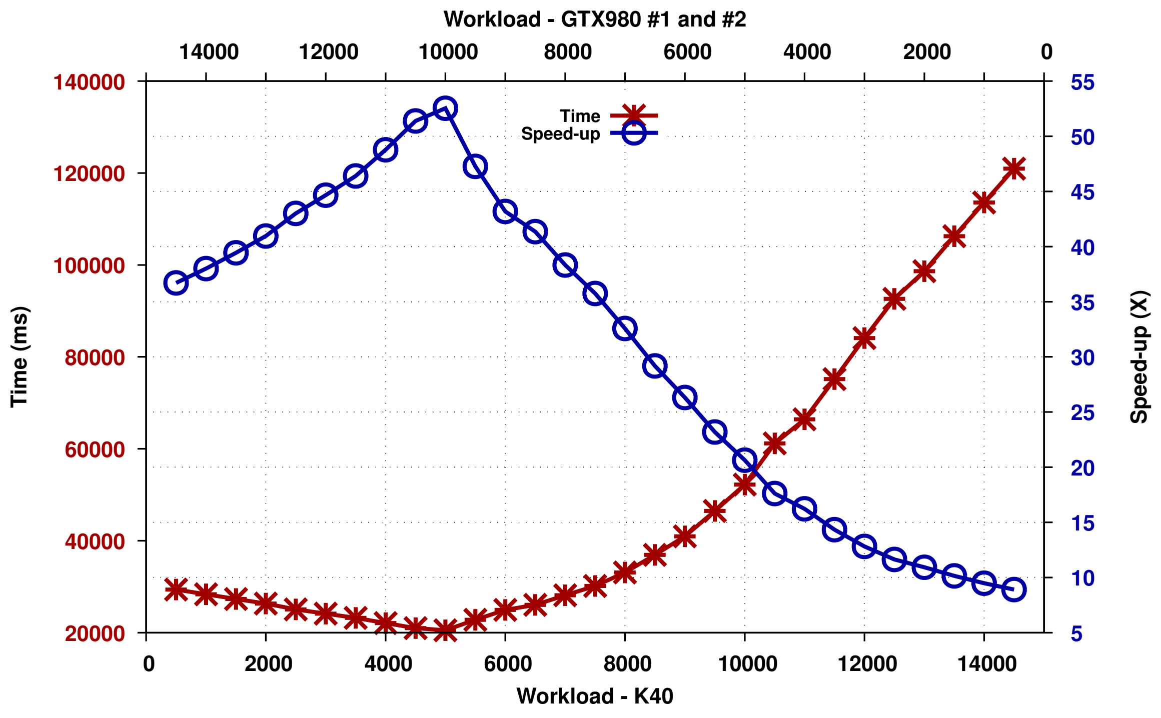
Table 6.9: *Sobel* benchmark performance metrics on two (a) and three (b) GPUs. Best speed-up case is highlighted in dark gray.

configuration. The main reason is that the *standard* test (see Section 5.9.1) has a relatively small number of cells (≈ 300000) making the efforts of further parallelization on several accelerators unnecessary since the communication overhead is dominant in this case. As it is known GPUs need a large number of threads in order to hide the intrinsic latency of the memory controller due to such latency/memory bound kernels. Figures 6.21a and 6.22 shows that a negative speedup (with the respect to a **single GPU execution**) are achieved. In particular, when adopting a `K40` and a `GTX980` $-1.5\times$ slowdown is achieved at best. Figure 6.24b and 6.24a show timing and speed-ups metrics for the large dataset, for the case where a pair of `GTX980` and a `K40` and a `GTX980` are adopted, respectively. The former shows that slightly super-linear speedup is achieved when an equal amount of work is assigned to each device while the latter configuration achieves best performance when only $\approx 30\%$ of the domain is assigned to the `K40` reaching a peak of $\approx 15\times$ speedup. Figure 6.24c shows timing and speed-up in the case where 3 GPUs are adopted. In this case best performance are achieved when only $\approx 15\%$ of the domain is assigned to the `K40` obtaining a speedup of $\approx 25\times$.

The last experiments concern the execution of this benchmark on the test cluster on both the small and large datasets considering the *naive* and the *active cells* versions. The small dataset is scattered equally (a total of 300×496 cells assigned to each device) among two `GTX980`s each located on a different node. As expected the **small** dataset incurs in a high communication overhead giving rise to worsening performance w.r.t. the *single node single GPU* configuration with a $\approx 5\times$ slowdown. Measured communication and computation

(a) Performance on 2 GPUs: `NVIDIA K40` and 1 `GTX980`.



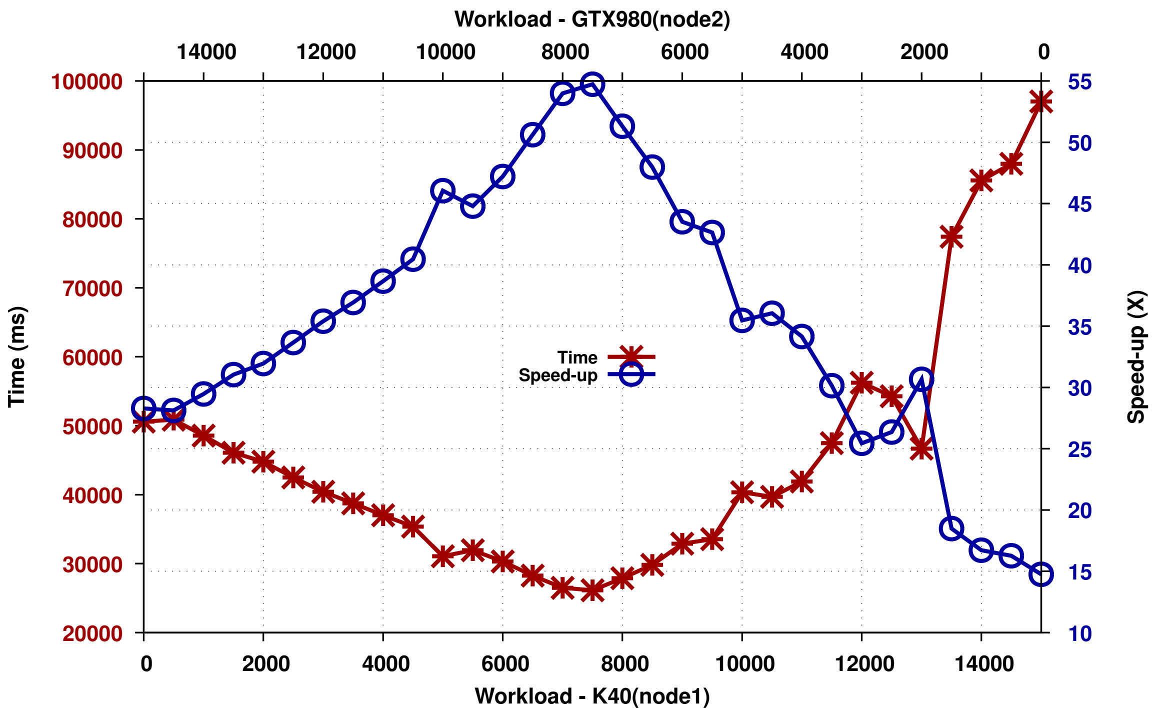


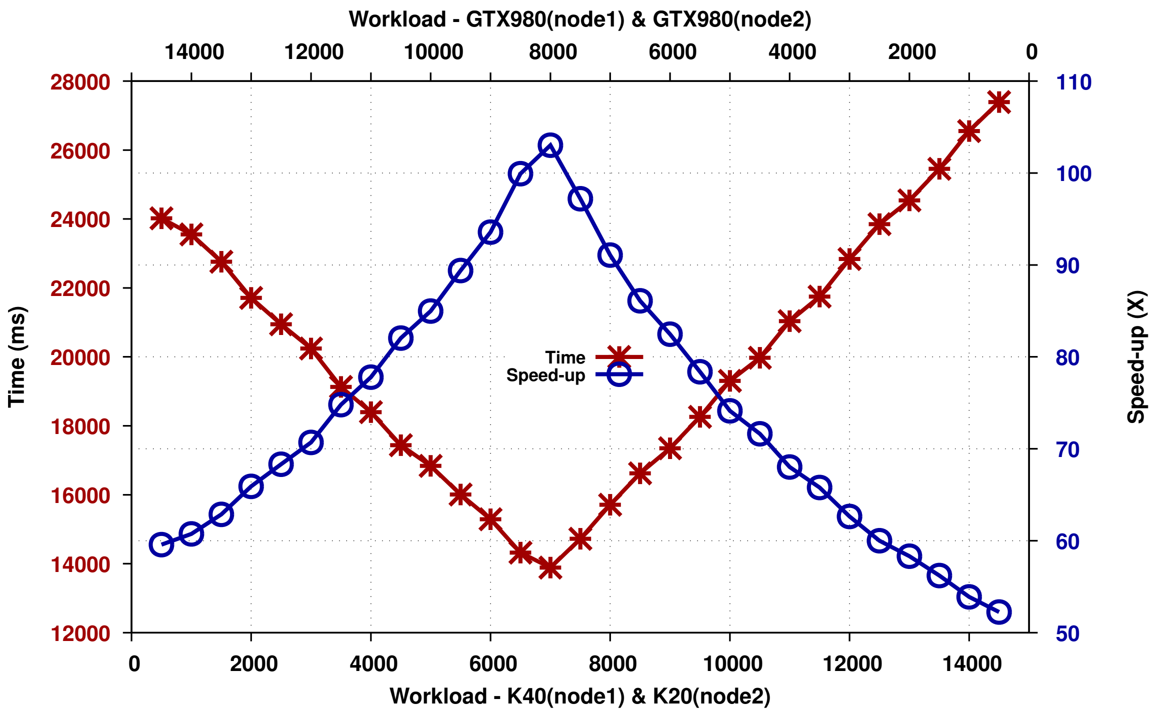
(b) Performance on 3 GPUs: `NVIDIA K40` and a pair of `GTX980`.

FIGURE 6.19: *Gaussinan Blur* benchmark performance metrics on two (a) and three (b) GPUs. Time in red (lower is better), speed-up in blue (higher is better).

time are reported in Table 6.11 shows that communication time dominates the overall execution time.

(a) Performance on 2 nodes and 2 GPUs: `NVIDIA K40`(*node 1*) and 1 `GTX980`(*node 2*).





(b) Performance on 2 nodes each with 2 GPUs.

FIGURE 6.20: *Sobel* filter benchmark performance metrics on two Ethernet interconnected nodes. (a) reefers to the case where node 1 is equipped with 1 *K*40 and node 2 with a *GTX*980. (b) to the case where *node 1* uses a *K*40 and a *GTX*980 while *node 2* a *K*20 and a *GTX*980. Time in red (lower is better), speed-up in blue (higher is better).

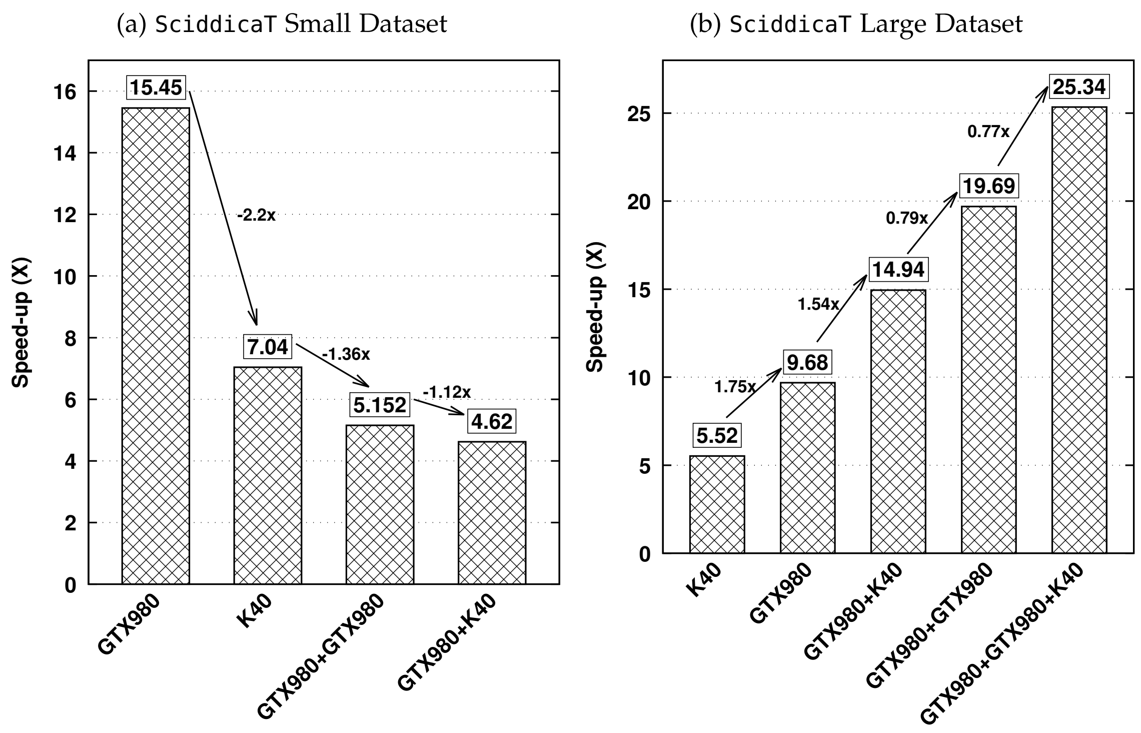


FIGURE 6.21: sciddicaT (small and large datasets) speed-up scaling as the number of GPUs is increased.

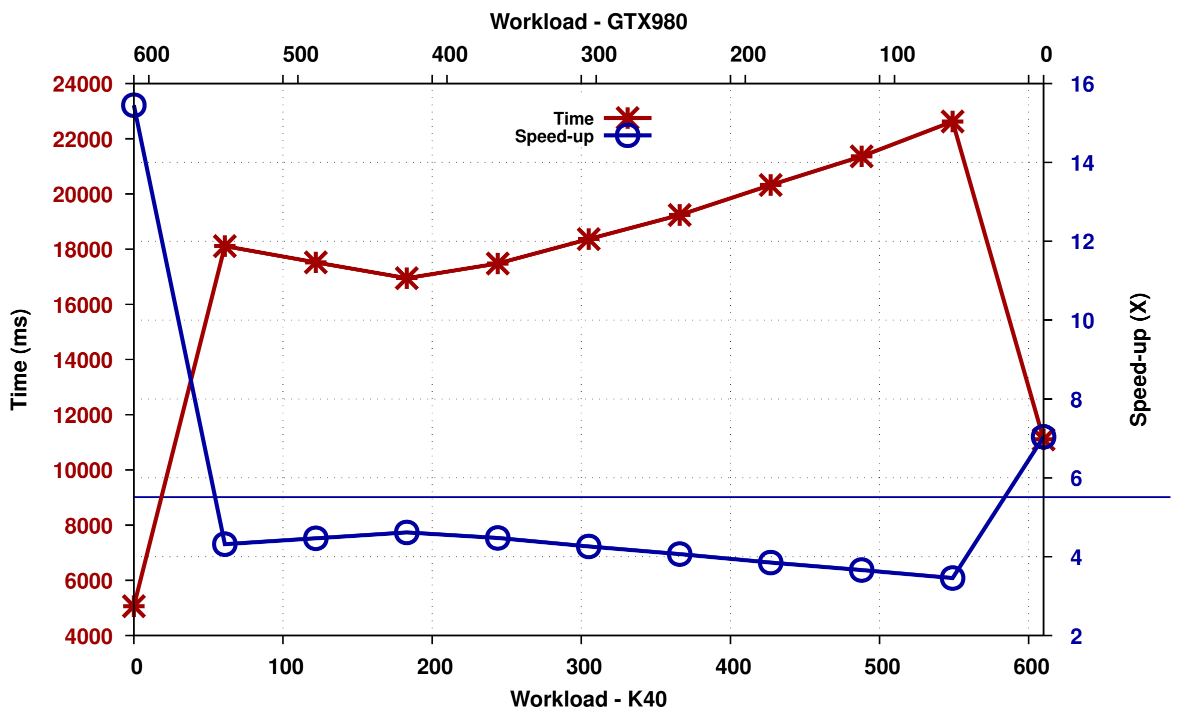


FIGURE 6.22: sciddicaT standard dataset on 2 GPUs. Time in red (lower is better), speed-up in blue (higher is better).

Active cells version of the **small** dataset benchmark on *two nodes* exposes even worse performance due to the fact that active cells need further communication computation since a

Communication Time	Computation Time
20527.2 ms	5163.8 ms

TABLE 6.11: Communication and computation time for the execution of the **naive** version **small dataset** on two nodes and each equipped with a `GTX980`.

Communication Time	Computation Time
19720.99 ms	10204.1 ms

TABLE 6.12: Communication and computation time for the execution of the **active cell** version **small dataset** on two nodes and each equipped with a `GTX980`.

cell can be activated by its owning GPU or by a neighboring one. Once the boundary active cells data are exchanged, activation information need to be processed by each device, by means of an additional *active cell merging kernel*. Communication and computation timings are reported in Table 6.12 and show that even in this case, communication time dominates and also that the computation time is higher, leading to a $10\times$ slowdown w.r.t. the **single GTX980** execution.

Regarding the **large** dataset **naive** version, good performance is achieved for the two adopted run-configurations.

- When the domain is scattered equally across the two `GTX980` (each one residing of the two nodes) a speedup of ≈ 20 is reached (similarly to the *single-node* 2 `GTX980` experiment reported in Figure 6.24b).
- When the domain is scattered across 4 devices i.e. 2 `GTX980` and 1 `K40` and 1 `K20` in the same cluster configuration of the benchmark reported in Section 6.4.3 good overall performances but no improvement is obtained with the addition of two GPUs. This is mainly due to two factors:
  - I The additional devices perform worse ($\approx 2\times$) than the `GTX980` on this benchmark as can be seen in Table 5.7.
  - II When divided among 4 devices each sub domain size is relatively small impeding good utilization of the device.

