Università della Calabria

Dipartimento di Matematica e Informatica

Dottorato di Ricerca in Matematica e Informatica
XXXI CICLO

Settore Scientifico Disciplinare: INF/01

Tesi di Dottorato

Domain Specific Languages for Parallel numerical modeling on structured grids

Il Candidato

Alessio De Rango

I Supervisori

Prof. Donato D'Ambrosio

Prof. William Spataro

Prof. Gihan Mudalige

Il Coordinatore

Ch.mo Prof. Nicola Leone

A.A. 2017-2018

Contents

Sommario

Il calcolo ad alte prestazioni (HPC) attualmente in un periodo di enormi cambiamenti. Per molti anni, alte prestazioni sono state raggiunte attraverso l'aumento della frequenze di clock, ma tale tendenza stata interrotta bruscamente dal corrispondente aumento consumo energetico. La direzione ora verso il miglioramento prestazioni attraverso l'aumento del parallelismo, anche riducendo la frequenza di clock per migliorare l'efficienza energetica. Tuttavia, non vi ancora un chiaro consenso su quale sia la migliore architettura per HPC. Da un lato ci sono acceleratori multi-core come GPU e la nuova Intel Xeon Phi.. D'altra parte, abbiamo mainstream CPU Intel / AMD con cache molto grandi e un numero pi modesto di unit logiche (core) ciascuna con le proprie componenti vettoriali (ad esempio AVX unit), o i sistemi IBM BlueGene basati su una vasta rete di CPU relativamente piccole ma ad alta efficienza energetica. Di conseguenza, alla luce di questi sviluppi, un programmatore HPC, per ottenere alte prestazioni su diversi hardware, costretto ad ottimizzare la propria applicazione per una determinata piattaforma, allo stesso tempo, richiedendo una quantit crescente di conoscenza specifica dell'hardware, e in alcuni casi una riscrittura completa del codice. Una possibile soluzione a questo problema l'utilizzo di una strategia di sviluppo ad alta astrazione (HLA) basata su DSL (Domain Specific Languages). Esempi sono OpenCAL e OPS, che sono stati specificatamente proposti come librerie parallele in C / C ++ per computazione su griglie strutturate. Lo scopo di queste librerie di semplificare e rendere pi veloce il processo di sviluppo di modelli complessi, consentendo di implementare l'applicazione e riferendosi a un modello di programmazione di sviluppo seriale, delegando il processo di parallelizzazione alla libreria per diverse soluzioni di calcolo parallelo. In questa tesi, ho contribuito alla progettazione e allo sviluppo di entrambi i progetti OpenCAL e OPS. In particolare, il mio contributo a OpenCAL ha riguardato lo sviluppo dei componenti single-GPU e multi-GPU / multi-nodo, ovvero OpenCAL-CL e OpenCAL-CLM. Mentre, il mio contributo a OPS ha riguardato l'introduzione del supporto OpenMP 4.0 / 4.5, in alternativa alle gi esistenti versioni OpenCL, CUDA e OpenACC, per sfruttare i moderni sistemi di elaborazione a molti core.

Abstract

High performance computing (HPC) is undergoing a period of enormous change. Due to the difficulties in increasing clock frequency indefinitely (i.e., the breakdown of Dennard's scaling and power wall), the current direction is towards improving performance through increasing parallelism. However, there is no clear consensus yet on the best architecture for HPC, and different solutions are currently employed. As a consequence, applications targeting a given architecture can not be easily adapted to run on alternative solutions, since this would require a great effort due to the need to deal with platform-specific details. Since it is not known *a priori* which HPC architecture will prevail, the Scientific Community is looking for a solution that could tackle the above mentioned issue. A possible solution consists in the adoption of a high-level abstraction development strategy based on Domain Specific Languages (DSLs). Among them, OpenCAL (Open Computing Abstraction Layer) and OPS (Oxford Parallel Structured) have been proposed as domain specific C/C++ data parallel libraries for structured grids. The aim of these libraries is to provide an abstract computing model able to hide any parallelization detail by targeting, at the same time, different current (and possibly future) parallel architectures. In this Thesis, I have contributed to the design and development of both the OpenCAL and OPS projects. In particular, my contribution to OpenCAL has regarded the development of the single-GPU and multi-GPU/multi-node components, namely OpenCAL-CL and OpenCAL-CLM, while my contribution to OPS has regarded the introduction of the OpenMP 4.0/4.5 support, as an alternative to OpenCL, CUDA and OpenACC, for exploiting modern many-core computing systems. Both the improved DSLs have been tested on different benchmarks, among which a fractal set generator, a graphics filter routine, and three different fluid-flows applications, with more than satisfying results. In particular, OpenCAL was able to efficiently scale over larger computational domains with respect to its original implementation, thanks to the new multi-GPU/multi-node capabilities, while OPS was able to reach near optimal performance using the high-level OpenMP 4.0/4.5 specifications on many-core accelerators with respect to the alternative low-level CUDA-based version.

Acknowledgments

I would like to express my special thanks of gratitude to my supervisors Donato D'Ambrosio and William Spataro for their constant help and the great support given to me during the whole period of my Ph.D.. I would also like to thank Davide Spataro for his help and for the excellent work done together. I would also like to thank Prof. Gihan Mudalige, who supervised me during a six months visiting period (January - July 2017) at University of Warwick. I would also like to thank Carlo Bertolli and Istvan Reguly for the valuable advices given to me during my research. I would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work. I would also like to thank all the people who I have met during these years and from which I have learned. Finally, I wish to extend my deepest gratitude to my family for their support and love.

Chapter 1

Introduction

Current High-Performance Computing (HPC) machines are large ensembles of powerful computing nodes with heterogeneous hardware interconnected via network. In fact, beside traditional CPUs, GPGPU (General-Purpose Computation on Graphics Processing Unit) has become mainstream. This trend was a consequence of the end of frequency scaling, caused by the unsustainable energy consumption of CPUs. Consequently, current HPC processor architectures have moved towards to massively parallel designs, increasing computational power. The use of this recent advanced hardware is constantly being applied in many fields of Science and Engineering. In particular, in the field of Scientific Computing, parallel machines are used to obtain approximate numerical solutions of differential equations which model a physical system. In fact, the classical approach based on Calculus often fails to solve these kinds of equations analytically, making a numerical computer-based approach necessary. An approximate numerical solution of a system of partial differential equations can be obtained by applying methods such as, among others, Cellular Automata (CA) and Finite Volume Method (FVM), which yield approximate values at a discrete number of points over the considered domain.

Achieve optimal performance on scientific applications by exploiting modern heterogeneous machines, each one with its programming model, may require a major effort, since most of them have been developed over the years and usually consist of many lines of code. Often, the HPC Community is forced to use hand-tuning code, or performing architecture-specific transformations to the specific application, significantly increasing the programming effort. The HPC Community is trying to overcome these obstacles maintaining, at the same time, some important attributes to the applications, the most important of which are: (i) code *portability*, which allows to exploit the same code across current different architectures; (ii) *programmability*, which provides simple features to reach performance; (iii) *long-term viability*, which allows to adapt the applications to future architectures and pro-

gramming models. Many solutions have been proposed to reach the above goals. Among them, the RAJA [51, 75], Kokkos [67, 75], and FastFlow [23] C++ template libraries have been developed at Lawrence Livermore National Laboratory (LLNL, USA), Sandia National Laboratories (USA), and University of Pisa (Italy), respectively, with the purpose of running models on heterogeneous parallel platforms. At the same time, other software systems have been proposed that utilizes a high-level abstraction (HLA) development strategy based on Domain Specific Languages (DSLs). Examples are OpenCAL [33] and OPS [104], that were specifically proposed as domain specific C/C++ data parallel libraries for structured grids. The aim of these libraries is to simplify and make faster the development process of complex models by allowing to implement the application and referring to a serial development programming model, delegating the parallelization process to the library for different parallel computing solutions.

In particular, OpenCAL has been developed at the University of Calabria (Italy) and provides a C Application Programming Interface (API) exposing a DSL based on the Extended Cellular Automata formalism (XCA) [47]. Also known as Complex or Multi-Component Cellular Automata, XCA were introduced for the modeling of macroscopic fluid-flows (see e.g., [35, 14, 39, 96, 38]), forest fire spreading [13], ecohydrologic dynamical systems [86, 102, 87, 37, 58, 109], besides others. Note that, since XCA are a general structured-grid based computational paradigm, other numerical methods are supported by OpenCAL, such as Finite Differences and Cellular Automata (of which XCA can be considered an extension). Low-level parallel programming details can be ignored and different efficient serial, multi-core, single-GPU, multi-GPU and cluster of GPUs applications obtained. Even if quite recent, OpenCAL has already been applied in different contexts like the generation of fractal sets [101], convolutional graphics filtering [101], and the simulation of fluid [33] and particle systems [100].

The OPS system has been developed at the University of Oxford (UK) and is a high-level abstraction framework targeting computation on multi-block structured meshes. OPS supports both C/C++ and Fortran languages, the most widespread languages for scientific applications. The aim of OPS is to separate the abstract definition of the computation from its parallel implementations and execution in order to concentrate the effort only on the development of the sequential code, totally delegating the generation of parallel code to the source-to-source translator. The abstract definition of the computation is defined by a special loop API function which is automatically recognized by the source-to-source translator that generates the appropriate translation to the specific platform. The high level abstraction approach proposed by OPS has been also utilized in many research projects, such as OpenSBLI (Open Shock-Boundary Layer Interactions), for solving the compressible Navier-Stokes equations [70] and HiLeMMS (High-Level Mesoscale Modelling System), for developing a high-level mesoscale

modelling system [4]. Moreover, an integrated collection of small software programs (mini-apps) that model the performance of full-scale applications, part of the Mantevo [6] software, have been implemented in OPS as benchmark applications. In particular, among all mini-apps, the Cloverleaf and the Tealeaf benchmarks were imported in OPS.

In this Thesis, I have contributed to the design and development of both the OpenCAL and OPS projects. In particular, my contribution to OpenCAL has regarded the development of the single-GPU and multi-GPU/multi-node components, namely OpenCAL-CL and OpenCAL-CLM. As the names suggest, they were based on the OpenCL and MPI APIs to accelerate the computation on many-core devices and clusters of interconnected workstations, respectively. Both developed components allow scientists to conceptually design the computational model at a high level of abstraction, by referring to the Extended Cellular Automata general formalism. Parallelism, as well as memory transfer operations between the involved heterogeneous computing systems, is transparent to the user. Moreover, optimized algorithms, such as the *active cells* one (also known as *quantization*), were implemented and ready to use within the library.

My contribution to OPS has regarded the introduction of the OpenMP 4.0/4.5 support, as an alternative to OpenCL, CUDA and OpenACC, for exploiting modern many-core computing systems. In order to develop the OpenMP4 OPS extension, a new OPS source-to-source translator has been designed and implemented. Two C/C++ compilers were considered, which provide support the newer OpenMP specifications, namely Clang and IBM XL. The above cited Cloverleaf and Tealeaf mini-apps were considered for performance assessment. Furthermore, additional analysis regarding the registers per thread and the achieved occupancy metrics were carried out to better characterize the new OPS source-to-source translator with respect to the ones previously developed and targeting the same devices. The research on OPS started during a six-month internship period I spent at the University of Warwick (UK), still in progress as a scientific collaboration with the University of Calabria (Italy). Other research groups are also involved, among which the IBM Thomas J. Watson Research Center (USA).

In the following, Chapter 2 provides a brief description of two numerical methods widely adopted for the simulation of complex physical phenomena, namely Extended Cellular Automata and Finite Volume Method. Such methods are those considered by OpenCAL and OPS and were also taken into account in the examples of applications considered. Chapter 3 provides a briefly overview of modern parallel computing frameworks. In particular, OpenCL, CUDA, OpenMP and MPI are described, which are the parallel APIs considered in the development of this work. Chapter 4 describes OpenCAL and OPS, by pointing out my specific contributions. Different examples of applications are also described. Chapter 5 presents and discusses the computational results achieved by new OpenCAL and OPS releases.

Eventually, Chapter 6 concludes the Thesis with a review of the carried out work and outlines possible future developments.

Chapter 2

Numerical Methods

This Chapter provides a brief description of two numerical methods widely adopted for the simulation of complex physical phenomena, namely Extended Cellular Automata and Finite Volume Method. Extended Cellular Automata represent an alternative formulation of the original Cellular Automata computing paradigm, which is particularly useful in the simulation of complex natural processes. Classic Cellular Automata, together with some theory and applications, are also preliminary presented in this Chapter. Finite Volume Method, which represents a discretization method for the approximation of partial differential equations (PDEs) systems into integral form, is discussed in the last Section.

2.1 Cellular Automata

Cellular Automata (CA) are computational models whose evolution is governed by laws which are purely local. In its essential definition, a CA can be described as a d-dimensional space composed of regular cells. Each cell can be in a finite number of states and embeds a *finite automaton (fa)*, one of the most simple and well known computational model in Computer Science. A *finite automaton (fa)* can be seen as a system to which a *state* is associated which can change on the basis of an input. At time $t = 0$, cells are in an arbitrary state and the CA evolves by changing the states of the cells in discrete steps of time and by applying simultaneously to each of them the same law, or *transition function*. Input for each cell is given by the states of neighboring cells and the *neighborhood* conditions are determined by a geometrical pattern, which is invariant in time and space.

Despite their simple definition, CA may give rise to extremely complex behavior[130] at a macroscopic level. In fact, even if local laws that regulate the dynamics of the system are known, the global behavior of the system is very hard to be predicted [26]. In other words, the dynamics of the system emerges in a nontrivial manner by the mutual interaction of its basic

components.

CA are considered universal computation models because of their equivalence with Turing Machines [46, 128], as demonstrated by Codd [29] and Thatcher [117].

CA are adapt to model and simulate systems characterized by the interaction of numerous elementary constituents and they have been largely employed in several fields of study as pattern recognition [81, 41], image processing [106, 107, 98], cryptography [120, 22] and so on. An example of CA application is the study of the behavior of the fluid (considered at the microscopic level as particles systems) through Lattice Gas[115]. Other important studies are related to the consideration of CA as parallel computing systems [118, 32, 116].

In this chapter after a brief history of CAs, a quick overview, with the introduction of an informal and formal definition of the model, is presented. Some theoretical aspects of CA are also reported and a brief description of some examples of CA applications in different areas conclude the chapter.

2.1.1 A Brief History of Cellular Automata

CA was developed thanks to a study in the 1940s by John Von Neumann and Stanisław Ulam. Von Neumann was a wide-ranging mathematician. He studied and contributed in the set theory, in economics and in game theory. Ulam was a mathematician. He invented the Monte Carlo simulation technique and at the same time he made contributions to number theory and set theory. In 1948 von Neumann, after the read a paper called "The General and Logical Theory of Automata", was concerned to many challenging questions, among them if or not it would ever be possible for a automaton to reproduce itself. For this reason, he tried to devise an abstract model for the features and the complexity of self-reproducing systems. The model was created by using the idea of machines made up of multiple copies of a small number of standardized elements, all put into a reservoir. His proof was base on the idea that an automaton could have a blueprint for building itself. However, due to the complexity of the developed model, Von Neumann couldn't make his proof convincing. Thanks to Ulam, who was working with von Neumann in those years, the abstract model was redesigned in way to not considering the reservoir full of automata parts but think in terms of an idealized space of cells that could hold finite state-numbers representing different sorts of part. Von Neumann prematurely died in 1957 and his work was published later in 1966, edited and completed by A.Burks [124].

Immediately after von Neumann and Ulam work, two immediate research paths emerged. The first, mostly carried out in the 1960s, was an increasingly whimsical discussion of building actual self-reproducing automata, while the second studies regarded an attempt to capture more of the essence of self-reproduction by mathematical studies of detailed properties

of cellular automata.

By the end of the 1950s, CA were viewed as parallel computers, and in particular in the 1960s a sequence of increasingly detailed and technical theorems, often analogous to ones about Turing machines, were proved about their formal computational capabilities. At the end of the 1960s there then began to attempt to link cellular automata to mathematical discussions of dynamical systems.

However, CA became famous in the 70s thanks to one of the simplest CA application, the well-known *Game of Life* defined by the English mathematician John Horton Conway and described by Martin Gardner in his work [55].

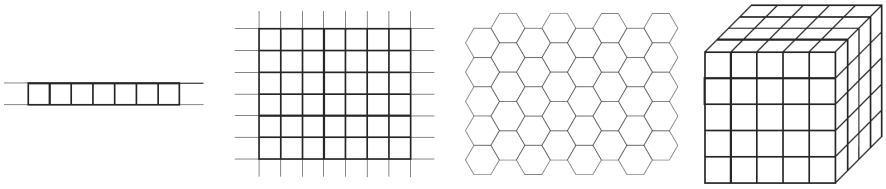


FIGURE 2.1: Example of cellular spaces: (a) one-dimensional, (b) two-dimensional with square cells, (c) two-dimensional with hexagonal cells, (d) three-dimensional with cubic cells.

2.1.2 Informal Definition of Cellular Automata

It is possible to identify an informal definition of cellular automaton by simply listing its main properties:

- it is formed by a d-dimensional space (*the cellular space*), partitioned into cells of uniform size (triangles, squared, hexagons, cubes) or by a d-dimensional regular lattice (see Figure 2.1);
- the number of cell states is finite;
- the evolution occurs through discrete steps;
- each cell evolves by simultaneously changing its state by applying the same transition function to the cellular space;
- the transition function depends on the state of the central and neighboring cells;
- the relationship of closeness that defines the neighborhood of a cell is local, uniform and invariant over time.

2.1.2.1 Dimension and Geometry of Cellular Automata

The definition of Cellular Automata requires the discretization of the space in cells. In the simplest situation (one-dimensional Cellular Automata), the CA space is one-dimensional and cells are aligned next to each other. Regarding multi-dimensional Cellular Automata, the CA space can be discretized in different ways: a two-dimensional CA can be represented, for example, with triangle, square or hexagonal tessellation while for three-dimensional cellular automata cubic cells are usually chosen. Figure 2.1 shows examples of different cellular spaces.

Even if a square tessellation, for two-dimensional cellular automata, can be easily represented by a matrix structure (both for graphics and computation representation), by considering some applications it can be lead to anisotropic problems. In these cases, it is preferable to adopt an hexagonal tesselation.

2.1.2.2 Number of States of a Cell

The number of states of a cell is finite and it is based on the study or application context. In first theoretical studies in which CA were considered as abstract models [29, 117], the number of states of a cell was usually quite small.

When the CA is adopted to describe particle systems, it is not necessary to adopt a large number of states to model the interactions [114, 127]. In contrast, when studying systems with a continuum of possible states, CA may require a large number of states [47].

2.1.2.3 Relationship of Closeness

The cell's neighbourhood relationship depends on the geometry of cells and it has to have the following properties:

1. it must be local because only a limited number of cells near the central one are involved;
2. it must be homogeneous because it is the same for each cell of the cellular space;
3. it must be invariant over time;

For one-dimensional CA, usually, the neighborhood is considered in terms of radius, r, which defines a neighborhood consisting of $n = 2r + 1$ cells [129]. For example, a radius $r = 1$ consists of $n = 2r + 1 = 3$ cells: the central cell, the aligned left one and the aligned right one. Figure 2.2 shows two different examples of neighborhood for an uni-dimensional CA.

In the case of two-dimensional CA with square tesselation, the neighborhoods most commonly used are von Neumann and Moore ones, with the first one comprises the four cells orthogonally surrounding a central cell (north, east, south, west) while the second also contains the north-west, northeast, south-west, and south-east cells.

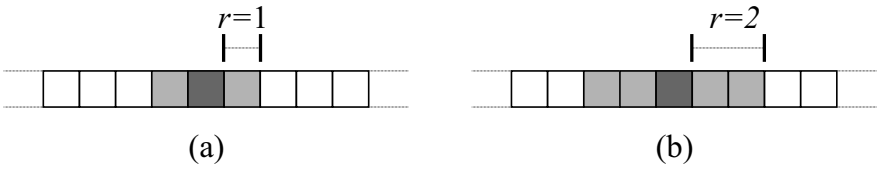


FIGURE 2.2: Example of neighborood with radius (a) $r = 1$ and (b) $r = 2$ for uni-dimensional cellular automata.

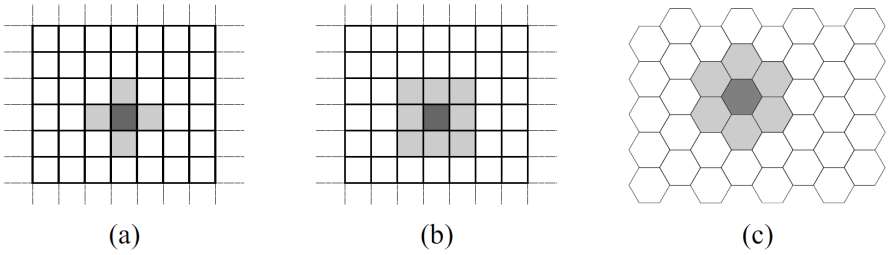


FIGURE 2.3: von Neumann (a) and Moore (b) neighboroods for a two-dimensional cellular automata with square cells and with exagonal ones (c).

For exagonal two-dimensional cellular automata a typical neighborood is composed by north, north-east, south-east, south, south-west and north-west cells.

Figure 2.3 shows (a) the von Neumann neighborhood and (b) the Moore one for square tesselation and (c) the one adopted for hexagonal tesselation.

It is worth to note that different neighbourhoods can be applied such as, for example, the Margolus neighborhood widely used in simulations of gas diffusion [119].

2.1.2.4 State-Transition Function

At each CA step, the transition function is simultaneously applied to all cells of the cellular space, by determining the new state of each cell in function of the state of the neighborhood cells. Parallelism and decentralization are characteristics of the CA computation.

When the number of states is small, usually the transition rules are defined through a look-up table which specifies the new state of the central cell for each possible configuration of the neighborhood [130]. Differently, when the number of CA states is too large, the transition function is usually defined by an algorithm [47].

2.1.3 Formal Definition of Classical Cellular Automata

The homogeneous cellular automata is defined as a quadruple:

$$A =< \mathbb{Z}^d, Q, X, \sigma > \tag{2.1}$$

where:

- $\mathbb{Z}^d = \{i = (i_1, i_2, ..., i_d) | i_k \in \mathbb{Z}\ \forall k = 1, 2, ..., d\}$ is the d-dimensional cellular space;
- Q is the finite set of states of the cellular automata;
- $X = \{\xi_0, \xi_1 ...\xi_{m-1}\}$ is the finite set of m d-dimensional vectors

  $$\xi_j = \{\xi_{j1}, \xi_{j2}, ...\xi_{jd}\}$$

  that define the set

  $$V(X, i) = \{i + \xi_0, i + \xi_1, ..., i + \xi_{m-1}\}$$

  of coordinates of cells close to the generic cell i with coordintates $(i_1, i_2, ...i_d)$.

  X is the geometrical pattern that specifies the neighborhood relationship;
- $\sigma : Q^m \longrightarrow Q$ is the transition function for the CA.

2.1.4 Theory of Cellular Automata

In this section, some theoretical aspects of cellular automata are reported. In particular, results on reversibility, conservation laws, universality and topological dynamics of CA are discussed. Since most of them are related to one-dimensional CA, it is important to first introduce some definitions.

2.1.4.1 One-dimensional Cellular Automata

The simplest forms of CA are elementary CA [129]. They are one-dimensional CA characterized by N cells, $k = 2$ states (0 and 1), neighborhood radius $r = 1$ and periodic boundary conditions (one-dimensional cellular space is toroidal, seen as a ring where first and last cells are adjacent). Figure 2.4

shows an example of a one-dimensional CA with periodic boundary conditions. Representing a CA as a ring allows to define an unlimited space in which the CA can evolve.

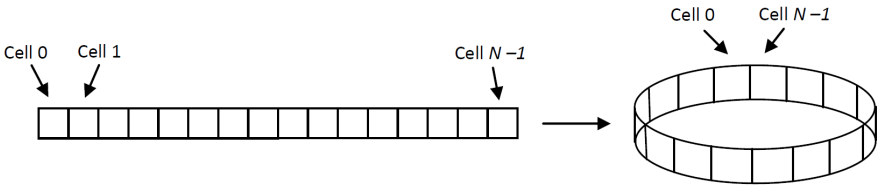


FIGURE 2.4: Example of one-dimensional cellular automata with periodic boundary conditions. First and last cells are adjacent.

The transition function σ is defined through a look-up table. For example, by considering the generic neighborhood's configuration η (the number of neighborhood's configuration is k^{2r+1} and for one-dimensional CA is $2^3 = 8$), the following transition function determines the central cell new state, $s = \sigma(\eta)$:

η	000	001	010	011	100	101	110	111
s	0	0	1	1	0	1	1	0

By adopting this convention, any transition rule for the elementary CA elementary can be defined by listing the central cell new states as follow:

$\sigma_{00110110} =$

η	000	001	010	011	100	101	110	111
s	0	0	1	1	0	1	1	0

Each possible transition rule can be identified through the decimal number corresponding to the binary number that defines the same rule($\sigma_{000000000} = \sigma_0, \sigma_{000000001} = \sigma_1, ..., \sigma_{000000001} = \sigma_{255}$).

If $r > 1$ the number of possible configuration grows rapidly. For example, by considering $(k, r) = (2, 2)$ the number ot total transition rules is $2^{32} = 4294967296$ because $k^{2r+1} = k^n = 2^5 = 32$, making impossible a comprehensive analysis.

2.1.4.2 Universality and Complexity in Cellular Automata

Wolfram [129, 130] proposed a classification of the one-dimensional CA based on their qualitative behavior, identifying four different complexity classes:

- **class 1** Class 1 CA, nearly all initial patterns, evolve quickly into a uniform final state. Any randomness in the initial pattern disappears;

- **class 2** Class 2 CA evolve into the final state with stable or oscillating structures. Some of the randomness in the initial pattern remains;
- **class 3** Class 3 CA are characterized by an extremely pseudo-random or chaotic behavior. Structures that appear are quickly destroyed;
- **class 4** Class 4 CA are characterized by both uniforms an chaotic behavior. In this class structures can interact with each other in extremely complex ways;

Although other classifications have been proposed [25, 60, 84], the Wolfram one is certainly the most known. Examples of CA belonging to the Wolfram four complexity classes are illustrated in Figure 2.5.

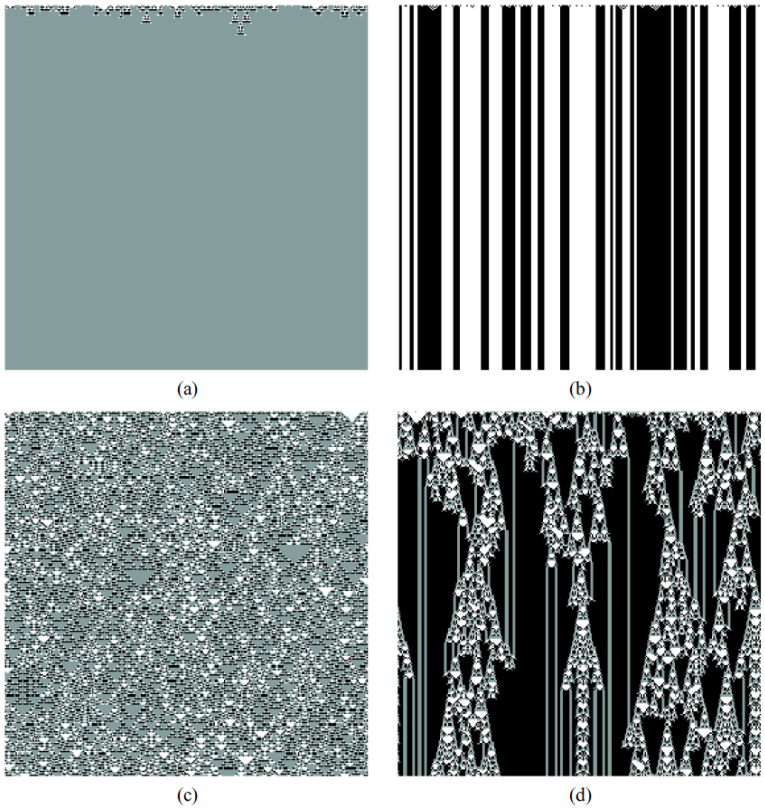


FIGURE 2.5: First 250 CA calculation steps with k = 3 and r = 1 for (a)$\sigma_{c=1014}$, (b)$\sigma_{c=1008}$, (c)$\sigma_{c=1020}$, (d)$\sigma_{c=2043}$. The illustrated CA belong respectively to the complexity classes 1, 2, 3 and 4. The initial configuration consists of 250 cells and it is randomly generated so that each cell can assume state 0 (white), 1 (gray) or 2 (black).

Class 4 proved to be particularly interesting for the presence of structures (e.g., gliders) able to propagate in space and time. For this reason, Wolfram

hypothesized that CA belonging to the class 4 can be capable of universal computation. The CA proposed by Wolfram can be seen as calculators; the initial configurations encode data and program, and final configuration (after several calculation steps) encode the computation result. This means that it is possible to represent with a CA and an appropriate transition function every possible computational program. The hypothesis of universality for simple CA comes from the observation that the glider can act as elaborator of information encoded into the initial configuration. By means of the glider, the state of a cell in a particular position of the cellular space can influence over time the states of cells in arbitrarily distant locations. Furthermore, the glider can interact among themselves in an extremely complex way and, theoretically, can play as demonstrated for the Game of Life by John Horton Conway [55] the logic gates of a universal computer.

2.1.4.3 Chaos Theory

Class 4 automata are considered as at the *edge of chaos* and give a good metaphor for the idea that the interesting complexity is in equilibrium between stability and chaos. The hypothesis of Wolfram that the simple one-dimensional CA are capable of universal computation was, subsequently, studied by Chris Langton. He has shown that an appropriate parameterization of the space of rules allows identifying both the relationship between the complexity classes and the regions of that space.

Langton in his paper [79] introduced the λ parameter as the fraction of the entries in the transition rule table that are mapped do the quiescent state q_s. Langton's major finding was that a simple measure such as correlates with the system behavior: as goes from 0 to 1, the average behavior of the systems goes from freezing to periodic patterns to chaos and functions with an average $\lambda \approx 1/2$ (please refer to [79] for a more general discussion) are being on the edge.

In particular, the parameter λ was defined as:

$$\lambda = \frac{k^n - n_q}{k^n} = 1 - \frac{n_q}{k^n} \tag{2.2}$$

where k is the number of states of the cell, $n = 2r + 1$ is the number of neighborhood cells and n_q is the number of transitions that terminate in the quiescent state. If $n_q = k^n$, all the transitions of the look-up table go to the quiescent state and $\lambda = 0$; if $n_q = 0$ there are no transitions that terminate in the quiescent state and $\lambda = 1$; finally, $\lambda = 1 - 1\lambda$ when in the look-up table when all the states are represented by the same measure.

Langton has analyzed the behaviour of several totalistic CA with k = 4 and r = 1 with a variation range for λ of [0,0.75]. The results showed that for small values of λ the CA behaviour is uniform, typical of complexity classes 1 and 2, while for large values the observed behaviours are chaotic, typical

of class 3. Between these two zones (order and chaos), however, Langton has observed a very small third zone near to the value $\lambda = 0.45$. In this zone, defined as edge of chaos, the CA dynamics is able to generate both static and dynamic structures that can be propagated in space and time, typical of the class 4 of Wolfram.

Figure 2.6 shows an example of a CA at the edge of chaos. Only in the edge of chaos zone, the encoded information in the CA initial configuration can propagate over long distances, which is a necessary condition for the concept of computation.

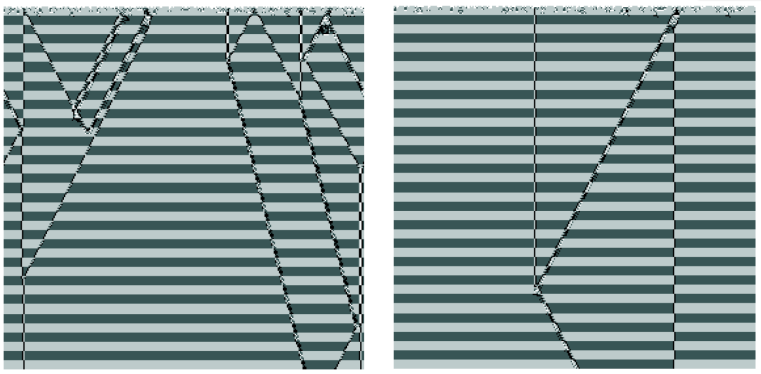


FIGURE 2.6: Examples of CA at the edge of chaos. Figures (a) and (b) show the evolution of the same CA with k = 4 states and r = 1 with two different initial conditions. The shades of gray represent the four possible states of the cell, from with for the state 0 to black for the state 3.

2.1.4.4 Other Theoretical Works on Cellular Automata

The studies discussed in this chapter are only a part of the whole set of theoretical researches on CA and many different contributions have stemmed from researchers from all parts of the world. For example, the problem of reversibility of CA has been studied by Moore [90], Myhill [95], Di Gregorio and Trautteur [49], Kari [72], and Toffoli and Margolus [119].

Jiménez-Morales has adopted an evolutionary approach based on GA for the study of non-trivial collective behavior in CA [71].

Other interesting studies regard, the self-reproduction problem in the CA. Among them, Azpeitia and Ibáñez [18] and Bilotta et al. [21] worked in this direction.

Finally, Roli and Zambolli [105] studied the emergence of macro spatial structures in dissipative CA seen as open systems where the environment can influence the dynamics.

2.1.5 CA Applications

CA are particularly suited to modeling and simulation of some classes of complex systems characterized by the interaction of a large number of elementary components. The assumption that if a system behaviour is complex, the model that describes it must necessarily be of the same complexity is replaced by the idea that its behavior can be described, at least in some cases, in very simple terms [130].

In some areas, the CA application gave results comparable to those obtained from traditional approaches. A particularly significant example is the CA application to modeling turbulent flows behaviour through lattice gas and lattice Boltzmann models. Another important field of CA application is the Artificial Life, a discipline that deals with the examination of systems related to life, its processes, and its evolution. Moreover, in recent years, CA have been applied with success in the modeling of natural complex phenomena.

A brief description of some examples of CA applications in Artificial Life, Lattice Gas and lattice Boltzmann models is described below. CA application to modeling natural complex phenomena is discussed in the next chapter.

2.1.5.1 Artificial Life with CA

Artificial life can be defined as the discipline that deals with the life and the behavior of artificial systems that *live* in an artificial environment. It seeks to study life not out in nature or in the laboratory, but in the computer. Langton suggested that CAs could be an extremely effective model to study artificial life[78]. In fact, John von Neuman since 40's studied the reproduction in living organisms by adopting an artificial approach based on the CA paradigm.

Subsequently, Codd at the end of the 60's and then Langton in the mid-80's, have proposed a simplified model compared to the von Neumann original one for the self-reproduction with self-replicating structures.

von Neumann was convinced that the self-reproduction should incorporate the property of universal computability; for this reason, his model was very complex. Cood even if shared the von Neuman hypothesis proposed an alternative model with 8 states [29].

However, Langton proposed a model with self-replicating structures (Langton's loops) not computational equivalent to the Turing Machine, thus he has shown that the universal computability property is a sufficient condition for self-reproduction but not a necessary condition [77].

Chou and Reggia [28] have demonstrated, for the first time, that it is possible to implement CA with 'general' transition functions in order to emerge self-replicating structures with initial configurations completely ran-

dom. These structures may have different characteristics and shapes and they interact with other structures that simultaneously emerge in the cellular space.

