Algorithmic contributions to scientific computing on high performance architectures

Pierre Fortin

Sorbonne Université - Faculté des Sciences et Ingénierie
(Université Pierre et Marie Curie - Paris 6)
Laboratoire d'informatique de Paris 6

HABILITATION À DIRIGER DES RECHERCHES

Spécialité : informatique

Présentée et soutenue publiquement par

Pierre FORTIN

le 5 juillet 2018

Contributions algorithmiques au calcul scientifique sur architectures haute performance

(Algorithmic contributions to scientific computing on high performance architectures)

---

après avis des **rapporteurs**

François	BODIN	Université de Rennes 1
Dimitrios S.	NIKOLOPOULOS	Queen's University of Belfast
Richard	VUDUC	Georgia Institute of Technology

devant le **jury** composé de

François	BODIN	Université de Rennes 1
Pierre	BOULET	Université de Lille
Christophe	CALVIN	CEA
Stef	GRAILLAT	Sorbonne Université
Laura	GRIGORI	Inria Paris – présidente
Stéphane	VIALLE	CentraleSupélec
Richard	VUDUC	Georgia Institute of Technology

– 2018 –

Contents

Introduction

Computer hardware evolves at a much faster pace than algorithms and their implementations. Since mid-2000s, we have thus seen the advent of multi-core CPUs, with an ever increasing number of cores and with ever larger SIMD (Single Instruction, Multiple Data) units. Graphics Processing Units (GPUs) and other many-core architectures (like the Intel Xeon Phi processors, the PEZY-SC2 chip, the Matrix-2000 accelerator or the Sunway SW26010 processor) have also been increasingly used in HPC (High Performance Computing): almost all most powerful[1] and most energy-efficient[2] supercomputers now rely on such many-core architectures. These hardware changes are all meant to deliver ever greater compute powers, as well as ever greater energy efficiencies. New hardware changes are still required to reach exascale computing (10^{18} floating-point operations per second) in the forthcoming years, and could also imply technological breakthroughs such as Memristor-based memories, photonics for interconnection networks ...

However, when such new hardware changes are introduced, many numerical applications requiring HPC, including those in scientific computing, do not fully exploit these hardware improvements. This is partly due to the difficulty and development time required to adapt at best these numerical applications to a new architecture or to new hardware features. This makes the current state of many application codes being designed to a previous hardware generation. For example, some applications do not exploit many-core architectures, or CPU SIMD units, or even multi-core CPUs. Targeting multiple architectures at a time for performance portability can also be challenging. Programming paradigms and tools are continuously improving to ease this development process and minimize its time. Yet, we argue that in many cases algorithmic changes are also required, in addition to the programming efforts, to reach the best performance. These changes can modify more or less the original algorithm in order to expose more parallelism levels, higher parallelism degrees or more regular computations. These changes also often require both expertise in HPC and in the application domain: strong algorithmic changes, which can have the highest impact on performance, can only be done with a complete understanding of the algorithm and with a deep knowledge of the application domain, including the possible algorithmic variants. Such breakthroughs involve thus often interdisciplinary research.

Our research work has therefore mainly focused on such *algorithmic changes in order to adapt at best numerical applications in scientific computing to high performance architectures*, while relying on *new and relevant programming paradigms*. This research topic can be related to algorithm-architecture matching, but here the architecture is fixed and we aim to adapt existing algorithms or design new algorithms to exploit at best the selected architecture. Of

[1] See the top of the TOP500 list of the fastest supercomputers in the world: https://www.top500.org/

[2] See the top of the GREEN500 list which ranks the TOP500 list by energy efficiency: https://www.top500.org/green500

course some applications, such as classical HPC kernels (dense matrix multiplication, naive solving of the N-body problem ...), can naturally match new HPC architectures. As far as we are concerned, we have instead focused on *several specific or key applications in scientific computing*, whose efficient deployment on the targeted architecture was *challenging* (and beneficial in the end). We have mainly used *performance as the ultimate criterion*, but we have also considered *power efficiency* when relevant. Besides, while distributed memory parallelism also requires algorithmic changes to scale at best with the number of nodes, this is a well-know and older issue in HPC. Since several of our applications were efficiently exploiting such parallelism via MPI, we have mainly targeted single node improvements on various HPC architectures.

Our work has been structured according to the following research directions.

- **Designing algorithms for many-core or multi-core architectures, or both.** Current HPC architectures can be classified in multi-core or many-core architectures, mainly depending on their number of cores. Multi-core CPUs are designed to maintain a high serial compute power on each core. On the contrary, many-core architectures are based on numerous cores with a lower compute power each: this enables in total greater compute power and greater energy efficiency. But this requires *massive and fine-grained parallelism* on the algorithmic side.

  For a given application, if the most efficient algorithm (in terms of total work performed) is or can be designed as massively parallel, then one can benefit from the greater compute power of many-core architectures. However, for some applications (for example the N-body problem, or the computation of topological abstractions in scientific visualization) there may exist a more efficient algorithm, requiring less total work when performed in serial, but exposing fewer levels or degrees of coarse-grained parallelism. It may then be worth here designing a parallel algorithm for multi-core CPUs which may compete with massively parallel but less efficient algorithms on many-core architectures. *Task parallelism* turned out to be especially relevant for such efficient algorithms with constrained parallelism. Depending on the application and on the possible algorithms, one can therefore choose to design algorithms for many-core or multi-core architectures, or even for both via *hybrid algorithms* which can take advantage of the best of each architecture. The aim here is to determine the algorithm-architecture couple that will lead in the end to the best performance for this application.

  We have hence managed to modify a birth and death process for cell nuclei extraction in histopathology images in order to obtain massive parallelism suitable to GPUs. Conversely, thanks to task-based parallelism we have been able to rewrite for multi-core CPUs a scientific visualization algorithm which was very efficient but intrinsically sequential. Finally, we have shown the interest of a hybrid CPU-GPU algorithm on integrated GPUs for the dual tree traversal of a fast multipole method in astrophysics, when considering performance results as well as power and cost efficiencies.

- **Handling SIMD divergence**. The SIMD (or vector) execution is present on all HPC architectures. While GPUs heavily rely on such execution model, the SIMD share in the overall compute power of CPUs has constantly increased over the past years: starting from 128-bit SSE units, to 256-bit AVX ones, and now to 512-bit AVX-512 ones. Unfortunately, many applications can expose parallel but irregular execution flows which

lower, partially or totally, the SIMD gain. Rewriting the algorithm in order to handle SIMD divergence and to improve SIMD regularity can hence offer important performance gains.

Such contributions have been obtained with the *SPMD*[3]*-on-SIMD programming model* for two applications in computer arithmetic: for generating correctly rounded elementary functions and for numerical validation using stochastic arithmetic.

- **Taking advantage of new heterogeneous architectures**. During the past years, we have seen the introduction of several new heterogeneous HPC architectures: the Cell processor, GPUs, Intel Xeon Phi processors, integrated GPUs ... We have hence been able to study the impact and the relevance of these new heterogeneous architectures in order to take advantage of these for scientific applications. These studies have mostly been based from a performance point of view, but also from a power efficiency point of view (especially for the integrated GPUs). Each time, we have selected an application for which the new architecture could be beneficial in theory, but numerous issues (regarding low level algorithmics and implementation) had to be solved to convert these theoretical advantages in practical gains. Such work has been performed for the fast multipole method on the Cell processor, for atomic physics on GPUs and for seismic imaging on integrated GPUs.

Finally, along these three research directions we have tried as much as possible to design algorithms that could also be suitable to different HPC architectures. This can however raise some issues due to, sometimes subtle, differences at the hardware level. This *performance portability* topic has thus been also investigated for some applications and algorithms.

The rest of this document is organized as follows.

- Chapter 1 offers a brief introduction to HPC from a technical point of view. We aim thereby at defining and briefly presenting the technical terms that will be used in the rest of the document.
- Chapter 2 contains a summary of our research results since our PhD thesis in 2006. These are classified into the three research directions mentioned above.

  We have then chosen to detail two of our main research directions.
- Chapter 3 presents algorithmic contributions for different applications in order to obtain massive parallelism on GPU architecture, to efficiently exploit multi-core CPUs via task parallelism, and to combine multi-core and many-core architectures thanks to a hybrid CPU-GPU algorithm on integrated GPUs.
- The second research focus is detailed in chapter 4 which deals with the handling of SIMD divergence on HPC architectures for two applications in computer arithmetic.

Finally, conclusions and research perspectives are given at the end of the manuscript.

[3]SPMD stands for: single program, multiple data.

Chapter 1

Evolution of HPC architectures and their programming

We briefly describe here the evolution of HPC architectures since mid-2000s, along with their programming.

1.1 Architectures

Since the breakdown of Dennard scaling around 2006, the CPU frequencies have stopped increasing, and even sligthly decreased. Moore's law has since led to the advent of *multi-core CPUs* with an ever increasing number of cores. In 2018, we can target CPUs with up to 32 cores, with twice as many hardware threads (thanks to 2-way SMT - simultaneous multi-threading), within HPC nodes containing multiple sockets, possibly implying NUMA (non-uniform memory access) effects. In the meantime, CPU vector (or SIMD - single instruction, multiple data) units have also increased in size from 128-bit SSE units, to 256-bit AVX[2] units, and now to 512-bit AVX-512 units. The SIMD share in the overall compute power of CPUs has thereby constantly increased over the past years: SIMD computing offers indeed important performance gains at a relatively low hardware cost. In a HPC context, we usually use one thread per (logical or physical) CPU core: memory and compute latencies have therefore to be overlapped thanks to caches and to ILP (instruction-level parallelism).

Also starting from mid-2000s, we have seen the advent of *heterogeneous architectures* in HPC, which offer increased raw compute-power, as well as increased power efficiencies, by differing from standard CPUs in the usage of the chip transistors. While an important share of a standard CPU chip is devoted to memory caches, instruction control (out-of-order execution, branch prediction ...) and to comptatibility with previous architectures, these heterogeneous architectures reduce this share in favor of an increased number of (simpler) compute units.

The Cell processor has hence enabled to break the Petaflop barrier (10^{15} flop/s) in 2009 thanks to its heterogeneous architecture. This architecture was composed of a general-purpose core and of 8 HPC cores with in-order execution and 128-bit SIMD units, offering more than 200 Gflop/s in single precision. Each HPC core had its own local store instead of a cache. This local store had to be manually managed through explicit data transfers from/to

the main memory over a 200 GB/s internal bus.

Concurrently, Graphics Processing Units (GPUs) have been increasingly used for high-performance scientific computing. Each GPU core relies also on in-order execution, without branch prediction, and the GPU caches are much smaller than the CPU ones. This enables a much higher number of cores[1] (hundreds, and nowadays, thousands of cores) with however lower frequencies than on CPU, resulting in a *many-core* architecture. These GPUs offer theoretically one order of magnitude greater compute power and internal memory bandwidth than multi-core CPUs. Contrary to CPU cores, best performance is usually obtained by overloading as much as possible each GPU core with numerous threads in order to overlap memory and compute latencies. Besides, GPUs rely on an implicit SIMD execution for both computations and memory accesses. The SIMD width is 32 on NVIDIA GPUs (a *warp*) and 64 on AMD GPUs (a *wave-front*), which is greater than CPU SIMD widths. Overall, best GPU performance is therefore obtained for massive, regular and fine-grained data parallelism. It also has to be noticed that data has first to be transfered over a PCI Express bus with limited bandwidth (lower than 15.8 GB/s on PCI Express 3.0), which can bottleneck the overall GPU performance.

Starting from 2010-2011, AMD (with its APUs - accelerated processing units) and Intel have introduced integrated GPUs (iGPUs). The iGPU cores share the same die as the CPU cores and can directly access the CPU main memory (without PCI transfers) via specific *zero-copy buffers* where a large fraction of the main memory can be allocated. This enables to avoid explicit copies between the main memory and the GPU memory, to alleviate the possible performance bottleneck on discrete GPUs due to the PCI bus, and to allocate more memory than within a discrete GPU. Moreover, these iGPUs (along with their CPU) are usually more compute powerful than standard CPUs. They also offer reduced memory consumptions, especially compared to a discrete GPU with a dedicated CPU, thanks to lower GPU core frequencies and to lower CPU performance. However their compute power and memory bandwidth are lower than discrete GPU ones (see e.g. [SFLC17] for details). The possible performance gains over discrete GPUs depend thus on the application and algorithm features (frequency and volume of PCI transfers, proportion of work deported on GPU ...). Regarding Intel iGPUs, it has to be noticed that each compute unit contains two SIMD FPUs, concurrently performing four 32-bit flops each, but on the software side the SIMD width can range from 1 to 32 (see [78] for the Intel Graphics Gen8 architecture).

Another many-core architecture has been proposed by Intel with the Xeon Phi coprocessors. The first generation (*Knights Corner*, KNC) was only available accross a PCI Express bus and offered up to 61 x86 in-order cores, with 4-way SMT (4 hardware threads per core) and 512-bit SIMD units. The second generation (*Knights Landing*, KNL) is available as a standalone processor, offers up to 72 cores, with out-of-order execution and AVX-512 SIMD units. Other recent many-core architectures include the PEZY-SC2 chip (2048 cores with 8-way SMT), the Matrix-2000 accelerator (128 cores) or the Sunway SW26010 processor (260 cores with a heterogeneous architecture similar to the Cell processor).

1.2 Programming

For more than 20 years, the HPC standard for multi-process parallel programming (on shared or distributed architectures) has been MPI. For multi-thread parallelism, there ex-

[1] Note however that a GPU core does not match a CPU core, but rather a CPU SIMD lane.

ist different possibilities. Explicit, low-level programming is available with POSIX threads, which were the basis of the Cell programming (along with SIMD intrinsics and explicit DMA transfers). Higher-level programming is available for example via compiler directives in OpenMP [2]. *Task parallelism*, originally introduced in Cilk, has also gained a greater interest in the last 10 years, thanks to progress achieved by advanced task runtimes: StarPU [9], OmpSs [47], Swan [150] ...(see [144] for a review). This task parallelism has been progressively introduced in OpenMP for irregular or nested parallelism: independant tasks first; then data-dependencies, priorities ... Task parallelism is also available in the Intel TBB (Threading Building Blocks).

While these programming paradigms can also be used to deploy applications on the Xeon Phi many-core architecture, GPUs require different paradigms based on SPMD (single program, multiple data) programming. CUDA has first generalized the use of NVIDIA GPUs for HPC scientific computing. A CUDA program consists in GPU codes (kernels) and in a host code running on the CPU that can launch these kernels on the GPU, each kernel being executed for each thread of each block within a grid of blocks. With a programming similar to CUDA, the OpenCL standard has then been released to support most HPC architectures: NVIDIA and AMD GPUs, Intel iGPUs, multi-core CPUs, Xeon Phi coprocessors (Knights Corner, but not Knights Landing), the Cell processor, FPGAs (field-programmable gate arrays) ... An OpenCL kernel is written in the same SPMD style as a CUDA kernel, is launch from the host on the device via a command queue, and is executed for each work-item of each work-group of a ND-range. Higher-level approaches also exist for GPU programming, such as the ones based on compiler directives in the host code: see HMPP [15, 45], OpenACC [1] and OpenMP [2].

Finally, as far as SIMD programming on CPU is concerned, several programming paradigms are possible. Firstly, intrinsics enable the use of vector instructions without using assembly language by relying on vector-specific functions. Using such intrinsics is always possible, but rather tedious in general: one has to write specific code for each intrinsic set (SSE, AVX, AVX-512 ...) and each vector width. Secondly, automatic vectorizing is a compilation technique in which the compiler analyses the code and decides whether it is possible and efficient to vectorize it. In practice, this can be limited by the compiler ability to perform the dependency analysis, as well as by pointer aliasing issues, memory misalignments, nested loops, function calls within loops ... Thirdly, the newest versions of OpenMP (starting from OpenMP 4.0), as well as the Intel compiler and Intel Cilk Plus, contain compiler directives aimed at vectorizing loops. Using such directive-assisted vectorization, the user can force the vectorization and circumvent the limitations of automatic vectorization. Finally, another possibility to exploit the SIMD units is to rely on the *SPMD-on-SIMD* programming model [57, 119]. All computations are written as scalar ones and it is up to the compiler to merge such scalar computations in SIMD instructions. The main advantages are the ease of programming and the portability: the programmer needs neither to write the specific SIMD intrinsics for each architecture, nor to know the vector width, nor to implement data padding with zeros according to this vector width. The vector width will indeed be determined only at compile time (depending on the targeted hardware). Moreover, like compiler directives, no data dependency analysis is required by the compiler: it is up to the user to ensure that the scalar computations can be processed correctly in parallel. Such programming paradigm has been increasingly used in HPC: first on GPUs with CUDA and then on various compute devices with OpenCL. On CPU, such programming model is available in OpenCL (OpenCL implicit vectorization), as well as in the Intel SPMD Program Compiler (`ispc`) [119].

Chapter 2

Research summary

In this chapter, we present the research results we have obtained since the defense of our PhD thesis in 2006: first with an INRIA postdoctoral position at LAM (Laboratory of Astrophysics in Marseille), then starting from 2007 with an assistant professor position at *Université Pierre et Marie Curie* (UPMC, now *Sorbonne Université* since 2018) in the PEQUAN team (Performance and Quality of Numerical Algorithms) of the Scientific Computing department of the computer science laboratory (LIP6).

The first three sections summarize our results according to the three research directions presented in the introduction of the manuscript. Section 2.1 deals with the design of massively parallel algorithms for many-core architectures, of task-based algorithms for multi-core CPUs, and of hybrid algorithms on integrated GPUs. Section 2.2 presents how regular computations can be obtained for best SIMD performance, thanks to a strong algorithmic rewriting or to a specific implementation of the low level operations. Section 2.3 shows how we have been able to take advantage of the emergence of several new heterogeneous architectures for specific applications in scientific computing.

Besides, we have also worked on another topic dealing with the automatic multi-thread parallelization of domain-specific sequential code. So, we describe in Sect. 2.4 how we have managed to automatically generate efficient parallel code for shared memory architectures within a data assimilation framework.

Our publications are listed in Appendix A, and the supervised or co-supervised students are listed in Appendix B.

2.1 Massive parallelism, task parallelism and hybridization

We present here how we have managed to design a massively parallel algorithm for many-core architectures (Sect. 2.1.1), a task-based parallel algorithm for multi-core architectures (Sect. 2.1.2), and a hybrid CPU-GPU algorithm on integrated GPUs (Sect. 2.1.3).

2.1.1 Massive parallelism for image segmentation

Histopathologists, who study diseased tissues at microscopic level, are facing an ever increasing workload, especially for breast cancer grading. Automatic image analysis can here greatly reduce this workload and help improving the quality of the diagnosis [152]. In order

to perform the detection and extraction of cell nuclei in histopathology images for breast cancer grading, we have considered a marked point process combined with a "birth and death" algorithm. While this process offers good quality results, it is extremely compute intensive, and the targeted images can be huge. We have thus studied (with C. Avenel and D. Béréziat) its parallelization and its performance scaling on the number of cores and on the number of nuclei (hence on the image size) [AFB13]. Due to the intrinsic sequentiality of the death step of the original algorithm, we have proposed a new overlapping energy in order to design a new parallel death step, leading to an overall massively parallel algorithm. We have hence obtained very good speedups on multi-core CPUs (11× on 12 CPU cores), as well as on a GPU many-core architecture. Thanks to a specific deployment including a new scan of the ellipse area, the GPU is indeed twice faster than two multi-core CPUs. This work is presented in Sect. 3.1.

2.1.2 Task parallelism for scientific visualization

In scientific visualization, the merge trees [20, 132] and the contour trees [18, 26, 141] are fundamental topology-based data structures that enable the development of advanced data analysis, exploration and visualization techniques, for example on 3D tetrahedral meshes. Their augmented version enable all visualization applications including topology-based data segmentation and tree simplification to discard noisy data. In order to efficiently process ever larger scientific data, highly efficient parallel algorithms are required to build such trees on a visualization workstation with multi-core CPUs. The reference algorithm of Carr et al. [26] is however intrinsically sequential as a global view on the data is required for the efficient processing of each mesh vertex.

In [GFJT16] we have proposed, with C. Gueunet, J. Jomier (from Kitware) and J. Tierny, a parallelization of this algorithm on multi-core CPUs in shared memory based on a static decomposition of the range of the vertex values among the different threads. This approach leads to good speedups, but these are limited by extra work at the partition boundaries and by load imbalance among the partitions. Such extra work issue also applies to another multi-thread parallelization of the augmented merge-tree computation based on partitions of the geometrical domain [115]. On the other hand, massively parallel approaches (such as [99]) cannot produce augmented trees and lead to moderate speedups on multi-core CPU (or with a hybrid CPU-GPU deployment) when compared to the reference sequential algorithm: the sequential execution of this massively parallel algorithm is indeed up to three times slower than the reference implementation [44] of the optimal algorithm of Carr et al. [26].

In [GFJT17], we have therefore completely revisited the augmented merge tree computation on shared memory multi-core architectures by partitioning the tree construction at the arc level and by using independent arc constructions which can be expressed using tasks. This does not introduce extra work in parallel, and we can naturally benefit from the dynamic load balancing of the task runtime. This has also made it possible to accelerate the processing when there is only one task left. To our knowledge this implementation is the fastest to compute the augmented merge trees in sequential, being more than 1.5x faster for most data sets than the reference sequential implementation [44]. In parallel, we outperform our previous implementation [GFJT17] by a factor 5.0 (in average). This work is presented in Sect. 3.2.

Regarding the contour tree computation, we have recently improved our task-based approach in [GFJTxx] thanks to a new parallel algorithm to combine the two merge trees, and

thanks to a complete "taskification" of the algorithm which enables us to overlap the computations of the two merge trees in order to provide more tasks to the runtime (up to 1.34x performance gains).

2.1.3 Hybridization for astrophysical N-body simulations

After our PhD thesis dealing with HPC for fast multipole methods (FMMs), our postdoctoral study at LAM (Laboratory of Astrophysics in Marseille) has lead to a detailed comparison [FAL11] of hierarchical methods for astrophysical N-body simulations (used for galactic dynamics studies). This comparison has encompassed $\mathcal{O}(N \log N)$ Barnes-Hut tree-codes and $\mathcal{O}(N)$ FMMs (presented in Sect. 3.3.1), and has shown that Dehnen's algorithm [39], implemented in the *falcON* code, is one order of magnitude faster than serial executions of Barnes-Hut tree-codes and outperforms serial executions of a standard FMM. Dehnen's algorithm is a FMM specific to astrophysics: it relies in particular on a double recursion down the octree (*dual tree traversal*, DTT) which is well suited for the highly non-uniform astrophysical distributions of particles. However, at that time *falcON* was only serial since 2002.

In [LF14], we have (with B. Lange) parallelized *falcON* in the *pfalcON* code on multi-core CPUs thanks to task-based parallelism for the DTT, and to the SPMD-on-SIMD programming model for vectorization. *pfalcON* has been shown to be competitive with a Barnes-Hut tree-code on a high-end GPU. One then naturally aims at combining the best algorithm (FMM with DTT) with the most powerful hardware currently available (GPU). Due to its double recursion, obtaining an efficient DTT on many-core architectures like GPUs is however challenging. This has been achieved in [FTxx] (with M. Touche) with a new hybrid CPU-GPU algorithm on integrated GPUs (iGPUs): while the DTT is performed on the CPU cores (with task-based parallelism), all the computations can be efficiently performed on the iGPU cores (after specific optimizations). Using OpenCL, we have been able to target both Intel iGPUs and AMD APUs. Thanks to its lower SIMD width and its greater compute-power, the Intel iGPU performs in the end better than the AMD APU, and can match the performance of two standard CPUs, of one high-end CPU, or even of a GPU tree-code, being hence up to 4.2x more power-efficient (based on the theoretical TDP values) and 6.2x more cost-efficient than these architectures. This work is presented in Sect. 3.3.

2.2 Improving computation regularity

We present here how we have managed to handle SIMD divergence for two applications in computer arithmetic.

2.2.1 HPC for solving the Table Maker's Dilemma

In floating-point arithmetic, having fully-specified operations is a key-requirement in order to have portable and predictable numerical software. Since 1985 and the IEEE-754 standard (revised in 2008), the four arithmetic operations $(+, -, \times, /)$ are specified and must be correctly rounded: the computer must return the floating-point number that is the nearest to the exact result. This is not fully the case for the basic mathematical functions which may thus return significantly different results depending on the environment. Hence, it is almost impossible to estimate the accuracy of numerical programs using these functions, and their

portability is difficult to guarantee. This lack of specification is due to a computer arithmetic problem called the *Table Maker's Dilemma*. To compute the result $f(x)$ of a function f in a given format, where x is a floating-point number, we first compute an approximation to $f(x)$ and then we round it to the nearest floating-point number. The Table Maker's Dilemma consists in determining the accuracy of the approximation to ensure that the obtained result is always equal to $f(x)$ rounded to the nearest floating-point number. To solve this problem, we must locate, for each considered floating-point format and function f, what is the *hardest to round* point, i.e., the floating-point number x such that $f(x)$ is closest to the exact middle of two consecutive floating-point numbers. The exhaustive search over all floating-point numbers being prohibitive, specific algorithms [95, 136] can solve this problem but still require extremely long computation times (several years of CPU time at the time). The use of HPC is therefore crucial in order to obtain reasonable computation times. In the long term, such work should enable to require the correct rounding of some functions in a future version of the IEEE-754 standard, which will allow to completely specify all the components of numerical softwares.

Focusing on double precision, we have studied (with M. Gouicem [58] and S. Graillat) the Lefèvre algorithm [95] which offers massive and fine-grained data-parallelism over numerous sub-domains of the definition domain of f. For each sub-domain, two steps are performed: first the generation of an affine approximation of f, and then a search for *hard to round* points (HR-cases).

While the affine approximation generation has a regular control flow, we have first shown that the original HR-case search of Lefèvre algorithm presents divergence issues when executed on SIMD architectures [FGG12]. Thanks to the continued fraction formalism, we have rewritten in [FGG16] this algorithm in order to strongly reduce this divergence in the execution flow, leading to a GPU performance gain of 3.4x over the original algorithm on GPU, and of 7.4x over the original algorithm on a multi-core CPU. Thanks to the SPMD-on-SIMD programming model and to the OpenCL portability, we have then shown in [AFGZ16] (with C. Avenel, M. Gouicem and S. Zaidi) that such algorithm is more efficiently deployed on GPU (NVIDIA or AMD) SIMD units than on CPU or Xeon Phi (Knights Corner) ones. This is partly due to the dynamic hardware handling of divergence on GPUs, which is better suited for our regular algorithm than the static software handling of divergence on CPUs. This regular algorithm presents indeed important divergence in its control flow, but low divergence in its execution flow. This work on SIMD divergence handling for the Table Maker's Dilemma is presented in Sect. 4.1.

As far as the affine approximation generation is concerned, we have managed to obtain performance gains similar to the HR-case search thanks to a hybrid CPU-GPU deployment [FGG16]. In the end, for the complete solving of the Table Maker's Dilemma, we obtain overall speedups of up to 7.1x on one GPU over the original Lefèvre algorithm on one multi-core CPU. The (roughly) 2 months of computation time on a multi-core CPU currently required to process all binades of a given function can thus be reduced to (roughly) 1 week of computation time on a GPU.

2.2.2 High performance stochastic arithmetic

Numerical validation consists in estimating the rounding errors occurring in numerical simulations because of the finite representation of floating-point numbers in computers. When these rounding errors pile up, the result of a floating-point computation can greatly differ

from its exact result. In the context of high performance computing, new architectures, becoming more and more parallel and powerful, offer higher floating-point compute power. Thus, the size of the problems considered (and with it, the number of operations) increases, becoming a possible cause for increased uncertainty. As such, estimating the reliability of a result at a reasonable cost is of major importance for numerical software.