  Figure 6.23 shows timing and speed-up achieved for this specific case.

6.5 CONCLUSION AND FUTURE WORKS

This chapter introduced the preliminary implementation of the distributed memory and multi-GPU version of `OpenCAL`: `OpenCAL-CLUST`. It has been shown that it allows deploying numerical applications on regular grid on machines composed by several computational nodes interconnected by network each armed with multiple accelerators. Thanks to the adoption of `OpenCL`, different kinds of accelerators can be employed seamlessly. The performance benchmarks that have been used to test `OpenCAL-CLUST` (Section 6.4) show that it effectively use the computational power of multiple devices in order to speedup the computation.

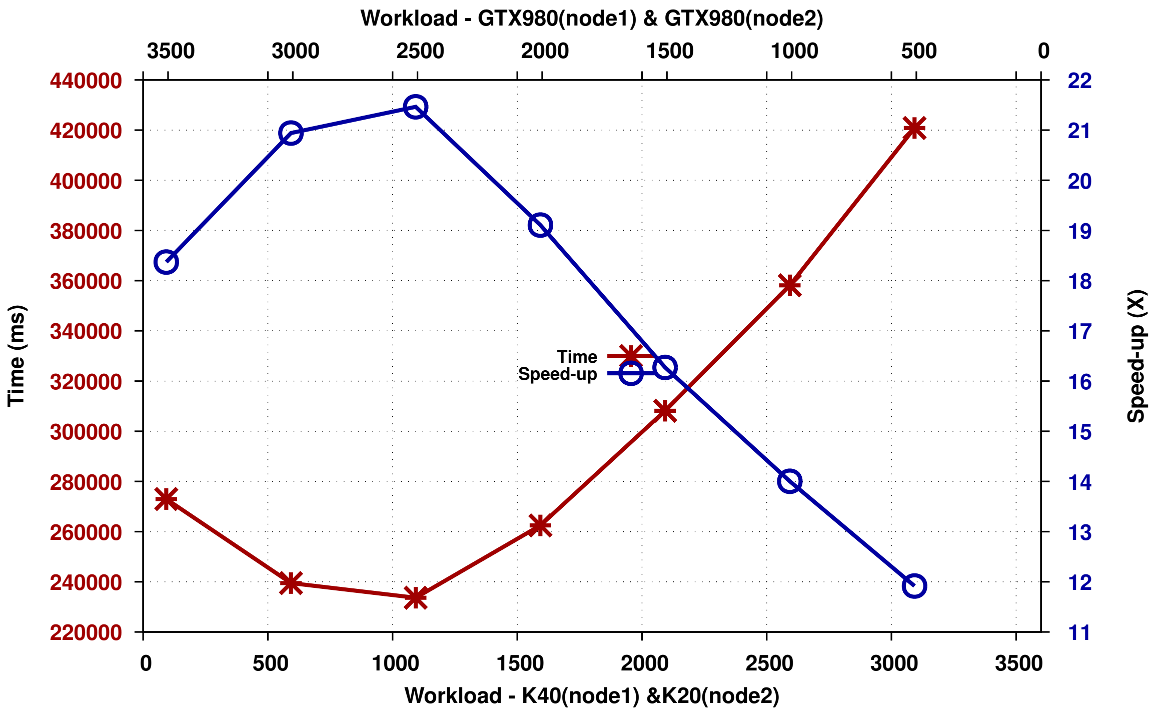
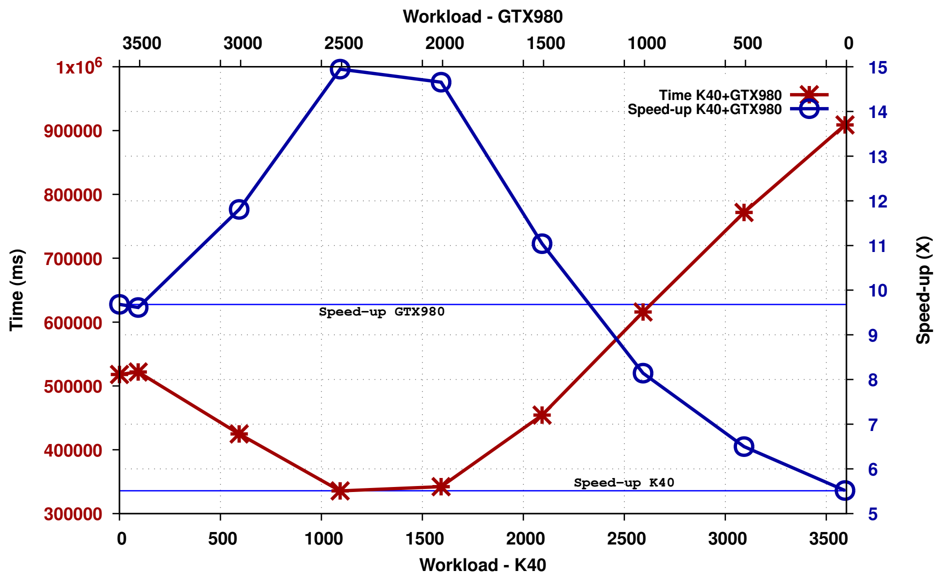
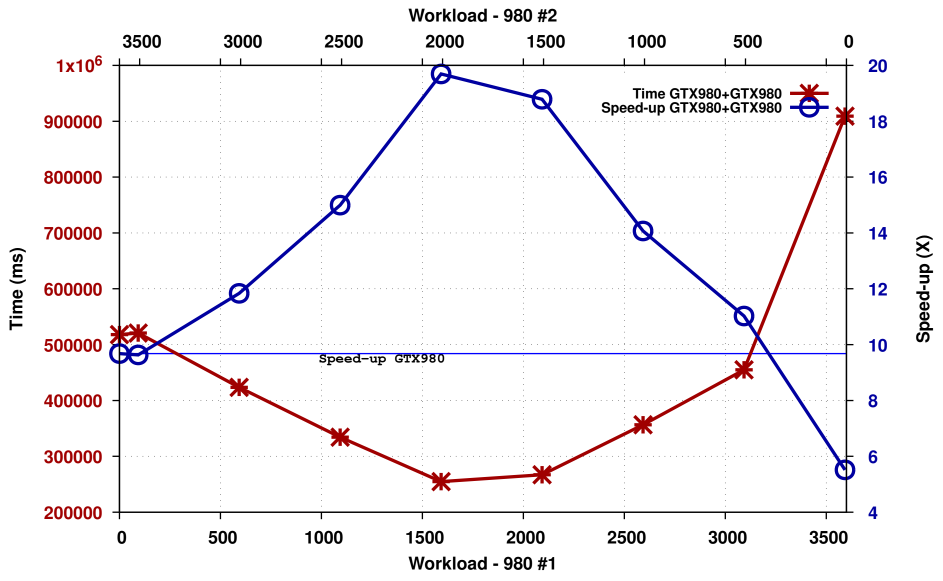


FIGURE 6.23: `sciddicaT` **large** dataset **naive** version on 4 devices installed onto 2 separate nodes. Time in red (lower is better), speed-up in blue (higher is better).

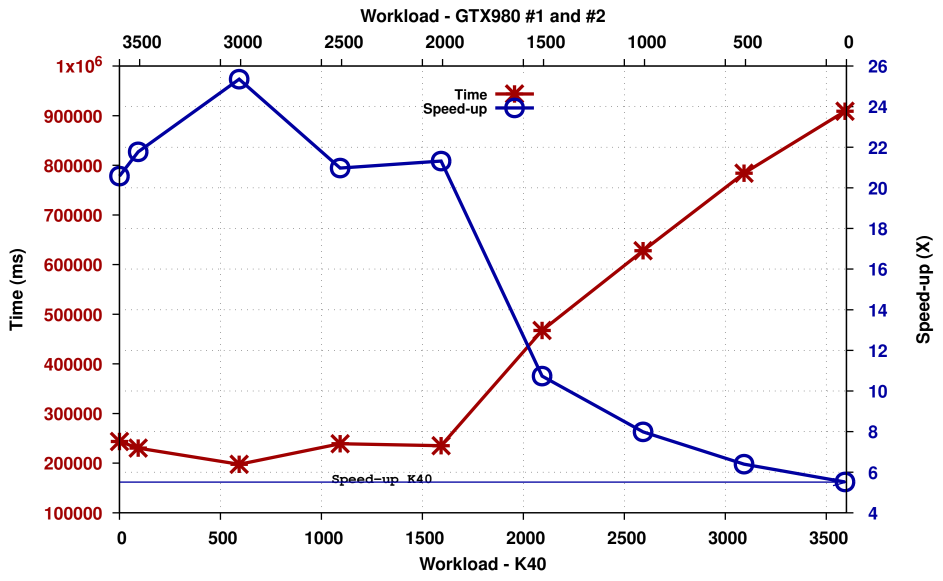
As regarding future developments, `OpenCAL-CLUST` will be extended allowing domain decomposition on multiple dimensions. As shown in Section 6.1.2, decomposing the domain on a single dimension is not always optimal, as in order to obtain better load balancing among the devices different decomposition may be necessary. The programmer would be able to decompose the domain multidimensional cubic portions and assign one to each available device. Another important issue that will be addressed is that the current implementation serializes communications at each step execution taking advantages of possible computation-communication overlapping. Another limitation that the current implementation exposed is that boundaries are always exchanged among intra-node devices. This might not be optimal in cases where boundaries grid cells between two devices or nodes do not change and thus, the communication of such cells avoided. This mechanism can be accomplished by performing boundaries exchange only if there is a relevant update i.e. by means of the so called *dirty-bits* mechanism. A scaling benchmark will be performed on a proper *HPC* cluster with at least 16 nodes interconnected by fast network (`Infiniband` et. similia for instance) in order to obtain information about the scalability of `OpenCAL-CLUST` as the number of nodes is increased.



(a) Performance on 2 GPUs: `NVIDIA K40` and 1 `GTX980`.



(b) Performance on 2 GPUs: a pair of `NVIDIA GTX980`.



(c) `sciddicaT` performance on 3 GPUs: a pair of `NVIDIA GTX980` and a `NVIDIA K40`.

FIGURE 6.24: `sciddicaT` benchmark performance metrics on a `NVIDIA K40` and `NVIDIA GTX980` (a) and on a pair of `NVIDIA GTX980` (b) GPUs. Time in red (lower is better), speed-up in blue (higher is better).

Part III

Other Related HPC Applications

This part of the thesis contains work that have been developed within the context of OpenCAL but has diverged over time into something self contained, and for this reason is only partially related to the content of the first part of the thesis. Both following Chapters, 7 and 8 have been originally investigated as test applications while Appendix A as an optimization for the OpenCAL framework. Chapter 7 describes an application of an early stage version of OpenCAL to the tracking of particle-like object and parallel image processing of time-lapse images. Chapter 8 is an application related to agents and crowd dynamics and it was an attempt to adapt the library to mesh-less spaces. Appendix A contains an algorithm which is highly optimized for new generation GPUs that have dedicated hardware for the so called *ballotting* functions.

7 A TRACKING ALGORITHM FOR PARTICLE-LIKE OBJECTS

All things as subsist from nature appear to contain in themselves a principle of motion and permanency; some according to place, others according to increase and diminuation; and others according to change in quality.
— Aristotle

This chapter is adapted from [142].

Studying the movement of sub-micron particles, micro-spheres and molecules under microscopic observation often requires their time trajectories from which important kinematic and dynamic properties can be computed. Several studies employ time-lapse microscopy, especially in the field of biophysics, as a tool to gather data and retrieve single particle time trajectories. The researcher usually relies on manual or semi-manual/interactive software to study such properties. However, this approach is unfeasible when the number of cells involved in the analysis is high.

In this Chapter an original algorithm for detecting and tracking particles that is based on image processing techniques and to shape difference and centroid displacement analysis to reconstruct the trajectories is presented. To our knowledge this is the first study regarding semi-automatic detection of particles-like objects adopting parallel CA. In particular, the method works for n-dimensional input data provided that particles are represented by at least a centroid space coordinate and a geometrical entity which describe its shape. Since 2D images are a common source of such data we also present framework for image-manipulation based on the Extended Cellular Automata(XCA) paradigm .

`TraCCA` has been successfully applied for the investigation of the motility of *B. subtilis.* injected in a micro-fluidic device using 4100 images taken at 100 frames per second, as reported in Section 7.3.

The Chapter is organized as follows: Sections 7.1 and 7.2 outline the proposed tracking algorithm and cellular automata based image processing framework, respectively; Section 7.3 shows a detailed application of `TraCCA` referred to a real case study regarding bacterial motility, while conclusions and possible future works are reported in Section 7.4.

7.1 TRACKING ALGORITHM

The objective of the tracking algorithm is to produce a set $T_n = \{t_i\}$, where $t_i = \{c^i_k, c^i_{k+1}, \ldots, c^i_l\}$ of trajectories each described as a time-ordered list of positions in space from a set of input particles $P = P_1 \cup P_2 \cup \ldots \cup P_n$ and a function $\mathcal{D} : P \times P \mapsto \mathbb{R}$, the *distance* function. $P_i = \{p^j_i \mid 1 \leq j \}$ indicates all particles at time i and each particle p^j_i is defined by a centroid position, and a bounding box which describes its geometrical properties. $\mathcal{D}(p, q)$ measures the likeliness that a particle p has been transformed into q as a result of the application of a number of geometrical transformations such as translation, scaling, shearing or rotation (see Eq. 7.2). Indexes k and l, $k \leq l$ indicate the trajectory starting and ending time of the tracked movement respectively, and the length $l - k + 1$ is its duration in time. Note that particles may appear or disappear at any time and hence $k \geq 0$ and $l \leq n$. Moreover, $|t_i| \leq l - k + 1$ since a particle which has been successfully tracked from P_k to P_{k^*} can disappear for a certain time and may appear again in $P_{\bar{k}}$, $k \leq k^* \leq \bar{k}$. We only allow disappearing time $\bar{k} - k^* \leq \xi$ where $\xi \geq 0$ is a parameter of the algorithm. Each trajectory $t_* = \{c^*_k, c^*_{k+1}, \ldots, c^*_l\}$ is composed by positions of particles $p^{j_1}_k, p^{j_2}_{k+1}, \ldots, p^{j_*}_l$ at different times. This means that, for our purpose, particle $p^{j_1}_k$ at frame k has moved from c^*_k to location of particle $p^{j_2}_{k+1}$,c^*_{k+1} at time $k+1$ and to location of $p^{j_*}_l$, c^*_l at time l.

As an example, let us consider a human tracking system where each P_i could correspond to all the detected bodies in a video frame i and the distance function a linear combination of the euclidean distance between two detected bodies centroids and pixel-by-pixel difference in colors of all the pixels within their bounding boxes. In this context, it would make sense to consider a not null disappearing time since it is not uncommon for the human detection module (which is in charge of producing the centroids and bounding box from the images) to skip recognizing a specific target only for a limited number of frames.

In order to construct the trajectories, the algorithm works sequentially from frame 1 to n processing, at each step, two subsets of particles, M_i and P_i, where M_i contains all the corresponding trajectories ending particles p^j_l that can still be expanded, i.e. $i - l \leq \xi$. Informally, the algorithm tries to augment an element in M_i using a particle in P_i making sure at most one particle is added to it, the same particle does not augment two different trajectories and the augmenting is performed s.t. the distance function is minimized.

Since at each step of the process a possible assignment between an element of M_i and one of P_i is sought, the algorithm can be thought to be similar to the *assignment problem* [143] and more specifically, it consists in finding a minimum weight matching (not necessarily perfect) in a weighted bipartite directed graph $G = (V, E)$ where $V = M_i \cup P_i$ is the set of nodes and M_i, P_i are the two partitions, $E = M_i \times P_i$ s.t. $e \in E, \mathcal{D}(e) \in \mathbb{R}$ is the weight of the edge. A valid matching $S \subseteq E$ must satisfy the following:

$$\forall (u,v) \in S \begin{cases} (w,x) \in S,\ v = x \iff u = w \\ \mathcal{D}(u,v) = \min_{x \in V_2} \mathcal{D}(u,x) \\ \nexists\,(w,v) \in E \text{ s.t. } \mathcal{D}(w,v) < \mathcal{D}(u,v) \end{cases} \tag{7.1}$$

If we denote the matching operator as the following recurrence relation $M_i \Diamond P_{i+1} = (T_{i+1}, M_{i+1})$, $M_0 = P_0$, then the tracking algorithm can be summarized as $(T_n, M_n) = M_{n-1} \Diamond P_n = (((P_0 \Diamond P_1) \Diamond P_2) \Diamond \ldots \Diamond P_n)$. If after the $\Diamond$ application a particle $p^* \in M_i \cup P_i$ remains unmatched[1], two cases have to be considered (see Figure 7.1):

1. if $p^* \in M_i$, the disappearing time counter u_{p^*} for p^* is updated and if $u_{p^*} > \xi$, p^* is not included in M_{i+1} and the corresponding tracked trajectory is flushed into T_i, otherwise it is retained. This handles the case when particles may disappear from the dataset for a number of time steps and appear again.

2. if $p^* \in P_i$, it corresponds to a newly appeared particle which is then inserted into M_{i+1}.

The pseudo-code reported in Algorithm 2 shows how the matching procedure is implemented. Note that the NEIGH procedure filters the possible candidates for a particle only to those which the euclidean spatial distance is less than a threshold parameter. In many

1 $\nexists\,(u,v) \in S$ s.t. $u = p^* \vee v = p^*$ (cf. Equation 7.1)

real life applications particle displacement between two subsequent time-steps are small, so it would be useless trying to match a particle at time i with one at time $i+1$ which are spatially far apart.

