A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures

Article *in* Communications in Computational Physics · September 2013

DOI: 10.4208/cicp.110113.010813a

CITATIONS
123

READS
10,403

3 authors:

Cristobal A Navarro
Universidad Austral de Chile
34 PUBLICATIONS **222** CITATIONS
SEE PROFILE

Nancy Hitschfeld
University of Chile
99 PUBLICATIONS **774** CITATIONS
SEE PROFILE

Luis Mateu
University of Chile
8 PUBLICATIONS **164** CITATIONS
SEE PROFILE

Commun. Comput. Phys.
doi: 10.4208/cicp.110113.010813a

Vol. **15**, No. 2, pp. 285-329
February 2014

REVIEW ARTICLE

A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures

Cristóbal A. Navarro[1,2,*], Nancy Hitschfeld-Kahler[1] and Luis Mateu[1]

[1] *Department of Computer Science (DCC), Universidad de Chile, Santiago, Chile.*
[2] *Centro de Estudios Científicos (CECS), Valdivia, Chile.*

Received 11 January 2013; Accepted (in revised version) 1 August 2013

Available online 10 September 2013

Abstract. Parallel computing has become an important subject in the field of computer science and has proven to be critical when researching high performance solutions. The evolution of computer architectures (*multi-core* and *many-core*) towards a higher number of cores can only confirm that parallelism is the method of choice for speeding up an algorithm. In the last decade, the graphics processing unit, or GPU, has gained an important place in the field of high performance computing (HPC) because of its low cost and massive parallel processing power. Super-computing has become, for the first time, available to anyone at the price of a desktop computer. In this paper, we survey the concept of parallel computing and especially GPU computing. Achieving efficient parallel algorithms for the GPU is not a trivial task, there are several technical restrictions that must be satisfied in order to achieve the expected performance. Some of these limitations are consequences of the underlying architecture of the GPU and the theoretical models behind it. Our goal is to present a set of theoretical and technical concepts that are often required to understand the GPU and its *massive parallelism* model. In particular, we show how this new technology can help the field of *computational physics*, especially when the problem is *data-parallel*. We present four examples of computational physics problems; *n-body*, *collision detection*, *Potts model* and *cellular automata* simulations. These examples well represent the kind of problems that are suitable for GPU computing. By understanding the GPU architecture and its massive parallelism programming model, one can overcome many of the technical limitations found along the way, design better GPU-based algorithms for computational physics problems and achieve speedups that can reach up to two orders of magnitude when compared to sequential implementations.

AMS subject classifications: 68W10, 65Y05, 68Q10, 68N19, 68M14, 00A79, 81Vxx

*Corresponding author. *Email addresses:* crinavar@dcc.uchile.cl (C. A. Navarro), nancy@dcc.uchile.cl (N. Hitschfeld), lmateu@dcc.uchile.cl (L. Mateu)

Key words: GPU computing, parallel computing, computing models, algorithms, data parallel, massive parallelism, Potts model, Ising Model, collision detection, *n*-body, Cellular Automata.

Contents

1 Introduction

For some computational problems, CPU-based algorithms are not fast enough to give a solution in a reasonable amount of time. Furthermore, these problems can become even larger, to the point that not even a *multi-core* CPU-based algorithm is fast enough. Problems such as these can be found in science and technology; natural sciences [52, 108, 116] (Physics, Biology, Chemistry), information technologies [115] (IT), geospatial information systems [11, 66] (GIS), structural mechanics problems [12] and even abstract mathematical/computer science (CS) problems [84, 98, 101, 109]. Today, many of these problems can be solved faster and more efficiently by using massive parallel processors.

The story of how massive parallel processors were born is actually one of a kind because it combines two fields that were not related at all; *computational science* and *video-game industry*. In science, there is a constant need for solving the largest problems in a reasonable amount of time. This need has led to the construction of massively parallel *super-computers* for understanding phenomena such as galaxy formation, molecular dynamics and climate change, among others. On the other hand, the video-game industry is in a constant need for achieving real-time photo-realistic graphics, with the major restriction of running their algorithms on consumer-level computer hardware. The need of realistic video-games led to the invention of the graphics accelerator, a small parallel processor that handled many graphical computations using hardware implemented functions. The two needs, combined together, have given birth to one of the most important hardware for parallel computing; the GPU.

The **GPU** (graphics processing unit) is the maximum exponent of parallel computing. It is physically small to fit inside a desktop machine and it is massively parallel as a small scale super-computer, capable of handling up to thousands of threads in parallel. The GPU is indeed attractive to any scientist, because it is no more restricted to graphical problems and offers impressive parallel performance at the cost of a desktop computer. It is not a surprise to see GPU-based algorithms achieve considerable amounts of speedup over a classic CPU-based solution [10,30], even by two orders of magnitude [25,78].

Some problems do not have a parallel solution [47]. For example, the approximation of $\sqrt{x}$ using the Newton-Rhapson method [100] cannot be parallelized because each iteration depends on the value of the previous one; there is the issue of *time dependence*. Such problems do not benefit from parallelism at all and are best solved using a CPU. On the other hand, there are problems that can be naturally split into many independent subproblems; *e.g.*, matrix multiplication can be split into several independent multiply-add computations. Such problems are massively parallel, they are very common in computational physics and they are best solved using a GPU. In some cases, these problems become so parallelizable that they receive the name *embarrassingly parallel*[†] or *pleasingly parallel* [87,97].

One of the most important aspects of parallel computing is its close relation to the underlying hardware and programming models. Typical questions in the field are: *What type of problem I am dealing with? Should I use a CPU or a GPU? Is it a MIMD or SIMD architecture? It is a distributed or shared memory system? What should I use: OpenMP, MPI, CUDA or OpenCL? Why the performance is not what I had expected? Should I use a hierarchical partition? How can I design a parallel algorithm?*. These questions are indeed important when searching for a high performance solution and their answers lie in the areas of algorithms, computer architectures, computing models and programming models. GPU computing also brings up additional challenges such as manual cache usage, parallel memory access patterns, communication, thread mapping and synchronization, among others. These challenges are critical for implementing an efficient GPU algorithm.

This paper is a comprehensive survey of basic and advanced topics that are often required to understand parallel computing and especially GPU computing. The main contribution is the presentation of massive parallel architectures as a useful technology for solving computational physics problems. As a result, the reader should become more confident in the fundamental and technical aspects of GPU computing, with a clear idea of what types of problems are best suited for GPU computing.

We organized the rest of the sections in the following way: fundamental concepts of parallel computing and theoretical background are presented first, such as basic definitions, performance measures, computing models, programming models and architectures (from Section 2 to Section 6). Advanced concepts of GPU computing start from Section 7, and cover strategies for designing massive parallel algorithms, the massive

[†]The term *embarrassingly parallel* means that it would be embarrassing to not take advantage of such parallelization. However in some cases, the term has been taken as of being embarrassed to make such parallelization; this meaning is unwanted. An alternative name is *pleasingly parallel*.

parallelism programming model and its technical restrictions. We describe four examples of computational physics problems that have been solved using GPU-based algorithms, giving an idea of what types of problems are ideal to be solved on GPU. Finally, Section 10 is a dedicated to the latest advances in the field. We choose this organization because it provides to the reader the needed background on parallel computing, making the GPU computing sections easier to understand.

2 Basic concepts

The terms *concurrency* and *parallelism* are often debated by the computer science community and sometimes it has become unclear what the difference is between the two, leading to misunderstanding of very fundamental concepts. Both terms are frequently used in the field of HPC and their difference must be made clear before discussing more advanced concepts along the survey. The following definitions of concurrency and parallelism are consistent and considered correct [14];

Definition 2.1. Concurrency is a property of a program (at design level) where two or more tasks can *be in progress simultaneously*.

Definition 2.2. Parallelism is a run-time property where two or more tasks *are being executed simultaneously*.

There is a difference between *being in progress* and *being executed* since the first one does not necessarily involve being in execution. Let C and P be concurrency and parallelism, respectively, then $P \subset C$. In other words, parallelism requires concurrency, but concurrency does not require parallelism. A nice example where both concepts come into play is the operating system (OS); it is concurrent by design (performs multi-tasking so that many tasks are in progress at a given time) and depending on the number of physical processing units, these tasks can run parallel or not. With these concepts clear, now we can make a simple definition for parallel computing:

Definition 2.3. Parallel computing is the act of solving a problem of size n by dividing its domain into $k \geq 2$ (with $k \in \mathbb{N}$) parts and solving them with p physical processors, simultaneously.

Being able to identify the type of problem is essential in the formulation of a parallel algorithm. Let P_D be a problem with domain D. If P_D is parallelizable, then D can be decomposed into k sub-problems:

$$D = d_1 + d_2 + \cdots + d_k = \sum_{i=1}^{k} d_i. \tag{2.1}$$

P_D is a **data-parallel problem** if D is composed of data elements and solving the problem requires applying a kernel function $f(\cdots)$ to the whole domain:

$$f(D)=f(d_1)+f(d_2)+\cdots+f(d_k)=\sum_{i=1}^{k}f(d_i). \tag{2.2}$$

P_D is a **task-parallel problem** if D is composed of functions and solving the problem requires applying each function to a common stream of data S:

$$D(S)=d_1(S)+d_2(S)+\cdots+d_k(S)=\sum_{i=1}^{k}d_i(S). \tag{2.3}$$

Data-parallel problems are ideal candidates for the GPU. The reason is because the GPU architecture works best when all threads execute the same instructions but on different data. On the other hand, task-parallel problems are best suited for the CPU. The reason is because the CPU architecture allows different tasks to be executed on each thread. This classification scheme is critical for achieving the best partition of the problem domain, which is in fact the first step when designing a parallel algorithm. It also provides useful information when choosing the best hardware for the implementation (CPU or GPU). Computational physics problems often classify as data-parallel, therefore they are good candidates for a massive parallelization on GPU. Since the aim of this work is to provide a survey on parallel computing for computational physics, most of the explanations will be in the context of data-parallel problems.

3 Performance measures

Performance measures consist of a set of metrics that can be used for quantifying the quality of an algorithm. For sequential algorithms, the metrics *time* and *space* are sufficient. For parallel algorithms the scenario is a little more complicated. Apart from time and space, metrics such as *speedup* and *efficiency* are necessary for studying the quality of a parallel algorithm. Furthermore, when an algorithm cannot be completely parallelized, it is useful to have a theoretical estimate of the maximum speedup possible. In these cases, the laws of Amdahl and Gustafson become useful for such analysis. On the experimental side, metrics such as *memory bandwidth* and *floating point operations per second* (Flops) define the performance of a parallel architecture when running a parallel algorithm.

Given a problem of size n, the running time of a parallel algorithm, using p processors, is denoted:

$$T(n,p). \tag{3.1}$$

From the theoretical point of view, the metrics *work* and *span* define the basis for computing other metrics such as speedup and efficiency.

3.1 Work and span

The quality of a parallel algorithm can be defined by two metrics as stated by Cormen *et al.* [27]; *work* and *span*. Both metrics are important because they give limits to parallel computing and introduce the notion of *work*. Parallel algorithms have the challenge of being fast, but also to generate the minimum amount of extra work. By doing less extra work, they become more efficient.

Work is defined as the total time needed to execute a parallel algorithm using one processor; denoted as $T(n,1)$. *Span* is defined as the longest time needed to execute a parallel path of computation by one thread; denoted as $T(n,\infty)$. Span is the equivalent of measuring time when using an infinite amount of processors.

These two metrics provide lower bounds for $T(n,p)$. The *work law* equation states the first lower bound:

$$T(n,p) \geq \frac{T(n,1)}{p}. \tag{3.2}$$

That is, the running time of a parallel algorithm must be at least $1/p$ of its *work*. With the *work law*, one can realize that parallel algorithms run faster when the work per processor is balanced.

The *span law* defines the second lower bound for $T(n,p)$:

$$T(n,p) \geq T(n,\infty). \tag{3.3}$$

This means that the time of a parallel algorithm cannot be lower than the span or the minimal amount of time needed by a processor in an infinite processor machine.

3.2 Speedup

One of the most important actions in parallel computing is to actually measure how much faster a parallel algorithm runs with respect to the best sequential one. This measure is known as *speedup*.

For a problem of size n, the expression for speedup is:

$$S_p = \frac{T_s(n,1)}{T(n,p)}, \tag{3.4}$$

where $T_s(n,1)$ is the time of the best sequential algorithm (*i.e.*, $T_s(n,1) \leq T(n,1)$) and $T(n,p)$ is the time of the parallel algorithm with p processors, both solving the same problem. Speedup is upper bounded when n is fixed because of the *work law* from equation (3.2):

$$S_p \leq p. \tag{3.5}$$

If the speedup increases linearly as a function of p, then we speak of *linear speedup*. Linear speedup means that the overhead of the algorithm is always in the same proportion with its running time, for all p. In the particular case of $T(n,p) = T_s(n,1)/p$, we then speak

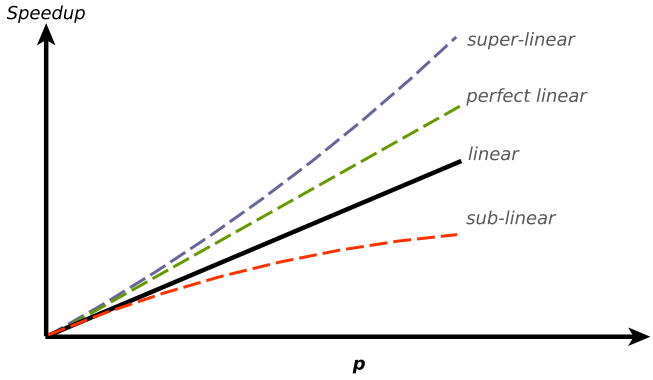

Figure 1: The four possible curves for speedup.

of *ideal speedup* or *perfect linear speedup*. It is the maximum theoretical value of speedup a parallel algorithm can achieve when n is fixed. In practice, it is hard to achieve linear speedup let alone perfect linear speedup, because memory bottlenecks and overhead increase as a function of p. What we find in practice is that most programs achieve *sub-linear* speedup, that is, $T(n,p) \geq T_s(n,1)/p$. Fig. 1 shows the four possible curves.

For the last three decades it has been debated whether *super-linear speedup* (*i.e.*, $S_p > p$) is possible or just a fantasy. Super-linear speedup is an important matter in parallel computing and proving its existence would benefit computer science, since parallel machines would be literally *more than the sum of their parts* (Gustafson's conclusion in [50]). Smith [110] and Faber *et al.* [35] state that it is not possible to achieve super-linear speedup and if such a parallel algorithm existed, then a single-core computation of the same algorithm would be no less than p times slower (leading to linear speedup again). On the opposite side, Parkinson's work [99] on parallel efficiency proposes that super-linear speedup is sometimes possible because the single processor has loop overhead. Gustafson supports super-linear speedup and considers a more general definition of S_p, one as the ratio of speeds ($speed = work/time$) [50] and not the ratio of times as in Eq. (3.4). Gustafson concludes that the definition of *work*, its assumption of being constant and the assumption of *fixed-size* speedup as the only model are the causes for thinking of the impossibility of super-linear speedup [51].