Several approaches exist for the numerical validation of scientific codes: interval arithmetic, backward error analysis or probabilistic methods. We have here focused on Discrete Stochastic Arithmetic [153], the probabilistic approach implemented in the CADNA[1] library. Discrete Stochastic Arithmetic considers several executions of each operation with a randomly chosen rounding mode. From the samples obtained, we can get, through statistical analysis, a 95 % confidence interval on the number of exact significant digits of the result. CADNA has been successfully used for the numerical validation of real-life applications [19,80,81,106,129] written in C, C++, or Fortran. At the start of this work, there already existed an MPI version of CADNA [105] as well as a prototype of CADNA on GPUs [84], but CADNA was not best suited for HPC. Its overhead on execution time was usually between 10 and 100 times [80] and could go up to several orders of magnitude on highly optimized codes [105]. Furthermore, it could not use one of the mainstay of parallel computing on CPU, namely SIMD (or vector) instructions, because of the explicit rounding mode change required for each scalar operation.

As part of a collaboration with P. Eberhart, J. Brajard and F. Jézéquel, we first have managed in [EBFJ15] to reduce the scalar overhead by up to 85%, while also enabling the support of SIMD codes with the SPMD-on-SIMD programming model and thus leading to additional speedups between 2.5 and 3. This has been accomplished thanks to an emulation of the floating-point operations with an opposite rounding mode, to function inlining, to a new random number algorithm and implementation and to the SPMD-on-SIMD model: this work is detailed in Sect. 4.2.

As GPUs also (partially, but heavily) rely on SIMD execution, we have also been interested in the impact of some of these improvements on the CADNA-GPU prototype. We have thus managed to improve the performance of this CADNA-GPU prototype by up to 61%: see [ELB+18] and Sect. 4.2.4. Finally, we have also enabled the support of OpenMP codes by CADNA in order to estimate the round-off error propagation in multi-threaded OpenMP codes: see [EBFJ16] and Sect. 4.2.4.

2.3 Taking advantage of new heterogeneous architectures

We present here how we have managed to benefit from new heterogeneous architectures (at that time: the Cell processor, GPUs, and integrated GPUs) for specific applications.

2.3.1 The Fast Multipole Method on the Cell processor

During our PhD thesis, we had started working on the fast multipole method (FMM) for Laplace equation in astrophysics and in molecular dynamics. This algorithm, considered as one of the most important in scientific and engineering computing [46], solves the N-body problem with a linear operation count for any given precision. Thanks to a hierarchical octree data structure, the potential or force field is decomposed in a near field part, directly

[1] Control of Accuracy and Debugging of Numerical Applications: http://cadna.lip6.fr

computed, and a far field part approximated with multipole and local expansions. The FMM is considered as a challenging HPC application since this algorithm is highly non-trivial and presents several phases with different computational intensities and different (possibly irregular) memory access patterns [93]. As part of our PhD thesis, we had presented a matrix formulation of the most time-consuming operator of the far-field computation (i.e. the *M2L* operator - *multipole-to-local*), which had enabled us to use level 3 BLAS routines (Basic Linear Algebra Subprograms) leading to significant speedups. For the targeted precisions, this approach outperformed the existing enhancements (FFT, rotations and plane waves), in case of both uniform [CFR08] and non uniform distributions [CFR10] (thanks to a new octree data structure). Thanks to a hybrid MPI-thread (POSIX threads) approach we had also parallelized this FMB code (*Fast Multipole with BLAS*) on shared and distributed memory architectures. A static octree decomposition allowed for load balancing and data locality (via cost functions and Morton ordering of the octree cells), and a mutual exclusion mechanism prevented write/write conflicts among the threads.

After our PhD thesis we have targeted the efficient deployment of the FMM on the hardware accelerators (HWAs) available at that time, namely the Cell processor and the GPUs. At that time, only N-body simulations via direct computation [8, 87] or with cut-off radius [49,98,138] had already been implemented on the Cell processor. The FMM had however already been deployed on GPUs for both uniform [60,160] and non-uniform [29,72,93,120] distributions of particles. The far field part was generally less efficiently implemented on GPU than the near field part (direct computation) [29, 60, 93]. Moreover, such implementations were based on a thorough deployment of the near field and/or far field computations of the FMM on the GPU, along with all the data structures, which required important algorithmic changes and programming efforts. This resulted in GPU or CPU-GPU implementations that outperformed CPU ones (highly optimized CPU implementations could also reduce the CPU-GPU performance gap [29]).

Given a new HWA, our approach has rather focused on offloading only the most time consuming FMM computations (the direct computations and the *M2L* operations) on this HWA following a performance-portability tradeoff. Thanks to the FMB matrix formulation, and provided that BLAS routines are available on the HWA, we could indeed benefit from a straightforward and highly efficient implementation of the *M2L* operations on this HWA, for any required precision. Contrary to other FMM implementations on HWA, we did not had to write and highly optimize specific far-field computations for each new HWA. Moreover, we could rely on the ability of the FMB code to group multiple *M2L* operations into one single matrix-matrix product (for efficient processing with level 3 BLAS routines): this enabled us to increase the computation grain, for both uniform and non-uniform distributions, and thus to offset the cost of offloading *M2L* operations on the HWA. A similar idea had been developed for GPUs in [140], where multiple *M2L* operations are performed at once (but without BLAS routines) on the GPU. Regarding the direct computation of the N-body problem, this key application in HPC is among the first ones to be efficiently implemented on new HWAs [8, 87, 111]. The direct computation could thus also be efficiently performed within our FMM deployment on a new HWA.

The most time consuming FMM operations correspond to small or medium computation grains, which are moreover involved in irregular computation schemes due to the possible non-uniform distributions of particles. That is why we have targeted (with J.L. Lamotte) the heterogeneous architecture of the Cell processor, whose internal bus has lower latency and higher bandwidth than the PCI Express bus of GPUs. Thanks to specific low-level algorith-

mics, regarding computation kernels, communication overlapping with computation, load balancing, task scheduling, conflict handling and synchronization overheads, we have first very efficiently deployed the near-field direct computations on the Cell processor [FL09]. Our code compared favorably with previous results in the literature for direct N-body computations on the Cell processor as well as on GPU, especially for low numbers of particles per octree leaf. Since, unfortunately, the latest IBM Cell SDK [74] did not provide efficient BLAS routines for complex numbers on the Cell processor, we had to write our own (simple but efficient) matrix-matrix multiplications, achieving up to 92.38% and 90.96% of the Cell peak performance in single [BFL10] and double [FL13] precisions, respectively. To our knowledge, we have in the end proposed the first (and last) deployment of the FMM on the Cell processor [FL13]. Thanks to suitable task scheduling and data handling within each Cell processor, our implementation scaled efficiently on several Cell blades, in both single and double precisions, and for both uniform and non-uniform particle distributions.

2.3.2 Atomic physics on GPUs

This section describes the deployment of a simulation in atomic physics (the PROP program) on GPUs. The PROP program is part of the 2DRMP [24, 128] suite which models electron collisions with H-like atoms and ions at intermediate energies. The primary purpose of the PROP program is to propagate a global R-matrix [23], $\Re$, in the two-electron configuration space. The propagation needs to be performed for all collision energies, for instance, hundreds of energies, which are independent. Propagation equations are dominated by matrix multiplications involving sub-matrices of $\Re$. However, the matrix multiplications are not straightforward in the sense that $\Re$ dynamically changes the designation of its rows and columns and increases in size as the propagation proceeds [137].

In a preliminary investigation PROP was selected by GENCI[2] and CAPS,[3] following their first call for projects (2009–2010) for GPU deployments. At that time, CAPS was delivering HMPP (Hybrid Multicore Parallel Programming) [15, 45], a hybrid and parallel compiler that relies on compiler directives to deploy, compile and execute legacy codes on heterogeneous architectures like GPUs. Such approach has since then been reused in OpenACC [1] or in OpenMP 4.X [2]. Using HMPP, CAPS developed a version of PROP in which matrix multiplications are performed on the GPU or the CPU, depending on the matrix size. Unfortunately this partial GPU implementation of PROP did not offer significant acceleration (speedup of 1.15x between one NVIDIA Tesla C1060 GPU and one Intel Xeon x5560 quad-core CPU) due to the use of double precision and to the small or medium sizes of most matrices.

As part of a collaboration with R. Habel, F. Jézéquel, J.-L. Lamotte and N.S. Scott (from Queen's University of Belfast, UK) we have first studied in [FHJ+11] [FHJ+13] the numerical stability of PROP using single precision arithmetic. The peak performance ratio between single and double precisions varies indeed between 2 to 4 on HPC GPUs and the GPU memory accesses and CPU-GPU data transfers are faster in single precision because of the format length. This study has shown that PROP using single precision, while relatively stable for some small cases, gives unsatisfactory results for realistic simulation cases above the ionization threshold where there is a significant density of pseudo-states.

[2] GENCI: Grand Equipement National de Calcul Intensif, http://www.genci.fr/en

[3] CAPS was a software company providing products and solutions for many-core application programming and deployment.

We have thus deployed all computation steps of PROP (in double precision) on GPUs in order to avoid the overhead generated by data transfers. In addition to all matrix products, other matrix operations (constructions, copies, additions, scalings, linear system solvings) are performed on the GPU. Moreover I/O operations on CPU are overlapped by GPU computations thanks to double-buffering with POSIX threads, and matrices are dynamically padded to match the inner blocking size of CUBLAS [110] and MAGMA [100]. These improvements have led to a speedup of 4.6x (respectively 9.0x) for one C1060 (resp. C2050) GPU over one Intel Q8200 quad-core CPU [FHJ+11], which justifies the deployment work. Starting from Fermi GPU architectures, we also rely on concurrent kernel executions on GPU, both among propagations of multiple energies and within each energy propagation: this is relevant here due to our small or medium computation grain kernels, and this leads to an additional gain of around 15% [FHJ+13]. Overall, this work has shown the relevance of GPUs for such a simulation in atomic physics.

2.3.3 Leveraging integrated GPUs for seismic imaging

In order to discover and study new oil deposits, oil and gas companies can send acoustic waves through the subsurface and collect the echoes reflected by the rock layers. Seismic imaging algorithms are then used to delineate the subsurface geologic structures from the collected data. Reverse Time Migration (RTM) is the most famous algorithm for seismic imaging and is widely used by oil and gas companies. However, RTM requires large memory capacities and long processing times. HPC solutions based on CPU clusters and hardware accelerators are thus widely embraced. Clusters of GPUs have hence been used since 2009 to accelerate RTM [4, 51, 96, 102, 108, 112, 114] thanks to their increased compute power and to their high memory bandwidth. However, the performance of these GPU deployments are limited by the GPU memory capacity, by the high (CPU+)GPU power consumption, and by the frequent CPU-GPU communications that may be bottlenecked by the PCI transfer rate. These CPU-GPU communications are due to required data snapshots (within each node) and to inter-node communications: depending on the number of compute nodes and on the data snapshotting frequency, these can bottleneck the performance of multi-node RTM GPU implementations [4, 33, 51, 112]. Although numerous software solutions, such as temporal blocking or overlapping the PCI transfers with computations, have been proposed to address this issue, they require extensive programming efforts.

As part of a collaboration (funded by TOTAL) between TOTAL (H. Calandra), AMD, CAPS Entreprise (R. Dolbeau) and LIP6 (I. Said [126] and J.-L. Lamotte), we have therefore considered the AMD APUs (*Accelerated Processing Units*) and their integrated GPUs (iGPUs, see Sect. 1.1) as an attractive hardware solution to the PCI bottleneck of the RTM, while allowing the iGPU cores to exploit the entire system memory and offering reduced memory consumptions. However, these APUs are almost one order of magnitude less compute powerful and have a lower memory bandwidth than discrete GPUs, which can lower or annihilate these assets in practice.

We have thus first studied the relevance of successive generations of APUs for high performance scientific computing, based on data placement strategies and on memory and applicative (a matrix-matrix multiplication and a 3D finite difference stencil) benchmarks [CDF+13] [SFL+16]. Theses OpenCL benchmarks have been carefully optimized for the three considered architectures: CPUs, discrete GPUs and APUs (using only the iGPUs of the APUs). In addition to a performance portability study, we have shown that the APU

iGPUs outperform CPUs and may outperform discrete GPUs for medium-sized problems (matrix-matrix multiplications) or for problems with high PCI communication requirements (finite difference stencils with data snapshotting). Moreover, our study has shown that APUs are more power efficient (up to 20%, when considering the complete compute node) than discrete GPUs. The feasibility of hybrid computing on the APU (using both the CPU and the iGPU, with either task-parallel or data-parallel deployments) has also been surveyed [ESFC14].

We have then extended this CPU/APU/GPU study for seismic applications (seismic modeling and RTM), on one node (using Fortran90+OpenCL and real power measurements) and on up to 16 nodes (using MPI in addition to Fortran90+OpenCL and estimated power efficiencies) [SFLC18]. We have shown the relevance of overlapping both I/O and MPI communications with computations on APU and GPU clusters, where all computations are performed on the APU iGPUs or on the discrete GPUs. In the end, APUs deliver performances that range between those of CPUs and those of GPUs, and the APUs can be as power efficient as the GPUs.

2.4 Parallel code generation for data assimilation

In environmental science, climatic events are usually modeled through very large numerical codes. Many parameters of these models are estimated by assimilation of in-situ or satellite observations. Using data assimilation one can indeed reduce the differences between the forecasts of the model and the actual observations. In variational data assimilation methods, this is achieved thanks to the minimization of a cost function whose gradient is computed based on the adjoint model. From the direct model, that runs the simulation, we can indeed deduce the adjoint model that computes the sensitivity of all the parameters with respect to the actual observations. Programming the adjoint model is a difficult task that can even require a complete rewriting of the direct model. These issues have led to the development of the YAO framework by the LOCEAN laboratory[4] at Sorbonne Université. YAO eases the development of software for variational data assimilation [109] thanks to a *modular graph* which defines the data dependencies between smaller units of computations (modules) written by the user. This modular graph then enables the code generation of both the direct and the adjoint models. This approach has proved suitable for small and medium sized codes.

In order to process larger problems efficiently, and targeting shared memory architectures, we have investigated (with L. Nardi, F. Badran, and S. Thiria from LOCEAN) the generation of parallel OpenMP codes with YAO, by taking advantage of the similarity between the modular graph and the reduced dependence graph used in automatic parallelization [NBFT12]. While data races (write/write conflicts) arising in the adjoint code prevent the use of generic software for automatic parallelization (such as for example CAPO [82], Gaspard2 [139] or PLuTo [16]), the parallel code generation by YAO can handle these data races automatically thanks to OpenMP *atomic* directives. This leads however to limited speedups (below 50% of parallel efficiency). We can avoid these atomic operations thanks to a sub-domain decomposition automatically generated (along with the suitable sub-domain size) thanks to the modular graph. This has enabled us to improve speedups up to 9.4x on 12 CPU cores [NBFT12].

[4] *Laboratoire d'Océanographie et du Climat : Expérimentations et Approches Numériques*, Oceanographic & Climate Laboratory: experiments and digital approaches, https://www.locean-ipsl.upmc.fr

Chapter 3

Designing algorithms for many-core or multi-core architectures, or both

3.1 Massive parallelism on many-core architecture for image segmentation

We first focus on an image segmentation application for breast cancer grading which has been rewritten to enable massive and fine-grained parallelism. Breast cancer grading in histopathology – the study of diseased tissues at microscopic level – is strongly based on the size and aspect of nuclei. Small cell nuclei of almost same sizes will denote a small grade, whereas a marked size variation will conduct to a higher grade. The detection and extraction of nuclei (see Fig. 3.1) are thus important issues in the domain of automatic image analysis of histopathology images. Moreover, the reduction of the computation time is becoming critical, because of the huge microscope slide sizes (up to $100,000$ by $100,000$ pixels).

Extraction algorithms can be divided into two main categories: classification and segmentation. The classification ones, see [158] for instance, will not be able to separate and count nuclei. The segmentation can be performed using active contour model (Snakes) [85], but as one snake would be needed for every nucleus, this solution is not relevant here. The segmentation can also be performed using the level set approach using free shapes [65, 151], or parametric ones [21]. A parametric shape based model decreases the amount of computation, and is sufficient for a good cell nuclei detection here. However, in breast cancer images the nuclei often appear joint or even overlapped. We thus consider here a marked point process, which enables us to extract nuclei as individual joint or overlapping objects without necessarily discarding overlapping parts and therefore without major loss in delineation precision [88]. Various authors applied point processes to image analysis [11, 42]. We use here a simulated annealing algorithm combined with a "birth and death" process as in [43].

This process is extremely compute intensive, especially on large images: its parallelization is therefore crucial, as well as its good scaling on the number of nuclei, hence on the image size. Several images can of course be processed in parallel on multiple compute nodes. Since this application is primarily intended for histopatologists in hospitals where only one single workstation will be available, we focus here on the parallel processing of a given image on one single node, targeting in particular massive and fine-grained parallelism for many-core architectures. To our knowledge, such birth and death process has not been

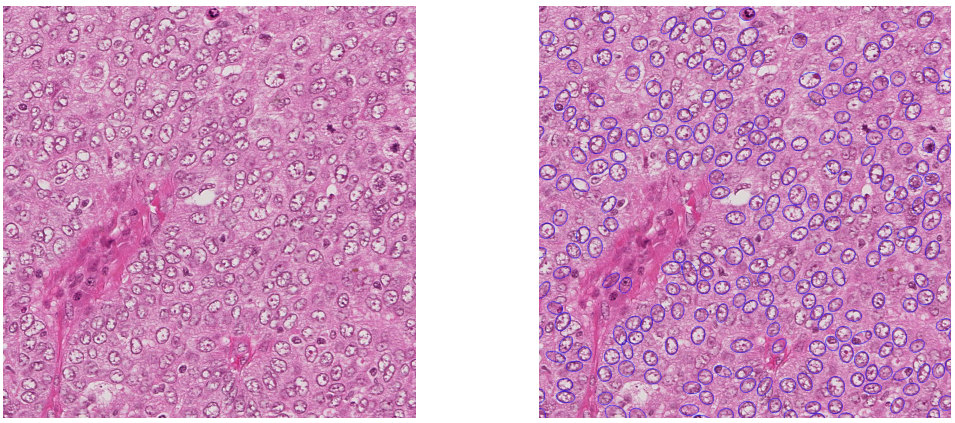


Figure 3.1 – Example of a good quality detection on a histopathology image

deployed on parallel architectures yet.

In the following, we first present the original birth and death process applied to histology images and we show that one step of this algorithm is inherently sequential. We then detail how we have revised this algorithm in order to obtain a parallel birth and death algorithm that scales on the number of cores and on the number of nuclei. Finally, we present performance results on both multi-core CPUs and GPU. More details can be found in [AFB13].

3.1.1 Birth and death process for cell nuclei extraction

We consider the framework of histopathology images, but this work could easily be generalized for the detection of any elliptically shaped object. The goal here is to perform the cell nuclei detection (as in Fig. 3.1). We give here a high-level description of the algorithm, and we refer the reader to [10] [AFB13] for the detailed theoretical background.

3.1.1.1 Marked Point Process and original birth and death algorithm

A Marked Point Process (MPP) can be used to detect an arbitrary number of objects. MPP is a stochastic process where a realization $\mathbf{w}$ is a set of marked points $w_1, ..., w_n$. As cell nuclei may be correctly approximated by ellipses (see [10] for a justification), an object w_i will be described by the center x_i of an ellipse, with small and big axes $a_i, b_i \in [r_{min}, r_{max}]$, and with an orientation $\theta_i \in [0, 2\pi]$.

The aim here is to determine the realization $\mathbf{w}$ which minimizes the energy function U:

$$U(\mathbf{w}) = \gamma_d \sum_{w_i \in \mathbf{w}} U_d(w_i) + \gamma_p \sum_{w_i \in \mathbf{w}} U_p(w_i) \quad , \tag{3.1}$$

where $U_d(w_i)$ are the data-fidelity terms, their sum measuring the relevance between $\mathbf{w}$ and the image, and $U_p(w_i)$ are the interaction terms, their sum measuring the coherence of $\mathbf{w}$. The parameters γ_d and γ_p are weighting coefficients. The goal of the data-fidelity term is to evaluate the relevance of an ellipse in the image: for instance an ellipse w_i correctly placed should give a low value for $U_d(w_i)$. Concretely, this term depends on x_i, a_i, b_i and θ_i, and is computed based on the Bhattacharyya distance between the inside and outside borders

of the ellipse w_i [10]. The interaction term must be defined in order to avoid the superposition of the ellipses w_i. Concretely, U_p controls the overlapping of ellipses by measuring the intersection between ellipses:

$$U_p(w_i) = \sum_{p_x \in A_i} |\{w_j \in \mathbf{w} \setminus w_i, p_x \in A_j\}| \ ,$$

with A_i the set of all pixels inside the ellipse w_i, and $\mathbf{w} \setminus w_i \equiv \{w_j, \forall j \neq i\}$. This term increases when the percentage of overlapped surface grows.

To compute the optimal realization $\mathbf{w}$, we use a simulated annealing algorithm combined to a "birth and death" process: the birth step consists in generating a large number of ellipses w_i, each one being one realization of a Poisson distribution; the death step examines each ellipse w_i: the ellipse is isolated from $\mathbf{w}$ and we compute the new energy $U(\mathbf{w} \setminus w_i)$. If the energy decreases the ellipse is effectively removed from $\mathbf{w}$, otherwise the ellipse is kept with a probability depending on $U(\mathbf{w})$, on $U(\mathbf{w} \setminus w_i)$ and on a parameter δ. In practice, this usually results in removing each ellipse overlapped by an ellipse with a better (lower) data-fidelity term, as well as in removing non-overlapped ellipses that are incorrectly placed. The birth and death process is iterated until convergence. To expect a fast convergence, the parameter δ should decrease: this parameter can be related to the decreasing "temperature" of the simulated annealing algorithm. The convergence of such algorithm has been proved in [43]. The original sequential birth and death algorithm can now be described as follows.

Initialization: give suitable values for the various parameters. Set $\mathbf{w}$ to an empty set.

1. *Birth:* randomly generate a large number of ellipses and add them to the current realization $\mathbf{w}$.
2. *Data-fidelity term computation:* for each ellipse of the realization, U_d is computed.
3. *Overlap map computation:* in order to compute U_p in the next step, we first build a map giving the number of ellipses overlapping each pixel. For each ellipse, we have to visit the pixels inside this ellipse and update the map.
4. *Death:* sort the ellipses w_i in decreasing order of their data-fidelity term. This sorting step ensures the best ellipses (i.e. the ones having the lowest U_d values) are processed as last, with thus lowest U_p values. Then for each sorted ellipse w_i, compute the overlapping energy U_p. This term is directly given by the sum of elements of the overlap map over the ellipse. Depending on U_d and U_p, the ellipse is then either removed or kept (with a computed probability), and if removed, its contribution is removed from the overlap map.
5. Stop if all the ellipses added in the birth step and only them are removed in the death step; otherwise, decrease δ and go to step 1.

3.1.1.2 Inherent sequentiality of the death step

The three first steps can be easily computed in parallel (with some synchronizations: see Sect. 3.1.2.2). The death step is however inherently sequential: the ellipses have to be treated in the decreasing order of their data-fidelity term. This is important for good quality results because the ellipses with better data-fidelity terms are hence computed last, ensuring they are unlikely to be deleted due to overlapping with ellipses with worse data-fidelity terms.

One could first consider not to parallelize the death step. But since the death step time corresponds to 20% of the overall computation time (in serial on CPU), the resulting speedup would be limited to 5x on any number of cores according to the Amdahl's law.

Since the number of ellipses is quite large (for example, around $20,000$ on a 1024×1024 image or $320,000$ on a 4096×4096 image), one could then try to parallelize the original death step by browsing the sorted list of ellipses with a fixed number of threads N_T. The ordering of the ellipse would therefore be only partially respected. But one could aim to have this way high enough parallel speedups with reasonably good result quality. The results of such a parallelization have been simulated in [AFB13]: unfortunately, the quality of the results quickly drops to unacceptable values as the degree of parallelism increases (above just 4 threads). Moreover, this reduction of quality comes along with an increase of the number of iterations.

3.1.2 Scalable parallel birth and death process

We now show how we have designed a new parallel birth and death (PBD) algorithm.

3.1.2.1 A new birth and death algorithm

A parallel death step. In order to obtain an algorithm which scales with the number of cores, a new way to compute the overlapping energy independently of any ordering is required. In that aim, we propose a new overlapping energy $U_p^*(w_i)$:

$$U_p^*(w_i) = |\{p_x \in A_i, U_d(w_i) > min(p_x)\}| \ ,$$

with $\min(p_x)$ the minimal data-fidelity term of all ellipses overlapping the pixel p_x.

We can now compute this overlapping energy based only on the ellipses with the best (*i.e.* minimal) data-fidelity terms. Each ellipse thus needs to know the exact number of its pixels which are overlapped by an ellipse with a better data-fidelity term. The 2D overlap map now stores for each pixel the minimal data-fidelity term among all ellipses overlapping this pixel. During the new overlap map computation step, for each ellipse we thus have to store its data-fidelity term d in the overlap map, for each of its pixels p, if d is lower than the current value of p.

During the death step, each ellipse can then count the number of pixels having a better data-fidelity term than its own one. It is thus possible to determine which ellipse to keep or to delete in any order. This allows the death step to be computed in parallel. Moreover, when an ellipse is deleted, the overlap map is not modified any more, which reduces the computation load. The overlapping energy is indeed now based on all possible ellipses, including both the already removed ones and the currently kept. This new algorithm ensures that no ellipse will be removed due to an overlapping with a worst ellipse.

We have shown in [AFB13] that the quality of the new algorithm matches indeed the one of the original algorithm in a serial execution, and that this quality is not degraded when increasing the number of threads (see Sect. 3.1.2.2 for the parallelization). Moreover, the number of iterations does not increase with the number of threads.

Stop criterion. The stop criterion used in [10] – waiting for no new ellipse created and no previous ellipse deleted – is too strict: the number of iterations grows proportionally to the image size. As we are looking for a scalable algorithm, which can handle the biggest images, our new birth and death algorithm now stops whenever $\frac{R_{\text{total}}}{R_{\text{new}}+R_{\text{old}}} > \mu$, with R_{total} the total number of ellipses kept in the realization after the death step, R_{new} the number of ellipses added during the last birth step and not deleted, and R_{old} the number of ellipses from the

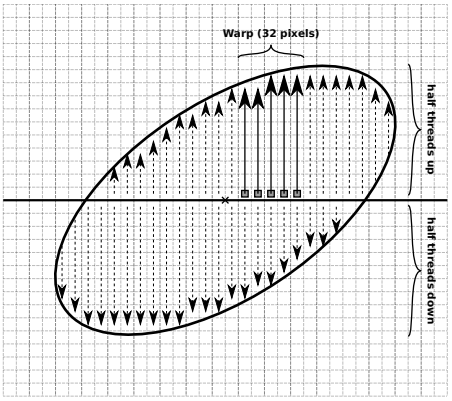


Figure 3.2 – A new scan of the ellipse area on GPU.

past realization deleted during the last death step. This new criterion checks that there is less than one change (a new ellipse added or an old one deleted) every μ ellipses ($\mu = 500$ in practice). As the number of ellipses depends on the image size, this leads to a stop criterion that scales with the image size.

3.1.2.2 Efficient deployment on parallel architectures

In order to show the scalability of our new PBD algorithm, we have deployed it on both multi-core CPUs and on GPU. Regarding multi-core CPUs with shared memory, the deployment has been performed with OpenMP, using a dynamic load balancing for each step, with appropriate computation grain sizes for the threads, and atomic operations and locks for the thread synchronizations. Regarding the GPU deployment in CUDA, we detail in the following how this has been efficiently performed.

