INFO335
High Performance Computing

Magíster en Informática
Universidad Austral de Chile

Cristóbal A. Navarro

July 21, 2021

Contents

Introduction

For some computational problems, sequential algorithms are not fast enough to provide a solution in a reasonable[1] amount of time. Problems such as these can be found in natural sciences [52, 106, 99] (Physics, Biology, Chemistry), information technologies [105] (IT), geospatial information systems [61, 10] (GIS), structural mechanics problems [11] and even abstract mathematical/computer science (CS) problems [100, 93, 90, 76], among other fields. In many cases, these problems can be solved within a reasonable amount of time with the use of a parallel algorithm, expanding the possibilities of research for the given field.

In the past, the only way to run parallel algorithms was by building a cluster of computers or by having exclusive access to a super-computer. The first attempt on building a parallel machine at human-scale came only when the silicon of conventional CPUs could not reach higher frequencies due to physical constraints. At that moment, the computer industry was forced for the first time to expand the chip's architecture by adding multiple *cores* that would work independently one to each other, thus increasing the performance through parallelism. From that point and on, the CPU evolved to the known multi-core CPU which is a flexible and parallel processor. Over the years, the computer science community noted that for problems with critical regions and complex memory patterns, the multi-core CPU algorithms worked efficiently. But there were other type of problems, very parallelizable, for which the multi-core CPU architecture was not performing as fast as required. These problems are known as *data-parallel* problems and they characterize for having a high number of sub-problems that grow as a function of the problem size. For these types of problems one can use massively parallel processors and achieve a higher performance.

The story of how massive parallel processors were born is an interesting one because it combines two fields that were thought to be unrelated; *computational science* and *video-game industry*. The constant need for solving larger scientific problems eventually led to the construction of *super-computers* for understanding phenomena such as galaxy formation, molecular dynamics and climate change, among many others. As the scientific community expanded, the need for high performance computers that could be cheaper, accessible and smaller became important. On the other hand, the video-game industry has the goal of achieving real-time photo-realistic graphics, with the major restriction of running their lighting and polygon algorithms on consumer-level computer hardware. The need of realistic video-games led to the invention of the graphics accelerator, which is a small parallel processor that handles millions of floating point computations per second. The two needs, combined together, gave birth to the modern GPU.

The high compute power of GPUs, combined with the flexible parallelism of multi-core CPUs, can produce over one Teraflops of performance in a workstation machine, making research on high performance methods accessible by the computer science community world-wide. A great portion of modern research on parallel computing is devoted to study the possibilities of GPU computing when applied to scientific problems, finding out that a considerable amount of speedup can be obtained with respect to a sequential CPU-based solution [9, 31], sometimes reaching over an order of magnitude of speedup[70, 27].

Some problems however, cannot have a parallel solution [46]. For example, the approximation of $\sqrt{x}$ using the Newton-Rhapson method [92] 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 an efficient sequential algorithm. On the other hand, there are problems that can be naturally split into many independent sub-problems; *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 parallel computing. In some cases, these problems become so parallelizable that they receive the name *embarrassingly parallel*[2] or *pleasingly parallel* [79, 89].

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*

[1]The notion of *reasonable* varies for each scientific field.

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

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?. The answers to these questions are indeed important when searching for a high performance solution and they 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 parallel algorithm.

This chapter is a comprehensive survey of basic and advanced topics that are often required as parallel computing background. The reader should become more confident in the fundamental and technical aspects of parallel CPU and GPU computing, with a clear idea of what types of problems are best suited for each architecture.

Chapter 1: Fundamental Concepts

Concurrency and Parallelism

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 [13];

Definition 1.0.1 ***Concurrency*** *is a property of a program (at design level) where two or more tasks can be in progress simultaneously.*

Definition 1.0.2
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 1.0.3 ***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:

$$D = d_1 + d_2 + ... + d_k = \sum_{i=1}^{k} d_i \tag{1.1}$$

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

$$f(D) = f(d_1) + f(d_2) + ... + f(d_k) = \sum_{i=1}^{k} f(d_i) \tag{1.2}$$

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:

$$D(S) = d_1(S) + d_2(S) + ... + d_k(S) = \sum_{i=1}^{k} d_i(S) \tag{1.3}$$

Data-parallel problems are ideal candidates for the GPU since its 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 because its architecture allows different tasks to be executed with flexible memory access patterns. Identifying the amount of data-parallelism and task-parallelism in a problem 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, as the chosen problems for this thesis, thus they are good candidates for a parallelization with multi-core CPUs and GPUs.

Performance measures

Performance measures consist of a set of metrics that can be used for quantifying the quality of an algorithm. For sequential algorithms, *time* and *space* give a rich amount of information about the algorithm, and in many occasions it is 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:

$$T(n,p) \tag{1.4}$$

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

Work and Span

The quality of a parallel algorithm can be defined by two metrics as stated by Cormen *et al.* [29]; *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 additional work from the sequential algorithm. By doing less additional 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* states the first lower bound:

$$T(n,p) \geq \frac{T(n,1)}{p} \tag{1.5}$$

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 $T(n,p)$:

$$T(n,p) \geq T(n,\infty) \tag{1.6}$$

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.

Speedup

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

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

$$S_p = \frac{T_s(n,1)}{T(n,p)} \tag{1.7}$$

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 (1.5):

$$S_p \leq p \tag{1.8}$$

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 of *ideal speedup* or *perfect linear speedup*. It is the maximum theoretical value of speedup a parallel algorithm can achieve when n is

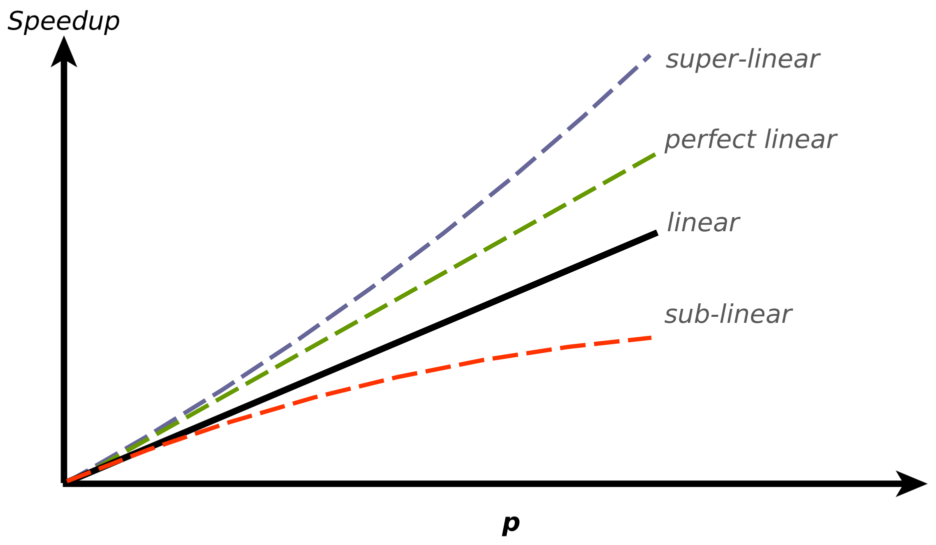

Figure 1.1: The four possible curves for speedup.

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, $T(n,p) \geq T_s(n,1)/p$. Figure 1.1 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 or not. 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 [101] and Faber *et al.* [37] 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 [91] 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 equation (1.7). 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 equation (1.7). Amdahl and Gustafson proposed each one an expression for computing partial speedup. They are known as the *laws of speedup*.

Amdahl's law

Let c be the fraction 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:

$$S(p) = \frac{1}{(1-c) + \frac{c}{p}} \tag{1.9}$$

If $p \approx \infty$, equation (1.9) becomes:

$$S(p) = \frac{1}{1-c} \tag{1.10}$$

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*.

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.

$$T(p) = s + c \tag{1.11}$$

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

$$S(p) = \frac{s + cp}{s + c} = \frac{s}{s + c} + \frac{cp}{s + c} \tag{1.12}$$

Defining α as the fraction of serial computation $\alpha = s/(s + c)$, then the parallel fraction is $1 - \alpha = c/(s + c)$. Finally, equation (1.12) becomes the *fixed-time speedup* $S(p)$:

$$S(p) = \alpha + p(1 - \alpha) = p - \alpha(p - 1) \tag{1.13}$$

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.

Efficiency

If we divide expression (1.8) by p, we get:

$$E_p = \frac{S_p}{p} = \frac{T_s(n, 1)}{pT(n, p)} \leq 1 \tag{1.14}$$

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).

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:

$$F_c = \frac{F_e}{F_h} \tag{1.15}$$

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.

In the year 2014, the fastest floating point performance reported was approximately 33.8 PFLOP/s by the *Tianhe-2* supercomputer located in the National Super Computer Center in Guangzhou, China. A list of the 500 most powerful super-computers in the world is kept updated each year at the site 'www.top500.org'. 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, which can be up to an order of magnitude faster than a CPU, the goal of Exaflops scale will be achieved. Figure 1.2 shows the performance between Nvidia's GPUs and Intel CPUs through the years. From the Figure one can see that the GPU performance is approximately five times higher for double precision (FP64) and almost to 10 times higher for single precision (FP32).