It is important to mention that there are three different models of speedup. (1) *Fixed-size speedup* is the one explained recently; fixes n and varies p. It is the most popular model of speedup. (2) *Scaled speedup* consists of varying n and p such that the problem size per processor remains constant. Lastly, (3) *fixed-time speedup* consists of varying n and p such that the amount of work per processor remains constant. Throughout this survey, *fixed-size speedup* is assumed by default. For the case of (2) and (3), speedup becomes a curve from the surface on (n,p).

If a problem cannot be completely parallelized (one of the causes for sub-linear speedup), a partial speedup expression is needed in the place of Eq. (3.4). Amdahl and

Gustafson proposed each one an expression for computing partial speedup. They are known as the *laws of speedup*.

3.2.1 Amdahl's law

Let c be the fraction (in a/b form or as a real number) of a program that is parallel, $(1-c)$ the fraction that runs sequential and p the number of processors. Amdahl's law [5] states that for a fixed size problem the expected overall speedup is given by:

$$S(p)=\frac{1}{(1-c)+\frac{c}{p}}. \tag{3.6}$$

If $p\approx\infty$, Eq. (3.6) becomes:

$$S(p)=\frac{1}{1-c}. \tag{3.7}$$

That is, if a computer has a large number of processors (*i.e.,* a super-computer or a modern GPU), then the maximum speedup is limited by the sequential part of the algorithm (*e.g,* if $c=4/5$ then the maximum speedup is 5x).

Amdahl's law is useful for algorithms that need to scale its performance as a function of the number of processors, fixing the problem size n. This type of scaling is known as *strong scaling*.

3.2.2 Gustafson's law

Gustafson's law [49] is another useful measure for theoretical performance analysis. This metric does not assume a fixed size of the problem as Amdahl's law did. Instead, it uses the fixed-time model where work per processor is kept constant when increasing p and n. In Gustafson's law, the time of a parallel program is composed of a sequential part s and a parallel part c executed by p processors.

$$T(p)=s+c. \tag{3.8}$$

If the sequential time for all the computation is $s+cp$, then the speedup is:

$$S(p)=\frac{s+cp}{s+c}=\frac{s}{s+c}+\frac{cp}{s+c}. \tag{3.9}$$

Defining α as the fraction of serial computation $\alpha=s/(s+c)$, then the parallel fraction is $1-\alpha=c/(s+c)$. Finally, Eq. (3.9) becomes the *fixed-time speedup* $S(p)$:

$$S(p)=\alpha+p(1-\alpha)=p-\alpha(p-1). \tag{3.10}$$

Gustafson's law is important for expanding the knowledge in parallel computing and the definition of speedup. With Gustafson's law, the idea is to increase the work linearly as a function of p and n. Now the problem size is not fixed anymore, instead the work per processor is fixed. This type of scaling is also known as *weak scaling*. There are many

applications where the size of the problem would actually increase if more computational power was available; weather prediction, computer graphics, Monte Carlo algorithms, particle simulations, etc. Fixing the problem size and measuring *time* vs p is mostly done for academic purposes. As the problem size gets larger, the parallel part p may grow faster than α.

While it is true that speedup might be one of the most important measures of parallel computing, there are also other metrics that provide additional information about the quality of a parallel algorithm, such as the efficiency.

3.3 Efficiency

If we divide Eq. (3.5) by p, we get:

$$E_p = \frac{S_p}{p} = \frac{T_s(n,1)}{pT(n,p)} \leq 1. \tag{3.11}$$

E_p is the efficiency of an algorithm using p processors and it tells how well the processors are being used. $E_p = 1$ is the maximum efficiency and means optimal usage of the computational resources. Maximum efficiency is difficult to achieve in an implemented solution (it is a consequence of the difficult to achieve perfect linear speedup). Today, efficiency has become as important as speedup, if not more, since it measures how well the hardware is used and it tells which implementations should have priority when competing for limited resources (cluster, supercomputer, workstation).

3.4 FLOPS

The FLOPS metric represents raw arithmetic performance and is measured as the number of floating point operations per second. Let F_h be the peak floating point performance of a known hardware and F_e the floating point performance measured for the implementation of a given algorithm, then F_c is defined as:

$$F_c = \frac{F_e}{F_h}. \tag{3.12}$$

F_c tells us the efficiency of the numerical computation relative to a given hardware. A value of $F_c = 1$ means maximum hardware usage for numerical computations.

The highest performance reported up to date (March 2013) is approximately 17.5 Pflops by the 'Titan' supercomputer from *DOE/SC/ Oak Ridge National Laboratory*[‡]. There is high enthusiasm for achieving for the first time the Exaflops scale. It is believed that in the following years, with the help of GPU-based hardware, the goal of Exaflops scale will be achieved.

[‡]An updated list of the 500 most powerful super-computers in the world is available at the 'www.top500.org'.

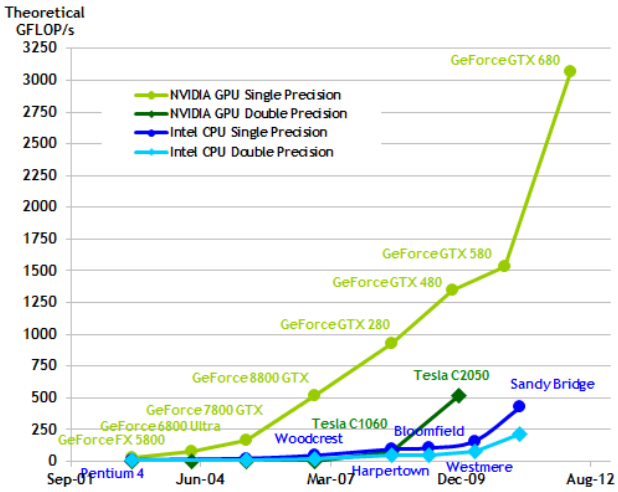

Figure 2: Comparison of CPU and GPU single precision floating point performance through the years. Image taken from Nvidia's *CUDA C programming guide* [93].

Today, high-end CPUs offer up to 240 GFlops (Intel Xeon E5 2690) of numerical performance while high-end GPUs offer approximately 4 TFlops (Nvidia Tesla K20X), equivalent to a super-computer from ten years ago. Due to this big difference in orders of magnitude, people from the HPC community are moving towards GPU computing. Fig. 2 shows how CPU and GPU performance has improved through the years.

3.5 Performance per Watt

In recent years, power consumption has become more important than brute speedup. Today the notion of *performance per watt*[§] is one of the most important measures for choosing hardware and has been the subject of research [7]. The initiative to develop energy efficient hardware began as a way of doing HPC in a responsible manner. Titan supercomputer runs at the cost of 8.2 MW, offering 2.1 GFlops/W, while a Nvidia Tesla K20X GPU offers 16.8 GFlops/W using 300W. As systems get larger, there is a substantial loss of performance per Watt. The integration of GPUs into supercomputers has helped these systems to be more energy efficient than before.

3.6 Memory Bandwidth

Memory Bandwidth is the rate at which data can be transferred between processors and main memory. It is usually measured as GB/s. The memory efficiency B_c of an implementation is computed by dividing the experimental bandwidth B_e by the maximum

[§]An updated list of the most energy efficient super computers is available at 'www.green500.org'.

bandwidth B_h of the hardware:

$$B_c = \frac{B_e}{B_h}. \tag{3.13}$$

A value of $B_c = 1$ means that the application is using the maximum memory bandwidth available on the hardware. Actual high-end CPUs have a memory bandwidth in the range $40GB/s \leq B_h \leq 80GB/s$ while high-end GPUs have a memory bandwidth in the range $200GB/s \leq B_h \leq 300GB/s$.

Achieving maximum bandwidth in the GPU is much harder than in CPU. The main reason is because memory performance is problem-dependent. Data structures have to be aligned in simple patterns so that many chunks of data are read or written simultaneously with minimal hardware cost. Irregular data accesses, alignments and different data chunk sizes result in lower memory bandwidth. Latest GPU architectures such as *Fermi* and *Kepler* can mitigate this effect by using an L2 cache for global memory (see [29, 92] for more information on the GPU's L2 cache).

All performance measures depend on the running time $T(n,p)$ of the parallel algorithm. Measuring the mean wall-clock time with a standard error below 5% is a good practice for obtaining an experimental value of $T(n,p)$. For the theoretical case, computing $T(n,p)$ is less trivial because it will depend on the chosen *parallel computing model*.

4 Parallel computing models

Computing models are abstract computing machines that allow theoretical analysis of algorithms. These models simplify the computational universe to a small set of parameters that define how much time a memory access or a mathematical operation will cost. Theoretical analysis is fundamental for the process of researching new algorithms, since it can tell us which algorithm is asymptotically better. In the case of parallel computing, there are several models available; *PRAM*, *PMH*, *Bulk parallel processing* and *LogP*. Each one focuses on a subset of aspects, reducing the number of variables so that mathematical analysis is possible.

4.1 Parallel Random Access Machine (PRAM)

The *parallel random access machine*, or PRAM, was proposed by Fortune and Wyllie in 1978 [39]. It is inspired by the classic *random access machine* (RAM) and has been one of the most used models for parallel algorithm design and analysis.

In the 1990s, the PRAM model gained reputation as an unrealistic model for algorithm design and analysis because no computer could offer constant memory access times for simultaneous operations, let alone performance scalability. Implementations of PRAM-designed algorithms did not reflect the complexity the model was suggesting. However, in 2006, the model became relevant again with the introduction of general purpose GPU (GPGPU) computing APIs.

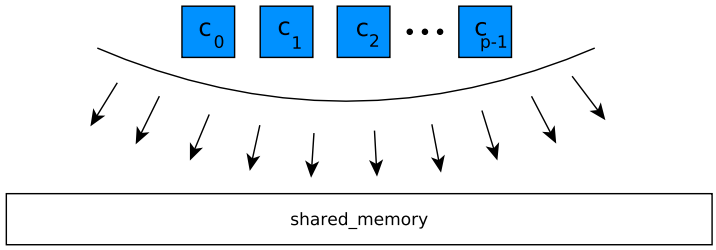

Figure 3: In the PRAM model each one of the cores has a complete view of the global memory.

In the PRAM model, there are p processors that operate synchronously over an unlimited memory completely visible for each processor (see Fig. 3). The p parameter does not need to be a constant number, it can also be defined as a function of the problem size n. Each r/w (read/write) operation costs $\mathcal{O}(1)$. Different variations of the model exist in order to make it more realistic when modeling parallel algorithms. These variations specify whether memory access can be performed *concurrently* or *exclusively*. Four variants of the model exist.

EREW, or *Exclusive Read – Exclusive Write*, is a variant of PRAM where all read an write operations are performed exclusively in different places of memory for each processor. The EREW variation can be used when the problem is split into independent tasks, without requiring any sort of communication. For example, vector addition as well as matrix addition can be done with an EREW algorithm.

CREW, or *Concurrent Read – Exclusive Write*, is a variant of PRAM where processors can read from common sections of memory but always write to sections exclusive to one another. CREW algorithms are useful for problems based on tilings, where each site computation requires information from neighbor sites. Let k be the number of neighbors per site, then each site will perform at least k reads and one write operation. At the same time, each neighbor site will perform the same number of memory reads and writes. In the end, each site is read concurrently by k other sites but only modified once. This behavior is the main idea of a CREW algorithm. Algorithms for fluid dynamics, cellular automata, PDEs and N-body simulations are compatible with the CREW variation.

ERCW, or *Exclusive Read – Concurrent Write*, is a variant of PRAM where processors read from different exclusive sections of memory but write to shared locations. This variant is not as popular as the others because there are less problems that benefit from an ERCW algorithm. Nevertheless, important results have been obtained for this variation. Mackenzie and Ramachandran proved that finding the maximum of n numbers has a lower bound of $\Omega(\sqrt{\log n})$ under ERCW [81], while the problem is $\Theta(\log n)$ under EREW/CREW.

CRCW, or *Concurrent Read – Concurrent Write*, is a variant of PRAM where processors can read and write from the same memory locations. Beame and Hastad have studied optimal solutions using CRCW algorithms [9]. Subramonian [111] presented an $\mathcal{O}(\log n)$ algorithm for computing the minimum spanning tree.

Concurrent writes are not trivial and must use one of the following protocols:

- *Common*: all processors write the same value;
- *Arbitrary*: only one write is successful, the others are not applied;
- *Priority*: priority values are given to each processor (*e.g.*, rank value), and the processor with highest priority will be the one to write;
- *Reduction*: all writes are reduced by an operator (add, multiply, OR, AND, XOR).

Over the last decades, Uzi Vishkin has been one of the main supporters of the PRAM model. He proposed an on-chip architecture based on PRAM [120] as well as the notion of explicit Multi-threading for achieving efficient implementations of PRAM algorithms [121].

4.2 Parallel Memory Hierarchy (PMH)

The *Parallel Memory Hierarchy* model, or PMH, was proposed in 1993 by Alpern *et al.* [4] and inspired by related works [2,3] (HMM and UHM memory models). This model was proposed to deal with the problems of PRAM regarding constant time memory operations. Actual CPUs (such as Intel Xeon E5 series or AMD's Opteron 6000 series) have memory hierarchies composed of registers, L1, L2 and L3 caches. GPUs such as Nvidia GTX 680 or AMD's Radeon HD 7850 also have a memory hierarchy composed of registers, L1, L2 caches and the global memory. Indeed, the memory hierarchy should be considered when designing a parallel algorithm in order to match the theoretical complexity bounds.

The PMH model is defined by a hierarchical tree of memory modules. The leaves of the tree correspond to processors and the internal nodes represent memory modules. Modules closer to the processors are fast, but small, and modules far from the processors are slow, but larger. For the i-th level module, the following parameters are defined; s_i as the number of items per block (or blocksize), n_i the number of blocks, l_i the latency and c_i is the child-count. In practice, it is easier to model an algorithm by using the uniform parallel memory hierarchy (UPMH) which is a simplified version of the PMH model. The UPMH model defines a complete τ-ary tree (see Fig. 4).

In UPMH, composite parameters are used, such as the aspect ratio $\alpha = n_i/s_i$, the packing factor $\rho = s_i/s_{i-1}$ and the branching factor τ which is the tree arity. Additionally, the UPMH model defines the transfer cost t_i as a function of the tree level; $t_i = f(i)$. Typical values of the transfer cost function are $f(i) = 1, i, \rho^i$. Function $f(i) = \rho^i$ is considered a realistic transfer cost function for modern architectures. Usually, the model is referred to as $\text{UPMH}_{\alpha,\rho,f(i),\tau}$ to indicate its four parameters. This model has proven to be more realistic than PRAM, but harder for analyzing algorithms.