Other interesting works that reflect the original von Neumann study regarding the self-reproduction problem were conducted by Azpeitia and Ibáñez [18] and Bilotta [21].

As it can be seen from the research branch related to self-reproduction, the Artificial life has produced hypothesis and original results of extreme interest, both from the theoretical point of view and from that of the possible applications and, in this context, CA have played a very important role.

2.1.5.2 Lattice Gas Cellular Automata and Lattice Boltzman Models

Fluid dynamics is a branch of physics that deals with the behavior of gases and liquids. The classical fluid dynamics is based on the Navier-Stokes equations that formalize the laws of conservation of mass and momentum.

The non-linearity of such equations is the main cause of the difficulty to apply them for not idealized cases [114] and for this reason, an alternative approach to the study of fluid dynamics based on CA, namely Lattice Gas, emerged.

2.1.5.3 Lattice Gas Cellular Automata

The basic idea of lattice gas is to model a fluid through a system of particles that can move, with constant velocity, only along the directions of a discrete lattice. Local laws are defined in order to ensure the invariance of the number of particles (conservation of mass) and the conservation of momentum.

More formally, a Lattice Gas Automaton (LGA) is a lattice of cells $\vec{r}$. Each cell $\vec{r}$ contains $z+1$ quantities $n_i(\vec{r},t), i = 0, ...z$ where z is the coordination number. Neighbors of $\vec{r}$ are obtained as $\vec{r} + \vec{v}_i$, where $\vec{v}_i$ are given vectors and by convention $\vec{v}_0 = 0$. The LGA dynamics consist of two steps. The first one is the interaction step where the quantities n_i locally collide and new values $n_i^{'}$ are computed according to a predefined collision operator $\Omega_i(n)$. The second step regards the propagation by sending the quantity $n_i^{'}(\vec{r})$ to the neighboring site along lattice direction $\vec{v}_i$.

The first lattice-gas cellular automata (LGCA) was proposed in 1973 [62] by Hardy, Pomeau and de Pazzis. It is named HPP and represents the simplest LGCA. In particular, HPP is a two-dimensional lattice-gas cellular automata model over a square lattice. The vectors $c_i (i = 1, 2, 3, 4)$ connecting nearest neighbors are called *lattice vectors*. At each node there are four cells (see Figure 2.7) each associated to a link with the nearest neighbor. Cells may be empty or occupied by at most one particle (exclusion principle). The evolution in time is deterministic and proceeds with local collisions and

propagation along links to the nearest neighbors. The collision conserves mass and momentum while changing the occupation of the cells and when two particles enter a node from opposite directions and the other two cells are empty a head-on collision takes place which rotates both particles by 90^o in the same sense.

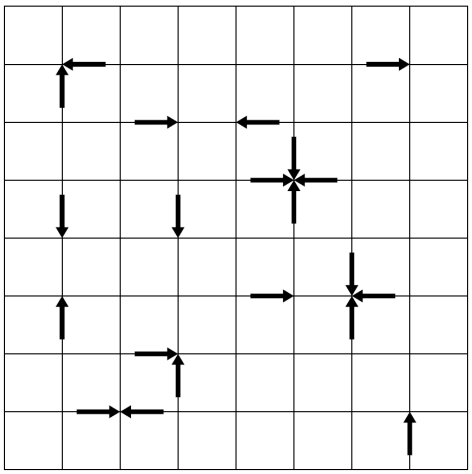


FIGURE 2.7: Example of LGA on a square lattice. In this example $n_i \in 0, 1$. The arrows directions indicate the sites with $n_i = 1$. The lattice direction are $\vec{v}_1 = (1, 0), \vec{v}_2 = (0, 1), \vec{v}_3 = (-1, 0), \vec{v}_4 = (0, -1)$.

The first LGA reproducing a correct hydrodynamic behavior has been introduced in 1986. Frish, Hasslacher and Pomeau showed that an LGCA over a lattice with a larger symmetry group than for the square lattice yields the Navier-Stokes equation in the macroscopic limit. This model, named FHP, presents a hexagonal symmetry.

In particular, the properties of FHP lattice can be described as following:

- The lattice shows hexagonal symmetry (the lattice is composed of triangles).
- Nodes are linked to six nearest neighbors located all at the same distance with respect to the central node.
- c_i is the lattice vectors and it links the neighbor nodes.

$$c_i = (\cos\frac{\pi}{3}i, \sin\frac{\pi}{3}i), i = 1, ..., 6 \tag{2.3}$$

- A cell is associated with each link at all nodes.
- Cells can be empy or occupied by at most one particle(exclusion principle)

- All particles have the same mass.
- The evolution proceeds by collissions C and streaming S (propagation) as:

$$\epsilon = S \circ C \tag{2.4}$$

  where ϵ is the evolution operator.
- The collisions are local.

As for HPP there are 2-particle head-on collisions but in contrast to HPP the FHP model encompasses non-deterministic rules. A pseudo-random choice is used where the rotational sense changes by chance for the whole domain from time step to time step or the sense of rotation changes from node to node but is constant in time.

The basic FHP model defines only two- and three-particle collisions, but it turns out that this is sufficient to yield the desired behavior. If two particles traveling in opposite directions meet at a node, then the particle pair is randomly rotated either clockwise or counterclockwise by sixty degrees. If three particles meet at a node in a symmetric configuration, then they collide in such a way that this configuration is *inverted*.

Please refer to [44] for a more detailed description of the FHP model.

2.1.5.4 Lattice Boltzmann Models

Lattice Boltzmann Models (LBM) have been introduced because the LGA are plagued by several diseases for Navier-Stroker equations computation. Lattice Boltmann equations have been applied by Frisch et al [44] in 1987 to calculate the viscosity of LGCA.

In 1988 LBM have been used by McNamara and Zanetti as a numerical method for hydrodynamic simulations [85]. LGCA have been replaced with LBM because the authors decided to completely eliminate the statistical noise that plagues the usual lattice-gas simulations so the boolean fields were replaced by continuous distributions over the FHP lattices.

Higuera and Jiménez introduced an alternative simulation procedure for lattice hydrodynamics [65], based on the lattice Boltzmann equation instead of on the microdynamical evolution. In particular, in this work, the collision operator is expressed in terms of its linearized part with the introduction of few parameters to decrease viscosity.

Koelman [76], Qian et al. [99] and others replaced the collision operator with the Bhatnagar-Gross-Krook (BGK) approximation. This new model, compared to lattice gases, is noise-free and collisions are not anymore defined explicitly.

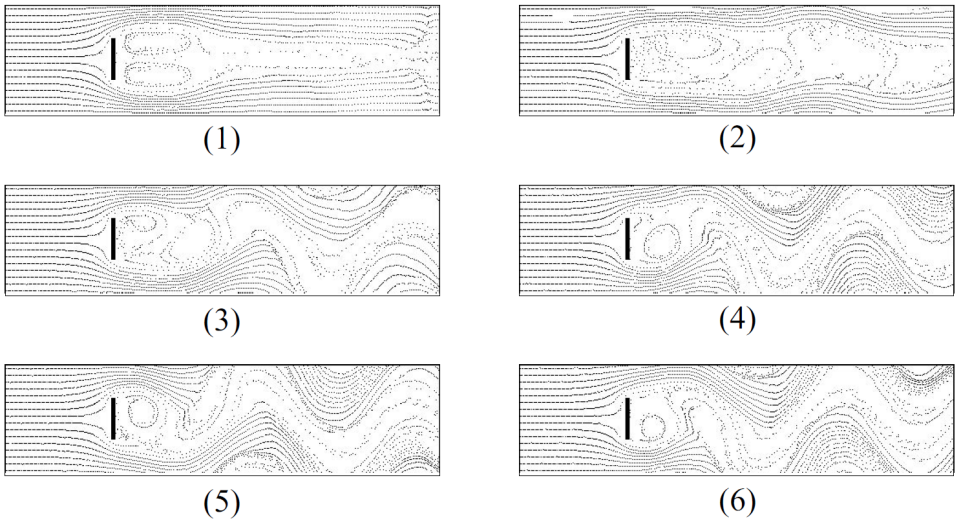


FIGURE 2.8: Simulation of a flow around a thin plate with a Boltzmann Lattice model. Figures 1 to 6 illustrate the evolution of the system.

Lattice Boltzmann models are most popular today. A significant advantage of the LBM, compared to the Lattice Gas, is that only the density of particles is taken into account so the number of components of the system is considerably reduced.

The dynamics of a LBM [99] can be described as follows:

$$f_i(\vec{r}+\tau\vec{v_i},t+\tau)-f_i(\vec{r},t)=\Omega_i(f_i(\vec{r},t))=\frac{1}{\xi}({f_i}^{(eq)}(\vec{r},t)-(f_i(\vec{r},t)) \quad (2.5)$$

where $f_i(\vec{r},t)$ represents the density of particles that at time t are in the cell $\vec{r}$ with velocity $\vec{v_i}$; ${f_i}^{(eq)}(\vec{r},t)$ is the local equilibrium distribution and ξ is the number of calculation steps to reach the equilibrium at local neighborhood level.

The ${f_i}^{(eq)}(\vec{r},t)$ function specifies the conditions of local equilibrium of the system in function of density, $\rho=\sum f_i$ and of momentum, $\rho\vec{u}=\sum f_i\vec{v_i}$, of fluid in the cell. The parameter ξ expresses how the system is dependent of the fluid viscosity $v=K(\xi-1/2)$ where K is a constant that depends on the lattice geometry.

Unlike the Lattice Gas Automaton, the viscosity becomes an explicit parameter of the model. Figure 2.8 illustrates the dynamics of the BKG model [99] in the simulation of a flow around a thin plate.

2.2 Extended Cellular Automata

As discussed in the previous Section, CAs have been applied with success in modeling and simulating complex systems, whose dynamics can describe in

term of local interactions. Among different fields, fluid-dynamics is one of the most important field of application for CA and, in this research branch, many different CA-based methods were used to simulate fluid flows. Lattice Gas Automata models [43] were introduced for describing the motion and collision of particles on a grid and it was shown that such models can simulate fluid dynamical properties. The continuum limit of these models leads to the Navier-Stokes equations. Lattice Gas models can be regarded as microscopic models, as they describe the motion of fluid *particles* which interact by scattering.

An advantage of Lattice Gas models is that the simplicity of particles, and of their interactions, allow for the simulation of a large number of them, making it therefore possible to observe the emergence of flow patterns. Furthermore, since they are cellular automata systems, it makes easier to run simulations with parallel computing. A different approach to LGA is represented by Lattice Boltzmann models [85] in which the state variables can take continuous values, as they are supposed to represent the density of fluid particles, endowed with certain properties, located in each cell (here space and time are discrete, as in lattice gas models). Both Lattice Gas and Lattice Boltzmann Models have been applied for the description of fluid turbulence [27, 115].

Because many complex natural phenomena evolve on very large areas, they are therefore difficult to be modeled at a microscopic level of description. Among these, lava flows can be considered, at the same time, one of the most dangerous and difficult phenomena to be modeled as, for instance, the temperature drops along the path by locally modifying the magma dynamical behavior (because of the effect of the strong temperature-dependence of viscosity). Furthermore, lava flows generally evolve on complex topographies that can change during eruptions, due to lava solidification, and are often characterized by branching and rejoining of the flow. Extended Cellular Automata (XCA) represent a valid alternative to classical CA regarding macroscopic phenomena.

As regards the modeling of natural complex phenomena, Crisci and coworkers proposed a method based on an extended notion of homogeneous CA, firstly applied to the simulation of basaltic lava flows [31], which makes the modeling of spatially extended systems more straightforward and overcomes some unstated limits of the classical CA, such as having few states and look-up table transition functions [47]. Mainly for this reason, the method is known as Extended Cellular Automata, even though it was also known as Macroscopic Cellular Automata [112] or Multicomponent Cellular Automata [17].

XCA were in fact adopted for the simulation of many macroscopic phenomena, such as lava flows [16], debris flows [40], density currents [108], water flux in unsaturated soils [53], soil erosion/degradation by rainfall [34] as well as pyroclastic flows [16], bioremediation processes [48] and forest fires

[121].

2.2.1 Informal Definition of Extended Cellular Automata

Informally, XCA compared to classical CA, are different because of the following reasons:

- the state of the cell must account for all the characteristics, which are assumed to be relevant to the evolution of the system: these refer to the space portion of the cell. Each characteristic corresponds to a *substate*. The state of the cell is divided into substates and permitted values for a substate must form a finite set. The set of the possible states of a cell represents the global state of the cell and is given by the Cartesian product of the sets of the substates.
- the state of the cell can be decomposed in substates and the transition function may be split into local interactions: the *elementary processes*. Each of them represents a particular aspect that rules the dynamic of the considered phenomenon. Different elementary processes may involve different neighborhoods. The CA neighborhood is given by the union of all the neighborhoods associated to each process. If the neighborhood of an elementary process is limited to a single cell, such a process is considered as an *internal transformation*.
- a set of global parameters to reproduce the several different dynamic behaviors of the considered phenomenon is defined.
- a subset of the cells is also influenced by external influences, represented by a function. External influences are used to model those features that are difficult to describe as local interactions.

2.2.2 Formal Definition of Extended Cellular Automata

Formally, a XCA is a 7-tuple:

$$A =< \mathbb{Z}^d, Q, X, P, \tau, E, \gamma > \tag{2.6}$$

where:

- $\mathbb{Z}^d$ is the d-dimensional cellular space;
- $Q = Q_1 \times Q_2 \times .... \times Q_n$ is the set of states of the cell obtained as the Cartesian product of *substates* $Q_1 \times Q_2 \times .... \times Q_n$ each one representing a particular feature of the phenomenon to be modelled;
- X is the geometrical pattern that specifies the neighbourhood relationship;

- $P = p_1, p_2, ..., p_p$ is the set of CA *parameters*.They allow to tune the model for reproducing different dynamical behaviours of the phenomenon of interest;
- $\tau : Q^m \longrightarrow Q$ is the transition function for the CA and it is splitted in *elementary processes* $\tau_1, \tau_2, ..., \tau_s$, each one describing a particular aspect that rules the dynamic of the considered phenomenon.
- $E = E_1 \cup E_2 \cup ... \cup E_l \subseteq \mathbb{Z}^d$ is the set of cells of $\mathbb{Z}^d$ that are subject to *external influences*. External influences were introduced in order to model features which are not easy to be described in terms of local interactions;
- $\gamma = \{\gamma_1, \gamma_2, ..., \gamma_t\}$ is the finite set of functions that define the external input for the CA.

2.2.3 Modelling Surface Flows through CA

Many geological processes like lava or debris flows can be described in terms of local interactions and thus modeled by XCA. By opportunely discretizing the surface on which the phenomenon evolves, the dynamics of the system can be in fact described in terms of flows of some quantity from one cell to the neighboring ones. Moreover, as the cell dimension is a constant value throughout the cellular space, it is possible to consider characteristics of the cell (i.e. substates), typically expressed in terms of volume (e.g. lava volume), in terms of thickness. This simple assumption permits to adopt a straightforward but efficacious strategy that computes outflows from the central cell to the neighbouring ones in order to minimize the non-equilibrium conditions.

In the XCA approach, by considering the third dimension (the height) as a property of the cell, outflows can be computed by procedures based on one of *distribution* algorithms as the Minimisation Algorithm of the Differences [47], briefly described in the next Section.

2.2.3.1 The Minimization Algorithm of the Differences

The Minimisation Algorithm of the Differences (MAD), proposed by Di Gregorio and Serra [47], reduces the non-equilibrium conditions by minimizing quantities between the central cell and its neighbors. In other words, outflows from the central cell to the other n neighbouring cells must be determined in order to minimize the differences of a quantity q in the neighboring cells.

The MAD is based on the following assumptions:

- two parts of the considered quantity must be identified in the central cell: these are the unmovable part, $u(0)$, and the mobile part, m;

- only m can be distributed to the adjacent cells. Let $f(x, y)$ denote the flow from cell x to cell y; m can be written as:

$$m = \sum_{i=0}^{\#X} f(0, i) \tag{2.7}$$

  where $f(0, 0)$ is the part which is not distributed, and $\#X$ is the number of cells belonging to the X neighbourhood. It is worth to note that this definition preserves the principle of conservation of mass for the distributable quantity m.

- the quantities in the adjacent cells, $u(i)(i = 1, 2, ..., \#X)$ are considered unmovable;

- let $c(i) = u(i) + f(0, i)(i = 0, 1, ..., \#X)$ be the new quantity content in the i-th neighbouring cell after the distribution and let c_{min} be the minimum value of $c(i)(i = 0, 1, ..., \#X)$. The outflows are computed in order to minimise the following expression:

$$m = \sum_{i=0}^{\#X} (c(i) - c_{min}) \tag{2.8}$$

Basically, the MAD operates as follows:

1. the following average is computed:

$$a = \frac{m + \sum i \in A u(i)}{\#A}$$

   where A is the set of not eliminated cells (i.e. those that can receive a flow); note that at the first step $\#A = \#X$;

2. cells for which $u(i) \geq a(i = 0, 1, ..., \#X)$ are eliminated from the flow distribution and from the subsequent average computation;

3. the first two points are repeated until no cells are eliminated; finally, the flow from the central cell towards the *i-th* neighbour is computed as the difference between $u(i)$ and the last average value a:

$$f(0, i) = \begin{cases} a - u(i) & \text{i} \in A \\ 0 & \text{i} \notin A \end{cases}$$

An example of MAD application is reported in Figure 2.9.

Note that the simultaneous application of the minimization principle to each cell gives rise to the global equilibrium of the system. The correctness of the algorithm is stated in [47].

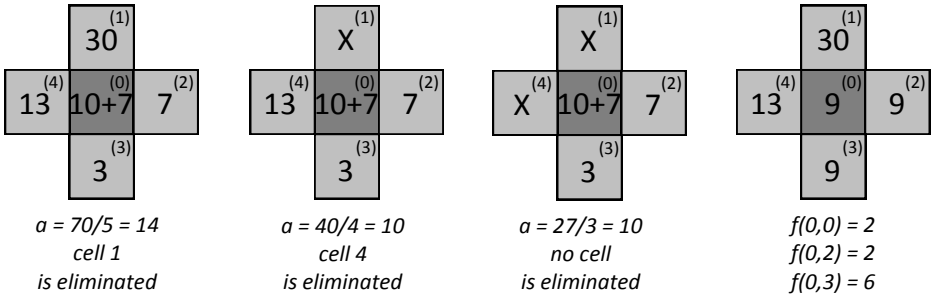


FIGURE 2.9: Example of application of the Minimization Algorithm of the Differences by considering a bidimensional CA with square cells and Von Neumann neighborhood.

2.3 Finite Volume Method

Nowadays, Computational Fluid Dynamics (CFD) studies fluid flows by using methods that are based a combination of physics, numerical mathematics, and computer science. CFD analyzes systems involving fluid flows, heat transfer and associated phenomena such as chemical reactions. CFD is used in a wide range of applications such as aerodynamics of aircraft and vehicles [80], hydrodynamics of ships [74], chemical process engineering [66], marine engineering [50], meteorology [126], and biomedical engineering [68] among others. The main reason for CFD spreading can be identified by the description of fluid flows that is at the same time economical and sufficiently complete. Moreover, CFD allows reducing costs and time for new designs, at the same time permitting the study of systems that are difficult or impossible to be simulated under safe conditions.

All phenomena that can be analyzed by CFD are governed by a system of equations, usually partial differential equations (PDEs). The CFD methodological aim is to provide a numerical approximation to these equations. Depending on the adopted scheme, the system of partial differential equations is usually approximated by a recurrence formula (implicit method) or a system of linear equations (explicit method), and then solved at discrete sites of the computational domain, by providing an approximation of the analytical solution (which in most cases is unknown).

The computational domain is preliminarly discretized as a *mesh (or grid) of cells*, generally defined by a number of small non-overlapping elements. Depending on the geometry of the cells, the mesh can be structured or unstructured, as shown in Figure 2.10. The choice of the domain discretization

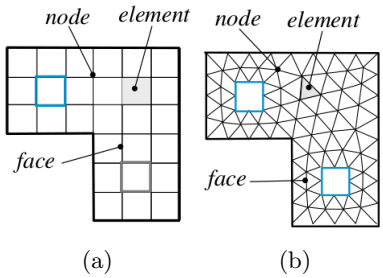


FIGURE 2.10: Examples of two-dimensional (a) structured and (b) unstructured meshes (from [91]).

depends on the problem to be simulated. Both structured and unstructured meshes are widely adopted in modeling complex fluid flows. The structured mesh generally allows for faster simulations, while unstructured meshes provide for more precise results. In certain cases, both kinds of meshes are used together in hybrid discretization methods. One of the most important stages of CFD is the specification of boundary conditions for the system, which define constraints at domain boundary. For instance, the Dirichlet condition is a value-specified constraint, while the Neumann one is related to the flux.

A CFD application can be identified by three main stages: A *pre-processor*, which provides the geometry of the computational domain, defines the flow parameters and the boundary conditions; the *numerical solver*, which allows solving the governing equations under the conditions provided; a *post-processor*, which produces the simulation output in a readable or graphical format.

Three main numerical solvers have been proposed and are currently widely adopted: *finite difference method (FDM)*, *finite elements method (FEM)* and *finite volume method (FVM)*. The FEM has become the most popular and used method in modern Computational Solid Mechanics [30, 111, 69], whereas FVM has become one of the most popular methods in the area of Computational Fluid Dynamics [73, 88, 59]. Both methods have surpassed the historical FDM and other discretization methods and nowadays, researchers widely use such method in Science and Engineering.

The following sections present a mathematical description of fluids, the general conservation equation, convective and diffusive phenomena. The Finite Volume Method is therefore described and a simple application shown.

2.3.1 Mathematical Formulation of Fluid-Flow Phenomena

Fluid dynamics regards the investigation of the interactive motion of a large number of individual particles, defined by molecules or atoms. In general,

the density of the fluid can be defined high enough, so that it can be approximated as a continuum. Even if an infinitesimally small element of the fluid still contains a sufficient number of particles, we can specify the mean velocity and mean kinetic energy of the system. With this assumption, we are able to define velocity, pressure, temperature, density and other properties at each point of the fluid.

The derivation of the equations of fluid dynamics is based on the dynamical behaviour of the fluid, determined by the following conservation laws: conservation of mass, momentum, and energy. A generic conservation equation for the fluid flow can be expressed in the following form:

$$\frac{\partial \phi}{\partial t} + \nabla \cdot F = Q^{\phi} \tag{2.9}$$

where ϕ is the conserved quantity, F is the flux of the conserved state and Q^{ϕ} is the source term.

Example 1 *Euler Equations for a Compressible Fluid.*

The two-dimensional Euler Equations for a Compressible Fluid can be formulated through the general conservation equation 2.9 by specifying the state ϕ, the flux F and the source Q^{ϕ} as follows:

$$\phi = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{pmatrix}, F = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uH \end{pmatrix} i + \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vH \end{pmatrix} j, Q^{\phi} = 0 \tag{2.10}$$

where ρ is the density, u and v the velocity components along x and y directions, respectively, E the total energy per unit mass, p the static pressure, and H the total enthalpy per unit mass, defined as $H = E + p/\phi$.

The conserved states of the systems are the density, x- and y- momenta, ρu and ρv, and the total energy, ρE. In particular, the first row of the system corresponds to the conservation of the mass, the second and third to the conservation of x and y momentum respectively, and the fourth to the conservation of the energy. In order to solve the system, since we have four conservation equations and a total of five dependent variables, we need to define an equation state. Assuming an ideal gas and using the ideal gas law, we can define the static pressure as follows:

$$p = (\gamma - 1)[\rho E - 1/2\rho(u^2 + v^2)] \tag{2.11}$$

where γ is the ratio of specific heats. Specifying the equation state of p the system can be solved.

2.3.1.1 Convection Equation

In many applications, the dominant physical transport phenomenon is modeled as convection. To derive the convection equation using the conservation law 2.9, the flux of the conserved state and source term can be defined as follows:

$$F = \rho \boldsymbol{v} \phi, \quad Q^{\phi} = 0 \tag{2.12}$$

where $\boldsymbol{v}$ is the velocity vector and ρ is the density. For simplicity, the source term is set to zero. Nevertheless, a non-zero source term could be included. Finally, the convection equation becomes:

$$\frac{\partial \phi}{\partial t} + \boldsymbol{v} \cdot \nabla \phi = 0 \tag{2.13}$$

2.3.1.2 Diffusion Equation

In other applications, the dominant physical transport phenomenon is modeled as diffusion. To derive the diffusion equation using the conservation law 2.9, the flux of the conserved state can be defined as follows:

$$F = -\Gamma^{\phi} \nabla \phi \tag{2.14}$$

where Γ^{ϕ} is the diffusion coefficient. Finally, the diffusion equation becomes:

$$\frac{\partial \phi}{\partial t} - \Gamma^{\phi} \nabla \phi = Q^{\phi} \tag{2.15}$$

2.3.1.3 Convection-Diffusion Problem

In most of fluid dynamic phenomena both convection and diffusion are important. For this reason, the convection-diffusion equation can be derived from a general conservation law. The differential form is:

$$\frac{\partial (\rho \phi)}{\partial t} + \nabla \cdot (\rho \boldsymbol{v} \phi) = \nabla \cdot (\Gamma^{\phi} \nabla \phi) + Q^{\phi} \tag{2.16}$$

The following section describes the application of the FVM to the convection-diffusion problem.

2.3.2 The FVM Method for Convection-Diffusion Problem

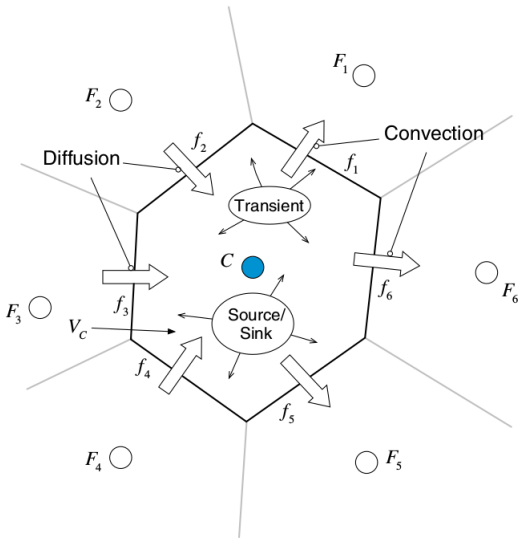


FIGURE 2.11: Control volume elements. (from [91])

As in other approximated approaches, in the FVM the numerical solution of the governing equations is obtained by calculating the values of the property ϕ at specific non-overlapping domain points, called *control volume elements*. Figure 2.11 illustrates an example of control volume with the different terms of the general transport equation.

Generally, there are two common approaches for the finite volume discretization:

- **Cell-centred approach**: the control volumes are defined by a suitable grid and computational nodes are assigned at the control volume center.
- **Vertex-centred approach**: the location of the node is first defined and then the control volume are constructed around them so that the control volume faces lie midway between the nodes.

The latter approach permits an accurate resolution of the face fluxes for all kind of meshes. Since the vertex is not necessarily at the element centroid, it can yield a lower order accuracy of element integrations. Instead, the cell-centred approach is the most popular with FVM, with all the variables and the quantities stored at the centroid of the grid element.

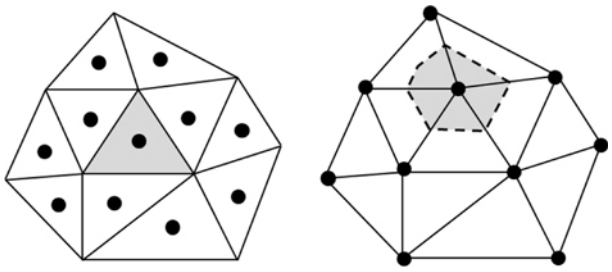


FIGURE 2.12: cell-centered and vertex-centered FVM.

Once the geometry of the domain is specified, the equation is therefore discretized over the control volume elements. For steady-state convection-diffusion problems, the general transport equation 2.16 loses the unsteady term, as shown in Equation 2.17. The next stage is to integrate the obtained equation to the control volume (Equation 2.18).

$$\nabla \cdot (\rho \boldsymbol{v} \phi) = \nabla \cdot (\Gamma^{\phi} \nabla \phi) + Q^{\phi} \tag{2.17}$$

$$\int_{V_c} \nabla \cdot (\rho \boldsymbol{v} \phi) dV = \int_{V_c} \nabla \cdot (\Gamma^{\phi} \nabla \phi) dV + \int_{V_c} Q^{\phi} dV \tag{2.18}$$

By applying the Gauss theorem it is possible to transform the volume integrals of convection and diffusion terms into surface integrals. The final expression is thus given in Equation 2.19.

$$\oint_{\partial V_c} (\rho \boldsymbol{v} \phi) dS = \oint_{\partial V_c} (\Gamma^{\phi} \nabla \phi) dS + \int_{V_c} Q^{\phi} dV \tag{2.19}$$

Replacing the surface integral by a summation of the flux from the face of the elements, the diffusion, and convection terms become:

$$\oint_S (\rho \boldsymbol{v} \phi) dS = \sum_{f}^{faces(V_c)} (\int_f (\rho \boldsymbol{v} \phi) \cdot dS) \tag{2.20}$$

$$\oint_S (\Gamma^{\phi} \nabla \phi) dS = \sum_{f}^{faces(V_c)} (\int_f (\Gamma^{\phi} \nabla \phi) \cdot dS) \tag{2.21}$$

Applying to Gauss quadrature the faces integral becomes:

$$\oint_S (\rho \boldsymbol{v} \phi) dS = \sum_{f}^{faces(V_c)} \sum_{ip}^{ip(f)} (\int_{ip} (\rho \boldsymbol{v} \phi) \cdot S_f) \tag{2.22}$$

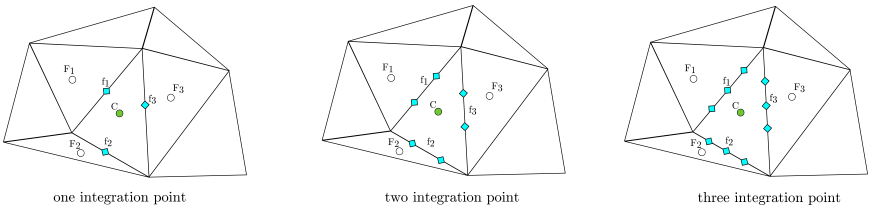


FIGURE 2.13: One, two or three integration points over the element faces.

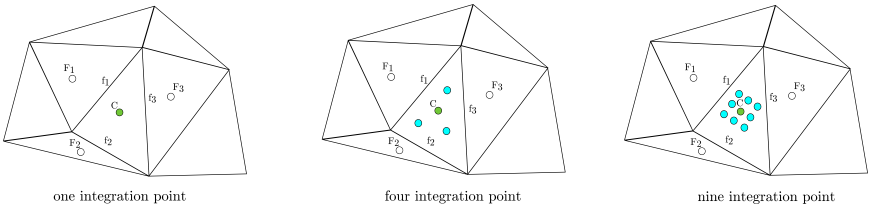


FIGURE 2.14: One, four or nine integration points over the source surface.

$$\oint_S (\Gamma^\phi \nabla\phi) dS = \sum_{f}^{faces(V_c)} \sum_{ip}^{ip(f)} (\int_{ip} (\omega_{ip} \Gamma^\phi \nabla\phi) \cdot S_f) \tag{2.23}$$

where ϕ is the fluid density, $\boldsymbol{v}$ the velocity vector, $\boldsymbol{S}_f$ the surface vector and ω_{ip} is weighing function. The figure 2.13 illustrates alternative cases in which the different number of integration points over the element faces are considered. The order accuracy depends on the integration points used and on the values of the weighing function ω. In case of single integration point and a weighing function $\omega = 1$, a second order accuracy is achieved and it is applicable in two or three dimensions.

The Gaussian quadrature is also applied to the source term. The volume integration of the source term becomes:

$$\int_V Q^\phi dV = \sum_{ip}^{ip(V)} (Q^\phi_{ip} \omega_{ip} V) \tag{2.24}$$

Figure 2.14 illustrates alternative configurations with one, four and nine integration points for the source term. As for the other terms, the order accuracy depends on the number of integration points and on the values of the weighing function ω. As before, in the case of a single integration point and $\omega = 1$, a second order accuracy is achieved and it is applicable in two or three dimensions.

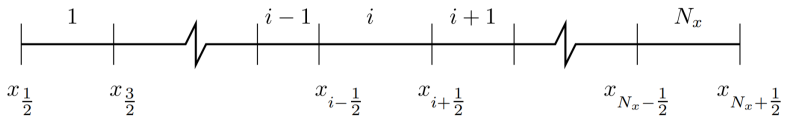


FIGURE 2.15: Mesh for one-dimensional finite volume method. (from [3])

2.3.3 An example of Finite Volume Method in 1-D

As shown above, the main idea of the FVM is to subdivide the domain into many control volumes and approximate the integral conservation law on each of them. Figure 2.15 shows an example of one-dimensional domain partition.
The general conservation law (Equation 2.9) can be discretized over the control volume. The integral form of the general conservation law is shown by Equation 2.25.

$$\frac{d}{dt}\int_{x_L}^{x_R}\phi dx + F(\phi)\Big|_{x_R} - F(\phi)\Big|_{x_L} = \int_{x_L}^{x_R} Q^{\phi} dx \tag{2.25}$$

Applying 2.25 to the general control volume i and assuming $S = 0$, equation 2.25 becomes:

$$\frac{d}{dt}\int_{x-\frac{1}{2}}^{x+\frac{1}{2}}\phi dx + F(\phi)\Big|_{x+\frac{1}{2}} - F(\phi)\Big|_{x-\frac{1}{2}} = 0 \tag{2.26}$$

By defining the mean value of ϕ in control volume i as:

$$\phi^i \equiv \frac{1}{\Delta x_i}\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}}\phi dx, \quad where \quad \Delta x_i \equiv x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}} \tag{2.27}$$

the Equation 2.30 becomes:

$$\Delta x_i \frac{d\phi_i}{dt} + F(\phi)\Big|_{x+\frac{1}{2}} - F(\phi)\Big|_{x-\frac{1}{2}} = 0 \tag{2.28}$$

By assuming that the solution on every control volume is constant, we obtain:

$$\phi(x,t) = \phi_i(t) \quad for \quad x_{i-\frac{1}{2}} < x < x_{i+\frac{1}{2}} \tag{2.29}$$

The Figure 2.16 shows the piecewise constant finte volume method approximation.
Considering the case of convection equation 2.12, the solution convects with the velocity $\boldsymbol{v}(t)$. So, for the initial condition after t we can consider:

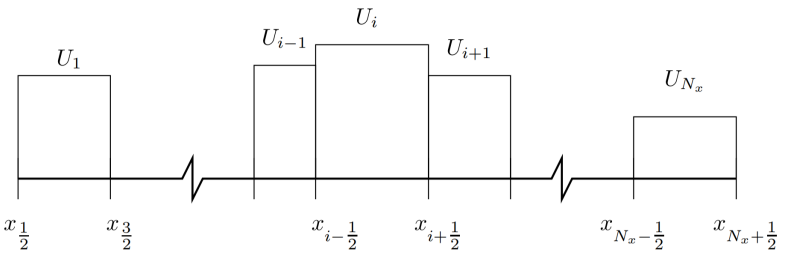


FIGURE 2.16: The piecewise constant finte volume method approximation (from [3]).

$$\phi(x_{i+\frac{1}{2}}, t^+) = \begin{cases} \phi_i & if \quad \boldsymbol{v}(t) > 0 \\ \phi_{i+1} & if \quad \boldsymbol{v}(t) < 0 \end{cases} \tag{2.30}$$

Where $t^+ = t + \xi$, and ξ identifies an infinitesimal positive number. The corrisponding flux can then calculate from this value of ϕ.

$$F(x_{i+\frac{1}{2}}, t^+) = \begin{cases} \boldsymbol{v}\phi_i(t) & if \quad \boldsymbol{v}(t) > 0 \\ \boldsymbol{v}\phi_{x+1}(t) & if \quad \boldsymbol{v}(t) < 0 \end{cases} \tag{2.31}$$

Note that an alternative manner to write this flux is shown in the following equation:

$$F(x_{i+\frac{1}{2}}, t^+) = \frac{1}{2}\boldsymbol{v}(t)(\phi_{i+1}(t) + \phi_i(t)) - \frac{1}{2}\left|\boldsymbol{v}(t)\right|(\phi_{i+1}(t) - \phi_i(t)) \tag{2.32}$$

This kind of flux, which uses the upstream value of ϕ, is called as an upwind flux. Eventually, a forward Euler ODE integration method can be adopted in order to discetize Equation 2.32 in time. The final fully-discrete form of the finite volume method is shown in Equation 2.33.

$$F(x^n_{i+\frac{1}{2}}) = \frac{1}{2}\boldsymbol{v}^n(\phi^n_{i+1} + \phi^n_i) - \frac{1}{2}\left|\boldsymbol{v}^n\right|(\phi^n_{i+1} - \phi^n_i) \tag{2.33}$$

Being Equation 2.33 expressed in term of recurrence formula, it can be straightforwardly solved at each control volume, and therefore the phenomenon can be numerically approximated.