Birth step: the GPU deployment of the birth step is straightforward. Each GPU thread handles one single pixel, within a 2D grid of the size of the image, and the blocks of threads are organized so as to ensure coalesced memory loads. CUDA atomic operations are required for the creation of ellipses, as all threads store the created ellipses in the same array. These atomic operations are not expected to be problematic in practice since the number of created ellipses is very small regarding the number of pixels: on a 1024×1024 image, around $20,000$ ellipses are created whereas the number of pixels is 50 times greater.

Data-fidelity term: in a first naive version, directly based on the OpenMP version, each GPU thread handles one single ellipse to compute its data-fidelity term. In order to better match the fine-grained parallelism of GPUs and to reduce the compute divergence among threads within the same warp, we modified this using multiple threads per ellipse. Each thread computes the value of a point from its polar coordinates inside or outside the ellipse. Reductions are then applied to retrieve the variances and the means from the inside and outside borders, needed to compute the Bhattacharyya distance. In that aim, we use an efficient GPU reduction provided in the NVIDIA CUDA GPU computing SDK[1].

Overlap map computation: in a first naive GPU version, directly based on the OpenMP version, each thread handles one single ellipse, in order to compute the minimal data-fidelity

[1]See: https://developer.nvidia.com/cuda-code-samples

term for each of its pixels. We scan the ellipse area line by line, from top left to bottom right, and for each pixel we store in the overlap map the data-fidelity term of the current ellipse if this one is lower than the current value. This however leads to uncoalesced memory accesses, to irregularities in the control flow within each warp (each ellipse having a different size), and to a too large computation grain per thread.

We thus propose here a new area scan of the ellipse more efficient on GPU. As presented in Fig. 3.2, we start the scan from the horizontal line passing by the center of the ellipse. One half of the threads then scans up, while the other half scans down. This ensures much more coalesced memory accesses and threads within a warp now share their cache lines.

In order to compute the minimal value during the scan, we could use an atomic "min" function. Unfortunately, the corresponding CUDA function only deals with integers, whereas our data-fidelity term is a single precision floating-point number. Our solution is to use the atomic "exchange" function: for each pixel p of each ellipse, in case the data-fidelity term d of the current ellipse is smaller than the actual value of p in the overlap map, we exchange this value with d and we check that no other thread has performed an exchange with an even lower data-fidelity term in between. In this case, the exchange process is repeated until the correct minimal value has been written.

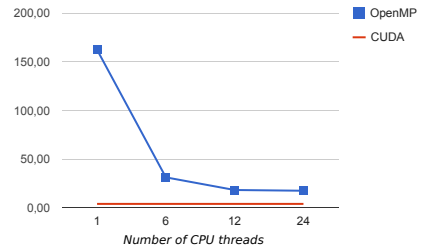
Death step: for this step, the overlapping energy computation relies on a scan of each ellipse, which is performed in the same way as in the previous step. However, we need here to sum the number of pixels which are overlapped by an ellipse with a better data-fidelity term. Therefore, since an ellipse area scan is performed by multiple threads, a reduction is needed to sum up the values of all threads. In that aim, we use the GPU reduction provided in the NVIDIA CUDA GPU computing SDK. Besides, we store the kept ellipses continuously in memory by performing an atomic "increment" on the number of ellipses. These atomic operations are not expected to be problematic in practice as the number of kept ellipses is very small regarding the number of created ellipses: on a 1024×1024 image, around 200 ellipses are kept whereas the total number of ellipses is 100 times greater.

3.1.3 Performance results

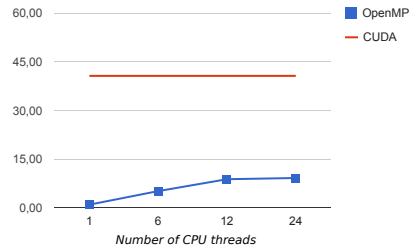
The following tests have been performed with a 4096×4096 image on one compute server composed of one NVIDIA Fermi C2070 GPU and two Intel X5650 hex-core CPUs with 2-way SMT (hence 24 hardware threads in total).

As presented in Sect. 3.1.2.2, the parallelization of the birth step of the PBD process is straightforward (except for the atomic operations). We therefore obtain very good parallel speedups on both CPU (up to 9.21) and GPU (up to 39.46) in Figs. 3.3a and 3.3b. The parallelization of the data-fidelity term computation step is also straightforward and offer very good speedups on CPU (up to 14.47) as presented on Figs. 3.3c and 3.3d. A good speedup of 21.28 is also obtained on GPU when using one GPU thread per ellipse (CUDA naive). Using multiple threads per ellipse enables us to reach an acceleration of 29.58.

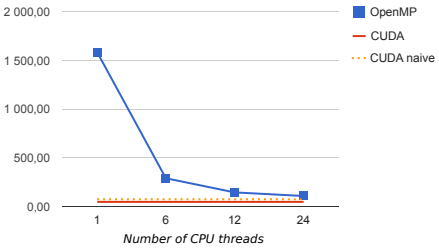
The performance of the overlap map computation step (see Figs. 3.3e and 3.3f) is however constrained by the numerous synchronizations required to update, in each pixel, the minimum data-fidelity value of all the ellipses that overlap on this pixel. In OpenMP, we rely on multiple locks and we obtain a speedup of 5.79 on the two CPUs: this could be improved with new OpenMP 4.0 atomic operations (not available at the time of this work). The CUDA code can already benefit from atomic "exchange" operations for this overlap map computa-



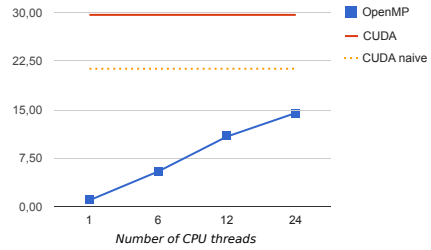
(a) Birth computation times per iteration (ms)



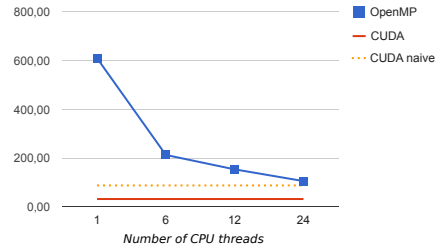
(b) Speedups for the birth step



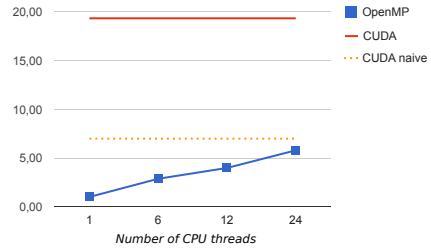
(c) Data-fidelity term computation times per iteration (ms)



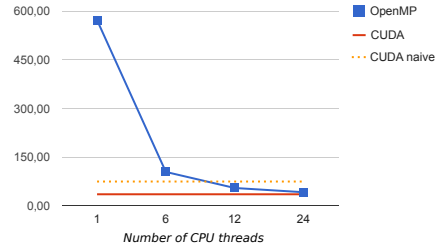
(d) Speedups for the data-fidelity term computation



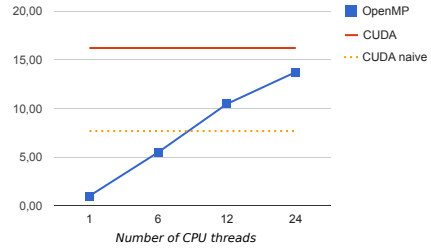
(e) Overlap map computation times per iteration (ms)



(f) Speedups for the overlap map computation

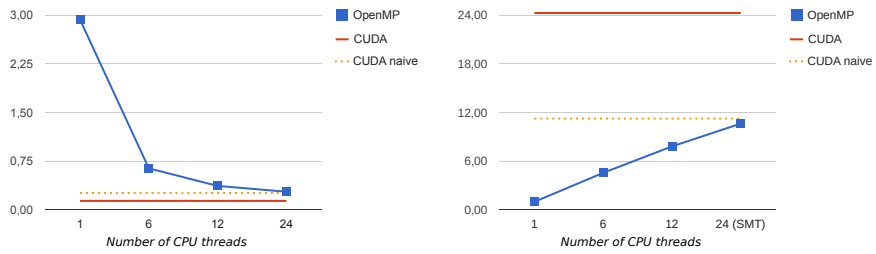


(g) Death computation times per iteration (ms)



(h) Speedups for the death step

Figure 3.3 – Average computation times and corresponding speedups (with respect to the sequential CPU run without OpenMP) of the birth and death steps.



(a) Total computation times per iteration (s)

(b) Total speedups

Figure 3.4 – Total computation times and speedups (with respect to the sequential CPU run without OpenMP) per iteration of the birth and death process.

tion step, but the naive CUDA implementation (with one CUDA thread per ellipse) offers only a speedup of 6.99. Our new scan of the ellipse area presented in Sect. 3.1.2.2 enables here to have much more coalesced memory accesses and to reduce the cache trashing within each warp. Moreover, the reduction over all pixels within each ellipse is also efficiently performed. Thanks to this new ellipse area scan we thus obtain a good speedup of 19.35.

With the new PBD algorithm, the death step is also efficiently performed in parallel on both CPU and GPU as shown in Figs 3.3g and 3.3h. On the GPU, the death step also benefits strongly from our new ellipse area scan, which leads to a twice greater speedup.

When considering the complete execution on Figs 3.4a and 3.4b, we obtain very good overall speedups (up to 11.00) with OpenMP on multi-core CPUs. The naive GPU implementation offers a limited speedup of 11.64, whereas a better match with the fine-grained parallelism of GPUs enables us to obtain a good GPU speedup of 22.94 over a sequential CPU run. We recall that according to Amdahl's law, we were limited to a maximum speedup of 5.0 for the complete original birth and death process with a non-parallelized death step.

3.1.4 Conclusion

We have presented a new scalable parallel birth and death algorithm for cell nuclei extraction in histopathology images. Contrary to the original birth and death algorithm, whose death step was inherently sequential, this new algorithm scales on the number of cores and on the number of ellipses. Thanks to efficient deployements in OpenMP and in CUDA, we manage to obtain good speedups on multi-core CPUs and on GPU. While the power consumption of the two CPUs is roughly the same as the GPU one, the GPU is 2 times faster, thanks to its greater compute power, than the two multi-core CPUs for this application which justifies the GPU code development and optimization. It has to be noticed that an OpenCL implementation could have enabled us to have one single source code for both CPU and GPU, to study which step of the birth and death process can benefit from the implicit vectorization on multi-core CPU (SSE, AVX) and to test the scalability of our algorithm on the many-core architecture of the Intel Xeon Phi processors.

Finally, we emphasize that such birth and death process can accelerate breast cancer grading applications, as well as numerous other applications based on extraction of elliptically-shaped objects.

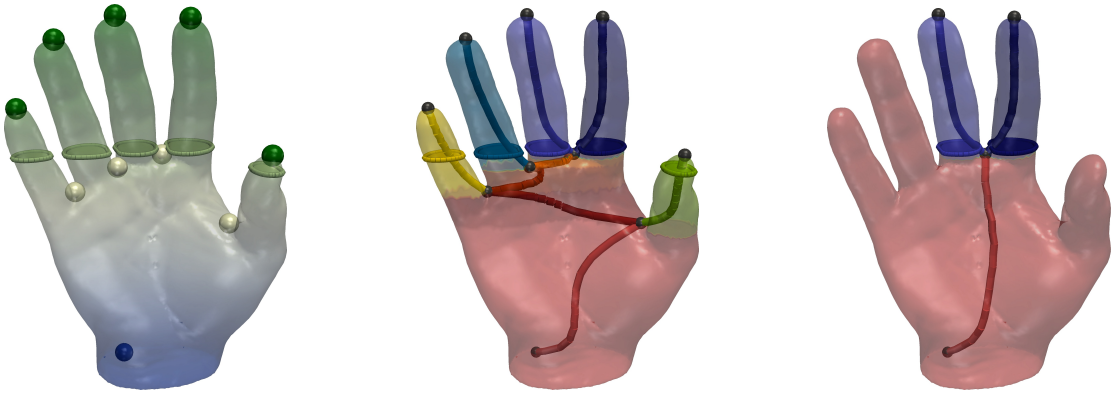
3.2 Task parallelism on multi-core architecture for scientific visualization

3.2.1 Merge and contour trees in scientific visualization

Scientific data sets, resulting from acquisition devices or from numerical simulations, are more and more complex and constantly increasing in size. In order to efficiently visualize and interactively explore such data sets, advanced data analysis algorithms are required. For scalar field visualization, topological data analysis techniques [48, 68, 118] enable to capture the structure of the input data into high-level topological abstractions such as merge trees [20, 132], contour trees [18, 26, 141], Reeb graphs [117, 121] or Morse-Smale complexes [61]. Such topological abstractions are fundamental data structures that enable the development of advanced data analysis, exploration and visualization techniques, including for instance: small seed set extraction for fast isosurface traversal [27, 149], feature tracking [133], data-summarization [116, 156], transfer function design for volume rendering [155] and similarity estimation [71, 145]. Moreover, their ability to capture the features of interest in scalar data in a generic, robust and multi-scale manner has contributed to their use in a variety of applications such as: turbulent combustion [20,62,92], computational fluid dynamics [50, 86], material sciences [63, 64], chemistry [59] or astrophysics [124, 130, 134]. However, as computational resources and acquisition devices improve, the resolution of the geometrical domains on which scalar fields are defined also increases. In order to enable truly interactive exploration sessions, highly efficient parallel algorithms are therefore required for the computation of these topological abstractions. However, most topological analysis algorithms are originally intrinsically sequential as they often require a global view on the data.

We focus here on merge trees and contour trees, which are fundamental topology-based data structures in scalar field visualization, and more precisely on the augmented versions of these trees (where the arcs hold all the corresponding regular vertices of the domain), which enable all visualization applications including e.g. topology-based data segmentation. Considering a visualization workstation composed of multi-core CPUs with shared memory, previous parallelizations [115] [GFJT16] of the reference sequential algorithm [26] lead to extra work in parallel and to possible load imbalance. On the other hand, massively parallel approaches (such as [99]) cannot produce augmented trees and lead to moderate speedups when compared to the reference sequential algorithm.

We present here a new approach (detailed in [GFJT17]) which completely revisits the traditional, sequential merge tree algorithm to re-formulate the computation as a set of local tasks that are as independent as possible. This requires local sorting traversals, new data structures (Fibonacci heaps), a new criterion for the detection of the saddles which generate branching in the output tree, as well as an efficient procedure to process the output arcs in the vicinity of the root of the tree. No extra work is introduced in parallel, and we naturally benefit from the dynamic load balancing of the task runtime. This results in lower computation times, in sequential as well as in parallel on multi-core CPUs with shared memory thanks to the OpenMP task runtime. In the context of augmented contour tree computation, we show that a direct usage of our merge tree computation procedure also results in lower overall computation times, both in sequential and parallel. Performance results on real-life data sets demonstrate that our approach outperforms sequential [44] and parallel [GFJT16] reference implementations, both for augmented merge and contour tree computations.



(a) (b) (c)

Figure 3.5 – Topology driven hierarchical data segmentation. (a) Input scalar field f (color gradient), level-set (light green) and critical points (blue: minimum, white: saddle, green: maximum). (b) Split tree of f and its corresponding segmentation (arcs and their pre-images by ϕ are shown with the same color). (c) Split tree of f and its corresponding segmentation, simplified according to persistence.

Background. Although our algorithm supports arbitrary dimensions, we will consider here a 3D tetrahedral (unstructured) mesh $\mathcal{M}$ where a scalar field f is defined on the vertices of $\mathcal{M}$. This scalar field can for example correspond to the results of a numerical simulation in each mesh vertex. The *star* $St(v)$ of a vertex v is the set of tetrahedrons of $\mathcal{M}$ which contain v as a vertex corner. The *link* $Lk(v)$ is the set of faces of the tetrahedrons of $St(v)$ which do not intersect v. Let $Lk^-(v)$ be the *lower link* of the vertex v: $Lk^-(v) = \{\sigma \in Lk(v) \mid \forall u \in \sigma : f(u) < f(v)\}$. The *upper link* $Lk^+(v)$ is given by $Lk^+(v) = \{\sigma \in Lk(v) \mid \forall u \in \sigma : f(u) > f(v)\}$. Then, given a vertex v, if its lower (respectively upper) link is empty, v is a local *minimum* (respectively *maximum*). If both $Lk^-(v)$ and $Lk^+(v)$ are simply connected, v is a regular point. Any other configuration is called a *saddle* point (white spheres in Fig. 3.5a). *Critical* points denote minima, maxima and saddles.

A level-set is defined as the pre-image of an isovalue $i \in \mathbb{R}$ onto $\mathcal{M}$ through f: $f^{-1}(i) = \{p \in \mathcal{M} \mid f(p) = i\}$ (see Fig. 3.5a). Each connected component of a level-set is called a *contour*. In Fig. 3.5b, each contour of the level-set of Fig. 3.5a is shown with a distinct color. Similarly, the notion of *sub-level set*, noted $f^{-1}_{-\infty}(i)$, is defined as the pre-image of the open interval $(-\infty, i)$ onto $\mathcal{M}$ through f: $f^{-1}_{-\infty}(i) = \{p \in \mathcal{M} \mid f(p) < i\}$. Symmetrically, the *sur-level set* $f^{-1}_{+\infty}(i)$ is defined by $f^{-1}_{+\infty}(i) = \{p \in \mathcal{M} \mid f(p) > i\}$. Let $f^{-1}_{-\infty}\big(f(p)\big)_p$ (respectively $f^{-1}_{+\infty}\big(f(p)\big)_p$) be the connected component of sub-level set (respectively sur-level set) of $f(p)$ which contains the point p. The *split tree* $\mathcal{T}^+(f)$ (see Fig. 3.5b) tracks the merges of the connected components of the sur-level sets, each connected component corresponding to one arc in $\mathcal{T}^+(f)$. The *join tree*, noted $\mathcal{T}^-(f)$, is defined similarly to track the merges of the connected components of the sub-level sets. Irrespective of their orientation, the *join* and *split* trees are usually called *merge trees*, and noted $\mathcal{T}(f)$ in the following. The notion of *Reeb graph* [121] is also defined similarly to track the merges of the connected components of the level sets. The Reeb graphs defined on simply-connected domains being loop-free, such a Reeb graph is called a *contour tree* and we will note it $\mathcal{C}(f)$. Contour trees can be computed efficiently by combining the join and split trees with a linear-time traversal [26, 141]. In Fig. 3.5, since $\mathcal{M}$ is simply connected, the contour tree $\mathcal{C}(f)$ is also the Reeb graph of f. Since f has only one minimum, the split tree $\mathcal{T}^+(f)$ is equivalent to the contour tree $\mathcal{C}(f)$.

Note that f can be decomposed into $f = \psi \circ \phi$ where $\phi : \mathcal{M} \to \mathcal{T}(f)$ maps each point in $\mathcal{M}$ to its equivalence class in $\mathcal{T}(f)$ and where $\psi : \mathcal{T}(f) \to \mathbb{R}$ maps each point in $\mathcal{T}(f)$ to its f value. Since the number of connected components of $f^{-1}_{-\infty}(i)$, $f^{-1}_{+\infty}(i)$ and $f^{-1}(i)$ only changes in the vicinity of a critical point [48, 103], the pre-image by ϕ of any vertex of $\mathcal{T}^-(f)$, $\mathcal{T}^+(f)$ or $\mathcal{C}(f)$ is a critical point of f (blue, white and green spheres in Fig. 3.5a). In particular, the pre-images of vertices of valence 1 necessarily correspond to extrema of f [121]. The pre-images of vertices of higher valence correspond to saddle points which join (respectively split) connected components of sub- (respectively sur-) level sets. Since $f^{-1}_{-\infty}(f(M)) = \mathcal{M}$ for the global maximum M of f, $\phi(M)$ is called the *root* of $\mathcal{T}^-(f)$ and the image by ϕ of any local minimum m is called a *leaf*. Symmetrically, the global minimum of f is the root of $\mathcal{T}^+(f)$ and local maxima of f are its leaves.

Note that the pre-image by ϕ of $\mathcal{T}(f)$ induces a complete partition of $\mathcal{M}$. In particular, the pre-image $\phi^{-1}(\sigma_1)$ of an arc $\sigma_1 \in \mathcal{T}(f)$ is guaranteed by construction to be connected. This latter property is at the basis of the usage of the merge tree in visualization as a data segmentation tool (see Fig. 3.5b) for feature extraction. In practice in the augmented version of the merge trees, ϕ^{-1} is represented explicitly by storing, for each arc $\sigma_1 \in \mathcal{T}(f)$, the list of regular vertices of $\mathcal{M}$ that map to σ_1. Moreover, $\mathcal{T}(f)$ can be progressively simplified by iteratively removing its "shortest" arcs connected to leaves (see Fig. 3.5c): this can help discard noisy simulation or acquisition results.

Related work. In order to compute (augmented and non-augmented) merge trees, the reference and optimal, but sequential, algorithm of Carr et al. [26] relies on a preliminary sort of all vertices of $\mathcal{M}$ according to their scalar value. All vertices are then processed by increasing (respectively decreasing) scalar value in order to construct the join tree $\mathcal{T}^-(f)$ (resp. the split tree $\mathcal{T}^+(f)$) by keeping track of the connectivity evolution of the sub-level sets $L^-(i)$ (resp. sur-level sets $L^+(i)$) with Union-Find data structures [36]. We emphasize here the importance of the vertex ordering: this ensures that when visiting a vertex v in e.g. the join tree computation, all vertices u with $f(u) < f(v)$ (that it to say, all vertices belonging to the sub-level set of $f(v)$) have already been visited. In particular, when visiting a saddle all the connected components of its sub-level set have been processed and can be merged.

Previous shared memory parallel approaches have considered partitioning the mesh in a static decomposition among the threads, by either dividing the geometrical domain [115] or the range of scalar values [GFJT16]. Due to the non-respect of the global ordering in parallel, this leads in both cases to extra work (with respect to the sequential mono-partition computation) at the partition boundaries when joining results from different partitions. This can also lead to load imbalance among the different partitions [GFJT16].

Regarding shared memory parallel approaches for tetrahedral meshes, Maadasamy et al. [99] introduced a multi-threaded variant of the output-sensitive algorithm by Chiang et al. [31], where the arcs of the (non-augmented) merge trees are evaluated by triggering multiple monotone path computations for each saddle point. This massively parallel algorithm results in good scaling performances on tetrahedral meshes, but its sequential version is up to three times slower than the reference implementation (*libtourtre* [44], see Table I in [99]) of the optimal algorithm [26]. This only yields eventually speedups between 1.6x and 2.8x with regard to libtourtre [44] on a 8-core CPU [99]. Besides, their hybrid CPU-GPU version does not offer any performance gain over the multi-core CPU for such unstructured meshes. These moderate speedups over libtourtre can be explained by the lack of efficiency of the sequential algorithm based on monotone paths by Chiang et al. [31] in comparison to that of Carr et al. [26], due to the cost of extracting all critical points and to some unnecessary mono-

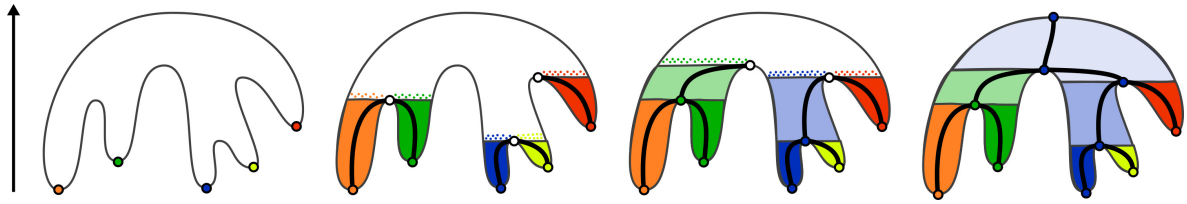


Figure 3.6 – Overview of our augmented merge tree algorithm (2D toy elevation example).

tone path computations. Moreover, this approach cannot produce an augmented merge tree. Carr et al. [28] presented a data parallel algorithm following a similar approach, also for non-augmented trees. Smirnov et al. [132] described a new data structure for computing in parallel the same information as the merge tree. However, the parallel implementation (with atomic variables) requires at least 4 threads to outperform the sequential version (without atomic variables) and offers thus limited speedups (up to 5.8x on 32 CPU cores) compared to this sequential version.

3.2.2 Task-based parallel merge tree computation

We now present our new merge tree computation algorithm, here illustrated with the join tree $\mathcal{T}^-(f)$, which has been designed for an efficient task-based parallelization. Instead of introducing extra work with a static decomposition of the mesh or of the range of scalar values among the threads, our algorithm naturally distributes the computations of the merge tree arcs on the CPU cores via independent tasks. We hence avoid any extra work in parallel, while enabling an efficient dynamic load balancing on the CPU cores thanks to the task runtime. This dynamic load balancing will handle the varying computation cost associated to each arc. Moreover, this algorithm enables us to build the augmented merge tree.

Leaf search. First, we search for all leaves in $\mathcal{T}^-(f)$ (see Fig. 3.6, left): for each vertex $v \in \mathcal{M}$, its lower link $Lk^-(v)$ is constructed via a local operation. If it is empty, v is a leaf of $\mathcal{T}^-(f)$. This embarrassingly parallel step is processed with an OpenMP for loop.

Leaf growth. For each local minimum m, the arc σ_m of the join tree connected to it is constructed (with its segmentation) with a procedure that we call *local leaf growth* (see Fig. 3.6, center left). This is achieved by implementing an ordered breadth-first search traversal of the vertices of $\mathcal{M}$ initiated in m. This traversal is performed with a priority queue $\mathcal{Q}_m$ which ensures that, within the current connected component, the vertices are processed by increasing scalar values. Contrary to the global sorting traversal of the reference sequential algorithm [26], we thus rely here on local sorting traversals. When visiting a saddle, we cannot ensure anymore that all the connected components of its sub-level set have already been processed: a local leaf growth has therefore to stop when reaching a saddle, and the merge of the connected components will be performed only when all leaf growths have reached the saddle. Each leaf growth being independent from the others, spreading locally until it finds a saddle, we naturally implemented it as a task.

Saddle stopping condition In order to stop the leaf growth procedure started at a leaf m when reaching the saddle s corresponding to the other extremity of σ_m (white disks, center left in Fig. 3.2.2), a local saddle detection is required. Instead of the costly critical point detection, which would also introduce unnecessary computations for non-critical points, we rely on the following faster test: when the next vertex returned by the priority queue has

a lower f value than the vertex visited last, this implies that the last visited vertex was a saddle [GFJT17], which closes the arc σ_m.

Union-Find data structures [36] (one Union-Find node per leaf) enable us to check whether a vertex has already been visited by the current growth. Our Union-Find implementation supports concurrent *find* operations from parallel arc growths (executed simultaneously by distinct tasks). A *find* operation on a Union-Find currently involved in a *union* operation is also possible but infrequent, and safely handled in parallel by atomic operations.

We will also have to detect the last task reaching a saddle. In this purpose, each task detecting a saddle s atomically decrements an integer counter, initialized to the size of $Lk^-(s)$ during the leaf search step, by the number of vertices below s coming from the current growth. The task setting this counter to zero is the last reaching this saddle.

Saddle growth The last leaf growth reaching a saddle s will be in charge of the growth for the arc of $\mathcal{T}^-(f)$ initiated in s, noted σ_s (see Fig. 3.6, center right). However, in order to represent all the connected components of sub-level set merging in s, this last task will have first to perform the union of the priority queues $\mathcal{Q}_{m_0}, \mathcal{Q}_{m_1}, \dots \mathcal{Q}_{m_n}$ of all the arcs merging in s. If done naively, this operation could yield a quadratic runtime complexity for our approach overall. We thus model each priority queue with a Fibonacci heap [36, 53], which supports the merge of two queues in constant time. Similarly to the traditional merge tree algorithm [26, 141], we also rely on the Union-Find data structure to precisely keep track of the arcs which also need to be merged (by the last task) at a given saddle s.