Alpern *et al.* showed that an unblocked matrix multiplication algorithm (*i.e.*, the basic matrix multiplication algorithm) can cost $\Omega(N^5/p)$ time instead of $\mathcal{O}(N^3/p)$ [13] as in

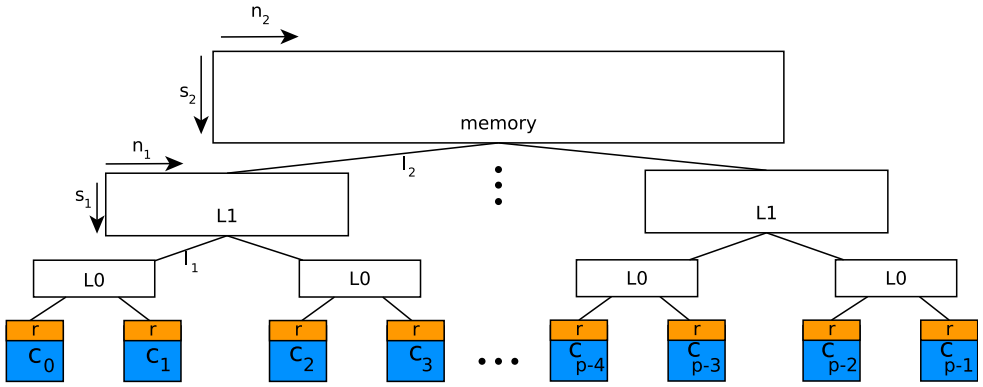

Figure 4: The uniform parallel memory hierarchy tree.

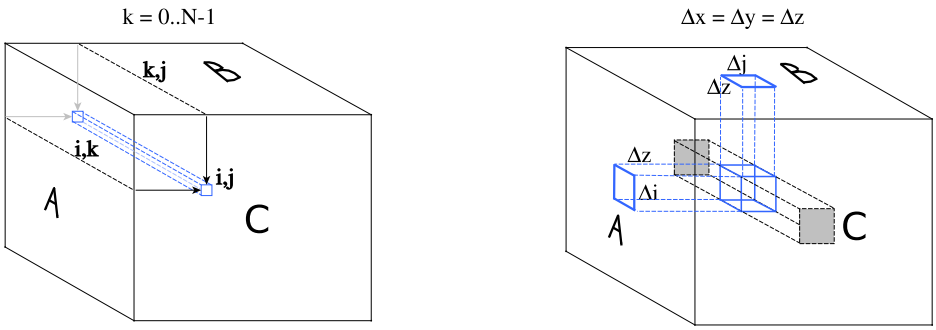

Figure 5: The classic algorithm processes sticks of computation as seen in the left side. The blocked version computes sub-cubes of the domain in parallel, taking advantage of locality.

PRAM. In the same work, the authors prove that a parallel block-based matrix multiplication algorithm (see Fig. 5) can indeed achieve the desired $\mathcal{O}(N^3/p)$ upper bound by reusing the data from the fastest memory modules.

The entire proof of the matrix multiplication algorithm for one processor can be found in the work of Alpern *et al.* [3]. The UPMH model can be considered a complement to other models such as PRAM or BSP.

4.3 Bulk Synchronous Parallel (BSP)

The *Bulk synchronous parallel*, or BSP, is a parallel computing model focused on communication, published in 1990 by Leslie Valiant [119]. Synchronization and communication are considered high priority in the cost equation. The model consists of a number of processors with fast local memory, connected through a network and capable of sending and receiving messages to and from any other processor. A BSP-based algorithm is composed of *super-steps* (see Fig. 6).

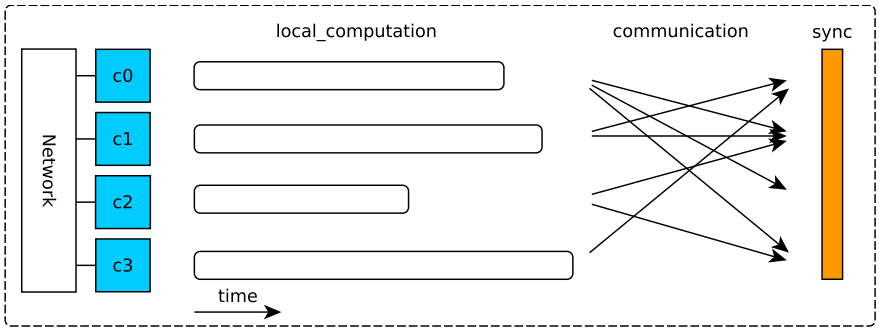

Figure 6: A representation of a super-step; processing, communication and a global synchronization barrier.

A super-step is a parallel block of computation composed of three steps:

- Local computation: p processors perform up to L local computations;
- Global communication: Processors can send and receive data among them;
- Barrier synchronization: Wait for all other processors to reach the barrier.

The cost c for a super-step using p processors is defined as:

$$c = \max_{i=1}^{p}(w_i) + g \max_{i=1}^{p}(h_i) + l, \tag{4.1}$$

where w_i is the computation time of the i-th processor, h_i the number of messages used by the i-th processor, g is the capability of the network and l is the cost of the barrier synchronization. In practice, g and l are computed empirically and available for each architecture as lookup values. For an algorithm composed of S super-steps, the final cost is the sum of all the super-step costs:

$$C = \sum_{i=1}^{S} c_i. \tag{4.2}$$

4.4 LogP

The LogP model was proposed in 1993 by Culler *et al.* [31]. Similar to BSP, it focuses on modeling the cost of communicating a set of distributed processors (*i.e.*, network of computers). In this model, local operations cost one unit of time but the network has latency and overhead. The following parameters are defined:

- *latency (L)*: the latency for communicating a message containing a word (or small number of words) from its source to its target processor;
- *overhead (o)*: the amount of time a processor spends in communication (sending or receiving). During this time, the processor cannot perform other operations;

- *gap (g)*: the minimum amount of time between successive messages in a given processor;
- *processors (P)*: the number of processors.

All parameters, except for the processor count (P), are measured in cycles. Fig. 7 illustrates the model with an example of communication with one-word messages. The LogP model is similar to the BSP with the difference that BSP uses global barriers of synchronizations while LogP synchronizes by pairs of processors. Another difference is that LogP considers a message overhead when sending and receiving. Choosing the right model (BSP or LogP) depends if global or local synchronization barriers are predominant and if the communication overhead is significant or not.

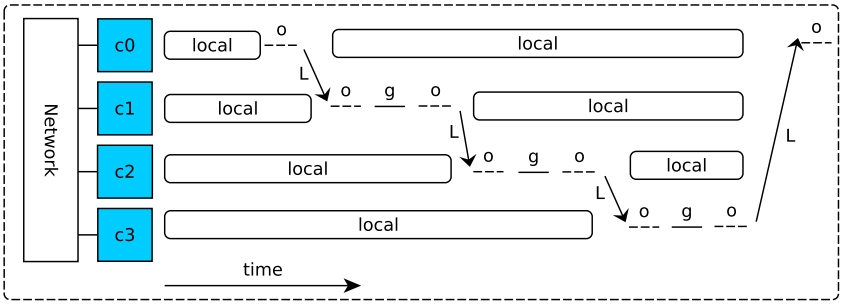

Figure 7: An example communication using the LogP model.

Parallel computing models are useful for analyzing the running time of a parallel algorithm as a function of n, p and other parameters specific to the chosen model. But there are also other important aspects to be considered related to the styles of parallel programming. These styles are well explained by the *parallel programming models*.

5 Parallel programming models

Parallel computing should also be analyzed down how processors communicate and how they are programmed. For example, PRAM and UPMH use the *shared memory* model, while LogP and BSP use a *message passing* model. These two models are actually *parallel programming models*. A *parallel programming model*[¶] is an abstraction of the programmable aspects of a computing model. While computing models from Section 4 are useful for algorithm design and analysis (*i.e.*, computing time complexity), *parallel programming models* are useful for the implementation of such algorithm. The following parallel programming models are the most important because they have been implemented by modern APIs.

[¶]Some of the literature may treat the concept of *parallel programming model* as equal to *computing model*. In this survey we denote a difference between the two; thus Sections 4 and 5.

5.1 Shared memory

In the shared memory model, threads can read and write asynchronously within a common memory. This programming model works naturally with the PRAM computing model and it is mostly useful for *multi-core* and GPU based solutions. A well known API for CPUs is the *Open Multiprocessing* interface or OpenMP [18] which is based on the Unix *pthreads* implementation [90]. In the case of GPUs, OpenCL [65] and CUDA [93] are the most common.

Many times, a shared memory parallel algorithm needs to manage non-deterministic behavior from multiple concurrent threads (the operating system thread scheduling is considered non-deterministic). When concurrent threads read and write on the same memory locations, one must supply an explicit synchronization and control mechanism such as *monitors [54], semaphores [34], atomic operations and mutexes (a binary semaphore)*. These control primitives allow threads to lock and work on shared resources without other threads interfering, making the algorithm consistent. In some scenarios the programmer must also be aware of the *shared memory consistency model*. These models define rules and the strategy used to maintain consistency on shared memory. A detailed explanation of consistency models is available in Adve *et al.* work [1].

For the case of GPUs, one can use atomic operations, synchronization barriers and memory fences [93].

5.2 Message passing

In a message passing programming model, or distributed model, processors communicate asynchronously or synchronously by sending and receiving messages containing words of data. In this model, emphasis is placed on communication and synchronization making distributed computing the main application for the model. Dijkstra introduced many new ideas for consistent concurrency on distributed systems based on exclusion mechanisms [33]. This programming model works naturally with the BSP and LogP models which were built with the same paradigm.

The standard interface for message passing is the *Message Passing Interface* or MPI [41]. MPI is used for handling communication in CPU distributed applications and is also used to distribute the work when using multiple GPUs.

5.3 Implicit

Implicit parallelism refers to compilers or high-level tools that are capable of achieving a degree of parallelism automatically from a sequential piece of source code. The advantage of implicit parallelism is that all the hard work is done by the tool or compiler, achieving practically the same performance as a manual parallelization. The disadvantage however is that it only works for simple problems such as *for* loops with independent iterations. Kim *et al.* [68] describe the structure of a compiler capable of implicit and explicit parallelism. In their work, the authors address the two main problems for

achieving their goal; *(1) how to integrate the parallelizing preprocessor with the code generator* and *(2) when to generate explicit and when to generate implicit threads*.

Map-Reduce [32] is a well known implicit parallel programming tool (sometimes considered a programming model itself) and has been used for frameworks such as Hadoop with outstanding results at processing large data-sets over distributed systems. Functional languages such as Haskell or Racket also benefit from the parallel map-reduce model. In the 90's, High performance Fortran (HPF) [77] was a famous parallel implicit API. OpenMP can also be considered semi-implicit since its parallelism is based on hints given by the programmer. Today, *pH* (parallel Haskell) is probably the first fully implicit parallel programming language [91]. Automatic parallelization is hard to achieve for algorithms not based on simple loops and has become a research topic in the last twenty years [48,74,79,105].

5.4 Algorithmic skeletons

Algorithm skeletons provide an important abstraction layer for implementing a parallel algorithm. With this abstraction, the programmer can now focus more on the strategy of the algorithm rather than on the technical problems regarding parallel programming. Algorithm skeletons, also known as parallelism patterns, were proposed by Cole in 1989 and published in 1991 [24]. This model is based on a set of available parallel computing patterns known as *skeletons* (implemented as higher order functions to receive other functions) that are available to use. The critical step when using algorithmic skeletons is to choose the right pattern for a given problem. The following patterns are some of the most important for parallel computing:

- Farm: or *parallel map*, is a master-slave pattern where a function $f()$ is replicated to many slaves so that slave s_i applies $f(x_i)$ to sub-problem x_i;
- Pipeline: or *function decomposition*, is a staged pattern where $f()_1 -> f()_2 -> \cdots -> f()_n$ are parts of a bigger logic that works as a pipeline. Each stage of the pipeline can work in parallel;
- Parallel tasks: In this pattern, $f()_1, f()_2, \cdots, f()_n$ are different tasks to be performed in parallel. These tasks can run completely independent or can include critical sections;
- Divide and Conquer: This is a recursive pattern where a problem A, a divide function $d : A \to \{a_1, a_2, \cdots, a_k\}$ and a combine function $c : \{a_1, a_2, \cdots, a_k\}$, $f() \to f(\{a_1, a_2, \cdots, a_k\})$ are passed as parameters for the skeleton. Then the skeleton applies a divide and conquer approach spanning parallel branches of computation as the recursion tree grows.

$$r(A,d,c,f) = c(\{r(d(A)_1,d,c), r(d(A)_2,d,c), \cdots, r(d(A)_k,d,c)\}, f), \tag{5.1}$$

$$r(A',d,c,f) = c(d(A'), f). \tag{5.2}$$

The recursion stops when the smallest sub-problems $d(A')$ are reached.

There are also basic skeleton patterns for managing *while* and *for* loops as well as conditional statements. The advantage of algorithmic skeletons is their ability to be combined or nested to make more complex patterns (because they are higher order functions). Their limitation is that the abstraction layers include an overhead cost in performance.

In the previous two sections we covered computing models and programming models which are useful for algorithm analysis and programming, respectively. It is critical however, when implementing a high performance solution, to know how the underlying architecture actually works.

6 Architectures

Computer architectures define the way processors work, communicate and how memory is organized; all in the context of a fully working computer (note, a working computer is an implementation of an architecture). Normally, a computer architecture is well described by one or two computing models. It is important to say that the goal of computer architectures is not to implement an existing computing model. In fact, it is the other way around; computing models try to model actual computer architectures. The final goal of a computer architecture is to make a computer run programs efficiently and as fast as possible. In the past, implementations achieved higher performance automatically because the hardware industry increased the processor's frequency. At that time there were not many changes regarding the architecture. Now, computer architectures have evolved into parallel machines because the single core clock speed has reached its limit in frequency[‖] [104]. Today the most important architectures are the *multi-core* and *many-core*, represented by the CPU and GPU, respectively.

Unfortunately, sequential implementations will no longer run faster by just buying better hardware. They must be re-designed as a parallel algorithm that can scale its performance as more processors are available. Aspects such as the type of instruction/data streams, memory organization and processor communication indeed help for achieving a better implementation.

6.1 Flynn's taxonomy

Computer architectures can be classified by using Flynn's taxonomy [38]. Flynn realized that all architectures can be classified into four categories. This classification depends on two aspects; number of instructions and number of data streams that can be handled in parallel. He ended with four classifications.