Algorithm 2: `TraCCA` tracking procedure

```
1  Function MATCH ($M_i, P_{i+1}$);
   Input  : Matched and to be matched particles $M$ and $P$ respectively.
   Output: $(T_{i+1}, M_{i+1})$, which are the updated set of trajectories and matched
           particles.
2  $T_{i+1} = T_i$
3  foreach $p \in M_i$ do
4      $neighbours[p] \leftarrow NEIGH(p, P_{i+1})$ ;
5      foreach $n \in neighbours[p]$ do
6          $d[p][n] \leftarrow DISTANCE(p, n)$;
7      $SORTBY(neighbours[p], d[p])$;
8  foreach  $p \in M_i$ do
9      if $neighbors[p].size \neq 0$ then
10         $candidate \leftarrow neighbors[p].first$
11     else
12         $p.u \leftarrow p.u + 1$;
13         continue;
14     if $match[candidate] = NIL$ then
15         $match[candidate] \leftarrow p$;
16         $p.u \leftarrow 0$;
17     else
18         $p' \leftarrow match[candidate]$
19         if $d[p][candidate] >= d[p'][candidate]$ then
20             $neighbors[p].pop()$
21         else
22             match[candidate] $\leftarrow p$;
23             $neighbors[p'].pop()$ ;
24             $p \leftarrow p'$;
25         go to 10;
26 foreach $p \in P_{i+1}$ do
27     if $match[p] = NIL$ then
28         $T_i + 1.push(< p >)$
29     else
30         $MATCH(T_{i+1}, match[p]).enqueue(p))$;
31 foreach $p \in M_i \cup P_{i+1}$ do
32     if $p.u < P_r$ then
33         $M_{i+1}.push(p)$
34 return $(T_{i+1}, M_{i+1})$;
```

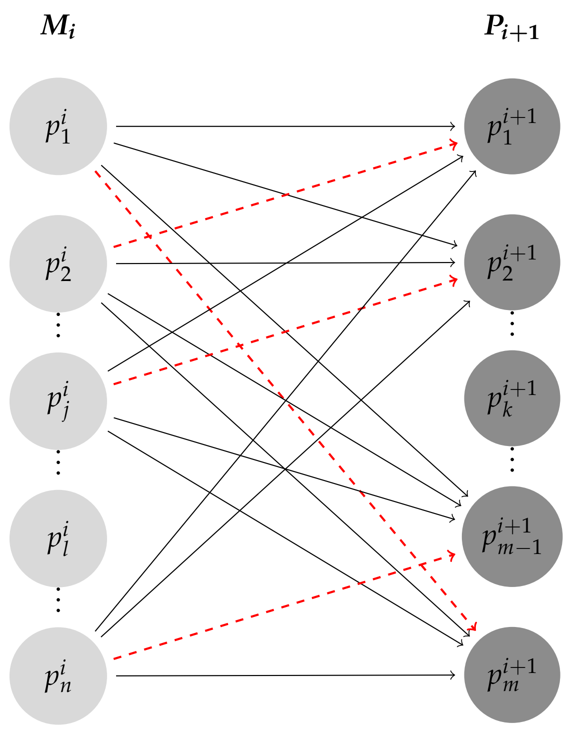


FIGURE 7.1: An application of the $\Diamond$ operator referred to a frame P_{i+1} of m particles. Dashed red arrows highlight the solution S. For instance, the trajectory $t_* = p_a^s \rightsquigarrow p_1^i$ is lengthened by p_m^{i+1} becoming $t = p_a^s \rightsquigarrow p_1^i \rightarrow p_m^{i+1}$. Unmatching (no incident dashed red arrow are present) particle p_k^{i+1} causes a trajectory of length one to be created, while unmatching particle p_l^i causes its disappearing time counter to be updated and the cornrensponding trajectory to be possibly finalized (if $u_{p_l^i} > \xi$). Note that, for the sake clarity, arrows for the unmatched nodes are omitted.

7.2 MANIPULATING IMAGES USING XCA

In this section we present a Cellular Automata ([10, 26]) based framework for manipulating images that allows seamless parallel filters application.

7.2.1 *Definition and Usage*

The XCA adopted for manipulating images is defined as a 7-tuple $IFCA = \langle R, X, Q, P, \sigma, \Gamma, \gamma \rangle$ where:

- R is the 2-dimensional cellular space.
- $\Gamma \subseteq R$ is the region over where the global operation is applied
- $X = X(x_0, y_0) = \{(x,y) : |x - x_0| \leq r \wedge |y - y_0| \leq r\}$ defines the *Moore*'s neighborhood relationship of radius r
- $Q = Q_r \times Q_g \times Q_b \times Q_a$ representing the pixel color channels in the RGBA space.
- P is the set of global parameters

- ψ is the initialization function (which is in charge of reading images from files and converting them to substates).
- $\sigma = \{\sigma_i : Q^{|X|} \mapsto Q\}$ is the set of image convolutional filters (corresponding to XCA elementary processes)
- $\gamma = \{\gamma_i : Q^{|\Gamma|} \to Q^{|\Gamma|}\}$ is the set of non-local filters (the XCA global functions).

In order to take advantage of the intrinsic parallel nature of CA, `IFCA` is implemented augmenting an existing CA library, **`OpenCAL`** [144] which is empowered with a set of procedures that allows seamless input/output and image convolution. More specifically each image is represented as an cellular automaton whose substates represent the color of the pixel and filters implemented as elementary processes composition (see listing 7.1).

```
 1 //Model and CA engine
 2 CALMooreNeighborhood<DIM,RADIUS> neighbor;
 3 MODELTYPE calmodel(IMG_SIZE,&neighbor,SPACE_FLAT);
 4 CALRun calrun(&calmodel, 1, STEPS, UPDATE_IMPLICIT);
 5 CALSUBSTATE* bgr=calmodel.addSubstate<PIXELTYPE>();
 6 //Image loading
 7 bgr->loadSubstate(*(new std::function(loadImage<PIXELTYPE>)), "image.tiff");
 8 //Image Filters Creations
 9 ContrastStrchFlt<...>csf(bgr,1285,1542,0,65535,1);
10 ThresholdFlt<...> tf (bgr,0,61680,0,65535);
11 calmodel.addElementaryProcess(&cst);
12 calmodel.addElementaryProcess(&tf);
13 //Trigger execution
14 calrun->run();
```

Listing 7.1: Example of usage of the IFCA XCA engine. Lines 1-5 create a model and runtime for an image of size IMG_SIZE and dimension DIM. Lines 7 loads the image into the substate which

`IFCA` is extensible as it allows the application of user-defined filters by only specifying the pixel transformation rules (or kernels in case of convolutional ones). Filters are then executed by the `IFCA` engine which comes with two parallel (`OpenMP` and `OpenCL`) and a serial implementations. The parallel execution is completely transparent and automatic and may be extremely useful in such cases where the size of the image is large or filters are particularl

7.3 MOTILITY ANALYSIS OF B. SUBTILIS

7.3.1 *Introduction*

In this section we present an application of `TraCCA` to the analysis of motion of *B. subtilis.* The analysis of trajectories in bacteria is really interesting because it is a non-invasive way of extracting much information about their chemotaxis which is the ability of bacteria to sense and respond to chemicals. Thanks to chemotaxis, bacteria are able to reach a source of nutrient and move away from repellents, which makes it a key characteristic for their survival. Studying chemotaxis is really interesting not only because it has a strong influence on many biological mechanisms, e.g. biofilm formation and disease pathogenesis but also because evolution's natural selection has optimized bacterial chemotaxis making bacteria excellent source-seekers and their strategies can be used to design bio-inspired, simple and

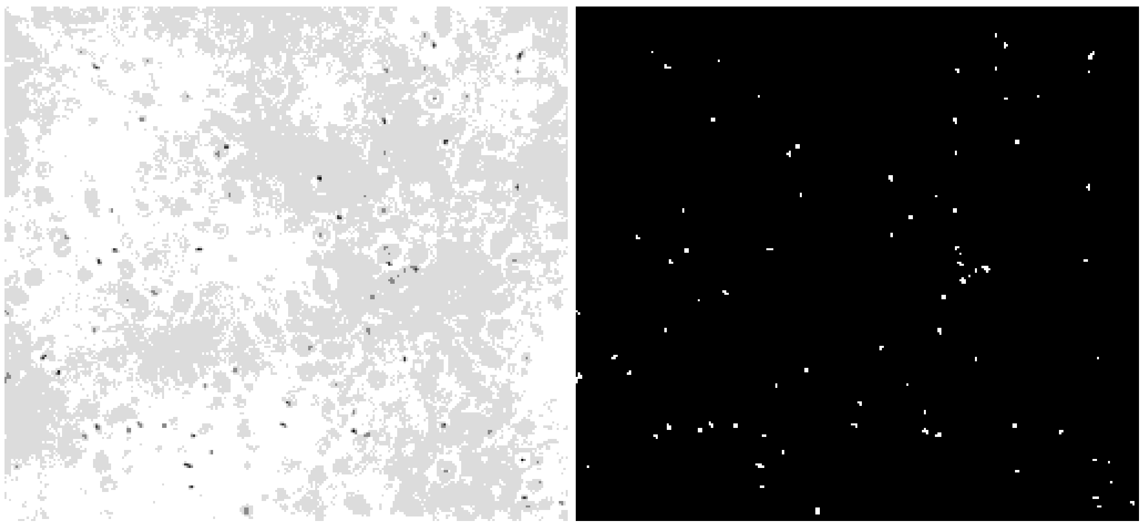


FIGURE 7.3: Segmented and color inverted version of Figure 7.2 as obtained by applying the filters (cf. Section 7.3.2) using the IFCA model. White cluster of pixels represent bacteria.

efficient algorithms for robotic source locating systems. The motility of *B. subtilis* is composed of a series of *run* (bacteria swim along smooth segments) and *"tumbles"* (cells stop and randomly select a new direction). Combining run and tumbles bacteria are able to direct their motility in order to reach nutrients and move away from repellent, in other words, to do their chemotaxis. An algorithm that is able to track bacteria is really useful because from bacterial trajectories much information about the way bacteria perform chemotaxis (e.g. duration of the run, swimming speed, frequency of tumbling events) can be extracted. This information is important in studying the strategy they adopt to reach a source of nutrient; moreover it is also interesting to investigating how the trajectories changes as certain environmental conditions change, such as temperature, oxygen concentration or gradient of chemicals.

In this work we used *B. subtilis* strain `OI1085` for the tracking experiment. Cells were taken from frozen stock, resuspended in CAM (Cap Assay Minimal) and shaken (37°, 100 rpm) until the optical density $OD600 = 0.3$ was reached; we then diluted the suspension 1:10 in CAM. The bacterial suspension was injected in a micro-fluidic device. The micro-fluidic device is a simple device made by *PDMS* and glass, composed of three parallel channels (height 100 µm). In the central channel (600 µm wide) bacteria were hosted and observed. Two walls of *PDMS* (200 µm) separate the central channel from the left and right channels were oxygen was flown in order to reach a concentration of oxygen closed to 100% in the central channel. 10 minutes after the injection of bacterial suspension a video was acquired at 100 frames per second for 41 s (#4100 frames). All images were acquired through a 10× phase contrast objective (*Nikon* microscope), using binning 2×2. All images are 512×512 pixels and have been exported in 16-bit grayscale `TIFF` image format.

7.3.2 *Bacteria Segmentation*

The *B. subtilis* cells typically have a large range of motion patterns and the cell soma generally appears as a dark area surrounded with a white halo. Colors can be inverted and the cell may appear white when it moves out of focus sufficiently (see Figure 7.2). In order

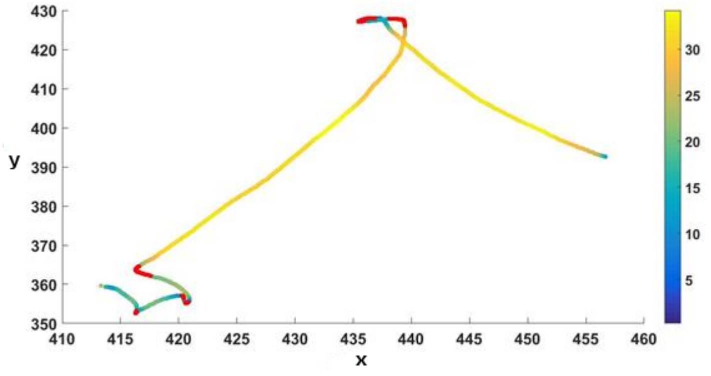


FIGURE 7.4: An example of tracked trajectory of a single bacterium. Blue-yellow colors levels are proportional to the velocity, while dark red segments represent the individuated tumbles.

to automatically detect *subtilis* cells, it is possible to use a histogram-based thresholding method as suggested in [145]. A key step of tracking bacteria is to individuate first, and then to label and describe each of the bacteria present in the images. For this purpose, a segmentation preprocessing phase is performed on all the images. Segmentation is carried out by means of a threshold method **[Stockman:2001:CV:558008]** which produces a bipartition of pixels based on the color intensity. The value of pixel (x, y) in image g is given by the following, where P is a predicate and f is the original image:

$$g(x,y) = \begin{cases} 1, & \text{if } P(f(x,y),T) \\ 0, & \text{otherwise} \end{cases}$$

As a consequence, a drawback of using the threshold method is that, among all inter-pixels relationships, color intensity difference is the only involved by the bi-partition process. This can easily lead to binary regions where pixels are not contiguous or to miss or include relevant or unwanted pixels respectively, with these effects getting worse as the noise increases. In the considered case, however, the threshold method works well because after the application of the `contrast stretch filter` the analysed images present high contrast between the background and cells soma (see Figure 7.2), since the filter stretches or scales the range of pixel values between an upper and lower limit. More specifically, color values that are above or below this range are saturated to the upper or lower limit values respectively, while values that lay in the interval are scaled according to the following formula:

$$g(x,y) = \begin{cases} L_o & \text{if } f(x,y) < L_i \\ H_o & \text{if } f(x,y) > H_i \\ L_o + (f(x,y) - L_i)\frac{H_o - L_o}{H_i - L_i} & \text{if } L_i \le f(x,y) \le H_i \end{cases}$$

where $[L_i, H_i]$ defines the interval in the original image which is linearly scaled into the interval $[L_o, H_o]$.

Eventually, all the images go through a noise reduction stage which employs a combinations of *gaussian*, *laplacian* and *blurring* filters [146]. It is worth to note that filters parameters are dataset dependent and the best set of values experimentally determined.

7.3.3 *Bacteria Tracking*

Binary images are then used to extract the relevant information (shape and centroid) for each of the segmented bacteria. This is done by interpreting each image as a graph whose nodes are pixels and arc (i, j) exists if pixels i and j are neighbours (according to *Moore*'s relationship, see section 2). A bacterium corresponds to a connected component that can be easily individuated by using the DFS visiting algorithm. Once all pixels that make up the cell soma are individuated, a unique *ID* id_i which identifies the bacterium uniquely, a centroid c_i and a bounding box s_i which describe the location, the shape and the area, respectively, are computed for each bacterium i. All the bacteria whose extension is less than $M_e = 2px$ are ignored and no further considered. The creation and manipulation of all the geometrical entities involved in this latter phase are carried out by means of the computational geometry library `CGAL` [147].

In order for the application of the algorithm described in Section 7.1, a distance function is required. Among possible functions, a linear combination of the centroids displacement and shapes difference was adopted:

$$D(c_i, c_j) = a\sqrt{(x_{c_i} - x_{c_j})^2 + (y_{c_i} - y_{c_j})^2} + b|S_i \cap S_j| \qquad (7.2)$$

where S_i and S_j are the sets of pixels within the boundaries of the bacteria's bounding box. Best values of weights a and b were experimentally determined in being 0.6 and 0.4 respectively. Parameters search is carried out by computing the confusion matrix for 5 randomly picked images and for different values of $a, b \in \{\frac{i}{20} \mid 0 \leq i \leq 20\}$ and then choosing the ones minimizing false negatives and false positives and maximizing true negatives and true positive ones.

Figure 7.6 shows 4 snapshots of the outcome of the tracking algorithm for 4 bacteria taken at 4 different times, while Figure 7.5 depicts the final collective view of all bacteria movements in the considered case study.

7.3.4 *Analysis and Validation*

Starting from the relationship $D_b = \frac{v^2 t_t}{2}$ [148] that links the diffusion coefficient to both swim velocity (v) and tumbling time (tt), we have partitioned each trajectory into running and tumbling events. These events allow the swimming speed to be calculated as difference between subsequent bacteria positions and average tumbling time as the overall time spent tumbling over by the number of tumbling events. Inspired by the work of *Wong-Ng et al.* [149] and *B. Masson et al.* [150] an algorithm, implemented in *MATLAB*, was adopted for detecting such running/tumbling events (see Figure 7.4) which in turn were used to extract all the relevant bacterial motility parameters from the tracked trajectories[2].

The main parameters that were considered as as descriptors of the motion of the bacteria were:

- *Mean Swimming Velocity* ($\overline{v}$),
- *Mean Run Time* ($\overline{r_t}$) and

2 Tumbles are associated both to a decrease in advancing velocity and an abrupt change in the angular velocity (i.e. in the direction of the motion). By fixing a threshold on both advancing and angular velocities it is possible to identify running/tumbling and use them to compute the mean values t_t and v over all the trajectories of the experiment.