We emphasize that the time complexity of our algorithm is therefore exactly equivalent to the linearithmic one of the reference sequential algorithm [26, 141].

Trunk growth. We have also managed to improve time performance by abbreviating the process when only one arc growth is remaining. This last arc growth, starting at a saddle s, will visit all the remaining, unvisited, vertices of $\mathcal{M}$ upwards until the global maximum of f is reached, possibly reaching on the way an arbitrary number of *pending* join saddles, where other arc growths have been stopped and marked terminated (white disks in Fig. 3.6, center right). As illustrated in Fig. 3.6 (right), this will constitute a monotone path from s up to the root of $\mathcal{T}^-(f)$. We call this sequence the *trunk* of $\mathcal{T}^-(f)$.

The trunk of the join tree can be computed faster than through the breadth-first search traversal. Let s be the join saddle where the trunk starts. Let $S = \{s_0, s_1, \dots s_n\}$ be the sorted set of pending join saddles (which still have unvisited vertices in their lower link). The trunk is constructed by simply creating arcs that connect two consecutive entries in S. Next, these arcs are augmented by simply traversing the vertices of $\mathcal{M}$ with higher scalar value than $f(s)$ and projecting each unvisited vertex v to the trunk arc that spans it scalar value $f(v)$. Using OpenMP, we parallelize the loop of this vertex projection procedure using chunks of contiguous vertex indices (chunks are dynamically distributed to the threads). For each chunk, the first vertex is projected on the corresponding arc of the trunk using dichotomy. Then, each new vertex processed next relies on its predecessor for its own projection.

As the number of tasks can only decrease, the detection of the trunk start is straightforward. Each time a task terminates at a saddle, it decrements atomically an integer counter, which tracks the number of remaining tasks. The trunk starts when this number reaches one.

3.2.3 Contour tree computation

Once the join and split trees have been computed, we can build the contour tree. In order to obtain an augmented contour tree, each arc of $\mathcal{T}(f)$ needs to be equipped with the explicit

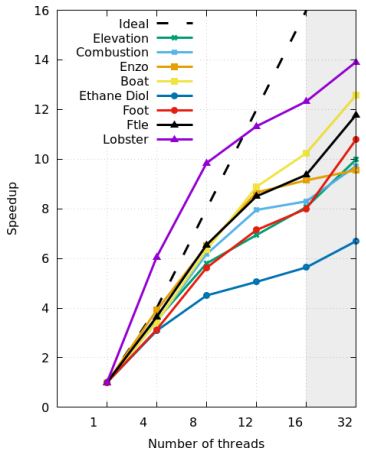


Figure 3.7 – Merge tree scalability results. The gray area denotes using 2 threads per core.

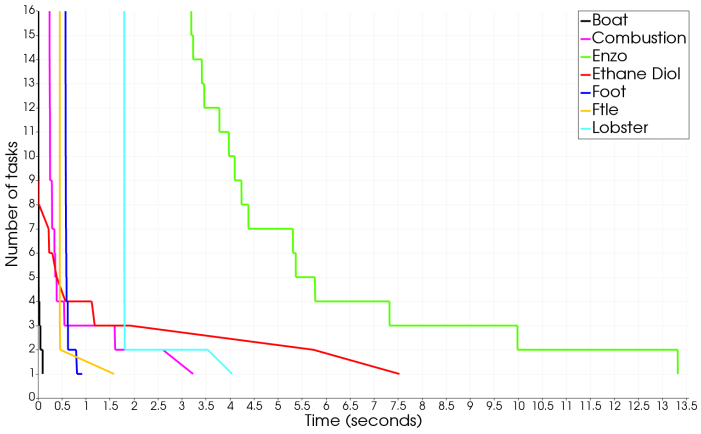


Figure 3.8 – Number of remaining tasks throughout time. The chart is cropped at 16 to highlight the "suboptimal" sections.

sorted list of vertices which project to it. In a *post-processing* step, we reconstruct these explicit sorted lists in parallel thanks to local orderings during the arc growth step, and thanks to a specific parallel procedure for the trunk step (see [GFJT17] for details).

Next, we combine the join and split trees into the output contour tree by adding arcs from both trees leaf after leaf, according to the reference algorithm [26]. Each time we add an arc of one of the two trees, we have to remove the list of regular vertices of this arc from the other tree. As this algorithm is not straightforward to parallelize, and as this *combination* step represents a minor part in the overall sequential computation (about 2% of the total time), we execute it sequentially in this implementation.

3.2.4 Performance results

In this section we present performance results obtained on a workstation with two Intel Xeon E5-2630 v3 CPUs (8 CPU cores and 16 hardware threads each). By default, parallel executions will thus rely on 32 threads. These results were performed with our VTK/OpenMP based C++ implementation using `g++` version 5.4.0 and OpenMP 4.0. This implementation (called *Fibonacci Task-based Merge tree*, or FTM) was built as a TTK [146] module. For the Fibonacci heap, we used the implementation available in Boost.

Our tests have been performed using eight data sets from various domains. The first one, Elevation, is a synthetic data set with only one arc in the output tree. Five data sets (Ethane Diol, Boat, Combustion, Enzo and Ftle) result from simulations and two (Foot and Lobster) from acquisition, containing large sections of noise. For the sake of comparison, these data sets have been re-sampled on the same regular grid of size 512^3.

Merge tree performance results. Speedups for the join tree are presented in Fig. 3.7. The split tree leads to similar speedups [GFJT17], and overall our FTM implementation achieves an average speedup of 10.4 on 16 cores (65% of parallel efficiency). The monotonous growth of all curves implies that more threads always leads to faster computations, which enables us to focus on the 32-thread executions.

One can notice that the Lobster data set presents speedups greater than the ideal one for 4 and 8 threads. This unexpected but welcome supra-linearity is due to the trunk processing of

Table 3.1 – *Sequential* join tree computation times (in seconds) and ratios between libtourtre (LT), Contour Forest (CF) and our Fibonacci Task-based Merge tree (FTM), on a 256^3 grid (bold: FTM speedups).

Dataset	LT	CF	FTM	LT / FTM	CF / FTM
Elevation	5.81	7.70	3.57	**1.63**	**2.15**
Ethane Diol	11.59	17.75	7.14	**1.62**	**2.48**
Boat	11.84	17.11	6.93	**1.70**	**2.46**
Combustion	11.65	16.87	8.06	**1.44**	**2.09**
Enzo	14.33	17.99	17.94	0.79	**1.00**
Ftle	11.32	15.62	7.15	**1.58**	**2.18**
Foot	9.45	12.72	5.94	**1.59**	**2.14**
Lobster	11.65	14.80	13.99	0.83	**1.05**

Table 3.2 – *Parallel* join tree computation times (in seconds) and ratios between libtourtre (LT), Contour Forest (CF) and our Fibonacci Task-based Merge tree (FTM), on a 256^3 grid.

Dataset	LT	CF	FTM	LT / FTM	CF / FTM
Elevation	5.00	2.33	0.43	11.63	5.42
Ethane Diol	8.95	4.54	1.33	6.73	3.41
Boat	8.24	4.40	0.69	11.94	6.38
Combustion	7.96	5.82	0.94	8.47	6.19
Enzo	12.18	8.92	1.98	6.15	4.51
Ftle	8.19	4.98	1.04	7.88	4.79
Foot	7.60	6.94	1.27	5.98	5.46
Lobster	8.40	9.02	2.40	3.50	3.76

our algorithm. The trunk step is indeed able to process vertices 30x faster than the arc growth step, since no breadth-first search traversal is performed in the trunk step (see Sect. 3.2.2). However, for a given data set, the size of the trunk highly depends on the order in which leaves have been processed. Since the trunk is detected when only one growth remains active, distinct orders in leaf processing will yield distinct trunks of different sizes, for a given data set. Hence maximizing the size of this trunk minimizes the required amount of computation, especially for data sets like Lobster where the trunk encompasses a large part of the domain. Note however, that the leaf ordering which would maximize the size of the trunk cannot be known in advance. In a sequential execution, it is unlikely that the runtime will schedule the tasks on the single thread so that the last task will be the one that corresponds to the greatest possible trunk. Instead, the runtime will likely process each available arc one at a time, leading to a trunk detection at the vicinity of the root. On the contrary, in parallel, it is more likely that the runtime environment will run out of leaves sooner, hence yielding a larger trunk than in sequential, hence leading to increased (possibly supra-linear) speedups.

We now compare our approach to two reference implementations, which are, to the best of our knowledge, the only two public implementations supporting augmented trees: (i) *libtourtre* (LT) [44], an open source sequential reference implementation of the traditional algorithm [26]; and (ii) the open source implementation [146] of the parallel Contour Forest (CF) algorithm [GFJT16]. Due to the important memory consumption of CF, these comparisons are performed on a 256^3 grid. As shown in Table 3.1, our sequential implementation is about twice faster than CF and more than one and half time faster than LT for most data sets. This is due to the faster processing speed of our trunk step. The parallel results for the merge tree implementation are presented in Table 3.2, where only the initial sort is parallelized in LT. Regarding CF we report the best time obtained on the workstation, which is not necessarily with 32 threads. Indeed, as detailed in [GFJT16] increasing the number of threads in CF can result in extra work due to additional redundant computations, especially on noisy data sets. On the contrary, FTM always benefits from the maximum number of hardware threads. In the end, FTM largely outperforms the two other implementations for all data sets: LT by a factor 7.8 (in average) and CF by a factor 5.0 (in average).

Contour tree performance results. As detailed in [GFJT17], when using our parallel merge tree implementation to build the contour tree, we obtain an average speedup of 7.9x on 16 cores (49% of average parallel efficiency). In sequential, our implementation is also in average 1.5x faster than LT and 1.3x faster than CT. In parallel, our implementation outper-

forms in average a naive parallel version of LT[2] by a factor 4.7x and CF by a factor 2.7x.

Limitations and following work. The performance of our approach is currently limited by two weaknesses. First, while the combination represents a minor part in the overall computation in sequential (about 2% of the total time), its current sequential implemantation can in fact limit the overall parallel speedups on such number of cores according to Amdahl's law. In [GFJTxx], we have recently proposed a parallel version of the combination which offers performance gains in parallel thanks to the introduction of a trunk step (similar to the trunk step of the merge trees, and offering a higher degree of parallelism).

Second, the number of tasks gradually decreases in our arc growth step. As presented in Fig. 3.8, and depending on the data set and on the number of cores, a significant share of this step can be performed with a number of remaining tasks lower than than the number of available threads, hence not fully exploiting our parallel CPU compute power and limiting our parallel speedups. In [GFJTxx], we have therefore fully "taskified" our merge tree computation by introducing tasks in all steps other than the already task-based arc growth step: namely the leaf search, the trunk step, the post-processing step and the combination step. This enables then us to perform the two merge tree computations in parallel and to overlapp the task processing of the two trees. The arc growth step of a given tree can then be processed concurrently with the arc growth step (or with the trunk or post-processing step) of the other tree, hence providing more tasks to the runtime. This results in speedups up to 1.34x (1.24x in average) for the computation of the merge trees.

3.2.5 Conclusion

We have presented here a new algorithm to compute the augmented merge trees on shared memory multi-core architectures [GFJT17]. This new approach completely revisits the traditional algorithm to compute the merge tree using independent local growths which can be expressed using tasks. This has required local sorting traversals, the use of Fibonnacci heaps and a new criterion for the saddle detection. This has also made it possible to accelerate the processing when there is only one task left. This implementation is the fastest to compute the merge trees in augmented mode in sequential, as well as in parallel where no extra work is introduced in our approach and where we can benefit from the dynamic load balancing of the task runtime. Moreover, we have recently improved our approach regarding the contour tree computation [GFJTxx], partly thanks to a complete "taskification" of the algorithm which enables us to take advantage of the overlap of the two merge tree computations.

We are currently extending this work to the Reeb graph construction, which generalizes the contour tree on non simply-connected meshes. Again, we target the deployment of the best sequential algorithm on a multi-core workstation, and we will have to adapt our task-based approach to the extra processing required to handle the possible loops in the domain (e.g. in a torus) which generate cycles in the Reeb graph. Besides, efficiently deploying our approach on a larger number of cores (such as the Intel Xeon Phi processor) would be challenging, and may require further algorithmic improvements (especially regarding the arc growth step). Another interesting research direction would finally consists in studying the relevance of our approach for *in-situ* visualization, where the analysis code is executed in parallel and in synergy with the simulation code generating the data.

[2]Using a parallel sort and processing the two merge trees in parallel.

3.3 Dual tree traversal on integrated GPUs for astrophysical N-body simulations

3.3.1 N-body algorithms

The N-body problem describes the computation of all pairwise interactions among N bodies (or particles). Once computed, the corresponding forces are used to update the body positions and velocities for the next time-step. In astrophysics, such N-body simulations are essential and widely used for galactic dynamics studies. The gravitational force computation is the most time-consuming part and limits in practice the number of bodies, which is currently much smaller than the number of stars in a real galaxy.

The direct computation of all pairwise interactions among N bodies leads to a prohibitive $\mathcal{O}(N^2)$ runtime complexity. Thanks to the mutuality of gravity (Newton's third law), which states that the force of a particle A on a particle B is the opposite of the force of B on A, one can halve the computation cost, but this latter remains too expensive for millions of particles. Hierarchical methods [12, 30] have therefore been introduced to reduce this runtime complexity, thanks to an octree data structure. This octree is built by inserting particles one by one and by subdividing octree leafs containing more than a given maximum number of particles, denoted by *Ncrit*. Thanks to this octree, the force field is decomposed in a near-field part, directly computed, and a far-field part approximated with various expansions.

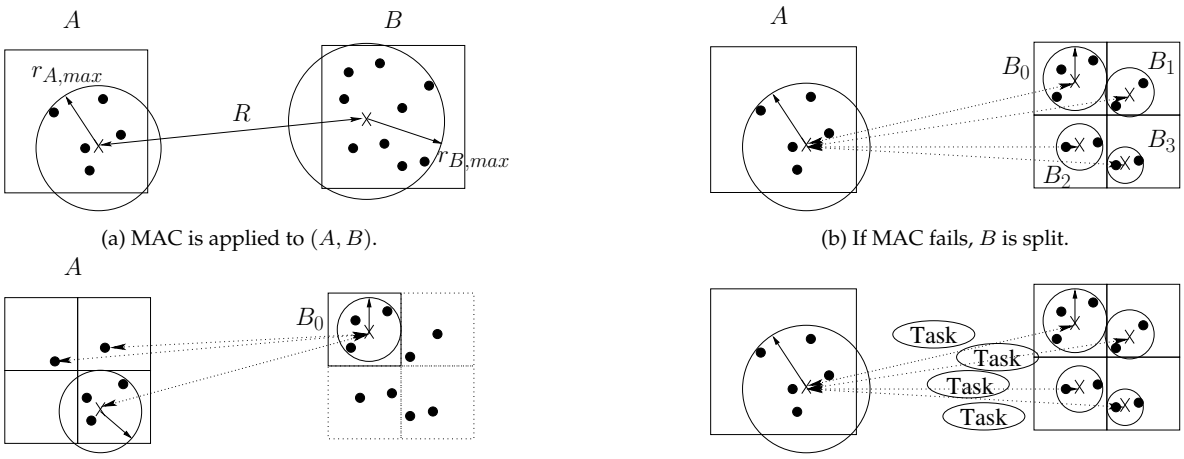
Barnes-Hut tree-codes. The Barnes-Hut tree-code algorithm [12] computes the gravitational forces among N particles with a $\mathcal{O}(N \log N)$ runtime complexity thanks to monopole (and possibly quadrupole) moments. For each target body, the octree is here recursively traversed and "body-cell" or "body-body" interactions are evaluated depending on the multipole acceptance criterion (MAC): $\frac{D}{r} < \theta$, where D denotes the octree cell side length, r is the distance from the target body to the cell center of mass, and θ is an input parameter that balances accuracy and computation cost. Such expansions are well-suited for the relatively low accuracies required in astrophysical N-body simulations, where a relative force error of few 10^{-3} is usually adequate. The loop on the target bodies is parallel which enables CPU parallel implementations with multi-threading and/or with MPI [135]. This inherent parallelism has also been efficiently exploited to develop GPU implementations that run entirely on the GPU [13, 14, 25] and outperform multi-core CPUs. The `Bonsai` code[3], is currently one of the fastest GPU tree-codes.

Fast multipole methods for astrophysics. Dehnen's algorithm [39] can be considered as a fast multipole method (FMM) [30] specific to the relatively low accuracies required in astrophysics. This $\mathcal{O}(N)$ algorithm indeed relies on "cell-cell" interactions and requires specific, low accuracy local expansions based on cartesian Taylor expansions, as well as a specific MAC that can balance (along with the expansion order, which is fixed to 3) the accuracy and the computation cost. This MAC is defined for two cells (A, B) (see Fig. 3.9a) as:

$$\frac{r_{A,max} + r_{B,max}}{R} < \theta \,,$$

where $r_{C,max}$ denotes an upper limit for the distance of any body within the node C from its center of mass [39]. Once the octree has been built using at most *Ncrit* particles per octree leaf, the multipole expansions are calculated during an upward pass in the octree. The interactions are then computed in the following two steps.

[3] See: https://github.com/treecode/Bonsai



(a) MAC is applied to (A, B).

(b) If MAC fails, B is split.

(c) If MAC(A, B_0) fails, A is split.

(d) Tasks spawn when MAC fails.

Figure 3.9 – Dual tree traversal in Dehnen's algorithm.

The first step (*interaction phase*) relies on the *dual tree traversal* (DTT) presented in Fig. 3.9. If the MAC succeeds between two cells (A, B), their interactions can be approximated: both local expansions of A and B are updated based on the multipole expansions of A and B (*M2L - multipole-to-local* - operation) as shown in Fig. 3.9a. More precisely, once the contribution of the multipole expansion of B on the local expansion of A has been computed, we can at low cost deduce the opposite contribution (of the multipole expansion of A on the local expansion of B) using the mutuality of *M2L* interactions. The DTT algorithm enables indeed to consider both operations at the same time. If the MAC fails, the larger cell (B here) is split and the MAC is applied between A and all the children of B (see Fig. 3.9b, with 8 children in 3D). This is applied recursively, and A can then be split when the MAC fails with A as the larger cell (see Fig. 3.9c). This thus leads to a dual recursive traversal of the octree. When the MAC fails for two octree leafs, or when the number of particles is too low (depending on empirical thresholds [39]), the direct computation (*P2P - particle-to-particle* - operation) is used instead of the expansions. Thanks to this DTT, Dehnen's algorithm consistently uses the mutuality of the interactions to (approximately) halve the computation cost in the near-field part as well as in the far-field part. This DTT also enables to better preserve the total momentum than tree-codes [39]. Other work regarding DTT in fast multipole methods can be found in [143, 154, 159], as well as in [34, 97] for molecular dynamics.

After the interaction phase, the *evaluation phase* is used to evaluate the local expansion of each cell for each body within this cell, thanks to a (simply) recursive downward pass of the octree. These two steps correspond together to the most time consuming part.

DTT implementations. Dehnen's algorithm has been implemented in the *falcON* [4] code. As first shown in [39], and as we have then detailed this in [FAL11], this code offers $\mathcal{O}(N)$ computation times one order of magnitude smaller than serial executions of Barnes-Hut tree-codes. Moreover, these computation times are much less sensitive to the distribution of particles: this is very important for astrophysical simulations where the particle distributions representing galaxies or groups of galaxies are highly non-uniform.

In [LF14], we have parallelized *falcON* in the *pfalcON* code[5] on multi-core CPUs and on

[4] *Force ALgorithm with Complexity* $\mathcal{O}(N)$, available in http://carma.astro.umd.edu/nemo/

[5] Available at: https://pfalcon.lip6.fr

Intel Xeon Phi thanks to task-based parallelism (with OpenMP and Intel TBB). Each time an interaction fails the MAC, we create one task for each of the (up to) eight interactions involving the children of the larger cell: see Fig. 3.9d. The mutuality of the interactions is here fully preserved, but introduces write conflicts among tasks that are updating the local expansion of the same cell via different *M2L* operations. These conflicts are handled via atomic operations emulating fast locks. A different, but also task-based, DTT parallelization has been presented in [142] and implemented in the `exaFMM` code (see: https://github.com/exafmm/exafmm) thanks to a rewriting of the DTT (see also [40]). Both *pfalcON* and `exaFMM` offer very good scaling on multi-core CPUs (up to 15.8x on 16 cores for *pfalcON*), as well as on many-core Xeon Phi processors (up to 60x on a 5110P Xeon Phi for *pfalcON*) [LF14] [5]. Since the *falcON* code is dedicated to astrophysical simulations, *pfalcON* is slightly faster than `exaFMM` for such simulations.

Moreover, in [LF14] the near-field part has been vectorized using the SPMD-on-SIMD model of `ispc` (see Sect. 1.2): `ispc` has been here preferred to OpenCL on CPU because of its faster kernel launches (fine SIMD computation grains in *pfalcON*) and because the same data structures can be shared between the *pfalcON* C/C++ code and the `ispc` code. Moreover with `ispc` we can use SIMD vectors twice larger than the hardware vector width, and we explicitely control the loop over all computations (contrary to OpenCL): this gives us more control for efficient and safe direct computations with the mutuality of gravity. Thanks to a hybrid strategy that efficiently combines scalar and vector code, we have thereby managed to have one single portable source code for SSE, AVX and Xeon Phi vector instructions, which offers in the end similar or better performance than the hand-tuned kernels of `exaFMM`.

As shown in [LF14], for astrophysical simulations *pfalcON* on multi-core CPUs is in the end only slightly slower than `Bonsai` on a high-end GPU, and *pfalcON* can run larger simulations (e.g. with 50M particles). One then naturally aims at combining the best algorithm (FMM with DTT) with the most powerful hardware currently available (GPUs). While the FMM has already been deployed on GPUs in numerous works [32,60,66,72,93,113,120,161], none of these applies to the DTT-based FMM. Indeed due to its double recursion, obtaining an efficient DTT on many-core architectures like GPUs is difficult. In [161] the FMM deployment on GPU is performed by concatenating all source particles and expansions in a large buffer that is then transfered over the PCI bus to the discrete GPU. As detailed in [FTxx], the lower compute intensity used in astrophysics leads to a too large share of PCI transfers and data copies in the overall computation time, and to possible memory problems on the limited GPU memory when N increases. In order to overcome these issues, we will thus aim at minimizing the data volume exchanged by the CPU and the GPU, while relying on integrated GPUs (see Sect. 1.1) to reduce the data exchange cost.

We will thus present in the following the first CPU-GPU heterogeneous deployment of a fast multipole method based on dual tree traversal, using integrated GPUs (iGPUs). This is also, to our knowledge, the first FMM deployment on integrated GPUs. Such deployment relies on a new hybrid CPU-GPU algorithm, where the DTT is performed on the CPU cores, and all the computations are performed on the iGPU cores. This has been implemented in OpenCL in the *pfalcON* code as *pfalcON-iGPU*. More details can be found in [FTxx].

3.3.2 The far-field part

The far-field part translates to numerous independent *M2L* computations on the iGPU. However, one has to ensure that two *M2L* computations will not concurrently update the same

local expansion due to the use of the mutuality of the *M2L* interactions. We have considered several strategies to synchronize the *M2L* computations on iGPU and avoid such conflicts: their performance results are compared in Fig. 3.10 on a reference (non-uniform) astrophysical model, the Plummer distribution, with 10M particles.

Preserving mutuality. The three first strategies preserve the mutuality of *M2L* interactions, hence save *M2L* computation cost. The first one, referred to as *atomic float*, relies on emulating atomic additions on floating-point variables with a loop over the OpenCL 1.2 "compare-exchange" atomic operation. Each *M2L* computation is then assigned to one work-item, and each local expansion coefficient is updated (in both cells) thanks to these atomic additions on floating-point variables. Contrary to other works such as [66,161] where each *M2L* operation was performed by a work-group, we use here only one work-item since the expansion orders are lower due to the low accuracies required in astrophysics. This results in massive, regular and rather fine-grained parallelism which is well suited for GPU processing, contrary to many GPU-based FMMs [60,72,93,113,120] where the far-field part is not efficiently processed on GPU, and hence often computed on CPU. In order to expose to the GPU cores all the *M2L* interactions that have to be performed, we simply use here a large zero-copy buffer (the *interaction buffer*) where all pairs of cell indices involved in a *M2L* operation are consecutively written by the CPU thread during the DTT. As shown in Fig. 3.10, the numerous atomic operation of this strategy are however too expensive.

The second strategy (*atomic bit*) aims at reducing the number of atomic operations performed and is close to our multi-core CPU parallelization of the DTT [LF14]. We use here atomic operations on a specific bit within the cell data structure to emulate locks and ensure exclusive access to the cell local expansion. But contrary to multi-core CPUs where the *atomic bit* synchronizations were fast, infrequent and interleaved with other computations, this strategy implies a too strong overhead on GPU. Using stridden memory accesses in the interaction buffer can reduce the contention on these bit locks (*stridden atomic bit* strategy).

Forsaking mutuality. Forsaking the mutuality of the interactions prevents us from saving *M2L* computation costs, but enables us to avoid any synchronization costs and leads in the end to better results. We have first considered here a *sort no-mutual* strategy where a sort on all couples of cell indices enables us to gather continuously in the interaction buffer the *M2L* operations depending on their target cell. We then launch the *M2L* kernel with one work-item per target cell. In order to avoid this sorting (whose cost is not shown in Fig. 3.10), we have proposed a last strategy, referred to as *no-mutual*. We use here a specific data structure (detailed in [FTxx]) to store, for each target cell C, the (linked) list of source cell indices involved in a *M2L* interaction with C. On the GPU, the work-item in charge of the target cell C has then to browse the corresponding linked list to retrieve all the source cell indices involved in a *M2L* interaction with C and to perform all these computations.

As the number of *M2L* interactions varies from a target cell to another, this introduces compute divergence among the work-items. We have thus finally proposed the *sort no-mutual WG* and *no-mutual WG* variants where one work-group is used per target cell C: each work-item within this work-group will process one *M2L* computation with C as target cell. One has of course to perform a reduction on the local expansion terms among all work-items of each work-group. Here distinct implementations are required for the APU and the Intel iGPU due to architectural differences regarding the OpenCL local memory banks and the work-group sizes [FTxx]. As shown in Fig. 3.10, the reduction overhead is largely offset by the performance gain due to the decrease in compute divergence, which justifies the final choice of *no-mutual WG* (on both the APU and the Intel iGPU).

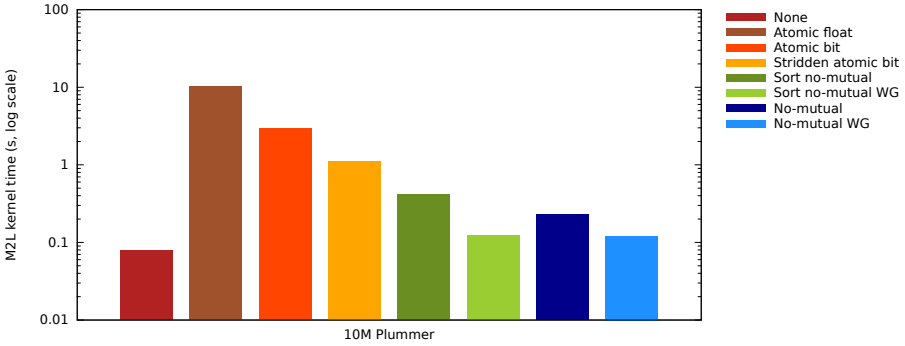


Figure 3.10 – Performance comparison of the *M2L* synchronizations on the APU.