SISD, or single instruction single data stream can only perform one instruction to one data stream at a time. There is no parallelism at all. Old single core CPUs of the 1950s, based on the original Von Neumann architecture, were all SISD types. Intel processors from *8086* to *80486* were also SISD.

[‖] Above 4.0 GHz of frequency, silicon transistors can become too hot for conventional cooling systems.

SIMD, or single instruction multiple data streams can handle only one instruction but apply it to many data streams simultaneously. These architectures allow *data parallelism*, which is useful in science for applying a mathematical model to different parts of the problem domain. Vector computers in the 70's and 80's were the first to implement this architecture. Nowadays, GPUs are considered an evolved SIMD architecture because they work with many SIMD batches simultaneously. SIMD has also been supported on CPUs as instruction sets, such as MMX, 3DNow!, SSE and AVX. These instruction sets allow parallel integer or floating point operations over small arrays of data.

MISD, or multiple instruction single data stream can handle different tasks over the same stream of data. These architectures are not so common and often end up being implemented for specific scenarios such as digital attack systems (*e.g.,* to destroy a data encryption) or fault tolerance systems and space flight controllers (NASA).

MIMD, or multiple instruction multiple data streams is the most flexible architecture. It can handle one different instruction for each data stream and can achieve any type of parallelism. However, the complexity of the physical implementation is high and often the overhead involved in handling such different tasks and data streams becomes a problem when trying to scale with the number of cores. Modern CPUs fall into this category (Intel, AMD and ARM *multi-cores*) and newer GPU architectures are partially implementing this architecture. The MIMD concept can be divided into SPMD (single program multiple data) and MPMD (multiple programs multiple data). SPMD occurs when a simple program is being executed in different processors. The key difference compared to SIMD is that in SPMD each processor can be at a different stage of the execution or at different paths of the code caused by conditional branching. MPMD occurs when different independent programs are being run on multiple processors.

6.2 Memory architectures and organizations

There are two forms of memory organization, shared and distributed. In distributed memory, each node has its own memory architecture and it is completely independent from other nodes. Communication is based on messages between nodes through a network. In a distributed memory scenario, the network plays a important role and its topology is different depending on the context. Some common topologies are *bus, star, ring, mesh, hypercube and tree*. Also, hybrid topologies are made based on the basic ones already mentioned.

In a shared memory organization, processors communicate through a common bank of global memory, not needing explicit messages as in a distributed memory scheme. Today, two architectures are mostly used; *UMA* and *NUMA*.

***Uniform Memory Access* or UMA** consists of a shared memory in which the access time for any processor takes the same amount of time no matter the data location. UMA is also known as *Symmetric Multi-Processors* or SMP. The main disadvantage of UMA is the low scalability when increasing the number of processors. This occurs because of the single memory controller shared for all processors.

***Non Uniform Memory Access* or NUMA** is an architecture where access time to shared memory depends on the location of data relative to the processor. This means that the memory that is closer to a processor is accessed much faster than memory closer to another processor (*i.e.*, cost is a function of distance). To take advantage of NUMA, the problem must be split into independent chunks of data, each one assigned to a unique CPU. Also, global *read-only* data is better replicated than shared. In practice, all NUMA architectures implement a hardware cache-coherence logic and become *cache-coherent NUMA* or ccNUMA.

One can find the SMP architecture in many desktop computers with dual core hardware and the NUMA architecture in modern multicore machines with two or more CPU sockets (*e.g.*, AMD Opteron and Intel Xeon). Fig. 8 shows the concepts of UMA and NUMA.

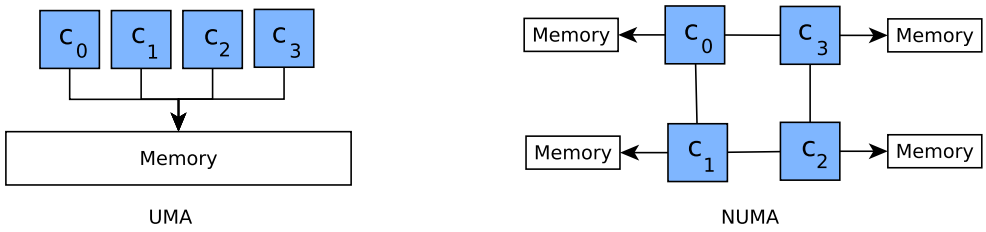

Figure 8: The UMA (*aka* SMP) and NUMA memory architectures.

Finally, shared and distributed memory architectures can also be mixed, leading to a *hybrid configuration* which is useful for MPI + OpenMP or MPI + GPGPU solutions.

6.3 Technical details of modern CPU and GPU architectures

The differences between *multi-core* and *many-core* architectures can be visualized in the schematics of modern CPUs and GPUs.

Actual high-end CPUs, such as the Xeon E5 2600, are built with many interconnections between the cores providing flexibility in communication (see Fig. 9).

Each core has a local L1 and L2 cache of 64KB and 256KB, respectively, and in the center of the chip there is a bigger L3 cache of size 20MB, shared by all cores through a ring scheme [57]. The *Quick-Path Interconnect* or QPI section (known as *Hyper-transport* for AMD processors) of the chip implements part of the NUMA memory architecture. The PCI module handles communication with the PCI ports and finally the *Internal Memory Controller*, or IMC, handles the memory access to its section of RAM, completing the rest of the NUMA architecture.

On the other hand, modern GPUs such as the Tesla K20X have a completely different chip schematic that is oriented to massive parallelism. Fig. 10 shows the schematic of an Nvidia Tesla K20X GPU as well as its actual chip. The cores of the GPU are grouped into

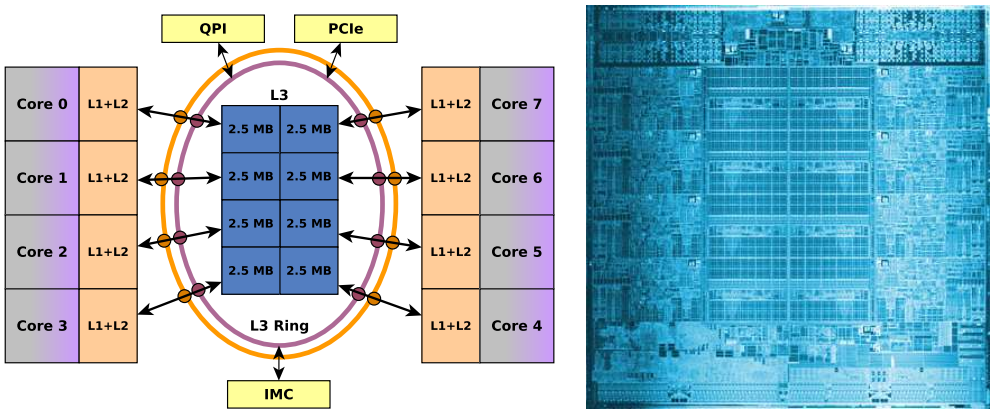

Figure 9: On the left, an Intel Xeon E5 2600 (2012) chip schematic, inspired from [28]. On the right, the processor die [57].

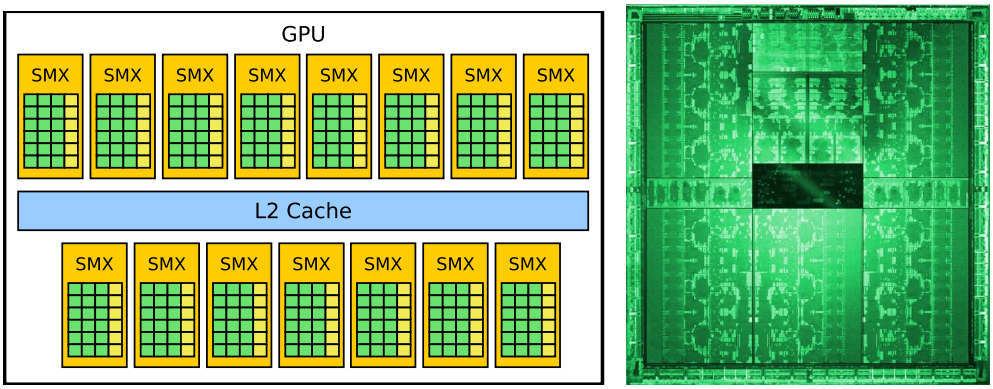

Figure 10: On the left, Nvidia's Tesla K20X GPU schematic. On the right, a picture of its chip [29].

SMX units, or *next generation streaming multiprocessors*. The most important aspects that characterize a GPU are inside the SMX units (see Fig. 11).

A SMX is the smallest unit capable of performing parallel computing. The main difference between a low-end GPU and a high-end GPU of the same architecture is the number of SMX units inside the chip. In the case of Tesla K20 GPUs, each SMX unit is composed of 192 cores (represented by the C boxes). Its architecture was built for a maximum of 15 SMX, giving a maximum of 2,880 cores. However in practice, some SMX are deactivated because of production issues.

The cores of a SMX are 32-bit units that can perform basic integer and single precision (FP32) floating point arithmetic. Additionally, there are 32 special function units or SFU that perform special mathematical operations such as *log*, *sqrt*, *sin and cos*, among others. Each SMX has also 64 double precision floating point units (represented as DPC boxes), known as FP64, and 32 LD/ST units (load / store) for writing and reading memory.

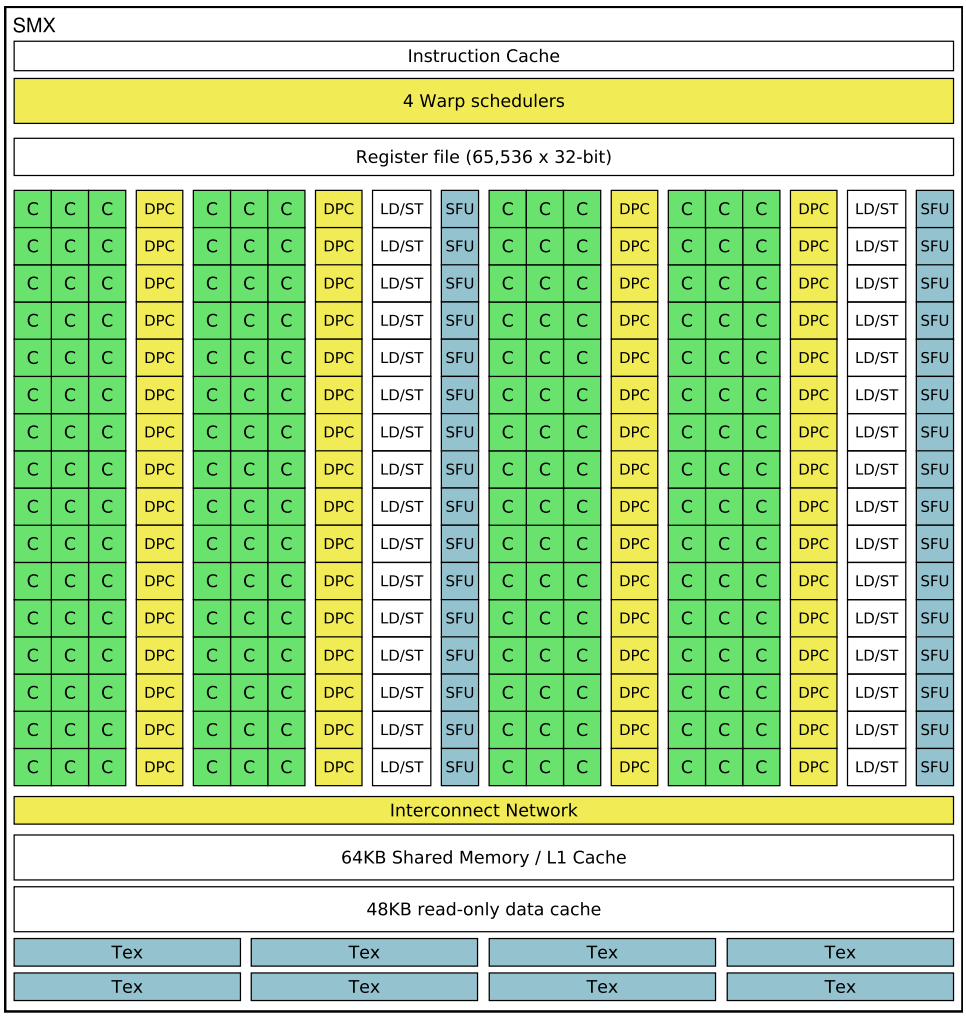

Figure 11: A diagram of a streaming multiprocessor next-generation (SMX). Image inspired from Nvidia's CUDA C programming guide [93].

Numerical performance of GPUs is classified into two categories; FP32 and FP64 performance. The FP32 performance is always greater than FP64 performance. This is actually a problem for massive parallel architectures because they must spend chip surface on special units of computation for increasing FP64 performance. The Tesla K20X GPU can achieve close to 4TFlops of FP32 performance while only 1.1TFlops in FP64 mode.

Actual GPUs such as the Tesla K20 implement a four-level memory hierarchy; (1) registers, (2) L1 cache, (3) L2 cache and (4) global memory. All levels, except for the global memory, reside in the GPU chip. The L2 cache is automatic and it improves memory accesses on global memory. The L1 cache is manual, there is one per SMX, and it can be as fast as the registers. Kepler and Fermi based GPUs have L1 caches of size 64KB that

are split into $16KB$ of programmable shared memory and $48KB$ of automatic cache, or *vice versa*.

6.4 The fundamental difference between CPU and GPU architectures

Modern CPUs have evolved towards parallel processing, implementing the MIMD architecture. Most of their die surface is reserved for control units and cache, leaving a small area for the numerical computations. The reason is, a CPU performs such different tasks that having advanced cache and control mechanisms is the only way to achieve an overall good performance.

On the other hand, the GPU has a SIMD-based architecture that can be well represented by the PRAM and UPMH models (Sections 4.1 and 4.2, respectively). The main goal of a GPU architecture is to achieve high performance through massive parallelism. Contrary to the CPU, the die surface of the GPU is mostly occupied by ALUs and a minimal region is reserved for control and cache (see Fig. 12). Efficient algorithms designed for GPUs have reported up to 100x speedup over CPU implementations [25,78].

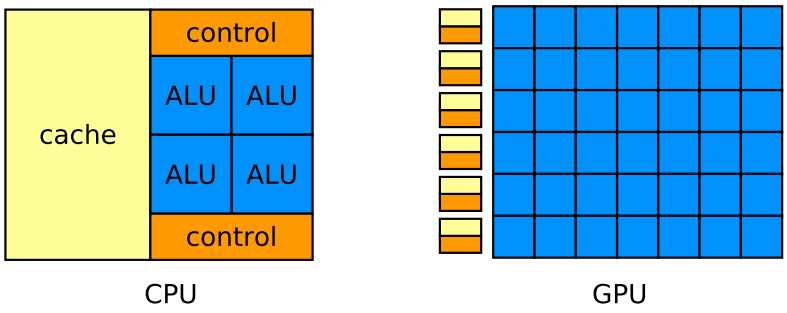

Figure 12: The GPU architecture differs from the one of the CPU because its layout is dedicated for placing many small cores, giving little space for control and cache units.