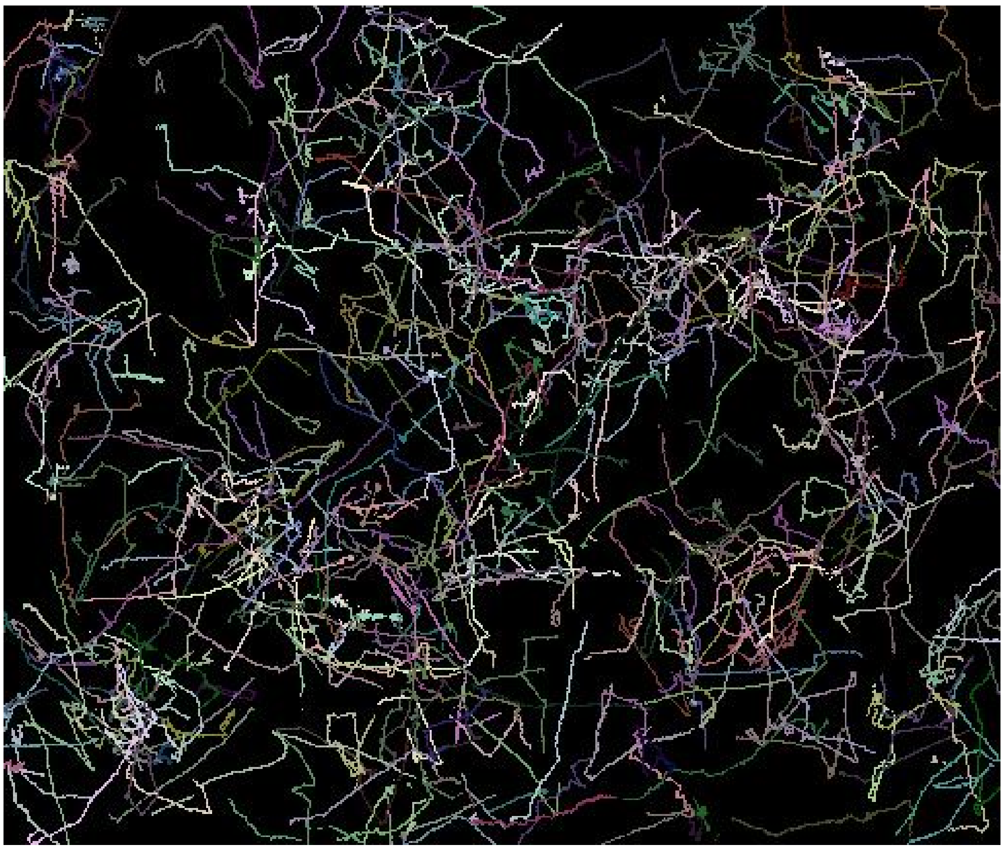


FIGURE 7.5: Final collective view of all the tracked bacteria tracjectories as obtained by processing all 4100 images of the case study dataset. For the sake of figure clarity, a random color is associated to each bacterium trajectory.

- *Tumble Time* ($\overline{t_t}$).

By analyzing the entire trajectory set produced by `TraCCA` on the dataset described in Section 7.2, the obtained values of the motion descriptors were the following:

- $\overline{v}$ =18 µm s^{-1},
- $\overline{rt}$ = 0.8 s
- $\overline{t_t}$= 0.18 s.

These experimental values[3] are in accordance with those available in the literature as for instance in the work of *Cisneros et al.* [153].

3 These motility parameters were also calculated using the *Mean Square Displacement* (`MSD`) in time (which for the sake of brevity is only outlined here) by means of the MATLAB `@msdanalyzer` implementation [151] and considering the following relationship $MSD(t) = \frac{1}{2} \frac{v^2 t_R^2}{2t/t_R + e^{\frac{-2t}{t_R}} - 1}$ [152] where v is the swimming speed, t_R is the timescale associated to rotational diffusion that is $t_R = 2t_t$, it is possible to obtain t_R and v via fitting. t_t and v are then used to compute D_b.

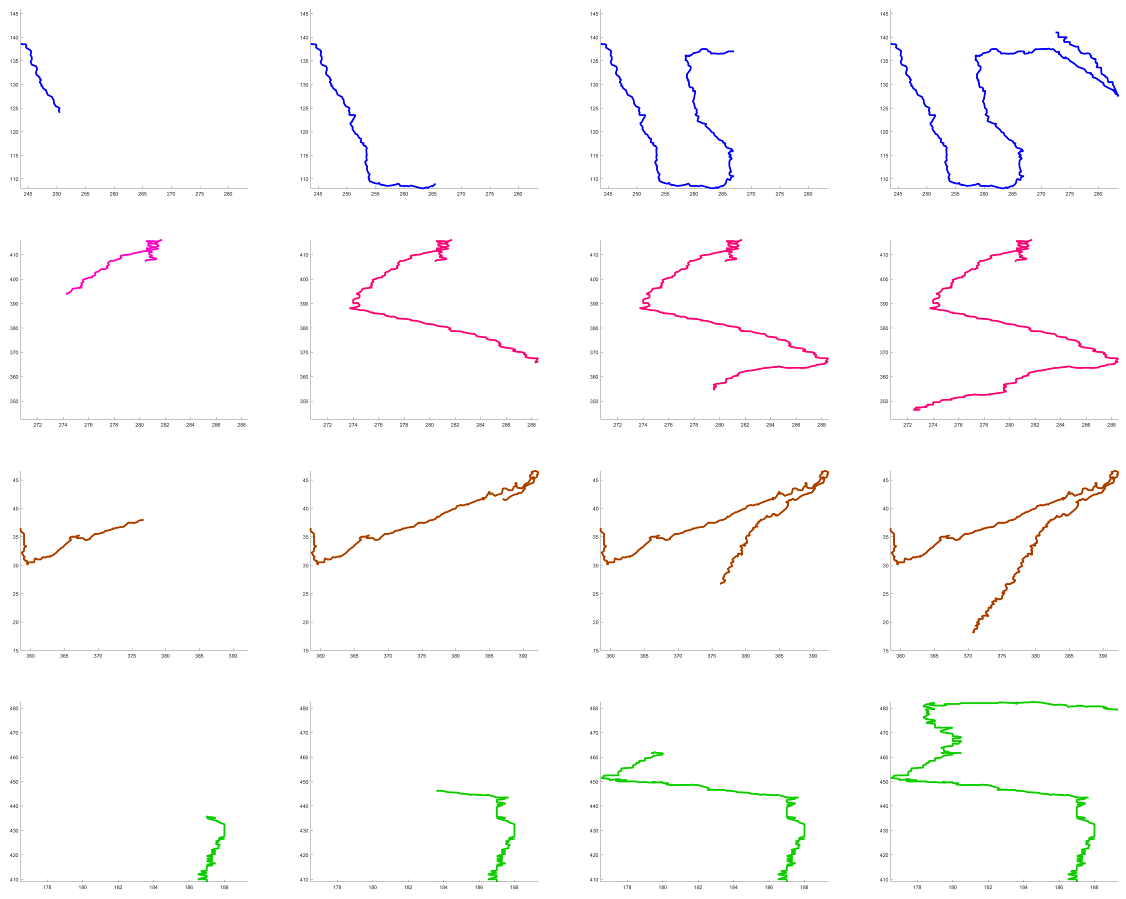


FIGURE 7.6: 4 different tracked bacterial trajectories shown at 4 different subsequent times.

7.4 CONCLUSION

In this preliminary work, a cellular-automata based tracking framework composed by a tracking algorithm and a CA support model for image processing, is presented for reconstructing trajectories of particle-like objects. This work reports the application to a real case study concerning the tracking of *B. subtilis*, by evaluating standard motility parameters (average swimming velocity, running and tumble times) in a microfluidic device. Results that were obtained during experimentation, proved that the `TraCCA` framework has correctly reconstructed bacterial trajectories. The tracking algorithm described in this work can also be effectively adopted in other fields as, for instance, crowd dynamics, provided that traceable elements can be described by means of a bounding box and a position. Due to the large number of particles often involved in such applications, a preliminary parallel GPGPU + MPI version of the framework is currently being developed which has provided promising results in terms of scalability and speed up, allowing much larger dataset to be analyzed. Eventually, it is worth to note that under the assumption of associativity of the *assignment operator* $\diamondsuit$, the tracking algorithm could be implemented using the *parallel reduction* design pattern.

8 MULTI-AGENT SYSTEM WITH MULTIPLE GROUP MODELLING FOR BIRD FLOCKING ON GPU

There cannot be a single, simple body which is infinite, either, as some hold, one distinct from the elements, which they then derive from it, nor without this qualification. For there are some who make this (i.e. a body distinct from the elements) the infinite, and not air or water, in order that the other things may not be destroyed by their infinity. They are in opposition one to another — air is cold, water moist, and fire hot—and therefore, if any one of them were infinite, the rest would have ceased to be by this time. Accordingly they say that what is infinite is something other than the elements, and from it the elements arise.
— Anaximander

This chapter is adapted from [154].

The study of collective coherent motion of self-propelled biological organisms such as flocks of birds, schools of fish and swarms of insects has fascinated humans from ancient time. This kind of behaviour, often referred to as flocking, exists in nature at almost every length scale of observation: from human crowds, mammalian herds, bird flocks, fish schools to unicellular organisms such as amoebae and bacteria, individual cells, and even at a microscopic level in the dynamics of acting and tubular filaments and molecular motors. Despite the huge differences in the scales of aggregations, the similarities in the patterns that such groups produce have suggested that general principles may underlie collective motion.

An effective approach to study these collective behaviours is represented by mathematical modelling and numerical simulation, as proven by numerous papers published in this field that are related with both biology and computer science (e.g., [155]). In models that study simple motion principles of organisms like flocking, shoaling or phenomena based on random motion, organisms are treated like gas molecules and their motion is Brownian combined with attraction/repulsion forces. Also 'mean-field' approaches, mainly carried out by adopting ordinary differential equation (ODE) models, might be useful to model some biological swarm systems, whenever the assumption of a 'well mixed' distribution may be applicable. However, organisms seldom move really randomly, nor are they just simple particles. They pursue specific goals, aggregate or disperse in space, communicate and memorize. They can be characterized by specific physiological states (e.g. energy-level) and morphologies (size, weights, etc.). These factors do not only affect their energetics, but may have prominently affect the behaviours that they (choose to) perform. In addition, they frequently interact by direct and indirect communication and they tend to memorize past effects. Eventually, also the environment in which they operate is highly structured and this heterogeneity is also dynamic. All these factors describe important discrepancies between biological life forms and atoms or molecules. Thus, it is likely that models, which were originally derived from physics and chemistry, might not hold well for biological swarm systems as soon as a certain level of abstraction has to be overcome. In these cases, individual-based models or even multi-agent models [156, 157] might be a better choice.

In this article we present the ACIADDRI[1] (*Aggregate Collection of Interactive Agents using NVIDIA CUDA Reliable Informatics*) multi-agent flocking model, besides an efficient GPGPU implementations on SIMT architectures. In particular, section 8.1 formalizes the model and 8.2 the parallel implementation using the CUDA framework. Specifically, starting from Reynolds works [158] [159] [160] on bird flocking behaviour, ACIADDRI extends it by means of additional features such as support for multi-species interaction, predator avoidance, partially observable environment and birds flight constraints (maximum thrust, stall and peak velocity, etc.). Section 8.2 reports experiments carried out on the specific GPU hardware and by considering both aggregate motion of a huge number (up to tens of millions) of boids in a virtual environment and other species or predators avoidance, significant performance improvement in terms of speedup were obtained (up to $500\times$), while conclusions and future works are reported at the end of the paper.

8.1 THE ACIADDRI MODEL

This work is based on flocking behaviour that was proposed by Craig W. Reynolds in 1987 and extends it by adding both the support to coexistence and interaction between different species, and the predator avoidance. Reynolds was amongst the first to abstract this behaviour, in order to steer a swarm of simulated birds which he called boids **[Reynolds]**(contraction of birdoid). Every boid has some limitations: it has a strictly local knowledge of the space it occupies and its knowledge comes from a simulated vision from its current position. In other words there is no centralized control. The flock takes its decisions in a totally distributed manner in order to obtaining a synchronized movement. More specifically, each bird obeys three behavioural rules:

COHESION
to attempt to stay close to nearby flockmates

COLLISION AVOIDANCE/SEPARATION
to evade obstacles and flock mates which are too close

VELOCITY/HEADING MATCHING
also called *alignment*, to move in the same direction as nearby flock mates.

8.1.1 *Model Parameters*

In our ACIADDRI model, the environment and each bird's species is described by means of sets of parameters as shown in the following subsections.

8.1.1.1 *Environment Parameter*

The environment is described by means of its width, length, height and by the time step parameter i.e. the duration in seconds of a single computational step.

W_x, W_y, and W_z are the dimension of the environment that represent 1 pixel as 1 meter. t is computational duration where 1 time step equivalent to 1 processing time.

1 Meaning *birds* in Calabrian language.

Name	Symbol	Dimension	Description
Length	W_x	$[\mathbf{L}]$	Length of the environment
Height	W_y	$[\mathbf{L}]$	Height of the environment
Width	W_z	$[\mathbf{L}]$	Width of the environment
Time Step	t	$[\mathbf{T}]$	Computational step duration

TABLE 8.1: List of Environment Parameters of Birds Flocking

Name	Symbol	Dimension	Description
Size	s	$[\mathbf{L}]$	Size of the bird
Peak Velocity	v_p	$[\mathbf{L}]\,[\mathbf{T}^{-1}]$	The maximum velocity
Thrust	a	$[\mathbf{L}]\,[\mathbf{T}^{-2}]$	The maximum acceleration
Horizontal Range of View	s_h	–	Maximum horizontal range of view
Vertical Range of View	s_v	–	Maximum vertical range of view
Sight Distance	d_s	$[\mathbf{L}]$	Maximum sight distance
Minimum Distance	d_{min}	$[\mathbf{L}]$	The minimum distance between two birds to avoid collision
Alignment Radius	d_a	$[\mathbf{L}]$	The maximum distance bird consider to align
Other Species Avoidance Radius	r_s	$[\mathbf{L}]$	The minimum distance bird avoid other species
Predator Avoidance Radius	r_p	$[\mathbf{L}]$	The minimum distance bird avoid predator
Maximum Turn	θ_{max}	$[\mathbf{rad}]\,[\mathbf{T}^{-1}]$	Maximum turn for each time step
Wander Distance	w_d	$[\mathbf{rad}]\,[\mathbf{T}^{-1}]$	The maximum wandering distance
Wander Radius	w_r	$[\mathbf{rad}]\,[\mathbf{T}^{-1}]$	The maximum radius wandering from the target

TABLE 8.2: List of Bird Parameters with values considered for the simulation

8.1.1.2 *Species Parameters*

Each specie is described by a set of parameter that represent quantities that are involved in the flight and flocking dynamics (see table 8.2).

The bird's wingspan s is used as an approximation of the volume it occupies. v_p is the maximum velocity it can travel and a represents bird's maximum acceleration. Each bird has a limited sight of view that is described by its maximum horizontal, s_h, and vertical, s_v vision span (see images 8.2 and 8.1) and by d_s that is the maximum sight distance of bird i.e. the maximum distance at which the bird can observe objects. Vertical and horizontal field of view (FOV) together with the maximum sight distance define the viewing frustum. Table 8.2 shows the complete list of the used parameters and corresponding alongside.

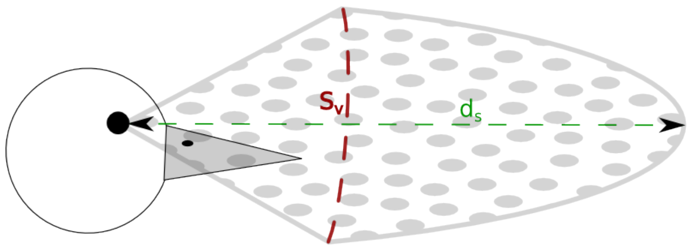


FIGURE 8.1: Birds vertical field of view. d_s and s_v represent the sight distance and the vertical perceptual span, respectively.

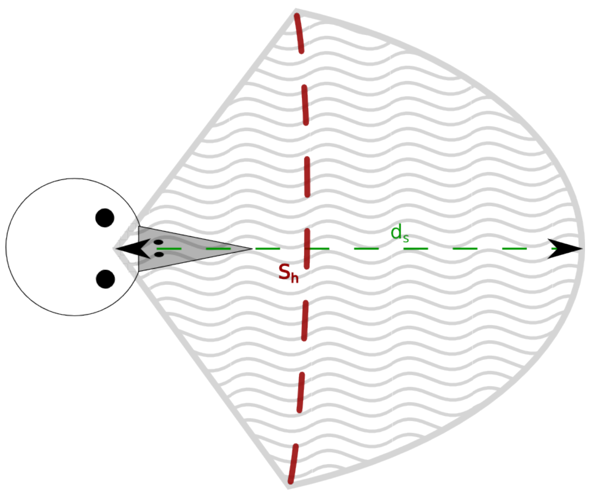


FIGURE 8.2: Birds horizontal field of view. s_h represents the horizontal perceptual span. d_s as in Fig. 8.1.

8.1.2 *Flight Model*

Bird b flight at step i is described by its 3-dimensional space vectors position $\vec{p}^i_b = \left\langle p^i_x, p^i_y, p^i_z \right\rangle$ and velocity $\vec{v}^i_b = \left\langle v^i_x, v^i_y, v^i_z \right\rangle$, that correspond to its current sight direction.

The evolution of the bird's velocity over time is regulated by the following:

$$\vec{v}^{i+1}_b = (r sin\theta' cos\phi',\ r sin\theta' sin\phi',\ r cos\theta') \tag{8.1}$$

where:

- $\theta' = \begin{cases} \theta_d & \text{if } |\theta_b - \theta_d| < \theta_{max} \\ \theta_b + \theta_{max} & \text{otherwise} \end{cases}$
- $\phi' = \begin{cases} \phi_d & \text{if } |\phi_b - \phi_d| < \phi_{max} \\ \phi_b + \phi_{max} & \text{otherwise, it should be adjusted} \\ & \text{depending on the sign if it have to} \\ & \text{go right or left} \end{cases}$
- $\vec{v}^i_d = \mu_c \vec{v}_c + \mu_s \vec{v}_s + \mu_a \vec{v}_a + \mu_a \left(\vec{\tau}_i + \vec{\Gamma}_i \right) + \vec{\omega}_i$
  - μ_c, μ_s, μ_a are the cohesion, separation and alignment coefficient (social coefficients) and μ_a is the avoidance coefficient.
- $\vec{v}^i_c$,$\vec{v}^i_s$, $\vec{v}^i_a$ are the *social velocities* [161]
  - $\vec{v}^i_c$, the cohesion velocity that has direction parallel to the line that pass through $\vec{p}^i_b$ and the average position of its neighbors

  - $\vec{v}_s^i$, the separation velocity, keep the bird at a minimum safety distance from its flockmates
  - $\vec{v}_c^i$, the align velocity, synchronize boids heading.