3.3.3 The near-field part

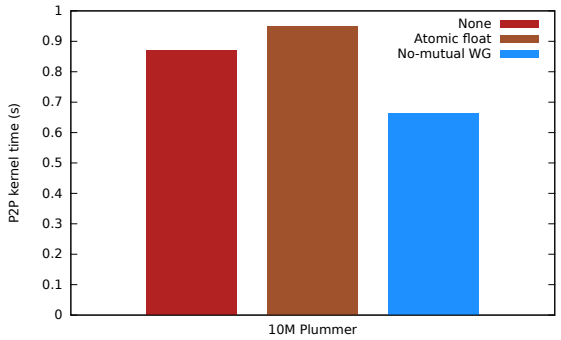
The direct computation of the near-field part is a classical HPC kernel, and its efficient GPU deployment has been extensively studied (e.g. [111]). Numerous implementations are publicly available: in the NVIDIA CUDA and OpenCL code samples, in the AMD OpenCL SDK 2.8 and in other OpenCL tutorials[6]. They use local memory to reduce global memory accesses to the source bodies, as well as loop unrolling. All these HPC implementations do not exploit the mutuality of gravity: it is indeed more efficient to perform twice the direct computations than to introduce divergence with the mutuality of gravity. The compute core of our *P2P* kernel relies on such implementations, however these are designed for one (very) large *P2P* operation involving thousands of bodies. In our case, we rather have to deal with numerous independant *P2P* operations involving few bodies (up to 64 in practice). This relates to GPU deployments of adaptive FMMs (without DTT) such as [66, 93, 113, 120, 161].

We have chosen to consider one work-group per *P2P* operation, even with our low number of particles per cell. Using one work-item per *P2P* operation as in the *M2L* strategies, would have indeed introduced a too coarse computation grain per work-item and compute divergence among the work-items, while preventing the use of local memory.

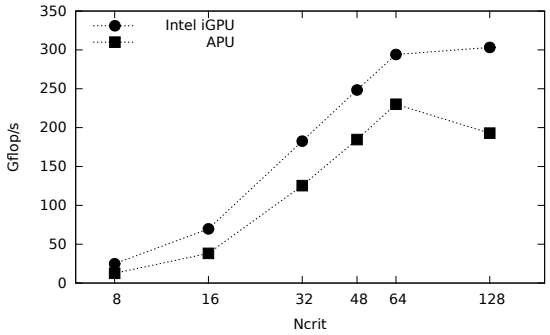
***P2P* synchronization strategies.** Following the results obtained for the far-field part, we have considered the *atomic float* and *no-mutual WG* strategies for the near-field part, which correspond to the homonymous *M2L* strategies. The *P2P no-mutual WG* strategy can here be related to the sparse U-List of the kernel-independent adaptive FMM (without DTT) mentioned in [93]. Figure 3.11a shows the relevance of the *no-mutual WG* strategy.

***P2P* specific optimizations.** For best performance, we have performed several low-level optimizations (detailed in [FTxx]) regarding the implementation of our *P2P* kernel in order to efficiently support various work-group sizes and various *Ncrit* values. Since moreover most cells do not have *Ncrit* (but less) particles, one of our main challenges was to minimize the number of idle work-items. We have first generalized the technique presented in [111] by dynamically choosing to have multiple work-items contributing to the computation of a given target body (*multi-work-item*). Then, since the number of bodies in the source cell can be low and in order to best benefit from the loop unrolling, we have prefered to fill the local memory buffer with bodies from different cells. While this is naturally obtained in [161] since the source bodies of all source cells are stored contiguously in memory, in our case it

[6] See for example: http://www.browndeertechnology.com/docs/BDT_OpenCL_Tutorial_NBody-rev3.html, or: https://developer.apple.com/library/content/samplecode/OpenCL_NBody_Simulation/Introduction/Intro.html



(a) Performance comparison of the *P2P* synchronizations on the APU.



(b) Performance results of the *P2P* kernel depending on *Ncrit* for a 10M Plummer distribution.

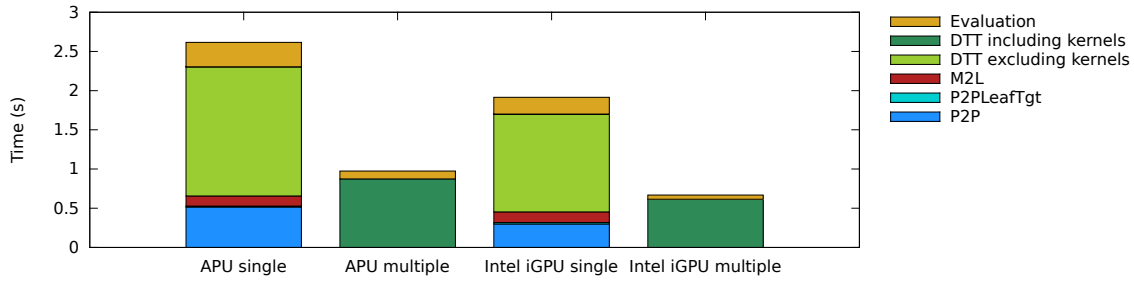


Figure 3.12 – Performance gain due to the DTT overlap with GPU computations on a 10M Plummer distribution. `single` denotes executions with one single CPU thread and one single kernel launch for *P2P*, *P2PLeafTgt* and *M2L*. `multiple` denotes executions with the best number of CPU threads and the best interaction buffer size (resulting in multiple *P2P*, *P2PLeafTgt* and *M2L* kernel launchs).

is up to each work-group to dynamically (and efficiently) concatenate in local memory the source bodies from multiple source cells scattered in global memory. If relevant, the target bodies are also stored in local memory in order to avoid multiple global memory accesses. Besides, due to the *falcON* implementation, we had to use a separate kernel launch (referred to as *P2PLeafTgt*, but still using the *P2P* kernel) for cells with one single particle (`leaf` in *falcON*).

As shown in Fig. 3.11b, the Intel iGPU reaches 303.1 Gflop/s for our numerous small *P2P* operations with very different body numbers, which represents 34% of the single precision peak performance of this iGPU. This is largely satisfactory when comparing to the CUDA N-body SDK which reaches 40% of a NVIDIA K40c GPU peak performance for one single very large (and regular) *P2P* computation. The Intel GPU outperforms here the APU, which can be explained by the different SIMD widths. The 64 wide APU wave-front is a too high value for our simulations: even with *Ncrit* = 64 or *Ncrit* = 128, most cells have a particle number around 20 or 25. A non-negligible share of the work-items are then idle, even with our *multi-work-item* optimization. On the contrary, the lower and flexible SIMD width of the Intel iGPU (see Sect. 1.1 and [78]) enables to better adapt to such cells

Name	Detailed name	Compute features	Launch date	TDP	Current price
APU	AMD A10-7850K Radeon R7 APU	2 2-way SMT CPU cores + 512 iGPU PE	Q1'14	95W	$150
Intel iGPU	Intel Xeon E3-1285L v4 - Iris Pro Gr. P6300	4 2-way SMT CPU cores + 384 iGPU PE	Q2'15	65W	$445
[2x]8C CPU	[2x] Intel Xeon E5-2630 v3	[2x] 8 2-wax SMT CPU cores - AVX2	Q3'14	[2x]85W	[2x]$667
18C CPU	Intel Xeon E5-2695 v4	18 2-way SMT CPU cores - AVX2	Q1'16	120W	$2424
K40c	NVIDIA K40c GPU	2880 GPU PE	Q4'13	235W	$2400

Table 3.3 – Architectures considered. PE stands for (OpenCL) Processing Element.

3.3.4 Overlapping the parallel CPU traversal with GPU computations

In order to efficiently overlap the DTT on CPU with GPU computations, we rely on our task-based parallel DTT with OpenMP [LF14]. Since no computations are performed on the CPU, we do not require here task synchronizations. Instead of filling one single interaction buffer set with multiple threads in parallel (requiring thread synchronizations), we have prefered to assign one interaction buffer set to each thread, which processes its part of the DTT independently. Once one of its interaction buffer is filled, the corresponding kernel (*P2P*, *P2PLeafTgt* or *M2L*) is executed on the iGPU and the thread waits for its termination. The smaller the interaction buffers are, the sooner the GPU computations will start, but the GPU computations must still be large enough to exploit all compute units and to offset the OpenCL kernel launch overhead. Three in-order command queues, one for each kernel type, are used in order to prevent from running two kernels of the same type concurrently (write conflicts) and to let kernels of different types run concurrently (as supported by the APU), with no scheduling constraint among kernels launched concurrently by different threads.

After extensive manual tuning of the different parameters on each iGPU, we show in Fig. 3.12 that using a parallel DTT with multiple CPU threads and multiple kernel launches, we manage to largely overlap the DTT on CPU with GPU computations, the remaining non-overlap part of the DTT representing 25% of the overall times (on both the APU and the Intel iGPU). In the end, the Intel iGPU performs here better than the APU, since: (i) the Intel iGPU compute power is 20% higher than the APU iGPU one, (ii) the *P2P* kernel performs better on the Intel iGPU for the optimal *Ncrit* values (32 or 48), (iii) the CPU compute power associated with the Intel iGPU is greater than the APU one (twice more cores), which eases minimizing the time of the non-overlap part of the DTT, as well as the evaluation step time.

3.3.5 Comparison with CPUs and discrete GPUs

Using the architectures listed in Table 3.3, Fig. 3.13 shows that both the AMD APU and the Intel iGPU with *pfalcON-iGPU* outperform a standard 8-core CPU with *pfalcON*. The Intel iGPU performance matches even the two 8-core CPU one or the 18-core CPU one (within a 7.5% margin). With respect to `Bonsai` on a K40c GPU, *pfalcON-iGPU* on the Intel iGPU is also 13% faster. We also emphasize here that 50M distributions can be run with *pfalcON* and *pfalcON-iGPU*, but not with `Bonsai`.

Morover, when considering the power efficiencies in Fig. 3.14a, the Intel iGPU is 1.7x to 2.7x more power-efficient than the CPUs (based on the theoretical TDP values), and without considering the CPU associated with the GPU, 4.2x more power-efficient than the K40c. When considering the cost efficiencies in Fig. 3.14b, the Intel iGPU is 3.0x to 5.0x more cost-efficient than the CPUs, and still without considering the associated CPU, 6.2x more cost-efficient than the K40c. Due to its very low price, the AMD APU offers here the best ratio,

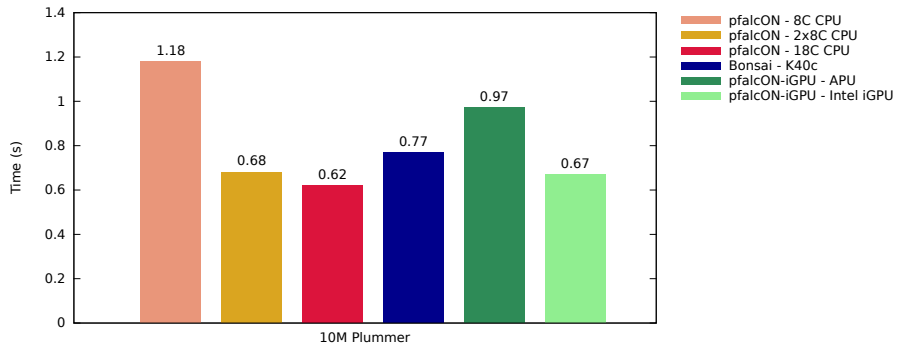
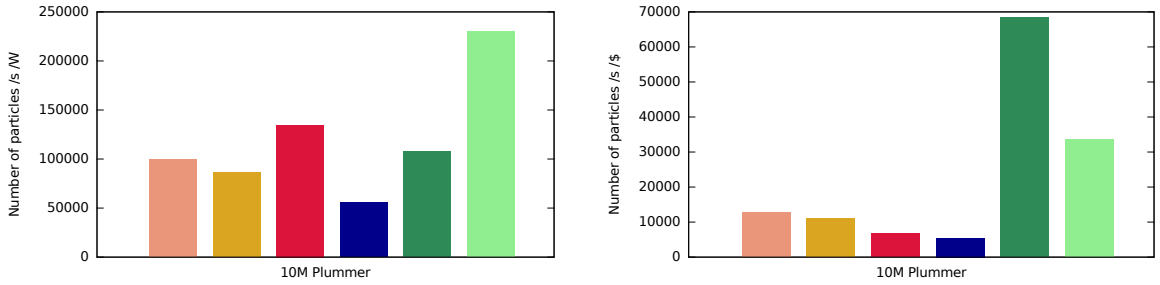


Figure 3.13 – Computation times (interaction and evaluation steps) for *pfalcON*, *pfalcON-iGPU* and `Bonsai`.



(a) Power efficiency (TDP based)

(b) Cost efficiency

Figure 3.14 – Power and cost efficiencies (based on Table 3.3 and on computation times of Fig. 3.13).

being 2.0x more cost-efficient than the Intel iGPU.

3.3.6 Conclusion

We have presented a hybrid CPU-GPU algorithm that deploys a fast multipole method (FMM) based on a dual tree traversal (DTT) on integrated GPUs (iGPUs) in an astrophysical context. In order to obtain the best performance results, we had to forsake the use of the mutuality of the far-field and near-field interactions in this heterogenous deployment. However, this deployment offers efficient SIMD processing of both the far- and near-field computations by aggregating multiple computations on the GPU. Thanks to its lower SIMD width and its greater compute-power, the Intel iGPU performs here better than the AMD APU, and can match the performance of two standard CPUs, of one high-end CPU, or even of the `Bonsai` GPU tree-code, being hence clearly more power- and cost-efficient.

It would be straightforward to extend such work to multiple compute nodes, using the LET (Local Essential Tree) technique as e.g. in `exaFMM`, since all data are stored in the main memory: this is another asset compared to GPU tree-codes. In the future, we plan to test *pfalcON-iGPU* on new integrated GPUs such as the forthcoming AMD Ryzen APUs. We also believe that our hybrid CPU-GPU algorithm could be efficiently deployed on other architectures such as integrated FPGAs (with OpenCL programming), or even discrete GPUs, especially those equipped with the NVIDIA NVLink interconnect. Finally, this work could be extended to other application domains of the FMM where low accuracies are required, e.g. when using FMM as a preconditioner [73].

Chapter 4

Handling the SIMD divergence

4.1 For generating correctly rounded mathematical functions

4.1.1 The Table Maker's Dilemma and Lefèvre algorithm

Since 1985, the IEEE 754 standard specifies the implementation of floating-point operations in order to have portable and predictable numerical software. Its latest revision [37,75] recommends the correct rounding of some elementary functions, like $\log$, $\exp$ and the trigonometric functions. Since such functions are transcendental, one cannot evaluate them exactly but have to approximate their evaluation. Hence, for precision-p floating-point numbers[1], a typical implementation of a mathematical function f will have to approximate the exact mathematical result $f(x)$ by $\hat{f}(x)$ with precision ϵ, and then round this approximation to p bits of precision. However, if for some argument x, $\hat{f}(x)$ is at a distance less than ϵ to a rounding breakpoint (where the result of the rounding function changes), it is impossible to determine the correct rounding of $f(x)$ from $\hat{f}(x)$ as illustrated in Fig. 4.1. Such an argument x is called a (p, ϵ) *hard-to-round* case (abbreviated as HR-case). The *Table Maker's Dilemma* [107] is hence defined as finding the necessary accuracy ε (*hardness-to-round*) such that both $f(x)$ and an approximation $\hat{f}(x)$ with accuracy ε round to the same precision-p floating-point number for every argument in the definition domain of f. This largest ε is given by the *hardest-to-round* arguments of f [107], that is to say the arguments requiring the highest precision to be correctly rounded when f is evaluated at. The hardest-to-round cases can be found by *exhaustive search*, which implies to browse each floating-point number in the domain of definition of the function. This approach is however prohibitive for double precision and for higher precisions.

In order to speed up the search for hardest-to-round arguments, the Lefèvre algorithm [95] uses local affine approximations of the targeted function. Using probabilistic assumptions [107], a "convenient" ϵ is first chosen so that there will be (in the end) few (p, ϵ) HR-cases. The domain of definition of the function f is then split into several domains D_i and an affine approximation P_i of the function is computed for each D_i. Thanks to the affine approximations, and taking into account the affine approximation error ϵ_{approx} with $\epsilon' = \epsilon + \epsilon_{approx}$, one can search for the (p, ϵ') HR-cases of P_i (corresponding to the (p, ϵ) HR-cases of f). This *HR-case search* can be efficiently performed on each domain D_i in polynomial time in p,

[1] p is the number of bits of the mantissa, e.g. $p = 53$ in double precision.

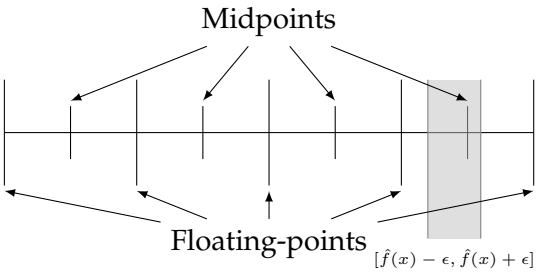


Figure 4.1 – Example of undetermined correct rounding for a value y computed with precision ϵ in the case of rounding to nearest, where the rounding breakpoints are the midpoints of floating-point numbers.

against an exponential time for the exhaustive search. The hardest-to-round cases are then found among the HR-cases with a localized exhaustive search, and one can finally deduce the *hardness-to-round* ε.

Higher degree approximations have been introduced since (SLZ algorithm [136]) in order to further reduce the asymptotic operation count for large values of p. However quadruple precision ($p = 113$) is still currently out of reach. We thus focus here on the double precision format ($p = 53$), for which the Lefèvre algorithm is as efficient as the SLZ algorithm in practice [38, 107]. The Lefèvre algorithm has already been used to generate all known hardness-to-round in double precision [107], and it offers fine-grained parallelism which is suitable for massively parallel architectures like GPUs [FGG12]: we therefore study the Lefèvre algorithm here. Even if the Lefèvre algorithm makes it possible to compute the hardness-to-round of elementary functions, it remains very computationally intensive. For example, it requires around five years of CPU time (at the end of the 90's) for the exponential function over all double precision arguments. Moreover, even if the hardest-to-round cases of some functions in double precision are known [107], this is still not the case for about half of the univariate functions recommended by the IEEE standard 754-2008. Furthermore, some scientific computations may require correctly-rounded implementations of other elementary functions, of specific compositions of elementary functions or even of elementary functions using non-standard formats or precisions. Being able to find the hardness-to-round of any elementary function in double precision in a reasonable amount of time would therefore be very useful.

In practice, both the affine approximation generation and the HR-case search are independent among the D_i domains. This data-parallel algorithm is thus embarrassingly and massively parallel which suits well to multi-core and many-core parallel architectures. However, while the polynomial approximation generation has a regular control flow [FGG16], the original HR-case search of Lefèvre algorithm (*Lefèvre HR-case search*), which is also the most time consuming step, presents divergence issues when executed on SIMD architectures. In the following, we show that efficiently solving the TMD on various multi-core and many-core SIMD architectures (CPUs, GPUs, Intel Xeon Phi), and scaling performance with the number of SIMD lanes, requires to jointly handle this divergence at multiple levels: algorithm, programming and hardware. We start by presenting a new regular HR-case search algorithm, detailed in [FGG16], which drastically reduces divergence in the execution flow on NVIDIA GPUs. Thanks to the OpenCL SPMD-on-SIMD (*Single Program Multiple Data*) programming model, we then present a performance portability study (detailed in [AFGZ16]) on various architectures.

Algorithm 1: Lefèvre HR-case test algorithm.

input : $b - a \cdot x, \epsilon', N$

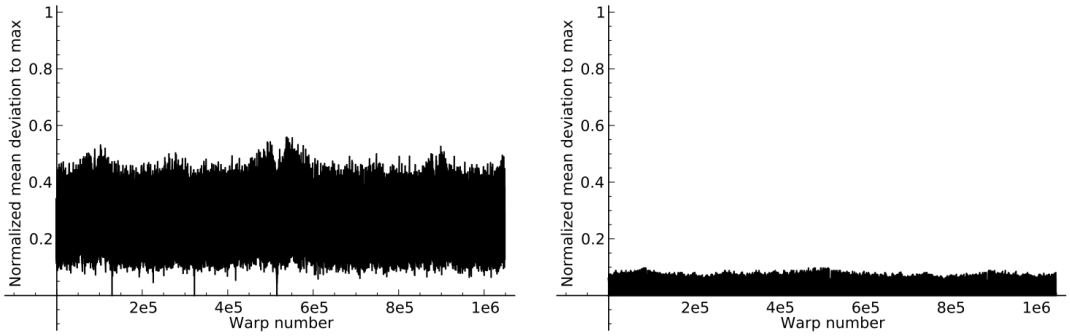
initialisation: $p \leftarrow \{a\}$; $q \leftarrow 1 - \{a\}$; $d \leftarrow \{b\}$; $u \leftarrow 1$; $v \leftarrow 1$;

if $d < \epsilon'$ **then return** Failure;
while True **do**
  if $d < p$ **then**
    $k = \lfloor q/p \rfloor$;
    $q \leftarrow q - k * p$; $u \leftarrow u + k * v$;
    if $u + v \geq N$ **then return** Success;
    $p \leftarrow p - q$; $v \leftarrow v + u$;
  else
    $d \leftarrow d - p$;
    if $d < \epsilon'$ **then return** Failure;
    $k = \lfloor p/q \rfloor$;
    $p \leftarrow p - k * q$; $v \leftarrow v + k * u$;
    if $u + v \geq N$ **then return** Success;
    $q \leftarrow q - p$; $u \leftarrow u + v$;

Related work. General solutions have been proposed to handle divergence on SIMD architectures, at the hardware level [22, 55, 101] as well as at the software level [54, 67, 127, 162]. We target here currently available hardware, and our HR-case searches offer very fine computation grains: the overhead of software solutions to handle divergence would be too high here (see [FGG12]). Up to our knowledge, there is no other specific work to reduce the SIMD divergence when solving the TMD. The reference C code of V. Lefèvre [95] is a CPU scalar code that can target multi-core and distributed multi-processor architectures, but does not exploit SIMD parallelism within each CPU core. It can be noticed that another implementation to solve the TMD has been designed for FPGA architectures [38], but this implementation relies on the exhaustive search.

HR-case test. The HR-case search on the polynomials P_i is done using an isolation strategy [94]. A HR-case Boolean *test* is executed on each domain D_i: it successes if there is no (p, ϵ') HR-case for P_i in D_i, and fails otherwise. If the test fails (that is to say, there might be a HR-case in D_i; false negatives are possible), we split D_i into sub-domains $D_{i,j}$, upon which we repeat the HR-case test. For each of these sub-domains failing the HR-case test, we finally perform exhaustive search.

Lefèvre HR-case test takes as argument a degree one polynomial P_i or $P_{i,j}$. As we do not need the dynamic range of floating-point numbers, we use fixed-point arithmetic to avoid rounding errors, and we apply a suitable change of variable to write P_i or $P_{i,j}$ as a polynomial $b - a \cdot x$, while representing only the 64 bits after the p^{th} bit of the significands of a and b as 64-bit integers. Hence we also consider $x \in \mathbb{N}$. This HR-case test then computes a lower bound on the distance between the values of $b - a \cdot x$ for $x < N$ and the rounding breakpoints, with N the number of arguments to test in D_i or $D_{i,j}$. Thanks to the three distance theorem [131] and to the continued fraction formalism, this is achieved by computing the continued fraction expansion of a with the Euclidean algorithm, and a particular decomposition of b in the sequence of partial remainders. This leads to a $\mathcal{O}(\log N)$ operation count, against $\mathcal{O}(N)$ for the exhaustive search. Comparing this lower bound to ϵ', we can then determine whether there is potentially a (p, ϵ') HR-case in the domain or not. The Lefèvre HR-case test is presented in Algorithm 1.



(a) Lefèvre HR-case search

(b) Regular HR-case search

Figure 4.2 – Normalized mean deviation to the maximum of the number of main loop iterations per CUDA warp, on NVIDIA GPUs, among the 2^{20} CUDA warps required for the *exp* function in the domain $[1; 1 + 2^{-13}]$.

4.1.2 A regular algorithm for the HR-case search

In [FGG12], we deployed the Lefèvre HR-case search on NVIDIA GPUs in CUDA, using one GPU thread per domain D_i or $D_{i,j}$: each thread performs the corresponding HR-case test over its domain. Although offering performance gain over a multi-core CPU, we underlined a problem in Lefèvre algorithm execution on GPU architectures: the execution flow is highly divergent from one thread to another. There are indeed three sources of divergence in Algorithm 1: (i) the main unconditional loop (line 3), whose number of iterations depends on the value of the arguments; (ii) the main conditional statement (line 4), whose scope contains all the intructions within the main loop; and (iii) finally the divisions, which are computed using a hybrid implementation with a user tunable parameter to potentially enable repeated subtractions instead of the division instruction.

Even though the hybrid divisions affect the control flow, they do not lead to a strong divergence issue at runtime since almost all the computed quotients are expected to be small in practice [FGG16]. However when processing multiple instances of the Lefèvre HR-case test in parallel on GPUs, the main conditional statement and the main loop have a strong performance impact because of the partial SIMD execution of GPUs. Both are induced by conditioning the computation of the quotients of the continued fraction of a by the value of b. To our knowledge there is no *a priori* information on the number of loop iterations or on the branch executed at each iteration that would enable us to statically reorder the domains D_i in order to decrease this divergence. We also tried to use software solutions to reduce the impact of the loop divergence [FGG12]: no performance gain was obtained because the computation is very fine-grained.

To highlight the impact of loop divergence during Lefèvre test execution, we introduced in [FGG12] an indicator named the *normalized mean deviation to the maximum*. When processing concurrently n independent instances of a divergent loop on a SIMD unit with n lanes, the number of loop iterations issued in total is the maximum number of loop iterations issued among all the lanes of the SIMD unit. This indicator aims thus at giving the average percentage of loop iterations for which a lane remains idle during the SIMD execution. More formally, we denote ℓ_i the number of loop iterations to issue for the lane i and we number the lanes within a SIMD vector from 1 to n. If $\ell = \{\ell_i, i \in [\![1, n]\!]\}$, the Normalized Mean Deviation to the Maximum (NMDM) is defined as: $\text{NMDM}(\ell) = 1 - \frac{\text{mean}(\ell)}{\max(\ell)}$. In Fig. 4.2a,

Algorithm 2: Regular HR-case test.

```
   input : b − a · x, ε′, N
1  initialisation:
2  p ← {a}; q ← 1; d ← {b}; u ← 1; v ← 0;
3  if d < ε′ then return Failure;
4  while True do
5  |  if p < q then
6  |  |  k = ⌊q/p⌋;
7  |  |  q = q − k ∗ p; u = u + k ∗ v;
8  |  |  d = d mod p;
9  |  else
10 |  |  k = ⌊p/q⌋;
11 |  |  p = p − k ∗ q; v = v + k ∗ u;
12 |  |  if d ≥ p then
13 |  |  |  d = (d − p) mod q;
14 |  |  if u + v ≥ N then return d > ε′;
```

Algorithm 3: Regular HR-case test, unrolled.

```
   input : b − a · x, ε′, N
1  initialisation:
2  p ← {a}; q ← 1; d ← {b}; u ← 1; v ← 0;
3  while True do
4  |  k = ⌊q/p⌋;
5  |  q = q − k ∗ p; u = u + k ∗ v;
6  |  d = d mod p;
7  |  if u + v ≥ N then return d > ε′;
8  |  k = ⌊p/q⌋;
9  |  p = p − k ∗ q; v = v + k ∗ u;
10 |  if d ≥ p then
11 |  |  d = d − p mod q;
12 |  if u + v ≥ N then return d > ε′;
```

we measured the NMDM of the main unconditional loop of Lefèvre HR-case search execution on a NVIDIA GPU ($n = 32$) on a set of domains D_i for the exponential function. We can see that the NMDM is uniformly high with an average NMDM of 25.6%, which means that a SIMD lane remains idle on average 25.6% of the number of loop iterations issued on its SIMD unit. This divergence in Lefèvre HR-case test is mainly due to the fact that, in order to minimize the number of instructions, the algorithm goes from the subtraction-based Euclidean algorithm to the division-based one depending on the value of b.