This difference in architecture has a direct consequence, the GPU is much more restrictive than the CPU but it is much more powerful if the solution is carefully designed for it. Latest GPU architectures such as Nvidia's Fermi and Kepler have added a significant degree of flexibility by incorporating a L2 cache for handling irregular memory accesses and by improving the performance of atomic operations. However, this flexibility is still far from the one found in CPUs.

Indeed there is a trade-off between flexibility and computing power. Actual CPUs struggle to maintain a balance between computing power and general purpose functionality while GPUs aim at massive parallel arithmetic computations, introducing many restrictions. Some of these restrictions are overcome at the implementation phase while some others must be treated when the problem is being parallelized. It is always a good idea to follow a strategy for designing a parallel algorithm.

7 Strategy for designing a parallel algorithm

Designing a new algorithm is not a simple task. In fact, it is considered an art [70,95] that involves a combination of mathematical background, creativity, discipline, passion and probably other unclassifiable abilities. In parallel computing the scenario is no different, there is no golden rule for designing perfect parallel algorithms.

There are some formal strategies that are frequently used for creating efficient parallel algorithms. Leighton and Thomson [76] have contributed considerably to the field by pointing out how data structures, architectures and algorithms relate when facing the act of implementing a parallel algorithm. In 1995, Foster [40] identified a four-step strategy that is present in many well designed parallel algorithms; *partitioning, communication, agglomeration and mapping* (see Fig. 13).

7.1 Partitioning

The first step when designing a parallel algorithm is to split the problem into parallel sub-problems. In *partitioning*, the goal is to find the best partition possible; one that generates the highest amount of sub-problems (at this point, communication is not considered yet).

Identifying the ***domain type*** is critical for achieving a good partition of a problem. If the problem is *data-parallel*, then the data is partitioned and we speak of ***data parallelism***. On the other hand, if the problem is *task-parallel*, then the functionality is partitioned and we speak of ***task-parallelism***. Most of the computational physics problems based on simulations are suitable for a *data-parallelism* approach, while problems such as parallel graph traverse, communication flows, traffic management, security and fault tolerance often fall into the *task-parallelism* approach.

7.2 Communication

After partitioning, communication is defined between the sub-problems (task or data type). There are two types of communication; *local communication* and *global communication*. In local communication, sub-problems communicate with neighbors using a certain geometric or functional pattern. Global communications involve broadcast, reductions or global variables. In this phase, all types of communication problems are handled; from race conditions handled by critical sections or atomic operations, to synchronization barriers to ensure that the strategy of computation is working up to this point.

7.3 Agglomeration

At this point, there is a chance that sub-problems may not generate enough work to become a thread of computation (given a computer architecture). This aspect is often known as the *granularity* of an algorithm [19]. A *fine-grained* algorithm divides the problem into a massive amount of small jobs, increasing parallelism as well as communication

overhead. A *coarse-grained* algorithm divides the problem into less but larger jobs, reducing communication overhead as well as parallelism. *Agglomeration* seeks to find the best level of granularity by grouping sub-problems into larger ones. A parallel algorithm running on a *multi-core* CPU should produce larger agglomerations than the same algorithm designed for a GPU.

7.4 Mapping

Eventually, all agglomerations will need to be processed by the available cores of the computer. The distribution of agglomerations to the different cores is specified by the *mapping*. The *Mapping* step is the last one of Foster's strategy and consists of assigning agglomerations to processors with a certain pattern. The simplest pattern is the 1-to-1 geometric mapping between agglomerations and processors, that is, to assign agglomeration k_i to processor p_i. Higher complexity mapping patterns lead to higher hardware overhead and unbalanced work. The challenge is to achieve the most balanced and simple patterns for complex problems.

Fig. 13 illustrates all four steps using a data-partition based problem on a dual core architecture (c_0 and c_1).

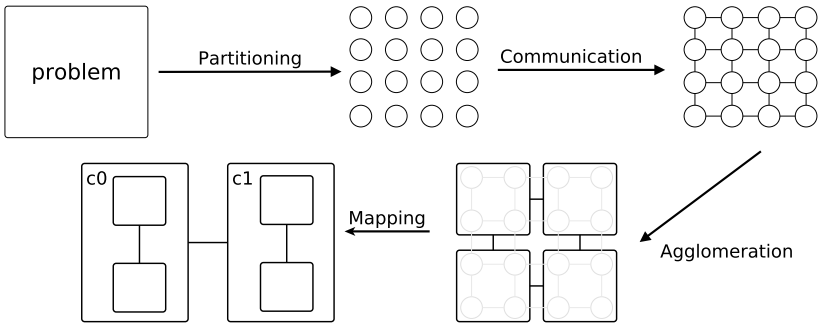

Figure 13: Foster's diagram of the design steps used in a parallelization process.

Foster's strategy is well suited for computational physics because it handles data-parallel problems in a natural way. At the same time, Foster's strategy also works well for designing massive parallel GPU-based algorithms. In order to apply this strategy, it is necessary to know how the massive parallelism programming model works for mapping the computational resources to a data-parallel problem and how to overcome the technical restrictions when programming the GPU.

8 GPU Computing

GPU computing is the utilization of the GPU as a general purpose unit for solving a given problem, unrestricted to the graphical context. It is also known by its acronym; GPGPU

(General-Purpose computing on Graphics Processing Units). The goal of GPU computing is to achieve the highest performance for data-parallel problems through a massive parallel algorithm that is run on the GPU.

GPU computing started as a research field for computer graphics (CG) in the early 2000s and gained high importance as a general purpose parallel processor [80]. In 2001, for the first time the graphics processing unit was built upon a programmable architecture, permitting programmable lighting [22, 64, 67], shadow [23] and geometry [107] effects to be computed and rendered in real-time. These effects were achieved using a high level shading language such as GLSL (OpenGL Shading Language) [83], HLSL (High-level Shading Language) [94] and CG (*C* for Graphics) [82]. At that time, the massive parallelism paradigm was already in the minds of the CG researchers who were designing per-vertex and per-fragment algorithms to work in a set of millions of primitives. As the years passed, the scientific community became interested in the power of GPUs and its low cost compared to other solutions (clusters, super-computers). However, adapting a scientific problem to a graphics environment was hard and challenging from the technical side. In the early days, the act of adapting different kinds of problems to the GPU was considered as *hacking the GPU*.

In 2002, McCool *et al.* published a paper detailing a meta-programming GPGPU language, named *Sh* [85]. In 2004, Buck *et al.* proposed *Brook for GPUs*, also known as *Brook-GPU* [15]. This was an extension of the C language that allowed general purpose programming on programmable GPUs. Both *Sh* and *Brook-GPU* played a fundamental role in expanding the idea of GPU computing by hiding the graphical context of shading languages.

In the year 2006 another general purpose GPU computing API was released. This time by Nvidia and named CUDA (Compute Unified Device Architecture) [93]. Technically, the CUDA API is an extension of the C language and compiles general purpose code to be executed on the GPU (based on the shared memory programming model). The release of CUDA became an important milestone in the history of GPU computing because it was the first API that offered effective documentation for getting started in the field. The CUDA acronym refers to the general purpose architecture of Nvidia's GPUs [29], suitable for GPU computing. At the moment, only Nvidia GPUs support CUDA.

In the year 2008, an open standard was released with the name of OpenCL (Open Computing Language), allowing the creation of multi-platform, massively parallel code [65]. Its programming model is similar to that of CUDA but uses different names for the same structures. The programming model behind CUDA and OpenCL is a key aspect for GPU computing because it defines several components that are essential for implementing a massively parallel algorithm.

8.1 The massive parallelism programming model

The programming models explained in Section 5 are necessary but not sufficient for understanding the programming model of the GPU. There are important aspects regarding

thread and memory organization that are relevant to the implementation of a GPU-based algorithm. This section covers these aspects.

The GPU programming model is characterized by its high level of parallelism, thus the name *Massive parallelism programming model*. This model is an abstract layer that lies on top of the GPU's architecture. It allows the design of massive parallel algorithms independent of how many physical processing units are available or how execution order of threads is scheduled.

The abstraction is achieved by the *space of computation*, defined as a discrete space where a massive amount of threads are organized. In CUDA, the space of computation is composed of a *grid*, *blocks* and *threads*. For OpenCL, it is *work-space*, *work-group* and *work-item*, respectively. A *grid* is a discrete k-dimensional (with $k=1,2,3$) box type structure that defines the size and volume of the space of computation. Each element of the grid is a *block*. Blocks are smaller k'-dimensional (with $k'=1,2,3$) structures identified by their coordinate relative to the grid. Each block contains many spatially organized *threads*. Finally, each thread has a coordinate relative to the block for which it belongs. This coordinate system characterizes the space of computation and serves to map the threads to the different locations of the problem. Fig. 14 illustrates an example of two-dimensional space of computation. Each block has access to a small local memory, in CUDA it is known as the *shared memory* (in OpenCL it is known just as the *local memory*). In practical terms, the shared memory works as a manual cache. It is important to make good use of this fast memory in order to achieve peak performance of the GPU.

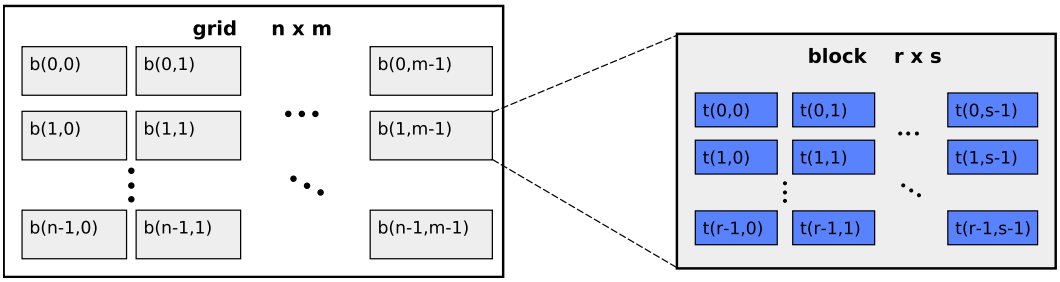

Figure 14: Massive parallelism programming model presented as a 2D model including grid, blocks and threads. Image inspired from Nvidia's CUDA C programming guide [93].

The programming work-flow of GPU computing is viewed as a *host-device* relationship between the CPU and GPU, respectively. A host program (*e.g.*, a C program) uploads the problem into the device (GPU memory), and then invokes a *kernel* (a function written to run on the GPU) passing as parameter the grid and block size. The host program can work in a synchronous or asynchronous manner, depending if the result from the GPU is needed for the next step of computation or not. When the kernel has finished in the GPU, the result data is copied back from device to host. Fig. 15 summarizes the work-flow.

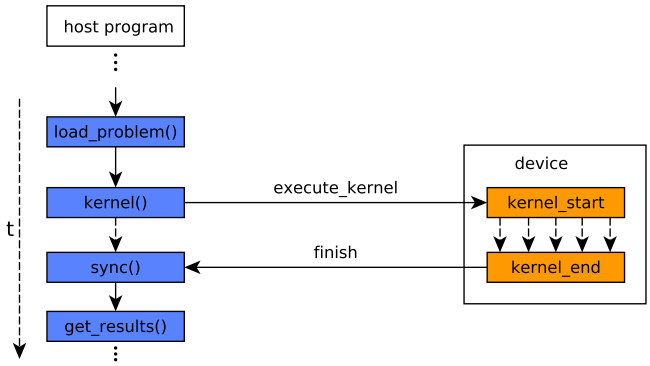

Figure 15: The GPU's main function, named *kernel*, is invoked from the CPU host code.

8.2 Thread managing and GPU concurrency

Actual GPUs manage threads in small groups that work in SIMD mode. For AMD GPUs, these groups are known as *wavefronts* and its size is 64 threads for their actual GCN (graphics core next) architecture. For Nvidia GPUs, these groups are known as *warps* and the actual architectures such as Fermi and Kepler work with a size of 32 threads. The OpenCL standard uses a more descriptive name; *SIMD width*. For simplicity reasons, we will refer to these groups as *warps*.

Both AMD's and Nvidia's GPUs support some degree of concurrency for handling the entire space of computation. Most of the time, there will be more threads than what can really be processed in parallel. While all threads are in progress (concurrency), only a subset are really working in parallel. The maximum number of parallel threads running on a GPU normally corresponds to the number of processing units. However, the maximum number of concurrent threads is much higher. For example, the Geforce GTX 580 GPU can process up to 512 threads in parallel, but can handle up to 24,576 concurrent threads. For most problems, it is recommended to overflow the parallel computing capacity. The reason is that the GPU's thread scheduler is smart enough to switch idle warps (*i.e.*, warps that are waiting a memory access or a special function unit result, such as *sqrt()*) with new ones ready for computation. In other words, there is a small pipeline of numerical computation and memory accesses that the scheduler tries to maintain busy all the time.

8.3 Technical considerations for a GPU implementation

The GPU computing community frequently uses the terms *coalesced memory*, *thread coarsening*, *padding* and *branching*. These are the names of the most critical technical considerations that must be taken into account in order to achieve the best performance on the GPU.

Coalesced memory refers to a desired scenario where consecutive threads access consecutive data chunks of 4, 8 or 16 bytes long. When this access pattern is achieved, mem-

ory bandwidth increases, making the implementation more efficient. In every other case, memory performance will suffer a penalty. Many algorithms require irregular access patterns with crossed relations between the chunks of data and threads. These algorithms are the hardest to optimize for the GPU and are considered great challenges in HPC research [95].

Thread coarsening is the act of reducing the fine-grained scheme used on a solution by increasing the work per thread. As a result, the amount of registers per block increases allowing to re-use more computations saved on the registers. Choosing the right amount of work per thread normally requires some experimental tuning.

Padding is the act of adjusting the problem size on each dimension, n_d, into one that is multiple of the block size; $n'_d = \rho\lceil n_d/\rho \rceil$ (where ρ is the number of threads per block per dimension) so that now the problem fits tightly in the grid (the block size per dimension is a multiple of the warp size). An important requirement is that the extra dummy data must not affect the original result. With padding, one can avoid putting conditional statements in the kernel that would lead to unnecessary branching.

Branching is an effect caused when conditional statements in the kernel code lead to sequential execution of the *if* and *else* parts. It has a negative impact in performance and should be avoided whenever possible. The reason why branching occurs is because all threads within a warp execute in a lock-step mode and will run completely in parallel only if they follow the same execution path in the kernel code (SIMD computation). If any conditional statement breaks the execution into two or more paths, then the paths are executed sequentially. Conditionals can be safely used if one can guarantee that the program will follow the same execution path for a whole warp. Additionally, tricks such as *clamp, min, max, module* and *bit-shifts* are hardware implemented, cause no branching and can be used to evade simple conditionals.