Chapter 3

Brief Overview of Modern Parallel Computing

Recent trends in Parallel Computing see GPGPU increasingly spreading in high performance computing centers around the world, as evidenced by the Top 500 list[1]. According to the latest HPC User Site Census data and additional researches, of the 50 most popular application packages mentioned by HPC users, 34 offer GPU support [5]. The spreading of GPUs has been almost driven by NVIDIA, which developed a robust software ecosystem for its hardware, introducing the CUDA language for general purpose computation on proprietary devices. At the same time, other companies have invested in HPC by providing new efficient hardware such as FPGA (field-programmable gate array) systems or solutions base on Intel Xeon Phi many core processors. Every company, as NVIDIA, usually provides its own programming ecosystem, pushing the HPC community to deal with different software tools. In order to simplify the effort required to the developer, new cross-platform parallel standard APIs have been proposed. Among them, two of the most important are OpenCL and OpenMP, this latter starting from the 4.0 specifications. Instead, little has changed in the programming of distributed memory systems in recent years, with MPI (Message Passing Interface) still being the widely adopted solution. Since they were adopted in this work, the above cited software systems are briefly described in the next Sections.

3.1 Heterogeneous Computing with OpenCL

Released on December 2008 by the Kronos Group, OpenCL [57, 113, 94] is an open standard for programming heterogeneous computers built from CPUs, GPUs and other processors. One of the advantages of OpenCL is

[1] `https://www.top500.org/lists/2018/06/`

that it is not restricted, as in the case of CUDA, 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 interfacing process with them. Moreover, OpenCL is open, free, and cross-compatible across vendors and supported by all major hardware producers.

An OpenCL application is subdivided in two parts, one or more running on a compliant device (*device* application) and one running on the CPU (*host* application). The parallel computations are identified by a small different tasks called kernels. The kernel is a special C function, which is compiled at runtime for each target device. The kernels are executed to the target device by the host. Futhermore, the host manages all the devices by an abstract container called `context`. To create a kernel, the host extracts a function from a container of kernels called `program`, mapping all the data to the specific kernel. All kernels that have to be executed with the relative data are sent to the `command queue`, a special structure that allows to run the kernel on the device.

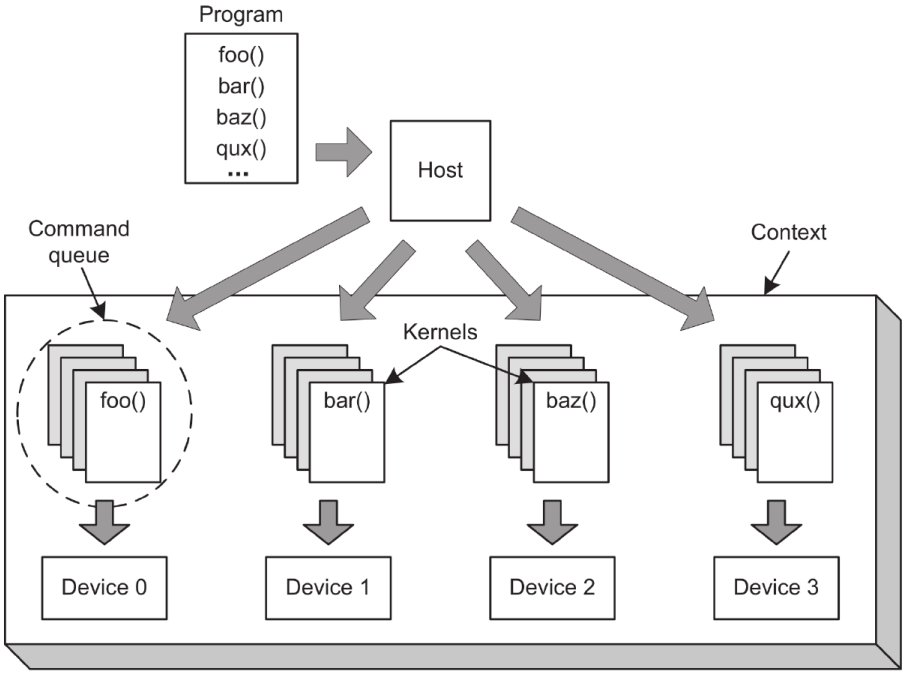


FIGURE 3.1: OpenCL abstraction.

3.1.1 Platform and Devices

In OpenCL, the hardware concept is generalized in platform and device. In particular, the platform identifies the vendor of the target accelerator, whereas the device identifies the device name. During the execution of the program, the user must specify one object that is composed by the name of the platform and the name of the device. This abstraction derives from the heterogeneous nature of OpenCL that allows to target different device from different vendors like AMD, NVIDIA, FPGA, Intel.

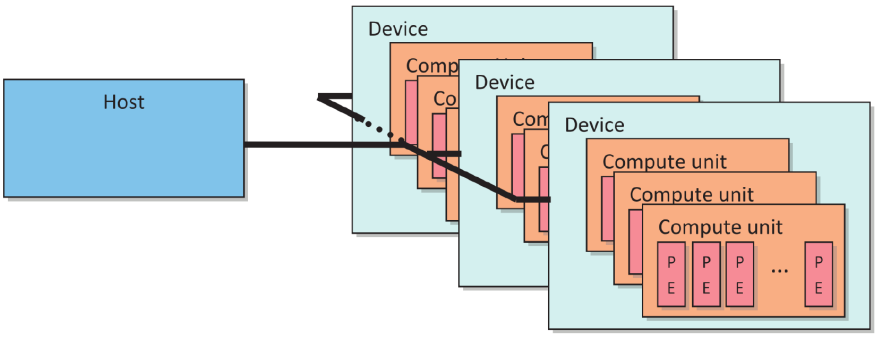


FIGURE 3.2: OpenCL abstraction.

The execution model is represented by different level of abstraction. The smallest execution entity is identified by the **thread**, which executes the kernel code. Threads are also called **work-items** and 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 (Figure 3.2). A work-item has to be considered a thread in terms of its control flow and memory model, but the hardware and the compiler can run multiple work-items on a single thread.

A work-group is a collection of related work-items that are executed on a single compute unit. A work group must map to a single compute unit (a core on a CPU, or - using CUDA terminology - a streaming multiprocessor).

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 demand 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 3.3).

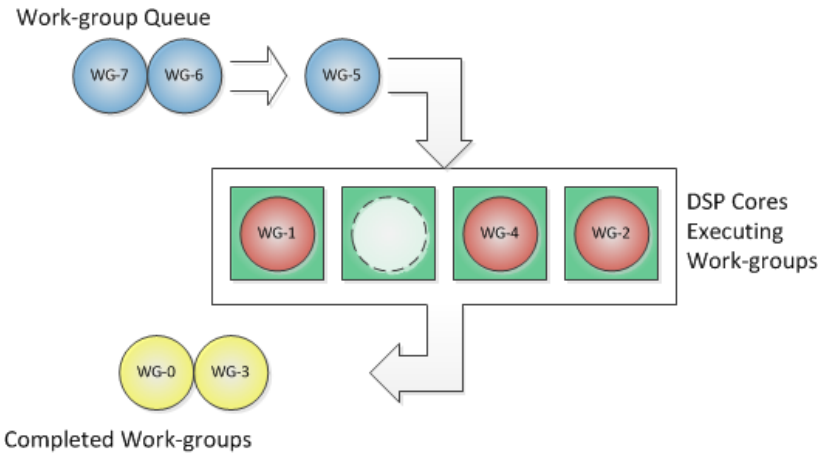


FIGURE 3.3: OpenCL work-groups scheduling. The green boxes represent the computing unit. The circles represent the work-groups. Blue work-groups are waiting to be executed, pink work-groups are currently executing and yellow work-groups have been completed. Each work group is queued for execution and executes on a single computing unit (a GPU multiprocessor, CPU core, etc.) Note that execution order is not guaranteed by the standard.

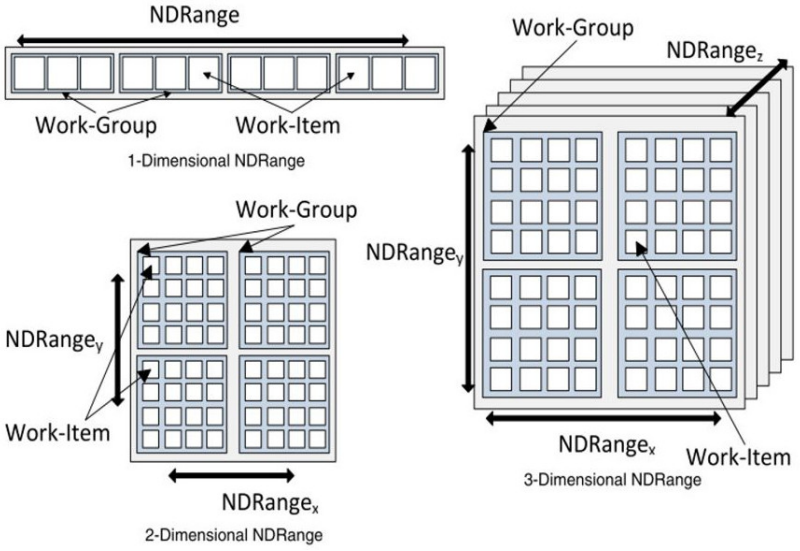


FIGURE 3.4: OpenCL 1D,2D,3D work-items and work-groups partitioning.

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 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

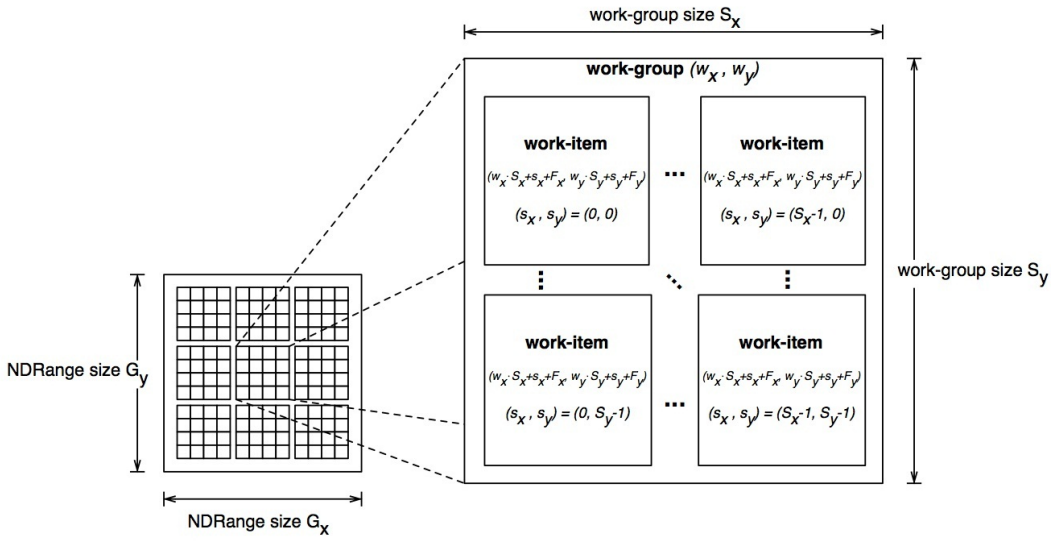


FIGURE 3.5: OpenCL 2D work-items and work-groups detailed partitioning. The computation of global index from local item and group index is also shown.

global ID. Figure 3.4 depicts how items and groups can be arranged when partitioned in $1D$, $2D$ and $3D$. Figure 3.5 shows a $2D$ decomposition with details on global ID computation from local group and thread indices.

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`.

3.1.2 Memory Hierarchy

In general, different devices, for example GPUs and FPGAs, can have different levels of memory. To guarantee code portability, OpenCL defines an abstract memory model. According to this model, memory is subdivided in four parts:

- Global memory;
- Local memory;
- Constant memory;
- Private memory.

Global memory is used to store data passed from/to the host, and can be accessed by all work-items. The keyword `__global` identifies that the specific data is stored in the global memory. A read-only memory, equivalent to the global one in terms of latency and dimension, called constant memory, is also available. Local memory, identified by the keyword `__local`, can be access only by the work-item in a work-group, different work-items from different work-groups can not share the same space local memory. The local memory makes it possible to have access to the local fast memories and caches (similar to CPU L1 caches) in order to minimize the cost of read/write operations. Local memory is generally smaller with respect to the global one, but allows for faster access (about 100× faster on modern GPUs). Eventually, private memory is a special, usually small but fast, portion of memory that can be accessed by the single work-item.

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 3.1).

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

LISTING 3.1: OpenCL API function responsible for memory allocation.

Nevertheless, data can move among different memory levels during kernel execution (e.g., from global to local, and *vice versa*).

The following Listing shows how to create a `context` specifying platform and device.

```
#include <CL/cl.h>

int main(){
 cl_platform_id * platforms;
 cl_uint num_platforms;

 //get platforms number
 clGetPlatformIDs(1, NULL, &num_platforms);
 platforms = (cl_platform_id*)malloc(sizeof(cl_platform_id) *
    num_platforms);
 //get platform
 clGetPlatformIDs(num_platforms, platforms, NULL);

 cl_platform_id first_platform = platforms[0];

 cl_device_id *devices;
 cl_uint num_devices;

 //get number of device in the first platform
```

```
19  clGetDeviceIDs(first_platform, CL_DEVICE_TYPE_ALL,1, NULL, &
      num_devices);
20  devices = (cl_device_id*)malloc(sizeof(cl_device_id) *
      num_devices);
21  //get all devices from the first platform
22  clGetDeviceIDs(first_platform, CL_DEVICE_TYPE_ALL, num_devices,
      devices, NULL);
23
24  cl_context context;
25
26  //create a context
27  context = clCreateContext(NULL, 1, devices, NULL, NULL, NULL);
28
29  return 0;
30 }
```

LISTING 3.2: Example of OpenCL platform and defice specification.

Once platform and device have been defined, and kernels has been developed, these latter have to be grouped in the OpenCL program container, in order to be subsequently extracted, compiled and executed on the target device. Regarding the compilation phase, two alternative approaches can be adopted: static or dynamic. In the static compilation the source code is compiled by an extern compiler producing a binary file. The host reads the binary file and makes the program. In this manner, no kernels source code needs to be released. However, every time the source code of the kernel changes, a new binary file must be created and distributed. On the other hand, the dynamic compilation compiles the kernel's source code at run time. Accordingly, kernel source code have to be available. The following code illustrates an example of dynamic compilation.

```
1 #include <CL/cl.h>
2
3 #define NUM_FILES 2
4 #define PROGRAM_FILE_1 "P1.cl"
5 #define PROGRAM_FILE_2 "P2.cl"
6 #define KERNEL_NAME "k1"
7
8 int main(){
9
10  /*
11  .... allocate and initialize some variables ....
12  */
13
14  cl_program program;
15  FILE *program_handle;
16  char *program_buffer[NUM_FILES];
17  const char *file_name[] = {PROGRAM_FILE_1, PROGRAM_FILE_2};
18  size_t program_size[NUM_FILES];
```

```
19
20  //read all kernels functions from source files
21  for(i=0; i<NUM_FILES; i++){
22   program_handle = fopen(file_name[i], "r");
23   fseek(program_handle, 0, SEEK_END);
24   program_size[i] = ftell(program_handle);
25   rewind(program_handle);
26   program_buffer[i] = (char*)malloc(program_size[i]+1);
27   program_buffer[i][program_size[i]] = '\0';
28   fread(program_buffer[i], sizeof(char),
29   program_size[i], program_handle);
30   fclose(program_handle);
31  }
32
33  //create a program
34  program = clCreateProgramWithSource(context, NUM_FILES,(const
       char**)program_buffer, program_size, &err);
35  //build a program
36  clBuildProgram(program, 1, &device ,NULL, NULL, NULL);
37  //create a kernel
38  cl_kernel kernel = clCreateKernel(program, KERNEL_NAME, NULL);
39
40  return 0;
41 }
```

LISTING 3.3: OpenCL example of dynamic kernel compilation.

3.1.3 Kernel Execution

The abstract structure that allows the communication between the host and the device is the `command queue`. The `command queue` does not only manages the execution of the kernel on the device but also memory communication between host and device and between device and device, as shown in Figure 3.6.

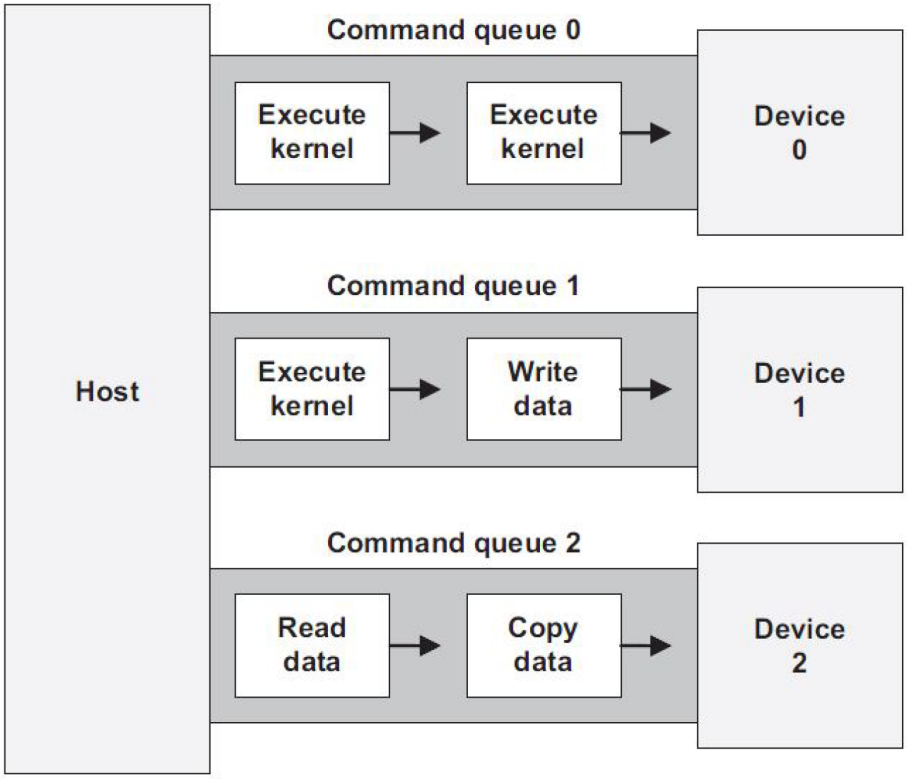


FIGURE 3.6: OpenCL example of command queues.

To execute the kernel on the GPU, the number of work-items and work-groups should be specified through the `clEnqueueNDRangeKernel` function. If the number of work-item and work-group are not specified, OpenCL will set these parameters by itself, by trying to optimize partitioning and resources.

3.2 The NVIDIA CUDA Programming Language

In 2007, NVIDIA saw an opportunity to bring GPUs into the HPC and game world by adding an easy-to-use programming interface, called CUDA, or Compute Unified Device Architecture. CUDA is an extension of the C language that allows GPU code to be written in regular C. The code is either targeted for the host processor (the CPU) or targeted at the device processor (the GPU). The host processor spawns multithread tasks (or kernels as they are known in CUDA) onto the GPU device. The GPU has its own internal scheduler that will then allocate the kernels to whatever GPU hardware is present. Provided there is enough parallelism in the task, as the number of SMs in the GPU grows, so should the speed of the program. You have to ask what percentage of the code can be run in parallel. The maximum speedup possible is limited by the amount of serial code. If you have an infinite amount of processing power and could do the parallel tasks in zero time, you would still be left with the time from the serial code part. Therefore, we have to consider at the outset if we can indeed parallelize a significant amount of the workload.

3.2.1 CUDA Threads and Kernels

A GPU can be seen as a computing device that is capable of executing an elevated number of independent threads in parallel. In addition, it can be thought of as an additional coprocessor of the main CPU (called in the CUDA context Host). In a typical GPU application, data parallel-like portions of the main application are carried out on the device by calling a function (called kernel) that is executed by many threads. Host and device have their own separate DRAM memories, and data is usually copied from one DRAM to the other by means of optimized API calls.

CUDA threads can cooperate together by sharing a common fast shared-memory, implemented using fast DRAM memory similar to first level cache, eventually synchronizing in some points of the kernel, within a so-called thread-block, where each thread is identified by its thread ID as illustrated by Figure 3.7. In order to better exploit the GPU, a thread block usually contains from 64 up to 1024 threads, defined as a three-dimensional array of type dim3 (containing three integers defining each dimension). A thread can be referred to within a block by means of the built-in global variable *threadIdx*. While the number of threads within a block is limited, it is possible to launch kernels with a larger total number of threads by batching together blocks of threads by means of a grid of blocks, usually defined as a two-dimensional array, which is also of type dim3 (with the third component set to 1). In this case, however, thread cooperation is reduced since threads that belong to different blocks do not share the same memory and thus cannot synchronize and communicate with each other. As for threads, a built-in

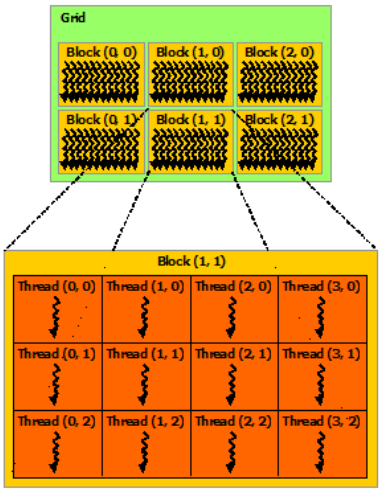


FIGURE 3.7: Grid of thread blocks

global variable, *blockIdx*, can be used for accessing the block index within the grid. Threads in a block are synchronized by calling the syncthreads() function: once all threads have reached this point, execution is resumed normally. As previously reported, one of the fundamental concepts in CUDA is the kernel. This is nothing but a C function, which once invoked is performed in parallel by all threads that the programmer has defined. To define a kernel, the programmer uses the *__global__* qualifier before the definition of the function. This function can be executed only by the device and can be only called by the host. To define the dimension of the grid and blocks on which the kernel will be launched on, the user must specify an expression of the form $<<< Grid\_Size, Block\_Size >>>$, placed between the kernel name and the argument list, such as in the following simple example 3.4:

```
1  // Kernel definition
2  __global__ void VecAdd(float* A, float* B, float* C)
3  {
4      int i = threadIdx.x;
5      C[i] = A[i] + B[i];
6  }
7  int main()
8  {
9      ...
10     // Kernel invocation with N threads
11     VecAdd<<<1, N>>>(A, B, C);
12 }
```

LISTING 3.4: An Example of kernel CUDA invocation.

The above code first defines a kernel called *VectAdd* which will run on all N threads, with the aim to compute in the i-th position of the vector C, the sum of vectors A and B. Assuming that all three vectors have dimension N, each thread in parallel will be the sum of a position. For example, the thread with $ID = 2$ will calculate the sum of $A[2] + B[2]$ and store the result in $C[2]$.

3.2.2 Memory Hierarchy

In CUDA, threads can access different memory locations during execution. Each thread has its own private memory, each block has a (limited) shared memory that is visible to all threads in the same block and finally all threads have access to global memory. In addition to these memory types, two other read-only, fast on-chip memory types can be defined: texture memory and constant memory. In CUDA, memory usage is crucial for the performance. For example, the shared memory is much faster than the global memory and the use of one rather than the other can dramatically increase or decrease performance. By adopting variable type qualifiers, the programmer can define variables that reside in the global memory space of the device (with *__device__*) or variables that reside in the shared memory space (with *__shared__*) that are accessible only from threads within a block. Typical latency for accessing global memory variables is 200-300 clock cycles, compared with only 2-3 clock cycles for shared memory locations. In addition, global memory suffers from coalesced access problems, meaning that access to data should be performed in a particular fashion in order to fetch (or store) the data in the fewest number of transactions [10]. For these reasons, global memory access should be replaced by shared memory access whenever possible. A CUDA C program can allocate global memory of the device in two different ways: through the linear memory or by means of CUDA arrays. CUDA arrays are types of memory optimized for texture management and were not exploited in this work. The more common adopted linear memory type is allocated using the *cudaMalloc()* function for allocating and *cudaFree()* function for memory de-allocation. Once allocated, it is possible to transfer data from the Host memory to the global device memory, and vice-versa, by means of a special call to the *cudaMemcpy()* function. Specifically, *cudaMemcpy()* takes as parameters four kinds of memory type transfers: Host to Host, Host to Device, Device to Host and Device to Device. Note that all of the previous functions can only be called on the host. Figure 3.8 illustrates the GPU typical memory architecture. As shown, the fast on-chip shared memory is shared by all threads of a block.

As expected, to improve performance, variable access should be carried out in the shared memory rather than global memory, wherever possible. Unfortunately, as Figure 3.8 shows, each variable or data structure allocated in shared memory must first be initialized in the global memory, and

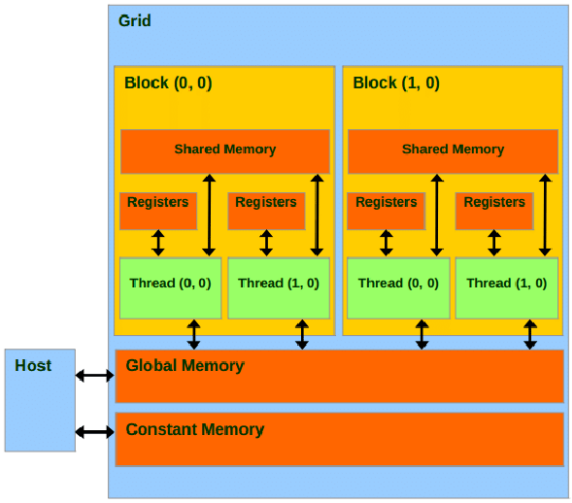


FIGURE 3.8: Typical memory architecture of a Graphic Processing Unit

afterwards transferred in the shared one. This means that to copy data in the shared memory, global memory access must be first performed. So, the more this type of data is accessed, the more convenient is to use this type of memory, while for few accesses it is evident that shared memory might be somewhat degrading. As a consequence, a preliminary analysis of data access of the considered algorithm should be performed in order to evaluate the tradeoff and thus, convenience of using shared memory and how. As reported later in this work, the implementation with a hybrid allocation of variables results in an optimal performance, despite a total shared-memory version as it may be expected.

3.2.3 Programming with CUDA C

CUDA C is an extension of C language that permits to write programs for NVIDIA GPUs. With additional constructs and API functions, the programmer is able to allocate and de-allocate memory on the video card (the *device*), transfer the data from the host device (*host*), launch kernels, etc. The CUDA C extension is built on the basis of the CUDA API driver, a low-level library that allows one to perform all the above steps, but which of course is much less user-friendly. On the other hand, the CUDA API driver offers a higher degree of control and is independent of the particular language (e.g., C, Fortran, Java), being written in assembly language. A typical CUDA program can exploit the computing power of both the host (CPU and RAM) and the device (the GPU and memory devices). What follows is a classic pattern of a CUDA application:

1. Allocation and initialization of data structures in RAM memory;

2. Allocation of data structures in the device and transfer of data from RAM to the memory of the device;
3. Definition of the block and thread grids;
4. Performing one or more kernel;
5. Transfer of data from the device memory to Host memory.

In addition, a CUDA application has parts that are normally performed in a serial fashion, and other parts that are performed in parallel.
Listing 3.5 shows a *VecAdd* problem solved by CUDA.

```
#include <iostream>
#include <cuda.h>
using namespace std;



__global__ void VecAdd(float *A,float *B, float *C)
{
      int  i = threadIdx.x;
      C[i] = A[i] * B[i];
}

int main(int argc,char **argv)
{
   int N = 100;
   int *a, *b;
   int *c;
   int nBytes = N*sizeof(float);

   a = (float *)malloc(nBytes);
   b = (float *)malloc(nBytes);
   c = (float *)malloc(nBytes);
   int *a_d,*b_d,*c_d;

   for(int i=0;i<N;i++)
      a[i]=i,b[i]=i;

   //Allocate memory on the device
   cudaMalloc((void **)&a_d,N*sizeof(float));
   cudaMalloc((void **)&b_d,N*sizeof(float));
   cudaMalloc((void **)&c_d,N*sizeof(float));

   //Copy memory on the device
   cudaMemcpy(a_d,a,N*sizeof(float),cudaMemcpyHostToDevice);
   cudaMemcpy(b_d,b,N*sizeof(float),cudaMemcpyHostToDevice);

   //Kerenl invocation
   vecAdd<<< 1, N>>>(a_d,b_d,c_d,n);

```

```
40    //Kerenl synchronization
41    cudaThreadSynchronize();
42
43    //Copy back the result
44    cudaMemcpy(c,c_d,N*sizeof(float),cudaMemcpyDeviceToHost);
45
46    //free device memory
47    cudaFree(a_d);
48    cudaFree(b_d);
49    cudaFree(c_d);
50    return 0;
51 }
```

Listing 3.5: VecAdd problem solved by CUDA API.

3.3 OpenMP 4.0/4.5 in Clang and LLVM

The OpenMP API has been massively used by the High Performance Community in the last years. OpenMP requires little programming effort to achieve good performance and exposes a single programming interface that is architecture independent. Originally, OpenMP was mainly created to exploit traditional CPU technology, nevertheless nowadays high performance accelerators, like GPUs, Xeon Phi and FPGA, have increasingly leveraged to increase performance on data-parallel applications [89] [97]. This tendency is due to fact that a single accelerator can be 100 times faster than a traditional CPU.

In general, to obtain significant performance on current generations of high performance devices, HPC programmers use different degree of parallelism, usually by hand-tuning their code, performing architecture-specific transformations or using a different variety of languages. This development can be time consuming and requires a big effort from the HPC programmer.

To solve this problem, different researchers of many research centers across EU and US have demonstrated that one solution is to utilize a high-level abstractions (HLA) development strategy based on Embedded Domain Specific Languages (EDSLs) [54] [103]. Here, the aim is to simplify the development, separating what has to be computed from how is computed, totally hiding how the parallel code is implemented.

For this reason, the new OpenMP 4.0 has been released on July 2013, adding support for accelerators. This new release has introduced new compiler directives and library routines, making it easy to execute the computation on modern HPC devices, with respect to the low-level programming language like CUDA and OpenCL, where the HPC programmer is language dependent and has to specify different information in order to exploit the accelerators. The new OpenMP *target* construct allows specifying the code region and the data to be executed on the compliant device. The

data-mapping is completely seamless, the programmer can transfer the data specifying only the new *map* construct, delegating the transfer directly to the embedded implementation. The following code shows an example of OpenMP 4.0/4.5:

```
1 #pragma omp target teams distribute parallel map(to:data[0:DIM])
2 {
3   for(int i = 0; i < X_CELLS; i++)
4     for(int j = 0; j < Y_CELLS; j++)
5       kernel(i,j);
6 }
```

LISTING 3.6: OpenMP 4.0/4.5 example where a double nested loop is executed on the taeget device by means of the `target` directive.

Here, in particular, the new `teams` construct creates a league of thread teams (i.e., CUDA blocks) and the master thread of each team executes the region, while the `distribute` directive distributes the iterations to the master thread of each team.

The new OpenMP 4.5 has been implemented in different C/C++ compilers such as Clang, IBM XL, gcc. In this work, we focused on the Clang compiler, developed by Carlo Bertolli at the Thomas J. Watson Reaserch Center [9] [8] [42] and is based on LLVM . In addition, the code developed for Clang was adapted to the IBM XL compiler for the POWER 8 architecture.

The following sections describe the design of the Clang implementation of OpenMP, focusing on the `libomptarget` offloading library.

3.3.1 Inside the Driver

In the C/C++ LLVM Clang compiler, the main entry point for the application is the *driver*. The driver is responsible for selecting the right commands from the appropriate tool chains given by the information inside the input files passed by the user. Furthermore, the complexity of the driver is extended by the OpenMP offloading support. The driver implementation has to invoke different tool chains from the same set of input files, and, at the same time, give a fully functional binary file containing host and device executable images.

Clang's driver implementation can be divided into two components. The first component is mainly target agnostic, which is in charge of identifying the right sequence of commands and dependency between the input files and the user requests (linking, assembling, etc.). The other component contains all the tool chain data used by the given device. The Target-Agnostic component introduces the new notion of *action* and *job*. The driver represents the compilation phases and their dependencies, by a graph of supported actions (Preprocess, Compile, Assembly, Linking, etc). Each

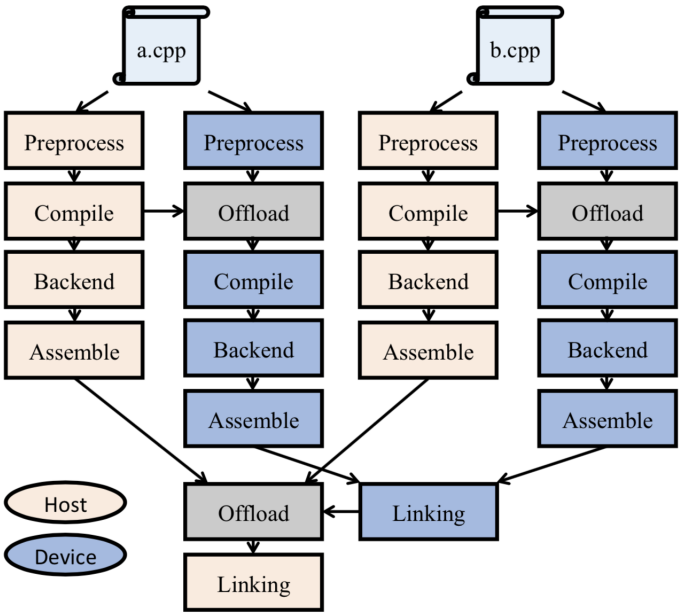


FIGURE 3.9: Action graphs from a compilation of two source files and the related dependences between host and device actions [12].

action identifies a node of the graph. There may be a single graph or one for each input file, depending on the user request. Once all the graphs are produced, the driver scans the graphs from root to leaves, and generates a sequence of *jobs*, representing a pattern of actions. A job is charactered by the tool, the input files, and other target-specific arguments. At the end of the process, a final command is generated to execute the user request.