- θ_d, θ_b are the polar angle of the velocity vector $\vec{v}_b^i$ and $\vec{v}_d^i$ respectively.
- ϕ_d, ϕ_b are the azimuthal angle of the velocity vector $\vec{v}_b^i$ and $\vec{v}_d^i$ respectively.

8.1.3 *Bird's Field of View*

Each bird has a limited visual capacity described by its field of view (FOV). This implies that it can only perceive objects that are within its FOV. Bird o's FOV $\mathcal{F}_p$, is here defined as the set of points p_n that satisfy equations 8.2,8.3 and 8.4. An object n, in order to be within the observer o's neighbourhood, must fall within its viewing frustum i.e. the polar and azimuthal angle between observer's view direction and the object should be less or equal than s_h and s_v respectively, and the distance between them should be less than the maximum sight distance of the observer s_d. Let $\mathbf{p'_n} = \langle p_n^x - v_o^x, p_n^y - v_o^y, p_n^z - v_o^z \rangle$ the position vector of the object n in the o's frame of reference, then n is o's neighbor if and only if the followings hold:

$$\delta_s = ||\mathbf{p_o} - \mathbf{p_n}||,\ \delta_s \le d_s \tag{8.2}$$

$$-\frac{s_h}{2} \le \theta \le \frac{s_h}{2} \tag{8.3}$$

$$-\frac{s_v}{2} \le \phi \le \frac{s_v}{2} \tag{8.4}$$

where:

$$\phi = \arccos\left(\frac{p_n'^z}{\sqrt{(p_n'^x)^2 + (p_n'^y)^2 + (p_n'^z)^2}}\right)$$
$$\theta = atan2\left(\frac{p_n'^y}{p_n'^x}\right)$$

We use the two arguments tan version in order to gather information on the signs of the inputs in order to return the appropriate quadrant of the computed angle, which is not possible for the single argument arctan function. Additionally, the ordinary arctan suffers when required to produce $\pm\frac{\pi}{2}$ angle [2].

$$\theta = \begin{cases} \frac{\pi}{2}, & \text{if } p_n'^x = 0,\ p_n'^y > 0 \\ \frac{3\pi}{2}, & \text{if } p_n'^x = 0,\ p_n'^y < 0 \\ \text{undefined} & \text{if } p_n'^x = 0, p_n'^y = 0 \\ \arctan\frac{p_n'^y}{p_n'^x} & \text{if } p_n'^x > 0,\ p_n'^y \ge 0 \\ \arctan\frac{p_n'^y}{p_n'^x} + 2\pi & \text{if } p_n'^x > 0,\ p_n'^y < 0,\ \textit{or } p_n'^x < 0,\ p_n'^y > 0 \\ \arctan\frac{p_n'^y}{p_n'^x} + \pi & \text{if } p_n'^x < 0,\ p_n'^y \le 0 \end{cases}$$

2 Computing angle between x and y axis would require a division by zero.

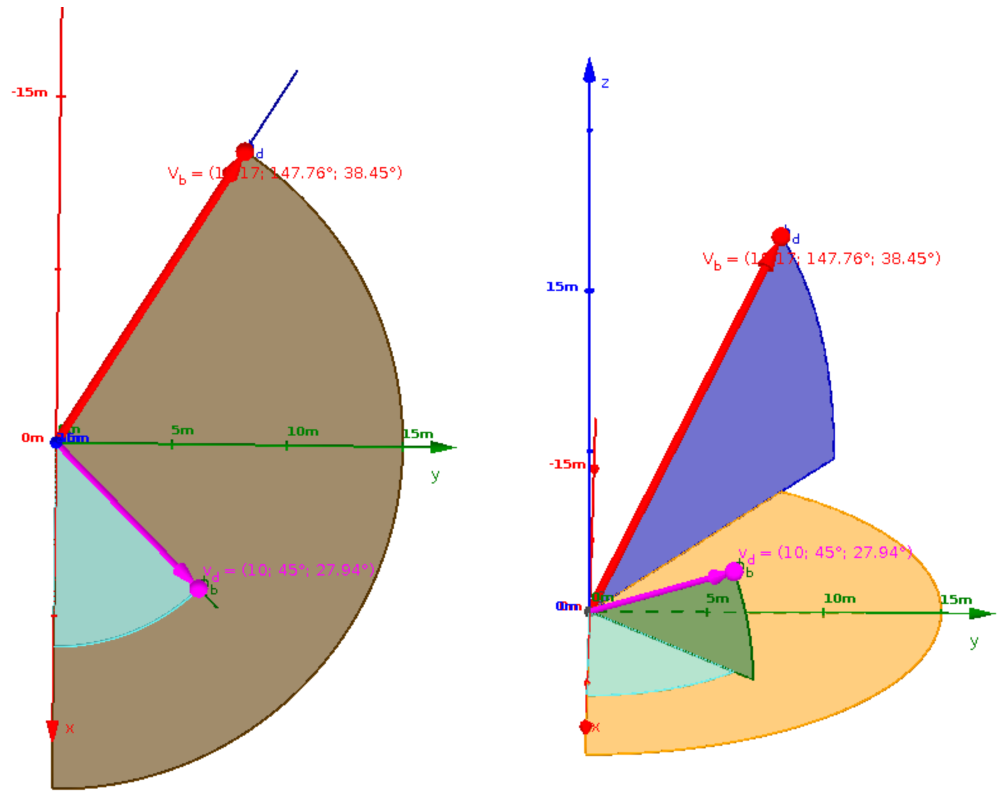


FIGURE 8.3: Example of spherical coordinate representation of two vectors. .V_d for instance is a vector of magnitute 10 and polar and aximuthal angles 45° and 27.94° respectively.(a) and (b) show views from the top and from the side of the reference frame.

8.1.4 *Cohesion*

Cohesion is a flight behaviour that attracts a bird to centroid of its perceived neighbourhood. In formal terms $\vec{C}_b^i$, the bird b's centroid at time i, is given by:

$$\vec{C}_b^i = \frac{1}{|\mathcal{N}_b|}\sum_{n=1}^{|\mathcal{N}_b|}\vec{p}_n\frac{d_{i,j}}{d_s} \tag{8.5}$$

where:

1. $\mathcal{N}_b$ the set of birds in b's FOV.
2. $\vec{p}_n$ is the position of $n \in \mathcal{N}_b$
3. $d_{i,j}$ is the distance between bird b and its neighbour c

The b's cohesion vector $\vec{v}_c$ is then defined as follows.

$$\vec{v}_c^i = \begin{cases} \frac{\vec{C}_b^i - \vec{p}_b}{||\vec{C}_b^i - \vec{p}_b||} + a, & \text{if } 0 < |\vec{v}_d| \leq v_p \\ v_p, & \text{otherwise} \end{cases} \tag{8.6}$$

8.1.5 *Separation*

A bird try to keep certain distance between itself and its neighbors. Bird b's separation velocity $\vec{S}_b^i$ at time i is given by:

$$\vec{S}_b^i = \begin{cases} \left[\sum_{j \in \mathcal{N}_b} \frac{\vec{p}_b - \vec{p}_j}{||\vec{p}_b - \vec{p}_j||} f_s\right] + a, & \text{if } 0 < |\vec{S}_i| \leq v_p \\ v_p, & \text{otherwise} \end{cases} \tag{8.7}$$

where:

1. $\mathcal{N}_b$ is the set of neighbours,
2. $f_s = \begin{cases} 0 & \text{if } d_{i,j} > d_{min} \\ 1 - \frac{d_{i,j}}{d_{min}} & \text{otherwise} \end{cases}$

8.1.6 *Alignment*

A bird tries to match its velocity (speed and heading) with those of nearby flockmates. This behaviour is called velocity alignment. Real life birds only consider a relatively small number flockmates while performing this behaviour and it usually is about seven neighbours [161]. In formal terms, bird b's alignment $\vec{A}_b^i$ is here defined as

$$\vec{A}_i = \left[\sum_{j \in \mathcal{N}_b'} \vec{v}_j \, f_a\right] + a, \;\; 0 < |\vec{A}_i| \leq v_p \tag{8.8}$$

where:

1. $\mathcal{N}_b' \subseteq \mathcal{N}_b$ is the set of birds considered by b for the alignment (e.g. the nearests).
2. $\vec{v}_j$ is the j's velocity.
3. f_a is the alignment coefficient. Let $d_{i,j}$ the distance between bird i and j then f_a is given by:

$$f_a = \begin{cases} 0 & \text{, if } d_{i,j} > d_a \\ 1 - \frac{d_{i,j}}{d_a} & \text{, otherwise} \end{cases}$$

8.1.7 *Other species and predator avoidance*

ACIADDRI is a multi-agent with multiple group model where each group correspond to a different bird's species or to the group of predators. Different species interaction is described in section 1 and predator avoidance in section 2.

1. **Other species avoidance** This behaviour is similar to *separation* (see section 8.1.5) with the difference that a only birds from other species are taken into consideration. In formal terms the other specie avoidance vector $\vec{\tau}_i$ is given by:

$$\vec{\tau}_i = \left[\sum_{j \in \mathcal{N}_b} \frac{\vec{p}_b - \vec{p}_j}{||\vec{p}_b - \vec{p}_j||} f_s \right] + a \tag{8.9}$$

where:

a) $\mathcal{N}_b$ is the number of neighbours of specie different from the one of b,

b) $f_s = \begin{cases} 0 & \text{if } d_{i,j} > r_s \\ 1 - \frac{d_{i,j}}{r_s} & \text{otherwise} \end{cases}$

2. **Predator avoidance** The predator avoidance vector is computed by taking in consideration position and velocity (speed and heading) of all the predators within the bird's FOV. Intuitively birds will flee from the future (step $i+1$) centroid of predators's position [162]. Formally the predator avoidance vector $\vec{\Gamma}_b^i$ is defined as follows:

$$\vec{\Gamma}_b^i = \left[\sum_{j \in \mathcal{P}_b} \frac{\vec{p}_i - (\vec{p}_j + \vec{v}_j)}{||\vec{p}_i - (\vec{p}_j + \vec{v}_j)||} f_p \right] + a, \;\; 0 < |\vec{\Gamma}_i| \leq v_p \tag{8.10}$$

where:

a) $\mathcal{P}_b$ is the b's set of predators

b) $f_p = \begin{cases} 0 & \text{if } d_{i,j} > r_p \\ 1 - \frac{d_{i,j}}{r_p} & \text{otherwise} \end{cases}$
is the predator avoidance coefficient.

8.1.8 *Wandering*

When the neighbourhood of a bird is empty it flies pseudo-randomly in the space. This kind of behaviour is called *wandering*. Wandering is obtained combining a current and a random direction

$$\vec{\omega}_b^i = \begin{cases} 0 & \text{if } \mathcal{N}_b \neq \emptyset \\ \frac{\vec{s}}{|\vec{s}|} w_r + w_d, \; 0 < |\vec{\omega}_i| \leq v_p & \text{otherwise .} \end{cases}$$

8.2 GPGPU PARALLEL IMPLEMENTATION AND RESULTS

In this work we adopt GPUs and the CUDA framework to accelerate the flocking simulation of a large number of boids using the the model presented in section 8.1 in an environment with a number of agents up to 5×10^6. According to the APOD development methodology we produced two different parallel versions, both sharing the high-level implementation structure that consists in (the well-known host-managed accelerated program structure):

- Initialization of data structures on *CPU*
- Data transfer from ***CPU* to *GPU***

- Kernels execution on *GPU*
- Copying the result back from ***GPU* to *CPU***

The parallelization strategy is designed with the purpose to avoid as much as possible the very undesirable data copy from *host to device* , or vice versa [163] [7] [116]. The computation of $\vec{p}_b^{\,i+1}$ and $\vec{v}_b^{\,i+1}$ is entirely performed on GPU and implemented as composition of CUDA kernels. Moreover Parameters are stored in constant memory for fast access. An OpenGL 3D visualization tool comes with the simulation system and permits real time and interactive rendering of the flocking model.

8.2.1 *Naïve version*

In this version each agent is mapped to a CUDA thread organized in a 1D block-grid structure. All data resides in global memory and user managed cache (shared memory) remains unused. Due to the high parallel nature of the simulation, although its simplicity, this version already gives rise to a speedup of $\approx 20\times$. In addition, an optimization was carried out by considering the *If-Divergence mitigation*. As known, thread divergence is a well known issue, that inhibits full parallelism at warp level. Two threads are said to diverge if they belong to the same warp but execute different instructions[3]. If some threads in a warp evaluate a conditional to *true* and others to *false*, then threads will branch to different instructions paths and those paths are executed in *serial manner*[4] [69]. As a consequence, this serialization may result in significant performance loss.

A series of workarounds have been implemented in order to mitigate this problem and more specifically, a number of if construct have been substituted with an equivalent arithmetical operation that are performed by all threads and preserves the original semantic of the code. Listings 8.1 and 8.2 show an example of such code transformation.

```
private var;
if(threadIdx.x > 16) then
var:= A
else
var:= B
```

LISTING 8.1: Example of thread divergent code.

```
bool c = threadIdx.x > N
private var;
var:= c*A + !c*B
```

LISTING 8.2: If mitigated version of the listing 8.1

8.2.2 *Shared memory version*

This version exploits the shared memory in order to cache birds's frequently accessed data. Shared memory is much faster than global memory but is of limited capacity (and depends on compute capability of the device, 48*KB* in the *GTX980*) [164], only accessible at block

3 Common code constructs that usually cause thread divergence are conditionals that depends on thread-id variable

4 It is important to stress that serial execution happens only when thread of the same warps diverge.

level and cleared at each kernel invocation. This implies a number of restrictions on its usage, namely:

1. it has to be initialized (i.e. requires a global memory access, see line 7 in listing 8.3),
2. limited size of data available per thread,
3. cannot be used to share data between threads of different blocks.

The adopted strategy divides the computation in a number of phases that depend on the chosen block size. Each phase can be then performed exploiting the fast memory as shown in listing 8.3. The above points 2 and 3 are the main reason for the division in phases. Moreover the constraint posed by ACIADDRI that requires that each bird know about all the other birds (to decide if it falls within its FOV for instance) make impossible to load all the data in shared memory [165, 166].

```
1  extern __shared__ Bird shBird[];
2  uint loop = NBIRDS/BLOCK_X  + (NBIRDS % BLOCK_X ? 1 : 0);
3  #pragma unroll
4  for (uint i = 0; i < loop; i++) {
5  int idx = i * blockDim.x + threadIdx.x;
6  if(idx<NBIRDS)
7  shBird[threadIdx.x]=birds[idx];
8  
9  ...
10 COMPUTATION USING SM shBird
11 ...
```

LISTING 8.3: Main loop of the agent function kernel. The loop variable represents the number of phases which one at time use SM to store data of a portion of the whole bird population.

Three devices were adopted for testing different CUDA version of the model: the high-end GTX 980, a GT 635M and a low-end mobile chip (see table 8.3 for further details).

In order to ensure the correctness of the parallelization the output of each parallel version were matched against the corresponding serial output. In particular, the sequential CPU version is identical to the version that was developed for the GPUs, that is, no optimizations were adopted in the former version. In practice, at every step, the CA space array is scrolled and the transition function applied to each cell of the CA where bird is present.

Name	Comp. Capability	RAM	SM-Clock	# cores
GT 653M	2.1	1024MB	675 MHz	635
GTX 980	5.5	4096MB	1216 MHz	2048
TESLA K40	5.2	12288MB	875 MHz	2880

TABLE 8.3: Hardware utilized for experiments

Different tests were carried out regarding both parallelizations described in the previous sections, and by considering different number of boids and an environment composed of $1000 \times 1000 \times 1000$ cells. Each simulation was carried out for 10^4 time steps.

Tables 8.4 and 8.5 shows the execution times of the naïve and shared memory version, respectively.

# birds	Sequential	GT 635M (×)	GTX 980 (×)
1024	263.9	29.1, (9.07)	10.5, (25.13)
5120	4913.0	574.4, (8.55)	51.7, (95, 02)
10240	19074.5	2241.6, (8.51)	109.0, (174.99)
15360	43332.3	5004.7, (8.65)	235.2, (184.23)
20480	86065.7	8868.9, (9.70)	312.5, (275.408)
40960	452423.1	-	1023.8, (441.90)
81920	1966134.9	-	3663.5, (536.70)
163840	8003173.0	-	14877.4, (537.94)
327680	35815012.0	-	58003.0, (617.47)

TABLE 8.4: Timing (in seconds) and speed-ups for the Parallel CUDA Naïve implementation

# birds	Sequential	GT 635M	GTX 980
1024	263.9	19.9	7.9
5120	4913.0	366.3	34.6
10240	19074.5	1398.7	96.5
15360	43332.3	3110.0	154.9
20480	86065.7	5522.0	280.8
40960	452423.1	-	825.4
81920	1966134.9	-	3307.2 -
163840	8003173.0	-	13565.9
327680	35815012.0	-	54113.4

TABLE 8.5: Timing (in seconds) for the Parallel *shared memory* CUDA implementation

Timings reported for the considered GPU devices indicate their full suitability for parallelizing multi-agent models, demonstrating the effectiveness of GPGPU to cut down computational time. In fact, the adoption of the CUDA technology has produced dramatic improvements in model speedup on the considered hardware up to 617x.

8.3 CONCLUSIONS AND FUTURE OUTCOMES

Starting from Reynolds's behavioural model, we have here presented a preliminary multi-agent and multiple group approach for bird flocking, together with an efficient implementation by means of the CUDA framework. Experiments carried out by adopting different hardware have proven the full suitability of the GPGPU paradigm for efficiently simulating multi-agent systems. Although our model describes bird movement adequately, future versions of ACIADDRI can take into account more behaviours by adding some parameters to improve the bird's flight modelling [167–169] as the introduction of stall velocity (that represents the velocity's lower bound) or wing's length and width (for thrust and lift forces computation), to better adhere with aerodynamics theory.

