Università degli Studi di Siena
Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche

# PHAST Library – A productive solution to program heterogeneity

PhD Thesis

Author
**Biagio Peccerillo**

Advisor
**Ing. Sandro Bartolini**

Siena, May 2020

XXVII Cycle

*To those who have always been there,*
*To those for whom I'll always be there.*

"The way the processor industry is going,
is to add more and more cores,
but nobody knows how to program those things.
I mean, two, yeah; four, not really;
eight, forget it."

— Steve Jobs, *CEO, Apple Inc.*

"We stand at the threshold of a many core world.
The hardware community is ready to cross this threshold.
The parallel software community is not."

— Tim Mattson, *Senior Principal Engineer, Intel Corporation*

# Acknowledgements

I'd like to thank Professor Sandro Bartolini, who has always been close to me in this project, with enthusiasm and conviction. During these long years, he's been the best mentor and travel companion I could ever wish for.

I'd like to thank Maria Rita, who has always supported, incited, and motivated me, with love, serenity, and patience. Her contribution, sometimes invisible but always tangible, has been fundamental to the realization of this work.

I'd like to thank my parents and my sister, who have always believed in me since I was born and are still doing it today. Above all, they did it when I needed it most.

I'd like to thank my friends, who have contributed to give both levity and substance together.

I'd like to thank my son, who has brought infinite joys in my life. The many contributions he added to my thesis every time I recklessly left my computer unattended are invaluable.

I'd like to thank CTRL+Z, who has been able to take care of the aforementioned contributions.

# Ringraziamenti

Ringrazio il Professor Sandro Bartolini, che mi è sempre stato vicino in questo progetto, con entusiasmo e convinzione. In questi lunghi anni è stato il miglior mentore e il miglior compagno di viaggio che potessi desiderare.

Ringrazio Maria Rita, che mi ha sempre sostenuto, spronato e motivato, con amore, serenità e pazienza. Il suo apporto, a volte invisibile ma sempre tangibile, è stato fondamentale alla realizzazione di questo lavoro.

Ringrazio i miei genitori e mia sorella, che hanno creduto in me sin da quando sono nato e continuano a farlo ancora oggi. Soprattutto, lo hanno fatto nel momento in cui ne ho avuto più bisogno.

Ringrazio i miei amici, che hanno contribuito a dare leggerezza e spessore allo stesso tempo.

Ringrazio mio figlio, che ha portato infinite gioie nella mia vita. I numerosi contributi che ha aggiunto ogni volta che ho lasciato il computer incautamente incustodito restano preziosissimi.

Ringrazio CTRL+Z, che ha saputo prendersi cura dei suddetti contributi.

# Abstract

Nowadays, parallel architectures are everywhere: desktop computers, mobile devices, high-performance workstations, and servers are just a few examples. Not only multi-core microprocessors, but also GPUs, FPGAs, TPUs, and other computing devices are widely used in a broad range of applications. Programmers willing to code in this *post-PC era* need to take advantage of heterogeneity in order to fully utilize these devices and reach unprecedented performance.

Programming heterogeneity can be difficult: each device can have its own languages, frameworks, and coding styles. Heterogeneous approaches that let programmers code *once* must be preferred for productivity reasons. They must provide portability as a minimum requirement, but also *performance portability* to reduce the need of device-specific coding and tuning. Despite the wide diffusion of heterogeneous computing devices, frameworks with these facilities are yet to come.

This Ph.D. thesis presents my contribution to parallel and heterogeneous programming. After presenting an in-depth study of the state-of-the-art programming approaches for heterogeneous devices, it presents a programming framework that aims at embodying their strengths while limiting their weaknesses.

This framework, called PHAST Library, is a modern C++ single-source programming library that provides near-native performance, productivity, portability, and performance portability across parallel computing devices. This thesis discusses it in-depth and evaluates it against other productive heterogeneous frameworks from both performance and productivity points of view. The comparison is done in the case of various toy and real-world applications, which shows that the framework is mature. It also improves the state-of-the-art in a measurable way, and thus its use can help programmers to write high-level, high-performance parallel heterogeneous code. In conclusion, a possible evolution of the framework is presented.