In [FGG16], we proposed a new HR-case test which presents a regular execution, as illustrated in Algorithm 3. Thanks to the continued fraction formalism, we have rewritten the algorithm in order to avoid the dependence between the computation of the continued fraction expansion of a and the value of b. This first turns the unpredictable main conditional statement of Lefèvre algorithm into a deterministic test (see Algorithm 2), which is alternatively true and false and can thus be removed by unrolling two loop iterations as in Algorithm 3. Second, contrary to the Lefèvre test which aims at minimizing the number of operations performed (in scalar mode on CPU), we now rely only on the division-based Euclidean algorithm: a full quotient of the Euclidean algorithm is entirely computed at each loop iteration in the regular HR-case test, which can introduce extra computations. Thereby however, as the number of quotients to compute is almost constant from one domain D_i to the next, the number of iterations is also very stable from one domain to the next in the regular HR-case test. We therefore reduce the mean NMDM per SIMD computation on NVIDIA GPUs from 25.6% to 0.1% (see Fig. 4.2b). Finally, GPU branch predication can efficiently handle the remaining short " if " blocks.

As shown in Table 4.1, and detailed in [FGG16], such regular test offers performance gains up to 3.44x over Lefèvre test on NVIDIA GPUs (with CUDA). When comparing an MPI parallelization of the reference C code of V. Lefèvre [95] on a high-end[2] hex-core CPU with our CUDA deployment on a high-end NVIDIA GPU, the regular HR-case search (based on the regular HR-case test) on GPU delivers a 6.63x speedup over the regular HR-case search on CPU and a 7.43x speedup over the Lefèvre HR-case search on CPU. Due to the extra computations introduced by the regular HR-case test, such performance gain may vary

[2] At the time of writing.

	1 CPU core	6 CPU cores (MPI)	GPU	$\frac{\text{6 CPU cores}}{\text{GPU}}$
Lefèvre HR-case search	36816.10	5292.67	2446.27	2.16
Regular HR-case search	34039.94	4716.97	711.92	6.63
Lefèvre / Regular	1.08	1.12	3.44	-

Table 4.1 – Timings (in seconds) and performance gains for $\exp(x)$ in double precision over the binade $[1, 2[$ (Intel X5650 6-core CPU, NVIDIA C2070 GPU).

depending on the binade and on the targeted mathematical function, but remains overall significant [FGG16]. Such performance gap is partly due to the lack of SIMD computations on the CPU: we will therefore now consider the vectorization of the HR-case searches on CPUs and on Xeon Phi, as well as on other GPUs.

4.1.3 Performance portability of the SIMD divergence handling

Relevant programming paradigm. As detailed in [AFGZ16], despite a strong rewriting of the original code, C programming has been found unsuitable for vectorizing the (regular or Lefèvre) HR-case search on CPUs and on the Xeon Phi. Manual SIMD programming with intrinsics would indeed be an especially tedious task because of the multiple nested `while` loops and conditional branches, each one implying a different mask to handle the divergence on CPU SIMD units. Automatic vectorization, or guided vectorization with compiler directives (from Intel C/C++ Compiler, or from OpenMP 4.X), also failed to vectorize the code, due to `while` loops with unknown iteration numbers [76], and to an `output` dependency among loop iteration. Indeed, the found HR-cases are written consecutively in memory thanks to a unique counter.

We thus rely on the SPMD-on-SIMD (*Single Program Multiple Data*) programming model (see Sect. 1.2), using OpenCL in order to maintain one single source code for both CPUs and GPUs, and to target other GPUs like the AMD ones. On multi-core CPUs, we use the Intel OpenCL SDK[3] which provides OpenCL implicit vectorization while supporting conditional statements as well as `while` loops in the OpenCL kernels [125]. Like in the previous CUDA implementation, atomic operations are used in OpenCL to consecutively write the found HR-cases in memory. We thus emphasize that the HR-case search of the Table Maker's Dilemma fits naturally with the SPMD-on-SIMD programming model: each work-item processes one (or a few) D_i domain(s), and only a few atomic operations are required for correct work-item synchronization. We then fully exploit the data parallelism of this massively parallel application to concurrently process the numerous work-items on the SIMD units (as well as on all the available CPU cores).

Performance portability. We now consider performance tests for the HR-case search on the *exp* function in double precision over the 1024 first intervals $I_{0..1023} = [1; 1 + 2^{-3}[$ of the binade $[1; 2[$. As shown in Fig. 4.3, compared to the previous Fermi architecture (C2070), our HR-case search GPU implementation scales well on the newer Kepler GPU architecture (K20c) which offers a much higher number of GPU cores. Moreover, thanks to its regular execution flow, the regular HR-case search offer a 2.7x or 2.9x performance gain over the Lefèvre HR-case search on the NVIDIA GPUs: the gain is here similar since both GPUs

[3] See: https://software.intel.com/en-us/intel-opencl

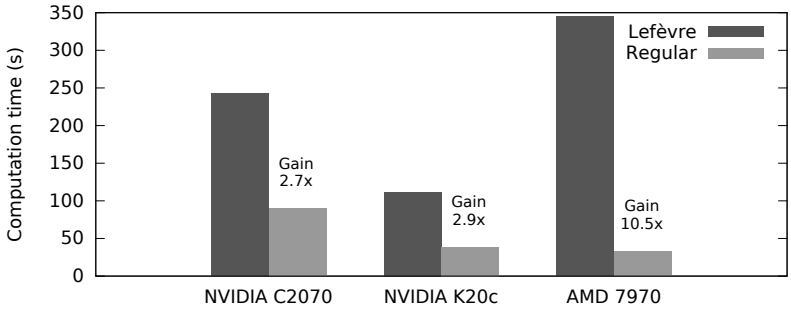


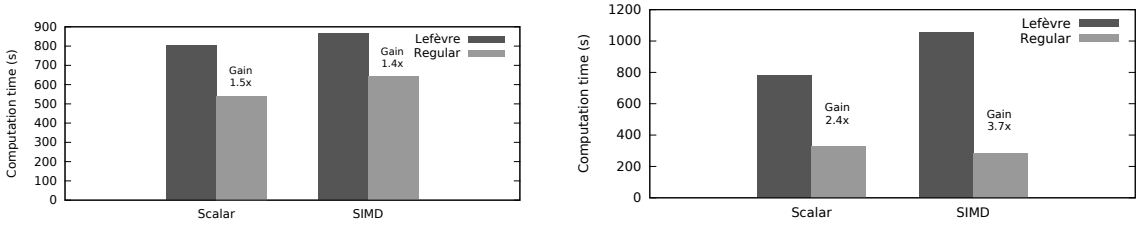
Figure 4.3 – HR-case searches computation times on various GPUs.

have the same SIMD width (32 work-items in a NVIDIA "warp"). The performance gain is however much greater (10.5x) on AMD GPUs due to their larger SIMD width (64 work-items in an AMD "wave-front" [7]). Comparing the AMD Radeon HD 7970 and the NVIDIA K20c, whose hardware compute powers are similar, one can see that only thanks to the regular HR-case search similar application performance can be achieved on these two GPUs.

As shown in Fig. 4.4, compared to the Lefèvre HR-case search, the regular HR-case search still improves performance on CPU (AVX2) and on a Xeon Phi coprocessor (Knights Corner), but the SIMD versions offer no (AVX2) or very low (Xeon Phi) performance gains over their scalar counterpart. This is first explained by the differences in SIMD divergence handling on CPU and on GPU [69]. When the control flow diverges on a GPU SIMD unit, a mask register is set according to the condition evaluation: each processing element then either performs the following instruction or remains idle. A stack of mask registers is used to handle nested divergence levels. This is handled dynamically by the GPU hardware, which can then skip at runtime branches where all processing elements would be idle. When control flows diverge within a CPU SIMD unit, mask registers are also used to handle divergence among the SIMD lanes. On AVX, all computations are always performed by all the SIMD lanes (*predication*) [147]. On Xeon Phi, the overhead of masking is lower since all Xeon Phi SIMD instructions directly support a 16-bit mask to control which lanes are active or not during the instruction execution. However, on CPUs and on Xeon Phi all this is handled explicitly in software by the compiler. This implies a general overhead[4] compared to the GPU hardware management, and can also be crucial for the SIMD performance of our specific application. Both HR-case tests show indeed important static divergence (at compile time, in their control flow: see Algorithms 1 and 3), but the regular HR-case test presents low dynamic divergence (at runtime, in its execution flow), as shown in Fig. 4.2b. This low execution flow divergence can thus be handled efficiently by the GPU hardware, while the CPU - Xeon Phi compiler has to set all the required masks for predication according to the control flow divergence of the source code. It can be noticed that SIMD divergence handling is still improving on GPU at the hardware level: see for example the independent thread scheduling of the new NVIDIA Volta architecture [3].

The second issue with the vectorization of the HR-case test on CPU and on Xeon Phi is the lack of vector integer division instruction in AVX2 [77], in the Xeon Phi SIMD instructions and even in the forthcoming AVX-512 [79]. The compiler therefore uses the scalar integer

[4] As far as masks with all zeros or all ones are concerned, it has to be noticed that recent work can detect these cases at runtime in order to avoid using code with predication when possible on CPUs [147].



(a) AVX2 server (Intel Xeon E3-1275 v3 CPU)

(b) Xeon Phi coprocessor (Knights Corner 5110P)

Figure 4.4 – Performance results for Lefèvre and regular HR-case searches, in scalar and SIMD modes.

division instruction (on AVX2), or emulates the vector integer division with optimized intrinsics from the Intel Short Vector Math Library - SVML (on Xeon Phi), which offer limited speedups (see [AFGZ16] for details). Finally, the low Xeon Phi performance is also explained by the compiler emulation of 64-bit integer SIMD arithmetic operations with 32-bit integer SIMD operations (while 64-bit scalar operations are not emulated).

In the end, for a (roughly) same power consumption, both NVIDIA and AMD high-end GPUs outperform a CPU or Xeon Phi server by a 6.25x performance gap, due to this inefficient SIMD execution of the HR-case search on the CPUs and on the Xeon Phi.

4.1.4 Conclusion

We have here shown that handling efficiently the divergence on SIMD architectures for the most time consuming step of the Table Maker's Dilemma solving requires to introduce regular algorithms, implemented with the relevant programming model, but also depends on the hardware.

Thanks to the continued fraction formalism, we have first rewritten this step in order to strongly reduce the divergence in the conditional statements within the main loop, as well as to reduce the execution flow divergence on this main loop, at the price of some extra computations. This results in a 3.4x performance gain on GPU compared to the original, divergent, algorithm.

Thanks to the SPMD-on-SIMD programming model and to the OpenCL portability, we have shown that such algorithm is more efficiently deployed on GPU (NVIDIA or AMD) SIMD units than on CPU or Xeon Phi (Knights Corner) ones. This is mainly due to the SIMD integer division implementation and to the static software handling of divergence on CPUs and on Xeon Phi. This latter implies indeed an overhead compared to the dynamic hardware handling of divergence on GPUs, and cannot take full advantage of our regular algorithm, which presents important divergence in its control flow, but low divergence in its execution flow.

In the future, we could target the new AVX-512 SIMD instruction set which could lead to better SIMD performance gains on CPU, especially for our regular algorithm. It would also be very interesting to compare an OpenCL deployment of our algorithm on FPGAs with these GPU and CPU deployments, as well as with other FPGA implementations for solving the TMD [38].

4.2 For numerical validation using stochastic arithmetic

Three main approaches exist for the numerical validation of scientific codes.

Fistly, interval arithmetic [6, 89] replaces all operands in floating-point operations by intervals containing the exact value. However, these intervals can grow very large as the compensation in rounding errors is not taken into account: this leads to an overestimation of the rounding errors. To prevent intervals from expanding too much, specific algorithms and methods have been developed [6, 89], but usually require recoding the application. In terms of performance, recent implementations can show a good scalability and a low overhead [123]. An interval arithmetic library for GPUs is presented in [35].

Secondly, backward error analysis [157] considers that instead of an approximate solution to an exact problem, we compute the exact solution to an approximate problem. By studying the behaviour of an application when its entry is perturbed, the direct error can be deduced by an estimation of the condition number [70]. This method has a low overhead in terms of execution time, but does not support every type of problem: it is used mainly for linear problems, e.g. in the MAGMA library [148] for high performance linear algebra.

Thirdly, numerical validation can also be performed with a probabilistic approach based on several executions of the program to control. This approach enables one to estimate rounding errors and is used in various tools such as CADNA [83, 91], MCALIB [52], Verificarlo [41], VERROU [56], that differ by the number of executions required, by their implementation in the user program and by their ability to detect floating-point operations responsible for numerical instabilities.

We have focused on Discrete Stochastic Arithmetic [153], the probabilistic approach implemented in the CADNA[5] library, and we present here how we have improved the CADNA library for the numerical validation of scientific high performance simulations on CPUs. This has been achieved by reducing the overhead of scalar executions and by enabling the (previously impossible) use of the SIMD units, these two improvements being closely related. More details can be found in [EBFJ15].

4.2.1 The CADNA library

Thanks to three executions of floating-point operations with a random rounding mode in the user program, CADNA estimates, with a 95% confidence level, the number of exact significant digits of any computed result. This number of exact significant digits is an estimation of the number of digits unaffected by the rounding errors. We first present the CADNA implementation at the start of this work (version 1.1.9).

The CADNA library relies on new numerical types: the stochastic types. In practice, classic floating-point variables are replaced by the corresponding stochastic variables, which are composed of three floating-point fields and an integer field to store the accuracy. The library contains the definition of all arithmetic operations, order relations and mathematical functions for the stochastic types. For instance, let us consider an arithmetic operation $\circ \in \{+, -, *, /\}$ between two stochastic variables A and B. This arithmetic operation is performed three times: once for each of the three associated floating-point fields A_i and B_i ($i = 0..2$). The rounding mode is randomly set to rounding towards $+\infty$ or $-\infty$ for the first two operations, and $A_2 \circ B_2$ and $A_3 \circ B_3$ are computed with opposite rounding modes.

[5] Control of Accuracy and Debugging of Numerical Applications: http://cadna.lip6.fr

Because all operators are redefined for stochastic variables, the use of CADNA in a program requires only a few modifications: essentially changes in the declarations of variables and in input/output statements.

CADNA can also detect numerical instabilities which occur during the execution of the code. These instabilities are usually due to numerical noise, *i.e.* results that have no more correct digits because of rounding errors, and are of four types. An instability can occur first in an overloaded mathematical function or in the test evaluation of a branching statement. A numerical instability is also reported in the case of a cancellation, *i.e.* the subtraction of two very close values which generates a sudden loss of accuracy. A last type of instabilities is related to the self-validation of CADNA: a multiplication where the two operands are numerical noise and a division where the divisor is numerical noise. These instabilities indicate that the validity of the accuracy estimation has been compromised and the CADNA results cannot be relied on. The user can specify the instabilities to be detected.

Performance impact of CADNA. A program that uses the CADNA library executes three times each arithmetic operations and a few additional operations if instability detection is activated. However, once a program had been instrumented with the CADNA library, there is a very important overhead on computation time, up to about 2 orders of magnitude slower [80], depending on the program and on the level of instability detection. In highly optimized programs, such as BLAS routines, it could even go up to 1000 times [105]. This is mainly due to the cost of instability detection and to the cost of stochastic operations.

Instability detection is based mostly on the test of whether a stochastic value is significant or not. In the case of cancellation detection, the test depends on the difference of number of exact significant digits between the operands and the result. The computation of each number of exact significant digits however relies on the `log10` function, whose execution is costly. Thanks to an approximation of the integer part of the `log10` evaluation by the base 2 exponent of the argument, we have managed to reduce the CADNA overhead by 43% when using cancellation detection (see [EBFJ15]).

Stochastic arithmetic operations are implemented with the help of standard operations and the explicit change of rounding mode in the FPU (Floating-Point Unit). At the beginning of a program using CADNA, the rounding mode is arbitrarily set to $-\infty$ or $+\infty$. Then the rounding mode is explicitely and randomly changed, from $+\infty$ to $-\infty$ or inversely. Changing the rounding mode is of relatively low cost in itself (only a few assembler instructions required: reading, modifying and writing the control word of the FPU), but it flushes the pipelines of the FPU, requiring several processor cycles to refill them. This is especially disadvantageous for CADNA, as up to three rounding mode changes can occur in every stochastic operation. As HPC applications aim to fill these pipelines as much as possible to improve their performance, CADNA has an even more detrimental impact in a HPC context. Besides, the stochastic operations are implemented by overloading the arithmetic operators for stochastic types. As such, they are defined as functions or methods, which implies a function call overhead (for each arithmetic operation) and can also prevent pipelining successive operations. Finally, the reliance on the rounding mode of the hardware makes it impossible to use SIMD parallelism with the current CADNA version. Indeed, SSE, AVX or Xeon Phi vector units only enable the rounding mode control on the whole vector, not on a lane by lane basis. This would result in the same rounding mode being selected for operations in the same SIMD instruction, breaking the hypothesis of CADNA that the rounding mode should be chosen independently for each operation. This is why CADNA (version 1.1.9) was unable to use vector instructions.

4.2.2 A new CADNA version for HPC applications

We now present our improvements for the CADNA library in order to reduce its overhead and enable the vectorization of CADNA codes.

Rounding mode emulation. The random rounding mode of CADNA relies on changing the rounding mode of the FPU. In addition to flushing the pipelines of the FPU, and to preventing vectorization for codes instrumented with CADNA, this forces to disable any optimization when compiling the CADNA code with the gcc compiler, as optimizing may generate incorrect code with gcc when changing the rounding mode, even when using the `-frounding-math` option[6]. We thus propose in our new CADNA version to emulate the rounding modes toward infinity taking advantage of the following properties: $a \oplus_{+\infty} b = -(-a \oplus_{-\infty} -b)$ (similarly for $\ominus$), and $a \otimes_{+\infty} b = -(a \otimes_{-\infty} -b)$ (similarly for $\oslash$), where $\oplus_{+\infty}$ and $\otimes_{+\infty}$ (resp. $\oplus_{-\infty}$ and $\otimes_{-\infty}$) are the floating-point operations rounded towards $+\infty$ (resp. $-\infty$). Since the results of each rounding mode can be obtained from computation made in the other rounding mode, there is no need to change the rounding mode of the FPU during the execution of the program. We only require to set the rounding mode towards $+\infty$ or $-\infty$ once, in the CADNA initialization function.

As our goal is also to enable SIMD parallelism, we avoid using `if` blocks depending on the chosen rounding mode, since this would introduce divergence in the execution flow. We could have multiplied the operands and the results, according to the aforementioned properties, by 1 or -1. Although this avoids the divergence, this would come at the cost of two or three floating-point multiplications for each sample of the stochastic value. Instead, we apply a random mask to the sign bit of the binary representation of the floating-point numbers to change their sign as required, without relying on the multiplication.

As there is no more rounding mode change in the computation part of the application code, we can use optimization options of `gcc` without the risk of floating-point instructions being moved and executed in an unintended rounding mode. This enables the optimization of the CADNA library for high performance.

Inlining. To enable inlining, we have moved the code of the stochastic arithmetic operators from the library source code to the CADNA header file. Thereby, we can get rid of much of the overhead of CADNA due to function calls. Moreover, this enables optimizations such as pipelining several stochastic operations, or interleaving their instructions.

Random generator. The rounding mode selection for stochastic operations is based on a randomly generated bit. The 1.1.9 version of CADNA uses an intrinsically sequential, and difficult to vectorize, method which aims at reducing the computation cost. An array is pre-filled with randomly generated numbers in the CADNA initialization, and bits are picked by sequentially reading each number bit per bit. In a SIMD context, this implies to compute a different bit index for each lane and to increment the index according to the vector width.

Following a similar idea proposed for the CADNA prototype on GPU [84], we have chosen to replicate the random generator for each SIMD lane in order to account for any possible vector width and to have a straightforward and efficient vectorization. Instead of pre-generating an array that would be duplicated, the random number generation will now be executed on the fly. As such, an integer will be randomly generated and read bit per bit for each random pick. When every bit has been picked, every lane will produce a new number at the same time, as the bits were consumed at the same rate on each lane. Where the pre-

[6] GCC bug 34678 - optimization generates incorrect code with -frounding-math option.
See: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=34678