In general, different target devices may have different tools (assemblers, linkers, etc). In order to use the right tools, the driver memorizes the tool chains and tools information inside the target-dependent part of the driver implementation. Every time a job is created, the target-dependent part is queried.

The OpenMP4 CUDA driver implementation has many parts in common with the already existing CUDA implementation for the compilation of native CUDA language. Nevertheless, the new OpenMP4 needs to support devices from different brands, forcing the development to a more generic implementation. Therefore, the new OpenMP4 proposes a generic *offload action*, managing offloads from different programming models. The *offload actions* are used to specify a host dependence to the device compile action and to specify a device dependence to the host link action. Figure 3.9 presents the action graph generated by the driver when compiling two source files with OpenMP offloading support for host and a single device.

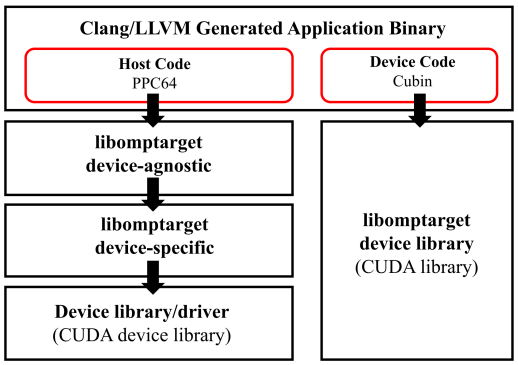


FIGURE 3.10: Clang OpenMP4 generated application binary. From [12].

The OpenMP4 offloading allows the user to specify to the tool chain, which programming model use, and which device to refer to. The flag compilation is defined as *-f ⟨programming model⟩-target=⟨target triple⟩*. For example, if the user wants to target a CUDA device with a OpenMP 4 target, the user should use *-fopenmp-target=nvptx64-nvidia-cuda*.

3.3.2 Runtime Library for Generic Offloading on NVIDIA GPUs

When Clang meets a target offload construct, it generates a set of calls to the runtime library. The runtime library, called `libomptarget`, contains all the necessary entry points generated by the code pragma transformation. The aim of the runtime library is to separate the code generation from the compiler device-specific architecture. The runtime library is designed on a fat binary organization, where the binary can contain different generated code from different OpenMP architectures. The runtime library is composed by three components: the device-agnostic, the device-specific and device library/driver. The device-agnostic is the first components meeting the code. The device-specific is an interface from the calls generated by the device-agnostic and their implementation inside the device library/driver. This double layer of abstraction separates the device-agnostic from the specific device implementation, allowing to easily manage different architectures (cf. figure 3.10).

One of most time expensive tasks in parallel applications is data mapping. Clang OpenMP4 manages data mapping in the first two components of the runtime library, the device-agnostic and the device-specific. To avoid unnecessary data transfer, OpenMP uses a mechanism called *reference counts*. It can be considered as an integer value, zero meaning that the data is not yet passed to the target device, while positive values meaning than the data is already copied. For targeting a CUDA device the data mapping is imple-

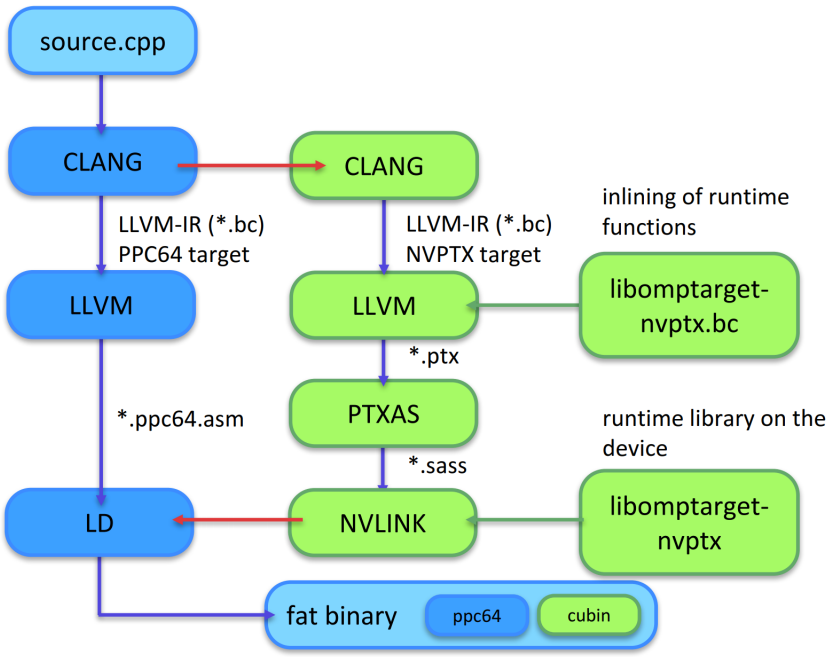


FIGURE 3.11: Clang OpenMP4 compilation process.

mented through the CUDA device library. Initial data mapping is implemented using `cuMemAlloc`, copying with `cuMemcpyHtoD` and `cuMemcpyDtoH`, and de-allocation with `cuMemFree`. While for the data mapping for Generic ELF-enabled Devices, the main used function are: `malloc` for the data allocation, `memcpy` for the copies between host and device, and `free` for the data de-allocation.

As shown before, when Clang compiles a source file it generates a fat binary, containing the compiled code both for the host and device. The compilation of the host and the target region are parallel compiled. For the CUDA target, first Clang LLVM compiles and links for NVIDIA GPUs, generating the PTX code (a low-level parallel thread execution), an intermediate representation of the code, and still not executable by the GPU. The final code translation is computed by two NVIDIA tools: ptxas and nvlink. The output of the process produces a *cubin* object that will be embedded in the fat binary. The compilation is illustrated by 3.11.

3.4 Brief Overview of MPI

In general, there are two main classifications based on memory layout of parallel computers: distributed memory and shared memory. A distributed memory computer can be defined as a collection of nodes, where each node uses its own local memory and can communicate via messages. Through the message passing mechanism, nodes can work together to solve a specific problem. Instead, a shared memory computer can be defined by a single node with multiple processing elements that share a common memory space. The most popular parallel programming model for distributed memory architectures is MPI (Message Passing Interface). MPI is used to send/receive data between processes and can be adopted on either shared or distributed memory architectures. The following Section describes a brief overview of the basic concepts of MPI.

3.4.1 What is MPI

MPI is a library that provides the most optimal message-passing features that have been developed over years, by also supplying a standard de-facto to the Scientific Community. MPI allows to use the provided API to exchange data by sending and receiving messages, without knowledge of the adopted back-end message mechanisms. In fact, the MPI advantage is to hide message mechanisms and its implementation and leaving the programmer free of these specifications, allowing to benefit the interoperability and portability of the code. The MPI API does not depend on any programming language, and can be used with the most diffused programming languages such as C, C++, Fortran, Java and Python.

3.4.2 Communication mechanisms in MPI

In the message-passing model, the processes executing in parallel have a separate address space. The communication mechanism, when invocated, copies a portion of the address space to another address space. In MPI, the Communication is cooperative, meaning that when the first process invokes a *send* then the second process invokes a *receive* operation.

In MPI every process belongs to a group and are identified by a *rank*, which are integers from 0 to n - 1, where n is the total number of processes. Another important object in the MPI environment is the *Communicator* which describes a group of processes and as such must be specified in most point-to-point and collective operations.

The basic (blocking) send API operation is illustrated in Listing 3.7.

```
1 MPI_Send(address, count, datatype, destination, tag, comm)
```

LISTING 3.7: MPI function used to send a blocking message to a specific process.

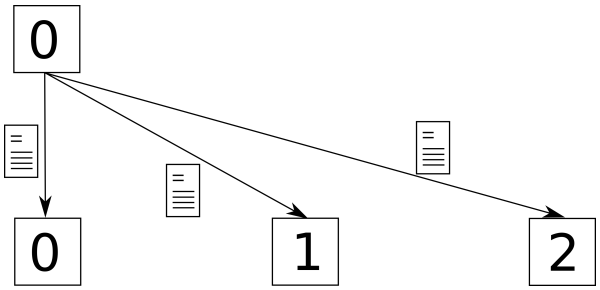


FIGURE 3.12: MPI broadcast mechanisms.

where the sender process needs to specify the address of the buffer, the number of elements in the buffer, the datatype of the buffer, the rank of the destination, the message tag and the communicator. The correspondent receiving API function is shown in Listing 3.8.

```
1 MPI_Recv(address, count, datatype, source, tag, comm, status)
```

LISTING 3.8: MPI function used to receive a message from a specific process.

where the source specifies the sender process and the status holds information of the message. Besides, MPI also provides non-blocking functions that permit asynchronous communications between processes. In this case, specific MPI API functions (e.g., `MPI_Test(MPI_Request *request, int *flag, MPI_Status * status)`) can be used to check if the communication has completed.

MPI also allows for collective communications. The most common API function used for this purpose probably is the `MPI_Bcast` (see Listings 3.9). As the name suggests,it allows the sender process to broadcast the same data to all processes (belonging to the specified communicator). Figure 3.14 illustrates the broadcast mechanism.

```
1 MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root
    , MPI_Comm communicator)
```

LISTING 3.9: MPI function used to broadcast a message to other processes in the specidied communicator.

Another important MPI API communication function is `MPI_Scatter` (Listings 3.10). The difference between `MPI_Bcast` and `MPI_Scatter` is that the latter sends *different portions* of the data to the corresponding processes, while the broadcast communication sends the *same* data.

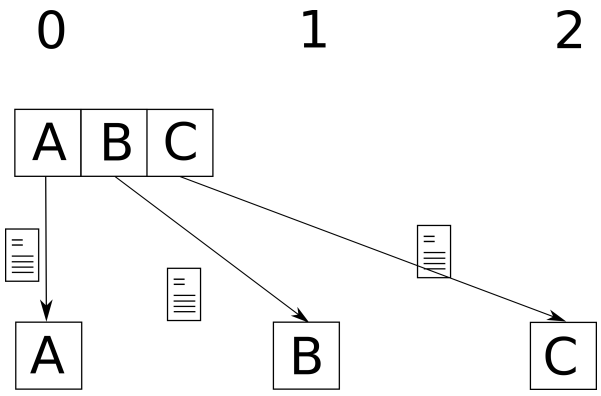


FIGURE 3.13: MPI scatter mechanisms.

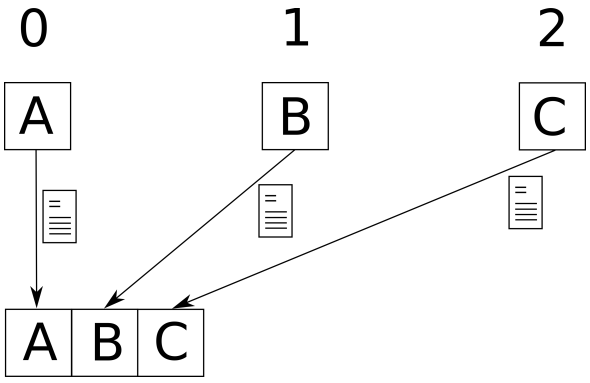


FIGURE 3.14: MPI gather mechanisms.

```
1 MPI_Scatter( void* send_data, int send_count, MPI_Datatype
    send_datatype, void* recv_data, int recv_count, MPI_Datatype
    recv_datatype, int root, MPI_Comm communicator)
```

LISTING 3.10: MPI function to broadcast a portion data to the processes.

Another important MPI API communication function is `MPI_Gather`, corresponding to the inverse of `MPI_Scatter`. It allows to take elements from many processes and gathers them to one single process. With this function, only the master process receives all the data in one buffer. Usually, the `MPI_Gather` is used in final part of an MPI program where the results of the problem need to be reconstructed for output purposes.

Listings 3.11 shows the scatter MPI API function.

```
1 MPI_Gather(void* send_data, int send_count, MPI_Datatype
    send_datatype, void* recv_data, int recv_count, MPI_Datatype
```

```
    recv_datatype, int root, MPI_Comm communicator)
```

LISTING 3.11: MPI API gather function.

3.4.3 The MPI+X programming models

Modern HPC hardware is usually composed by a large number of nodes with multi-core CPUs and some attached accelerators. In order to achieve best performances, the use of only MPI is usually not sufficient, and other programming models should be adopted simultaneously. Over the past 20 years, the combination of MPI and OpenMP has lead the parallel programming model, maintaining a good level of performance. However, the programming of next-generation HPC systems will drive to more hybrid model called MPI+X, where X is one of following programming models: OpenMP, OpenACC, OpenMP 4, OpenCL, CUDA or Vulkan. One the main disadvantages of the MPI+X techniques is the conversion from a combination of programming models to another, which can be very hard and time consuming. This is due to many factors, i.e., in-depth knowledge of the machine topology and memory hierarchy, compute-memory synchronization, vendor programming language dependence and other system characteristics. These kinds of limitations bring the programmers to tradeoffs between performance, productivity and portability.

The next HPC frontier is to overcome these obstacles. To reach this goal, a high level portable programming strategy has been introduced by many HPC research centers over the world. MPI will remain the primary message passing library to exploit cluster architectures together with other programming models. The main goal is not to force the user to rewrite the code for different architectures, requiring low-level or explicit programming models to obtain the best performance. Indeed, the user should only choose which hardware to exploit, such as multi-core, multi-node or many cores architectures, and delegating the management of optimized parallel code to the high level model. This alternative approach will be discussed later in the next Section.

Chapter 4

Domain Specific Languages for Parallel Computing

Software development for HPC systems is still hard and error prone, though standard APIs have been proposed as shown in Chapter 3. For instance, OpenMP4/OpenCL have become the standard to accelerate computation on multi- and many-core devices, while MPI for parallel computation on clusters of workstations. Although considerable progress has been made with respect to proprietary software solutions, developers are still forced to use different languages/APIs (each one characterized by a different underlying conceptual model), resulting in a big effort during both development and long-term management/improvement of the source code, with a consequent waste of resources and time. In addition, Scientist/Engineers would take a great advantage if they could exploit the computational power of moder parallel computers without being forced to deal with parallel programming details.

One possible solution to the above issues is offered by high-level abstraction (HLA) development strategy based on Domain Specific Languages (DSLs). Here, the aim is to simplify and make faster the development process by allowing to implement the application by referring to a serial development model, delegating the parallelization process to the library for different parallel computing solutions. Specifically, this Chapter introduces two DSLs that I have contributed to, namely OpenCAL (Open Computing Abstraction Layer) and OPS (Oxford Parallel library for Structured grid computation). The development of OpenCAL was mainly carried out at the University of Calabria (Italy). My contribution is related to the design and implementation of the OpenCL and MPI based components, namely OpenCAL-CL and OpenCAL-CLM, while my contribution to OPS has concerned in the introduction of the new OpenMP4 programming model. The OPS research was mainly carried out during a six-month internship period at the University of Warwick (UK), under the supervision of Prof. Gihan

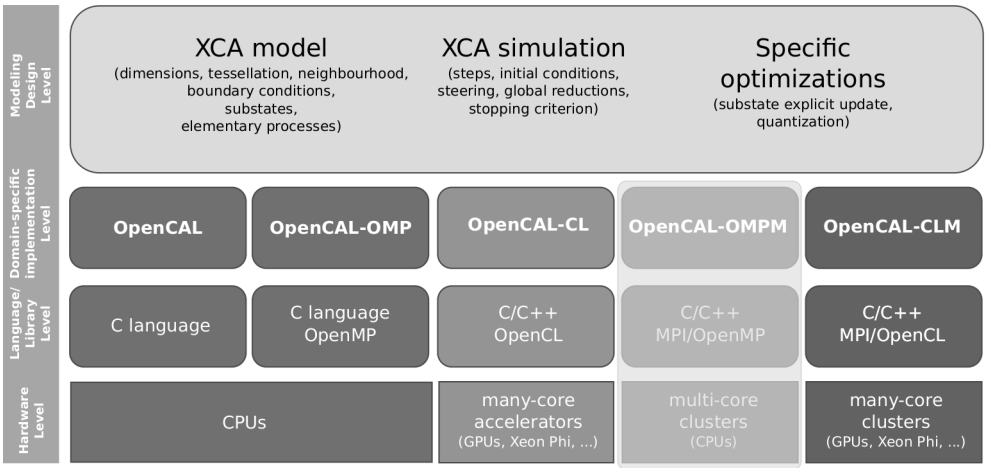


FIGURE 4.1: OpenCAL architecture. At the higher level of abstraction, the model, together with the simulation process and possible optimizations, is designed. The OpenCAL libraries can be found at the implementation abstraction layer, allowing for a straightforward implementation of the designed computational model. OpenCAL-based applications can be therefore executed at the hardware level on both multi-core CPUs and many-core devices. The execution on distributed memory systems is currently under development.

Mudalige and Dr. Carlo Bertolli from the IBM Thomas J. Watson Research Center (USA). Different examples of application are also described, which were taken into account for evaluating permformances on different hardware configurations. In particular, a Sobel graphics edge detection filter, a Jiulia set fractal generator and the SciddicaT slow-moving fluid flow simulation model were implemented in OpenCAL, while the Cloverleaf and Tealeaf benchmarks in OPS.

4.1 OpenCAL

In this section we describe the software architecture, main structures and underlying algorithms of the OpenCAL library, besides a first example of application to highlight how easy model development is. The serial version of the library will be simply referred as OpenCAL in the following, while OpenCAL-OMP and OpenCAL-CL will refer to the OpenMP- and OpenCL-based parallelizations, respectively. Eventually, the distributed memory version of OpenCAL will be referred as OpenCAL-CLM. The main API data types and functions specifications are here presented. A full API description can be found in the OpenCAL user guide on GitHub().

4.1.1 Software Architecture

The OpenCAL architecture is depicted in Figure 4.1. At the higher level of abstraction, the Scientist conceptually designs the computational model, by referring to 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. Note that, being supported by OpenCAL, at this stage some specific optimizations can be applied. 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.

The different versions of OpenCAL can be found in the implementation level. Since they provide high-level data structures and algorithms that match the higher abstraction level components, all of them allow for a straightforward implementation of the previously designed computational model, by also allowing to ignore low-level issues like memory management and I/O operations. All OpenCAL versions are written in C for the maximum efficiency and, as pointed out by the language/library level, the OpenMP and OpenCL APIs were considered for implementing the corresponding parallel versions of OpenCAL. Finally, at the hardware level, depending on the adopted version of the library, execution can be performed on single- and multi-core CPUs, as well as on many-core accelerators like GPUs, transparently to the user. Figure 4.1 also shows hybrid MPI/OpenMP and MPI/OpenCL parallel implementations of OpenCAL. The latter, a preliminary implementation of which is in an advanced development state, is that we will refer as OpenCAL-CLM in this paper.

4.1.2 OpenCAL Domain Specific API Abstractions

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 specific OpenCAL-CL data types), followed by a type identifier formed by one or more capitalized 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, respectively (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 characterized by the `cal` prefix (or `calcl` for some specific 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 of API functions with one optional and two mandatory suffixes: the first one, if present, indicates that the function is able to access neighborhood data (`X` is the symbol commonly used in the XCA formalism to refer to the neighborhood), while the other two indicate the domain dimension and the basic type of the data to be accessed, respectively.

One of the most important API objects is the *model*, which is the implementation level object corresponding to the XCA model formalized at the design level. It is simply declared as a pointer to the `CALModel{2D|3D}` built-in data type, and can straightforwardly be defined by means of the `calCADef{2D|3D}()` function. The model object essentially allows to define the domain dimensions (2D and 3D models are natively supported, even if 1D models can be defined as degenerate case of the 2D one), the size of each of them, the cell geometry (square, rectangular and hexagonal cells are supported), the domain topology (e.g., if a 2D domain has to be considered as bounded or as a torus) and the neighborhood pattern, besides embedding a built-in data structure needed by the quantization optimization algorithm (cf. below in this Section). As regards neighborhoods, a set of predefined patterns is provided (e.g., the `CAL_MOORE_NEIGHBORHOOD_{2D|3D}` enumeral refers to the Moore pattern), even if generic neighborhoods can be

explicitly defined (by using the `CAL_CUSTOM_NEIGHBORHOOD_{2D|3D}` enumeral at definition time and then the `calAddNeighbor{2D|3D}()` function to add neighbors to the initially empty set). Besides the predefined Moore neighborhood, the von Neumann one and 2D-specific hexagonal neighborhoods are also provided by the API. The model object seamlessly manages both the data, mainly represented by *substates* (which are declared as `CALSubstate{2D|3D}{b|i|r}` objects), and the local rules of evolution for the system (i.e., the automaton *transition function*), expressed in terms of *elementary processes* (that are defined as callback functions or OpenCL kernels, depending on the specific OpenCAL implementation). For this purpose, both substates and elementary processes must be registered to the model object (by means of the `calAddSubstate{2D|3D}{b|i|r}()` and `calAddElementaryProcess{2D|3D}()` functions, respectively), which in this way can store pointers to each of them for subsequent seamless indirect access. Note that, a further device-side model object (`CALCLModel{2D|3D}`) is provided by the OpenCAL-CL API, which makes data transfer and parallel execution on OpenCL compliant devices transparent to the user. This latter object is declared as a pointer to `CALCLModel{2D|3D}`, and defined by means of the `calclCADef{2D|3D}()` function. Specifically, data transfer from the host to the device global memory is performed at definition time, while data is seamlessly copied back to the host at the end of the simulation process, by minimizing in this way time consuming host to/from device data movements during the computation. To further speed-up the device-side execution, the library also provides the `calclGlobaltoLocal[X]()` API function that can be used within the kernels to transfer data (i.e., central cell and neighborhoods states) from the global to the faster local memory.

According to the XCA computational paradigm, substates define specific characteristics considered to be relevant for the system initial state definition and its evolution. For instance, in a fluid-dynamic computational model, different substates can be used for modeling mass, viscosity and velocity field components. Each substate object has the same extent of the whole computational domain, so that each cell is characterized by specific substates values. Implicitly, this leads to a SoA (Structure of Arrays) approach, which proved to be the most effective in the case of parallel programming on GPUs (see e.g., [10]). For efficient access due to memory coalescence issues, substates objects are implemented by means of linearized arrays. Nevertheless, internal format is transparent to the user, which can access data by means of multidimensional indices and neighborhoods identifiers (e.g., the `calGet[X]{2D|3D}()` function allows to retrieve the current state of the central cell and - if the `X` optional suffix is present - its neighbors, while the `calSet{2D|3D}()` one permits to update the central cell's state). Behind the scene, substates are defined by means of two *computational layers*: the *current* layer represents a read-only memory and is used for retrieving central and neighboring cells current states, while the *next*

one is used only for updating the new value of the central cell. Once all new states have been written to the next layer, the substate is seamlessly updated (even if the update phase can be made explicit by means of the `calUpdateSubstate{2D|3D}{b|i|r}` function - cf. below in this Section), i.e. the next layer is copied into the current one, and the substate object is ready for further processing. Note that, as already stated, in the case of OpenCAL-CL, substates are updated device-side, by allowing to perform the whole computation process on the device. Eventually, each OpenCAL implementation also provides *single layer substates*, which only consist of the current computational layer. They are declared as standard double-layered objects, even if the *next* layer is lost at registration time, where the `calAddSingleLayerSubstate{2D|3D}{b|i|r}()` function must be used (instead of the `calAddSubstate{2D|3D}{b|i|r}()` one). Single layer substates can be considered for internal transformations processing, i.e. for those modeling specific rules which determine the substate change within the cell as a function of the central cell state only. Not needing to be updated, they represent a lighter memory and more efficient alternative to double-layer substates.

The system evolution is obtained by applying the elementary processes composing the transition function in the same order in which they have been registered to the model object (even if the predefined order can be overridden - cf. below in this Section) and, after the application of each of them, by updating the involved double-layer substates. As anticipated, elementary processes are implemented by means of callback functions or OpenCL kernels and their execution is transparently performed by the library (even if, in case of host-side execution, elementary processes application, and also substates updating, can be made explicit - cf. below in this Section). According to the XCA paradigm, each cell must appear to be updated simultaneously to each other (*implicit parallelism*). For this purpose, a pool of concurrent threads/work-items should apply the elementary process simultaneously to each cell of the computational domain. However, depending on the domain dimensions, this is not always possible, even in the case of parallel execution on many-core devices. Nevertheless, implicit parallelism is guaranteed in each OpenCAL implementation thanks to double-layer substates. In fact, by using the current layer as read only memory and the next one for updating purposes only, cells appear to be simultaneously updated with respect to each other, even in the case of serial computation. In this respect, an elementary process is equivalent to a `parallel for` loop, which transparently applies its local rule of evolution simultaneously to each cell of the computational domain. In the case of parallel execution, a data parallel approach is adopted. In particular, the domain is decomposed in uniform chunks in the case of OpenCAL-OMP, while a one cell/one work-item decomposition is adopted in OpenCAL-CL. OpenCAL-CLM currently adopts a classical chunk-based domain decomposition with seamless halos exchange,

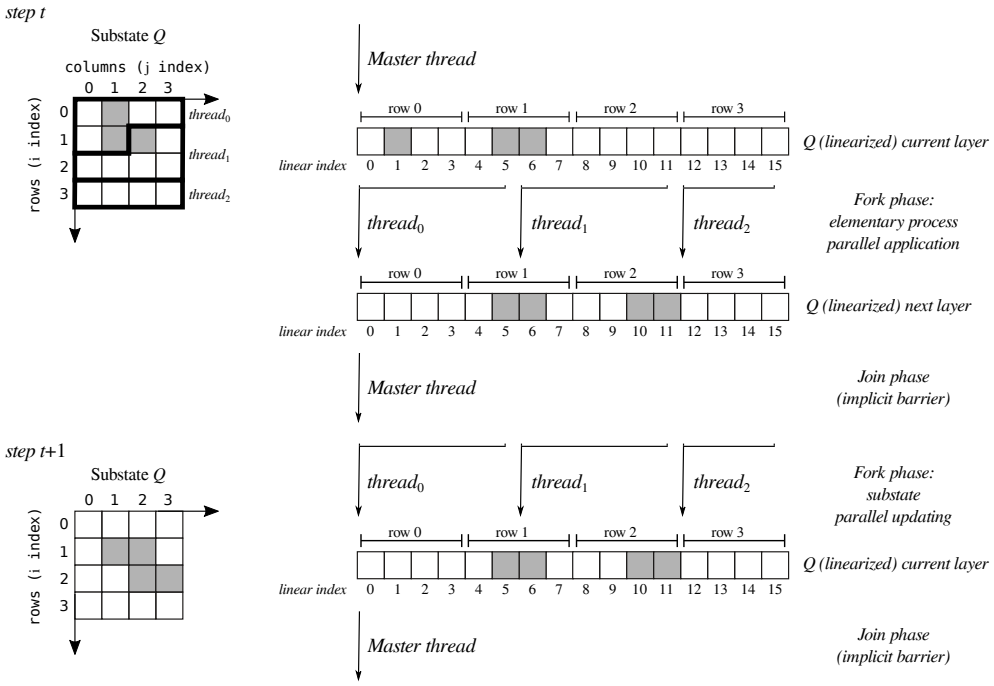


FIGURE 4.2: An example of OpenCAL-OMP parallel application of an elementary process to a substate Q and its subsequent parallel updating. The computational domain is initially partitioned by means of a pool of three threads (fork phase). These latter concurrently apply the elementary process by reading 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.

and also a one cell/one work-item model at GPU level. Note that double layer substates allow for a lock-free parallelization in all cases. In fact, no race conditions can occur since, in particular, the update phase is limited by definition of the XCA computational paradigm to the memory location associated with the central cell. An example is shown in Figure 4.2 for the case of OpenCAL-OMP, where a pool of three threads concurrently process an uniformly partitioned domain for both elementary processes application and substates updating. In this case, the third thread is completely wasted, since it only processes a subset of stationary cells, and therefore a load unbalance occurs. In such a case, a dynamic scheduling is seamlessly adopted in OpenCAL-OMP to mitigate the unbalance among chunks. Regarding the OpenCAL-CL specific case, a grid of OpenCL work-items is adopted for a SIMD-based parallelization. Depending on the dimension of the computational model, two- or three-dimensional OpenCL index spaces (i.e. OpenCL NDRanges) are transparently considered, while a one-dimensional one is adopted in the case the quantization optimization is exploited (cf. below in this Section). The number of work-items to be adopted is evaluated for each model dimension by preliminary querying OpenCL for the (device-dependent) preferred work-group size multiple w_s (i.e. the warp/wavefront size in NVIDIA/AMD GPUs), 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 work-groups. The choice of the number of work-groups to be considered (and therefore the work-group size) depends on the device architecture and can be both transparently determined (default setting), or explicitly set for finer tuning.

In case of host-side execution, i.e. when OpenCAL and OpenCAL-OMP are considered, simulation execution is managed by a specific *simulation* object, that must be declared as a pointer to the `CALRun{2D|3D}` data type and then defined by means of the `calRunDef{2D|3D}()` function. Nevertheless, in the case of device-side execution, i.e. when OpenCAL-CL is considered, the role of the simulation object is played by the device-side model. Among others, the simulation object defines the substates updating policy: in case of implicit scheme, the built-in transition function is applied (i.e. elementary processes are applied in the same order in which they have been registered to the model and all registered substates are updated after the application of each of them); when the explicit policy is adopted, the transition function must be overridden and elementary processes explicitly applied, as well as substates explicitly updated. Allowing to avoid the update of unmodified substates (therefore unneeded memory copy operations)

the OpenCAL implicit naive approach, which is provided as first instance to allow the developer to completely ignore underlying data structures issues, can be overcome. For this and other purposes, the simulation object can optionally register one or more global callback functions, listed below:

- `init()`: It is executed once before the simulation loop and can be used used to set the initial conditions of the system.
- `globalTransition()`: It overrides the built-in transition function and can be used to redefine the execution order of the registered elementary processes and to perform selective substates updating. The function also allows to perform global operations over the computational domain, e.g., reductions. Built-in 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.
- `steering()`: It is executed at the end of each computational step and can be used to perform generic global operations, as well as global reductions.
- `stopCondition()`: It is checked after the steering function (if defined) at the end of each computational step and can be used to define a stopping criterion for the simulation. Differently to the other callbacks, which do not return any value, the function returns a Boolean value: *true* if the termination criterion is satisfied, *false* in the other case.

Algorithm 1 outlines the OpenCAL implicit simulation process, that applies the default model transition function if not differently specified. In the other case, the `globalTransition()` function is applied. The `init()` function, if defined, is called first and subsequently active cells (if quantization is enabled - cf. below in this Section) 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 *false*, respectively. The main simulation loop follows, which is triggered by the `calRun{2D|3D}()` or `calclRun{2D|3D}()` function call, depending on the adopted OpenCAL implementation. At each step, after the application of each elementary process to each cell of the computational domain, active cells (if the quantization optimization is used) and substates are updated. If defined, the `steering()` global function is therefore called and active cells and substates again updated. The `stopCondition()` function is eventually called and the step counter increased. The simulation loop continues while the `halt` variable, whose value is set by the `stopCondition()` function, is *false* or the final step of computation is met.

Algorithm 1: OpenCAL main implicit simulation process.

```
init()                                  // Call the init() global function
if quantization then
    update (A)                          // Update the array of active cells
forall q ∈ Q do
    update (q)                          // Update the substate q
step ← initial_step
halt ← false
while ¬halt ∧ (step ≤ final_step ∨ final_step = CAL_RUN_LOOP) do
    forall e of σ do
        forall (A ≠ ∅ ∧ i ∈ A)∨ i ∈ R do
            e(i) // Apply the elementary process e to the cell
             i
        if quantization then
            update (A)                  // Update the array of active cells
        forall q ∈ Q do
            update (q)                  // Update the substate q
    steering()                          // Call the steering() global function
    if quantization then
        update (A)                      // Update the array of active cells
    forall q ∈ Q do
        update (q)                      // Update the substate q
    halt ← stopCondition()              // Check the stop condition
    step ← step + 1
return
```

4.1.3 The Quantization Optimization

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 in 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 sub-region, proved to be a simple but effective approach in different cases (see e.g., [36]). However, HRBB demonstrated its limit in the simulation of scattered phenomena, where the hyper-rectangle can easily grow up and include the whole domain, 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 only of the active cells during the simulation, by restricting the computation only to this set (see e.g., [45]). The activation state for a cell generally depends on the specific system to be simulated. In many cases, for instance 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, was considered in OpenCAL.

Its implementation is based on a compacted array, A, containing the computational domain's active cells coordinates. A, which is initially empty, is generally defined at the system initialization stage. For this purpose, and also to maintain A updated, the activation value must be explicitly set (to *true*, which means that the cell is active, or to *false*, on the other hand) only for the cells that change state during the current computational step (a Boolean working array F is updated in this preliminary step). An efficient stream compaction algorithm (as implemented in [52]) is therefore transparently applied at the end of each computational step to update the set A, based on the activation states stored in F. For illustrative purposes, an example of application of the OpenCAL-CL stream compaction algorithm is described in Figure 4.3. Note that, in this way, domain-sized data is processed only once during the stream compaction phase, while only the subset of cells belonging to A is involved in the remaining actual computation. The quantization optimization clearly introduces an overhead. However, depending on the domain dimension and the affected area (or volume), as well as

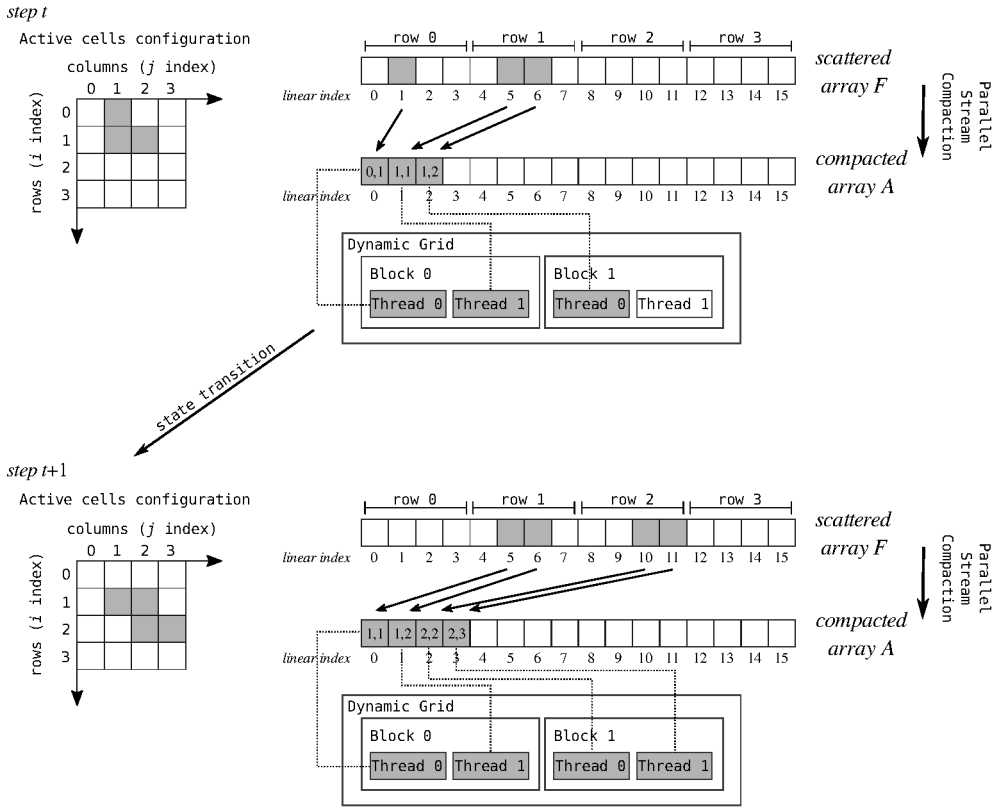


FIGURE 4.3: An example of application of the OpenCAL-CL parallel stream compaction algorithm. Active cells are represented in gray within a two-dimensional 4x4 matrix of flags, implemented as a linearized array, F. The parallel stream compaction algorithm processes F and produces the compacted array 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.

on the computational intensity of the model, it can produce a considerably speed-up of the overall computational process.