In current ACIADDRI implementation, all the computational intensive computations are carried out on the device and only final results are sent back to the host. When the proper mode is active, data is transferred from GPU to CPU at fixed times steps for visualization purposes. This results in additional time costs related to both the transfer and rendering on CPU. In order to avoid this issue, the adoption of OpenGL / CUDA interoperability will be investigated, permitting to directly and effectively copy data from device memory to the GPU display buffer to avoid the aforementioned additional costs, resulting in an more efficient solution.

Eventually, future developments will also regard model improvement, such as the possibility of the environment to contain obstacles, and the usage of multi-GPU hardware [136, 170], which can further scale the performance of the application and dramatically speed up the overall simulation process.

9 CONCLUSION

Forse l'immobilità delle cose intorno a noi è loro imposta dalla nostra certezza che sono esse e non altre, dall'immobilità del nostro pensiero nei loro confronti.

— Marcel Proust

Computers performance improvements have usually come from adding more transistors onto silicon or increasing the clock speed of the chips. As stated by the Moore's law, computer systems performance has growth steadily since the 70's at a pace of almost a twofold improvement every two years. But this trend is not sustainable since cramming more transistors or increasing the clock speed require more power, which in turn generates more heat. Modern chips have already a ratio of heat over cm^2 that is higher of that of nuclear reactor core and transistors size is also hard to shrink since their size is already almost of the same order of magnitude of a single atom.

Demand for speed did not stop over the years and thus in order to be able to create computer systems able to tackle the challenges posed by the modern big-data and scientific computing fields. it is necessary to use multiple computing cores and nodes concurrently. Parallel computing is ubiquitous as even low-end smartphones feature, multi-core processors but parallel dedicated machines have evolved over years into complex and hardware heterogeneous agglomerate of computing devices in order to be able to solve bigger and bigger instances. Programming this ecosystem of devices, each with its own peculiarity in terms of hardware architecture and programming model and tools, at his full efficiency is notoriously hard, especially for non specialized computer scientist because specialized knowledge (algorithms, tools, programming languages, systems tools, high-speed networking, etc.) is absolutely necessary.

This work aimed at the design of a programming abstraction for seamless implementation of numerical methods on regular grid targeting a plethora of different parallel computer architectures: from commodity PC, to large clusters of accelerators. The `OpenCAL` framework has been developed which exposes a domain specific language for the definition of a large class of numerical models and their subsequent deployment on the targeted machines. At this stage of development there are a number of specialized implementation of `OpenCAL` each targeting a different architecture (or a mix of them). Each version was designed to be the most reliable and fast possible and, for this purpose, the C/C++ language was adopted and efficient data types and algorithms considered. In particular, also to permit a more straightforward OpenCL parallelization, linearized arrays were adopted to represent both one-dimensional and higher order structures like substates and neighborhoods.

Though preliminary, obtained results confirm correctness and efficiency of the different `OpenCAL` versions here presented, by highlighting their goodness for numerical model development of complex systems in the field of Scientific Computing and their execution on parallel heterogeneous devices. Among all versions, `OpenCAL-CLUST` targets distributed memory machines and multi-accelerators architectures. Results show that `OpenCAL` allows deploying numerical applications on regular grid on machines composed by several computational nodes interconnected by network each armed with multiple accelerators. Thanks to the adoption of `OpenCL`, different kinds of accelerators can be employed seamlessly and efficently. The performance benchmarks that have been used to test OpenCAL-CLUST show that it effectively use the computational power of multiple devices in order to speedup the computation.

As regarding future developments, `OpenCAL` will be extended allowing domain decomposition on multiple dimensions. As shown in Section 6.1.2, decomposing the domain on a single dimension is not always optimal, as in order to obtain better load balancing among the devices different decomposition may be necessary. The programmer would be able to decompose the domain multidimensional cubic portions and assign one to each available device. Another important issue that will be addressed is that the current implementation serializes communications at each step execution taking advantages of possible computation-communication overlapping. Another limitation that the current implementation exposed is that boundaries are always exchanged among intra-node devices. This might not be optimal in cases where boundaries grid cells between two devices or nodes do not change and thus, the communication of such cells avoided. This mechanism can be accomplished by performing boundaries exchange only if there is a relevant update i.e. by means of the so called *dirty-bits* mechanism. A scaling benchmark will be performed on a proper *HPC* cluster with at least 16 nodes interconnected by fast network (`Infiniband` et. similia for instance) in order to obtain information about the scalability of `OpenCAL` as the number of nodes is increased. Nevertheless, a fine tuning of underlying data structures and algorithms. As regard the OpenCL implementation, the seamless management of GPUs local memory will be introduced in the next releases. Subsequent releases will also progressively support further computational paradigms, like the Lattice Boltzmann, the Smoothed Particle Hydrodynamics (SPH), as well as other mesh-free numerical methods, with the aim to become a general software abstraction layer for computation.

The `OpenCAL` software libraries, together with a comprehensive installation and user manual accompanied by numerous examples, are currently freely available on GitHub, at `https://github.com/OpenCALTeam/opencal`.

A CUDA BALLOTTING INTRINSIC STREAM COMPACTION

An efficient implementation of stream compaction on SIMD processors based on *ballotting* instructions is here presented and based on the followings works: [171, 172].

A.1 INTRODUCTION

Stream compaction/reduction/scan is commonly referred to as the operation of removing unwanted elements in a collection (see Figure A.1). More formally, we have a list of element $A_{0 \dots N}$ of N elements and a predicate $p : A \rightarrow \{True, False\}$ that bisects A in wanted and unwanted elements (some of which satisfy the predicates p while others don't). The stream compaction of A under p is an array $B = \{x \in A | p(x) = True\}$. Sometimes it is sufficient to return $B_{0 \dots N}$ s.t. all valid (suppose $M \leq N$) elements are grouped at the first M position of B. A more general and rigorous definition of prefix-sum is the following: given an associative operator $\Diamond$, a vector $V_{0 \dots N}$ of N elements and an identity element I, the scan operation returns a vector

$$P = \{I, v_0, (v_0 \Diamond v_1), (v_0 \Diamond v_1 \Diamond v_2), \dots, (v_0 \Diamond v_1 \Diamond \dots \Diamond v_{n-2})\}$$

.

Stream reduction is a key building block in several important algorithms, especially in the field of parallel programming where it is not uncommon to have large and sparse data structures to process such as in the parallel breadth tree traversing, ray tracing problems, etc. Sparsity can often be the cause of performance degradation of the overall parallel algorithm and it is often the cause of load imbalance/communication imbalance. As an example, the stream compacting with the predicate $p(x) = x > 0$ the following array of twelve integers $A = \{1, 0, 0, 0, 4, 3, 2, 0, 6, 8, 9, 0\}$ would produce $B = \{1, 4, 3, 2, 6, 8, 9\}$.

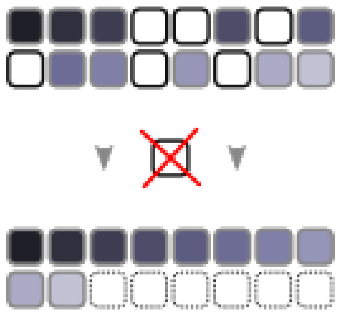


Figure A.1: Stream Compaction consists in removing all the elements for which the predicate is not satisfied. In this case the predicate is: *color* $\neq$ *white*.

A.1.1 *Serial Algorithm*

Serial implementation in a single thread is straightforward:

```
template <typename T>
void serialCompact(T* input, T*output, uint length,bool (*predicate)(T)){
  uint j=0;
  for(uint i=0;i<length;i++)
    if(predicate(input[i])){
      output[j] = input[i];
      j++;
    }
}
```

LISTING A.1: Naive Serial Stream Compaction Algorithm

Elements satisfying the predicate are pushed into an output buffer. The algorithm can be easily implemented as a one line filter operation using the `copy_it` operation in `C++`.

A.1.2 *Parallel Algorithm*

The parallel implementation is more challenging and the most effective parallel implementations produced so far (Thrust [173], Chag:pp [171]) are mainly based on the computation of the so called *exclusive* or *inclusive* prefix-sum. The exclusive prefix-sum on a vector $V_{0...N}$ with validity test p consists in producing a vector $S_{0...N}$ s.t. $S_0 = 0$ and $S_i = k$ where k is the number of valid elements strictly preceding (or up to the element i in case of inclusive prefix sum) the element of i. The inclusive prefix sum can be obtained from the exclusive one using the following: let $A = \{0, a_1, \dots, a_{n-1}\}$ the vector of elements and $E = \{0, e_1, \dots, e_{n-1}\}$ its exclusive prefix sum then $I = \{e_1, e_2 \dots, e_{n-1} + p(a_{n-1})\}$ is the corresponding inclusive prefix sum.

Suppose P is the number of processors and $N, N > P$, is the size of the vector. The input stream is divided in sub-streams of size of size S. The parallel algorithm is divided in three distinct phases:

1. Each processor p_i counts the number of valid elements independently in its sub-stream and saves this value in $procCount[p_i]$
2. A prefix sum operation is performed on the sub-array $procCount$ producing the vector $procOffset_{0...P}$
3. Each processor can flush out its valid elements independently from the others in the correct location (at the correct offset) $output[procOffset[p_i] + currentValidElm]$

```
//phase1
for each processor p in parallel
  int count=0;
  for(int i=0;i<S;i++)
    if(valid(input_p[i])
      count++;
```

```

procCount[p]=count;

//phase 2
procOffset = prefixSum(procCount)

//phase3
for each processor p in parallel
  int j=0;
  for(int i=0;i<S;i++)
    if(valid(input_p[i]){
      output[procOffset[p] + j ]
      j++;
    }
}
```

LISTING A.2: Naive Parallel Stream Compaction Algorithm Pseudocode

It is worth noticing that phase 2 can also be carried out in parallel and that its implementation on SIMT hardware is not straightforward and is not discussed here. We will then use available implementation as the one shipped with the Thrust library. In CUDA this phase load is usually negligible with respect to the other two, as the number of processors P is usually several orders of magnitude smaller than N.

A.2 SIMT/BALLOTTING INSTRUCTION IMPLEMENTATION

A.2.1 *CUDA Hardware*

GPUs hardware is made of a number of order of tens streaming multiprocessors (SM), that can be considered as SIMD processors made up of streaming processor (the SIMD lanes, SP). For instance, the newest 2017 Volta Architecture introduced by NVIDIA counts 84 SM and 64 SP per SM, for a total of 5376 cores (see Figure A.2 and A.3). SM schedule kernel execution in a SIMD fashion in groups of 32 threads (a warp) which in turn perform the same instructions in a synchronized manner. SM are not scalar processors and hence, using the parallel approach described in Section A.1.2, where each SM is considered as a single scalar processor p would result in poor performance due to the large number of idle lanes lanes $84 * (64 - 1) = 5376 - 84$. In this implementation each SM is entirely used to perform the block counting of valid elements and to finally compact the input.

A.2.2 *Phase* 1

The pseudocode A.3 shows how block counting can be performed on a SIMT hardware effectively using all computing lanes.

```
//input is a SM private vector containing the portion of the //original input
   to be processed by the processor
for each SM processor  p in parallel
  lanesCount[0]=0;
```

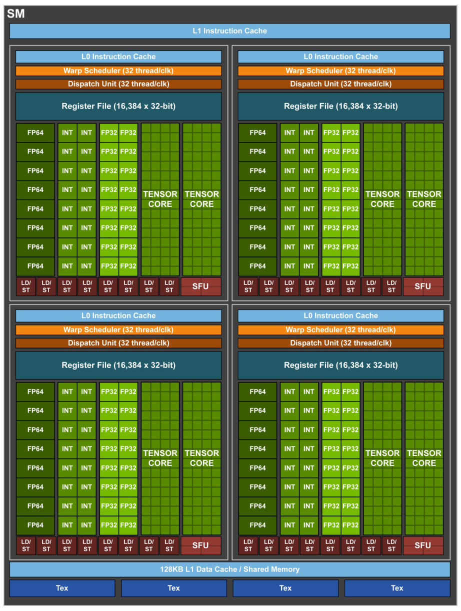

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Figure A.2: NVIDIA GPU Volta SM Architecture.

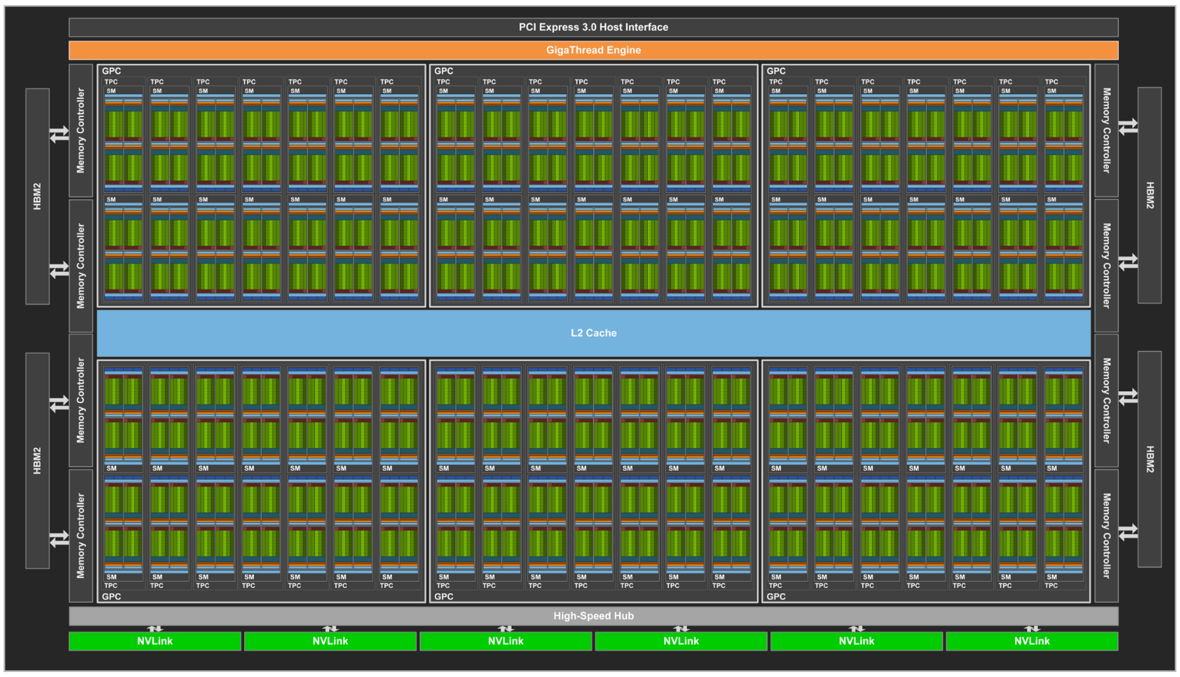


Figure A.3: Volta GV100 Full GPU with 84 SM Units.

```
  ...
  lanesCount[32]=0;
for(int i=0;i&amp;lt;S;i+=32)
  for each SIMD lane s in parallel
    if(predicate[input[i+s])
      lanesCount[s]++;

procCount[p] = reduce(+,lanesCount);
```

Listing A.3: Stream Compaction phase 1 on SIMT processor

With the introduction of the intrinsic function `__syncthreads_count(predicate)` this phase is easier (compared to the pseudo-code mentioned above) to implement and results in a more efficient and faster execution due to specialized transistors to execute the function. This instruction synchronizes threads at a block level and takes an integer as predicate. It returns to **all threads of the block** the number of non-zero predicates passed to it by all threads of the block. For instance, suppose that a kernel is executed only by one block made up of four threads and that each thread is calling `__syncthreads_count()` as shown in Listing A.4:

```
//thread0
int BC=__syncthreads_count(1)
//thread1
int BC=__syncthreads_count(0)
//thread2
int BC=__syncthreads_count(1)
//thread3
int BC=__syncthreads_count(0)
```

Listing A.4: `__syncthreads_count()` calls from threads within a block. 1 and 0 represents predicate expressions.

At this point, each thread t_i will own its private copy of the BC variable containing the value 2. This kind of operation is exactly what is needed in order to count the number of valid elements per block. The previous approach can be summarized as shown in listing A.5.

```
template <typename T,typename Predicate>
__global__ void computeBlockCounts(T* d_input,int length,int*d_BlockCounts,Predicate predicate){

int idx = threadIdx.x + blockIdx.x*blockDim.x;
if(idx < length){
  int pred = predicate(d_input[idx]);
  int BC=__syncthreads_count(pred);
  if(threadIdx.x==0)
    d_BlockCounts[blockIdx.x]=BC;
}
```

Listing A.5: Phase 1 implemented using ballotting intrinsic built-in functions.

A.2.3 *Phase 2*

While the output of phase 1 is a count of valid elements per block, phase 2 takes care of computing a prefix-sum among the block counters, whose number is much smaller than the original input array. For the sake of brevity and since it does not affect the overall performance because this phase is not an hot-spot for the algorithm,`Thrust` prefix sum is employed here in order to produce a vector of offsets. Offset i indicates how many valid elements will be pushed by blocks $j < i$, thus actually indicating what is the block's i writing index in the output array. Listing A.6 shows how to perform a prefix-sum on blocks' counter, output of phase 1.

```
//prefix sum thrust call
thrust::exclusive_scan(blockCounters, blockCounters + numBlocks, blockOffsets);
```

Listing A.6: Phase 2 implemented using a scan operation in `Thrust`.

A.2.4 *Phase* 3

The most elaborate part of the algorithm takes the *to be compacted* stream as input, and the block offsets computed during the previous phases outputting the compacted stream. It is based on the intra warp voting function `__ballot()`, ans `__popc()` procedure and a bit of manipulation.