# Sommario

Oggigiorno, le architetture parallele sono ovunque: computer desktop, dispositivi mobile, workstation ad alte prestazioni e server sono solo alcuni esempi. Oltre ai classici multi-core, anche GPU, FPGA, TPU e altri dispositivi vengono utilizzati in un ampio spettro di applicazioni. In questa *era post-PC*, i programmatori devono necessariamente sfruttare l'eterogeneità per utilizzare pienamente l'hardware e raggiungere prestazioni senza precedenti.

Scrivere codice eterogeneo può essere difficile: ogni dispositivo può avere i suoi linguaggi, i suoi framework e i suoi stili di programmazione. Pertanto, per essere produttivi, bisognerebbe preferire degli approcci eterogenei che consentano la scrittura di codice *una volta sola*. Questi approcci devono fornire come minimo la portabilità del codice tra dispositivi, ma anche la *portabilità delle prestazioni*, così da ridurre la necessità di programmare e ottimizzare diverse architetture in maniera specifica. Nonostante l'ampia diffusione di dispositivi eterogenei, tali framework sono ancora di là da venire.

Questa tesi di Dottorato presenta il mio contributo alla programmazione parallela ed eterogenea. In una prima parte viene discusso approfonditamente lo stato dell'arte dell'ambito di appartenenza. In seguito viene presentato un framework che mira a porsi come punto di riferimento, tentando di conciliare i punti forti delle diverse soluzioni concorrenti, limitandone le debolezze.

Questo framework, dal nome PHAST Library, è una libreria di programmazione single-source in C++ moderno che fornisce produttività e portabilità di codice e prestazioni tra dispositivi paralleli. In questo lavoro si discutono le caratteristiche in maniera approfondita e se ne valuta il valore tramite confronto con altri framework eterogenei ad alto livello di astrazione, sia da un punto di vista prestazionale che di produttività. Dal confronto, che annovera esempi semplici e applicazioni reali, emerge che il framework è maturo. Inoltre, PHAST Library rappresenta un miglioramento misurabile dello stato dell'arte e perciò può essere utilizzato per lo sviluppo di applicazioni parallele ed eterogenee in maniera produttiva e portabile. La tesi si conclude con la discussione delle sue possibili evoluzioni.

# List of publications

## International Journals

1. B. Peccerillo, and S. Bartolini, "PHAST - A Portable High-Level Modern C++ Programming Library for GPUs and Multi-Cores", IEEE Transactions on Parallel and Distributed Systems 30, 1 (Jan 2019), 2019, pp. 174-189.

2. B. Peccerillo, S. Bartolini, and Ç. K. Koç, "Parallel Bitsliced AES through PHAST: a Single-Source High-Performance Library for Multi-Cores and GPUs", Journal of Cryptographic Engineering, 2017.

## International Conferences/Workshops with Peer Review

1. B. Peccerillo and S. Bartolini, "Task-DAG Support in Single-Source PHAST Library: Enabling Flexible Assignment of Tasks to CPUs and GPUs in Heterogeneous Architectures", Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), Washington DC, 2019, pp. 91-100.

2. B. Peccerillo and S. Bartolini, "Single-source Library for Enabling Seamless Assignment of Data-parallel Task-DAGs to CPUs and GPUs in Heterogeneous Architectures", Proceedings of the 10th and 8th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM), Valencia, 2019, pp. 3:1-3:4.

3. G. Gioioso, M. Pozzi, M. Aurilio, B. Peccerillo, G. Spagnoletti, and D. Prattichizzo, "Using Wearable Haptics for Thermal Discrimination in Virtual Reality Scenarios". In: Kajimoto H., Lee D., Kim SY., Konyo M., Kyung KU. (eds) Haptic Interaction. AsiaHaptics 2018. Lecture Notes in Electrical Engineering, vol 535. Springer, Singapore, 2019.