Using the quantization optimization is quite straightforward. Firstly, it must be enabled at model definition time by 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 no 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 guaranteed (even if the actual thread performing the operation is undefined - cf. e.g., [11]). 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 lead 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.

4.1.4 Conways Game of Life in OpenCAL

As a first illustrative example of the library, we here present the OpenCAL implementation of the Turing complete Conway's Game of Life (simply Life in the following), one of the most simple, yet powerful example of CA [56]. It can be thought as an infinite two-dimensional grid of square cells, each of them being in one of two possible states, *dead* or *alive*. Every cell interacts with the eight adjacent neighbors belonging to the Moore neighborhood. At each time step, one of the following transitions occur:

1. Any alive cell with fewer than two alive neighbors dies, as if by loneliness;

2. Any alive cell with more than three alive neighbors dies, as if by overcrowding;

3. Any alive cell with two or three alive neighbors lives, unchanged, to the next generation;

4. Any dead cell with exactly three alive neighbors comes to life.

Formally, Life can be defined as:

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

where

1. R is the set of points, with integer coordinates, which defines a two-dimensional toroidal cellular space;

2. $X = \{(0,0), (-1,0), (0,-1), (0,1), (1,0), (-1,-1), (1,-1), (1,1), (-1,1)\}$ is the Moore neighborhood, i.e. the set of relative coordinates that, when added to the coordinate vector of the central cell, give the absolute coordinates of the neighboring cells;

3. $Q = \{0,1\}$ is the set of cell states, 0 representing the dead state, 1 the alive;

4. $\sigma : Q^9 \rightarrow Q$ is the deterministic cell transition function. It is composed by one elementary process, which implements the aforementioned transition rules.

In the following, two OpenCAL/OpenCAL-OMP and OpenCAL-CL implementations of Life are presented and commented. The program in Listings 4.5, containing the main application, and 4.2, containing the transition function, shows a possible OpenCAL/OpenCAL-OMP implementation of Life.

```
1  #include <OpenCAL/cal2D.h>     //#include <OpenCAL-OMP/cal2D.h>
2  #include <OpenCAL/cal2DIO.h>   //#include <OpenCAL-OMP/cal2D.h>
3  #include <OpenCAL/cal2DRun.h> //#include <OpenCAL-OMP/cal2D.h>
4  #include <stdlib.h>
5
6  struct CALModel2D* life;
7  struct CALSubstate2Di* Q;
8  struct CALRun2D* life_simulation;
9
10 void lifeTransitionFunction(struct CALModel2D* life, int i, int
       j);
11
12 int main() {
13   life = calCADef2D(8, 16, CAL_MOORE_NEIGHBORHOOD_2D,
       CAL_SPACE_TOROIDAL, CAL_NO_OPT);
14   life_simulation = calRunDef2D(life, 1, 1, CAL_UPDATE_IMPLICIT)
       ;
```

```
15
16   Q = calAddSubstate2Di(life);
17   calAddElementaryProcess2D(life, lifeTransitionFunction);
18
19   calInitSubstate2Di(life, Q, 0);
20   calInit2Di(life, Q, 0, 2, 1);
21   calInit2Di(life, Q, 1, 0, 1);
22   calInit2Di(life, Q, 1, 2, 1);
23   calInit2Di(life, Q, 2, 1, 1);
24   calInit2Di(life, Q, 2, 2, 1);
25
26   calSaveSubstate2Di(life, Q, "./life_0000.txt");
27   calRun2D(life_simulation);
28   calSaveSubstate2Di(life, Q, "./life_LAST.txt");
29
30   calRunFinalize2D(life_simulation);
31   calFinalize2D(life);
32   return 0;
33 }
```

LISTING 4.1: An OpenCAL/OpenCAL-OMP implementation of Conway's Game of Life.

Concerning the main application, header files are included at lines 1-3 that allow to define the required 2D model and substate, besides providing some basic I/O facilities. The model object, `life`, is declared at line 6, while lines 7 and 8 declare the required substate, `Q`, and simulation object, `life_simulation`, respectively. These objects are defined later in the `main` function at lines 13-14. In particular, the model definition function, `calCADef2D()`, takes the domain dimensions (an 8 rows $\times$ 16 columns domain is considered here), the neighborhood pattern (Moore in this case), the boundary topology (a toroidal domain is considered in the example to account for an unlimited domain) and the optimization to be used (the quantization optimization is not adopted in the example). Furthermore, the simulation object definition function, `calRunDef2D()`, requires the address of a model object to be evolved (which is `life` in this example), the initial and final simulation steps (set both to one to perform a single computational step), and eventually the substates update policy (here set to implicit) as parameters. Line 16 allocates memory and registers the integer-based Q substate to the model object by means of the `calAddSubstate2Di()` function, while line 17 registers an elementary process to `life` by means of the `calAddElementaryProcess2D()` function. Here, `lifeTransitionFunction` is a developer-defined callback function implementing the model local rules. At this aim, the `calGet[X]2Di()` and `calSet2Di()` API functions are used for reading and updating purposes at cell level (cf. Listing 4.2).

```
1  void lifeTransitionFunction(struct CALModel2D* life, int i, int
       j) {
2    int sum = 0, n;
3    for (n=1; n<life->sizeof_X; n++)
4      sum += calGetX2Di(life, Q, i, j, n);
5
6    if ((sum == 3) || (sum == 2 && calGet2Di(life, Q, i, j) == 1))
7      calSet2Di(life, Q, i, j, 1);
8    else
9      calSet2Di(life, Q, i, j, 0);
10 }
```

LISTING 4.2: The OpenCAL/OpenCAL-OMP callback function implementing the elementary process of Game of Life application shown in Listing 4.5.

The `calInitSubstate2Di()` function at line 19 initializes the Q substate to 0 (for both the current and next layers), while lines 20-24 define a so called *glider* pattern by means of the `calInit2Di()` function. The `calSaveSubstate2Di()` function at line 26 saves the Q substate to file, while the subsequent call to `calRun2D()` 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 28 to save the new (last) system configuration, while the last two API calls release memory previously allocated by OpenCAL. The `return` statement at line 32 ends 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  struct CALModel2D* life;
10 struct CALSubstate2Di* Q;
11
12 int main() {
13   struct CALCLDeviceManager* calcl_device_manager =
       calclCreateManager();
14   calclPrintPlatformsAndDevices(calcl_device_manager);
15   CALCLdevice device = calclGetDevice(calcl_device_manager,
       PLATFORM_NUM, DEVICE_NUM);
16   CALCLcontext context = calclCreateContext(&device);
17   CALCLprogram program = calclLoadProgram2D(context, device,
       KERNEL_SRC, NULL);
18   // <Missing>: Here source code as in lines 12-24 from Listing
       1
19
```

```
20  struct CALCLModel2D* life_device = calclCADef2D(life, context,
      program, device);
21
22  CALCLkernel life_transition_function =
      calclGetKernelFromProgram(&program, LIFE_TRANSITION_FUNCTION)
      ;
23  calclAddElementaryProcess2D(life_device, &
      life_transition_function);
24
25  calSaveSubstate2Di(life, Q, "./life_0000.txt");
26  calclRun2D(life_device, 1, 1);
27  calSaveSubstate2Di(life, Q, "./life_LAST.txt");
28
29  calclFinalizeManager(calcl_device_manager);
30  calclFinalize2D(life_device);
31  calFinalize2D(life);
32  return 0;
33 }
```

LISTING 4.3: An OpenCAL-CL implementation of Conway's Game of Life.

According to OpenCL, a possible OpenCAL-CL implementation of Life is subdivided in two different parts: a device- and a host-side application. The host-side application, running on the CPU and controlling the computation on the compliant device (e.g., a GPU), is shown in Listing 4.3. The `calcl2D.h` header file is included at lines 1-2, together with the OpenCAL `cal2DIO.h` header for I/O purposes. The path of the directory containing the transition function elementary processes (implemented as OpenCL kernels) is defined at line 4, while the name of the only kernel required at line 5. Lines 6-7 define the OpenCL identifiers for the platform and device to be used. Note that OpenCAL-CL can query the system for platforms and compliant devices, by allowing the user to select them at run time. However, for the sake of simplicity, in this example the first device belonging to the first platform is set. The substate numerical handle `Q` is also defined at line 8, as it is required to refer to the object from both the host and device application. Lines 13-16 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 17 reads kernels (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. The host-side object definition follows, together with the substate and its initialization (cf. line 18). Line 20 defines the `life_device` device-side object by means of the `calclCADef2D()` function, also by performing host to device data transfer transparently to the user. The elementary process (which actually is an OpenCL kernel) is therefore extracted from the previously compiled program by means of the `calclGetKernelFromProgram()` function at

line 22. It returns an OpenCL kernel, which is subsequently registered to the device-side model by means of the `calclAddElementaryProcess2D()` function at line 23. Lines 25 and 27 are used to save the CA state before and after simulation execution, respectively. The CA simulation is executed by means of the `calclRun2D()` function at line 26. In this example, the only elementary process defined is executed in parallel on the compliant device in a transparently way to the user. Eventually, lines 29-31 perform memory deallocation for the previously defined objects. The return statement at line 32 terminates the program.

```
1 #include <OpenCAL-CL/calcl2D.h>
2 #define DEVICE_Q 0
3
4 __kernel void lifeTransitionFunction(_CALCL_MODEL_2D) {
5   calclThreadCheck2D();
6   int i = calclGlobalRow();
7   int j = calclGlobalColumn();
8   CALint sizeof_X = calclGetNeighborhoodSize();
9
10  int sum = 0, n;
11  for (n=1; n<sizeof_X; n++)
12    sum += calclGetX2Di(MODEL_2D, DEVICE_Q, i, j, n);
13
14  if ((sum==3) || (sum==2 && calclGet2Di(MODEL_2D, DEVICE_Q, i,
      j)==1))
15    calclSet2Di(MODEL_2D, DEVICE_Q, i, j, 1);
16  else
17    calclSet2Di(MODEL_2D, DEVICE_Q, i, j, 0);
18 }
```

LISTING 4.4: The OpenCAL-CL kernel implementing the elementary process of the Game of Life application shown in Listing 4.3.

The device-side kernel implemeting the Life transition function is shown in Listing 4.4. The `calcl2D.h` header is included at line 1, and a numeric handle defined at line 2 to refer the Q substate device-side (this is needed to access the correct buffer in the device global memory - cf. The OpenCAL User Guide on GitHub). The transition rules are implemented as an OpenCL kernel at lines 4-18. In particular, line 5 checks for redundant work-items, while lines 6-7 get the indices corresponding to the integer coordinates of the cell that the kernel is going to process. Similarly, line 8 retrieves the neighborhood size by means of the `calclGetNeighborhoodSize()` function. Eventually, lines 10-17 implement the transition rules by using the `calclGet[X]2Di()` and `calclSet2Di()` functions for reading and updating purposes, respectively.

```
1  #include <OpenCAL/cal2D.h>     //#include <OpenCAL-OMP/cal2D.h>
2  #include <OpenCAL/cal2DIO.h>   //#include <OpenCAL-OMP/cal2D.h>
3  #include <OpenCAL/cal2DRun.h> //#include <OpenCAL-OMP/cal2D.h>
4  #include <stdlib.h>
5
6  struct CALModel2D* life;
7  struct CALSubstate2Di* Q;
8  struct CALRun2D* life_simulation;
9
10 void lifeTransitionFunction(struct CALModel2D* life, int i, int
       j);
11
12 int main() {
13   life = calCADef2D(8, 16, CAL_MOORE_NEIGHBORHOOD_2D,
       CAL_SPACE_TOROIDAL, CAL_NO_OPT);
14   life_simulation = calRunDef2D(life, 1, 1, CAL_UPDATE_IMPLICIT)
       ;
15
16   Q = calAddSubstate2Di(life);
17   calAddElementaryProcess2D(life, lifeTransitionFunction);
18
19   calInitSubstate2Di(life, Q, 0);
20   calInit2Di(life, Q, 0, 2, 1);
21   calInit2Di(life, Q, 1, 0, 1);
22   calInit2Di(life, Q, 1, 2, 1);
23   calInit2Di(life, Q, 2, 1, 1);
24   calInit2Di(life, Q, 2, 2, 1);
25
26   calSaveSubstate2Di(life, Q, "./life_0000.txt");
27   calRun2D(life_simulation);
28   calSaveSubstate2Di(life, Q, "./life_LAST.txt");
29
30   calRunFinalize2D(life_simulation);
31   calFinalize2D(life);
32   return 0;
33 }
```

LISTING 4.5: An OpenCAL implementation of the Conway's Game of Life. To obtain the equivalent OpenCAL-OMP implementation it is sufficient to consider the (currently commented) OpenCAL-OMP header files instead of the OpenCAL ones (cf. lines 1-3).

4.1.5 The Multi-GPU/Multi-Node Implementation of OpenCAL

In its first release [33], described in Section 4.1, the OpenCAL OpenMP- and OpenCL-based components were released, permitting to exploit multi-core CPUs and many-core GPUs, respectively. Parallelism was almost completely made transparent and the *quantization optimization* built in to accelerate the computation in case of topologically connected phenomena (e.g., the

simulation of a bounded fluid flow developing over a wide topographic surface). In fact, quantization permits to avoid to compute the next state of *stationary* cells. Based on user-defined criteria, a set of *active* cells, A, is maintained updated by the system, and the transition function application restricted to it.

The current OpenCAL software architecture can provide four different components at the implementation level, namely OpenCAL (serial reference version), OpenCAL-OMP (OpenMP-based multi-thread implementation), OpenCAL-CL (OpenCL-based component) and OpenCAL-CLM (MPI-based implementation for single and multiple OpenCL instances). In this section, we show the extension of OpenCAL-CL to support single-node/multi-GPU workstations (for a *pure* multi-GPU execution) and to introduce the new OpenCAL-CLM component to support clusters of multi-GPU nodes. Note that, this latter component also permits to run single-GPU instances of OpenCAL-CL within a multi-GPU workstation (for a *hybrid* MPI/OpenCL multi-GPU execution).The library and hardware levels complete the OpenCAL general architecture pointing out the underlying parallel APIs and computing system targets.

The first implementation of the OpenCAL-CL/CLM components was deliberately straightforward and permitted to build the new components on top of the existing OpenCAL-CL one. In particular: A classic data-parallel decomposition scheme was adopted; Halos exchange was scheduled to take place after the execution of each elementary process (i.e., after at least a substate was updated); A different halo was considered for each defined substate (even for those substates that were not affected by the elementary process).

Figure 4.4 shows an example of the data-parallel domain decomposition scheme adopted by OpenCAL-CL/CLM for the case of a two-dimensional domain and a dual-node system with two GPUs per node. Periodic boundary conditions are assumed for the sake of simplicity. The domain is decomposed along the rows and preliminarily assigned to the available nodes. This phase is managed by the OpenCAL-CLM component through MPI over the interconnection network. In turn, each sub-domain is further partitioned among the available GPUs by considering a similar data partitioning scheme, relying on OpenCL (for the pure multi-GPU implementation) or on MPI (for the hybrid MPI/OpenCL solution on the single node). Note that the library permits for non-uniform domain partitioning (e.g., the user can assign more rows to more performing GPU devices) in order to balance the workload among nodes/GPUs with different computational capabilities.

Halos exchange is transparently managed by OpenCAL-CL/CLM, by exploiting the PCI Express bus (in case of halos residing on two or more GPUs on the same node) and/or the interconnection network. Preliminarily, substates's halos are packed in order to reduce communication latency and to maximize the bandwidth. Nevertheless, communication time can sensibly

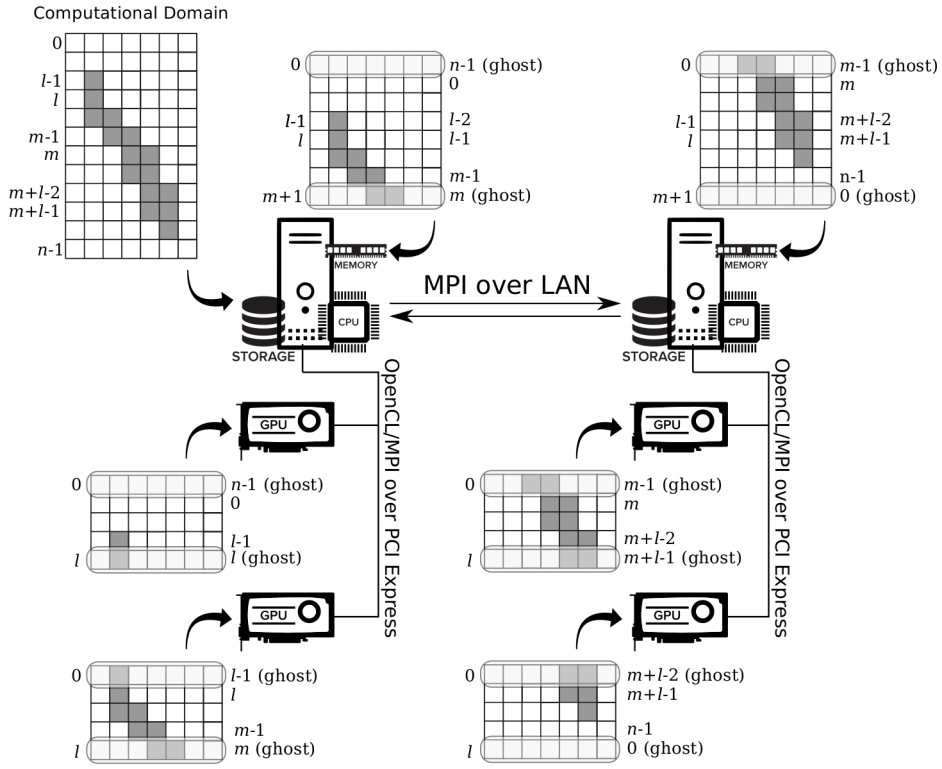


FIGURE 4.4: Domain decomposition adopted by the multi-node/multi-GPU release of OpenCAL. The figure shows the adopted row-major order decomposition of a two-dimensional computational domain for the case of a dual-node cluster, with two GPUs per node. MPI is adopted for halo exchange over the interconnection network. Each sub-domain can be further partitioned among the available GPUs within the node by considering the same partitioning scheme. At node level, either OpenCL (for a *pure* multi-GPU implementation) or MPI (for a *hybrid* OpenCL/MPI multi-GPU implementation) can be used for halo exchange. In this example, a homogeneous computing system is assumed, therefore data is equally partitioned among the available GPUs. However, in real systems, non-uniform partitions can be adopted.

grow according to number and type of involved substates. In summary, for a halo exchange to take place, it is necessary to:

1. Pack and upload boundary data of all defined substates to the CPU (device to host memory transfer);
2. If the GPUs involved in the communications are controlled by different nodes, an extra communication step over the network is performed between the two nodes;
3. Unpack and upload boundary data to the receiving GPU (host to device memory transfer).

Note that, if multi-GPU execution entirely relies on OpenCAL-CL, the host application is involved in the halo exchange among the GPUs, as usually occurs in OpenCL applications. In this case, the process is serialized host-side, as for the OpenCL API specifications, even if non-blocking enqueueing read/write calls are considered. As known, the same host-side serial policy can be found in the OpenGL graphics API and represents one of the reasons that led Khronos Group (i.e. the consortium that defines the OpenCL and OpenGL API specifications, beside others) to propose the new Vulkan compute/graphics unified API, which can exploit the parallelism both host- and device-side. As a consequence, non-optimal exploitation of host-side resources could be achieved by OpenCAL-CL in multi-GPU systems. Nevertheless, the previously mentioned hybrid approach can be alternatively adopted to exploit parallelism also at CPU level, where OpenCAL-CLM can be used to run a single-GPU OpenCAL-CL process for each GPU in the node.

4.2 OPS

OPS (Oxford Parallel library for Structured grid computation), developed by the University of Oxford (UK), is a high-level abstraction framework based on DSL (Domain Specific active Library), targeting computation on multi-block structured meshes. OPS supports both C/C++ and Fortran languages, the most widespread languages on scientific applications. The main aim of OPS is to separate the abstract definition of the computation from its parallel implementations and execution. The OPS abstraction permits to concentrate the effort only on the development of the sequential code, totally delegating the generation of parallel code to the library. To exploit different parallel architectures and to organize execution and data movement, OPS uses a combination of two fundamental techniques: code generation and back-end logic. The code generation (defined also as the source-to-source translator) transforms the sequential OPS application to different parallel implementations and the back-end logic is used to organize the execution and the communication in order to satisfy data dependencies and to improve parallelism, locality or resilience [104]. The OPS workflow is described by Figure 4.5. Initially, the developer implements a structured mesh application using the OPS C/C++ API. Subsequently, the code generator automatically creates the platform specific optimized parallel implementations. The generated code is compiled by the corresponding compiler architecture, such as icc, gcc or nvcc, and linked against the platform specific back-end. At the end of the process, the application is executed on the target architecture(s).

The reached near-optimal OPS performance and the handled wide range of parallel architectures have been achieved by a good level of abstraction. In particular, the OPS abstraction is composed of four principal elements:

1. Blocks: defined by a dimensionality, is a collection of structured grid blocks.

2. Datasets: data on the blocks, identifies by a size.

3. Halos: description of the interface between datasets defined on different blocks.

4. Computations: description of a computation operation to grid points.

The definition of blocks, datasets, halos, and computations are assisted by the OPS API, facilitating the definition of the abstract model. One of the aims of OPS is to improve the long-term maintenance of the code. For this reason, the user programs only a traditional single-threaded sequential application, defining at a high level the data and the computation. The principal OPS API is illustrated by the following functions:

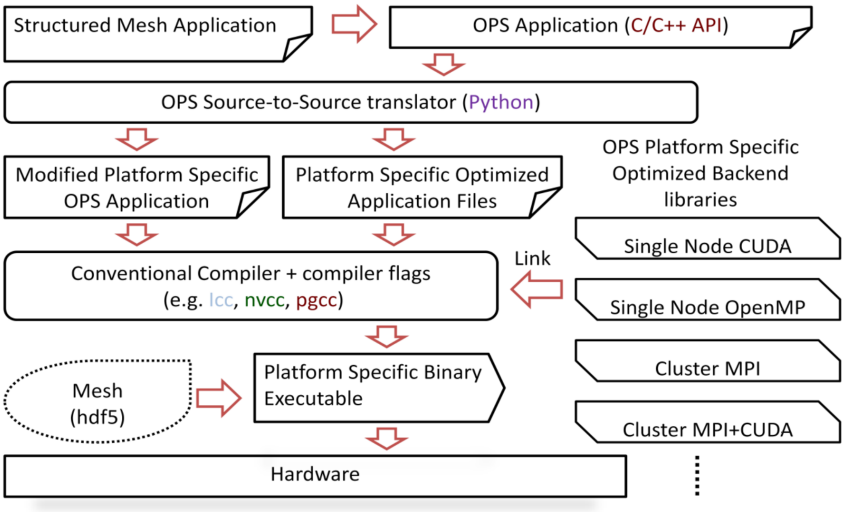


FIGURE 4.5: OPS workflow (from [92]).

- `ops_block ops_decl_block(num_dims, ...)` defines a structured block.
- `ops_dat ops_decl_dat(block, size, ...)` defines a dataset on a given block with a specific size.
- `ops_halo ops_decl_halo(...)` defines a halo between the datasets on different blocks.
- `void ops_par_loop(void (*kernel)(...), block, ndim, range[], arg1, ... ,argN)` defines a parallel loop on a given block, computating the user kernel `kernel` on the grid point.
- `ops_arg ops_arg_dat(dataset, stencil)` defines the data for the computation and the `stencil` which defined the way how to access the data on the grid.
- `ops_arg ops_arg_gbl(data, size, type, access)` defines global data that are independent from the datasets and blocks.

Listing 4.6 shows how to declare a two-dimensional block with a dataset of 2×6 size.

```
1 ops_block block = ops_decl_block(2, "block");
2
3 int halo_neg[] ={0,0};
```

```
int halo_pos[] = {1,0};
int size[] = {2,6};
int base[] = {0,0};

ops_dat dataset = ops_decl_data (block, 1, size, base,halo_pos,
    halo_neg, "double", "dataset");
```

LISTING 4.6: Block and data defition in OPS.

The definition of the computation, through the OPS API, can be implemented by the `ops_par_loop` function that represents an interface where the user defines the grid points to apply the user kernel. Moreover, data is defined by the `ops_arg_dat` function, specifying the type, the access specification, and the stencil points. Listing 4.7 shows a classical sequential loop for the computation on a domain defined by the `dim` array. Listing 4.8 points out the definition of the same computation with OPS API. In particular, the user kernel is identified by the `compute` function, the variable `a` and `b` are the pointers to the dataset, where a one-point and a three-point stencil have been applied, respectively, and the OPS_ACC macros are used to the index offset to access the different point stencil.

```
int dim[4] = {23,45,23,45};
for(int j = dim[2];j < dim[3];j++)
   for(int i = dim[0];i < dim[1];i++)
   {
    a[j][i] = b[j][i]
            + b[j+1][i]
            + b[j][i+1];
   }
```

LISTING 4.7: Example of sequential computation in C++.

```
void compute(double* a, double* b){
    a[OPS_ACC0(0,0)] = b[OPS_ACC1(0,0)]
                     + b[OPS_ACC1(0,1)]
                     + b[OPS_ACC1(1,0)];
}

...

int dim[4] = {23,45,23,45};

ops_par_loop(compute,block,2,dim,
    ops_arg_dat(a,S2D_0,"double",
     OPS_WRITE),
    ops_arg_dat(b,S2D_1,"double",
     OPS_READ));
```

LISTING 4.8: OPS implementation of the example in Listing 4.7.

4.2.1 Code Generation

Once the development of the sequential application has been implemented in OPS, the user can generate the parallel code. Figure 4.6 illustrates the genaration process. The OPS source-to-source translator takes as input the sequential code, then the API calls in the application are parsed and a modified parallel program is generated. The source-to-source code translator is written in Python and only needs to recognize OPS API calls. Moreover, the code generator is human readable which helps with maintenance, longterm support and the introduction of new optimisations. Currently, OPS parallel

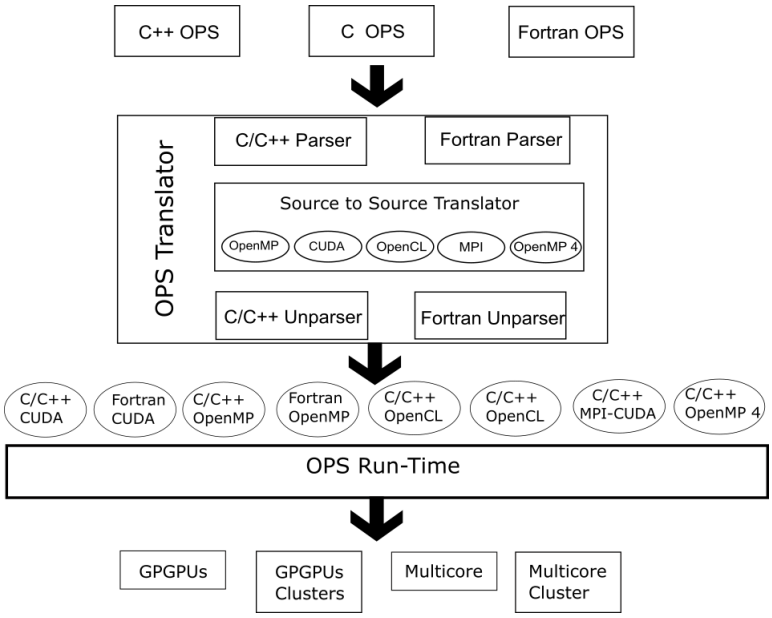


FIGURE 4.6: Genaration process of a OPS application.

code generation supports:

- single threaded vectorized CPUs;
- multi-threaded CPUs/SMPs using OpenMP;
- single GPUs (OpenCL, OpenMP 4, CUDA);
- distributed memory clusters of GPUs(OpenCL, CUDA) using MPI;
- distributed memory clusters of single threaded CPUs using MPI;
- a cluster of multi-threaded CPUs using MPI and OpenMP.

4.2.2 The Cloverleaf Benchmark in OPS

Developed by AWE (Atomic Weapons Establishment) [1], Cloverleaf 4.3.2.1 is a hydrodynamics simulator. It is part of the Mantevo software suite [6], an integrated collection of small software programs (mini-apps) that model the performance of full-scale applications. Many companies have been part of the Mantevo projects such as Intel, IBM, NVIDIA, AMD, Cray along with other universities and laboratories. The scope of the Mantevo project was to use the collection of mini-apps for rapid design-space exploration in the development of the next generation of high-performance computers. In general, scientific applications are composed of a thousand of lines of code,

usually not easy to read, and with only a few lines of code which impacts on the performance. For this reason, the Mantevo mini-app has been split in small fragments, keeping the same performance profile and pointing out the most time consuming computational operations. The provided re-designed applications allowed the high-performance community to develop tools to accelerate and improve the design of high-performance computers [64].

This section illustrates the porting of Cloverleaf in OPS by describing the main development stages. As discussed in 4.2, OPS separates the abstract definition of the computation from its parallel implementations and execution. Thus, the specification of the problem can be split into four distinct parts: (1) structured blocks, (2) data defined on blocks, (3) stencils defining how data is accessed and (4) operations over blocks [93]. The first step defines, by using the `ops_decl_block` API call, the dimensionality of the regular mesh where the computation will be carried out. Since Cloverleaf works on many data arrays (e.g., density, energy, x and y velocities of the fluid), the `ops_decl_dat` API call is used to declare the `density0,energy0, ..., pressure` and `vol ops_dats` variables. Defining the `ops_dat`, the user totally delegates OPS framework for data management. OPS automatically will arrange the optimal memory layout to gain the best performance on the execution hardware. Listing 4.9 illustrates the OPS Cloverleaf code for the definition of the `ops_block` and `ops_dat` variables.

```
1  int dims[2] = {x_cells+5, y_cells+5};
2
3  /* Declare a single structured block */
4  ops_block cl_grid = ops_decl_block(2, dims, "clover grid");
5
6  int d_p[2] = {-2,-2};
7  int d_m[2] = {-2,-2};
8  int size[2] = {x_cells+5, y_cells+5};
9
10 double* dat = NULL;
11
12 /* Declare data on block */
13 ops_dat density0 = ops_decl_dat(cl_grid,1,size,d_m,d_p,dat,"
       double","density0");
14 ops_dat energy0 = ops_decl_dat(cl_grid,1,size,d_m,d_p,dat,"
       double","energy0");
15 ...
16 ...
17 ops_dat pressure = ops_decl_dat(cl_grid,1,size,d_m,d_p,dat,"
       double","pressure");
18 ops_dat volume = ops_decl_dat(cl_grid,1,size,d_m,d_p,dat,"double
       ","volume");
```

LISTING 4.9: Definition of block and data in the OPS implementation of Cloverleaf.

Generally, an intensive computation over a structure grid mesh application can be defined as operations over a specific block. Thus, this corresponds to a loop over the defined block, accessing the data through a stencil, performing the computation and at the end, writing back the results the defined data arrays. For example, examining the `ideal_gas` kernel from the Fortran code, a computation over each point of the structured mesh is executed (Figure 4.11) [92]. The computation loop can be defined in OPS using the `ops_par_loop` API call, specifying the computation by a pointer function, the block ranges and the `ops_arg_dat` data arrays where the computation will be applied. The figure below represents the definition of the `ideal_gas` kernel in OPS.

```
DO k=y_min,y_max
    DO j=x_min,x_max
        v=1.0/density(j,k)
        pressure(j,k)=(1.4-1.0)*density(j,k)*energy(j,k)
        pressurebyenergy=(1.4-1.0)*density(j,k)
        pressurebyvolume=-density(j,k)*pressure(j,k)
        sound_speed_squared=v*v*(pressure(j,k) *pressurebyenergy
    -pressurebyvolume)
        soundspeed(j,k)=SQRT(sound_speed_squared)
    ENDDO
ENDDO
```

Listing 4.10: CloverLeaf ideal_gas kernel as implemented in Fortran.

```
/*ideal_gas user kernel*/
void ideal_gas_kernel( const double *density, const double *
    energy,
double *pressure, double *soundspeed) {

    double sound_speed_squared, v, pressurebyenergy,
    pressurebyvolume;

    v = 1.0/density[OPS_ACC0(0,0)];
    pressure[OPS_ACC2(0,0)] = (1.4-1.0)*density[OPS_ACC0(0,0)]*
    energy[OPS_ACC1(0,0)];
    pressurebyenergy = (1.4-1.0)*density[OPS_ACC0(0,0)];
    pressurebyvolume = -density[OPS_ACC0(0,0)]*pressure[OPS_ACC2
    (0,0)];
    sound_speed_squared = v*v*(pressure[OPS_ACC2(0,0)]*
    pressurebyenergy-pressurebyvolume);
    soundspeed[OPS_ACC3(0,0)] = sqrt(sound_speed_squared);
}

int rangexy_inner[] = {x_min,x_max,y_min,y_max}; //mesh
    execution range
```

```
16
17 /*single point stencil*/
18 int s2D_00[] = {0,0};
19 ops_stencil S2D_00 = ops_decl_stencil( 2, 1, s2D_00, "00"); //
       declare an ops_stencil
20
21 /*example 4 point stencil*/
22 int s2D_4POINT[] = {0,1, 1,0, -1,0, 0,-1};
23 ops_stencil S2D_4POINT = ops_decl_stencil( 2, 1, s2D_4POINT, "
       0,0:1,0:-1,0:0,-1");
24
25 /*example of strided stencil in y*/
26 int str2D_y[] = {0,1};
27 ops_stencil S2D_STRIDE_Y = ops_decl_strided_stencil( 2, 4,
       s2D_00, str2D_y, "s2D_00_stride_y");
28
29 /*ideal_gas loop*/
30 ops_par_loop(ideal_gas_kernel, "ideal_gas_kernel", clover_grid,
       2, rangexy_inner,
31 ops_arg_dat(density0, S2D_00, "double", OPS_READ),
32 ops_arg_dat(energy0, S2D_00, "double", OPS_READ),
33 ops_arg_dat(pressure, S2D_00, "double", OPS_WRITE),
34 ops_arg_dat(soundspeed, S2D_00, "double", OPS_WRITE));
```

Listing 4.11: CloverLeaf ideal_gas kernel as implemented in OPS.

In particular, the macros `OPS_ACC0, OPS_ACC1, OPS_ACC2`, etc. will be resolved to the relevant array index to access the data stored in `density0, energy0, pressure`, etc. arrays. The `OPS_READ` specifies the data as read-only, thus avoiding useless memory update. In this example, only `ops_arg_dat` variables are used, but a similar function `ops_arg_gbl` can be use to indicate global reductions. The code below shows an example of OPS reduction through the defintion of the `ops_arg_reduce` and specifying the `OPS_MIN` parameter to obtain the global minimum and store the result inside the `local_dt` variable.