The `unsigned int __ballot(int predicate);` function. evaluates `predicate` for all threads of the warp and returns an integer whose i^{th} bit is set if and only if `predicate` evaluates to **non-zero** for the i^{th} thread of the warp.

`__popc(int number)` function returns the number of set bits in its parameter.

It is worth noticing that the underlying matching has to support a word size which is not smaller that the size of the warp. This could be a problem if the warp-size is increased in the future releases of CUDA (even if this is unlikely to happen).

As shown in A.7, this phase starts computing a per warp offset (intra warp prefix sum) offset. This means that the offset computed for the last thread of the warp is the warp's number of valid elements. Each thread stores its offset in a register while warp's valid elements are stored in a **shared memory** buffer (`warpTotal[warpIDX]`).

```
int pred= predicate(threadInput);
int w_i = threadIdx.x/warpSize; //warp index
int w_l = idx % warpSize;//thread index within a warp (warp lane)
int t_m = INT_MAX >> (warpSize-w_l-1);
int b = __ballot(pred) & t_m;
int t_u = __popc(b);
//last thread of the warp stores in Shared Memory
if(w_l==warpSize-1)
  warpTotals[w_i]=t_u + pred;
__syncthreads();
```

Listing A.7: Intra warp prefix-sum using ballotting intrisic

where `w_i` is the warp index within the block,`w_l` is the thread index within the warp and `t_m` is the thread mask, i.e. a number the only bits sets are the ones with an index less then `w_l`. `b` contains only set bits corresponding to the validity of predicated of threads of lower index. `__popc(int number)` is then used to retrieve the number of set bits. `t_u` is the warp offset of thread `w_l`. A block memory fence is necessary in order to ensure correctness for future access to the shared memory array `warpTotals`.

At this point an intra block scan operation is performed on $warpOffset$ in order to compute a per-block offset, as shown in the listing A.8. Assuming $warpsize >= \frac{blocksize}{warpsize}$, i.e. the number of warps in a block does not exceed the size of a single warp, this operation can be performed by a single warp. This scan operation is performed in $log_2(warpsize)$ steps because we are summing up numbers whose max value is *warpsize* (each warp cannot perform more write than its number of threads). Usually in CUDA this number is 32 hence bit-scan is performed in $log_2(32) = 5$ steps.

```
if(w_i==0 && w_l<blockDim.x/warpSize){
  int w_i_u=0;
  for(int j=0;j<log2(warpsize);j++){
    int b_j =__ballot( warpTotals[w_l] & pow2i(j) );
    w_i_u += (__popc(b_j & t_m)  ) << j;
  }
  warpTotals[w_l]=w_i_u;
}
```

LISTING A.8: `Per-block offset computation.`

where b_j is a number in which each bit is one if and only if the j^{th} bit of the j^{th} per-warp counter is one. Each warp then masks only the bits $i' < i$ (of the warps before it) and finally sum them up, effectively completing the prefix sum operation.

Knowing the *warp*, *block*, and *grid offsets* it is possible to flush out the valid elements from the input array at the correct locations in the output array. If the current thread is managing a valid element then it will flush it out at the following location: $output[t_u + warpTotals[w\_i] + blocksOffset[blockIdx.x]] = input[idx]$; which reads as: (thread's offset within its warp) + (thread's warp offset within the block) + (thread's block offset within the grid).

As an example let's suppose that a block of 4 warps produces the following `warpTotals`:

```
warpTotals[0] = 16 = 1 0 0 0 0
warpTotals[1] = 18 = 1 0 0 1 0
warpTotals[2] = 17 = 1 0 0 0 1
warpTotals[3] = 13 = 0 1 1 0 1
```

Note that warp 0 always produces zero as output because `t_m = 0 >= (b_j & t_m) =0`.

STEP 0

b_0 = `__ballot( warpTotals[w_l] & pow2i(0) )` = 12 (column zero)
w_1 = `__popc(12 & 1) << 0`$= 0 << 0 = 0$
w_2= `__popc(12 & 3) << 0` $= 0 << 0 = 0$
w_3= `__popc(12 & 7) << 0` $= 1 << 0 = 1$

STEP 1

b_0 = `__ballot( warpTotals[w_l] & pow2i(1) )` = 2 (column one)
w_1 = `__popc(2 & 1) << 1`$= 0 << 1 = 0$
w_2= `__popc(2 & 3) << 1` $= 0 << 1 = 2$
w_3= `__popc(2 & 7) << 1` $= 1 << 1 = 2$

STEP 2

b_0 = `__ballot( warpTotals[w_l] & pow2i(2) )` = 8 (column two)
w_1 = `__popc(8 & 1) << 2`$= 0 << 2 = 0$

w_2= `__popc(8 & 3) << 2` $= 0 << 2 = 0$
w_3= `__popc(8 & 7) << 2` $= 1 << 2 = 0$

STEP 3
b_0 = `__ballot( warpTotals[w_l] & pow2i(2) )` = 8 (column three)
w_1 = `__popc(8 & 1) << 3`$= 0 << 3 = 0$
w_2= `__popc(8 & 3) << 3` $= 0 << 3 = 0$
w_3= `__popc(8 & 7) << 3` $= 1 << 3 = 0$

STEP 4
b_0 = `__ballot( warpTotals[w_l] & pow2i(4) )` = 7 (column four)
w_1 = `__popc(7 & 1) << 4`$= 16 << 4 = 0$
w_2= `__popc(7 & 3) << 4` $= 36 << 4 = 0$
w_3= `__popc(7 & 7) << 4` $= 48 << 4 = 1$

The final result for each warp is simply the sum of the $w_{i\,u}$ at all steps
$w_0 = 0 + 0 + 0 + 0 + 0 = 0$
$w_1 = 0 + 0 + 0 + 0 + 16 = 16$
$w_2 = 0 + 2 + 0 + 0 + 36 = 38 = 16 + 18 = 34$
$w_3 = 1 + 2 + 0 + 0 + 48 = 51 = 16 + 18 + 17 = 51$

B
HARDWARE - TECHNICAL SPECIFICATION

B.0.1 *Accelerators Specification*

Spec.	**K40**	**K20**	**GTX980**
CUDA Cores	2880	2496	2048
Core Clock	745-845 MHz	706 MHz	$1126 - 1216$ MHz
Memory Size	12 GB	5 GB	4 GB
Memory Clock Speed	3.0 GHz	2.6 GHz	1.753 GHz
Bandwith	288 GB/s	208 GB/s	224 GB/s
Interface	384 bit	308 bit	256 bit
API Supported	OpenACC, OpenCL 1.2	OpenACC, OpenCL 1.1	OpenACC, OpenCL 1.1

Table B.1: Technical Specification for the NVIDIA K40,K20 and GTX980 GPUs.

B.0.2 *Test Cluster HW Specification*

The following table reports the main technical specifications for the test cluster adopted for benchmarking the preliminary version of `OpenCAL-CLUST` (see Section 6.4). Nodes are interconnected to a *Cisco Catalyst 3750 Series* switch via standard GigaBit Ethernet. The real performance of the network has been tested using `iperf` Linux command line tool giving raise to a bidirectional bandwidth of $\approx 820 Gbit/s$.

Spec.	**Node 1**	**Node 2**
Core Count	20*cores* - 40*threads*	8*cores* - 16*threads*
Processor	*Intel(R) Xeon(R) E5-2650*	*Intel(R) Xeon(R) E5440*
Memory Size	32 GB	16 GB
Memory Clock Speed	2.133 GHz	0.667 GHz

Table B.2: Technical Specification for the two nodes composing the test cluster adopted for benchmarking `OpenCAL-CLUST`.

BIBLIOGRAPHY

1. Toffoli, T. Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics. *Physica D: Nonlinear Phenomena* **10**, 117 (1984).
2. T. Toffoli, N. M. *Cellular Automata Machines: A New Environment for Modeling* (MIT Press, 1987).
3. Frisch, U., Hasslacher, B. & Pomeau, Y. Lattice-Gas Automata for the Navier-Stokes Equation. *Phys. Rev. Lett.* **56**, 1505 (1986).
4. S. Succi R. Benzi, F. H. The lattice Boltzmann equation: A new tool for computational fluid dynamics. *Physica* **47**, 219 (1991).
5. Gregorio, S. D. & Serra, R. An empirical method for modelling and simulating some complex macroscopic phenomena by cellular automata. *Future Generation Computer Systems* **16**, 259 (1999).
6. Sirakoulis, G., Karafyllidis, I. & Thanailakis, A. A cellular automaton model for the effects of population movement and vaccination on epidemic propagation. *Ecological Modelling* **133**, 209 (2000).
7. Spataro, D., D'Ambrosio, D., Filippone, G., Rongo, R., Spataro, W. & Marocco, D. The new SCIARA-fv3 numerical model and acceleration by GPGPU strategies. *The International Journal of High Performance Computing Applications* **31**, 163 (2017).
8. Crisci, G. M., Rongo, R., Di Gregorio, S. & Spataro, W. The simulation model SCIARA: the 1991 and 2001 lava flows at Mount Etna. *Journal of Volcanology and Geothermal Research* **132**, 253 (2004).
9. Spataro, W., Avolio, M. V., Lupiano, V., Trunfio, G. A., Rongo, R. & D'Ambrosio, D. The latest release of the lava flows simulation model SCIARA: First application to Mt Etna (Italy) and solution of the anisotropic flow direction problem on an ideal surface. *Procedia Computer Science* **1**, 17 (2010).
10. Neumann, V. Theory of Self Reproducing Automata (1966).
11. Conway, J. The game of life. *Scientific American* (1970).
12. Thatcher, J. Universality in the Von Neumann Cellular mode. *A.W. Burks (Ed.), Essays on Cellular Automata*, 103 (1970).
13. N. Margolus T. Toffoli, G. V. Cellular Automata supercomputers for fluid-dynamics modelling. *Phys. Rev. Lett.* **56**, 1694 (1986).
14. Wolfram, S. Computation theory of cellular automata. *Communications in Mathematical Physics* **96**, 15 (1984).
15. Hopcroft, J. E., Motwani, R. & Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006).
16. Wolfram, S. Cellular automaton uids 1: Basics theory. *Journal of Statistical Physics* **45**, 471 (1986).

17. Wolfram, S. Statistical mechanics of cellular automata. *Reviews of Modern Physics* **55**, 471 (1983).
18. Arrighi, P., Schabanel, N. & Theyssier, G. Stochastic Cellular Automata: Correlations, Decidability and Simulations. *Fundam. Inf.* **126**, 121 (2013).
19. Chomsky, N. Three models for the description of language. *IRE Transactions on Information Theory* **2**, 113 (1956).
20. Wolfram, S. *A new kind of science* (Wolfram Medica, Inc, Champaign, 2002).
21. Cook, M. Universality in Elementary Cellular Automata. *Complex Systems* **15**, 1 (2004).
22. Karel Culik, S. Y. *Undecidability of CA Classification Schemes* (1998).
23. Langton, C. G. Computation at the edge of chaos: phase transitions and emergent computation. *Phys. D* **42**, 12 (1990).
24. Langton, C. *Computation at the edge of chaos* MA thesis (Univeristy of Michigan, 1990).
25. Berlekamp, E., Conway, J. & Guy, R. *Winning Ways for your Mathematical Plays* (Academic, 1982).
26. Ilachinski, A. *Cellular Automata: A Discrete Universe* (World Scientific, Singapore, 2001).
27. D'Ambrosio, D., Filippone, G., Rongo, R., Spataro, W. & Trunfio, G. A. Cellular Automata and GPGPU: An Application to Lava Flow Modeling. *Int. J. Grid High Perform. Comput.* **4**, 30 (2012).
28. D'Ambrosio, D. & Spataro, W. Parallel evolutionary modelling of geological processes. *Parallel Computing* **33**, 186 (2007).
29. Trunfio, G. A., D'Ambrosio, D., Rongo, R., Spataro, W. & Di Gregorio, S. A New Algorithm for Simulating Wildfire Spread Through Cellular Automata. *ACM Trans. Model. Comput. Simul.* **22**, 6:1 (2011).
30. D'Ambrosio, D., Di Gregorio, S., Gabriele, S. & Gaudio, R. A cellular automata model for soil erosion by water. *Physics and Chemistry of the Earth, Part B: Hydrology, Oceans and Atmosphere* **26**, 33 (2001).
31. Mendicino, G., Senatore, A., Spezzano, G. & Straface, S. Three-dimensional unsaturated flow modeling using cellular automata. *Water Resources Research* **42**. W11419, n/a (2006).
32. Mairesse, J. & Marcovici, I. Around probabilistic cellular automata. *Theoretical Computer Science* **559**. Non-uniform Cellular Automata, 42 (2014).
33. G., V. Simulating physics with cellular automata. *Physica* **D 10**, 96 (1984).
34. Wu, F. Y. The Potts model. *Rev. Mod. Phys.* **54**, 235 (1 1982).
35. LeVeque, R. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems (Classics in Applied Mathematics Classics in Applied Mathemat)* (Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007).
36. Bhattacharya, M. C. Finite-difference solutions of partial differential equations. *Communications in Applied Numerical Methods* **6**, 173 (1990).
37. Mazumder, S. in *Numerical Methods for Partial Differential Equations* (ed Mazumder, S.) 103 (Academic Press, 2016).

38. Larsson, S. e. a. *Partial Differential Equations with Numerical Methods* 256 pp. (Springer, 2004).

39. McOwen, R. *Partial Differential Equations: Methods and Applications* 452 pp. (PRENTICE HALL, 2002).

40. Estep, D., Hansbo, P., Johnson, C. & Eriksson, K. *Computational Differential Equations* (Cambridge University Press, New York, NY, USA, 1996).

41. 1994. *Analysis of Numerical Methods* (Courier Corporation, Eugene Isaacson, Herbert Bishop Keller).

42. Anderson J. D., J. *Computational Fluid Dynamics: The Basics with Applications* (McGraw Hill, 1994).

43. J. Crank, P. N. A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type. *Advances in Computational Mathematics* (1996).

44. Datta, B. N. *Numerical Linear Algebra and Application* pp: 162 (SIAM, 2010).

45. Higham, N. J. *Accuracy and Stability of Numerical Algorithms* pp: 175 (SIAM, 2002).

46. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.* **28**, 135 (2002).

47. Kestur, S., Davis, J. D. & Williams, O. *BLAS Comparison on FPGA, CPU and GPU* in *Proceedings of the 2010 IEEE Annual Symposium on VLSI* (IEEE Computer Society, Washington, DC, USA, 2010), 288.

48. Tonti, E. A direct discrete formulation of field laws: The cell method. *CMES - Computer Modeling in Engineering and Sciences* **2**, 237 (2001).

49. Flynn, M. J. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers* **C-21**, 948 (1972).

50. Duncan, R. A survey of parallel computer architectures. *Computer* **23** (1990).

51. Fortes, J. & Wah., B. W. Systolic arrays - from design to implementation. *IEEE Computer*, 12 (1987).

52. Kung, S. On supercomputing with systolic/wawefront array procesors. *Proc IEEE. Trans. Computer*, 897 (1984).

53. Spurgeon, C. E. *Ethernet: The Definitive Guide* (O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000).

54. Shanley, T. *Infiniband* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002).

55. Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N. & Su, W.-K. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro* **15**, 29 (1995).

56. Forum, M. P. *MPI: A Message-Passing Interface Standard* tech. rep. (Knoxville, TN, USA, 1994).

57. Gropp, W., Lusk, E. & Thakur, R. *Using MPI-2: Advanced Features of the Message-Passing Interface* (MIT Press, Cambridge, MA, USA, 1999).

58. Chapman, B., Jost, G. & Pas, R. v. d. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)* (The MIT Press, Cambridge, MA, USA, 2007).

59. Nichols, B., Buttlar, D. & Farrell, J. P. *Pthreads Programming* (O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996).

60. Akenine-Moller, T., Moller, T. & Haines, E. *Real-Time Rendering* 2nd (A. K. Peters, Ltd., Natick, MA, USA, 2002).

61. Foster, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995).

62. Sodani, A. *Knights landing (KNL): 2nd Generation Intel xAE; Xeon Phi processor* in *2015 IEEE Hot Chips 27 Symposium (HCS)* (2015), 1.

63. Jun, H., Cho, J., Lee, K., Son, H. Y., Kim, K., Jin, H. & Kim, K. *HBM (High Bandwidth Memory) DRAM Technology and Architecture* in *2017 IEEE International Memory Workshop (IMW)* (2017), 1.

64. Strohmaier, E. *TOP500 Supercomputer* in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (ACM, Tampa, Florida, 2006).

65. Shirley, P., Ashikhmin, M. & Marschner, S. *Fundamentals of Computer Graphics* 2nd (A. K. Peters, Ltd., Natick, MA, USA, 2005).

66. Gaster, B., Howes, L., Kaeli, D. R., Mistry, P. & Schaa, D. *Heterogeneous Computing with OpenCL* 1st (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011).

67. Stone, J. E., Gohara, D. & Shi, G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test* **12**, 66 (2010).

68. Munshi, A., Gaster, B., Mattson, T. G., Fung, J. & Ginsburg, D. *OpenCL Programming Guide* 1st (Addison-Wesley Professional, 2011).

69. Nvidia. *CUDA C Programming Guide* (ed 5.0) (2016).

70. Golub, G. H. & Ortega, J. M. *Scientific computing: an introduction with parallel computing* (Academic Press, London, UK, 2014).

71. Mazumder, S. in *Numerical Methods for Partial Differential Equations* (ed Mazumder, S.) 1 (Academic Press, Cambridge, MA, USA, 2016).

72. Chaigne, A. & Askenfelt, A. Numerical simulations of piano strings. I. A physical model for a struck string using finite difference methods. *Journal of the Acoustical Society of America* **95**, 1112 (1994).

73. Brański, A. & Prdka, E. *Description of the room acoustic field with meshless methods* in *Proceedings - 7th Forum Acusticum 2014, Krakow, Poland* (2014).