9 Examples of spatial and tiled GPU compatible problems

The last three sections have covered different aspects of parallel computing with special emphasis on GPU computing. In several instances we have mentioned that computational physics is a field that can benefit greatly from GPU-based algorithms because many of its problems are *data-parallel*. The following subsections will describe four examples of computational physics problems that benefit from GPU computing because of data-parallelism.

9.1 The n-body problem

The *n-body problem*[**] is an interaction problem where n bodies are affected by gravity in a k-dimensional space (usually $k=1,2,3$). The problem is relevant for this survey because

[**]Other non-astrophysical problems can also be solved with an n-body simulation. In this case we refer to the astrophysical n-body problem.

it is one of the most emblematic *data-parallel* problems found in computational physics [58], thus an ideal candidate for GPU computing. Additionally, some of the algorithmic challenges present in the n-body problem, such as partitioning space, are generic for every spatial interaction problem.

The n-body problem states that for each particle q_i, its force F_i is given by its interactions with the other $n-1$ particles:

$$F_i = G\sum_{k\neq i}\frac{m_i m_k(q_k-q_i)}{|q_k-q_i|^3}, \quad i=1,\cdots,n. \tag{9.1}$$

The gravitational constant is $G=6.67*10^{-11}N(\frac{m}{kg})^2$.

Forces can be computed simultaneously for all particles. The trick is to save the resulting positions in a different array so that no synchronization is needed. The naive solution to this problem requires a $\mathcal{O}(n^2)$ algorithm; for each particle, sum up the contributions of the other $n-1$. A GPU solution for this algorithm can indeed be achieved by splitting the problem domain into one job per particle; resulting in a $\mathcal{O}(\frac{n^2}{p})$ algorithm (where p is the amount of parallelism provided by the GPU). The book GPU Gems 3 includes a chapter for the GPU-based implementation of the $\mathcal{O}(n^2)$ algorithm [89]. But GPU algorithms should not stop on the simplest parallelization. As with the sequential case, GPU algorithms can also improve by using more advanced data-structures such as trees.

A faster hierarchical algorithm can be achieved by sacrificing some numerical precision. In Eq. (9.1), one can see that the contributions of the summation decrease quadratically as the distance from q_k to q_i increases. As a result, particles very far from q_i will make no significant contribution to F_i. On the other hand, particles close to q_i will make most of the contribution. This analysis on the significance of contributions as a function of distance is the key for designing a faster n-body algorithm.

The Barnes-Hut tree-code algorithm [6] is a well known spatial partitioning solution that achieves $\mathcal{O}(n\log_k n)$ average time per time step. It uses a 2^k-tree known as *quad-tree* for $k=2$ and *oct-tree* for $k=3$. The tree data-structure is used for storing average measures for cluster of points far from the reference point. The bigger the distance from the reference point, the bigger the cluster (See Fig. 16). The algorithm works as follows; each internal node of the tree structure contains k children, its position to the center of the sub-space contained and an average measure of the contained particles. Computing the interaction of each particle with position $\vec{p}_i$ requires to traverse the tree starting from the root. At each internal node with average position n_j, the following value is computed:

$$c=\frac{s}{|p_i-n_j|}, \tag{9.2}$$

where s is the length of the sub-region contained by node n_i. If $c>\theta$, then the node is close enough and it is necessary to look into the children of n_j recursively. If $c\leq\theta$, then all the children particles are far enough for n_j and the average contribution is used, stopping there. The value $\theta\in[0,1]$ corresponds to the accuracy of the simulation. Lower values of θ

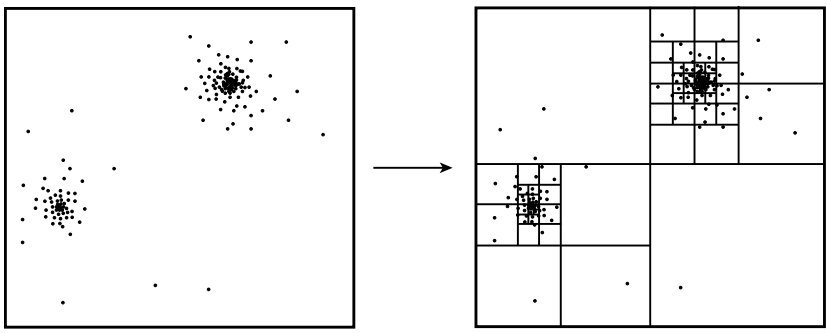

Figure 16: A galaxy simulation (left) and its quad-tree structure computed (right).

lead to faster, but less accurate simulations and higher θ leads to slower, but more precise simulations.

A parallelization of the algorithm is done by first building the tree and then assigning m bodies to each of the p processors such that $n=mp$. Each parallel computation uses the tree as read-only to obtain the total quantity of force applied to it (a CREW algorithm). The cost of the algorithm under the PRAM model is $\mathcal{O}((n\log_k n)/p)=\mathcal{O}(m\log_k n)$. In the best possible case, when $p=n$ processors, then $\mathcal{O}(\log_k n)$. Bedorf *et al.* solution [10] implements all parts of the algorithm in the GPU, achieving up to 20x speedup compared to a sequential execution. The n-body problem has also been solved with hierarchical algorithms running on supercomputers with thousands of GPUs [126]. The authors do not report any speedup value, however they report up to 1.0 Pflops with an efficiency of 74%, all using 4096 GPUs.

9.2 Collision detection

At a first glance, collision detection can be seen as equivalent to the n-body problem but it is not. In a collision detection problem, the goal is to detect pairs of collisions rather than computing a magnitude for each body as in n-body. This problem is important for computational physics because it is fundamental for the simulation of rigid bodies [53]. Performing collision detection on the GPU brings up the opportunity to achieve real-time simulation and visualization for many interactive problems that are based on collisions.

The collision detection problem is defined as: *given n bodies at a time step t, compute all collisions pairs where bodies intersect, then perform an action in step* $t+1$ *for each colliding object*. Similar to the n-body problem, a brute force $\mathcal{O}(n^2)$ algorithm is sufficient but not necessary. Practical case shows that bodies collide with their closest neighbors before anything else; this gives the opportunity for a faster algorithms.

The kd-tree is a spatial binary tree for partitioning a k-dimensional space in two parts recursively. Kd-trees do not partition around a point as the 2^k-tree used in the n-body simulation but instead they partition along a plane, forcing the tree to remain binary at every level. A partition can also be non centered, see Fig. 17. Search costs $\mathcal{O}(\log n)$ time,

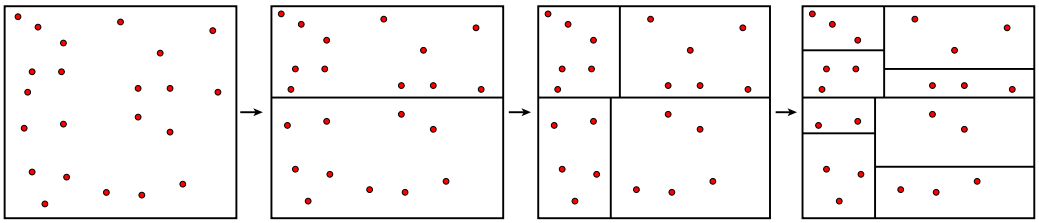

Figure 17: Building a 2D kd-tree. Partitions can be non-centered for better balance.

making an improvement on the brute force $\mathcal{O}(n)$ search. Before any algorithm begins, the kd-tree is built in $\mathcal{O}(n\log n)$ time using a linear-time median-finding algorithm [27]. Then, the problem can be solved with a parallel divide and conquer strategy; start with a single thread at the root of the kd-tree and recursively spawn two threads that will access the two children of the tree. Since we are in the massive parallelism programming model, such an amount of threads is indeed possible and will not become a problem. When a thread hits a leaf of the tree, it will check the collisions inside that region of space. Since that region will have a constant number of elements (*e.g.*, less than 5), the computation will cost $\mathcal{O}(1)$. The cost of an algorithm of this type is theoretically $\mathcal{O}(\log n/p)$ because all branches are computed in parallel using p processors. If $p\approx n$, then $T(n,p)\approx\mathcal{O}(\log n)$. This problem can also be solved with an 2^k-tree in a similar way as the N-body problem.

Today, parallel divide and conquer algorithms can be implemented fully in modern GPU architectures like *GK110* from Nvidia. A survey on collision detection methods is available in [60] and GPU implementations in [69] and [96].

9.3 Probabilistic Potts model simulations

The Potts model is a spin based model and one of the most important in statistical mechanics [125]. It allows the study of phase transitions for different lattice topologies. Exact methods are characterized for being NP-hard [123] and intractable. Instead, Monte Carlo based simulations are often used because of their polynomial time.

There most popular Monte Carlo algorithm for the Potts model is the Metropolis algorithm [86]. The main idea is that each spin uses only neighborhood information (no global data), making each computation independent and fully parallelizable. The Metropolis algorithm is relevant for GPU computing because it introduces the concept of *stencil computation*. A stencil is a fixed pattern defined along the problem domain, in this case the lattice. *Stencil computation* means to apply a kernel to each element of the stencil using just the neighborhood information. Stencil computations are *data-parallel* problems, and therefore, good candidates for a massive parallel solution on the GPU.

When parallelizing the Metropolis algorithm, the partition is done at a particle level, that is, every particle is an independent job. Communication is only local; read from neighbor particles. Agglomeration will be high for a CPU simulation and low for a GPU-based one. The mapping is 1-to-1 for a square lattice because each agglomeration is as-

signed to a different thread. The theoretical complexity of such a parallel algorithm under the PRAM model using the CREW variation is $\mathcal{O}(n/p)$ per step.

The algorithm starts with a random initial state $\sigma_{t=0}$ for the lattice and defines a transition from σ_t to σ_{t+1} as follows: for each vertex $v_i \in V$ with spin value $s_i \in [1 \cdots q]$, compute the present and future energies $h(s_i^t)$ and $h(r)$, respectively (with $r = rand(0 \cdots q)$ and $r \neq s_i^t$). The spin energy is computed by summing up the contributions from the neighborhood $< s_i >$:

$$h(s_i) = -J \sum_{s_k \in < s_i >} \delta_{s_k, s_i}. \tag{9.3}$$

Once $h(s_i^t)$ and $h(r)$ are computed, the new value s_i^{t+1} is chosen with a probability based on $\Delta h = h(r) - h(s_i^t)$:

$$P(s_i^{t+1} \Rightarrow r) = \begin{cases} 1 & \text{if } \Delta h \leq 0, \\ e^{\frac{-\Delta h}{\kappa T}} & \text{if } \Delta h > 0. \end{cases} \tag{9.4}$$

When $\Delta h \leq 0$, the new value for the spin becomes the random value r with full probability of success. If $\Delta h > 0$, the new value of the spin may become r with probability $e^{\frac{-\Delta h}{\kappa T}}$ (with κ being Boltzmann constant), otherwise it remains the same (see Fig. 18).

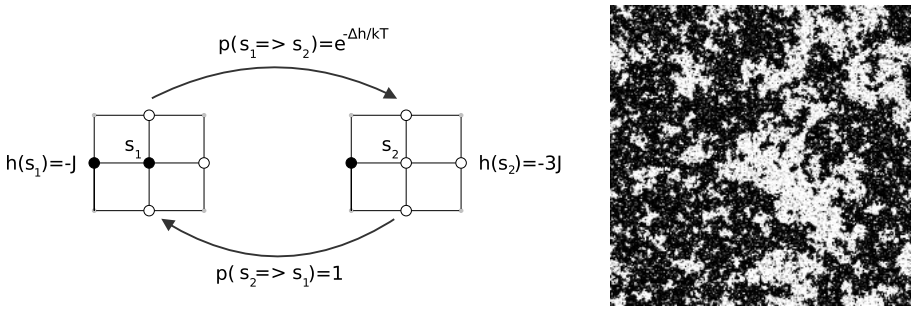

Figure 18: On the left, the probability function. On the right, an example of an evolved lattice using $q = 2$ spin states per site [21].

The simulation stops when the lattice reaches thermodynamic equilibrium. In practice, equilibrium occurs when σ_{t+1} is similar to σ_t.

Achieving massive parallel Potts model simulations using GPUs is currently a research topic [102,117]. Ferrero et al. have proposed a GPU-based version of the Metropolis algorithm [37].

Apart from the Metropolis algorithm, there also exist cluster based algorithms for the Potts model such as the multi-cluster and single cluster algorithms proposed by Swendsen *et al.* [113] and Wolff [124], respectively. These algorithms are not local, instead they generate random clusters of spins identified by a common label (in Wolff algorithm, it is just one cluster). The idea is to flip the clusters to a new random state. The advantage of cluster algorithms is that they do not suffer from auto-correlations and converge

much faster to the equilibrium than Metropolis. However, they present additional challenges for a parallelization. Komura *et al.* have proposed GPU based versions of such algorithms [71,72].

9.4 Cellular Automata simulation

Cellular Automata (CA) were first formulated by the hands of John Von Neumann, Konrad Zuse and Stanislaw Ulam in the 1940s (see [88, 122]). CA are important for GPU computing because they are inherently data-parallel dynamic systems based on *stencil computations*. CA are used for the simulation of galaxy formation, typhoon propagation, fluid dynamics [61,118], biological dynamics [45], social interactions, and theoretical parallel machines. In general, it can simulate any problem that can be reduced to a system of cells interacting under a local rule. It has been proven that even the simplest CA are capable of being *Turing complete* machines [26].

CA are defined as a dynamical discrete space, where each cell c_i^t has one of k possible states at time t. The computation of the next state c_i^{t+1} is given by:

$$c_i^{t+1} = f(< c_i >^t), \tag{9.5}$$

where $f(< c_i >^t)$ is the *local transition function* or *local rule* applied to the neighborhood $< c_i >$. The challenge of CA research is to discover new rules that could simulate events of our universe and help in the quest of understanding how complex and chaotic systems work.

In a CA simulation, every cell can compute its next state for time step $t+1$ independently by only using the information from its neighborhood (including itself) in time step t. It is important to note that along the simulation, some cells are in *quiescent state*, which means that they will remain in that same state as long as the neighborhood is in quiescent state too (in some CA this is considered as the "dead" state).

A simple parallel algorithm can assign p processors to the n different automatons and perform one step of simulation in a theoretical time of $\mathcal{O}(\frac{n}{p})$ under the PRAM model using the CREW variation. In the ideal case, with $p \approx n$, one step of evolution would take $\mathcal{O}(1)$ time. Fig. 19 shows a time-space evolution of a known one-dimensional CA, the rule 161, as well as a snapshot of *John Conway's game of life* [42] in its present state. Both images were taken from a GPU application implemented by the authors.