```
1  ops_par_loop_calc_dt_kernel_min(
2     "calc_dt_kernel_min", clover_grid, 2, rangexy_inner,
3     ops_arg_dat(work_array1, 1, S2D_00, "double", OPS_READ),
4     ops_arg_reduce(local_dt, 1, "double", OPS_MIN));
```

Listing 4.12: An `ops_par_loop` using `ops_arg_reduce`.

4.2.3 Cloverleaf Porting Effort

The main porting effort was performed manually, by extracting the user kernels, keeping the naming conventions of routines and files as similar to

the original as possible [93]. The converted OPS Cloverleaf was composed by 80 `ops_pas_loop` with about 7000 lines of code. After the conversion, the application was validated with a single thread execution, linking the OPS back-end library and using conventional compilers (such as gcc, clang).

As known, the development of a sequential application is simpler to debug and validate respect to a parallel one. Moreover, OPS does not require any parallel knowledge from the user, the only effort resides on the development of the serial application. The user programming cost is rewarded by the benefits of the different automatic parallel generated applications. The current parallel computing models implemented in OPS are: single threaded vectorized CPUs, multi-threaded CPUs/SMPs using OpenMP, NVIDIA GPUs using CUDA and OpenACC, OpenCL devices such as AMD GPUs, the Intel XeonPhi, etc. distributed memory clusters of single threaded CPUs using MPI a cluster of multi-threaded CPUs using MPI and OpenMP and a cluster of GPUs using MPI and CUDA.

The user can generate the parallel code just calling the code generator and passing the source files. The following code 4.13 illustrates the call of the source-to-source translator for the Cloverleaf application:

```
1 ../../../ops_translator/c/ops.py clover_leaf.cpp revert.cpp
    reset_field.cpp ideal_gas.cpp PdV.cpp accelerate.cpp
    advec_cell.cpp accelerate.cpp advec_mom.cpp calc_dt.cpp
    field_summary.cpp flux_calc.cpp viscosity.cpp
    initialise_chunk.cpp generate.cpp update_halo.cpp
```

Listing 4.13: Command for the OPS code generator invocation.

The code is automatically parsed by every parallel computing Python model, creating in the same application location, different folders containing the generated parallel code. The main advantage of OPS DSL library is the drawing up to the future architectures. In fact, a new parallel model can be added to OPS just developing a new parallel computing model.

4.2.4 The CUDA Parallel Model in OPS

As all other parallel programming models, the CUDA code is automatically generated by the source-to-source translator, parsing every `ops_par_loop` and generating a *.cu file containing the host and device code. Through the `ops_dec_dat` function, OPS allocates the data on both CPU and GPU side. Moreover, OPS has been designed to avoid the performance bottleneck produced by the PCIe bus, preventing unnecessary data passing. Furthermore, a specific code is also generated to handle global reduction and global constants [92]. `OPS_reduct_h` and `OPS_reduct_d` are host and device pointers pointing to the memory reserved for the reductions. The kernel

function `ops_reduction<OPS_MIN>(...)` performs the reduction on each thread-block. Eventaully, the global constants on the GPU required special handling, as they were declared on the GPUs constant memory and for access via the GPUs read-only cache. Listing 4.14 shows the CUDA code generated from the OPS source-to-source translator of the Cloverleaf `ideal_gas_kernel` function.

```
1  //user kernel
2  __device__ void ideal_gas_kernel( const double *density, const double *energy,
3  double *pressure, double *soundspeed) {
4
5  double sound_speed_squared, v, pressurebyenergy, pressurebyvolume;
6  v = 1.0 / density[OPS_ACC0(0,0)];
7  pressure[OPS_ACC2(0,0)] = (1.4 - 1.0) * density[OPS_ACC0(0,0)] * energy[OPS_ACC1
       (0,0)];
8  pressurebyenergy = (1.4 - 1.0) * density[OPS_ACC0(0,0)];
9  pressurebyvolume = -*density[OPS_ACC0(0,0)] * pressure[OPS_ACC2(0,0)];
10 sound_speed_squared = v*v*(pressure[OPS_ACC2(0,0)] * pressurebyenergy-
       pressurebyvolume);
11 soundspeed[OPS_ACC3(0,0)] = sqrt(sound_speed_squared);
12 }
13
14
15
16 __global__ void ops_ideal_gas_kernel(
17 const double* __restrict arg0, const double* __restrict arg1,double* __restrict
       arg2, double* __restrict arg3,int size0, int size1 ){
18     int idx_y = blockDim.y * blockIdx.y + threadIdx.y;
19     int idx_x = blockDim.x * blockIdx.x + threadIdx.x;
20     arg0 += idx_x * 1 + idx_y * 1 * xdim0_ideal_gas_kernel;
21     arg1 += idx_x * 1 + idx_y * 1 * xdim1_ideal_gas_kernel;
22     arg2 += idx_x * 1 + idx_y * 1 * xdim2_ideal_gas_kernel;
23     arg3 += idx_x * 1 + idx_y * 1 * xdim3_ideal_gas_kernel;
24
25     if (idx_x < size0 && idx_y < size1) {
26         ideal_gas_kernel(arg0, arg1, arg2, arg3);
27     }
28 }
29
30
31
32 // host stub function
33 void ops_par_loop_ideal_gas_kernel(char const *name, ops_block Block, int dim,
       int* range,
34 ops_arg arg0, ops_arg arg1, ops_arg arg2, ops_arg arg3) {
35     ...
36     ...
37     int x_size = end[0]-start[0];
38     int y_size = end[1]-start[1];
39     ...
40     ...
41     dim3 grid( (x_size-1)/OPS_block_size_x+ 1, (y_size-1)/OPS_block_size_y + 1,
       1);
42     dim3 block(OPS_block_size_x,OPS_block_size_y,1);
43     ...
44     ...
45     ops_H_D_exchanges_cuda(args, 4);
46
47 //call kernel wrapper function, passing in pointers to data
48     ops_ideal_gas_kernel<<<grid, block >>> ( (double *)p_a[0], (double *)p_a[1],
49     (double *)p_a[2], (double *)p_a[3],x_size, y_size);
50
51     ops_set_dirtybit_cuda(args, 4);
52     ...
53     ...
54 }
```

Listing 4.14: Ideal gas kernel in OPS CUDA.

4.2.5 The Developed OpenMP4 Based Version of OPS

In order to develop the OpenMP4 OPS extension, a new OPS source-to-source translator was implemented. As explained in Section 4.2, OPS automatic parallelization is based on computation abstraction. In particular, through the definition of the `ops_par_loop`, the user specifies the data domain and the computation. Since OPS is based on DSL, every OPS version (such as CUDA, OpenCL, OpenACC, etc) interprets the `ops_par_loop` with their own specific paradigms. In fact, to target modern high performance architectures and to reach elevated performances, it is not sufficient to use a single programming language. On the contrary, it is necessary to use a specific language related to the target device. The OpenMP4 aim is to simplify the user programming effort, avoiding to switch to many programming languages and delegating the appropriate parallel translation to the compiler, according to the chosen target device.

During the development of the OpenMP4 version of OPS, two C/C++ compilers were considered, which provide support the newer OpenMP specifications, namely Clang and IBM XL. Among them, Clang was considered in the main development and tuning phases, while a subsequent porting was performed to XL.

As specifically regard the use of Clang, two levels of optimizations were exploited: compiler optimization and source code optimization. Regarding the compiler optimization, the *combined construct directive* was exploited [20]. It represents an optimized alternative to the default OpenMP code generator that allows to obtain more efficient parallel code. The example shown in Listing 4.15 presents a target region that can achieve the best performance by the Clang combined construct, since it is characterized by the following features:

- There is no team master only region, except the distribution of loop blocks to team masters;
- There is no data sharing within a team;
- There are no function calls and, as such, no possible nested OpenMP pragmas.

```
1 #pragma omp target teams distribute parallel for schedule(static
    ,1)
2 for(int i = 0 ; i< n; i++)
3     a[i]=b[i]+c[i];
```

Listing 4.15: Example of combined contruct directive.

Under these conditions, the generated parallel code is highly efficient, as shown in Listing 4.16. The generated code uses a CUDA-like notation. The implementation has no calls to the OpenMP runtime and it has no needs to coordinate threads over an OpenMP region. It has been determined that the combined construct should give the best performance for the NVIDIA target device, by minimizing the number of register occupancy [63].

```
1 for(int idx = threadsIdx.x + blockId.x * blockDim.x; idx < n;
      idx += blockDim.x*gridDim.x)
2  a[idx] = b[idx] + c[idx];
```

LISTING 4.16: Clang CUDA-like generated parallel code of Listing 4.15.

Another important Clang optimization regarded the number of adopted registers per thread, which can have a significant impact on performances [11]. Generally, the compiler attempts to minimize the number of register usage and, at the same time, keep the register spilling and number of instructions to the minimum. During the development of the OpenMP4 version of OPS, the number of registers was identified as one of the responsible for performance degradation. Since the number of registers per thread is closely linked to both the number of variables and instructions in the kernel code, the number of OPS variables and instructions were reduced by delegating the C pre-processor for some preliminary evaluations. For instance, the source code of the `OPS_ACC` function (that computes the index offsets required to access the different stencil points) was replaced by a C macro, by reducing both variables and instructions to be executed at run time.

Listing 4.17 shows the OpenMP4 translation of the `calc_dt_min` Cloverleaf kernel.

```
1 #include "./OpenMP4/clover_leaf_common.h"
2 #include <omp.h>
3 #define OPS_GPU
4
5 extern int xdim0_calc_dt_kernel_min;
6
7 #undef OPS_ACC0
8
9 #define OPS_ACC0(x, y)
                          \
10   (n_x * 1 * 1 + n_y * xdim0_calc_dt_kernel_min * 1 * 1 + x +
                          \
11    xdim0_calc_dt_kernel_min * (y))
12
13 // user function
14
15 void calc_dt_kernel_min_c_wrapper(double *p_a0, double *p_a1,
```

```
    int x_size,
                                    int y_size) {
  double p_a1_0 = p_a1[0];
#ifdef OPS_GPU

#pragma omp target teams distribute parallel for map(tofrom :
    p_a1_0)          \
    reduction(min : p_a1_0)
#endif
  for (int i = 0; i < y_size * x_size; i++) {
      const int id = omp_get_num_threads() *
omp_get_team_num() + omp_get_thread_num();
    const int n_x = id % x_size;
    const int n_y = id / x_size;
    const double *dt_min = p_a0;

    double *dt_min_val = &p_a1_0;

    *dt_min_val = MIN(*dt_min_val, dt_min[OPS_ACC0(0, 0)]);
  }
  p_a1[0] = p_a1_0;
}
#undef OPS_ACC0
```

Listing 4.17: calc_dt_min kernel in OPS OpenMP4

In particular, the `#pragma omp target teams distribute parallel for` exploited the Clang combined construct, by allowing to obtain efficient parallel code. The original double-nested kernel loop was collapsed in a single `for` of `x_size*y_size` iterations and an explicit index calculation was performed. The `map(tofrom:p_a1_0)` clause was used to move the reduction variable `p_a1_0` to and from the device, whereas the `reduction(min:p_a1_0)` clause was considered to compute the minimum value over the given data.

Memory operations have been also optimized, minimizing the communications between host and device, thus reducing the PCIe bottleneck between CPU and GPU. At the begin, data is passed to the device using `#pragma omp target enter data map(to:*)` directives, making the data available to the all following target loops. Only the reduction variables are passed back to the host at the end of each target directive, through the `tofrom:*` option of the `map` clause. It is worth noticing that the user can not choose in which kind of memory the data will transfer (global or local) to the GPU. The OpenMP 4.5 implementation will choose which kind of memory use for the transfer, usually the global one.

Moreover, many tests have been performed in order to find out the best directive configuration to be adopted during the source-to-source translation. As known, the same directive configuration of a OpenMP program may result in different perfomance depending on the considered compiler. In fact, each compiler adopts its own OpenMP back-end. For instance, I no-

ticed that the IBM XL compiler requires an additional and explicit `map(to:)` before every `target` region, whereas Clang does not need it. These different back-end implementations force the programmer to use different clauses and techniques to adapt the code to the specific compiler, increasing effort to gain the best performance with the same code. Therefore, to avoid loss of performances in the OpenMP4 implementation of OPS with respect to the Clang and XL compilers, I designed the source-to-source code generator to adopt the best translation depending on the considered compiler. In order to obtain the best result, the `OPS_COMPILER` environment variable must be set to the name of the compiler to use for the compilation. Listing 4.18 shows a snippet code from the source-to-source translator where is evident the different management reserved to Clang and XL, while Listing 4.19 and 4.20 the generated code of the Cloverleaf `calc_dt_kernel_min` kernel by the Clang and XL compilers, respectively.

```
...

    if compiler == "clang":
     line = "#pragma omp target teams distribute parallel for "
     for n in range (0,nargs):
       if arg_typ[n] == 'ops_arg_gbl' :
         if accs[n] <> OPS_READ:
           line = line + ' map(tofrom :'
           line = line + ' p_a'+str(n)+'_'+str(d)+','
           line = line[:-1]+')'
     for n in range (0,nargs):
       if arg_typ[n] == 'ops_arg_gbl':
         if accs[n] == OPS_MIN:
           for d in range(0,int(dims[n])):
             line = line + ' reduction(min:p_a'+str(n)+'_'+str(d)+')'
         if accs[n] == OPS_MAX:
           for d in range(0,int(dims[n])):
             line = line + ' reduction(max:p_a'+str(n)+'_'+str(d)+')'
         if accs[n] == OPS_INC:
           for d in range(0,int(dims[n])):
             line = line + ' reduction(+:p_a'+str(n)+'_'+str(d)+')'
         if accs[n] == OPS_WRITE: #this may not be correct
           for d in range(0,int(dims[n])):
             line = line + ' reduction(+:p_a'+str(n)+'_'+str(d)+')'

    if compiler == "xl":
     line = '#pragma omp target teams '
     line += 'map(to:'
     for n in range (0,nargs):
       if arg_typ[n] == 'ops_arg_dat':
         line = line + 'p_a'+str(n)+'[0:tot'+str(n)+'],'
       if arg_typ[n] == 'ops_arg_gbl' :
         if accs[n] == OPS_READ and (not dims[n].isdigit() or int(dims[n])>1):
           line = line + 'p_a'+str(n)+'[0:tot'+str(n)+'],'
     line = line[:-1]+')'
     for n in range (0,nargs):
       if arg_typ[n] == 'ops_arg_gbl' :
         if accs[n] <> OPS_READ:
           line = line + ' map(tofrom :'
           line = line + ' p_a'+str(n)+'_'+str(d)+','
           line = line[:-1]+')'
     line = line + 'map(to:'
     for nc in range (0,len(consts)):
       if re.search('[a-zA-Z]', consts[nc]['dim']) or  (int(consts[nc]['dim']) !=
     1) :
         num = str(consts[nc]['dim'])
         line = line + str(consts[nc]['name']).replace('"','')+'[0:'+num+'],'
     line = line[:-1]+')'
     for n in range (0,nargs):
      if arg_typ[n] == 'ops_arg_gbl':
        if accs[n] == OPS_MIN:
          for d in range(0,int(dims[n])):
            line = line + ' reduction(min:p_a'+str(n)+'_'+str(d)+')'
        if accs[n] == OPS_MAX:
```

```
54            for d in range(0,int(dims[n])):
55              line = line + ' reduction(max:p_a'+str(n)+'_'+str(d)+')'
56          if accs[n] == OPS_INC:
57            for d in range(0,int(dims[n])):
58              line = line + ' reduction(+:p_a'+str(n)+'_'+str(d)+')'
59          if accs[n] == OPS_WRITE: #this may not be correct
60            for d in range(0,int(dims[n])):
61              line = line + ' reduction(+:p_a'+str(n)+'_'+str(d)+')'
62      line = line + "\n#pragma omp  distribute parallel for schedule(static, 1) "
63
64      for n in range (0,nargs):
65        if arg_typ[n] == 'ops_arg_gbl':
66          if accs[n] == OPS_MIN:
67            for d in range(0,int(dims[n])):
68              line = line + ' reduction(min:p_a'+str(n)+'_'+str(d)+')'
69          if accs[n] == OPS_MAX:
70            for d in range(0,int(dims[n])):
71              line = line + ' reduction(max:p_a'+str(n)+'_'+str(d)+')'
72          if accs[n] == OPS_INC:
73            for d in range(0,int(dims[n])):
74              line = line + ' reduction(+:p_a'+str(n)+'_'+str(d)+')'
75          if accs[n] == OPS_WRITE: #this may not be correct
76            for d in range(0,int(dims[n])):
77              line = line + ' reduction(+:p_a'+str(n)+'_'+str(d)+')'
78
79  ...
```

LISTING 4.18: Code snipped from the source-to-source translator where the code generator adapts the best pragma configurations for the clang and IBM XL compiler.

```
#include "./OpenMP4/clover_leaf_common.h"
#include <omp.h>
#define OPS_GPU

extern int xdim0_calc_dt_kernel_min;

#undef OPS_ACC0

#define OPS_ACC0(x, y)                \
  (n_x * 1 * 1 + n_y * xdim0_calc_dt_kernel_min * 1 * 1 + x +                  \
   xdim0_calc_dt_kernel_min * (y))

// user function

void calc_dt_kernel_min_c_wrapper(double *p_a0, double *p_a1, int x_size,
                                  int y_size) {
  double p_a1_0 = p_a1[0];
#ifdef OPS_GPU

#pragma omp target teams distribute parallel for map(tofrom : p_a1_0) reduction(min : p_a1_0)
#endif
  for (int i = 0; i < y_size * x_size; i++) {
    const int id = omp_get_num_threads() * omp_get_team_num() + omp_get_thread_num();
    const int n_x = id % x_size;
    const int n_y = id / x_size;
    const double *dt_min = p_a0;

    double *dt_min_val = &p_a1_0;

    *dt_min_val = MIN(*dt_min_val, dt_min[OPS_ACC0(0, 0)]);
  }
  p_a1[0] = p_a1_0;
}
#undef OPS_ACC0
```

LISTING 4.19: Genereted code for Clang compiler.

```
#include "./OpenMP4/clover_leaf_common.h"
#include <omp.h>
#define OPS_GPU

extern int xdim0_calc_dt_kernel_min;

#undef OPS_ACC0

#define OPS_ACC0(x, y)                \
  (n_x * 1 * 1 + n_y * xdim0_calc_dt_kernel_min * 1 * 1 + x +                  \
   xdim0_calc_dt_kernel_min * (y))

// user function

void calc_dt_kernel_min_c_wrapper(double *p_a0, int base0, int tot0,
                                  double *p_a1, int x_size, int y_size) {
  double p_a1_0 = p_a1[0];
#ifdef OPS_GPU

#pragma omp target teams map(to : p_a0[0 : tot0]) map(tofrom : p_a1_0) map(to : states[0 : number_of_states]) reduction(min : p_a1_0)
#pragma omp distribute parallel for schedule(static, 1) reduction(min : p_a1_0)
#endif
  for (int i = 0; i < y_size * x_size; i++) {
    const int id = omp_get_num_threads() * omp_get_team_num() + omp_get_thread_num();
    const int n_x = id % x_size;
    const int n_y = id / x_size;
    const double *dt_min = p_a0 + base0;

    double *dt_min_val = &p_a1_0;