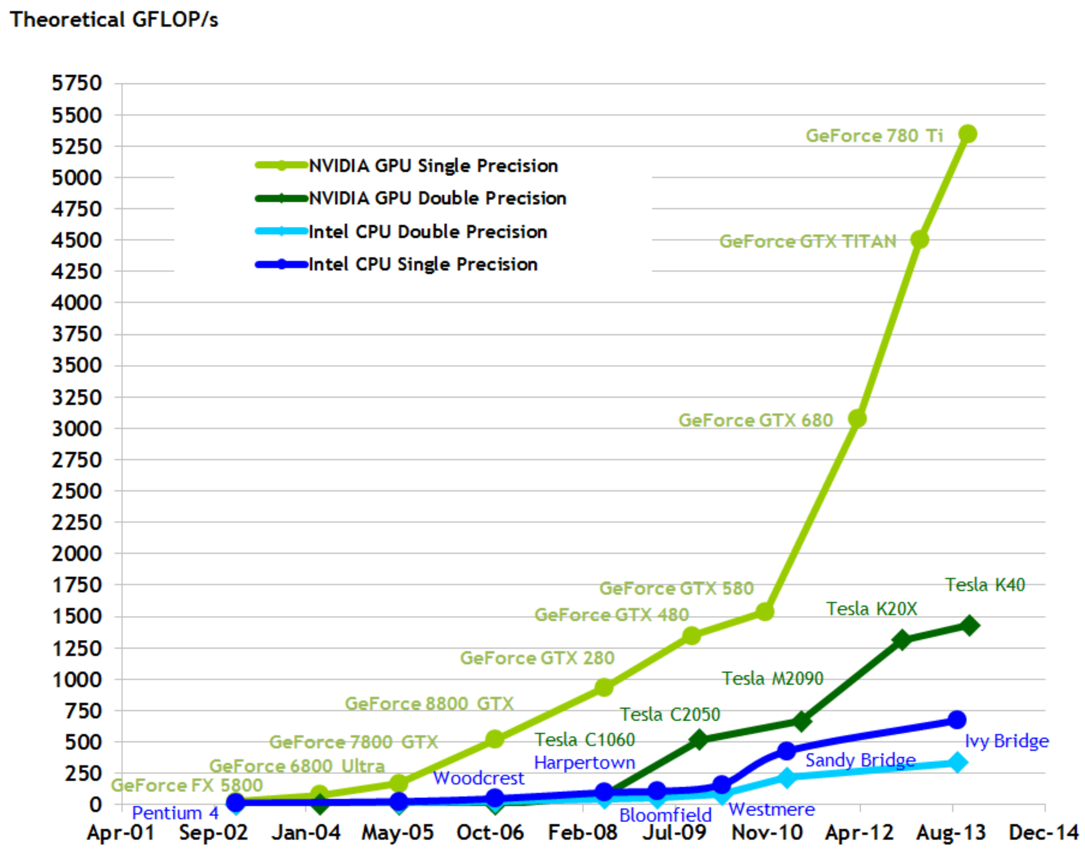

Figure 1.2: Comparison of CPU and GPU single precision floating point performance through the years. Plot taken from Nvidia's *CUDA C programming guide* [86].

Performance per Watt

In recent years, power consumption has become an important matter for sustainable technology. Today the notion of *performance per watt*[1] 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. Latest CPU architectures such as Intel's Haswell CPUs and Nvidia's Maxwell GPUs have opted at improving the performance per Watt.

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 bandwidth B_h of the hardware:

$$B_c = \frac{B_e}{B_h} \tag{1.16}$$

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 sometimes can be much harder than in CPU. The main reason is because memory performance is problem-dependent. Data structures have to be aligned in

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

simple patterns so that many chunks of data are read or written simultaneously for many threads. Irregular data accesses, non-compatible alignments and different data chunk sizes result in significant lower memory bandwidth. Latest GPU architectures such as *Fermi* and *Kepler* can mitigate this effect by using an L2 cache for global memory (see [85, 30] for more information on the GPU's L2 cache).

The performance measures presented in this section are all related in some way to 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, obtaining the analytical expression of $T(n, p)$ can be non-trivial because it will depend on *parallel computing model*.

Chapter 2: Parallel Computing Models

Computing models are abstract computing machines that allow theoretical analysis on the cost of algorithms. These models simplify the computational scenario to a reduced set of parameters that define how much time a memory access or an arithmetic 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, such as the *PRAM*, *PMH*, *Bulk parallel processing* and *LogP* models. The difference between one model and another basically resides on their definition of processor communication and memory access.

PRAM Model

The *parallel random access machine*, or PRAM, was proposed by Fortune and Wyllie in 1978 [40]. 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, although they are not pure PRAM machines.

In the PRAM model, there are p processors that operate synchronously over an unlimited memory completely visible for each processor (see Figure 2.1). The p parameter does not need to be a constant

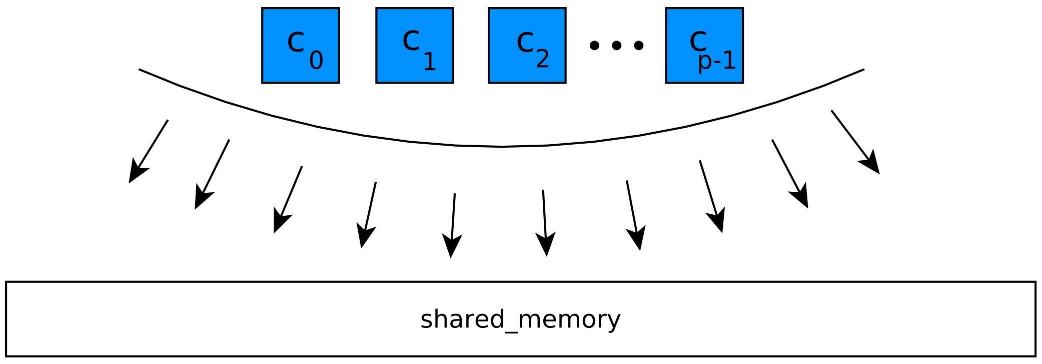

Figure 2.1: In the PRAM model each one of the cores has a complete view of the global memory.

number, it can also be defined as a function of the problem size n. Each r/w (read/write) operation costs $O(1)$. 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 situations one can model with the ERCW variation. 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 [73], 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 [8]. Subramonian [102] presented an $O(log\ n)$ 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 [109] as well as the notion of explicit Multi-threading for achieving efficient implementations of PRAM algorithms [110].

PMH Model

The *Parallel Memory Hierarchy* model, or PMH, was proposed in 1993 by Alpern *et al.* [4] and inspired by related works [3, 2] (HMM and UHM memory models). This model was proposed to deal with the inconsistency between theoretical and empirical performance of some PRAM algorithms, for assuming constant time memory accesses. 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 block-size), 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 Figure 2.2).

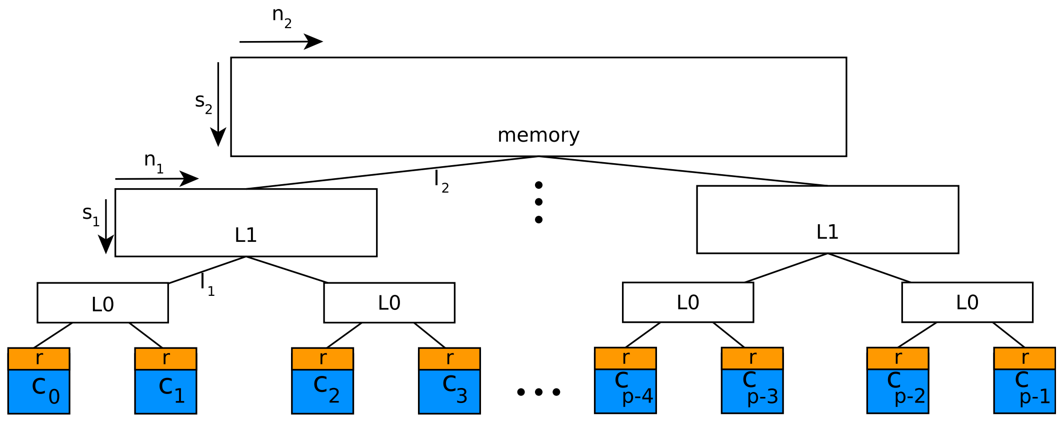

Figure 2.2: The uniform parallel memory hierarchy tree.

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 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 non-blocked matrix multiplication algorithm (*i.e.*, the basic matrix multiplication algorithm) can cost $\Omega(N^5/p)$ time instead of $O(N^3/p)$ [12] as in PRAM. In the same work, the authors prove that a parallel block-based matrix multiplication algorithm (see Figure 2.3) can indeed achieve the desired $O(N^3/p)$ upper bound by reusing the data from the fastest memory modules.