4. B. Peccerillo and S. Bartolini, "PHAST Library – Enabling Single-Source and High Performance Code for GPUs and Multi-cores", 2017 International Conference on

High Performance Computing & Simulation (HPCS), Genova, 2017, pp. 715-718.

## Book Chapters

1. S. Bartolini and B. Peccerillo, "Parallel Programming in Cyber-Physical Systems". In: Ç. K. Koç (eds.) Cyber-Physical Systems Security, pp. 111-134, Springer International Publishing, Cham, 2018.

## Patents

1. Bartolini S., Peccerillo B., inventori; Università degli Studi di Siena, titolare. Procedimento per la Generazione Automatica di Codice di Calcolo Parallelo. Italia National patent IT 102017000082213. 2017 july 19.

# Index

**1 Introduction** **1**
1.1 The laws of technological progress . . . 1
1.1.1 Moore's law . . . 1
1.1.2 Dennard scaling . . . 2
1.2 The crisis of Dennard scaling . . . 3
1.3 The advent of multi-core CPUs . . . 4
1.3.1 Multiprocessor architecture . . . 5
1.3.2 Multi-core and performance scaling . . . 8
1.4 The advent of GPUs . . . 11
1.4.1 GPGPU . . . 11
1.4.2 GPU architecture . . . 13
1.5 Heterogeneous computing . . . 17

**2 Programming heterogeneity** **19**
2.1 Multi-core programming . . . 20
2.1.1 C++ threads . . . 21
2.1.2 C threads . . . 22
2.2 GPU programming . . . 23
2.2.1 CUDA . . . 23
2.2.2 ROCm . . . 26
2.3 Heterogeneous programming . . . 28
2.3.1 OpenCL . . . 29
2.3.2 OpenMP and OpenACC . . . 32
2.3.3 SYCL . . . 33
2.3.4 C++ AMP . . . 35
2.3.5 SkePU and SkelCL . . . 36
2.3.6 Kokkos . . . 36
2.3.7 Other approaches . . . 37
2.3.8 Critical discussion . . . 38

3 **PHAST Library** **41**
3.1 Design principles . . . 41
3.2 Containers . . . 45
3.3 Iterators . . . 49
3.4 Algorithms . . . 53
3.5 Functors . . . 57
3.6 Parallelization parameters . . . 59
3.6.1 Multi-core . . . 60
3.6.2 NVIDIA GPUs . . . 61
3.7 Hierarchical parallelism . . . 62
3.7.1 In-functor algorithms . . . 63
3.7.2 Synchronization . . . 66
3.7.3 Innovation and novelty . . . 67
3.8 Interoperability and low-level optimizations . . . 67
3.9 Task programming . . . 68
3.9.1 `task` and `stream_task` . . . 68
3.9.2 Task- and data-parallelism . . . 73
3.9.3 Truly heterogeneous task-DAGs . . . 75

4 **Evaluation** **79**
4.1 Simple examples . . . 79
4.1.1 Triad . . . 82
4.1.2 DCT8x8 . . . 87
4.1.3 Black and Scholes . . . 94
4.1.4 N-Body . . . 98
4.1.5 Summary . . . 101
4.2 A real-world application: AES-based PRNG . . . 103
4.2.1 AES as PRNG . . . 103
4.2.2 Implementation . . . 104
4.2.3 Performance study . . . 109
4.2.4 Productivity study . . . 115
4.2.5 Summary . . . 116
4.3 Random task-DAGs . . . 116
4.3.1 Benchmark application . . . 117
4.3.2 Evaluation . . . 121
4.3.3 Discussion . . . 122
4.3.4 Summary . . . 133