    *dt_min_val = MIN(*dt_min_val, dt_min[OPS_ACC0(0, 0)]);
  }
  p_a1[0] = p_a1_0;
}
#undef OPS_ACC0
```

LISTING 4.20: Genereted code for IBM XL compiler.

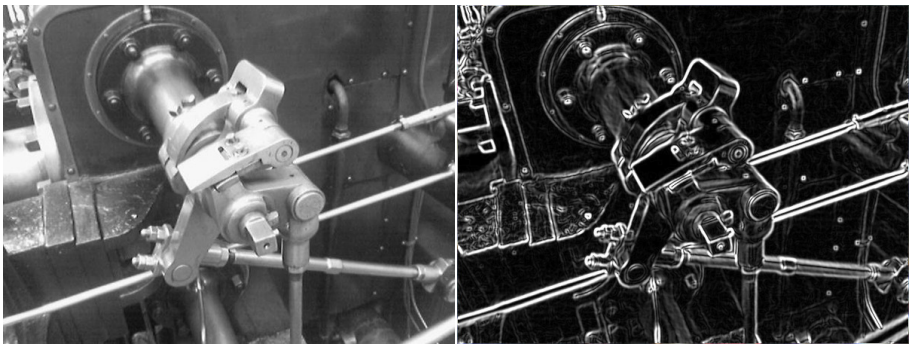
4.3 Applications

In the following, a series of HPC applications are illustrated, which were adopted for benchmarking the considered DSLs in this thesis. The related computational results are discussed in detail in Chapter 5.

4.3.1 OpenCAL Benchmarks

4.3.1.1 The Sobel Edge Detection Filter

The Sobel edge detection filter belongs to the wide family of convolutional graphics filters [110, 61], commonly used to modify the spatial frequency characteristics of an image. In general, the filtering is applied to each point of the domain (i.e., to each pixel of the image) and consists of determining the new value of the point as the weighted summation of its neighbors. For this



(a) (b)

FIGURE 4.7: Example of application of the Sobel edge detection convolutional graphics filter. (a) Original bitmap (`https://en.wikipedia.org/wiki/Sobel_operator`). (b) Result after the application of the Sobel filtering process.

purpose, a (usually small square) matrix K, called *kernel* of the convolution, defining the weights to be used, is considered. Formally, convolution can be expressed by the following formula:

$$f'_{ij} = \sum_{i'=0}^{n-1} \sum_{j'=0}^{m-1} f_{(i+i'-n/2)(j+j'-m/2)} k_{(i'-n/2)(j'-m/2)} \tag{4.1}$$

where f_{ij} and f'_{ij} are the old and new value of the point at coordinate (i, j), respectively, m and n the vertical and horizontal size of the kernel, while k_{ij} is the value of kernel at location (i, j).

In the case of the Sobel edge detection filter, a two-step process is considered, one per each (horizontal and vertical) direction. Accordingly, the following two kernels are adopted:

$$K_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad K_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

and applied separately to produce separate measurements, $G_{x_{ij}}$ and $G_{y_{ij}}$, of the gradient component in each orientation, by applying Equation 4.1. These values are eventually combined to find the absolute magnitude of the gradient at each point as $f'_{ij} = \sqrt{G^2_{x_{ij}} + G^2_{y_{ij}}}$.

Figure 4.7 shows an example of an application of the Sobel edge detection filter to the red channel of a sample image. For the purpose of this work, however, the filter, as implemented in OpenCAL, was applied to the single channel of the gray-scale image in Figure 4.8 (cmp. Section 4.3.1.2).

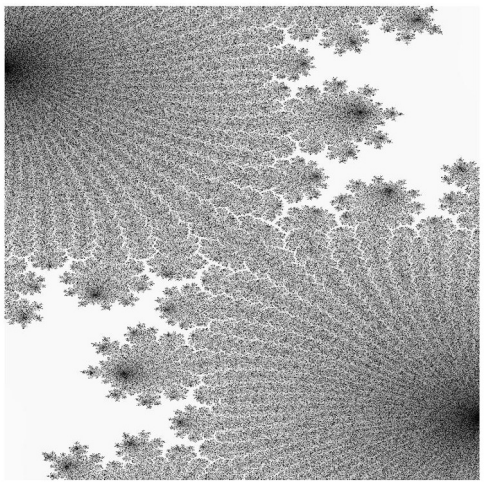


FIGURE 4.8: A Julia fractal set consisting of 15,000 x 15,000 pixels. Divergent pixels are in gray tones, with clearer tones identifying points diverging faster, while convergent pixels are in black.

4.3.1.2 The Julia Set Fractal Generator

Julia sets [24] are examples of fractals generated by mapping the discrete points (or pixels) of a grid $C = \{(x, y) \mid 0 \leq x < S_x, 0 \leq y < S_y\}$ to a rectangular region of the complex plane by applying the rule $z_0 = z_0(x, y) = Re(x) + Im(y)i$, where $i^2 = -1$ and $Re(x)$ and $Re(y)$ are functions of the x and y coordinates, respectively. Subsequently, z_0 is iteratively updated by considering a recurrence formula of the kind $z_{n+1} = z_n^2 + c$, where c is a complex parameter. By changing c, different Julia sets are obtained. The iterative process ends when the z module becomes greater than a given threshold $T \in \mathbb{R}$ (in this case the point (x, y) is said to be divergent and does not belong to the set), or after a predefined number of iterations N (in this case the point is said to be convergent and belongs to the set). Formally, the Julia set of the grid C can be defined as:

$$J(C) = \{z \in C \ : \ |z_n| < T, \ \forall n \leq N\}$$

Figure 4.8 shows the fractal considered in this work. It was obtained by considering $c = -0.391 + -0.587i$, and by mapping the discrete points of a

$15,000 \times 15,000$ grid (i.e. $S_x = S_y = 15,000$) as:

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

where the *zoom factor* K was set to 3. Eventually, the theshold T was set to the value 10^3 and $N = 10^4$ iterations were considered to evaluate the process of convergence.

4.3.1.3 The SciddicaT Landslide Simulation Model

SciddicaT is a classic example of Extended Cellular Automata model for simulating fluid-flows [15]. The model is extremely simple and fast. Nevertheless, it demonstrated to be able to properly simulate non inertial real phenomena like landslides on complex topographic surfaces. In brief, six information layers account for main system's characteristics, while three elementary processes determine its dynamical evolution. In the XCA formalism, information layers are expressed in terms of substates (i.e., as double buffered matrices, used alternatively for read and write access and swapped after the application of each elementary process). Specifically, they are: Q_z, which stores the information about the cell's altitude, Q_h representing the fluid thickness, and Q_o^4 being the outflows from the central cell to the four neighbors (belonging to the von Neumann neighborhood). The system's evolution is thus obtained by applying the following elementary processes simultaneously to each cell of the computational domain:

- $\sigma_1 : (Q_z \times Q_h)^5 \times p_\epsilon \times p_r \rightarrow Q_o^4$ computes outflows from the central cell to the four neighbors by applying the *minimization algorithm of the differences* [47]. A preliminary control avoids the computation of negligible outflows (i.e., if the fluid thickness is smaller than or equal to a predefined threshold p_ϵ). If this is not the case, the outflows are given by $q_o(0, m) = f(0, m) \cdot p_r$ $(m = 0, \ldots, 3)$, were $f(0, m)$ are the outgoing flows towards the 4 adjacent cells, as evaluated by the minimization algorithm, and $p_r \in]0, 1]$ a relaxation rate factor considered to damp outflows in order to obtain a smoother convergence to the system's global equilibrium. The Q_o^4 substates are updated accordingly with the values of the computed outflows.

- $\sigma_2 : Q_h \times (Q_o^4)^5 \rightarrow Q_h$ evaluates the new value of fluid thickness inside the cell by considering mass exchange in the cell's 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 the t and $t+1$ computational steps, respectively, while $q_o(m, 0)$ represents the inflow from the $n =$

$(m+1)^{th}$ neighbor. The Q_h substate is updated accordingly to account for the mass balance within the cell.

- $\sigma_3 : Q_o^4 \rightarrow Q_o^4$ resets the outflow substates (i.e., sets them to zero) for the next computational step.

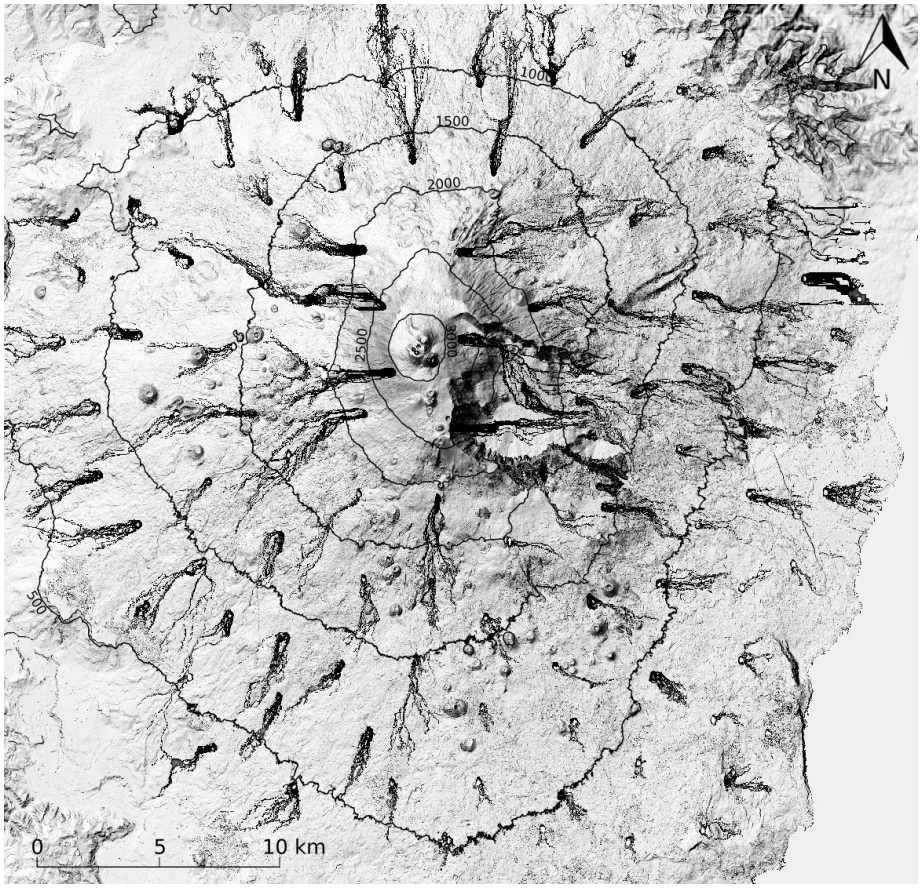


FIGURE 4.9: SciddicaT simulation of 100 flows over a 3,593 rows per 3,730 columns wide surface with square cells of 10 m side.

According to [15], the model parameters p_ϵ and p_r were set to the values 0.001 and 0.5, respectively, and 100 flows were simulated for a total of 4,000 computational steps over a wide surface represented by a $3,593 \times 3,730$ DEM (Digital Elevation Model), with square cells of 10 m side. Outcomes are shown in Figure 4.9.

4.3.2 OPS Benchmarks

4.3.2.1 The Cloverleaf Hydrodynamics Model

As anticipated, CloverLeaf is part of the R&D Top 100 award winning Mantevo software suite [6]. The Mantevo hydrodynamics application investigate the behavior and responses of materials when applied with varying levels of stress. In particular, CloverLeaf uses a Lagrangian-Eulerian scheme to solve Euler's equations of compressible fluid dynamics in two spatial dimensions. A system of three partial differential equations permits the conservation of mass, energy and momentum. A fourth auxiliary equation of state is used to close the system; CloverLeaf uses the ideal gas equation of state to achieve this [82]. The equations are solved on a staggered grid (Figure 4.10 (a)) in which each cell centre stores three quantities: energy, density and pressure; and each node stores a velocity vector.

To solve the equations with second-order accuracy, an explicit Finite-volume method has been used. The system is hyperbolic, meaning that the equations can be solved using explicit numerical methods, without the need to invert a matrix. Currently only single material cells are simulated by CloverLeaf.

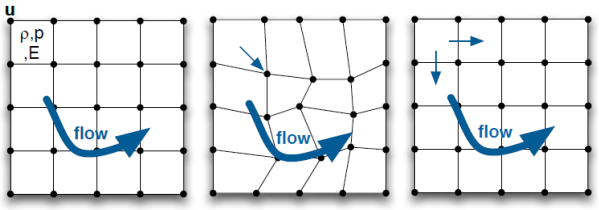


FIGURE 4.10: (a) Staggered grid. (b) Langrangian Step. (c) Advective remap. Figure from [83]

The solution is advanced forward in time repeatedly until the desired end time is reached. Unlike the computational grid, the solution in time is not staggered, with both the vertex and cell data being advanced to the same point in time by the end of each computational step. One iteration, or timestep, of CloverLeaf proceeds as follows [82]:

1. A Lagrangian step advances the solution in time using a predictor-corrector scheme, with the cells becoming distorted as the vertices move due to the fluid flow;

2. An advection step restores the cells to their original positions and calculates the amount of material which passed through each cell face.

This procedure is accomplished using two sweeps, one in the horizontal dimension and the other in the vertical, using the van Leer advection [122]. The direction of the initial sweep in each step alternates in order to preserve second order accuracy. The computational mesh is spatially decomposed into rectangular mesh chunks and distributed across processes within the application, in a manner which attempts to minimise the communication surface area between processes. Whilst simultaneously attempting to assign a similar number of cells to each process to balance computational load.

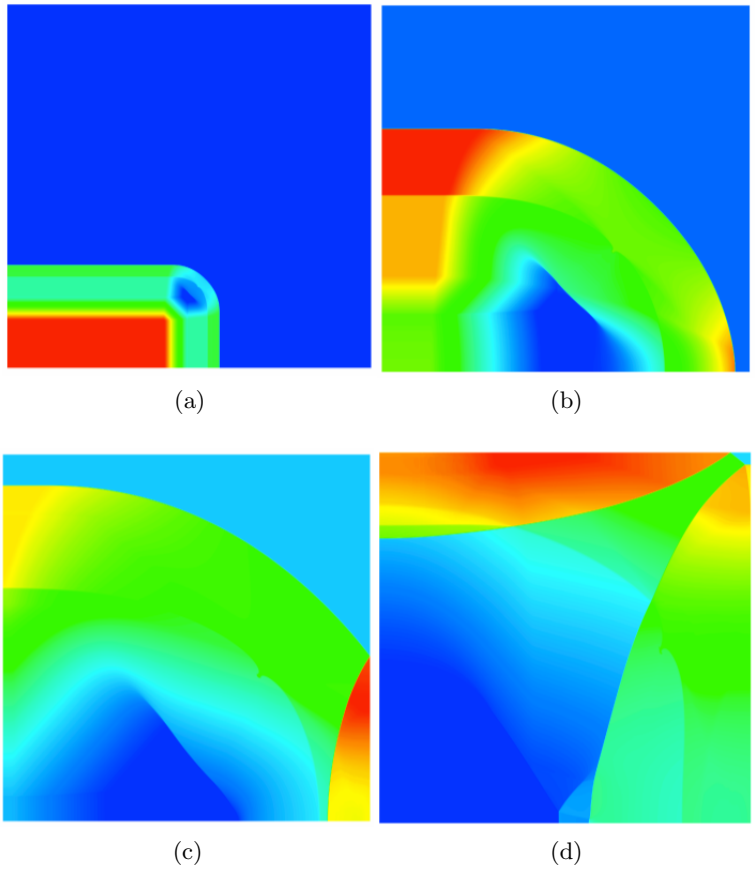


FIGURE 4.11: Graphical view of Cloverleaf application. [2]

Data exchanges occur multiple times during each timestep, between logically neighbouring processes within the decomposition. A global reduction operation is required by the algorithm during the calculation of the minimum stable timestep, which is calculated once per iteration.

4.3.2.2 Tealeaf

TeaLeaf is a mini-app that solves the linear heat conduction equation on a spatially decomposed regularly grid using a 5 point stencil with implicit solvers. The temperatures are stored at the cell centres while a conduction coefficient is calculated that is equal to the cell centred density or the reciprocal of the density. An implicit method has been used to solve the problem due to the severe timestep limitations imposed by the stability criteria of an explicit solution for a parabolic partial differential equation. As the approch of mantevo projects, the computation has been broken down into "kernels", low level building blocks with minimal complexity. Each kernel loops over the entire grid and updates the relevant mesh variables. Memory is sacrified in order to increase performance, and any updates to variables that would introduce dependencies between loop iterations are written into copies of the mesh [7].

Chapter 5

Computational Results and Discussion

This chapter presents the computational results achieved by the OpenCAL and OPS releases. For this purpose, I considered the applications described in Chapter 4, specifically a Sobel graphics edge detection filter, a Jiulia set fractal generator and the SciddicaT slow-moving fluid flow simulation model for OpenCAL. Instead, Tealeaf and Cloverleaf in two- and three-dimension were considered for the OPS library. In particular, a single GPU/multi GPU and single-Node/multi-GPU versions have been considered for OpenCAL tests, using three different GPUs, such as NVIDIA K40, GTX 980 and Titan Xp. Eventually, only a single GPU execution has been considered in OPS experiments, exploiting a NVIDIA K40 and P100 GPUs and considering different compilers such as Clang, IBM XL, OpenACC and nvcc.

5.1 OpenCAL Computational Results

The JPDM-SS dual node test cluster was adopted for evaluating the performances of the OpenCAL-CL/CLM components. The cluster nodes, namely JPDM-1 and JPDM-2, were interconnected by a Cisco Catalyst 3750 Series switch via standard Gigabit Ethernet (with a theoretical bandwidth of 820 Gbit/s). JPDM-1 was equipped with two Intel E5-2650 Xeon CPUs, two GTX 980 and one Tesla K40 Nvidia GPUs, while JPDM-2 with two E5440 Xeon processors and two Titan Xp Nvidia GPUs. In particular, the GTX 980 and the Titan Xp are game oriented graphics devices, while the Tesla K40 is a compute-dedicated solution. This is confirmed by the theoretical peak performances in single (fp32) and double (fp64) precision floating-point operations. Specifically, the GTX 980 reaches 4.98 TFLOPS in fp32, which drops to 0.156 TFLOPS in fp64. Similarly, the Titan Xp reaches the theoretical performance of 12.1 TFLOPS in single precision, value that drops to only 0.378 TFLOPS in double precision. On the contrary, theoretical

better fp64 performance characterizes the Tesla K40 solution, which provides 4.3 TFLOPS in fp32 and 1.43 TFLOPS in double precision. Other specifications of the above many-core devices are: the GTX 980 (Maxwell architecture) has 2048 CUDA cores, 4 GB global memory and 112 GB/s theoretical bandwidth communication for double precision data between CPU and GPU, the Titan Xp (Pascal Architecture) device has 3840 cores, 12 GB global memory and 273.85 GB/s bandwidth, while the Tesla K40 has 2880 cores, 12 GB global memory and 144 GB/s bandwidth.

A total of ten simulations were executed for each of the considered benchmarks (cf. Section 4.3) by considering different hardware configurations, ranging from single-node/single-GPU to multi-node/multi-GPU systems. Integer values were adopted for the Sobel benchmark, while double-precision floating point values were considered for the others, and speed-up evaluated with respect to the elapsed times of the corresponding OpenCAL-based serial implementation (as executed on the - more performing - Intel E5-2650 Xeon processor), by taking into account the minimum recorded times.

The compute/memory bound nature of the benchmarks was preliminarily investigated. For this purpose, the algorithmic instruction/byte ratio r was considered and compared with the so-called device-dependent balanced instruction/byte ratio r_d, this latter typically representing the number of fp32 operations per byte issued in order to obtain peak compute and bandwidth performance [123]. More specifically, if I (expressed in GInst/s) and M (expressed in GB/s) are the peak instruction and memory throughput, respectively, the balanced instruction/byte ratio is defined as $r_d = I/M$.

Accordingly, by considering that the Tesla K40 has a total of 2880 cores, each one with a frequency of 0.745 GHz, and by assuming that a fp32 operation is completed in 3 clock cycles, the theoretical fp32 instruction throughput is $I = 0.745 * 2880/3 = 715.2$ GInstr/s. By also considering that its fp32 theoretical bandwidth is $M = 288$ GB/s, the fp32 theoretical balanced ratio for the Tesla K40 is $r_{K40}^{(fp32)} = 715.2/288 = 2.83$. This value is reduced by one third in case of fp64 operations, since the number of double precision units of the K40 is 960 (i.e., one third of the total amount of single precision units which are 2880) [125]. The balanced ratio in case of fp64 is therefore $r_{K40}^{(fp64)} = 0.94$. As for the Tesla K40, the same evaluations allowed to evaluate the theoretical balanced ratios: $r_{GTX}^{(fp32)} = 3.43$, $r_{GTX}^{(fp64)} = 0.1$ (since the fp64 units of the GTX 980 GPU are 1/32 of the fp32 ones). Moreover, regarding the Titan Xp, we obtained: $r_{Titan}^{(fp32)} = 3.7$, $r_{Titan}^{(fp64)} = 0.12$ (since fp64 units are 1/32 of the fp32 ones even for the Titan Xp). Eventually, we assume $r_*^{(int)} = r_*^{(fp32)}$, where the * character is used as wildcard to indicate any GPU here considered. The above evaluations are summarized in Table 5.1.

In order to evaluate the compute/memory bound nature of the developed benchmarks, we assumed that an algorithm is considered as compute bound

for a given GPU if its instruction/byte ratio r is greater than the device balanced ratio r_d, while it is memory bound in the other case.

r_d	fp32	fp64	int
GTX 980	3.43	0.1	3.43
Titan Xp	3.7	0.12	3.7
Tesla K40	2.83	0.94	2.83

TABLE 5.1: Balanced instruction/byte ratio for the GPUs adopted in this work.

By considering the $N = 10^4$ iterations adopted for evaluating the convergence condition within the Julia elementary process, with a total of 4 calls to math instructions per iteration and only two memory accesses, the instruction/byte ratio for the Julia benchmark was evaluated to be $r \approx 13.333 \cdot 10^3 >> r_*^{(fp64)}$. The different orders of magnitude of the instruction/byte ratio with respect to the balanced one of all the considered devices suggest that the benchmark can significantly take advantage of a compute dedicated device.

In the case of Sobel, by considering that the only defined elementary process executes 36 integer operations against a total of 21 memory accesses (substate updating included), the instruction/byte ratio was evaluated to be $r \approx 1.71 < r_*^{(int)}$, pointing out a similar memory bound nature of the benchmark with respect to the considered devices. Accordingly, computational devices with more recent architectures are expected to perform better on this benchmark.

Eventually, SciddicaT exposed both kinds of bounds within its elementary processes. In particular, σ_1 instruction/byte ratio was evaluated to be about $r = 1.4$. This value was obtained by considering that the elementary process requires in average 21 fp64 operations and 15 memory accesses. However, since a preliminary (optimization) control is applied, its application is skipped for those cells with a negligible amount of fluid (cf. Section 4.3), being fully applied to about only the 3.9% of the domain cells during the whole simulation (cf. also [33]). On the contrary, σ_2 and σ_3 were characterized by a instruction/byte ratio of 0.8 and 0.0, respectively (i.e., no fp operations are performed by σ_3), thus resulting compute bound. By considering also the 36 memory accesses needed for updating purposes at the end of each elementary process, the overall SciddicaT instruction/memory ratio was evaluated to be $r \approx 0.17$. In this case, the algorithm resulted compute bound for the GTX 980 and the Titan Xp GPUs in a slight measure (since $r \approx r_{GTX}^{(fp64)} \approx r_{Titan}^{(fp64)}$), while more memory bound for the Tesla K40. Accordingly, devices more similar in terms of fp64 balanced ratio, like the adopted

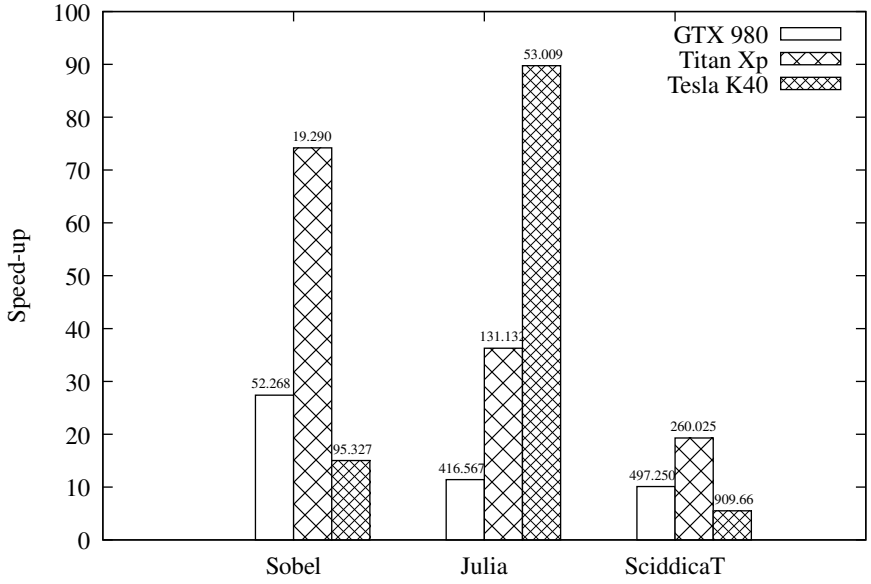


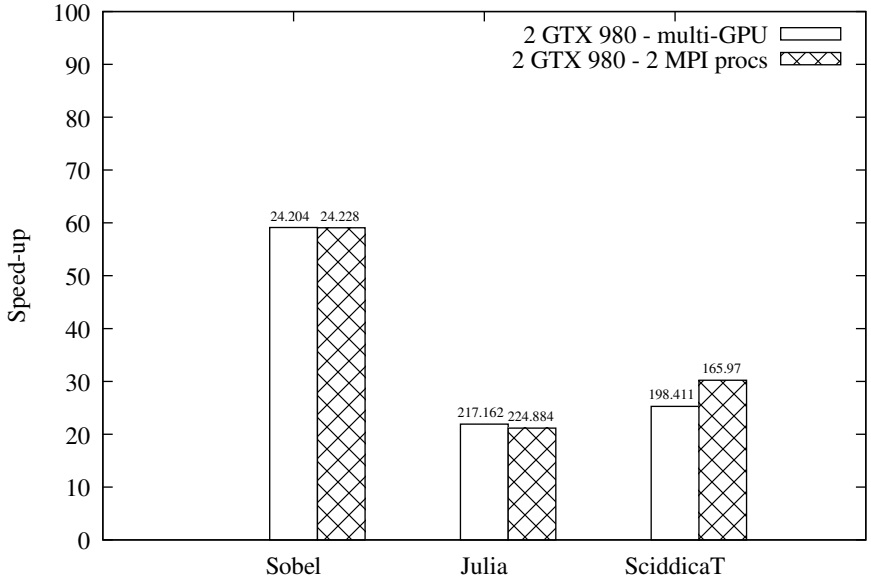
FIGURE 5.1: Speed-ups achieved by the different OpenCAL-CL single-GPU executions of the Sobel, Julia and SciddicaT benchmarks. Elapsed times in seconds are also shown on top of each speed-up bar. The sequential reference times were taken on the JPDM-1 workstation and are 1,431 s, 4,758 s and 5,015 s for the Sobel, Julia and SciddicaT, respectively. The adopted GPUs are an Nvidia GTX 980, Nvidia Titan Xp and Nvidia Tesla K40.

game-oriented GPUs, should run SciddicaT at almost their best, while the compute dedicated Tesla k40 device should perform sub-optimally.

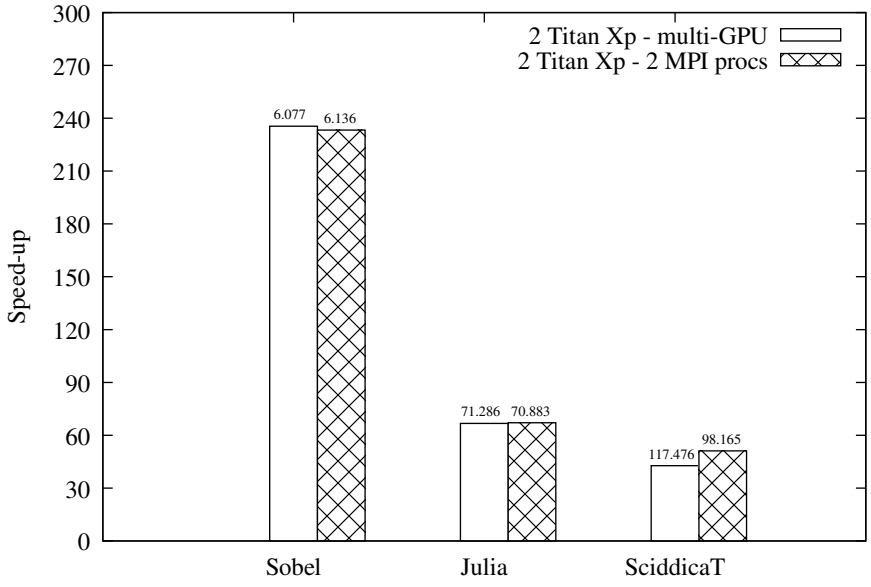
In the remaining part of this Section, computational results are presented, with reference to the adopted computing systems.

5.1.1 Single-Node/Single-GPU and Single-Node/Multi-GPU Computational Results

The speed-up achieved by the three developed benchmarks on the three available GPUs are shown in Figure 5.1. The figure reports also the elapsed times in seconds on top of each speed-up bar. The Titan Xp outperformed the GTX 980 in all tests, while the compute dedicated Tesla K40 performed better on the Julia benchmark in absolute terms. Compared with its sequential implementation (as executed on JPDM-1, which is equipped with the more performing CPUs), Julia ran about 89 times faster on the Tesla K40, 11 times faster on the GTX 980 and 36 times faster on the Titan Xp. On the contrary, the other benchmarks performed better on the game-oriented GPUs, as expected for the above consideration.



(a)



(b)

FIGURE 5.2: Speed-ups achieved by the different single-node/multi-GPU executions of the Sobel, Julia and SciddicaT benchmarks. Elapsed times in seconds are also shown on top of each speed-up bar. The sequential reference times were taken on the JPDM-1 workstation and are 1,431 s, 4,758 s and 5,015 s for the Sobel, Julia and SciddicaT, respectively. An equal partitioning of the domain for each benchmark was considered. (a) Results referred to the JPDM-1 node consisting of two GTX 980 GPUs. (b) Results referred to the JPDM-2 node consisting of two Titan Xp GPUs.

The second series of tests regarded the single-node/dual-GPU execution on the JPDM-1 and JPDM-2 workstations. The Tesla K40 GPU was not considered in these tests to maintain the dual-GPU systems perfectly balanced. Accordingly, an equal partitioning of the computational domains was adopted, by assigning an equal number of rows to each GPU. Moreover, two different types of execution were considered: a first one based on a *pure* multi-GPU approach, where halos were directly exchanged by means of OpenCL read/write buffer enqueueing operations on the PCI Express bus, a second by a *hybrid* OpenCL/MPI approach, where two MPI processes were considered for running two single-GPU OpenCAL-CL instances and for halos management (which however occurred still on the PCI Express bus). It is worth to note that both the pure and hybrid execution policies were obtained by simply changing few parameters in the (so-called) OpenCAL *cluster* configuration file, with no changes required at source code level. Figures 5.2-a and 5.2-b show the results achieved on aforementioned workstations. The figures report also the elapsed times in seconds on top of each speed-up bar. Superlinear effects were here observed for the Sobel and SciddicaT benchmarks, probably due to typical cache issues effects. In fact, the algorithms require the access to neighboring cells' data, that can be prefetched in cache. This is implicitly confirmed by the fact that the Julia benchmark, which does not require the access to data besides the one referring the central cell, did not show superlinear speed-up. In absolute terms, the dual Titan Xp system (JPDM-2) performed considerably better than the dual GTX 980 one, with the (integer-based) Sobel application running about four times faster. The other (double precision) benchmarks resulted roughly two times faster on JPDM-2 with respect to JPDM-1.

It is worth to note that, independently from the workstation considered, both types of execution produced similar results for the Sobel and Julia models, while SciddicaT evidences an about 17% discrepancy in favor of the OpenCL/MPI hybrid execution policy. The reason beneath these results can be imputed to the host-side serial nature of OpenCL, as already discussed in Section 4.1.5. The problem did not emerge for the Julia benchmark since it did not require communications to take place during the computation, while it was of negligible entity for Sobel, where a low amount of data and few messages were needed for each computational step. Eventually, the aforementioned discrepancy resulted with major evidence for the SciddicaT model since the halos have bigger size and are also exchanged more frequently (three times per step). Specifically, each Sobel communication required a total of 120 KB (only one integer substate is defined in Sobel), while 358.08 KB were necessary for SciddicaT (six double-precision subtates are defined in SciddicaT). Moreover, Sobel halo management required 8 read/write operations, versus the 48 ones required by SciddicaT. These messages were managed by the host application serially in the pure OpenCL-based execution policy (even if nonblocking OpenCL calls were

used), while they were executed in parallel when the hybrid OpenCL/MPI policy was considered. On the contrary, in case of hybrid execution, the two MPI processes performed the halo management in parallel. In this manner, the single process was devoted to only 24 halo exchanges for the case of SciddicaT, by reducing the overall time spent in this process.

5.1.2 Multi-Node Multi-GPU Computational Results

Eventually, the dual-node JPDM-SS cluster was adopted for the final speed-up tests with a total of four GPUs. Two execution policies were considered, as for the single-node case. In the first one, MPI was considered for messages that take place only between nodes, while OpenCL execution at multi-GPU level, including halo exchange between the GPUs in the node. In the second, MPI was considered for both for messages between the nodes and for halo exchange at node level. Eventually, different domain partitionings where considered between nodes to account for their different computational capabilities, while the same amount of rows were assigned to the GPUs within each node.

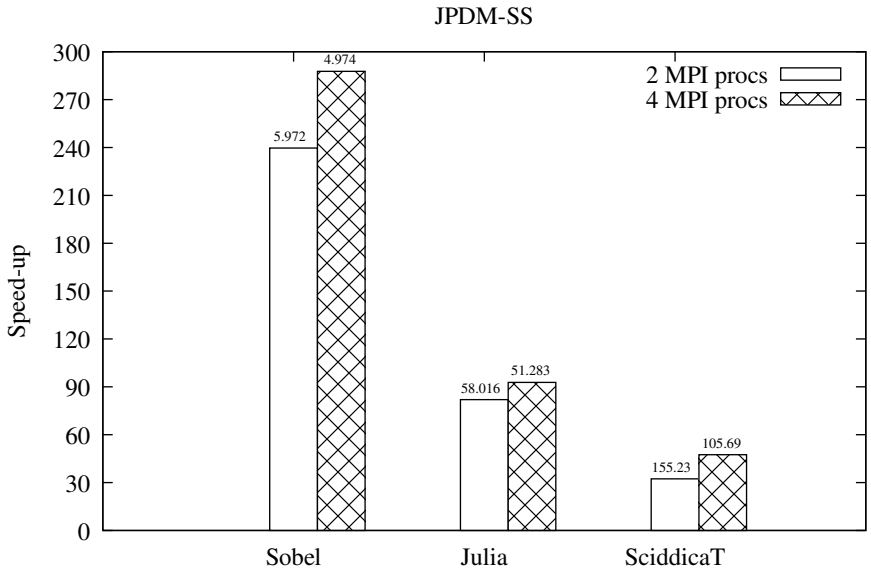


FIGURE 5.3: Speed-ups achieved by the different multi-node/multi-GPU executions of the Sobel, Julia and SciddicaT benchmarks. Elapsed times in seconds are also shown on top of each speed-up bar. The sequential reference times were taken on the JPDM-1 workstation and are 1,431 s, 4,758 s and 5,015 s for the Sobel, Julia and SciddicaT, respectively. Non uniform domain partitionings was adopted in order to balace the workload between the nodes. Optimal partitioning corresponded to maximun network bandwidth (cf. Figure 5.4).

Figure 5.3 shows the achieved speed-ups, together with the corresponding execution times.The best results were achieved in correspondence of the data partitioning that permitted to maximize the network bandwidth, as expected. As an example, Figure 5.4 shows the bandwidth measured during the SciddicaT execution on the different data partitions considered. In relative terms, JPDM-SS exhibited better results with respect to its single nodes for the Sobel and Julia benchmarks. The greater improvement was registered for Sobel, which evidenced a speed-up of about 1.4 with respect to JPDM-2 (i.e., the fastest node). Only SciddicaT showed a slight slow-down, nevertheless performing 1.57 times better than JPDM-1. In this case, the analysis of the obtained result is more difficult, due to the presence of the interconnection network. The registered peak bandwidth of 41.3 MiB/s (cf. 5.4) does not saturate the Gigabit channel and therefore did not represent a bottleneck in the SciddicaT execution. Nevertheless, as already asserted in [33], the SciddicaT behavior on JPDM-SS can be attributed to the memory-bound nature of the algorithm which requires a fast interconnection network to minimize GPUs idle time.

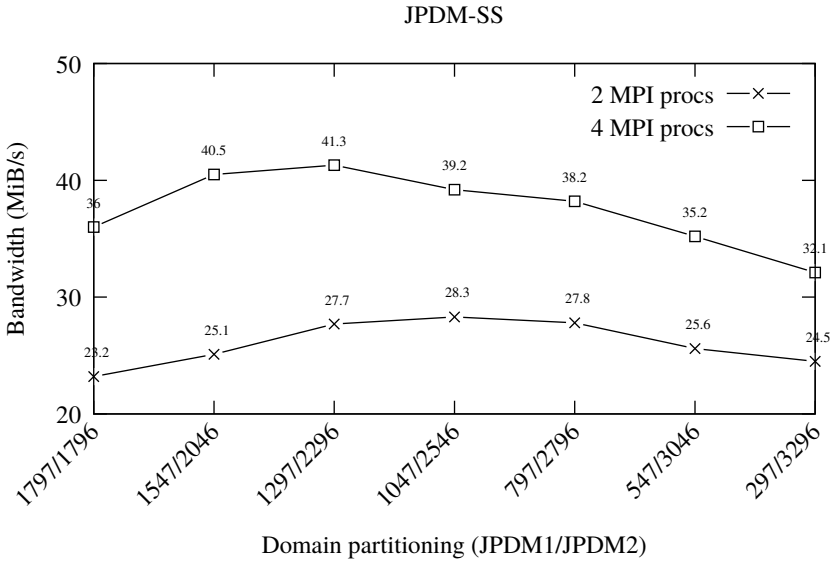


FIGURE 5.4: Bandwidth registered in correspondance of different domain partitioning for the execution of the SciddicaT benchmark on the JPDM-SS dual-node cluster.

5.2 OPS Computational Results

This section shows the performance achieved by the new OpenMP4 based version of OPS that I have designed and implemented. For this purpose, the Cloverleaf (described in Section 4.3.2.1) and Tealeaf (described Section 4.3.2.2) applications were considered.

The Clang and IBM XL compilers have been taken into account to buid both the library and the considered examples. Moreover, results have been compared with those obtained by considering the OPS OpenACC and CUDA versions. All tests have been performed on NVIDIA P100 and K40 GPUs. Table 5.2 illustrates the specifications of the adopted GPUs. Eventually, the compilers flags reported in Table 5.3 were considered.

Products	**Tesla K40**	**Tesla P100**
GPU / Form Factor	Kepler	Pascal
SMs	15	56
TPCs	15	28
FP32 CUDA Cores / SM	192	64
FP32 CUDA Cores / GPU	2880	3584
FP64 CUDA Cores / SM	64	32
FP64 CUDA Cores / GPU	960	1792
Base Clock	745 MHz	1126 MHz
GPU Boost Clock	810/875 MHz	1303 MHz
FP32 GFLOPs	5040	9340
FP64 GFLOP	1680	4670
Texture Units	240	224
Memory Interface	384-bit GDDR5	4096-bit HBM2
Memory Bandwidth	288 GB/s	732 GB/s
Memory Size	Up to 12 GB	16 GB
L2 Cache Size	1536 KB	4096 KB
Register File Size / SM	256 KB	256 KB
Register File Size / GPU	3840 KB	14336 KB
TDP	235 Watts	250 Watts
Transistors	7.1 billion	15.3 billion
GPU Die Size	551 mm	610 mm
Manufacturing Process	28-nm	16-nm

TABLE 5.2: Specifications of the Tesla K40 and P100 GPUs.

Compiler	Version	Flags
PGI	17.1	-O3 -ta=nvidia,cc35 -Mcuda=fastmath -Minline=reshape (-acc for OpenACC)
XL	13.1.6	-O3 -qsmp=omp -qthreaded -qmaxmem=-1 -qoffload -Xptxas -v
clang for OpenMP4	4.0	-O3 -ffast-math -fopenmp=libomp -Rpass-analysis -fopenmp-targets=nvptx64-nvidia-cuda -fopenmp-nonaliased-maps -ffp-contract=fast
nvcc	8.5	-O3 -gencode arch=compute 35,code=sm 35 use fast math

TABLE 5.3: Compiler, version and flags adopted in the performed tests.

5.2.1 Cloverleaf Results

The Cloverleaf domain was discretized by considering a regular grid of 960 rows and 960 columns, for a total of 921,600 cells, whereas the total number of iterations was set to 87. Moreover, the nvcc, Clang, XL and OpenACC compilers were considered for the execution on the Tesla K40, while only nvcc, clang and OpenACC have been taken into account for the P100, since this GPU was installed on a non IBM based workstation (the IBM Power architecture is needed by XL).

The elapsed times of the performed tests on the K40 and P100 GPUs for the Cloverleaf_2D and Cloverleaf_3D applications are shown in Figures 5.5 and 5.6, respectively. Here, as expected, nvcc always outperformed the other compilers. Nevertheless, Clang achieved almost optimal performance.

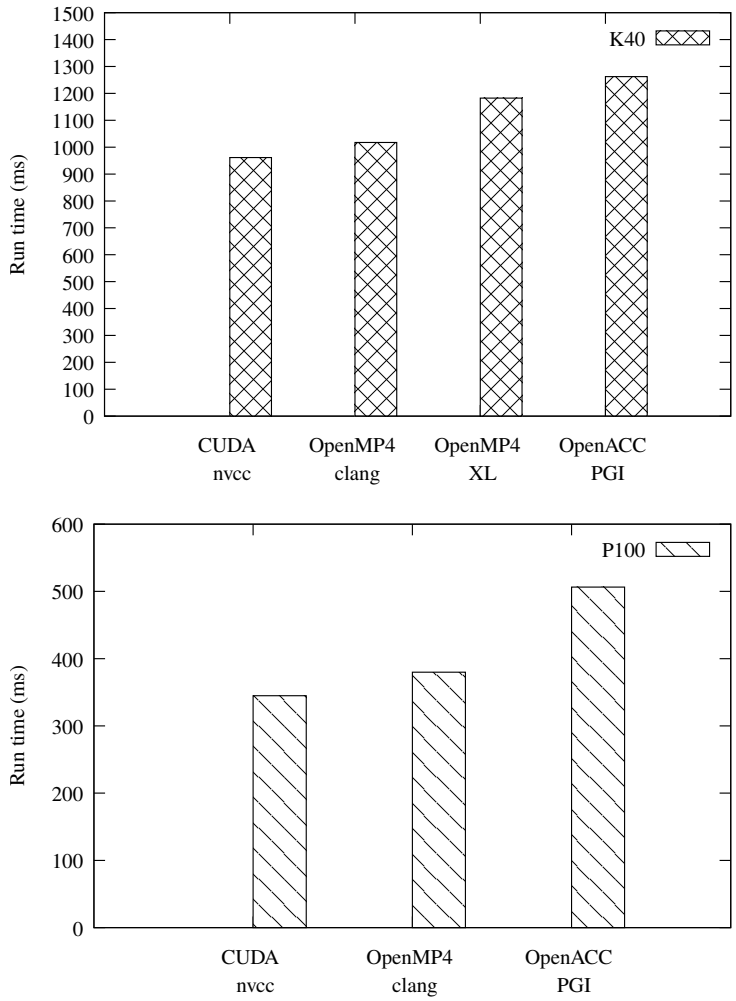


FIGURE 5.5: Cloverleaf 2_D measured run times of versions on the K40 and P100 GPUs.

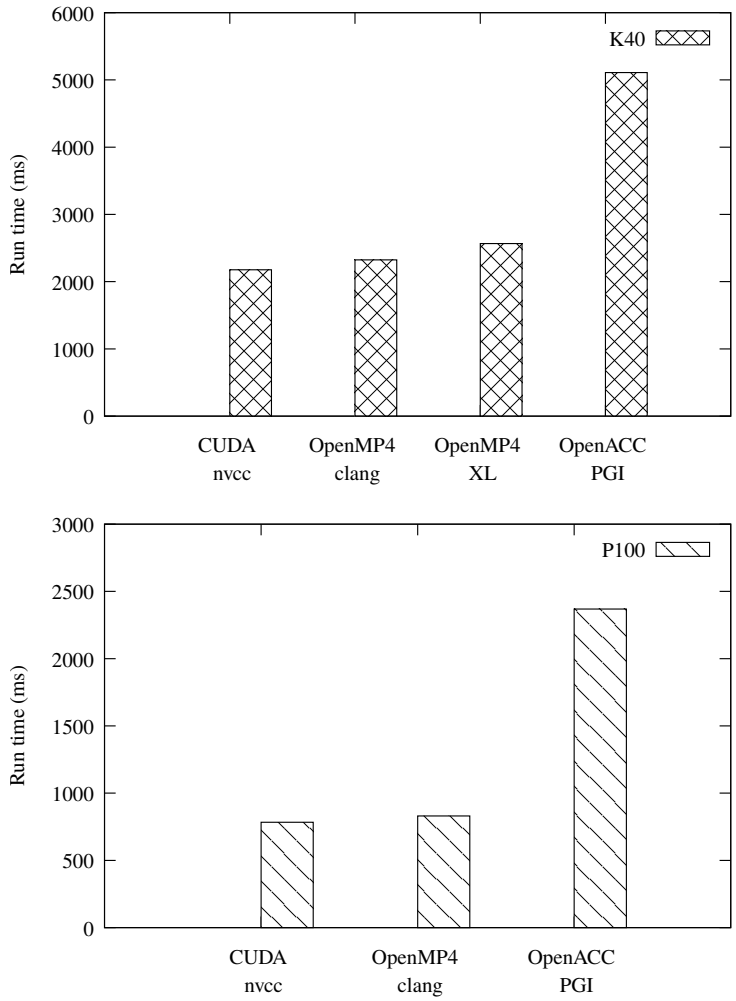


FIGURE 5.6: Cloverleaf 3_D measured run times of versions on the K40 and P100 GPUs.

In particular, the Clang Cloverleaf_2D execution was about 5,8% slower than the corresponding nvcc version on the K40 and about the 10,1% slower on the P100. Similarly, the Clang compiled Cloverleaf_3D performance was the 6,7% slower than the corresponding nvcc on the K40 and the 5,9% on the P100. These results can be considered more than satisfying in consideration of the fact that they were achieved by a high level eDSL, with a acceptable performance gap with respect to the native CUDA low-level (thus more difficult to exploit) API. The other compilers (IBM XL and OpenACC) evidenced worst performance with respect to both nvcc and Clang in all Cloverleaf benchmarks and on all GPUs.

In order to better understand the reason of this performance degradation, a further analysis was carried out. In particular, the nvprof profiler was used to collect information such as the number of registers per threads and the achieved occupancy on the device. Tables 5.4, 5.5, 5.6 and 5.7 show the collected information for the Cloverleaf_2D and Cloverleaf_3D benchmarks. Since Cloverleaf benchmarks are composed by 87 kernels, only the most six time consuming ones were analysed. From the collected data, the IBM XL and OpenACC compilers resulted to use a higher number of registers per thread with respect to Clang, resulting in a worst exploitation of the computational devices.

There have been studies on the effect of tuning the number of registers per thread on many applications and compilers [19]. They have shown that in some cases tuning manually the number of registers at compiler time (using the `maxrregcount` compilation option) can change the performance. In this work, further tests have been carried out by modifying the number of registers at compilation time. Nevertheless, the best performance have been achieved by the default value set by the compiler. Moreover, Clang has always a higher number of registers per threads than nvcc with both applications for the K40 GPU. Instead, for some kernels, such as `PdV_nopredict, PdV_predic, accelerate` with Cloverleaf_2D, Clang used a lower number of registers than nvcc for the P100, achieving a better occupancy. Only the `advec_mom_z_nonvector` kernel with `Cloverleaf_3D` on the P100 used a lower number of registers in the Clang version than the nvcc one. This confirms the difficulty behind the development of a back-end code generator in reaching the optimal number of registers per thread, since this metric depends on various factors such as hardware architecture, source code and compiler.

Kernel Name	nvcc CUDA	clang OpenMP4	XL OpenMP4	PGI OpenACC
PdV nopredict	36 (71.5 %)	42 (58.3 %)	80 (34.2 %)	63 (46.4 %)
viscosity	43 (59.9 %)	53 (53.9 %)	88 (29.2 %)	63 (44.7 %)
PdV predict	36 (71.1 %)	40 (68.7 %)	80 (34.2 %)	63 (46.7 %)
accelerate	42 (59.2 %)	44 (57.6 %)	62 (44.1 %)	63 (47.4 %)
advec_cell ydir	32 (90.4 %)	61 (47.2 %)	72 (39.3 %)	63 (52.3 %)
advec_cell xdir	30 (88.7 %)	61 (46.0 %)	64 (43.6 %)	54 (46.5 %)

TABLE 5.4: Register counts of the six most time consuming Cloverleaf_2D kernel on K40 GPU.

Kernel Name	nvcc CUDA	clang OpenMP4	XL OpenMP4	PGI OpenACC
viscosity	65 (42.0 %)	152 (16.7 %)	152 (16.6 %)	63 (27.6 %)
accelerate	65 (40.8 %)	69 (41.0 %)	69 (23.5 %)	63 (39.4%)
PdV nopredict	38 (72.0 %)	44 (58.6 %)	44 (23.6 %)	63 (29.6 %)
advec_mom y_nonvector	18 (68.5 %)	48 (57.1 %)	48 (46.7 %)	63 (35.6 %)
advec_mom z_nonvector	18 (68.0 %)	48 (56.8 %)	48 (46.4 %)	63 (36.0 %)
advec_mom x_nonvector	22 (68.1 %)	48 (56.3 %)	48 (46.4 %)	63 (36.0 %)

TABLE 5.5: Register counts of the six most time consuming Cloverleaf_3D kernel on K40 GPU.

Kernel Name	nvcc CUDA	clang OpenMP4	PGI OpenACC
PdV nopredict	48 (59.1 %)	32 (92.8 %)	64 (46.3 %)
advec_cell ydir	33 (70.6 %)	64 (45.1 %)	56 (52.1 %)
advec_cell xdir	34 (70.7 %)	64 (45.0 %)	54 (52.2 %)
PdV _predict	38 (71.0 %)	32 (92.7 %)	63 (46.7 %)
accelerate	36 (71.2 %)	32 (91.9 %)	64 (47.5 %)
advec_mom post_pre advec_x	30 (90.7 %)	32 (90.2 %)	40 (62.6 %)

TABLE 5.6: Register counts of the six most time consuming Cloverleaf_2D kernel on P100 GPU.

Kernel Name	nvcc CUDA	clang OpenMP4	PGI OpenACC
PdV nopredict	96 (29.5 %)	152 (16.2 %)	64 (10.1 %)
accelerate	21 (88.5 %)	28 (88.9 %)	64 (8.1 %)
viscosity	21 (88.7 %)	28 (88.8 %)	64 (10.9 %)
PdV predict	21 (88.6 %)	28 (88.8 %)	64 (7.3 %)
advec_mom z_nonvector	54 (52.7 %)	40 (72.0 %)	64 (8.0 %)
advec_mom y_nonvector	56 (52.0 %)	62 (46.8 %)	64 (10.2 %)

TABLE 5.7: Register counts of the six most time consuming Cloverleaf_3D kernel on P100 GPU.