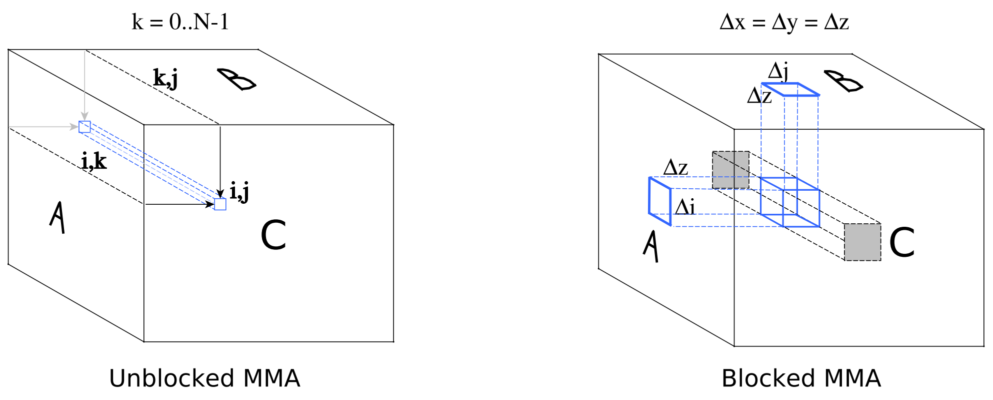

Figure 2.3: Classic algorithm processes sticks of computation as seen in the left side. The blocked version computes sub-cubes of the domain in parallel, taking advantage of locality.

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.

Bulk Synchronous Parallel (BSP)

The *Bulk synchronous parallel*, or BSP, is a parallel computing model focused on communication, published in 1990 by Leslie Valiant [108]. 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 Figure 2.4).

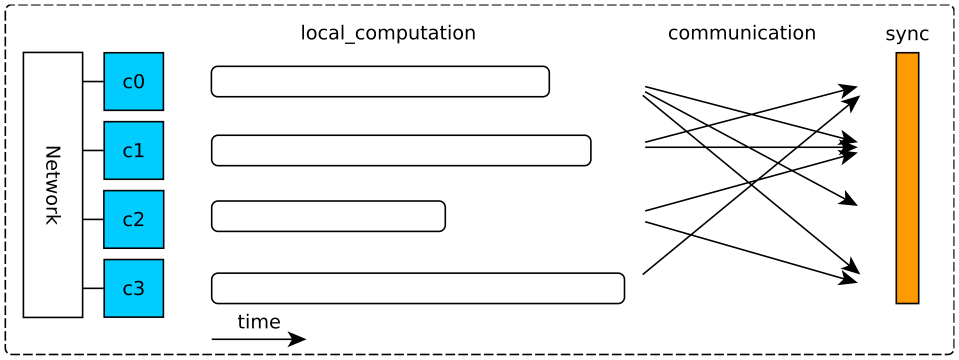

Figure 2.4: A representation of a super-step; processing, communication and a global synchronization barrier.

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:

$$c = max_{i=1}^{p}(w_i) + g\ max_{i=1}^{p}(h_i) + l \tag{2.1}$$

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:

$$C = \sum_{i=1}^{S} c_i \tag{2.2}$$

LogP

The LogP model was proposed in 1993 by Culler *et al.* [32]. 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. Figure 2.5 illustrates the model with an example of communication with one-word messages. The LogP model is similar to the

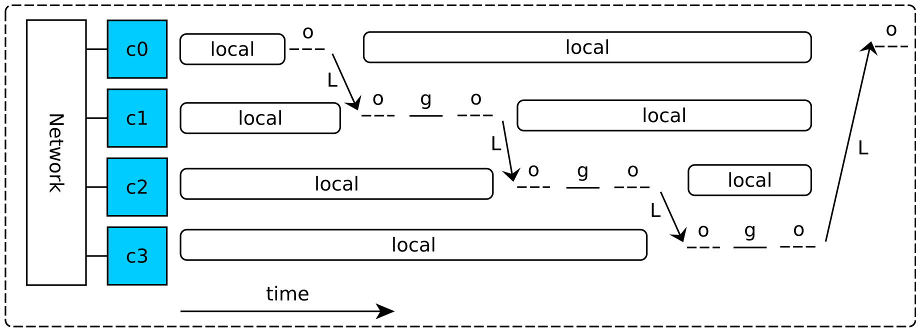

Figure 2.5: An example communication using the LogP model.

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.

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*.

Chapter 3: Parallel programming models

Parallel programming models focus on the expressiveness when programming parallel machines. 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*[1] is an abstraction of the programmable aspects of a computing model. While computing models from Chapter 2 are useful for algorithm design and analysis (*i.e.*, computing time complexity), *parallel programming models* are useful for expressing the parallelism of the algorithm. In this Chapter we cover four relevant parallel programming models.

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 [19] which is based on the Unix *pthreads* implementation [82]. In the case of GPUs, OpenCL [60] and CUDA [86] 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 [36], 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 [86].

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 [35]. 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 [43]. MPI is used for handling communication in CPU distributed applications and is also used to distribute the work when using multiple GPUs.

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.* [63] 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 [34] 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

[1]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 the Chapters (2) and (3).

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) [69] 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 [83]. Automatic parallelization is hard to achieve for algorithms not based on simple loops and has become a research topic in the last twenty years [97, 48, 67, 71].

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 [26]. 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 $f()_1 - > f()_2 - > ... - > f()_n$ are parts of a larger logic that works as a pipeline. Each stage of the pipeline can work in parallel;
- Parallel tasks: In this pattern, $f()_1, f()_2, ..., f()_n$ 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 $d : A \rightarrow \{a_1, a_2, ..., a_k\}$ and a combine function $c : \{a_1, a_2, ..., a_k\},\ f() \rightarrow f(\{a_1, a_2, ..., a_k\})$ 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.

$$r(A, d, c, f) = c(\{r(d(A)_1, d, c), r(d(A)_2, d, c), ..., r(d(A)_k, d, c)\}, f) \tag{3.1}$$
$$r(A', d, c, f) = c(d(A'), f) \tag{3.2}$$

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 these last two Chapters 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.

Chapter 4: 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[1] [96]. 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.

Flynn's taxonomy

Computer architectures can be classified by using Flynn's taxonomy [39]. 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.

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.

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

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 multi-core machines with two or more CPU sockets (*e.g.*, AMD Opteron and Intel Xeon). Figure 4.1 shows the concepts of UMA and NUMA.

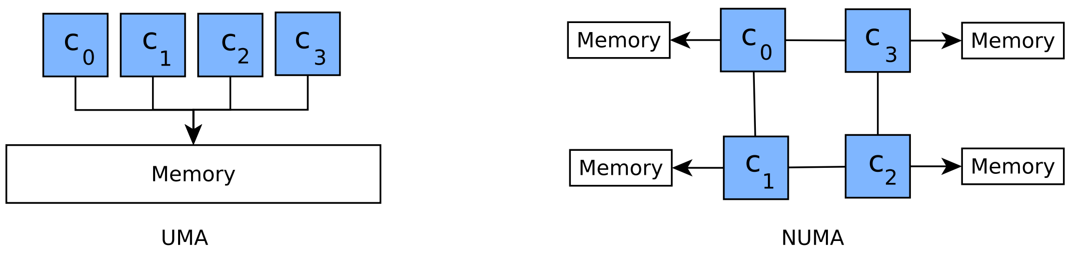

Figure 4.1: The UMA (*aka* SMP) and NUMA memory architectures.

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.

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 Figure 4.2). Each core has a local L1 and L2 cache of 64KB and 256KB, respectively, and in the center of the chip there is a larger L3 cache of size 20MB, shared by all cores. 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. Figure 4.3 shows the schematic of an Nvidia Tesla K20X GPU as well as its actual chip. The cores of the GPU are grouped into SMX units, or *next generation streaming multiprocessors*. The most important aspects that characterize a GPU are inside the SMX units (see Figure 4.4).

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

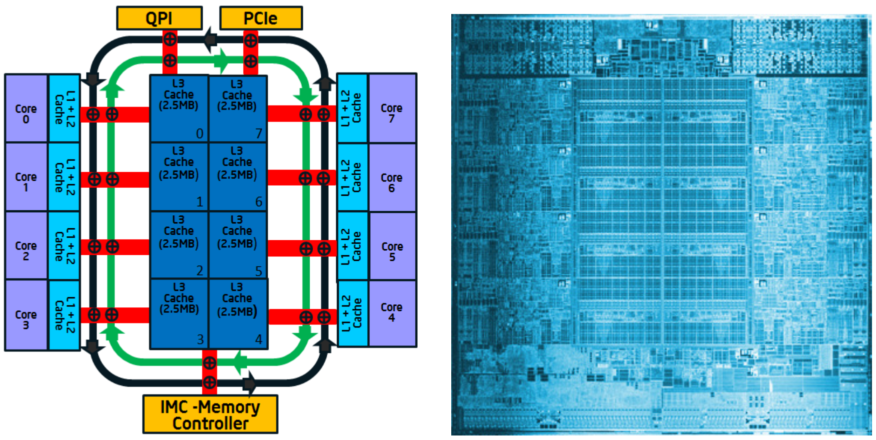

Figure 4.2: On the left, an Intel Xeon E5 2600 (2012) chip schematic. On the right, the actual chip.