An efficient algorithm is one that only processes non-quiescent cells, otherwise there is a substantial waste of computation per time step. Processing b non-quiescent cells with p processors is at least $\Omega(b/p)$. In many CA, space is much bigger than the number of non-quiescent cells, therefore $b \ll n$. For these cases, the space can be treated as a dimensional sparse matrix. However, adapting such a strategy for a massive parallel solution is not simple and problems such as access patterns and neighbor data layout must be solved.

Related work has been focused on solving the main challenges for efficient GPU implementations and comparing performance against multicore solutions (CPU) [59, 106].

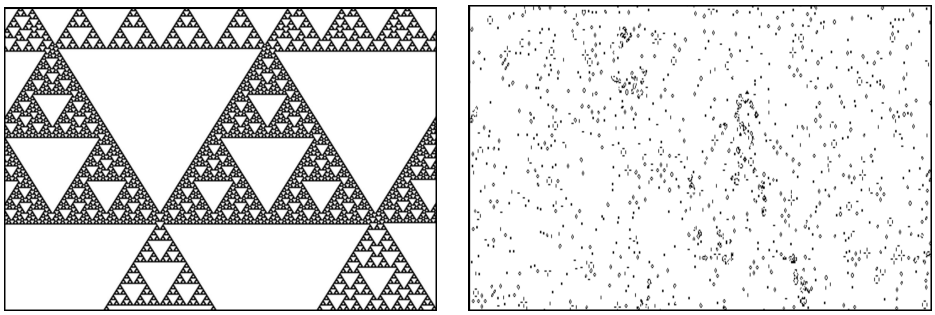

Figure 19: On the left, the time-space of the elementary CA 161. On the right, John Conway's *game of life*.

Tran *et al.* have identified the main challenges of Cellular Automata simulation on GPU [59]. Another interesting topic is to efficiently simulate CA working in other topologies such as hexagonal or triangular [8,43] where the GPU's *space of computation* does not match exactly with the problem domain. Furthermore, doing CA simulations on non-Euclidean topologies is an even more complex problem [130].

10 Latest advances and open problems in GPU computing

In the field of computational physics, $O(n)$ cost algorithms (the fast multi-pole expansion method [127–129]) have been implemented on GPU for n-body simulations. Single GPU implementations have been proposed for achieving high performance Potts model simulations [116, 117] even with biological applications [20]. Multi-GPU based implementations have also been proposed for the Potts model [72] and for n-body simulations [126]. In a multi-GPU scenario, two levels of parallelism are used; *distributed* and *local*. Distributed parallelism is in charge of doing a coarse grained partition of the problem, the mapping of sub-problems to computer nodes and the communication across the super-computer or cluster. Local parallelism is in charge of solving a sub-problem independently with a single GPU. Multi-GPU based algorithms have the advantage of computing solutions to large scale problems that cannot fit in a single machine's memory. The main challenge for multi-GPU methods is to achieve efficient distributed parallelism (*e.g.*, hiding data communication cost by overlapping communication with computation).

Cellular Automata are now being used as a model for fast parallel simulation of physical phenomena, traffic simulation and image segmentation, among others [36, 44, 63, 73]. In the field of computer graphics, new algorithms have been proposed for building kd-trees or oct-trees in GPU to achieve real-time ray-tracing [56, 62, 131] as well as real-time methods for 3D reconstruction and level set segmentation [46, 103]. The field of programming languages have contributed to parallel computing by proposed high level parallel languages for the programmer (*i.e.*, to abstract the programmer so that the job of partitioning, communication, agglomeration and mapping is part of the compiler or framework [17, 112]). Tools for automatically converting CPU code into GPU code are

now becoming popular [55] and useful for fields that use parallelism at a high level tool and not as a goal of their research. In a more theoretical level, a new GPU-based computational model has also been proposed; the K-model [16] which serves for analyzing GPU-based algorithms.

Architectural advances in parallel computing have focused on combining the best of the CPU and GPU worlds. Parallel GPU architectures are now making possible massive parallelism by using thousands of cores, but also with flexible work-flows, access patterns and efficient cache predictions. The latest GPU architectures have included *dynamic parallelism* [29]; a feature that consists of making it possible for the GPU to schedule additional work for itself by using a *command processor*, without needing to send data back and forth between host and device. This means that recursive hierarchical partition of the domain will be possible on the fly, without needing the CPU to control each step. Lastly, one of the most important revolutions in computer architecture (affecting parallel computing directly) is the Hybrid Memory Cube (HMC) [114] project. HMC is a three-dimensional memory architecture that promises 15$\times$ better performance than DDR3 memory, requiring 70% less energy per bit.

There are still open problems for GPU computing. Most of them exist because of the actual limitations of the massive parallelism model. In a parallel SIMD architecture, some data structures do not work so well. Tree implementations have been solved partially for the GPU, but data structures such as classic *dynamic arrays, heaps, hash tables and complex graphs* are not performance-friendly yet and need research for efficient GPU usage. Another problem is the fact that some sequential algorithms are so complex that porting them to a parallel version will lead to no improvement at all. In these cases, a complete redesign of the algorithm must be done. The last open problem we have identified is the act of mapping the space of computation (*i.e.,* the grid of blocks) to different kinds of problem domains (*i.e.,* geometries). A naive space of computation can always build a bounding box around the domain and discard the non useful blocks of computation. Non Euclidean geometry is a special case that illustrates this problem; finding an efficient map for each block of the grid to the fractal problem domain is not trivial. The only way of solving this problem is to find an efficient mapping function from the space of computation to the problem domain, or modify the problem domain so that it becomes similar to an Euclidean box. With *dynamic parallelism*, it will be possible for the first time to build fractal spaces of computation for non-Euclidean problems, increasing efficiency.

11 Discussion

Over the past 40 years, parallel computing has evolved significantly from being a matter of high equipped data centers and supercomputers to almost every electronic device that uses a CPU or GPU. Today, the field of parallel computing is having one of its best moments in history of computing and its importance will only grow as long as computer architectures keep evolving to a higher number of processors. Speedup and efficiency are

the most important measures in a parallel solution and will continue to be in the following years, especially efficiency, now that power consumption is a serious matter for every branch of science. However, to achieve such desired levels of performance, implementations need to be aware of the underlying architecture. The PRAM model is one of the most popular theoretical models but it lacks the considerations of the different memory hierarchies that are present in every computer architecture today. When using the shared memory programming model, it is important to combine the PMH and PRAM model so that the implementation behaves like the theoretical bounds. The BSP and LogP models are useful when designing distributed algorithms where communication is a critical aspect. As with PMH and PRAM, BSP and LogP can be combined for better modeling of a distributed algorithm. That is, one can include message overheads, gaps and global synchronization barriers all in one problem.

Automatic parallelization is now supported by some high level languages such as *pH*. Also, modern APIs can parallelize loop sections of programs by just adding a directive in the program code (*e.g.,* OpenMP). This trend will continue to evolve, each time including more and more patterns to the implicit parallelization programming model so that in the end only complex problems of science and engineering will need manually optimized parallelization.

GPU computing is a young branch of parallel computing that is growing each year. People from the HPC community have realized that many problems can be solved with energy-efficient hardware using the massive parallelism paradigm. For some problems, GPU implementations have reported impressive speedups of up to 100x; results accepted by most, though questioned by Lee *et. al.* [75]. The cost of achieving such extreme performance is the complex re-design. For some problems, a port from the sequential version is not enough and a re-design of the algorithm is needed in order to use the massive parallel capabilities of the GPU architecture. In general, data-parallel problems are well suited for a massive parallel execution in GPU; that is, problems where the domain can be split into several independent sub-problems. The split process can be an arbitrary distribution or a recursive divide and conquer strategy. Time dependent problems with little work to be done at each time step are not well suited for GPU computing and will lead to inefficient performance. Examples are the computation of small dynamical systems, numerical series, iterative algorithms with small work per iterations and graph traverse when the structure has only a few connections.

It is important to note how the natural-scientific and human-type problems differ in the light of parallel computing. Science problems benefit from parallelism by using a data-parallel approach without sacrificing complexity as a result. This is because many physical phenomena follow the idea of a dynamic system where simple rules can exhibit complex behavior in the long term. On the other side, human-type problems are similar to a complex graph structure with many connections and synchronization problems. Such level of complexity in communication make parallel algorithms perform slowly.

There is still much work to be done in the field of parallel computing. The challenge for massive parallel architectures in the following years is for them to become more flexi-

ble and energy efficient. At the same time, the challenge for computer science researchers will be to design more efficient algorithms by using the features of these new architectures.

Acknowledgments

Special thanks to Conicyt for funding the PhD program of the first author Cristóbal A. Navarro and to the reviewers of *Communications in Computational Physics* Journal for improving the quality of this survey. This work was supported by Fondecyt Project No. 1120495. Finally, thanks to Renato Cerro for improving the English of this manuscript.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Computer, 29(12):66–76, December 1996.

[2] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In Proceedings of the nineteenth annual ACM symposium on Theory of computing, STOC '87, pages 305–314, New York, NY, USA, 1987. ACM.

[3] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. Algorithmica, 12:72–109, 1994. 10.1007/BF01185206.

[4] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In In Proc. Programming Models for Massively Parallel Computers, pages 116–123. IEEE Computer Society Press, 1993.

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the April 18-20, 1967, spring joint computer conference, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[6] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. Nature, 324(6096):446–449, December 1986.

[7] L. A. Barroso. The price of performance. Queue, 3(7):48–53, September 2005.

[8] C. Bays. Cellular automata in triangular, pentagonal and hexagonal tessellations. In Robert A. Meyers, editor, Computational Complexity, pages 434–442. Springer New York, 2012.

[9] P. Beame and J. Hastad. Optimal bounds for decision problems on the crcw pram. In In Proceedings of the 19th ACM Symposium on Theory of Computing (New, pages 25–27. ACM.

[10] J. Bédorf, E. Gaburov, and S. P. Zwart. A sparse octree gravitational n-body code that runs entirely on the GPU processor. J. Comput. Phys., 231(7):2825–2839, April 2012.

[11] A. Bernhardt, A. Maximo, L. Velho, H. Hnaidi, and M.-P. Cani. Real-time terrain modeling using cpu-GPU coupled computation. In Proceedings of the 2011 24th SIBGRAPI Conference on Graphics, Patterns and Images, SIBGRAPI '11, pages 64–71, Washington, DC, USA, 2011. IEEE Computer Society.

[12] Z. Bittnar, J. Kruis, J. Němeček, B. Patzák, and D. Rypl. Civil and structural engineering computing: 2001. chapter Parallel and distributed computations for structural mechanics: a review, pages 211–233. Saxe-Coburg Publications, 2001.

[13] L. Carter B. Alpern. The ram model considered harmful towards a science of performance programming, 1994.

[14] C. P. Breshears. The Art of Concurrency – A Thread Monkey's Guide to Writing Parallel Applications. O'Reilly, 2009.
[15] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. ACM Trans. Graph., 23(3):777–786, August 2004.
[16] G. Capannini, F. Silvestri, and R. Baraglia. K-model: A new computational model for stream processors. In Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications, HPCC '10, pages 239–246, Washington, DC, USA, 2010. IEEE Computer Society.
[17] B. L. Chamberlain. Chapel (cray inc. hpcs language). In Encyclopedia of Parallel Computing, pages 249–256. 2011.
[18] B. Chapman, G. Jost, and R. van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press, 2007.
[19] D.-K. Chen, H.-M. Su, and P.-C. Yew. The impact of synchronization and granularity on parallel systems. SIGARCH Comput. Archit. News, 18(3a):239–248, May 1990.
[20] N. Chen, J. A. Glazier, J. A. Izaguirre, and M. S. Alber. A parallel implementation of the cellular potts model for simulation of cell-based morphogenesis. Computer Physics Communications, 176(11-12):670–681, 2007.
[21] P. Coddington. Visualizations of spin models of magnetism, online at `http://cs.adelaide.edu.au/ paulc/physics/spinmodels.html`, August 2013.
[22] F. Cohen, P. Decaudin, and F. Neyret. GPU-based lighting and shadowing of complex natural scenes. In Siggraph'04 Conf. DVD-ROM (Poster), August 2004. Los Angeles, USA.
[23] M. Colbert and J. Křivánek. Real-time dynamic shadows for image-based lighting. In ShaderX 7 - Advanced Rendering Technicques. Charles River Media, 2009.
[24] M. Cole. Algorithmic skeletons: structured management of parallel computation. MIT Press, Cambridge, MA, USA, 1991.
[25] A. Colic, H. Kalva, and B. Furht. Exploring nvidia-cuda for video coding. In Proceedings of the first annual ACM SIGMM conference on Multimedia systems, MMSys '10, pages 13–22, New York, NY, USA, 2010. ACM.
[26] M. Cook. Universality in Elementary Cellular Automata. Complex Systems, 15(1):1–40, 2004.
[27] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. Introduction to Algorithms. McGraw-Hill Higher Education, 2nd edition, 2001.
[28] Intel Corporation. IntelR XeonR Processor E5-2600 Product Family Uncore Performance Monitoring Guide, 2012.
[29] Nvidia Corporation. Kepler Whitepaper for the GK110 architecture, 2012.
[30] E. Scheihing, C. A. Navarro, N. Hitschfeld-Kahler. A GPU-based method for generating quasi-delaunay triangulations based on edge-flips. In Proceedings of the 8th International on Computer Graphics, Theory and Applications, GRAPP 2013, pages 27–34, February 2013.
[31] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. SIGPLAN Not., 28(7):1–12, July 1993.
[32] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. Commun. ACM, 51(1):107–113, January 2008.
[33] E. W. Dijkstra. Solution of a problem in concurrent programming control. Commun. ACM, 8(9):569–, September 1965.