Program 1 Loop of independent floating-point operations

```
for (int i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

vious version of CADNA used a 16-bit `short` integer generator, we now use the generator presented in Mohanty and al. [104] which produces 32-bit `int` values. This new generator also has a longer period, a good statistical distribution, and uses only integer addition, multiplication, and logical operations, whereas the previous also used integer division. Integer division is usually relatively inefficient and most vector instructions sets (such as SSE, AVX and the Xeon Phi ones) do not contain integer division.

The dynamic generation will have the added benefit of reducing the memory footprint and memory accesses of CADNA, at the cost of slightly more computation. As computation is becoming increasingly cheaper than memory accesses on current and future HPC architectures, this should also yield an improvement in the performance of CADNA applications.

Vectorizing. For vectorization, several programming paradigms are possible, as listed in Sect. 1.2. Using intrinsics with CADNA, would be especially tedious as we have to handle the composite data types and replace each intrinsic call with a corresponding CADNA version. On the other hand automatic vectorizing needs to ensure that the dependencies of the scalar code are respected when vectorizing. For instance, in IEEE floating-point arithmetic, the iterations of the loop of Program 1 are independent from each other and can be automatically vectorized. However, with CADNA, even though the variables in these operations are completely different, the process of choosing the rounding mode introduces a dependency. Indeed, the random bit chosen for one iteration is necessarily picked after the previous iteration. As such, automatic vectorizing cannot be achieved for any code instrumented with CADNA. Regarding compiler directive (such as in OpenMP 4.X), we must duplicate the random generator on each lane in order to ensure that the randomness of the rounding mode is retained. However there is no lane identifier, necessary to access each generator independently, when using these directives. In the end, we thus focus here on the SPMD-on-SIMD model. In addition to its assets (see Sect. 1.2), there is a lane identifier that enables us to easily replicate the random generator (with a different seed for each lane). We have elected to choose `ispc` over OpenCL, since at the time of this work OpenCL did not support the overloading of operators necessary for CADNA. Nevertheless, the same process could be applied for other SPMD-on-SIMD languages, as long as operator overloading is supported.

Thanks to our previous contributions, very few changes are necessary to adapt the CADNA library to `ispc`. Indeed, adding relevant ispc attributes to variables (`varying` for lane specific variables, `uniform` for vector shared ones) and initializing the seed for each lane were the only necessary adjustments. Like C++ code, ispc code can be instrumented with CADNA by simply changing the types of the variables to stochastic ones.

Execution masks. In the current version of CADNA, the detected instabilities are chosen at execution time. As such, detection flags are checked dynamically, leading to conditional branches in the CADNA code. However, when a given instability is not detected, these branches still produce divergence in the control flow, which can prevent vectorization or lower its performance due to the required software execution masks (predication). We have thus replaced the tests in these branches by preprocessor directives that can be evaluated at compile time. We can still change the detected instabilities, but now only at compile time.

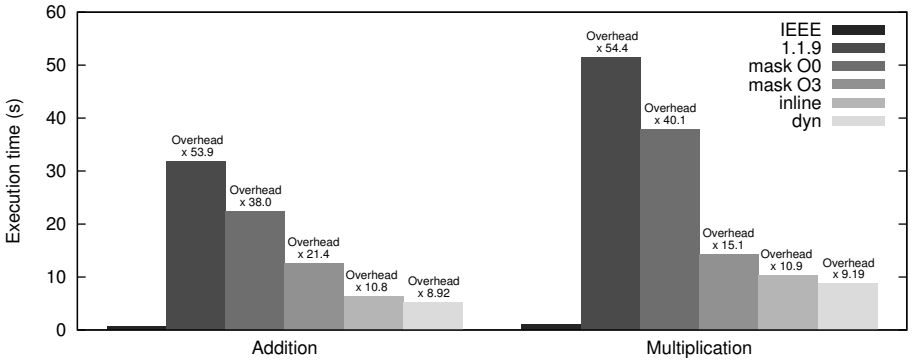


Figure 4.5 – Scalar performance for operations (compute-bound) and overhead over IEEE performance.

4.2.3 Performance results

To assess the impact of our different improvements on the performance of CADNA, we will first measure the CADNA overhead on purely "arithmetic" benchmarks. As such, we will perform only one arithmetic operation (addition or multiplication) and repeat it with a high number of floating-point numbers in both compute- and memory-bound versions. Then, considering more realistic applications, we will rely on a 3D finite difference stencil computation (memory-bound), as well as on a Mandelbrot set computation (compute-bound) for which the use of CADNA will allow us to better determine if the sequence corresponding to a specific point of the 2D plane is bounded or not. These benchmarks will be first considered in scalar C/C++ versions, then in vectorized versions written with `ispc` (version 1.8.2), on a single core of an Intel Xeon E3-1275 CPU with AVX2.

Scalar performance results. For each benchmark, we will compare a version implemented with IEEE arithmetic (*IEEE*) to several CADNA versions of the same code (with only self-validation activated): the previous version (named *1.1.9*); basing the computation on one rounding mode and masks, using the `gcc -O0` flag for no optimization (*mask O0*) or using high level optimizations with the `-O3 gcc` flag (*mask O3*); using *mask O3* and adding the inlining (*inline*); and finally using *inline* and the dynamic random generation (*dyn*).

For the compute-bound arithmetic benchmarks, we see in Fig. 4.5 that our successive modifications to the CADNA library significantly improve the performance. Most of the total gain in performance is gained from the *mask O3* version, due to the combined effects of compiler optimization and the absence of change in the rounding mode of the FPU. Performance further increases with the *inline* version, that allows a tighter integration of the CADNA code in the application code. Moreover, the *dyn* version also slighlty improves performance, even though its main focus was to prepare the random generator for vectorization. Overall, we reduced both addition and multiplication overheads by 83%.

For the memory-bound arithmetic benchmarks, we see in Fig. 4.6 that each optimization (*mask O0*, *mask O3*, *inline*) also leads to a significant improvement in performance. We can also see that the overhead of the addition and multiplication benchmarks are similar to their compute-bound equivalent. Indeed, with CADNA we have at least 3 times more computations (for each sample), while needing 4 times more memory accesses. But these memory accesses can be performed at once within the same cache line (as the members of the stochastic types are contiguous in memory), which lower their performance impact. Moreover, extra operations (random number generation, masking, instability detection) also increase the

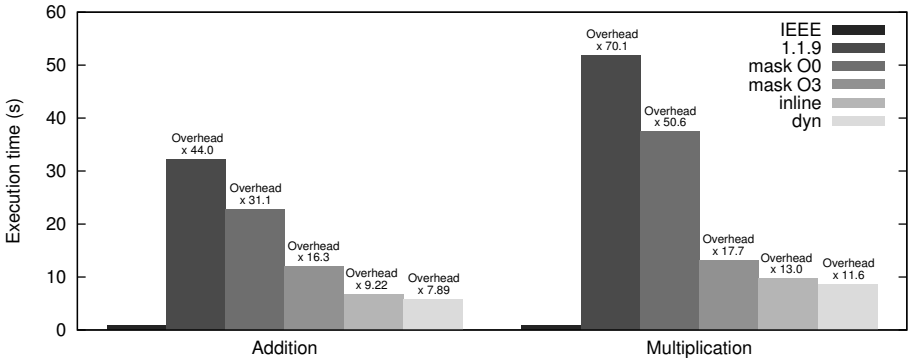


Figure 4.6 – Scalar performance for operations (memory-bound) and overhead over IEEE performance.

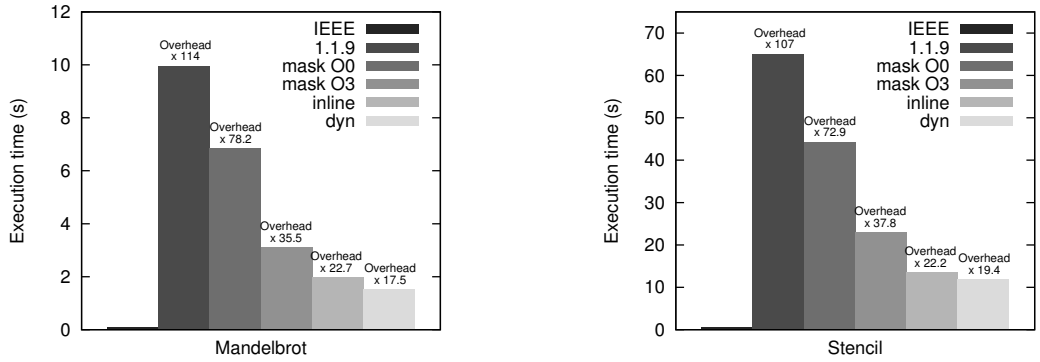


Figure 4.7 – Scalar performance for applications and overhead over IEEE performance.

computation cost with CADNA. The arithmetic intensities of our memory-bound benchmarks thus increase with CADNA. In the end, similarly to the compute-bound case, the overhead was reduced by 82% on addition and by 83% on multiplication.

On more realistic applications, we see from Fig. 4.7 that performance has much improved too. However, the overhead is higher than for our arithmetic benchmarks. The Mandelbrot set computation is indeed even more arithmetic intensive than our arithmetic benchmarks. For each iteration, there are more floating-point instructions than in the arithmetic benchmarks, and when the code is instrumented with CADNA, the overhead increases more. The stencil computation is memory-bound but contrary to the memory-bound arithmetic benchmarks, the 3D memory access pattern lowers the effectiveness of the memory caches and prefetch especially for the CADNA versions. Nevertheless, the overhead was reduced by 85% on the Mandelbrot set computation and by 82% on the finite difference stencil.

Vectorized performance results. We will consider here: a vectorized version of the best scalar version (*dyn*); and a version using *dyn* and instability detection tests with `#define` preprocessor directives evaluated at compile time (*define*) as presented in subsection 4.2.2.

The vectorized versions of the memory-bound arithmetic benchmarks have shown little to no gain over the corresponding scalar versions (tests not presented here). Their performance is indeed limited by the bandwidth of the caches and the memory prefetch, and is not improved by the vectorization. For compute-bound arithmetic benchmarks, we can see from Fig. 4.8 that the IEEE speedup on vectorization is almost maximum, here on AVX2

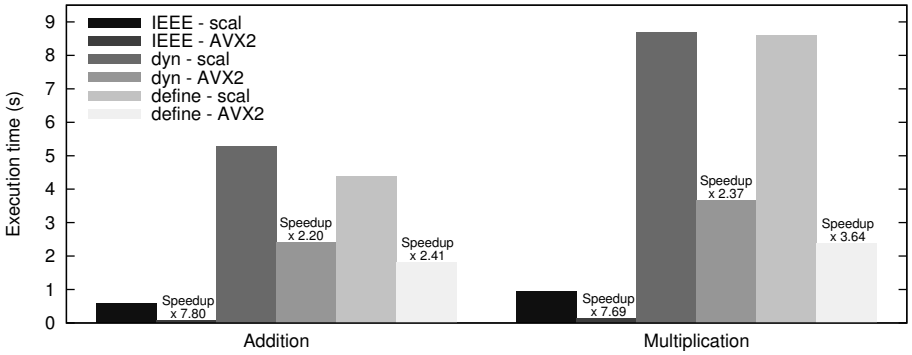


Figure 4.8 – Vectorized performance for operations on AVX2 (compute-bound), with speedup on scalar version.

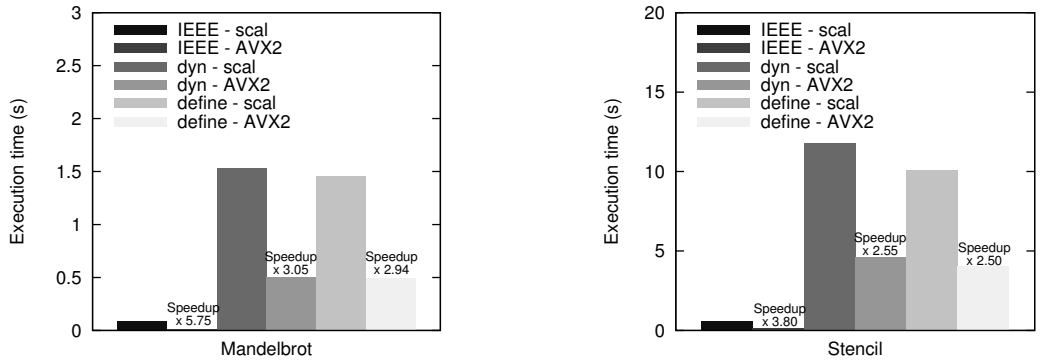


Figure 4.9 – Vectorized performance for applications on AVX2, with speedup on scalar version

SIMD units. We have also achieved vectorization for CADNA with speedups up to 3.64x, whereas no vectorization was possible with the previous CADNA version. The CADNA speedups are however lower than the IEEE ones. After analysis with the Intel VTune Amplifier profiler[7], we have found that this is due to the memory accesses that are much more costly with CADNA. This can be explained by the AoS (*Array of Structures*) memory layout of our stochastic types. The AoS data layout is indeed not best suited for SIMD processing, since it requires special memory loads (*gather*) and stores (*scatter*) that are less efficient than simple vector loads and stores. Finally, we also see the beneficial effect of removing the execution masks from our code with the *define* version. As only self-validation is activated, the multiplication benchmark is the only one that creates execution masks during the computation. As such, it benefits from this improvement much more than the addition.

On realistic applications, we see from Fig. 4.9 that we achieve a speedup of up to 3.05x. We also observe that the speedup on the Mandelbrot set computation with the *dyn* version is slightly higher than for our other benchmarks. This confirms that the AoS memory layout is partially responsible for the lower CADNA speedup, as this application is the only one that does not need to load stochastic values from memory. The *define* version improves performance, without improving the speedup due to vectorization.

On the whole, our vectorized CADNA versions have a global overhead on the IEEE vectorized versions that varies from 19.2x to 32.4x. Although these are higher than in scalar,

[7] https://software.intel.com/en-us/intel-vtune-amplifier-xe

there is still a net improvement over the former (non-vectorizable) version of CADNA which has an overhead between 407x and 651x over the vectorized IEEE benchmarks.

4.2.4 Conclusion

Through our successive modifications to the CADNA library, we have improved the scalar performance significantly, reducing its overhead by up to 85%. We have also enabled vectorization with the SPMD-on-SIMD programming paradigm, leading to an additional speedup between 2.5 and 3. With vectorization enabled, we make numerical validation possible for a wider variety of architectures and codes used in HPC.

Regarding many-core architectures, we could unfortunately not test this new CADNA version on the Xeon Phi processors due to compiler restrictions at the time of this work. Depending on the new compiler versions, we could in the future study its impact on this architecture whose SIMD units are wider and improved at the hardware level (e.g. execution masks for SIMD divergence, and scatter and gather operations).

As far as GPUs are concerned, since GPUs also (partially, but heavily) rely on SIMD execution, we have been interested in the impact of some of these improvements on the CADNA-GPU prototype presented in [84], which enables one to estimate rounding errors in CUDA codes. As shown in [ELBFJ18], the rounding mode emulation, which was here clearly beneficial on CPU, did not improve performance on GPU: this is likely due to the GPU predication mechanism and to the extra register consumption. Nevertheless, we have significantly reduced the overhead of CADNA by improving the random generator and by accessing this new generator twice less often. Through these modifications, we have reduced the computation time of single precision stochastic arithmetic on GPU by 39% to 61% (depending on the benchmark). We have also extended the CADNA-GPU library to double precision, with an even lower overhead than in single precision.

Regarding OpenCL, which generalizes the SPMD-on-SIMD programming model to various HPC architectures (GPUs, CPUs ...), C++ object-oriented features are supported since OpenCL 2.2, but unfortunately current OpenCL versions lack the minimal support for rounding mode changes that would enable to integrate CADNA in OpenCL codes.

Finally, we have also enabled the support of OpenMP codes by CADNA in order to estimate the round-off error propagations in multi-threaded OpenMP codes [EBFJ16]. Since there is no guarantee in the OpenMP standard that the worker threads have the same rounding mode as their master thread, we had first to design a compatibility check for OpenMP implementations (GOMP and Intel implementations have found to be compatible with CADNA). We have then used a distinct random generator (the same as in Sect. 4.2.2) in each OpenMP thread, and OpenMP *atomic* constructs were required for the safe updates of internal CADNA counters. We have also been able to support OpenMP reductions with CADNA in the user code (thanks to OpenMP 4.0 features), but not atomic operations (which have to be replaced by OpenMP critical sections in the user code). All this has resulted in similar or lower CADNA performance overheads for OpenMP codes than for serial codes (except if numerous atomic operations are used in the user code). In the end, this CADNA version detects numerical instabilities in OpenMP codes and estimates the number of exact significant digits of the results. For another parallel execution, or for the execution of the serial code, this number of exact significant digits can vary when the order of the floating-point operations varies, but we can still rely on the values of the exact significant digits given by CADNA.

Conclusion and future work

Conclusion

In this manuscript, we have presented our algorithmic contributions to scientific computing applications on high performance architectures.

After a brief introduction to the technical HPC aspects used in our work, we have first summarized our research results since our PhD thesis according to the following three directions: (i) designing algorithms for many-core or multi-core architectures, or for both via hybrid algorithms; (ii) handling the SIMD divergence; and (iii) taking advantage of new heterogeneous architectures for scientific applications.

Then, we have chosen to detail the two first research directions in order to highlight our algorithmic contributions, their interdisciplinary context, and the close combination they require between application specificities, algorithmics, programming and architectural features.

Regarding the design of algorithms for many-core or multi-core architectures, or both, we have first presented how we have modified the algorithm of a birth and death process for cell nuclei extraction in histopathology images. This has enabled us to obtain massive parallelism for all steps of the algorithm which has then be efficiently deployed on GPUs. Then we have strongly rewritten, based on task parallelism, the reference sequential algorithm building merge and contour trees in scientific visualization. This has led to important performance gains both in serial and in parallel on multi-core CPUs. Besides, the interest of hybrid CPU-GPU algorithms has been illustrated on integrated GPUs with a recursive fast multipole method in astrophysics. This hybrid algorithm benefits indeed from the efficient recursive tree traversal on the CPU cores, and from the compute power of the integrated GPU cores, along with their power and cost efficiencies, for the most compute-intensive operations.

Regarding the SIMD divergence handling, we have first revisited in computer arithmetics the main algorithm of the Table Maker's Dilemma solving. This rewriting has strongly reduced the execution flow divergence which has resulted in important performance gains on GPU SIMD units. Finally, the previous versions of the CADNA library, which estimates round-off error propagations in numerical codes, could not support CPU SIMD codes, since SIMD lanes cannot have different rounding modes. Partly thanks to a rounding mode emulation, we have here managed to enable the numerical validation of CPU SIMD codes, and also to significantly reduce the scalar overhead of CADNA: this makes CADNA suitable for HPC.

Future work

In addition to the extensions presented in Sects. 3.1.4, 3.2.5, 3.3.6, 4.1.4 and 4.2.4, we plan in the forthcoming years to continue providing algorithmic contributions to specific or key applications in scientific high performance computing, following these three research directions.

Hybrid algorithms for heterogeneous architectures

As detailed in Sect. 3.3, we have recently shown the interest of a hybrid algorithm for the deployment of a recursive fast multipole method (FMM), specific to astrophysical simulations, on a heterogeneous architecture composed of a multi-core CPU and of an integrated GPU. We plan to further study the impact of such hybrid algorithms, especially this FMM, on other heterogeneous architectures which should be relevant for such algorithms: multi-core CPU with a discrete GPU equipped with an NVIDIA NVLink bus offering much higher bandwidth than the traditional PCI Express bus, multi-core CPU with integrated FPGAs (now programmable with OpenCL [122]) ...

We also plan to consider for FMMs other application domains than astrophysics. As part of the PhD thesis of I. Chollet (co-supervised with X. Claeys and L. Grigori, 2017-), we have started to study the relevance of this recursive FMM in the context of boundary integral equation methods for wave propagation problems (Helmholtz kernel in low and high frequency regimes, with non-uniform distributions of particles). We are also investigating the benefit of such FMM in the context of polarizable molecular dynamics within the Tinker-HP[8] software [90] developed at Sorbonne Université (LCT laboratory, *Laboratoire de Chimie Théorique*). These molecular dynamics simulations offering uniform distributions of particles, one challenge here will consist in taking advantage of these uniform distributions (leading to regular computations) within such recursive FMM. We will rely in this purpose on our previous work for FMMs regarding SIMD processing (see Sect. 3.3) or regarding our matrix formulation for efficient processing with level 3 BLAS routines (see Sect. 2.3.1).

HPC and symbolic computing

So far, almost all our previous work has focused on numerical applications. As part of a secondment with CNRS in the CFHP[9] team at CRIStAL[10] - Université de Lille (2017-2018), we are currently investigating the impact of HPC on symbolic computing applications. Some progress has already been achieved on such topic these past years, but numerous algorithms and implementations used in symbolic computing are still currently designed in sequential mode. This has in particular led to the MEA4SRNC[11] proposal, in which we are involved, and which aims at revisiting symbolic computing algorithms in a HPC context.

During this secondment we have started to work in differential algebra on the Rosenfeld-Gröbner algorithm [17] implemented by F. Boulier in the BLAD[12] software. Similarly to

[8] See: http://tinker-hp.ip2ct.upmc.fr/

[9] Computer Algebra and High Performance Computing, see: http://www.cristal.univ-lille.fr/CFHP/

[10] See: https://www.cristal.univ-lille.fr/

[11] *Modern efficient algorithms for symbolic and reliable numeric computing*, principal investigator: G. Lecerf, submitted to the ANR 2018 generic call for proposals (ANR is the french National Research Agency).

[12] See: http://cristal.univ-lille.fr/~boulier/pmwiki/pmwiki.php/Main/BLAD

previous work in scientific visualization (see Sect. 3.2) and on the fast multipole method (see Sect. 3.3.1), task parallelism has turned out to be very suitable to deploy this algorithm on multi-core CPUs. Indeed such parallelism naturally matches the underlying computation tree that drives this algorithm, and this could moreover save computations with respect to the sequential execution. This also enables us to exploit the nested levels of parallelism available in this algorithm, each level offering a limited parallelism degree, on current HPC computers with a few dozen of cores. A performance bottleneck could also lie in the GCD (greatest common divisor) computation of multi-variate polynomials: this problem could be tackled via algorithms specific to our differential algebra context (leading to a large number of variables with low degrees), as well as via low-level algorithmics for SIMD computing.

It can also be noticed that a coupling between fast multipole methods and differential algebra has already been developed [163]: this may be further investigated, especially in a HPC context.

More generally, an interesting feature of some symbolic computations is their ability to (partly) check the correctness of the exact result. In GCD computations for example, one can easily verify that the resulting polynomial divides indeed the two input polynomials. This could open the way to fault-tolerant algorithms in symbolic computing.

Finally, due to the very high cost of some symbolic algorithms (e.g. exponential or double-exponential) and to operation count worst cases, a small increase in the input sizes or a slight difference in the input structure can lead to much greater computation times. Such computations could thus benefit from different execution environments, such as cloud computing which can offer elastic compute resources.

Taking advantage of new architectures

In addition to new architectures or new hardware features recently introduced in HPC (multi-core CPUs with integrated FPGAs, AVX-512 SIMD units ...), the convergence of HPC and Big Data has driven some of the main architectural developments over the latest years. The rise of deep learning in artificial intelligence has for example led to the introduction of "tensor cores" in the newest NVIDIA GPU architecture (Volta [3]), of "tensor processing units" (TPUs) by Google, and of new Intel Xeon Phi processors (Knights Mill) and forthcoming Intel Nervana chips, both specialized in deep learning. These architectures offer indeed increased compute power which could benefit to scientific computations other than deep learning (e.g. fast multipole methods). However this increased compute power applies to lower floating-point precisions, which will likely raises issues regarding performance-precision trade-offs. More generally, we plan to consider in the medium term algorithmic contributions required to adapt at best numerical applications in scientific computing to these new high performance architectures, considering performance, performance portability or power efficiency.

In the longer term, the introduction of memristor-based memories and of photonic interconnection networks between compute units and off-chip memories, as well as the foreseen end of Moore's law and the birth of quantum computers, will heavily impact HPC architectures and will require further strong algorithmic contributions to numerous applications in scientific computing.

Appendix A

Publications

Our publications in international journals and in international peer-reviewed conferences with proceedings are listed here. Other publications (abstract-only conference communications, research reports) are listed in: http://lip6.fr/Pierre.Fortin/CV_Fortin.pdf

Publications issued from our PhD thesis are indicated by *.

Journal articles

2018 [GFJTxx] C. Gueunet, P. Fortin, J. Jomier and J. Tierny, *Task-based Augmented Contour Trees with Fibonacci Heaps*, IEEE Transactions on Parallel and Distributed Systems (submitted)

2018 [FTxx] P. Fortin and M. Touche, *Dual tree traversal on integrated GPUs for astrophysical N-body simulations*, International Journal of High Performance Computing Applications (submitted)

2017 [SFLC18] I. Said, P. Fortin, J.-L. Lamotte and H. Calandra, *Leveraging the Accelerated Processing Units for seismic imaging: a performance and power efficiency comparison against CPUs and GPUs*, International Journal of High Performance Computing Applications (to appear)

2016 [FGG16] P. Fortin, M. Gouicem and S. Graillat, *GPU-Accelerated Generation of Correctly Rounded Elementary Functions*, ACM TOMS (Transactions on Mathematical Software), Vol. 43(3)

- [AFGZ16] C. Avenel, P. Fortin, M. Gouicem and S. Zaidi, *Solving the Table Maker's Dilemma on Current SIMD Architectures*, Scalable Computing: Practice and Experience, Vol. 17(3)

2015 [EBFJ15] P. Eberhart, J. Brajard, P. Fortin and F. Jézéquel, *High Performance Numerical Validation using Stochastic Arithmetic*, Reliable Computing, Vol. 21, pp 35-52

2013 [FL13] P. Fortin and J.-L. Lamotte, *An (almost) direct deployment of the Fast Multipole Method on the Cell processor*, The Journal of Supercomputing, Vol. 65, Issue 3, pp 1205-1222

2011 [FAL11] P. Fortin, E. Athanassoula, and J.-C. Lambert, *Comparisons of different codes for galactic N-body simulations*, Astronomy & Astrophysics, 531, A120

2010 * [CFR10] O. Coulaud, P. Fortin and J. Roman, *High performance BLAS formulation of the adaptive Fast Multipole Method*, Mathematical and Computer Modelling, Vol. 51/3-4, pp 177-188

2008 * [CFR08] O. Coulaud, P. Fortin and J. Roman, *High performance BLAS formulation of the multipole-to-local operator in the fast multipole method*, Journal of Computational Physics, Vol. 227/3, pp 1836-1862

Book chapters

2013 [HFJ+13] R. Habel, P. Fortin, F. Jézéquel, J.-L. Lamotte and N.S. Scott, *Numerical validation and performance optimization on GPUs of an application in atomic physics*, In Designing Scientific Applications on GPUs, R. Couturier Ed., Chapman & Hall/CRC

Conference papers, peer-reviewed with proceedings

2018 [ELB+18] P. Eberhart, B. Landreau, J. Brajard, P. Fortin and F. Jézéquel, *Improving CADNA performance on GPUs*, 19th IEEE Int. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-18), IPDPS Workshops

2017 [GFJT17] C. Gueunet, P. Fortin, J. Jomier and J. Tierny, *Task-based Augmented Merge Trees with Fibonacci Heaps*, IEEE Symposium on Large Data Analysis and Visualization (LDAV 2017)

2016 [EBFJ16] P. Eberhart, J. Brajard, P. Fortin and F. Jézéquel, *Estimation of round-off errors in OpenMP codes*, 12th International Workshop on OpenMP (IWOMP 2016)

- [GFJT16] C. Gueunet, P. Fortin, J. Jomier and J. Tierny, *Contour Forests: Fast Multi-threaded Augmented Contour Trees*, IEEE Symposium on Large Data Analysis and Visualization (LDAV 2016)

- [SFLC16] I. Said, P. Fortin, J.-L. Lamotte and H. Calandra, *hiCL: An OpenCL Abstraction Layer for Scientific Computing, Application to Depth Imaging on GPU and APU*, Int. Workshop on OpenCL (IWOCL 2016)

- [SFL+16] I. Said, P. Fortin, J.-L. Lamotte, R. Dolbeau and H. Calandra, *On the efficiency of the Accelerated Processing Unit for scientific computing*, 24th High Performance Computing Symposium (HPC 2016, *Best Paper Runner-Up award*)

2014 [LF14] B. Lange and P. Fortin, *Parallel dual tree traversal on multi-core and many-core architectures for astrophysical N-body simulations*, Euro-Par 2014

- [ESFC14] P. Eberhart, I. Said, P. Fortin and H. Calandra, *Hybrid strategy for stencil computations on the APU*, 1st Int. Workshop on High-Performance Stencil Computations (HiStencils 2014)

2013 [AFB13] C. Avenel, P. Fortin and D. Béréziat, *Parallel birth and death process for cell nuclei extraction in histopathology images*, 42nd Int. Conference on Parallel Processing (ICPP 2013)

- [CDF+13] H. Calandra, R. Dolbeau, P. Fortin, J.-L. Lamotte and I. Said, *Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil*, Special session "GPU Computing and Hybrid Computing", 21st Euromicro Int. Conference on Parallel, Distributed and Network-Based Processing (PDP 2013)

2012 [NBFT12] L. Nardi, F. Badran, P. Fortin and S. Thiria, *YAO: a generator of parallel code for variational data assimilation applications*, 14th IEEE Int. Conference on High Performance Computing and Communication (HPCC-2012)

- [FGG12] P. Fortin, M. Gouicem and S. Graillat, *Towards solving the Table Maker's Dilemma on GPU*, Special session "GPU Computing and Hybrid Computing", 20th Euromicro Int. Conference on Parallel, Distributed and Network-Based Computing (PDP 2012)

2011 [FHJ+11] P. Fortin, R. Habel, F. Jézéquel, J.-L. Lamotte and N.S. Scott, *Deployment on GPUs of an application in computational atomic physics*, 12th IEEE Int. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-11), IPDPS Workshops

2010 [BFL10] Q. Bourgerie, P. Fortin and J.-L. Lamotte, *Efficient Complex Matrix Multiplication on the Synergistic Processing Element of the Cell Processor*, Workshop on Parallel Programming and Applications on Accelerator Clusters (PPAAC10)

2009 [FL09] P. Fortin and J.-L. Lamotte, *Fast Multipole Method on the Cell Broadband Engine: the Near Field Part*, Parallel Computing: From Multicores and GPU's to Petascale, Selected Papers from the int. Parallel Computing Conference (ParCo2009), Advances in Parallel Computing, Vol. 19, pp 323-330, IOS Press

2007 * [CFR07] O. Coulaud, P. Fortin and J. Roman, *Hybrid MPI-thread parallelization of the Fast Multipole Method*, IEEE Int. Symposium on Parallel and Distributed Computing (ISPDC), pp 391-398

2005 * [CFR05] O. Coulaud, P. Fortin and J. Roman, *High-performance BLAS formulation of the Adaptive Fast Multipole Method*, Advances in Computational Methods in Sciences and Engineering 2005, Selected Papers from the Int. Conference of Computational Methods in Sciences and Engineering (ICCMSE), Vol. 4B, pp 1796-1799, VSP/Brill

Appendix B

List of supervised students

We have supervised or co-supervised the following students.

Post-doctoral researchers:

- Benoit Lange, *Astrophysical N-body simulations on multi-core and many-core architectures*, 2012-2013
- Christophe Avenel (with D. Béréziat), *Selective cell nuclei detection from histopathological images and deployment on many-core architectures* (2012-2013), then *The table maker's dilemma on massively parallel SIMD architectures* (2013)
- Mounira Bachir (with J. Brajard and F. Jézéquel), *Automatic generation of parallel numerical codes for data assimilation*, 2011-2012

PhD students

- Igor Chollet (with X. Claeys and L. Grigori), *High performance solvers based on compression techniques with application to electromagnetics*, 2017-