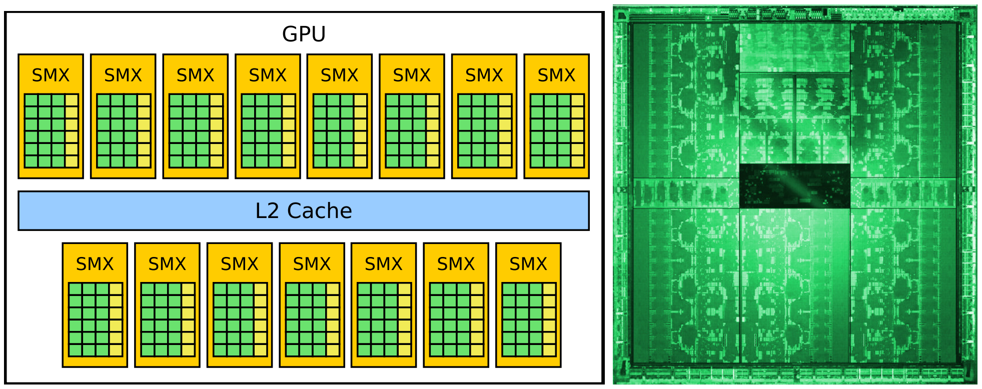

Figure 4.3: On the left, Nvidia's Tesla K20X GPU schematic. On the right, a picture of its chip.

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.

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*.

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

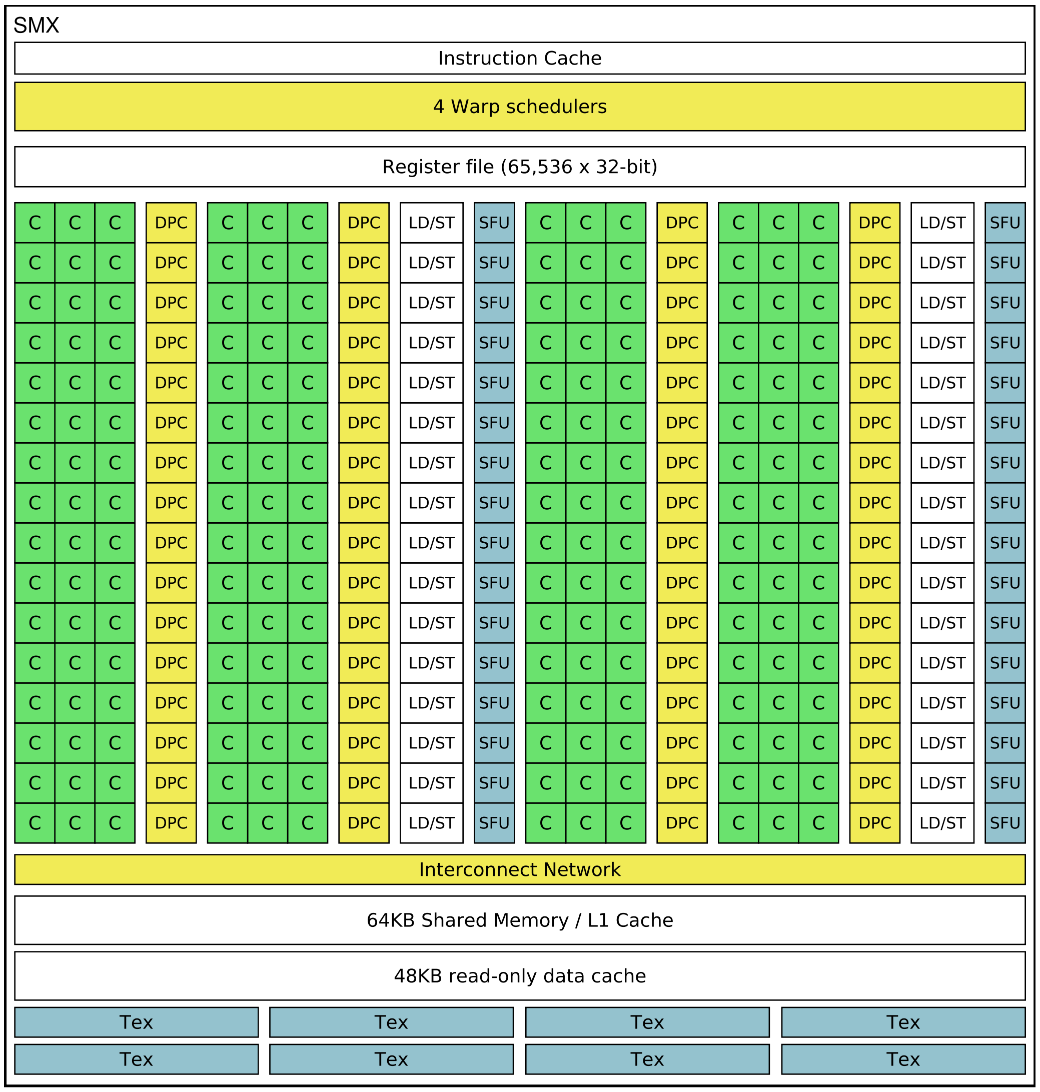
SMX
Instruction Cache
4 Warp schedulers
Register file (65,536 x 32-bit)
C
DPC
LD/ST
SFU
Interconnect Network
64KB Shared Memory / L1 Cache
48KB read-only data cache
Tex

Figure 4.4: A diagram of a streaming multiprocessor next-generation (SMX). Image inspired from Nvidia's CUDA C programming guide [86].

PRAM and UPMH models (sections 2 and 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 Figure 4.5). Efficient algorithms designed for GPUs have reported over $10\times$ speedup over CPU implementations [70, 27].

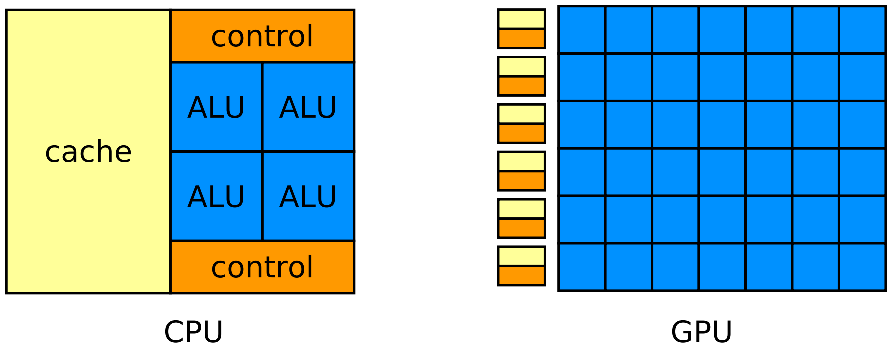

Figure 4.5: The GPU architecture differs from the one of the CPU because its layout is dedicated for placing many small cores, giving little space for control and cache units.

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.

Chapter 5: Strategy for designing a parallel algorithm

Designing a new algorithm is not a simple task. In fact, it is considered an art [64, 88] 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.

However, there are some formal strategies that are frequently used for creating efficient parallel algorithms. Leighton and Thomson [68] 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 [41] identified a four-step strategy that is present in many well designed parallel algorithms; *partitioning, communication, agglomeration and mapping* (see Figure 5.1).

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 possible partition; 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.

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 logical 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.

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 [20]. 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.

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 problems may require more elaborate mapping patterns in order to provide efficient performance.

Figure 5.1 illustrates all four steps using a data-partition based problem on a dual core architecture (c_0 and c_1).

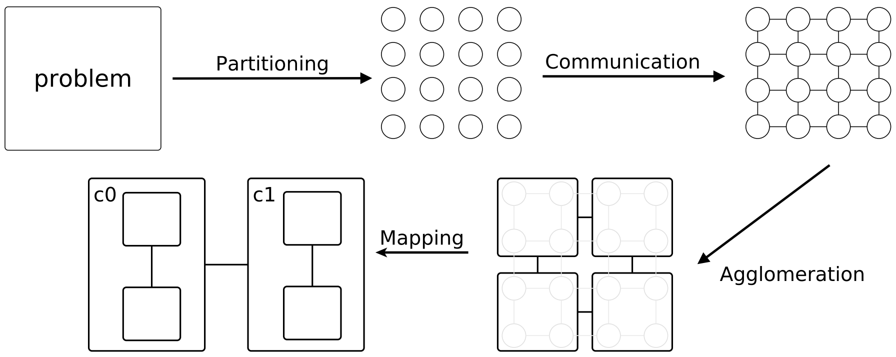

Figure 5.1: Foster's diagram of the design steps used in a parallelization process.

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.