5.2.2 Tealeaf Results

The test domain and number of total iteration of Tealeaf has been set to 500*500, for a total of 250000 cells, and to 10. The same considerations that were pointed out for the Cloverleaf applcation can be extended to the Tealeaf benchmark. However, in this benchmark all three compilers have achieved near optimal performance respect to the nvcc compiler. More in detail, for the K40, the clang and the IBM XL compilers are about 2% slower than CUDA, whereas for the P100 the OpenACC and OpenMP4 compiler achieved the best performance respect to CUDA. Table 5.7 shows the computational results obtained. Further analysis have been carried out for Tealeaf. Table 5.7 and 5.9 shown the number of registers and the achieved occupancy for the six most time consuming kernel on the K40 and P100 GPUs, respectively. The close performance achieved by OpenACC, clang anf IBM XL for Tealeaf benchmark respect to the more pronounced gap for Cloverleaf benchmark could be explained by the different computational intensive of the application. In fact, Cloverleaf is composed by 87 kernels, instead Tealeaf only 29.

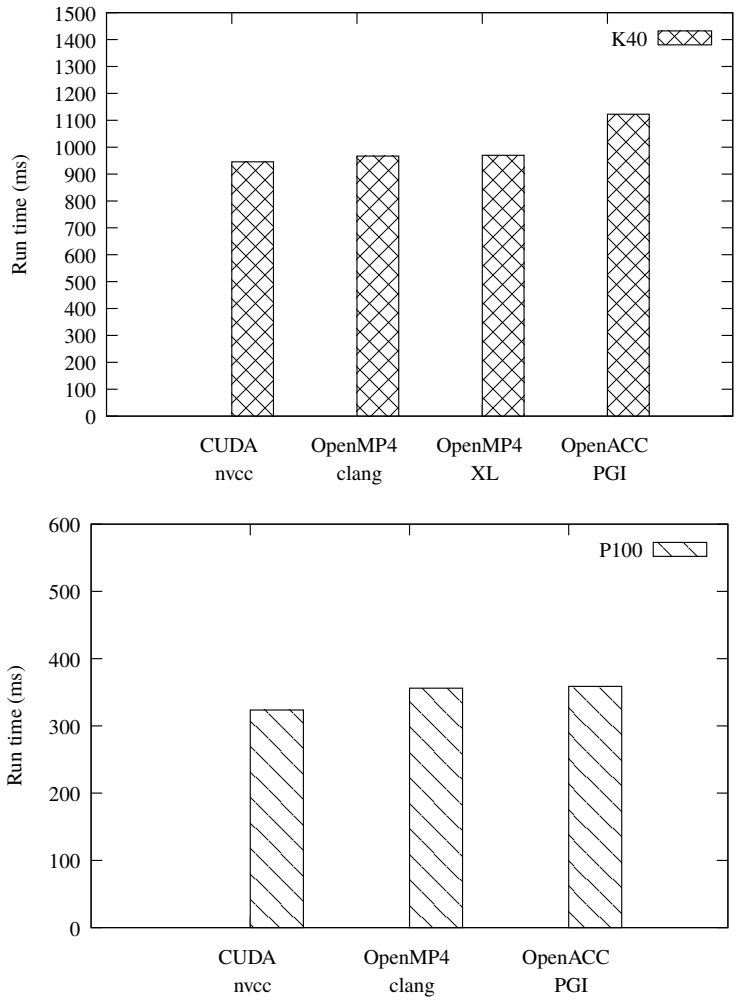


FIGURE 5.7: Tealeaf measured run times of versions on the K40 and P100 GPUs.

Kernel Name	nvcc CUDA	clang OpenMP4	XL OpenMP4	PGI OpenACC
cg_calc w_reduce	30 (49.8 %)	38 (70.0 %)	38 (49.7 %)	41 (41.6 %)
cg_calc ur_r_reduce	16 (44.8 %)	19 (90.8 %)	19 (96.3 %)	28 (8.8 %)
axpby	10 (84.8 %)	12 (82.3 %)	12 (84.6 %)	18 (85.3 %)
axpy	8 (85.4 %)	12 (82.9 %)	12 (84.3 %)	22 (85.8 %)
common _residual	28 (89.6 %)	34 (66.3 %)	34 (89.7 %)	41 (55.2 %)
common_init	30 (88.3 %)	34 (66.2 %)	34 (91.3 %)	55 (65.9 %)

TABLE 5.8: Register counts of the six most time consuming Tealeaf kernel on K40 GPU.

Kernel Name	nvcc CUDA	clang OpenMP4	PGI OpenACC
cg_calc w_reduce	32 (77.5 %)	32 (86.2 %)	40 (8.7 %)
cg_calc _ur_r_reduce	16 (68.5 %)	15 (82.8 %)	22 (50.5 %)
axpy	8 (77.6 %)	13 (78.1 %)	16 (49.9 %)
axpby	10 (77.6 %)	13 (78.5 %)	14 (50.0 %)
common _residual	32 (85.2 %)	30 (84.9 %)	44 (51.8 %)
common_init	28 (82.5 %)	29 (81.8 %)	40 (51.5 %)

TABLE 5.9: Register counts of the six most time consuming Tealeaf kernel on P100 GPU.

Chapter 6

Conclusions

To exploit new available HPC hardware, several programming languages have been developed. Some of them are fine-grained, such as CUDA and OpenCL, since they provide a low-level programming approach, while others are coarse-grained, such as OpenMP and OpenACC, since they allow to ignore most of hardware details, providing a high-level programming paradigm. However, to achieve optimal performance from heterogeneous high-performance accelerators, it is usually not sufficient to refer to a single language. The effort required to learn and to become familiar with this complex programming context is often excessive for Scientists who, in addition, are not willing to rewrite code from scratch each time a new language is released. At the contrary, they would prefer to rely on a solution that allows them to considerably reduce the implementation effort and to permit long-term code maintenance. Domain Specific Languages (DSLs) represent a possible solution, since they allow to exploit heterogeneous HPC hardware with a minor effort, by often hiding most parallel implementation detail to the user, providing a high level of abstraction, and to allow to convert existing (even serial) code straightforwardly.

This work has involved the design and development of different aspects of two DSLs specifically proposed for grid-based modeling, namely OpenCAL and OPS. The first one is an open source project developed at University of Calabria (Italy) and represents a computing abstraction layer providing Extended Cellular Automata as a formal computational paradigm. It allows to write the model to be implemented in a serial manner, and to easily obtain parallel versions for heterogeneous devices referring its specific components. Currently, multi-core, many-core accelerators like GPUs, and clusters of many-core accelerators can be exploited. Similarly, OPS is an open-source DSL for structured grid modeling able to run on heterogeneous devices, originally developed at the University of Oxford (UK), currently also supported by other research centers, among which the University of Warwick (UK). The approach is a bit different with respect to the one adopted by

OpenCAL. Specifically, OPS relies on a source-to-source translator, which permits to obtain different parallel versions from the serial code. Current support includes multi-core and many-core accelerators, as well as clusters of workstations.

Concerning OpenCAL, I have contributed to the design and implementation of its multi-GPU and multi-node components. In particular, the existing OpenCL-based OpenCAL-CL module was extended to allow for multi-GPU inter-node execution on PCI Express interconnected GPUs. Moreover, I have extended the preliminary version of the MPI-based OpenCAL-CLM component in order to allow OpenCAL to run multiple OpenCAL-CL instances both on inter-node and distributed memory computing solutions. More specifically, regarding OpenCAL-CL, I started working by considering its single-GPU version, which I had contributed to during my Master Degree Thesis. This version has been now improved by introducing the possibility to subdivide the computational domain in different chunks, by demanding to the available GPUs for their elaboration. Both uniform and non-uniform partitioning can be straightforwardly defined, by hiding data transfer details to the user. Similarly, regarding the OpenCAL-CLM component, I also started contributing from the beginning both at design and implementation level. At design level, the same scheme adopted for the multi-GPU OpenCAL component has been considered at inter-node level. Accordingly, the computational domain can be decomposed in uniform or non-uniform chunks and seamlessly distributed to the available nodes to be processed by OpenCAL-CL running instances.

In order to evaluate the performance of the resulting new OpenCAL version, a Sobel graphics edge detection filter, a Julia set fractal generator and the SciddicaT slow-moving fluid flow simulation model benchmarks were considered. A dual node cluster was considered for the tests, with nodes equipped with multi-GPU hardware. In particular, one node, called JPDM-2, adopted 2 Nvidia GTX 980, while two Titan Xp and a Tesla K40 Nvidia devices were installed on the other, called JPDM-1. Moreover, a single-node/single-GPU, single-node/multi-GPU and multi-node/multi-GPU systems were considered. The single-GPU performed tests pointed out that the Titan Xp outperformed the GTX 980 in all tests, while the compute dedicated Tesla K40 performed better on the Julia benchmark in absolute terms. The second series of tests regarded the single-node/dual-GPU execution on the JPDM-1 and JPDM-2 workstations. Two execution policies were considered, one based on the pure multi-GPU implementation, the other on a hybrid approach where MPI was considered for running two single-GPU instances in parallel. In all cases, superlinear effects were observed due to cache issues, except for the Julia benchmark that showed a almost linear scalability due to its pronounced compute bound nature. Eventually, the JPDM-SS cluster was employed for multi-node/multi-GPU tests. Even in this case, the above two execution policies were considered. Due to the non-

optimal interconnection network and to the unfavorable instruction/byte ratio of the adopted algorithms, sub-linear speed-ups were registered with respect to those achieved on the (most performing) JPDM-2 node, with SciddicaT showing even a slow-down. However, it is worth to note that SciddicaT was the most penalized algorithm on the distributed memory configuration. In fact, the optimization adopted in its more compute bound elementary process σ_1 actually reduced its application to only the 3.9% of the computational domain, undermining the scalability of the whole algorithm. Nevertheless, SciddicaT performed better on JPDM-SS than on its slower JPDM-1 node.

Regarding OPS, I have contributed to the design and implementation of a new source-to-source translator module based on the OpenMP 4.0/4.5 specifications. In particular, the new translator allows to automatically recognize the serial OPS API data and functions and generates the parallel OpenMP4 code for two different compilers, namely clang and IBM XL. The obtained generated code can run on many-core devices like GPUs, and represent an alternative to the already existing source-to-source translators based on CUDA, OpenCL and OpenACC. Specifically, the main effort has regarded the implementation of the translator for the clang compiler. Here, a thorough study has been performed to investigate different kind of optimizations at both source code and compiler level, in order to allow the code generator to reach a satisfying level of performance. Once the code generator was consolidated with respect to the clang compiler, a porting to XL has been performed. In this phase, many OpenMP4 directives have been introduced as alternatives to the ones already defined for clang. This was necessary since the considered compilers have different OpenMP back-end implementations, which can exhibit different behavior and therefore also different performances.

Single-GPU benchmark executions were considered for testing the new OPS translator on NVIDIA devices. In particular, a Tesla K40 and a Tesla P100 have been adopted. The state of art nvcc compiler for the NVIDIA hardware and the PGI OpenACC compilers were considered to compare the achieved OPS OpenMP4-based perfomance. The hydrodynamics Cloverleaf and Tealeaf mini-app benchmarks were considered for the tests. The tests pointed out that the nvcc compiler always outperformed the other compilers. Nevertheless, clang achieved an almost optimal performance. In particular, the Cloverleaf_2D execution was about 5,8% slower than the corresponding nvcc version on the K40 and about the 10,1% slower on the P100. Similarly, the Cloverleaf_3D performance was the 6,7% slower than the corresponding nvcc on the K40 and the 5,9% on the P100. Considering the Tealeaf benchmarks, all three compilers achieved near optimal performance with respect to the nvcc compiler. More in detail, for the K40, the clang and the IBM XL compilers are about 2% slower than CUDA, whereas, for the P100, the OpenACC and OpenMP4 compiler achieved the near optimal, very close to

the CUDA one. For both applications, further analysis were carried out. In particular, the number of registers per thread and the achieved occupancy for the six most time consuming kernel were considered. These analyses showed that clang always exploited a higher number of registers per thread than nvcc for the K40 GPU, whereas clang was able to use a lower number of registers than nvcc in some kernels on tests performed on the P100 GPU. These results showed up the difficulty behind the development of a back-end code generator in reaching the optimal performance in terms of number registers used per thread, depending on various factors such as hardware architecture, source code, and adopted compiler. Nevertheless, overall results can be considered more than satisfying in consideration of the fact that they were achieved by a high level embedded DSL, i.e. OpenMP4, with a more than acceptable performance gap with respect to the native CUDA low-level language.

Even if they provided more than satisfying results, the OpenCAL components designed and implemented in this work have to be considered preliminary. Future improvements will regard different design and implementation aspects. Among them, the next release of OpenCAL will provide different domain decomposition alternatives, currently restricted to only row-major partitioning. Moreover, OpenCAL will be also improved to allow a more flexible computation, giving the possibility to compute over specific subdomains. Furthermore, a preliminary implementation of SciddicaT model has been imported in OPS. Additional analysis, on both OPS and OpenCAL will be carried out on this model, in order to compare the performance of the two DSLs on parallel hardware such as many-core and multi-GPU/multi-node systems. This analysis will allow us to better understand the limits of the two software systems and also to study advantages and disadvantages of one with respect to the other.

Bibliography

[1] Applying openacc to the cloverleaf hydrodynamics mini-app. https://www.cray.com/sites/default/files/resources/OpenACC_213462.7_OpenACC_Cloverleaf_CS_FNL.pdf.

[2] Cloverleaf: Preparing hydrodynamics codes for exascale. https://cug.org/proceedings/cug2013_proceedings/includes/files/pap130-file2.pdf.

[3] Finite volume method in 1-d. http://web.mit.edu/16.90/BackUp/www/pdfs/Chapter16.pdf.

[4] Hilemms: High-level mesoscale modelling system. https://www.epcc.ed.ac.uk/projects-portfolio/hilemms-high-level-mesoscale-modelling-system.

[5] Hpc application support for gpu computing. https://www.nvidia.com/content/intersect-360-HPC-application-support.pdf.

[6] Sandia wins three r&d 100 awards. https://share-ng.sandia.gov/news/resources/news_releases/2013-rd100-awards/. July 8, 2013.

[7] Tealeaf. https://uk-mac.github.io/TeaLeaf/. July 8, 2013.

[8] *7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS@SC 2016, Salt Lake, UT, USA, November 14, 2016*. IEEE, 2016.

[9] *Third Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2016, Salt Lake City, UT, USA, November 14, 2016*. IEEE Computer Society, 2016.

[10] *CUDA C Best Practices Guide*. NVIDIA Corporation, 2017.

[11] *CUDA C Programming Guide*. NVIDIA Corporation, 2017.

[12] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung,

C. Bertolli, and K. O'Brien. Offloading support for openmp in clang and llvm. In *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*, LLVM-HPC '16, pages 1–11, Piscataway, NJ, USA, 2016. IEEE Press.

[13] B. Arca, T. Ghisu, and G. Trunfio. Gpu-accelerated multi-objective optimization of fuel treatments for mitigating wildfire hazard. *Journal of Computational Science*, (11):258–268, 2015.

[14] M. Avolio, S. Di Gregorio, V. Lupiano, and P. Mazzanti. SCIDDICA-SS3: a new version of cellular automata model for simulating fast moving landslides. *The Journal of Supercomputing*, 65(2):682–696, 2013.

[15] M. Avolio, S. Di Gregorio, F. Mantovani, A. Pasuto, R. Rongo, S. Silvano, and W. Spataro. Simulation of the 1992 tessina landslide by a cellular automata model and future hazard scenarios. *International Journal of Applied Earth Observation and Geoinformation*, 2(1):41–50, 2000.

[16] M. V. Avolio, G. M. Crisci, S. Di Gregorio, R. Rongo, W. Spataro, and D. D'Ambrosio. Pyroclastic flows modelling using Cellular Automata. *Computers & Geosciences*, 32:897–911, 2006.

[17] M. V. Avolio, S. Di Gregorio, W. Spataro, and G. A. Trunfio. A theorem about the algorithm of minimization of differences for multicomponent cellular automata. In G. C. Sirakoulis and S. Bandini, editors, *ACRI*, volume 7495 of *Lecture Notes in Computer Science*, pages 289–298. Springer, 2012.

[18] I. Azpeitia and J. Ibáñez. Spontaneous emergence of robust cellular replicators. In S. Bandini, B. Chopard, and M. Tomassini, editors, *Cellular Automata*, volume 2493 of *Lecture Notes in Computer Science*, pages 132–143. Springer Berlin Heidelberg, 2002.

[19] G. D. Balogh, I. Z. Reguly, and G. R. Mudalige. Comparison of parallelisation approaches, languages, and compilers for unstructured mesh algorithms on gpus. *CoRR*, abs/1711.01845, 2017.

[20] C. Bertolli, S. F. Antao, G.-T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien. Integrating gpu support for openmp offloading directives into clang. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 5:1–5:11, New York, NY, USA, 2015. ACM.

[21] E. Bilotta, A. Lafusa, and P. Pantano. Is self-replication an embedded characteristic of artificial/living matter? In *Proceedings of the eighth international conference on Artificial life*, ICAL 2003, pages 38–48, Cambridge, MA, USA, 2003. MIT Press.

[22] P. Bouvry, F. Seredyński, and A. Zomaya. Application of cellular automata for cryptography. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Waniewski, editors, *Parallel Processing and Applied Mathematics*, volume 3019 of *Lecture Notes in Computer Science*, pages 447–454. Springer Berlin Heidelberg, 2004.

[23] D. Buono, M. Danelutto, S. Lametti, and M. Torquati. Parallel patterns for general purpose many-core. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 131–139, Feb 2013.

[24] L. Carleson and T. W. Gamelin. *Complex Dynamics. Springer.* Springer, 1993.

[25] H. Chaté and P. Manneville. Criticality in cellular automata. *Physica D: Nonlinear Phenomena*, 45(13):122 – 135, 1990.

[26] B. Chopard. Cellular automata modeling of physical systems. In R. A. Meyers, editor, *Computational Complexity*, pages 407–433. Springer New York, 2012.

[27] B. Chopard and M. Droz. *Cellular Automata Modeling of Physical Systems.* Cambridge University Press, 1998.

[28] H. H. Chou and J. A. Reggia. Emergence of self-replicating structures in a cellular automata space. *Physica D*, 110:252–276, 1997.

[29] E. F. Codd. *Cellular Automata.* Academic Press, Inc., Orlando, FL, USA, 1968.

[30] R. D. Cook, D. S. Malkus, M. E. Plesha, and R. J. Witt. *Concepts and Applications of Finite Element Analysis.* John Wiley & Sons, Inc., USA, 2007.

[31] G. M. Crisci and S. R. G. Di Gregorio. A cellular space model of basaltic lava flow. In *International AMSE Conference Modelling & Simulation*, HPG '10, pages 65–67, Aire-la-Ville, Switzerland, 1982. Group 11 "Terrestrial resources and phenomena".

[32] J. P. Crutchfield, M. Mitchell, and R. Das. The evolutionary design of collective computation in cellular automata. In *MACHINE LEARNING JOURNAL*, 1998.

[33] D. D'Ambrosio, A. De Rango, M. Oliverio, D. Spataro, W. Spataro, R. Rongo, G. Mendicino, and A. Senatore. The Open Computing Abstraction Layer for Parallel Complex Systems Modeling on Many-Core Systems. *Journal of Parallel and Distributed Computing*, 2018. - In press.

[34] D. D'Ambrosio, S. Di Gregorio, S. Gabriele, and R. Gaudio. A cellular automata model for soil erosion by water. *Physics and Chemistry of the Earth, Part B: Hydrology, Oceans and Atmosphere*, 26(1):33 – 39, 2001.

[35] D. D'Ambrosio, S. Di Gregorio, and G. Iovine. Simulating debris flows through a hexagonal cellular automata model: SCIDDICA S3-hex. *Natural Hazards and Earth System Science*, 3(6):545–559, 2003.

[36] D. D'Ambrosio, G. Filippone, D. Marocco, R. Rongo, and W. Spataro. Efficient application of gpgpu for lava flow hazard mapping. *Journal of Supercomputing*, 65(2):630–644, 2013.

[37] D. D'Ambrosio, A. D. Rango, D. Spataro, R. Rongo, and W. Spataro. Applications of the opencal scientific library in the context of cfd: Applications to debris flows. In *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC)*, pages 720–725, May 2017.

[38] D. D'Ambrosio, R. Rongo, W. Spataro, M. Avolio, and V. Lupiano. Lava invasion susceptibility hazard mapping through cellular automata. *Lecture Notes in Computer Science*, 4173:452–461, 2006.

[39] D. D'Ambrosio, R. Rongo, W. Spataro, and G. Trunfio. Meta-model assisted evolutionary optimization of cellular automata: An application to the SCIARA model. *Lecture Notes in Computer Science*, 7204(PART 2):533–542, 2012.

[40] D. D'Ambrosio and W. Spataro. Parallel evolutionary modelling of geological processes. *Journal of Parallel Computing*, 33(3):186–212, 2007.

[41] M. de Menezes, E. Brigatti, and V. Schwämmle. Evolving cellular automata for diversity generation and pattern recognition: deterministic versus random strategy. *Journal of Statistical Mechanics: Theory and Experiment*, 2013(08):P08006, 2013.

[42] B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, editors. *Scaling OpenMP for Exascale Performance and Portability - 13th International Workshop on OpenMP, IWOMP 2017, Stony Brook, NY, USA, September 20-22, 2017, Proceedings*, volume 10468 of *Lecture Notes in Computer Science*. Springer, 2017.

[43] D. D'Humières, A. Clouqueur, and P. Lallemand. Lattice gases and parallel processors. *CALCOLO*, 25(1-2):129–151, 1988.

[44] D. D'Humières, P. Lallemand, and U. Frisch. Lattice Gas Models for 3D Hydrodynamics. *EPL (Europhysics Letters)*, 2(4):291+, Aug. 1986.

[45] S. Di Gregorio, G. Filippone, W. Spataro, and G. Trunfio. Accelerating wildfire susceptibility mapping through gpgpu. *Journal of Parallel and Distributed Computing*, 73(8):1183–1194, 2013.

[46] S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia. High performance scientific computing by a parallel cellular environment. *Future Generation Computer Systems*, 12(5):357–369, 1997.

[47] S. Di Gregorio and R. Serra. An empirical method for modelling and simulating some complex macroscopic phenomena by cellular automata. *Future Generation Computer Systems*, 16(2-3):259–271, 1999.

[48] S. Di Gregorio, R. Serra, and M. Villani. Applying cellular automata to complex environmental problems: The simulation of the bioremediation of contaminated soils. *Theoretical Computer Science*, 217(1):131 – 156, 1999.

[49] S. Di Gregorio and G. Trautteur. On reversibility in cellular automata. *Journal of Computer and System Sciences*, 11(3):382 – 391, 1975.

[50] M. Du, E. Hou, B. Wang, Y. Li, H. Wang, L. Duan, Q. Zhou, Y. Bai, and B. Jiang. The effect of tidal stream characteristic and mechanical support structure on horizontal axis tidal turbine performance. 2016. cited By 0.

[51] H. C. Edwards and C. R. Trott. Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*, pages 18–24, Aug 2013.

[52] G. Filippone, W. Spataro, D. D'Ambrosio, D. Spataro, D. Marocco, and G. Trunfio. Cuda dynamic active thread list strategy to accelerate debris flow simulations. In *Proceedings - 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015*, pages 316–320, 2015.

[53] G. Folino, G. Mendicino, A. Senatore, G. Spezzano, and S. Straface. A model based on cellular automata for the parallel simulation of 3D unsaturated flow. *Parallel Computing*, 32(5-6):357–376, 2006.

[54] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[55] M. Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123, 1970.

[56] M. Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[57] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[58] A. Giordano, A. d. Rango, D. Spataro, D. D'Ambrosio, C. Mastroianni, G. Folino, and W. Spataro. Parallel execution of cellular automata through space partitioning: The landslide simulation sciddicas3-hex case study. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 505–510, March 2017.

[59] Z. Gu, X. Cao, Y. Jiao, and W.-Z. Lu. Appropriate cfd models for simulating flow around spur dike group along urban riverways. *Water Resources Management*, 30(13):4559–4570, 2016. cited By 1.

[60] H. A. Gutowitz. A hierarchical classification of cellular automata. *Physica D: Nonlinear Phenomena*, 45(13):136 – 156, 1990.

[61] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[62] J. Hardy, Y. Pomeau, and G. de Pazzis. Thermodynamics and hydrodynamics for a modeled fluid. *Journal of Mathematical Physics*, 13(5):1949–1961, 1976.

[63] A. Hayashi, J. Shirako, E. Tiotto, R. Ho, and V. Sarkar. Performance evaluation of openmps target construct on gpus-exploring compiler optimizations. 2018.

[64] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[65] F. J. Higuera and J. Jiménez. Boltzmann approach to lattice gas simulations. *Europhysics Letters*, 9(7):663–668, 1989.

[66] B. Hjertager. Multi-fluid cfd analysis in process engineering. volume 276, 2017. cited By 0.

[67] R. D. Hornung and J. A. Keasler. The raja portability layer: Overview and status. 9 2014.

[68] I.-C. Hudrea, A.-F. Totorean, and D. Gaita. Computational fluid dynamics analysis of coronary stent malapposition. volume 68, pages 761–765, 2019. cited By 0.

[69] T. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, 1987.

[70] C. T. Jacobs, S. P. Jammy, and N. D. Sandham. Opensbli: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. *Journal of Computational Science*, 18:12 – 23, 2017.

[71] F. Jiménez-Morales. An evolutionary approach to the study of non-trivial collective behavior in cellular automata. In S. Bandini, B. Chopard, and M. Tomassini, editors, *Cellular Automata*, volume 2493 of *Lecture Notes in Computer Science*, pages 32–43. Springer Berlin Heidelberg, 2002.

[72] J. Kari. Reversibility of 2d cellular automata is undecidable. *Physica D: Nonlinear Phenomena*, 45(13):379 – 385, 1990.

[73] M. Karim, M. Rahman, M. Hai, M. Shimul, and S. Sudhi. Numerical investigation of flow around cavitating hydrofoil using finite volume method. volume 1980, 2018. cited By 0.

[74] S.-E. Kim, H. Shan, R. Miller, B. Rhee, A. Vargas, S. Aram, and J. Gorski. A scalable and extensible computational fluid dynamics software framework for ship hydrodynamics applications: Navyfoam. *Computing in Science and Engineering*, 19(6):33–39, 2017. cited By 0.

[75] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and S. A. Jarvis. Achieving performance portability for a heat conduction solver mini-application on modern multi-core systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 834–841, Sept 2017.

[76] J. M. V. A. Koelman. A Simple Lattice Boltzmann Scheme for Navier-Stokes Fluid Flow. *EPL (Europhysics Letters)*, 15(6):603, 1991.

[77] C. G. Langton. Self-reproduction in cellular automata. *Physica D*, 10D(1-2):135–44, 1984.

[78] C. G. Langton. Studying artificial life with cellular automata. *Physica D*, 2(1-3):120–149, Oct. 1986.

[79] C. G. Langton. Computation at the edge of chaos: phase transitions and emergent computation. *Physica D*, 42(1-3):12–37, 1990.

[80] Q. Li, D.-K. Wang, Y.-J. Yang, and J. Fang. Numerical simulation of transverse jet flow field under supersonic inflow. volume 139, 2017. cited By 0.

[81] P. Maji, N. Ganguly, S. Saha, A. Roy, and P. Chaudhuri. Cellular automata machine for pattern recognition. In S. Bandini, B. Chopard, and M. Tomassini, editors, *Cellular Automata*, volume 2493 of *Lecture Notes in Computer Science*, pages 270–281. Springer Berlin Heidelberg, 2002.

[82] A. C. Mallinson, S. A. Jarvis, W. P. Gaudin, and J. A. Herdman. Experiences at scale with pgas versions of a hydrodynamics application. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 9:1–9:11, New York, NY, USA, 2014. ACM.

[83] A. C. Mallinson, S. A. Jarvis, W. P. Gaudin, and J. A. Herdman. Experiences at scale with pgas versions of a hydrodynamics application. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 9:1–9:11, New York, NY, USA, 2014. ACM.

[84] H. V. McIntosh. Wolfram's class IV automata and a good life. *Physica D: Nonlinear Phenomena*, 45(13):105–121, 1990.

[85] G. R. McNamara and G. Zanetti. Use of the Boltzmann equation to simulate lattice-gas automata. *Physical Review Letters*, 61(20):2332–2335, 1988.

[86] G. Mendicino, J. Pedace, and A. Senatore. 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(1-3):128–146, 2015.

[87] G. Mendicino, A. Senatore, G. Spezzano, and S. Straface. Three-dimensional unsaturated flow modeling using cellular automata. *Water Resources Research*, 42(11):W11419, 2006.

[88] Y. Menni, A. Azzi, and A. Chamkha. Aerodynamics and heat transfer over solid-deflectors in transverse, staggered, corrugated-upstream and corrugated-downstream patterns. *Periodica Polytechnica Mechanical Engineering*, 62(3):209–217, 2018. cited By 0.

[89] V. Mironov, Y. Alexeev, K. Keipert, M. D'mello, A. Moskovsky, and M. S. Gordon. An efficient mpi/openmp parallelization of the hartree-fock method for the second generation of intel xeon phi processor. *CoRR*, abs/1708.00033, 2017.

[90] E. F. Moore. Machine models of self-reproduction. In *Mathematical Problems in Biological Sciences (Proceedings of Symposia in Applied Mathematics)*. American Mathematical Society, 1962.

[91] F. Moukalled, L. Mangani, and M. Darwish. *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM and Matlab*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[92] G. Mudalige, I. Reguly, M. Giles, W. Gaudin, A. Mallinson, and O. Perks. High-level Abstractions for Performance, Portability and Continuity of Scientific Software on Future Computing Systems. Technical report, 2014.

[93] G. R. Mudalige, I. Z. Reguly, M. B. Giles, A. C. Mallinson, W. P. Gaudin, and J. A. Herdman. Performance analysis of a high-level abstractions-based hydrocode on future computing systems. In *PMBS@SC*, 2014.

[94] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

[95] J. Myhill. The converse of Moores Garden-of-Eden theorem. *Proceedings of The American Mathematical Society*, 14:685–685, 1963.

[96] M. Oliverio, W. Spataro, D. D'Ambrosio, R. Rongo, G. Spingola, and G. Trunfio. OpenMP parallelization of the SCIARA Cellular Automata lava flow model: Performance analysis on shared-memory computers. In *Procedia Computer Science*, volume 4, pages 271–280, 2011.

[97] G. Palermo and J. Feo, editors. *Proceedings of the ACM International Conference on Computing Frontiers, CF'16, Como, Italy, May 16-19, 2016*. ACM, 2016.

[98] B. Priego, D. Souto, F. Bellas, and R. J. Duro. Hyperspectral image segmentation through evolved cellular automata. *Pattern Recognition Letters*, 34(14):1648–1658, 2013.

[99] Y. H. Qian, D. D'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17:479, 1992.

[100] A. D. Rango, P. Napoli, D. D'Ambrosio, W. Spataro, A. D. Renzo, and F. D. Maio. Structured grid-based parallel simulation of a simple dem model on heterogeneous systems. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, volume 00, pages 588–595, Mar 2018.

[101] A. D. Rango, D. Spataro, W. Spataro, and D. DAmbrosio. A first multi-gpu/multi-node implementation of the open computing abstraction layer. *Journal of Computational Science*, 2018.

[102] G. Ravazzani, D. Rametta, and M. Mancini. Macroscopic cellular automata for groundwater modelling: A first approach. *Environmental Modelling and Software*, 26(5):634–643, 2011.

[103] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Design and development of domain specific active libraries with proxy applications. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing*, CLUSTER '15, pages 738–745, Washington, DC, USA, 2015. IEEE Computer Society.

[104] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith. The ops domain specific abstraction for multi-block structured grid computations. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 58–67, Nov 2014.

[105] A. Roli and F. Zambonelli. Emergence of macro spatial structures in dissipative cellular automata. In *In Proceedings of Cellular Copyright 2007*, pages 144–155. John Wiley & Sons, 2002.

[106] P. L. Rosin. Training Cellular Automata for Image Processing. In H. Kalviainen, J. Parkkinen, and A. Kaarna, editors, *Image Analysis*, volume 3540 of *Lecture Notes in Computer Science*, pages 195–204. Springer Berlin Heidelberg, 2005.

[107] P. L. Rosin. Image processing using 3-state cellular automata. *Computer Vision and Image Understanding*, 114(7):790 – 802, 2010.

[108] T. Salles, S. Lopez, M. Cacas, and T. Mulder. Cellular automata model of density currents. *Geomorphology*, 88(12):1 – 20, 2007.

[109] A. Senatore, D. DAmbrosio, A. De Rango, R. Rongo, W. Spataro, S. Straface, and G. Mendicino. Accelerating a three-dimensional eco-hydrological cellular automaton on gpgpu with opencl. *AIP Conference Proceedings*, 1776(1):080003, 2016.

[110] L. G. Shapiro and G. C. Stockman. *Computer Vision*. 2001.

[111] I. M. Smith. The finite element method, 3rd edn, o. c. zienkiewicz, mcgraw-hill (u.k.) ltd. 1977. no. of pages 787. *International Journal for Numerical Methods in Engineering*, 12(6):1054–1054.

[112] W. Spataro, M. V. Avolio, V. Lupiano, G. A. Trunfio, R. Rongo, and D. D'Ambrosio. 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. In *International Conference on Computational Science*, pages 17–26, 2010.

[113] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[114] S. Succi. *Automi cellulari. Una nuova frontiera del calcolo scientifico.* Collana informatica domani / IBM SEMEA. Franco Angeli, 1991.

[115] S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond.* Clarendon Press, Oxford, 2001.

[116] D. Talia. The role of parallel cellular programming in computational science. In J. Palma, J. Dongarra, and V. Hernndez, editors, *Vector and Parallel Processing VECPAR 2000*, volume 1981 of *Lecture Notes in Computer Science*, pages 207–220. Springer Berlin Heidelberg, 2001.

[117] J. Thatcher. *Universality in the Von Neumann Cellular Model.* EthicsSA research report. IBM Watson Research Center, 1964.

[118] T. Toffoli and N. Margolus. *Cellular automata machines: a new environment for modeling.* MIT Press, Cambridge, MA, USA, 1987.

[119] T. Toffoli and N. Margolus. Invertible cellular automata: A review. *Physica D*, 45:229–253, 1990.

[120] M. Tomassini and M. Perrenoud. Cryptography with cellular automata. *Applied Soft Computing*, 1(2):151–160, 2001.

[121] G. A. Trunfio, D. D'Ambrosio, R. Rongo, W. Spataro, and S. Di Gregorio. A new algorithm for simulating wildfire spread through cellular automata. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 22(1):6, 2011.

[122] B. van Leer. Towards the ultimate conservative difference scheme. v. a second-order sequel to godunov's method. *Journal of Computational Physics*, 32(1):101 – 136, 1979.

[123] V. Volkov. *Understanding Latency Hiding on GPUs PhD thesis.* PhD thesis, EECS Department, University of California, Berkeley, 2016.

[124] J. Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[125] v.v. Nvidia's next generation cuda compute architecture: Kepler tm gk110 - whitepaper. Technical report, Nvidia Coorporation, 2012.

[126] J.-W. Wang, J.-J. Kim, W. Choi, D.-S. Mun, J.-E. Kang, H. Kwon, J.-S. Kim, and K.-S. Han. Effects of wind fences on the wind environment around jang bogo antarctic research station. *Advances in Atmospheric Sciences*, 34(12):1404–1414, 2017. cited By 0.

[127] J. Weimar. *Simulation with Cellular Automata*. Logos-Verlag, Berlin, 1998.

[128] L. Whitley. *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, San Francisco, 1993.

[129] S. Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601–644, 1983.

[130] S. Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

List of Figures

List of Tables