5 **Conclusions and future work** **135**

**Bibliography** **141**

# List of Figures


1.1 Growth rate of the number of components per IC as originally predicted by Moore's law. . . . . . 2
1.2 Callout of Intel Nehalem quad-core processor's die. . . . . . 5
1.3 Microprocessor trend data up to 2017 . . . . . . 8
1.4 Various speedups obtained doubling the number of cores for a few values of $p$, the fraction of code that can be executed in parallel. . . . . . 10
1.5 Evolution of Intel CPUs' and NVIDIA GPUs' peak performance in GFLOPS during the years. . . . . . 12
1.6 NVIDIA TU102 full GPU block diagram. . . . . . 13
1.7 Particular of NVIDIA Turing Streaming Multiprocessor. . . . . . 14
1.8 AMD Radeon RX 5700 XT full GPU block diagram. . . . . . 15
1.9 Particular of AMD RDNA architecture Dual Compute Unit. . . . . . 16

2.1 Coalesced access pattern to global memory. . . . . . 25

3.1 PHAST Library containers and the coordinate system adopted. From left to right: **`vector`**, **`matrix`**, and **`cube`**. . . . . . 46
3.2 Comparison between **`cudaMemcpy`** elapsed times: one copy of N elements VS N copies of one element. The elements are 4 byte **`float`** values. . . . . . 47
3.3 PHAST Library iterators. . . . . . 51
3.4 PHAST Library grid iterators. . . . . . 52
3.5 Two examples of coalesced accesses to global memory. . . . . . 56
3.6 Binary **`for_each`** with a functor that processes, at each iteration, two vector sections. . . . . . 57
3.7 The task-DAG corresponding to **`task4`** in Listing 3.2. . . . . . 70
3.8 Data races resolution strategy: data copying and fake dependency addition. 75

4.1 Triad bandwidth on multi-core systems, measured in GBps. . . . . . 84
4.2 Triad bandwidth on NVIDIA GPUs, measured in GBps. . . . . . 85
4.3 Visual representation of DCT8x8. . . . . . 88

4.4 PHAST DCT8x8 performance on multi-core systems, measured in Megapixels per second. . . . . . . . . . . . . . . . . . . . . . . . . . 90
4.5 DCT8x8 performance on NVIDIA GPUs, with `ONE_ELEM_PER_BLOCK` tiling strategy, measured in Gigapixels per second. . . . . . . . . . . . 91
4.6 DCT8x8 performance on NVIDIA GPUs, with `N_ELEM_PER_BLOCK` tiling strategy and no shared pre-load, measured in Gigapixels per second. 91
4.7 Black-Scholes performance on multi-core systems, measured in Megaoptions per second. . . . . . . . . . . . . . . . . . . . . . . . . . . 95
4.8 Black-Scholes performance on NVIDIA GPUs, measured in Gigaoptions per second. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
4.9 N-Body performance on multi-core systems, measured in Megainteractions per second. . . . . . . . . . . . . . . . . . . . . . . . . . . 99
4.10 N-Body performance on NVIDIA GPUs, measured in Gigainteractions per second. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
4.11 A grid of eight elements is applied to vector data, so to iterate over chunks of eight elements. . . . . . . . . . . . . . . . . . . . . . . . 105
4.12 AES performance on multi-core CPUs using PHAST Library in correspondence of different numbers of threads. . . . . . . . . . . . . . . . 111
4.13 The impact of affinity management on multi-core performance. . . . . 112
4.14 AES performance on GeForce GPU GTX970 in correspondence of different combinations of **`major_block_size`**, **`shared_pre_load`** and **`scheduling_strategy`**. . . . . . . . . . . . . . . . . . . . . . 113
4.15 AES performance on GeForce GPU GTX1080 in correspondence of different combinations of **`major_block_size`**, **`shared_pre_load`** and **`scheduling_strategy`**. . . . . . . . . . . . . . . . . . . . . . 114
4.16 Example of the task-DAG corresponding to benchmark 8-1-2. . . . . . 121
4.17 Comparison of elapsed times of task-DAG benchmarks on multi-core CPU with workload of 32x32 matrices. . . . . . . . . . . . . . . . . . 123
4.18 Comparison of elapsed times of task-DAG benchmarks on multi-core CPU with workload of 128x128 matrices. . . . . . . . . . . . . . . . . 126
4.19 Comparison of elapsed times of task-DAG benchmarks on multi-core CPU with workload of 512x512 matrices. . . . . . . . . . . . . . . . . 127
4.20 Comparison of elapsed times of task-DAG benchmarks on NVIDIA GPU with workload of 32x32 matrices. . . . . . . . . . . . . . . . . . 128
4.21 Comparison of elapsed times of task-DAG benchmarks on NVIDIA GPU with workload of 128x128 matrices. . . . . . . . . . . . . . . . . . 129
4.22 Comparison of elapsed times of task-DAG benchmarks on NVIDIA GPU with workload of 512x512 matrices. . . . . . . . . . . . . . . . . . 130