Chapter 6: Parallel CPU Computing

Chapter 7: 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 the acronym GPGPU coined by researcher Mark Harris, which means *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 runs 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 processing technique [72]. In 2001, for the first time the graphics processing unit was built upon a programmable architecture, permitting programmable lighting [62, 59, 24], shadow [25] and geometry [98] 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) [75], HLSL (High-level Shading Language) [87] and CG (*C* for Graphics) [74]. 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* [77]. In 2004, Buck *et al.* proposed *Brook for GPUs*, also known as *Brook-GPU* [16]. 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) [86]. 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 [30], suitable for GPU computing. At the moment, only Nvidia GPUs can be programmed using 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 [60]. 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.

The massive parallelism programming model

The programming models explained in Chapter 3 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 Chapter 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. Figure 7.1 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*

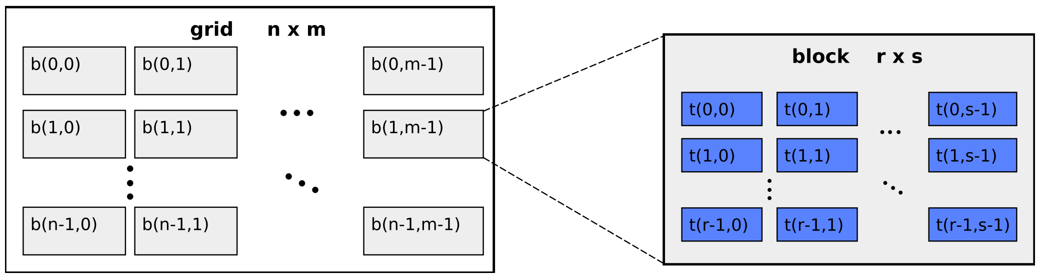

Figure 7.1: Massive parallelism programming model presented as a 2D model including grid, blocks and threads. Image inspired from Nvidia's CUDA C programming guide [86].

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.

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. Figure 7.2 summarizes the work-flow.

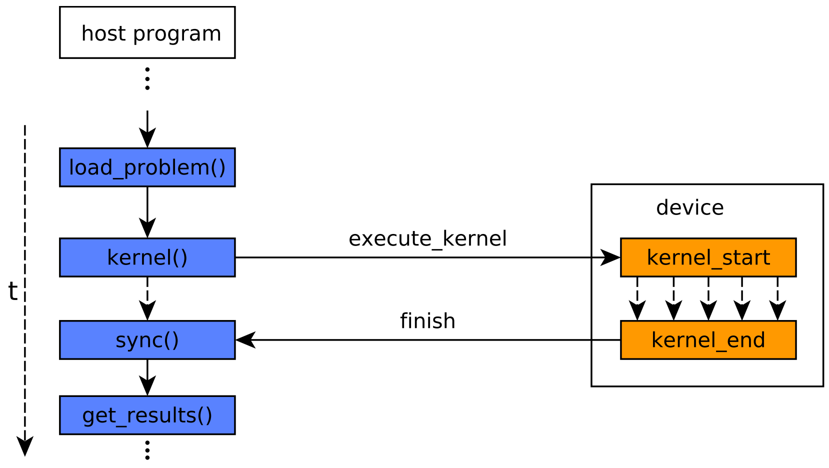

Figure 7.2: The GPU's main function, named *kernel*, is invoked from the CPU host code.

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.

Technical considerations for a GPU implementation

The GPU computing community frequently uses the terms *coalesced memory*, *thread coarsening*, *padding* and *branching*. These terms are 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, memory 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 [88].

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.

Latest advances and open problems in GPU computing

In the field of computational physics, $O(n)$ cost algorithms (the fast multi-pole expansion method [112, 113, 114]) have been implemented on GPU for n-body simulations. Single GPU implementations have been proposed for achieving high performance Potts model simulations [106, 107] even with biological applications [21]. Multi-GPU based implementations have also been proposed for the Potts model [65] and for n-body simulations [111]. 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 [38, 66, 44, 58]. 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 [57, 56, 115] as well as real-time methods for 3D reconstruction and level set segmentation [45, 95]. The field of programming languages have contributed to parallel computing with 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 [103, 18]). 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

[17] 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* [30]; 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 is the introduction 3D memories such as the High Bandwidth Memory (HBM) for GPUs and the Hybrid Memory Cube (HMC) for CPUs [104]. A three-dimensional memory architecture can provide up to $15\times$ 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 efficiently. Tree implementations have been implemented on the GPU, with an acceptable efficiency, 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 difficulty 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 an interesting case, since finding an efficient map for each block of the grid to the fractal problem domain is not trivial. One way of solving this problem would be to find an efficient mapping function from the space of computation to the problem domain, or modify the problem domain so that it becomes an Euclidean box, but for the last approach data organization would be an issue to consider. In this thesis we did a research on the problem of mapping threads onto 2D triangular domains, where we found an efficient map that runs up to 18% faster than the bounding box method [81]. The research can be found in Appendix **??**.

Chapter 8: Parallel Algorithms

Fast Fourier Transform

La transformada de Fourier (Fourier Transform o FT) es una transformación matemática que descompone una función, típicamente en el tiempo (como una señal), en las frecuencias que la constituyen. Existen diversos algoritmos para realizar esta transformación, y el *Fast Fourier Transform* es actualmente el más eficiente para procesar señales discretas, con un tiempo computacional de $\mathcal{O}(n \log n)$ con n la cantidad de muestras de la señal, o simplemente el largo de la señal. Desde que FFT se implementó en los computadores, la FT se ha vuelto una herramienta de primera importancia, siendo utilizada en diversas áreas de la ciencia y la ingeniería, tales como procesamiento de audio, astroinformática, ingenieria, física, geología, videojuegos, procesamiento de imágenes, compresión de información, entre muchas otras. En general, la FT puede ser de gran utilidad en cualquier ámbito donde exista una secuencia de datos en el tiempo.

Breve Historia

Debido al nombre de la transformada, *Fourier Transform*, se suele dar el crédito completo al matemático Joseph Fourier por la relación encontrada. Sin embargo, a lo largo de los años, se ha descubierto que Gauss realizó contribuciones importantes en la formulación de un algoritmo eficiente para calcular la FT. Es más, los primeros indicios de representar funciones con expresiones trigonométricas data de los años 1700s con la participación de matemáticos como Euler, Clairaut, Lagrange y Bernoulli.

Estudio de Fourier (1822)

Joseph Fourier, un matemático Francés (1768-1830), mostró en 1807 que algunas funciones podían ser representadas como una suma infinita de harmónicos. Su trabajo no fue publicado hasta 1822, ya que no fue bien recibido y rechazaron su publicación en la *Memoirs of the Academy*. Hay registros de que uno de sus primeros manuscritos en el tema data de los años 1804-1805 e incluye investigación que pudo comenzar en 1802 [53].

Contribución de Gauss (1805)

De forma paralela a Fourier, Carl Friedrich Gauss (1777-1855) trabajó alrededor del año 1805 en técnicas para lograr una interpolación trigonométrica de funciones periódicas. La expresión propuesta por Gauss es la primera evidencia de la idea de un algoritmo eficiente para calcular la *Discrete Fourier Transform* (DFT) [14].

FFT: Cooley and Tuckey (1965)

En 1965, Cooley y Tuckey publicaron una técnica eficiente [28], de costo $\mathcal{O}(n \log n)$, para calcular la DFT, que hoy en dia es conocida como la *Fast Fourier Transform* [84].

Otros Trabajos y Actualidad (1965-2020)

Un trabajo importante en la historia de FT y sus algoritmos es el de Danielson-Lanczos [33] en 1942, que establece las bases para el algoritmo de Cooley-Tuckey. El trabajo de Danielson-Lanczos no es el único caso, existen muchos otros que han contribuido de alguna u otra forma a FT y los algoritmos involucrados.

La DFT se puede extender a múltiples dimensiones sin mayor dificultad y en la actualidad se siguen realizando contribuciones a FFT. Es mas, existe una pregunta abierta en ciencias de la computación, y es si acaso existe o no una cota $\Omega(n \log n)$ para DFT. De existir, entonces el problema ya ha sido resuelto computacionalmente. De lo contrario, existiría la posibilidad de inventar un algoritmo mejor que el FFT. Por el momento no se ha podido encontrar respuesta a la pregunta.

Por un lado, FFT tiene versiones mejoradas para casos particulares, como lo es el Hexagonal FFT [78], el algoritmo de Rader [94], el algoritmo de Bruun [15]. En las últimas décadas, se ha propuesto formas eficientes para calcular la FFT en paralelo [47, 23] para aprovechar el rendimiento acelerado de las CPUs [42] modernas como también de GPUs modernas [80, 22]. Recientes investigaciones ya preparan las adaptaciones necesarias de FFT para lograr rendimiento en el orden *Exascale* [6].

A pesar de que los fundamentos de la FT, la DFT y los algoritmos fueron establecidos hace muchos años, aun existe interés científico en encontrar mejores algoritmos para el calculo de la DFT, que puedan reducir el numero de operaciones a un mínimo, o paralelizar las operaciones de forma eficiente en consideración de la jerarquía de memoria de las CPUs, GPUs y otros procesadores de la actualidad. Además, aún existe la pregunta abierta de si acaso la complejidad de la DFT necesita al menos $\Omega(n \log n)$ operaciones o no.