- Charles Gueunet (with J. Jomier and J. Tierny), *In-situ topological data analysis*, 2016-
- Pacôme Eberhart (with J. Brajard and F. Jézéquel), *Automatic generation of efficient and reliable codes for data assimilation*, 2013-interrupted
- Issam Said (with J.L. Lamotte), *Contributions of hybrid architectures to depth imaging: a CPU, APU and GPU comparative study*, 2011-2015
- Mourad Gouicem (with J.C. Bajard and S. Graillat), *Conception and deployment of efficient algorithms for solving the table maker's dilemma on parallel architectures*, 2010-2013

Research engineers

- Maxime Touche, *Astrophysical simulations on hybrid CPU-GPU architecture*, 2015

Master 2 students (5- or 6-month internships)

- Charles Gueunet (with J. Tierny), *In-situ visualization for high performance computing*, 2015
- Maxime Touche, *Astrophysical simulations on hybrid CPU-GPU architecture*, 2014
- Pacôme Eberhart (with I. Said), *Finite difference stencils on hybrid CPU-GPU architecture*, 2013
- Rachid Habel (with F. Jézéquel), *Deployment on GPU of an application in atomic physics*, 2010

Master 1 students (2-month internships)

- Baptiste Landreau (with P. Eberhart, J. Brajard and F. Jézéquel), *Numerical validation on GPU and application to oceanography*, 2015
- Richard Dang (with I. Said), *High-level programming of hybrid CPU-GPU architectures*, 2014
- Samia Zaidi (with M. Gouicem), *Solving the table maker's dilemma on CPUs and on GPUs*, 2012
- Joachim Dehais (with S. Graillat), *Path following for pseudospectra computations on GPUs*, 2010
- Hertz Hendrix Emani Emani (with S. Graillat), *Pseudospectra computations on GPUs*, 2010
- Sethy Montan, *Time integration of a parallel code for astrophysical simulations*, 2009
- Thomas Bussière (with L. Perret), *GPU implementation of the MD6 hash function*, 2009

Licence 3 students (3-month internships)

- Ambroise Fleury (with F. Lemaire), *SIMD speedup for polynomial evaluations*, 2018

Bibliography

[1] *OpenACC specifications*. Available from https://www.openacc.org.

[2] *OpenMP specifications*. Available from http://www.openmp.org.

[3] *NVIDIA Tesla V100 GPU Architecture*, August 2017. WP-08608-001_v1.1.

[4] R. Abdelkhalek, H. Calandra, O. Coulaud, G. Latu, and J. Roman. Fast seismic modeling and reverse time migration on a graphics processing unit cluster. *Concurrency and Computation: Practice and Experience*, 24(7):739–750, 2012.

[5] Mustafa Abduljabbar, Mohammed Al Farhan, Rio Yokota, and David Keyes. Performance evaluation of computation and communication kernels of the fast multipole method on intel manycore architecture. In *Euro-Par 2017: Parallel Processing: 23rd International Conference on Parallel and Distributed Computing, Proceedings*, pages 553–564. 2017.

[6] G. Alefeld and J. Herzberger. *Introduction to interval analysis*. Academic Press, 1983.

[7] AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide, revision 2.7, November 2013.

[8] N. Arora, A. Shringarpure, and R.W. Vuduc. Direct n-body kernels for multicore platforms. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 379–387, 2009.

[9] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[10] Christophe Avenel and Maria S. Kulikova. Marked point processes with simple and complex shape objects for cell nuclei extraction from breast cancer H&E images. In *SPIE Medical Image*, 2012.

[11] A. J. Baddeley and M. N. M. van Lieshout. Object recognition using markov spatial processes. In *International Conference on Pattern Recognition (ICPR)*, pages 136–139, 1992.

[12] J. E. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[13] Jeroen Bédorf, Evghenii Gaburov, Michiko S. Fujii, Keigo Nitadori, Tomoaki Ishiyama, and Simon Portegies Zwart. 24.77 Pflops on a Gravitational Tree-code to Simulate

the Milky Way Galaxy with 18600 GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 54–65, 2014.

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

[15] François Bodin and Stéphane Bihan. Heterogeneous multicore parallel programming for graphics processing units. *Scientific Programming*, 17(4):325–336, 2009.

[16] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 101–113, 2008.

[17] François Boulier, Daniel Lazard, François Ollivier, and Michel Petitot. Computing representations for radicals of finitely generated differential ideals. *Applicable Algebra in Engineering, Communication and Computing*, 20(1):73–121, 2009.

[18] R. L. Boyell and H. Ruston. Hybrid techniques for real-time radar simulation. In *Proc. of the IEEE Fall Joint Computer Conference*.

[19] J. Brajard, P. Li, F. Jézéquel, H.-S. Benavidès, and S. Thiria. Numerical Validation of Data Assimilation Codes Generated by the YAO Software. In *SIAM Annual Meeting*, 2013.

[20] P.T. Bremer, G. Weber, J. Tierny, V. Pascucci, M. Day, and J. Bell. Interactive exploration and analysis of large scale simulations using topology-based data segmentation. *IEEE TVCG*, 2011.

[21] X. Bresson, P. Vandergheynst, and J.P. Thiran. A variational model for object segmentation using boundary information and shape prior driven by the Mumford-Shah functional. *International Journal of Computer Vision*, 68(2):145–162, 2006.

[22] Nicholas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *International Symposium on Computer Architecture (ISCA'12)*, pages 49–60, 2012.

[23] P. G. Burke, C. J. Noble, and M. P. Scott. R-matrix theory of electron scattering at intermediate energies. *Proceedings of the Royal Society of London A*, 410:287–310, 1987.

[24] V. M. Burke, C. J. Noble, V. Faro-Maza, A. Maniopoulou, and N. S. Scott. FARM_2DRMP: a version of FARM for use with 2DRMP. *Computer Physics Communications*, 180:2450–2451, 2009.

[25] Martin Burtscher and Keshav Pingali. An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm. *GPU computing Gems Emerald edition*, page 75, 2011.

[26] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. In *Symposium on Discrete Algorithms*, 2000.

[27] H. Carr, J. Snoeyink, and M. van de Panne. Simplifying flexible isosurfaces using local geometric measures. In *Proc. of IEEE VIS*, pages 497–504, 2004.

[28] H. Carr, G. H. Weber, C. M. Sewell, and J. P. Ahrens. Parallel peak pruning for scalable smp contour tree computation. In *Proc. of IEEE Large Data Analysis and Visualization*, 2016.

[29] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, GA, USA, April 2010.

[30] H. Cheng, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics*, 155:468–498, 1999.

[31] Y. Chiang, T. Lenz, X. Lu, and G. Rote. Simple and optimal output-sensitive construction of contour trees using monotone paths. *Computational Geometry Theory and Applications*, 2005.

[32] Jee Choi, Aparna Chandramowlishwaran, Kamesh Madduri, and Richard Vuduc. A CPU-GPU Hybrid Implementation and Model-Driven Scheduling of the Fast Multipole Method. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 64:64–64:71, 2014.

[33] Robert G. Clapp, F. Haohuan, and O. Lindtjorn. Selecting the right hardware for reverse time migration. *Society of Exploration Geophysicists*, 2010.

[34] Jonathan P. Coles and Michel Masella. The fast multipole method and point dipole moment polarizable force fields. *The Journal of Chemical Physics*, 142(2):024109, 2015.

[35] Sylvain Collange, Marc Daumas, and David Defour. Interval Arithmetic in CUDA. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, pages 99–107. Morgan Kaufmann, 2011.

[36] T.H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[37] Marius Cornea. IEEE 754-2008 Decimal Floating-Point for Intel Architecture Processors. In *Proceedings of the 19th IEEE Symposium on Computer Arithmetic*, pages 225–228, 2009.

[38] Florent de Dinechin, Jean-Michel Muller, Bogdan Pasca, and Alexandru Plesco. An FPGA architecture for solving the Table Maker's Dilemma. In *Proceedings of the* 22^{nd} *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 187–194, 2011.

[39] W. Dehnen. A hierarchical O(N) force calculation algorithm. *Journal of Computational Physics*, 179:27–42, 2002.

[40] Walter Dehnen. A fast multipole method for stellar dynamics. *Computational Astrophysics and Cosmology*, 1(1), 2014.

[41] C. Denis, P. de Oliveira Castro, and E. Petit. Verificarlo: checking floating point accuracy through Monte Carlo Arithmetic. In *23rd IEEE International Symposium on Computer Arithmetic (ARITH'23), Silicon Valley, USA*, July 2016.

[42] X. Descombes, F. Kruggel, C. Lacoste, M. Ortner, G. Perrin, and J. Zerubia. Marked point process in image analysis: from context to geometry. In *International Conference on Spatial Point Process Modelling and its Application (SPPA)*, 2004.

[43] Xavier Descombes, Robert Minlos, and Elena Zhizhina. Object extraction using a stochastic birth-and-death dynamics in continuum. Research Report RR-6135, INRIA, 2007.

[44] S. Dillard. libtourtre: A contour tree library. http://graphics.cs.ucdavis.edu/~sdillard/libtourtre/doc/html/, 2007.

[45] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[46] J. Dongarra and F. Sullivan. Guest editors' introduction: The top 10 algorithms. *Computing in Science and Engineering*, 2(1):22–23, 2000.

[47] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesùs Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[48] H. Edelsbrunner and J. Harer. *Computational Topology: An Introduction*. American Mathematical Society, 2009.

[49] G. De Fabritiis. Performance of the cell processor for biomolecular simulations. *Computer Physics Communications*, 176:660–664, 2007.

[50] G. Favelier, C. Gueunet, and J. Tierny. Visualizing ensembles of viscous fingers. In *IEEE SciVis Contest*, 2016.

[51] D. Foltinek, D. Eaton, J. Mahovsky, P. Moghaddam, and R. McGarry. Industrial-scale reverse time migration on GPU hardware. In *SEG Annual Meeting*, 2009.

[52] Michael Frechtling and Philip H. W. Leong. MCALIB: Measuring Sensitivity to Rounding Error with Monte Carlo Programming. *ACM Transactions on Programming Languages and Systems*, 37(2):1–25, 2015.

[53] Michael Fredman and Robert Tarjan. Fibonacci Heaps and their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 1987.

[54] Steffen Frey, Guido Reina, and Thomas Ertl. SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms. In *Proceedings of the 20th International Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 399–406, 2012.

[55] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Trans. Archit. Code Optim.*, 6(2):7:1–7:37, 2009.

[56] François Févotte and Bruno Lathuilière. VERROU: a CESTAC evaluation without recompilation. In *International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN)*, 2016.

[57] B. Gaster, L. Howes, D.R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition*. Newnes, 2012.

[58] Mourad Gouicem. *Conception et implantation d'algorithmes efficaces pour la résolution du dilemme du fabricant de tables sur architectures parallèles*. PhD thesis, Université Pierre et Marie Curie (UPMC), October 2013.

[59] D. Guenther, R. Alvarez-Boto, J. Contreras-Garcia, J.-P. Piquemal, and J. Tierny. Characterizing molecular interactions in chemical systems. *IEEE Trans. on Vis. and Comp. Graph.*, 2014.

[60] N. A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *Journal of Computational Physics*, 227:8290–8313, 2008.

[61] A. Gyulassy, P.-T. Bremer, B. Hamann, and P. Pascucci. A practical approach to Morse-Smale complex computation: scalabity and generality. *IEEE Trans. on Vis. and Comp. Graph.*, pages 1619–1626, 2008.

[62] A. Gyulassy, P.T. Bremer, R. Grout, H. Kolla, J. Chen, and V. Pascucci. Stability of dissipation elements: A case study in combustion. *Comp. Graph. For.*, 2014.

[63] A. Gyulassy, A. Knoll, K.C. Lau, B. Wang, P.T. Bremer, M.E. Papka, L. A. Curtiss, and V. Pascucci. Interstitial and interlayer ion diffusion geometry extraction in graphitic nanosphere battery materials. *IEEE Trans. on Vis. and Comp. Graph.*, 2015.

[64] Attila Gyulassy, Vijay Natarajan, Mark Duchaineau, Valerio Pascucci, Eduardo Bringa, Andrew Higginbotham, and Bernd Hamann. Topologically Clean Distance Fields. *IEEE Trans. on Vis. and Comp. Graph.*, 13:1432–1439, 2007.

[65] A. Hafiane, F. Bunyak, and K. Palaniappan. Evaluation of level set-based histology image segmentation using geometric region criteria. In *International Conference on Symposium on Biomedical Imaging (ISBI)*, 2009.

[66] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC'09, pages 62:1–62:12, 2009.

[67] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 3:1–3:8, 2011.

[68] C. Heine, H. Leitte, M. Hlawitschka, F. Iuricich, L. De Floriani, G. Scheuermann, H. Hagen, and C. Garth. A survey of topology-based methods in visualization. *Comp. Graph. For.*, 2016.

[69] J.L. Hennessy and D.A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design, 2011.

[70] N.J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics (SIAM), second edition, 2002.

[71] Masaki Hilaga, Yoshihisa Shinagawa, Taku Kohmura, and Tosiyasu L. Kunii. Topology matching for fully automatic similarity estimation of 3D shapes. In *Proc. of ACM SIGGRAPH*, 2001.

[72] Qi Hu, Nail A. Gumerov, and Ramani Duraiswami. Scalable fast multipole methods on distributed heterogeneous architectures. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 36:1–36:12, 2011.

[73] Huda Ibeid, Rio Yokota, Jennifer Pestana, and David Keyes. Fast multipole preconditioners for sparse matrices arising from elliptic equations. *Computing and Visualization in Science*, 2017.

[74] IBM. *Basic Linear Algebra Subprograms Library Programmer's Guide and API Reference, Software Development Kit for Multicore Acceleration, Version 3.1*, 2008.

[75] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic, 2008.

[76] Intel. A Guide to Vectorization with Intel C++ Compilers, 2012.

[77] Intel. Intel Architecture Instruction Set Extensions Programming Reference, Number: 319433-012A, 2012.

[78] Intel. *The Compute Architecture of Intel Processor Graphics Gen8*, 2015. Version 1.1.

[79] Intel. Intel Architecture Instruction Set Extensions Programming Reference, Number: 319433-024, 2016.

[80] F. Jézéquel, J.-L. Lamotte, and O. Chubach. Parallelization of Discrete Stochastic Arithmetic on multicore architectures. In *10th International Conference on Information Technology: New Generations (ITNG)*, pages 160–166, 2013.

[81] F. Jézéquel, F. Rico, J.-M. Chesneaux, and M. Charikhi. Reliable computation of a multiple integral involved in the neutron star theory. *Mathematics and Computers in Simulation*, 71(1):44–61, 2006.

[82] H. Jin, M. A. Frumkin, and J. Yan. Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes. In *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing*, pages 440–456, 2000.

[83] F. Jézéquel and J.-M. Chesneaux. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933–955, 2008.

[84] F. Jézéquel, J.-L. Lamotte, and I. Saïd. Estimation of numerical reproducibility on CPU and GPU. In M. Ganzha, L. Maciaszek, and M. Paprzycki, editors, *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems*, volume 5 of *Annals of Computer Science and Information Systems*, pages 675–680. IEEE, 2015.

[85] Adam Karlsson, Kent Stråhlén, and Anders Heyden. Segmentation of histopathological section using snakes. In *Proceedings of the 13th Scandinavian conference on Image analysis (SCIA)*, pages 595–602, 2003.

[86] J. Kasten, J. Reininghaus, I. Hotz, and H.C. Hege. Two-dimensional time-dependent vortex regions based on the acceleration magnitude. *IEEE Trans. on Vis. and Comp. Graph.*, 2011.

[87] T.J. Knight, J.Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W.J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 226–236, 2007.

[88] Maria S. Kulikova, Antoine Veillard, Ludovic Roux, and Daniel Racoceanu. Nuclei extraction from histopathological images using a marked point process approach. In *Proc. SPIE Medical Imaging*, 2012.

[89] U.W. Kulisch. *Advanced Arithmetic for the Digital Computer*. Springer-Verlag, 2002.

[90] Louis Lagardere, Luc-Henri Jolly, Filippo Lipparini, Felix Aviat, Benjamin Stamm, Zhifeng F. Jing, Matthew Harger, Hedieh Torabifard, G. Andres Cisneros, Michael J. Schnieders, Nohad Gresh, Yvon Maday, Pengyu Y. Ren, Jay W. Ponder, and Jean-Philip Piquemal. Tinker-hp: a massively parallel molecular dynamics package for multiscale simulations of large complex systems with advanced point dipole polarizable force fields. *Chemical Science*, 9:956–972, 2018.

[91] J.-L. Lamotte, J.-M. Chesneaux, and F. Jézéquel. CADNA_C: A version of CADNA for use with C or C++ programs. *Computer Physics Communications*, 181(11):1925–1926, 2010.

[92] D. E. Laney, P.T. Bremer, A. Mascarenhas, P. Miller, and V. Pascucci. Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE Trans. on Vis. and Comp. Graph.*, 2006.

[93] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 58:1–58:12, 2009.

[94] Vincent Lefèvre. New Results on the Distance Between a Segment and $\mathbb{Z}^2$. Application to the exact Rounding. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 68–75, 2005.

[95] Vincent Lefèvre, Jean-Michel Muller, and Arnaud Tisserand. Toward correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, 1998.

[96] Hong-Wei LIU, Bo LI, Hong LIU, Xiao-Long TONG, and Qin LIU. The Algorithm of High Order Finite Difference Pre-Stack Reverse Time Migration and GPU Implementation. *Chinese Journal of Geophysics*, 53(4):600–610, 2010.

[97] Konstantin Lorenzen, Magnus Schwörer, Philipp Tröster, Simon Mates, and Paul Tavan. Optimizing the Accuracy and Efficiency of Fast Hierarchical Multipole Expansions for MD Simulations. *Journal of Chemical Theory and Computation*, 8(10):3628–3636, 2012.

[98] E. Luttmann, D. Ensign, V. Vaidyanathan, M. Houston, N. Rimon, J. Øland, G. Jayachandran, M. Friedrichs, and V. Pande. Accelerating molecular dynamic simulation on the cell processor and playstation 3. *Journal of Computational Chemistry*, 30(2):268–274, 2009.

[99] S. Maadasamy, H. Doraiswamy, and V. Natarajan. A hybrid parallel algorithm for computing and tracking level set topology. In *International Conference on High Performance Computing*, 2012.

[100] MAGMA (Matrix Algebra on GPU and Multicore Architectures). Available at: http://icl.cs.utk.edu/magma.

[101] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 235–246, 2010.

[102] David Michéa and Dimitri Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophysical Journal International*, 182(1):389–402, 2010.

[103] John Milnor. *Morse Theory*. Princeton U. Press, 1963.

[104] S. Mohanty, A. K. Mohanty, and F. Carminati. Efficient pseudo-random number generation for monte-carlo simulations using graphic processors. *Journal of Physics: Conference Series*, 368(1), 2012.

[105] S. Montan, J.-M. Chesneaux, C. Denis, and J.-L. Lamotte. Towards an efficient implementation of CADNA in the BLAS : Example of DgemmCADNA routine. In *15th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN)*, 2012.

[106] S. Montan and C. Denis. Numerical verification of industrial numerical codes. In *ESAIM: Proc.*, volume 35, pages 107–113, 2012.

[107] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-point Arithmetic*. Birkhauser, 2009.

[108] A. Narang, S. Kumar, A.S. Das, M. Perrone, D. Wade, K. Bendiksen, V. Slåtten, and T.E. Rabben. Performance optimizations for TTI RTM on GPU based hybrid architectures. *10th Biennial International Conference & Exposition*, 2013.

[109] L. Nardi, C. Sorror, F. Badran, and S. Thiria. YAO: A Software for Variational Data Assimilation Using Numerical Models. In *LNCS 5593, Computational Science and Its Applications - ICCSA 2009*, pages 621–636, 2009.

[110] *NVIDIA CUDA Toolkit* 4.1, CUBLAS Library, January 2012.

[111] Lars Nyland, Mark Harris, and Jan Prins. Fast N-Body Simulation with CUDA. *GPU gems*, 3:677–695, 2007.

[112] T. Okamoto, H. Takenaka, T. Nakamura, and T. Aoki. Accelerating large-scale simulation of seismic wave propagation by multi-GPUs and three-dimensional domain decomposition. In *GPU Solutions to Multi-scale Problems in Science and Engineering*, pages 375–389. Springer, 2013.

[113] R. E. Overman, J. F. Prins, L. A. Miller, and M. L. Minion. Dynamic Load Balancing of the Adaptive Fast Multipole Method in Heterogeneous Systems. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 1126–1135, 2013.

[114] J. Panetta, T. Teixeira, P.R.P. de Souza Filho, C.A. da Cunha Finho, D. Sotelo, F. da Motta, S.S. Pinheiro, I. Pedrosa, A.L.R. Rosa, L.R. Monnerat, L.T. Carneiro, and C.H.B. de Albrecht. Accelerating Kirchhoff migration by CPU and GPU cooperation. In *Int. Symp. on Computer Architecture and High Performance Computing, SBAC-PAD '09*, pages 26–32, 2009.

[115] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 2003.

[116] V Pascucci, K Cole-McLaughlin, and G Scorzelli. Multi-resolution computation and presentation of contour trees. 2004.

[117] V Pascucci, G Scorzelli, P T Bremer, and A Mascarenhas. Robust on-line computation of Reeb graphs: simplicity and speed. *ACM Trans. on Graph.*, 2007.

[118] V. Pascucci, X. Tricoche, H. Hagen, and J. Tierny. *Topological Data Analysis and Visualization: Theory, Algorithms and Applications*. Springer, 2010.

[119] M. Pharr and W.R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13, May 2012. See also: https://ispc.github.io/.

[120] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010.

[121] Georges Reeb. Sur les points singuliers d'une forme de Pfaff complètement intégrable ou d'une fonction numérique. *Comptes-rendus de l'Académie des Sciences*, 222:847–849, 1946.

[122] James Reinders and Tom Hill. FPGA Programming with OpenCL. *The Parallel Universe*, 31, 2018. Intel Software.

[123] N. Revol and P. Théveny. Parallel implementation of interval matrix multiplication. *Reliable Computing*, 19(1):91–106, 2013.

[124] Paul Rosen, Bei Wang, Anil Seth, Betsy Mills, Adam Ginsburg, Julia Kamenetzky, Jeff Kern, and Chris R. Johnson. Using contour trees in the analysis and visualization of radio astronomy data cubes. Technical report, University of South Florida, 2017.

[125] Nadav Rotem. Intel OpenCL Implicit Vectorization Module, 2011 LLVM Developers' Meeting, 2011.

[126] Issam Said. *Contributions of hybrid architectures to depth imaging: a CPU, APU and GPU comparative study*. Ph.D. Thesis, Université Pierre et Marie Curie (UPMC), December 2015.

[127] Thomas Schaub, Simon Moll, Ralf Karrenberg, and Sebastian Hack. The Impact of the SIMD Width on Control-Flow and Memory Divergence. *ACM Trans. Archit. Code Optim.*, 11(4):54:1–54:25, 2015.

[128] N. S. Scott, M. P. Scott, P. G. Burke, T. Stitt, V. Faro-Maza, C. Denis, and A. Maniopoulou. 2DRMP: A suite of two-dimensional R-matrix propagation codes. *Computer Physics Communications*, 180:2424–2449, 2009.

[129] N.S. Scott, F. Jézéquel, C. Denis, and J.-M. Chesneaux. Numerical 'health check' for scientific codes: the CADNA approach. *Computer Physics Communications*, 176(8):507–521, 2007.

[130] Nithin Shivashankar, Pratyush Pranav, Vijay Natarajan, Rien van de Weygaert, EG Patrick Bos, and Steven Rieder. Felix: A topology based framework for visual exploration of cosmic filaments. *IEEE TVCG*, 2016.

[131] Noel Bryan Slater. The distribution of the integers n for which $\{\theta n\} < \phi$. *Proceedings of the Cambridge Philosophical Society*, 46:525–534, 1950.

[132] Dmitriy Smirnov and Dmitriy Morozov. Triplet Merge Trees. In *TopoInVis*, 2017.

[133] B. S. Sohn and C. L. Bajaj. Time varying contour topology. *IEEE Trans. on Vis. and Comp. Graph.*, 2006.

[134] T. Sousbie. The persistent cosmic web and its filamentary structure: Theory and implementations. *Royal Astronomical Society*, 2011.

[135] V. Springel. "the cosmological simulation code gadget-2". *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

[136] Damien Stehlé, Vincent Lefèvre, and Paul Zimmermann. Searching Worst Cases of a One-Variable Function Using Lattice Reduction. *IEEE Transactions on Computers*, 54:340–346, 2005.

[137] T. Stitt, N. S. Scott, M. P. Scott, and P. G. Burke. 2-D R-matrix propagation: a large scale electron scattering simulation dominated by the multiplication of dynamically changing matrices. In *Proc. of VECPAR'02*, pages 354–367, 2002.

[138] Sriram Swaminarayan, Kai Kadau, Timothy C. Germann, and Gordon C. Fossum. 369 tflop/s molecular dynamics simulations on the roadrunner general-purpose heterogeneous supercomputer. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[139] J. Taillard, F. Guyomarc'h, and J.-L. Dekeyser. A Graphical Framework for High Performance Computing Using An MDE Approach. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2008*, pages 165–173, 2008.

[140] T. Takahashi, C. Cecka, W. Fong, and E. Darve. Optimizing the multipole-to-local operator in the fast multipole method for graphical processing units. *International Journal for Numerical Methods in Engineering*, 89(1):105–133, 2012.

[141] S. Tarasov and M. Vyali. Construction of contour trees in 3d in o(n log n) steps. In *SoCG*, 1998.

[142] K. Taura, J. Nakashima, R. Yokota, and N. Maruyama. A Task Parallel Implementation of Fast Multipole Methods. In *SC Companion*, pages 617–625, 2012.

[143] S.-H. Teng. Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation. *SIAM Journal on Scientific Computing*, 19(2):635–656, 1998.

[144] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, 2018.

[145] D. M. Thomas and V. Natarajan. Multiscale symmetry detection in scalar fields by clustering contours. *IEEE TVCG*, 2014.

[146] Julien Tierny, Guillaume Favelier, Joshua A. Levine, Charles Gueunet, and Michael Michaux. The Topology ToolKit. *IEEE TVCG (Proc. of IEEE VIS)*, 2017. https://topology-tool-kit.github.io/.

[147] Shahar Timnat, Ohad Shacham, and Ayal Zaks. Predicate Vectors If You Must. In *Workshop on Programming Models for SIMD/Vector Processing, colocated with the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014.

[148] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[149] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pasucci, and D.R. Schikore. Contour trees and small seed sets for isosurface traversal. In *SoCG*, 1997.

[150] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[151] L. Vese and T. Chan. A multiphase level set framework for image segmentation using the Mumford and Shah model. *International Journal of Computer Vision*, 50(3):271–293, 2002.

[152] M. Veta, J. P. W. Pluim, P. J. van Diest, and M. A. Viergever. Breast cancer histopathology image analysis: A review. *IEEE Transactions on Biomedical Engineering*, 61(5):1400–1411, 2014.

[153] J. Vignes. Discrete stochastic arithmetic for validating results of numerical software. *Numerical Algorithms*, 37(1-4):377–390, 2004.

[154] M. S. Warren and J. K. Salmon. A portable parallel particle program. *Computer Physics Communications*, 87, 1995.

[155] G. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann. Topology-controlled volume rendering. *IEEE TVCG*, 2007.

[156] Gunther H. Weber, Peer-Timo Bremer, and Valerio Pascucci. Topological Landscapes: A Terrain Metaphor for Scientific Data. *IEEE TVCG*, 2007.

[157] J. H. Wilkinson. *Rounding errors in algebraic processes*, volume 32. HMSO, London, 1963.

[158] Y. Xu, J.-Y. Zhu, E. Chao, and Z. Tu. Multiple clustered instance learning for histopathology cancer image classification, segmentation and clustering. In *Computeur Vision and Pattern Recognition (CVPR)*, pages 964–971, 2012.

[159] R. Yokota. An FMM Based on Dual Tree Traversal for Many-Core Architectures. *Journal of Algorithms & Computational Technology*, 7(3):301–324, 2013.

[160] R. Yokota, J.P. Bardhan, M.G. Knepley, L.A. Barba, and T. Hamada. Biomolecular electrostatics using a fast multipole bem on up to 512 gpus and a billion unknowns. *Computer Physics Communications*, 182(6):1272–1283, 2011.

[161] Rio Yokota and Lorena A. Barba. Chapter 9 - Treecode and Fast Multipole Method for N-Body Simulation with CUDA. In *GPU Computing Gems Emerald Edition*, pages 113 – 132. Morgan Kaufmann, 2011.

[162] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining GPU Applications on the Fly: Thread Divergence Elimination Through Runtime Thread-data Remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 115–126, 2010.

[163] He Zhang and Martin Berz. The fast multipole method in the differential algebra framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 645(1):338 – 344, 2011.

Abstract

High performance architectures are constantly evolving in order to deliver ever greater compute powers, as well as ever greater energy efficiencies. This applies to multi-core CPUs (with higher core count and wider vector units) as well as to various many-core, possibly heterogeneous, architectures (GPUs, Xeon Phi processors ...). Considering performance, power efficiency or performance portability, and relying on new and relevant programming paradigms, we have focused on algorithmic changes allowing to adapt at best specific or key applications in scientific computing to such high performance architectures.

Our research work has been structured according to three research directions: (i) designing algorithms for many-core architectures via massive parallelism, for multi-core architectures via task parallelism, or for both via hybrid algorithms; (ii) handling the vector divergence on high performance architectures; and (iii) taking advantage of new heterogeneous architectures for scientific applications. We present here our algorithmic contributions, their interdisciplinary context, and the close combination they require between application specificities, algorithmics, programming and architectural features.

Résumé

Les architectures de calcul haute performance évoluent en permanence afin d'offrir des capacités de calcul, et des efficacités énergétiques, toujours plus importantes. Ceci concerne d'une part les CPU multi-cœurs, qui comportent de plus en plus de cœurs et des unités vectorielles toujours plus grandes, et d'autre part les diverses architectures *many-core*, potentiellement hétérogènes (GPU, processeurs Xeon Phi ...). En prenant en compte la performance, l'efficacité énergétique ou la portabilité des performances, et en nous appuyant sur des paradigmes de programmation récents et appropriés, nous avons apporté des contributions algorithmiques permettant d'adapter au mieux des applications de référence, ou spécifiques, en calcul scientifique à ces architectures haute performance.

Nos travaux ont été structurés selon les trois axes de recherche suivants : (i) concevoir des algorithmes pour les architectures *many-core* via du parallélisme massif, pour les architectures multi-cœurs via du parallélisme de tâches, ou pour les deux via des algorithmes hybrides ; (ii) réduire la divergence au sein des calculs vectoriels sur les architectures haute performance ; et (iii) tirer parti des nouvelles architectures hétérogènes pour des applications en calcul scientifique. Nous présentons ici nos contributions algorithmiques, leur contexte interdisciplinaire, ainsi que les fortes interactions qu'elles requièrent entre les caractéristiques de l'application, l'algorithmique, la programmation et l'architecture matérielle.