74. Rana, P. & Bhargava, R. Flow and heat transfer of a nanofluid over a nonlinearly stretching sheet: A numerical study. *Communications in Nonlinear Science and Numerical Simulation* **17**, 212 (2012).

75. Şahin, H., Kocatepe, K., Kayikci, R. & Akar, N. Determination of unidirectional heat transfer coefficient during unsteady-state solidification at metal casting-chill interface. *Energy Conversion and Management* **47**, 19 (2006).

76. Chang, K.-S. & Song, C.-J. Interactive vortex shedding from a pair of circular cylinders in a transverse arrangement. *International Journal for Numerical Methods in Fluids* **11**, 317 (1990).

77. Deng, X., Min, Y., Mao, M., Liu, H., Tu, G. & Zhang, H. Further studies on Geometric Conservation Law and applications to high-order finite difference schemes with stationary grids. *Journal of Computational Physics* **239**, 90 (2013).

78. Hu, J., Wu, J., Wang, Y. & Yin, Y. *Electron behavior in hydrogen atom under electric fields* in. **2015-October** (2015), 640.

79. Farrokhabadi, A., Abadian, N., Rach, R. & Abadyan, M. Theoretical modeling of the Casimir force-induced instability in freestanding nanowires with circular cross-section. *Physica E: Low-Dimensional Systems and Nanostructures* **63**, 67 (2014).

80. Hutton, D. V. *Fundamentals of Finite Element Analysis* (McGraw-Hill Higher Education, 2003).

81. Moukalled, F., Mangani, L. & Darwish, M. *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM and Matlab* 1st (Springer Publishing Company, Incorporated, 2015).

82. Von Neumann, J. *Theory of Self-Reproducing Automata* (ed Burks, A. W.) (University of Illinois Press, Champaign, IL, USA, 1966).

83. Codd, E. F. *Cellular Automata* (Academic Press, Inc., Orlando, FL, USA, 1968).

84. Wolfram, S. Universality and complexity in cellular automata. *Physica D*, 1 (1984).

85. Langton, C. Computation at the edge of caos: phase transition and emergent computation. *Physica D*, 12 (1990).

86. Ninagawa, S. Dynamics of universal computation and 1/f noise in elementary cellular automata. *Chaos, Solitons and Fractals* **70**, 42 (2015).

87. Langton, C. Studying Artificial Life with Cellular Automata. *Physica D*, 120 (1986).

88. Beer, R. Autopoiesis and cognition in the game of life. *Artificial Life* **10**, 309 (2004).

89. Frish, U., Hasslacher, B. & Pomeau, Y. Lattice gas automata for the Navier-Stokes Equation. *Physical Review Letters* **56**, 1505 (1986).

90. McNamara, G. & Zanetti, G. Use of the Boltzmann equation to simulate lattice-gas automata. *Physical Review Letters* **61**, 2332 (1988).

91. Higuera, F. & Jimenez, J. Boltzmann approach to lattice gas simulations. *Europhysics Letters* **9**, 663 (1989).

92. Aidun, C. & Clausen, J. Lattice-boltzmann method for complex flows. *Annual Review of Fluid Mechanics* **42**, 439 (2010).

93. D'Ambrosio, D., Di Gregorio, S. & Iovine, G. Simulating debris flows through a hexagonal cellular automata model: SCIDDICA S3-hex. *Natural Hazards and Earth System Science* **3**, 545 (2003).

94. Avolio, M., Di Gregorio, S., Lupiano, V. & Mazzanti, P. SCIDDICA-SS3: a new version of cellular automata model for simulating fast moving landslides. *The Journal of Supercomputing* **65**, 682 (2013).

95. D'Ambrosio, D., Rongo, R., Spataro, W. & Trunfio, G. Meta-model assisted evolutionary optimization of cellular automata: An application to the SCIARA model. *Lecture Notes in Computer Science* **7204**, 533 (2012).

96. D'Ambrosio, D., Rongo, R., Spataro, W. & Trunfio, G. Optimizing cellular automata through a meta-model assisted memetic algorithm. *Lecture Notes in Computer Science* **7492**, 317 (2012).

97. Oliverio, M., Spataro, W., D'Ambrosio, D., Rongo, R., Spingola, G. & Trunfio, G. *OpenMP parallelization of the SCIARA Cellular Automata lava flow model: Performance analysis on shared-memory computers* in *Procedia Computer Science* **4** (2011), 271.

98. D'Ambrosio, D., Rongo, R., Spataro, W., Avolio, M. & Lupiano, V. Lava invasion susceptibility hazard mapping through cellular automata. *Lecture Notes in Computer Science* **4173**, 452 (2006).

99. Avolio, M., Crisci, G., Gregorio, S., Rongo, R., Spataro, W. & D' Ambrosio, D. Pyroclastic flows modelling using cellular automata. *Computers and Geosciences* **32**, 897 (2006).

100. Crisci, G., Di Gregorio, S., Rongo, R. & Spataro, W. PYR: A Cellular Automata model for pyroclastic flows and application to the 1991 Mt. Pinatubo eruption. *Future Generation Computer Systems* **21**, 1019 (2005).

101. Arca, B., Ghisu, T. & Trunfio, G. GPU-accelerated multi-objective optimization of fuel treatments for mitigating wildfire hazard. *Journal of Computational Science*, 258 (2015).

102. Avolio, M., Di Gregorio, S. & Trunfio, G. A randomized approach to improve the accuracy of wildfire simulations using cellular automata. *Journal of Cellular Automata* **9**, 209 (2014).

103. Mendicino, G., Pedace, J. & Senatore, A. Stability of an overland flow scheme in the framework of a fully coupled eco-hydrological model based on the Macroscopic Cellular Automata approach. *Communications in Nonlinear Science and Numerical Simulation* **21**, 128 (2015).

104. Ravazzani, G., Rametta, D. & Mancini, M. Macroscopic cellular automata for groundwater modelling: A first approach. *Environmental Modelling and Software* **26**, 634 (2011).

105. Cervarolo, G., Mendicino, G. & Senatore, A. A coupled ecohydrological-three-dimensional unsaturated flow model describing energy, H2O and CO2 fluxes. *Ecohydrology* **3**, 205 (2010).

106. Lubaś, R., Wąs, J. & Porzycki, J. Cellular Automata as the basis of effective and realistic agent-based models of crowd behavior. *The Journal of Supercomputing* **72**, 2170 (2016).

107. Wąs, J., Mróz, H. & Topa, P. GPGPU computing for microscopic simulations of crowd dynamics. *Computing and Informatics* **34**, 1418 (2015).

108. Wąs, J. & Lubaś, R. Towards realistic and effective Agent-based models of crowd dynamics. *Neurocomputing* **146**, 199 (2014).

109. Blecic, I., Cecchini, A. & Trunfio, G. How much past to see the future: a computational study in calibrating urban cellular automata. *International Journal of Geographical Information Science* **29**, 349 (2015).

110. Amritkar, A., Deb, S. & Tafti, D. Efficient parallel CFD-DEM simulations using OpenMP. *Journal of Computational Physics* **256**, 501 (2014).

111. Pop, A. & Cohen, A. OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs. *ACM Transactions on Architecture and Code Optimization* **9** (2013).

112. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. & Purcell, T. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* **26**, 80 (2007).

113. Blecic, I., Cecchini, A. & Trunfio, G. Cellular automata simulation of urban dynamics through GPGPU. *Journal of Supercomputing* **65**, 614 (2013).

114. D'Ambrosio, D., Filippone, G., Marocco, D., Rongo, R. & Spataro, W. Efficient application of GPGPU for lava flow hazard mapping. *Journal of Supercomputing* **65**, 630 (2013).

115. Di Gregorio, S., Filippone, G., Spataro, W. & Trunfio, G. Accelerating wildfire susceptibility mapping through GPGPU. *Journal of Parallel and Distributed Computing* **73**, 1183 (2013).

116. D'Ambrosio, D., Filippone, G., Rongo, R., Spataro, W. & Trunfio, G. Cellular automata and GPGPU: An application to lava flow modeling. *International Journal of Grid and High Performance Computing* **4**, 30 (2012).

117. Stone, J., Gohara, D. & Shi, G. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering* **12**, 66 (2010).

118. Macri, M., De Rango, A., Spataro, D., D'Ambrosio, D. & Spataro, W. *Efficient Lava Flows Simulations with OpenCL: A Preliminary Application for Civil Defence Purposes* in *Proceedings of the 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2015* (2015), 328.

119. Bedorf, J., Gaburov, E. & Portegies Zwart, S. A sparse octree gravitational N-body code that runs entirely on the GPU processor. *Journal of Computational Physics* **231**, 2825 (2012).

120. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G. & Dongarra, J. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing* **38**, 391 (2012).

121. Brown, W., Wang, P., Plimpton, S. & Tharrington, A. Implementing molecular dynamics on hybrid high performance computers - Short range forces. *Computer Physics Communications* **182**, 898 (2011).

122. Malcolm, J., Yalamanchili, P., McClanahan, C., Venugopalakrishnan, V., Patel, K. & Melonakos, J. *ArrayFire: A GPU acceleration platform* in. **8403** (2012).

123. Su, B.-Y. & Keutzer, K. *clSpMV: A cross-platform OpenCL SpMV framework on GPUs* in *ICS '12 Proceedings of the 26th ACM international conference on Supercomputing* (2012), 353.

124. *clBlas*

125. Reguly, I., Mudalige, G., Giles, M., Curran, D. & McIntosh-Smith, S. *The OPS domain specific abstraction for multi-block structured grid computations* in *Proceedings of WOLFHPC 2014: 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), 58.

126. Jammy, S., Mudalige, G., Reguly, I., Sandham, N. & Giles, M. Block-structured compressible Navier-Stokes solution using the OPS high-level abstraction. *International Journal of Computational Fluid Dynamics* **30**, 450 (2016).

127. Giles, M., Mudalige, G., Spencer, B., Bertolli, C. & Reguly, I. Designing OP2 for GPU architectures. *Journal of Parallel and Distributed Computing* **73**, 1451 (2013).

128. Reguly, I., Mudalige, G., Bertolli, C., Giles, M., Betts, A., Kelly, P. & Radford, D. Acceleration of a Full-Scale Industrial CFD Application with OP2. *IEEE Transactions on Parallel and Distributed Systems* **27**, 1265 (2016).

129. Cercos-Pita, J. AQUAgpusph, a new free 3D SPH solver accelerated with OpenCL. *Computer Physics Communications* **192**, 295 (2015).

130. *Advanced Simulation Library*

131. Dattilo, G. & Spezzano, G. Simulation of a cellular landslide model with CAMELOT on high performance computers. *Parallel Computing* **29**, 1403 (2003).

132. Spingola, G., D'Ambrosio, D., Spataro, W., Rongo, R. & Zito, G. *Modeling Complex Natural Phenomena with the libAuToti Cellular Automata Library: An example of Application to Lava Flows Simulation.* in *PDPTA - International Conference on Parallel and Distributed Processing Techniques and Applications* (2008), 277.

133. Avolio, M., Di Gregorio, S., Mantovani, F., Pasuto, A., Rongo, R., Silvano, S. & Spataro, W. Simulation of the 1992 Tessina landslide by a cellular automata model and future hazard scenarios. *International Journal of Applied Earth Observation and Geoinformation* **2**, 41 (2000).

134. NVIDIA Corporation. *CUDA C PROGRAMMING GUIDE* (NVIDIA Corporation, 2015).

135. Gardner, M. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American* **223**, 120 (1970).

136. Kirk, D. B. & Hwu, W.-m. W. *Programming Massively Parallel Processors: A Hands-on Approach* 1st (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010).

137. Anderson, D. P. *BOINC: A System for Public-Resource Computing and Storage* in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing* (IEEE Computer Society, Washington, DC, USA, 2004), 4.

138. QGIS Development Team. *QGIS Geographic Information System* Open Source Geospatial Foundation (2009).

139. Stockman, G. & Shapiro, L. G. *Computer Vision* 1st (Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001).

140. Hadwiger, M., Kniss, J. M., Rezk-salama, C., Weiskopf, D. & Engel, K. *Real-time Volume Graphics* (A. K. Peters, Ltd., Natick, MA, USA, 2006).

141. Volkov, V. *Understanding Latency Hiding on GPUs* PhD thesis (EECS Department, University of California, Berkeley, 2016).

142. Spataro, D., Arcuri, P., d. Rango, A., Spataro, W., D'Ambrosio, D. & Mari, A. *A Tracking Algorithm for Particle-Like Moving Objects* in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)* (2017), 491.

143. Bellman, R. E. R. E. L. L. Combinatorial optimization: networks and matroids. *Bull. Amer. Math. Soc. 84* (1978).

144. *OPENCAL download page*

145. S. Cho, R. H. & Yi, S. Improvement of Kittler and Illingworth's minimum error thresholding. *Pattern Recognit., vol. 22, no. 5, pp. 609– 617* (1989).

146. Deng, G. & Cahill, L. W. *An adaptive Gaussian filter for noise reduction and edge detection* in *Nuclear Science Symposium and Medical Imaging Conference, 1615-1619 vol.3, 10.1109/NSSMIC.1993.373563* (1993).

147. *CGAL, Computational Geometry Algorithms Library* ().

148. Berg, H. C. *Random walks in biology.* (1993).

149. Wong-Ng, J., Melbinger, A., Celani, A. & Vergassola, M. The Role of Adaptation in Bacterial Speed Races. *PLOS Computational Biology* **12**, 1 (2016).

150. Masson, J.-B., Voisinne, G., Wong-Ng, J., Celani, A. & Vergassola, M. Noninvasive inference of the molecular chemotactic response using bacterial trajectories. *Proceedings of the National Academy of Sciences* **109**, 1802 (2012).

151. Tarantino, N., Tinevez, J.-Y., Crowell, E. F., Boisson, B., Henriques, R., Mhlanga, M., Agou, F., Israël, A. & Laplantine, E. TNF and IL-1 exhibit distinct ubiquitin requirements for inducing NEMO–IKK supramolecular structures. *The Journal of Cell Biology* **204**, 231 (2014).

152. Howse, J. R., Jones, R. A. L., Ryan, A. J., Gough, T., Vafabakhsh, R. & Golestanian, R. Self-Motile Colloidal Particles: From Directed Propulsion to Random Walk. *Physical Review Letters* **99**, 048102 (2007).

153. Cisneros, L. H., Kessler, J. O., Ganguly, S. & Goldstein, R. E. Dynamics of swimming bacteria: Transition to directional order at high concentration. *Physical Review E* **83** (2011).

154. rahmat:2016. *Multi-Agent System with Multiple Group Modelling for Bird Flocking on GPU* in *2016 24th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)* (2017).

155. Schmickl, T., Hamann, H., Wörn, H. & Crailsheim, K. Two Different Approaches to a Macroscopic Model of a Bio-inspired Robotic Swarm. *Robot. Auton. Syst.* **57**, 913 (2009).

156. Ferber, J. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence* (Addison Wesley, 1999).

157. Woolridge, M. *Introduction to Multiagent Systems* (John Wiley and Sons, 2001).

158. Reynolds., C. W. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics* **21**, 25 (1987).

159. Reynolds., C. W. *Steering behaviors for autonomous characters.* in *Game Developers Conference* (1999), 763.

160. Reynolds., C. W. *Interaction with groups of autonomous characters.* in *Game Developers Conference* (2000), 449.

161. Hemelrijk Charlotte K. Hildenbrandt, H. Some Causes of the Variable Shape of Flocks of Birds. *PLoS ONE* **6** (2011).

162. Laird., J. *It knows what you're going to do: Adding anticipation to a quakebot.* in *AAAI 2000 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment* (2000), 41.

163. Nvidia. *CUDA C Best practices* (Nvidia, 2012).

164. Corporation, N. *Whitepaper NVIDIA GeForce GTX 980, Featuring Maxwell, The Most Advanced GPU Ever Made* (2014).

165. Richmond, P., Coakley, D. S. & Romano, D. D. *A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA*

166. Nguyen, H. *Gpu Gems 3* First (Addison-Wesley Professional, 2007).

167. Dutta., K. How birds fly together: The dynamics of flocking. *Resonance* **15**, 1097 (2010).

168. Gammon., K. Last Accesses 29/9/2017. http://phys.org/news/2011-10-secrets-flocking-revealed.html.

169. Gangshan Jingab, Y. Z. & Wang, L. Flocking of multi-agent systems with multiple groups. *International Journal of Control* **87**, 2573 (2014).

170. John Cheng, M. G. & Ty McKercher. *Professional CUDA C Programming* (John Wiley and Sons Inc., 2014).

171. (ed et al., M. B.) *Efficient Stream Compaction on Wide SIMD Many-Core Architectures* (2009).

172. (ed Hughes) *InK-Compact-: In kernel Stream Compaction and Its Application to Multi-kernel Data Visualization on GPGPU* (2015).

173. Hoberock, J. & Bell, N. *Thrust: A Parallel Template Library* 2010.

PUBLICATIONS

Articles in peer-reviewed journals:

4. The Open Computing Abstraction Layer for Extended Cellular Automata and the Finite Differences Method. *Journal of Parallel and Distributed Computing* (2017).
5. The new SCIARA-fv3 numerical model and acceleration by GPGPU strategies. *International Journal of High Performance and Applications* (2017).

Conference contributions:

1. *Applications of the OpenCAL Scientific Library in the context of CFD: Applications to Debris Flows* in *International Conference on Networking, Sensing and Control (ICNSC)* (2017).
2. *A Tracking Algorithm for Particle-like Moving Objects* in *Proceedings of The 2017 International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (2017).
3. *Multi-Agent System with Multiple Group Modelling for Bird Flocking on GPU* in *Proceedings of The 2016 International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (2016).
6. *Efficient Lava Flows Simulations with OpenCL: A preliminary application for Civil Defence Purposes* in *Proceedings of The 10 th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing* (2015).
7. *CUDA Dynamic Active Thread List Strategy to Accelerate Debris Flow Simulations* in *Proceedings of The 2015 International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (2015).