Fourier Transform

La transformada de Fourier (FT) es capaz de descomponer una onda, de carácter compleja, en una secuencia de ondas mas simples y elementales. La idea es análoga a como un acorde de musica se descompone en las distintas notas elementales.

Definición de la FT

Sea $f : \mathbb{R} \mapsto \mathbb{C}$ una función integrable, entonces la transformada de Fourier $\hat{f}$ se define como:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \xi x} \, dx \tag{8.1}$$

Típicamente $f(x)$ representa una cantidad en el dominio del tiempo x (segundos), mientras que $f(\xi)$ se convierte en una cantidad para el dominio de las frecuencias ξ (hertz). Si consideramos la formula de Euler:

$$e^{\varphi i} = \cos(\varphi) + i \sin(\varphi) \tag{8.2}$$

Entonces podemos escribir $\hat{f}(\xi)$ como:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)[\cos(-2\pi\xi x) + i \sin(-2\pi\xi x)] \, dx \tag{8.3}$$

Si las condiciones analíticas lo permiten, es posible recuperar $f(x)$ de $\hat{f}(\xi)$ mediante la transformada inversa:

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi) \, e^{2\pi i \xi x} \, d\xi \tag{8.4}$$

Hasta ahora las ecuaciones introducidas han asumido un espacio continuo en el dominio y recorrido de las funciones. En la práctica es bastante común trabajar en un espacio discreto ya sea por la naturaleza del dominio (ejemplo, un dominio virtual puede preferir enteros) o bien porque en la práctica muestreamos datos del entorno para ingresarlos al computador como una serie discreta en el tiempo. Para estos casos podemos usar *Discrete Fourier Transform* (DFT).

Discrete Fourier Transform

Cuando una señal es de naturaleza discreta, o proviene de un muestreo finito de una fuente continua, se usa la transformadad de fourier discreta o *discrete Fourier transform* (DFT) en Inglés. Para el ámbito computacional y experimental, la DFT es una herramienta de mucha utilidad pues permite la realizacion de algoritmos que dependen de utilizar el espectro de frecuencias de la señal.

Definición de la DFT

Discrete Fourier Transform (DFT) transforma una secuencia de n números complejos $x_0, x_1, ..., x_{n-1}$ que representa cantidades en el tiempo, en otra secuencia de números complejos $X_0, X_1, \ldots, X_{n-1}$ que representa cantidades en ciertas frecuencias. La transformación, $X_k = \mathcal{F}_k$, se define como

$$X_k = \mathcal{F}_k = \sum_{r=0}^{n-1} x_r \, e^{-\frac{i2\pi}{n} kr} \tag{8.5}$$

Utilizando la formula de Euler, tenemos que

$$X_k = \sum_{r=0}^{n-1} x_r \left[\cos\left(\frac{2\pi}{n} kr\right) - i \sin\left(\frac{2\pi}{n} kr\right) \right] \tag{8.6}$$

El valor de k funciona en el rango $[0..n-1]$ y estos valores no son directamente valores de frecuencia, sino que indices a los eventuales valores de frecuencia. Se asume que los valores $x_0, x_1, \cdot, x_{n-1}$ vienen uniformemente espaciados en intervalos de tiempo Δt, o equivalentemente con una frecuencia de muestreo de $f_s = 1/\Delta_t$. Entonces, dado un X_k, su frecuencia asociada f_k viene dada por

$$f_k = \frac{k}{n} f_s \tag{8.7}$$

Los valores f_k son importantes para graficar la intensidad de la señal en relación al espectro de frecuencias en el cual se desenvuelve.

Algoritmo Básico DFT

Dada una señal $x_0, x_1, \cdots, x_{n-1}$, un algoritmo básico de DFT puede calcular la expresión de la Eq. 8.5 o 8.6 para cada $X_0, X_1, \cdots, X_{n-1}$ tal como se muestra en el Algoritmo 1. Si bien el algoritmo básico es

Algorithm 1: Algoritmo básico DFT

```
Input: $x[n] : \{x_0, x_1, \cdots, x_{n-1}\}$
Result: $X[n] : \{X_0, X_1, \cdots, X_{n-1}\}$
$X[] \leftarrow$ new complex$[n]$;
for $k \in 0..n-1$ do
    s $\leftarrow \{0, 0\}$;
    c $\leftarrow 2\pi k/n$;
    for $r \in 0..n-1$ do
        $s.real \leftarrow s.real + x[r] \cdot \cos(c \cdot r)$;
        $s.imag \leftarrow s.imag - x[r] \cdot \sin(c \cdot r)$;
    end
    $X[k] \leftarrow$ s;
end
return X;
```

sencillo y puede calcular correctamente la DFT para cualquier tamaño n de señal, este tiene un costo computacional de $\Theta(n^2)$. Para valores pequeños de n el costo computacional es abordable, sin embargo a medida que crece n, el comportamiento cuadrático se hace notar rápidamente.

Si se analiza mejor la Eq. 8.5, es posible ver que parte del trabajo realizado para calcular un termino X_k de la DFT puede servir para el cálculo de otro X_j. Al usar esta simetría en su completitud, se puede llegar a la transformada rápida de Fourier, o más conocida como FFT por *Fast Fourier Transform*.

Algoritmo FFT

Fast Fourier Transform, o FFT, se refiere a una estrategia algorítmica que logra calcular la DFT en tiempo $\mathcal{O}(n \log n)$. En esta sección, revisaremos el algoritmo de Cooley-Tuckey, propuesto en 1965. En particular, nos referiremos a la versión *Radix-2*, un caso particular de Cooley-Tuckey que particiona recursivamente el problema en dos partes iguales.

Cooley-Tuckey: versión Radix-2

La principal idea de la variante Radix-2 es dividir el cómputo de la DFT de las muestras pares $\{x_0, x_2, \cdots, x_{n-2}\}$ con la DFT de las muestras impares $\{x_1, x_3, \cdots, x_{n-1}\}$. La ecuación Eq. 8.8 muestra la idea, donde el índice m seguirá una sucesión par o impar según corresponda.

$$X_k = \sum_{m=0}^{n/2-1} x_{2m} e^{-\frac{2\pi i}{n}(2m)k} + \sum_{m=0}^{n/2-1} x_{2m+1} e^{-\frac{2\pi i}{n}(2m+1)k} \tag{8.8}$$

Es posible factorizar el término $e^{-\frac{2\pi i}{n}k}$ de la segunda sumatoria, debido al $+1$ del exponente. Si además ubicamos el factor 2 que acompaña a m en cada sumatoria como divisor de n, quedan dos DFTs de tamaño $n/2$, una para pares E_k y otra para impares O_k,

$$X_k = \sum_{m=0}^{n/2-1} x_{2m} e^{-\frac{2\pi i}{n/2}mk} + e^{-\frac{2\pi i}{n}k} \sum_{m=0}^{n/2-1} x_{2m+1} e^{-\frac{2\pi i}{n/2}mk} \tag{8.9}$$

$$X_k = E_k + e^{-\frac{2\pi i}{n}k} O_k \tag{8.10}$$

Dado que el termino $e^{-\frac{2\pi i}{n}}$ aparece con frecuencia, usaremos la notación

$$W_n = e^{-\frac{2\pi i}{n}} \tag{8.11}$$

De esta forma podemos expresar X_k como

$$X_k = E_k + W_n^k O_k \tag{8.12}$$

con $W_n^k = (e^{-\frac{2\pi i}{n}})^k = e^{-\frac{2\pi i}{n}k}$. Ahora, tanto E_k como O_k pueden volver a ser tratados como problemas separables, esta vez en sub-problemas de $n/4$ cada uno. Siguiendo esta lógica, se puede construir un algoritmo recursivo que subdivide en mitades (par e impar) el problema. Si bien la subdivisión recursiva es una componente clave de la FFT, por si sola no es suficiente para generar un algoritmo de costo $\mathcal{O}(n \log n)$, ya que habria que hacerle el mismo tratamiento a todos los X_k por igual. Sin embargo, al combinar la recursión con las propiedades de periodicidad y simetría, se abre la posibilidad de encontrar el algoritmo eficiente FFT Radix-2.