[34] N. Dunstan. Semaphores for fair scheduling monitor conditions. SIGOPS Oper. Syst. Rev., 25(3):27–31, May 1991.
[35] V. Faber, O. M. Lubeck, and A. B. White, Jr. Superlinear speedup of an efficient sequential algorithm is not possible. Parallel Comput., 3(3):259–260, July 1986.
[36] N. Ferrando, M. A. Gosalvez, J. Cerda, R. G. Girones, and K. Sato. Octree-based, GPU implementation of a continuous cellular automaton for the simulation of complex, evolving surfaces. Computer Physics Communications, pages 628–640, 2011.
[37] E. E. Ferrero, J. P. De Francesco, N. Wolovick, and S. A. Cannas. q-state potts model metastability study using optimized GPU-based monte carlo algorithms. Computer Physics Communications, 183(8):1578 – 1587, 2012.
[38] M. J. Flynn. Some computer organizations and their effectiveness. IEEE Trans. Comput., 21(9):948–960, September 1972.
[39] S. Fortune and J. Wyllie. Parallelism in random access machines. In Proceedings of the tenth annual ACM symposium on Theory of computing, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.
[40] I. Foster. Designing and building parallel programs: Concepts and tools for parallel software engineering. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
[41] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In Proceedings, 11th European PVM/MPI Users' Group Meeting, pages 97–104, Budapest, Hungary, September 2004.
[42] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". Scientific American, 223:120–123, October 1970.
[43] S. Gobron, H. Bonafos, and D. Mestre. GPU accelerated computation and visualization of hexagonal cellular automata. In Proceedings of the 8th international conference on Cellular Automata for Reseach and Industry, ACRI '08, pages 512–521, Berlin, Heidelberg, 2008. Springer-Verlag.
[44] S. Gobron, A. Çöltekin, H. Bonafos, and D. Thalmann. GPGPU computation and visualization of three-dimensional cellular automata. The Visual Computer, 27(1):67–81, 2011.
[45] S. Gobron, F. Devillard, and B. Heit. Retina simulation using cellular automata and GPU programming. Mach. Vision Appl., 18(6):331–342, November 2007.
[46] S. Gobron, C. Marx, J. Ahn, and D. Thalmann. Real-time textured volume reconstruction using virtual and real video cameras. In proceedings of the Computer Graphics International 2010 conference, 2010.
[47] R. Greenlaw, J. H. Hoover, and W. L. Ruzzo. Limits to Parallel Computation: P-Completeness Theory. Oxford University Press, USA, April 1995.
[48] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. Int. J. Parallel Program., 28(6):537–562, December 2000.
[49] J. L. Gustafson. Reevaluating Amdahl's law. Communications of the ACM, 31:532–533, 1988.
[50] J. L. Gustafson. Fixed time, tiered memory, and superlinear speedup. In In Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5, 1990.
[51] J. L. Gustafson. The consequences of fixed time performance measurement. In Proceedings of the 25th Hawaii International Conference on Systems Sciences, IEEE Computer Society, 1992.
[52] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 tflops hierar-

chical *n*-body simulations on GPUs with applications in both astrophysics and turbulence. In SC, 2009.

[53] T. Harada. Real-time rigid body simulation on GPUs. In Hubert Nguyen, editor, GPU Gems 3, pages 611–632. Addison-Wesley, 2008.

[54] C. A. R. Hoare. Monitors: an operating system structuring concept. Commun. ACM, 17(10):549–557, October 1974.

[55] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between cpu and GPU. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.

[56] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree GPU raytracing. In Proceedings of the 2007 symposium on Interactive 3D graphics and games, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[57] M. Huang, M. Mehalel, R. Arvapalli, and S. He. An energy efficient 32nm 20 MB L3 cache for IntelR XeonR processor E5 family. In CICC, pages 1–4. IEEE, 2012.

[58] L. Ivanov. The *n*-body problem throughout the computer science curriculum. J. Comput. Sci. Coll., 22(6):43–52, June 2007.

[59] D. Luebke J. Tran, D. Jordan. New challenges for cellular automata simulation on the GPU. Technical Report MSU-CSE-00-2, Virginia University, 2003.

[60] P. Jimenez, F. Thomas, and C. Torras. 3d collision detection: A survey. Computers and Graphics, 25:269–285, 2000.

[61] S. F. Judice, B. Barcellos, S. Coutinho, and G. A. Giraldi. Lattice methods for fluid animation in games. Comput. Entertain., 7(4):56:1–56:29, January 2010.

[62] S. Kashyap, R. Goradia, P. Chaudhuri, and S. Chandran. Implicit surface octrees for ray tracing point models. In Proceedings of the Seventh Indian Conference on Computer Vision, Graphics and Image Processing, ICVGIP '10, pages 227–234, New York, NY, USA, 2010. ACM.

[63] C. Kauffmann and N. Piche. Seeded nd medical image segmentation by cellular automaton on GPU. Int. J. Computer Assisted Radiology and Surgery, 5(3):251–262, 2010.

[64] J. Kautz, W. Heidrich, and H.-P. Seidel. Real-time bump map synthesis. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, HWWS '01, pages 109–114, New York, NY, USA, 2001. ACM.

[65] Khronos OpenCL Working Group. The OpenCL Specification, version 1.0.29, 8 December 2008.

[66] D. B. Kidner, P. J. Rallings, and J. A. Ware. Parallel processing for terrain analysis in GIS: Visibility as a case study. Geoinformatica, 1(2):183–207, August 1997.

[67] M. J. Kilgard. A practical and robust bump-mapping technique for todays GPUs. Nvidia, 2000.

[68] S. W. Kim and R. Eigenmann. The structure of a compiler for explicit and implicit parallelism. In Proceedings of the 14th international conference on Languages and compilers for parallel computing, LCPC'01, pages 336–351, Berlin, Heidelberg, 2003. Springer-Verlag.

[69] P. Kipfer. LCP algorithms for collision detection using CUDA. In Hubert Nguyen, editor, GPUGems 3, pages 723–739. Addison-Wesley, 2007.

[70] D. E. Knuth. Computer programming as an art. Commun. ACM, 17(12):667–673, December 1974.

[71] Y. Komura and Y. Okabe. GPU-based single-cluster algorithm for the simulation of the ising model. J. Comput. Phys., 231(4):1209–1215, February 2012.

[72] Y. Komura and Y. Okabe. Multi-GPU-based swendsenVwang multi-cluster algorithm for the simulation of two-dimensional -state potts model. Computer Physics Communications, 184(1):40 – 44, 2013.

[73] P. Korček, L. Sekanina, and O. Fučik. Cellular automata based traffic simulation accelerated on GPU. In Proceedings of the 17th International Conference on Soft Computing (MENDEL2011), pages 395–402. Institute of Automation and Computer Science FME BUT, 2011.

[74] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. SIGPLAN Not., 42(6):235–244, June 2007.

[75] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. cpu myth: an evaluation of throughput computing on cpu and GPU. SIGARCH Comput. Archit. News, 38(3):451–460, June 2010.

[76] F. T. Leighton. Introduction to parallel algorithms and architectures: array, trees, hypercubes. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[77] D. Loveman. High performance Fortran. IEEE Parallel & Distributed Technology: Systems & Applications, 1(1):25–42, 1993.

[78] P. Lu, H. Oki, C. Frey, G. Chamitoff, L. Chiao, E. Fincke, C. Foale, S. Magnus, W. McArthur, D. Tani, P. Whitson, J. Williams, W. Meyer, R. Sicker, B. Au, M. Christiansen, A. Schofield, and D. Weitz. Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the international space station. Journal of Real-Time Image Processing, 5:179–193, 2010. 10.1007/s11554-009-0133-1.

[79] X. Ma, J. Li, and N. F. Samatova. Automatic parallelization of scripting languages: Toward transparent desktop parallel computing. In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1–6, 2007.

[80] M. Macedonia. The GPU enters computing's mainstream. Computer, 36(10):106–108, 2003.

[81] P. D. Mackenzie and V. Ramachandran. ERCW PRAMs and optical communication. In in Proceedings of the European Conference on Parallel Processing, EUROPAR 96, pages 293–302, 1996.

[82] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. ACM Trans. Graph., 22(3):896–907, July 2003.

[83] R. Marroquim and A. Maximo. Introduction to GPU programming with glsl. In Proceedings of the 2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing, SIBGRAPI-TUTORIALS '09, pages 3–16, Washington, DC, USA, 2009. IEEE Computer Society.

[84] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. In Michael Paterson, editor, Automata, Languages and Programming, volume 443 of Lecture Notes in Computer Science, pages 729–743. Springer Berlin / Heidelberg, 1990. 10.1007/BFb0032070.

[85] M. D. McCool, Z. Qin, and T. S. Popa. Shader metaprogramming. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[86] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. J. Chem. Phys., 21:1087, 1953.

[87] A. S. Mikhayhu. Embarrassingly Parallel. Tempor, 2012.

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

[89] H. Nguyen. GPU gems 3. Addison-Wesley Professional, first edition, 2007.
[90] B. Nichols, D. Buttlar, and J. P. Farrell. Pthreads Programming. O'Reilly, 101 Morris Street, Sebastopol, CA 95472, 1998.
[91] R. Nikhil and Arvind. Implicit Parallel Programming in pH. Morgan Kaufmann, May 2001.
[92] Nvidia. Fermi Compute Architecture Whitepaper.
[93] Nvidia-Corporation. Nvidia CUDA C Programming Guide, 2012.
[94] M. Oneppo. Hlsl shader model 4.0. In ACM SIGGRAPH 2007 courses, SIGGRAPH '07, pages 112–152, New York, NY, USA, 2007. ACM.
[95] S. Openshaw and I. Turton. High Performance Computing and the Art of Parallel Programming: An Introduction for Geographers, Social Scientists, and Engineers. Routledge, New York, NY, 10001, 1999.
[96] S. Pabst, A. Koch, and W. Straßer. Fast and scalable CPU/GPU collision detection for rigid and deformable surfaces. Computer Graphics Forum, 29(5):1605–1612, 2010.
[97] D. A. Padua, editor. Encyclopedia of Parallel Computing, volume 4. Springer, 2011.
[98] M. Pagani and P. Tranquilli. Parallel reduction in resource lambda-calculus. In APLAS, pages 226–242, 2009.
[99] D. Parkinson. Parallel efficiency can be greater than unity. Parallel Computing, 3(3):261 – 262, 1986.
[100] H. A. Peelle. To teach Newton's square root algorithm. SIGAPL APL Quote Quad, 5(4):48–50, December 1974.
[101] V. P. Plagianakos, N. K. Nousis, and M. N. Vrahatis. Locating and computing in parallel all the simple roots of special functions using pvm. J. Comput. Appl. Math., 133(1-2):545–554, August 2001.
[102] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. GPU accelerated monte carlo simulation of the 2d and 3d ising model. J. Comput. Phys., 228(12):4468–4477, July 2009.
[103] M. Roberts, J. Packer, M. C. Sousa, and J. R. Mitchell. A work-efficient GPU algorithm for level set segmentation. In Proceedings of the Conference on High Performance Graphics, HPG '10, pages 123–132, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.
[104] P. E. Ross. Why cpu frequency stalled. IEEE Spectr., 45(4):72–72, April 2008.
[105] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In In Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 72–83, 1999.
[106] S. Rybacki, J. Himmelspach, and A. M. Uhrmacher. Experiments with single core, multicore, and GPU based computation of cellular automata. In Proceedings of the 2009 First International Conference on Advances in System Simulation, SIMUL '09, pages 62–67, Washington, DC, USA, 2009. IEEE Computer Society.
[107] P. V. Sander and J. L. Mitchell. Progressive buffers: view-dependent geometry and texture lod rendering. In Proceedings of the third Eurographics symposium on Geometry processing, SGP '05, Aire-la-Ville, Switzerland, Switzerland, 2005. Eurographics Association.
[108] A. Di Serio and M. B. Ibáñez. Evaluation of a nearest-neighbor load balancing strategy for parallel molecular simulations in mpi environment. In PVM/MPI, pages 226–233, 2002.
[109] Y. Shiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. J. Algorithms, 3(1):57–67, 1982.
[110] J. R. Smith. The design and analysis of parallel algorithms. Oxford University Press, Inc., New York, NY, USA, 1993.
[111] R. Subramonian. An o(log n) time common CRCW PRAM algorithm for minimum span-

ning tree. Technical Report UCB/CSD-92-673, EECS Department, University of California, Berkeley, Mar 1992.
[112] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. Gramps: A programming model for graphics pipelines. ACM Trans. Graph., 28(1):4:1–4:11, February 2009.
[113] R. H. Swendsen and J. S. Wang. Nonuniversal, critical dynamics in Monte Carlo simulations. Phys. Rev. Lett., 58:86, 1987.
[114] N. Tanabe, N. Hori, B. Nuttapon, and H. Nakajo. Preliminary evaluations for hybrid memory cube with gather functions using FPGA. IPSJ SIG Notes, 2012(6):1–10, 2012-03-19.
[115] D. Taniar, C. H. C. Leung, W. Rahayu, and S. Goel. High-Performance Parallel Database Processing and Grid Databases. Wiley Series on Parallel and Distributed Computing, 2008.
[116] J. J. Tapia and R. D'Souza. Data-parallel algorithms for large-scale real-time simulation of the cellular Potts model on graphics processing units. 2009 IEEE International Conference on Systems Man and Cybernetics, (10):1411–1418, 2009.
[117] J. J. Tapia and R. D'Souza. Parallelizing the cellular potts model on graphics processing units. Computer Physics Communications, 182(4):857–865, 2011.
[118] P. Topa and P. Mlocek. GpGPU implementation of cellular automata model of water flow. In Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics - Volume Part I, PPAM'11, pages 630–639, Berlin, Heidelberg, 2012. Springer-Verlag.
[119] L. G. Valiant. A bridging model for parallel computation. Commun. ACM, 33(8):103–111, August 1990.
[120] U. Vishkin. A pram-on-chip vision (invited abstract). In SPIRE, page 260, 2000.
[121] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit multi-threading (XMT) bridging models for instruction parallelism (extended abstract). In SPAA, pages 140–151, 1998.
[122] J. von Neumann. The general and logical theory of automata. In Cerebral Mechanisms in Behaviour. Wiley, 1951.
[123] G. J. Woeginger. Combinatorial optimization - eureka, you shrink! chapter Exact algorithms for NP-hard problems: a survey, pages 185–207. Springer-Verlag New York, Inc., New York, NY, USA, 2003.
[124] U. Wolff. Collective Monte Carlo updating for spin systems. Physical Review Letters, 62:361–364, 1989.
[125] F. Y. Wu. The Potts model. Reviews of Modern Physics, 54(1):235–268, January 1982.
[126] R. Yokota, L. Barba, T. Narumi, and K. Yasuoka. Scaling fast multipole methods up to 4000 GPUs. In Proceedings of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing: Does Asia Lead the Way?, ATIP '12, pages 9:1–9:6, Singapore, Singapore, 2012. A*STAR Computational Resource Centre.
[127] R. Yokota and L. A. Barba. Fast n-body simulations on GPUs. CoRR, abs/1108.5815, 2011.
[128] R. Yokota and L. A. Barba. A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems. CoRR, abs/1106.2176, 2011.
[129] R. Yokota and L. A. Barba. Hierarchical n-body simulations with autotuning for heterogeneous systems. Computing in Science and Engineering, 14(3):30–39, 2012.
[130] S. Yukita. Cellular automata in non-euclidean spaces. In Proceedings of the 7th WSEAS International Conference on Mathematical Methods and Computational Techniques In Electrical Engineering, MMACTE'05, pages 200–207, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS).
[131] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. ACM Trans. Graph., 27(5):126:1–126:11, December 2008.