Entendiendo Periodicidad y Simetría

Existen dos propiedades criticas que deben aprovecharse para generar un algoritmo FFT de costo $\mathcal{O}(n \log n)$, i) periodicidad y ii) simetría.
i) Periodicidad:. Para un input de tamaño $n/2$, se tiene que $X_{k+\frac{n}{2}}$ se puede calcular con los mismos términos E_k e O_k usados para calcular X_k. Para el caso par, tenemos

$$E_{k+\frac{n}{2}} = \sum_{m=0}^{n/2-1} x_{2m} W_{n/2}^{(k+n/2)m} \tag{8.13}$$
$$= \sum_{m=0}^{n/2-1} x_{2m} W_{n/2}^{km} W_{n/2}^{(n/2)m} \tag{8.14}$$

en el cual $W_{n/2}^{(n/2)m} = e^{-2\pi i m}$ y por formula de Euler $W_{n/2}^{(n/2)m} = e^{-2\pi i m} = \cos(-2\pi m) + i\sin(-2\pi m) = 1$. Así, emerge la primera propiedad, que establece

$$E_{k+\frac{n}{2}} = E_k \tag{8.15}$$

El proceso es análogo para los impares, *i.e.*, $O_{k+\frac{n}{2}} = O_k$.
ii) Simetría de media distancia:. Para la expresión completa, tenemos entonces que

$$X_{k+\frac{n}{2}} = E_{k+\frac{n}{2}} + W_n^{k+\frac{n}{2}} O_{k+\frac{n}{2}} \tag{8.16}$$
$$= E_k + W_n^k e^{-\pi i} O_k \tag{8.17}$$
$$= E_k - W_n^k O_k \tag{8.18}$$

donde por formula de Euler, $e^{-\pi i} = \cos(-\pi) + i\sin(-\pi) = -1$. De la expresión, emerge la segunda propiedad, la cual establece que

$$W_n^{k+n/2} = -W_n^k. \tag{8.19}$$

Finalmente, como resumen de las simetrías, se tiene que

$$X_k = E_k + W_{n/2}^k O_k \tag{8.20}$$
$$X_{k+\frac{n}{2}} = E_k - W_{n/2}^k O_k \tag{8.21}$$

para $k \in [0..(n-1)]$.

Combinando las dos propiedades junto con la estructura recursiva, es posible formular un algoritmo FFT eficiente de costo $\mathcal{O}(n \log n)$. Existen dos formas de implementar la idea, una es usando un algoritmo recursivo mientras que la segunda idea es utilizar un algoritmo iterativo. La diferencia entre cada forma es que la primera (recursivo) aplica un esquema *depth-first-search* (DFS) mientras que la segunda *breadth-first-search* (BFS) en el arbol de recursion.

FFT: Algoritmo Radix-2 Recursivo

La idea principal del Radix-2 recursivo es ilustrada en la Figura 8.1.

```
X_{0,...,N−1} ← ditfft2(x, N, s):             DFT of (x_0, x_s, x_{2s}, ..., x_{(N-1)s}):
    if N = 1 then
        X_0 ← x_0                                        trivial size-1 DFT base case
    else
        X_{0,...,N/2−1} ← ditfft2(x, N/2, 2s)            DFT of (x_0, x_{2s}, x_{4s}, ...)
        X_{N/2,...,N−1} ← ditfft2(x+s, N/2, 2s)          DFT of (x_s, x_{s+2s}, x_{s+4s}, ...)
        for k = 0 to N/2−1 do                            combine DFTs of two halves into full DFT:
            t ← X_k
            X_k ← t + exp(−2πi k/N) X_{k+N/2}
            X_{k+N/2} ← t − exp(−2πi k/N) X_{k+N/2}
        end for
    end if
```

Figure 8.1: Algoritmo recursivo para el Radix-2 FFT.

FFT: Algoritmo Radix-2 Iterativo

La idea principal del Radix-2 iterativo es ilustrada en la Figura 8.1.

```
algorithm iterative-fft is
    input: Array a of n complex values where n is a power of 2.
    output: Array A the DFT of a.

    bit-reverse-copy(a, A)
    n ← a.length
    for s = 1 to log(n) do
        m ← 2^s
        ω_m ← exp(−2πi/m)
        for k = 0 to n-1 by m do
            ω ← 1
            for j = 0 to m/2 – 1 do
                t ← ω A[k + j + m/2]
                u ← A[k + j]
                A[k + j] ← u + t
                A[k + j + m/2] ← u – t
                ω ← ω ω_m

    return A
```

Figure 8.2: Algoritmo iterativo para el Radix-2 FFT.

Notar que esta versión utiliza un proceso de bit-reverse para ubicar de forma correcta los resultados.

Chapter 9: Colors

This package provides several global color variables to style `DndComment`, `DndReadAloud`, `DndSidebar`, and `DndTable` environments.

Box Colors

Color	Description
`commentcolor`	`DndComment` background
`readaloudcolor`	`DndReadAloud` background
`sidebarcolor`	`DndSidebar` background
`tablecolor`	background of even `DndTable` rows

They also accept an optional color argument to set the color for a single instance. See Table 9.1 for a list of core book accent colors.

```
\begin{DndTable}[color=PhbLightCyan]{cX}
  \textbf{d8} & \textbf{Item} \\
  1 & Small wooden button \\
  2 & Red feather \\
  3 & Human tooth \\
  4 & Vial of green liquid \\
  6 & Tasty biscuit \\
  7 & Broken axe handle \\
  8 & Tarnished silver locket \\
\end{DndTable}
```

d8	Item
1	Small wooden button
2	Red feather
3	Human tooth
4	Vial of green liquid
6	Tasty biscuit
7	Broken axe handle
8	Tarnished silver locket

Themed Colors

Use `\DndSetThemeColor[<color>]` to set `commentcolor`, `readaloudcolor`, `sidebarcolor`, and `tablecolor` to a specific color. Calling `\DndSetThemeColor` without an argument sets those colors to the current `themecolor`. In the following example the group limits the change to just a few boxes; after the group finishes, the colors are reverted to what they were before the group started.

Colors Supported by This Package

Color	Description
`PhbLightGreen`	Light green used in PHB Part 1 (Default)
`PhbLightCyan`	Light cyan used in PHB Part 2
`PhbMauve`	Pale purple used in PHB Part 3
`PhbTan`	Light brown used in PHB appendix
`DmgLavender`	Pale purple used in DMG Part 1
`DmgCoral`	Orange-pink used in DMG Part 2
`DmgSlateGray` (`DmgSlateGrey`)	Blue-gray used in PHB Part 3
`DmgLilac`	Purple-gray used in DMG appendix

```
\begingroup
\DndSetThemeColor[PhbMauve]

\begin{DndComment}{This Comment Is in Mauve}
  This comment is in the the new color.
\end{DndComment}

\begin{DndSidebar}{This Sidebar Is Also Mauve}
  The sidebar is also using the new theme color.
\end{DndSidebar}
\endgroup
```

This Comment Is in Mauve

This comment is in the the new color.

This Sidebar Is Also Mauve

The sidebar is also using the new theme color.

Bibliography

[1] Sarita V. Adve and Kourosh 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] Gene 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] Alan Ayala, Stanimire Tomov, Azzam Haidar, and Jack Dongarra. heffte: Highly efficient fft for exascale. In *International Conference on Computational Science*, pages 262–275. Springer, 2020.

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

[8] Paul Beame and Johan Hastad. Optimal bounds for decision problems on the crcw pram. In *In Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 25–27. ACM, 1987.

[9] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational n-body code that runs entirely on the GPU processor. *J. Comput. Phys.*, 231(7):2825–2839, April 2012.

[10] Adrien Bernhardt, Andre Maximo, Luiz Velho, Houssam Hnaidi, and Marie-Paule 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.

[11] Z. Bittnar, J. Kruis, J. Němeček, B. Patzák, and D. Rypl. Parallel and distributed computations for structural mechanics: a review. *Civil and structural engineering computing: 2001*, pages 211–233, 2001.

[12] Larry Carter Bowen Alpern. The ram model considered harmful towards a science of performance programming. Technical report, IBM Watson research center, 1994.

[13] Clay P. Breshears. *The Art of Concurrency - A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly, 2009.

[14] E Oran Brigham. 6 the discrete fourier transform. *The fast Fourier transform and its applications*, 1988.

[15] Georg Bruun. z-transform dft filters and fft's. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(1):56–63, 1978.

[16] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[17] Gabriele Capannini, Fabrizio Silvestri, and Ranieri 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.

[18] Bradford L. Chamberlain. Chapel (cray inc. hpcs language). In *Encyclopedia of Parallel Computing*, pages 249–256. Springer, 2011.

[19] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[20] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The impact of synchronization and granularity on parallel systems. *SIGARCH Comput. Archit. News*, 18(3a):239–248, May 1990.

[21] Nan Chen, James A. Glazier, Jesús A. Izaguirre, and Mark 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.

[22] Yifeng Chen, Xiang Cui, and Hong Mei. Large-scale fft on gpu clusters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 315–324, 2010.

[23] Eleanor Chu and Alan George. *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press, 1999.

[24] Florent Cohen, Philippe Decaudin, and Fabrice Neyret. GPU-based lighting and shadowing of complex natural scenes. In *Siggraph'04 Conf. DVD-ROM (Poster)*, August 2004. Los Angeles, USA.

[25] Mark Colbert and Jaroslav Křivánek. Real-time dynamic shadows for image-based lighting. In *ShaderX 7 - Advanced Rendering Technicques*. Charles River Media, 2009.

[26] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[27] Aleksandar Colic, Hari Kalva, and Borko 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.

[28] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[29] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[30] Nvidia Corporation. *Kepler Whitepaper for the GK110 architecture*, 2012.

[31] Eliana Scheihing Cristobal A. Navarro, Nancy 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.

[32] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[33] Gordon Charles Danielson and Cornelius Lanczos. Some improvements in practical fourier analysis and their application to x-ray scattering from liquids. *Journal of the Franklin Institute*, 233(5):435–452, 1942.

[34] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[35] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[36] Neil Dunstan. Semaphores for fair scheduling monitor conditions. *SIGOPS Oper. Syst. Rev.*, 25(3):27–31, May 1991.

[37] 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.

[38] Néstor Ferrando, M. A. Gosálvez, Joaquín Cerdá, Rafael Gadea Gironés, 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.

[39] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[40] Steven Fortune and James 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.

[41] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[42] Franz Franchetti, Markus Puschel, Yevgen Voronenko, Srinivas Chellappa, and José MF Moura. Discrete fourier transform on multicore. *IEEE Signal Processing Magazine*, 26(6):90–102, 2009.

[43] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy 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.

[44] Stéphane Gobron, Arzu Çöltekin, Hervé Bonafos, and Daniel Thalmann. GPGPU computation and visualization of three-dimensional cellular automata. *The Visual Computer*, 27(1):67–81, 2011.

[45] Stéphane Gobron, Clément Marx, Junghyun Ahn, and Daniel Thalmann. Real-time textured volume reconstruction using virtual and real video cameras. In *proceedings of the Computer Graphics International 2010 conference*, 2010.

[46] Raymond Greenlaw, James H. Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, USA, April 1995.

[47] Anshul Gupta and Vipin Kumar. The scalability of fft on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, 1993.

[48] Manish Gupta, Sayak Mukhopadhyay, and Navin Sinha. Automatic parallelization of recursive procedures. *Int. J. Parallel Program.*, 28(6):537–562, December 2000.

[49] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[50] John L. Gustafson. Fixed time, tiered memory, and superlinear speedup. In *In Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)*, 1990.

[51] John 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] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 42 tflops hierarchical n-body simulations on GPUs with applications in both astrophysics and turbulence. In *SC*, 2009.

[53] Michael Heideman, Don Johnson, and Charles Burrus. Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine*, 1(4):14–21, 1984.

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

[55] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo 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] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat 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] Sriram Kashyap, Rhushabh Goradia, Parag Chaudhuri, and Sharat 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.

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

[59] Jan Kautz, Wolfgang Heidrich, and Hans-Peter 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.

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

[61] David B. Kidner, Philip J. Rallings, and J. Andrew Ware. Parallel processing for terrain analysis in GIS: Visibility as a case study. *Geoinformatica*, 1(2):183–207, August 1997.

[62] M.J. Kilgard. A practical and robust bump-mapping technique for today's GPUs. Technical report, Nvidia, 2000.

[63] Seon Wook Kim and Rudolf 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.

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

[65] Yukihiro Komura and Yutaka Okabe. Multi-GPU-based swendsen–wang multi-cluster algorithm for the simulation of two-dimensional -state potts model. *Computer Physics Communications*, 184(1):40 – 44, 2013.

[66] Pavol Korček, Lukáš Sekanina, and Otto Fučík. 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.

[67] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN Not.*, 42(6):235–244, June 2007.

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

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

[70] Peter Lu, Hidekazu Oki, Catherine Frey, Gregory Chamitoff, Leroy Chiao, Edward Fincke, C. Foale, Sandra Magnus, William McArthur, Daniel Tani, Peggy Whitson, Jeffrey Williams, William Meyer, Ronald Sicker, Brion Au, Mark Christiansen, Andrew Schofield, and David 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.

[71] Xiaosong Ma, Jiangtian 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.

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

[73] Philip D. Mackenzie and Vijaya Ramachandran. ERCW PRAMs and optical communication. In *in Proceedings of the European Conference on Parallel Processing, EUROPAR '96*, pages 293–302, 1996.

[74] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, July 2003.

[75] Ricardo Marroquim and André 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.

[76] Yossi Matias and Uzi 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.

[77] Michael D. McCool, Zheng Qin, and Tiberiu 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.

[78] Russell M Mersereau. The processing of hexagonally sampled two-dimensional signals. *Proceedings of the IEEE*, 67(6):930–949, 1979.

[79] A.S. Mikhayhu. *Embarrassingly Parallel*. Tempor, 2012.

[80] Kenneth Moreland and Edward Angel. The fft on a gpu. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, 2003.

[81] Cristobal A. Navarro and Nancy Hitschfeld. GPU maps for the space of computation in triangular domain problems. In *2014 IEEE International Conference on High Performance Computing and Communications, 6th IEEE International Symposium on Cyberspace Safety and Security, 11th IEEE International Conference on Embedded Software and Systems, HPCC/CSS/ICESS 2014, Paris, France, August 20-22, 2014*, pages 375–382, 2014.

[82] Bradford Nichols, Dick Buttlar, and Jacqueline P. Farrell. *Pthreads Programming*. O'Reilly, 101 Morris Street, Sebastopol, CA 95472, 1998.

[83] Rishiyur Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann, May 2001.

[84] Henri J Nussbaumer. The fast fourier transform. In *Fast Fourier Transform and Convolution Algorithms*, pages 80–111. Springer, 1981.

[85] Nvidia. *Fermi Compute Architecture Whitepaper*.

[86] Nvidia-Corporation. *Nvidia CUDA C Programming Guide*, 2012.

[87] Michael Oneppo. Hlsl shader model 4.0. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 112–152, New York, NY, USA, 2007. ACM.

[88] Stan Openshaw and Ian Turton. *High Performance Computing and the Art of Parallel Programming: An Introduction for Geographers, Social Scientists, and Engineers*. Routledge, New York, NY, 10001, 1999.

[89] David A. Padua, editor. *Encyclopedia of Parallel Computing*, volume 4. Springer, 2011.

[90] Michele Pagani and Paolo Tranquilli. Parallel reduction in resource lambda-calculus. In *APLAS*, pages 226–242, 2009.

[91] D. Parkinson. Parallel efficiency can be greater than unity. *Parallel Computing*, 3(3):261 – 262, 1986.

[92] Howard A. Peelle. To teach Newton's square root algorithm. *SIGAPL APL Quote Quad*, 5(4):48–50, December 1974.

[93] 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.

[94] Charles M Rader. Discrete fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, 1968.

[95] Mike Roberts, Jeff Packer, Mario Costa Sousa, and Joseph Ross 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.

[96] P. E. Ross. Why cpu frequency stalled. *IEEE Spectr.*, 45(4):72–72, April 2008.

[97] Radu Rugina and Martin 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.

[98] Pedro V. Sander and Jason 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.

[99] Angela Di Serio and María Blanca Ibáñez. Evaluation of a nearest-neighbor load balancing strategy for parallel molecular simulations in mpi environment. In *PVM/MPI*, pages 226–233, 2002.

[100] Yossi Shiloach and Uzi Vishkin. An o(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[101] Justin R. Smith. *The design and analysis of parallel algorithms*. Oxford University Press, Inc., New York, NY, USA, 1993.

[102] Ramesh Subramonian. An o(log n) time common CRCW PRAM algorithm for minimum spanning tree. Technical Report UCB/CSD-92-673, EECS Department, University of California, Berkeley, Mar 1992.

[103] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. Gramps: A programming model for graphics pipelines. *ACM Trans. Graph.*, 28(1):4:1–4:11, February 2009.

[104] Noboru Tanabe, Nobuhiro Hori, Boonyasitpichai Nuttapon, and Hironori Nakajo. Preliminary evaluations for hybrid memory cube with gather functions using FPGA. *IPSJ SIG Notes*, 2012(6):1–10, 2012-03-19.

[105] David Taniar, Clement H. C. Leung, Wenny Rahayu, and Sushant Goel. *High-Performance Parallel Database Processing and Grid Databases*. Wiley Series on Parallel and Distributed Computing, 2008.

[106] Jose Juan Tapia and Roshan DSouza. Data-parallel algorithms for large-scale real-time simulation of the cellular Potts model on graphics processing units. In *2009 IEEE International Conference on Systems Man and Cybernetics*, pages 1411–1418. IEEE, 2009.

[107] José Juan Tapia and Roshan D'Souza. Parallelizing the cellular potts model on graphics processing units. *Computer Physics Communications*, 182(4):857–865, 2011.

[108] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[109] Uzi Vishkin. A pram-on-chip vision (invited abstract). In *SPIRE*, page 260, 2000.

[110] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. Explicit multi-threading (xmt) bridging models for instruction parallelism (extended abstract). In *SPAA*, pages 140–151, 1998.

[111] Rio Yokota, Lorena Barba, Tetsu Narumi, and Kenji 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.

[112] Rio Yokota and Lorena A. Barba. Fast n-body simulations on GPUs. *CoRR*, abs/1108.5815, 2011.

[113] Rio Yokota and Lorena A. Barba. A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems. *CoRR*, abs/1106.2176, 2011.

[114] Rio Yokota and Lorena A. Barba. Hierarchical n-body simulations with autotuning for heterogeneous systems. *Computing in Science and Engineering*, 14(3):30–39, 2012.

[115] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):126:1–126:11, December 2008.