# List of Tables


1.1 Dennard's scaling's relations between MOSFET dimensions and other device parameters. . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.2 Scaling effects on interconnections lines. . . . . . . . . . . . . . . . . 4
1.3 SIMD extensions supported by modern x86 processors. . . . . . . . . 7

2.1 Ten most popular programming languages according to March 2020 TIOBE index. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.2 CUDA memory types with their (de-)allocation and access capabilities by host and device. . . . . . . . . . . . . . . . . . . . . . . . . . . 24
2.3 Some correspondences between CUDA and HIP APIs. . . . . . . . . . 27
2.4 Heterogeneous programming approaches. . . . . . . . . . . . . . . . . 29
2.5 Thread hierarchical organization terminology used in various programming frameworks. . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.1 List of PHAST Library iterators. . . . . . . . . . . . . . . . . . . . . 50
3.2 List of PHAST Library algorithms. . . . . . . . . . . . . . . . . . . . 54
3.3 List of PHAST Library parallelization parameters. . . . . . . . . . . . 59
3.4 List of PHAST Library in-functor iterators. . . . . . . . . . . . . . . . 63
3.5 List of the in-functor algorithms included in PHAST Library. . . . . . 64

4.1 Multi-core based machines used to evaluate simple examples. . . . . . 80
4.2 NVIDIA GPUs used to evaluate simple examples. . . . . . . . . . . . 81
4.3 Triad bandwidth comparison on multi-core systems, measured in GBps. 85
4.4 Triad bandwidth comparison on NVIDIA GPUs, measured in GBps. . 86
4.5 Complexity metrics calculated on PHAST, CUDA, SYCL, Kokkos, and OpenCL implementations of the Triad benchmark. . . . . . . . . . . . 87
4.6 DCT8x8 performance comparison on multi-core systems, measured in Gigapixels per second. . . . . . . . . . . . . . . . . . . . . . . . . . 92
4.7 DCT8x8 performance comparison on NVIDIA GPUs, measured in Gigapixels per second. . . . . . . . . . . . . . . . . . . . . . . . . . 93

4.8 Complexity metrics calculated on PHAST, CUDA, SYCL, Kokkos, and OpenCL implementations of the DCT8x8 benchmark. . . . . . . . . . 94
4.9 Black-Scholes performance comparison on multi-core systems, measured in Gigaoptions per second. . . . . . . . . . . . . . . . . . . . . 96
4.10 Black-Scholes performance comparison on NVIDIA GPUs, measured in Gigaoptions per second. . . . . . . . . . . . . . . . . . . . . . . . . 97
4.11 Complexity metrics calculated on PHAST, CUDA, SYCL, Kokkos, and OpenCL implementations of the Black-Scholes benchmark. . . . . . . 97
4.12 N-Body performance comparison on multi-core systems, measured in Gigainteractions per second. . . . . . . . . . . . . . . . . . . . . . . . 100
4.13 N-Body performance comparison on NVIDIA GPUs, measured in Ginteractions per second. . . . . . . . . . . . . . . . . . . . . . . . . . 101
4.14 Complexity metrics calculated on PHAST, CUDA, SYCL, Kokkos, and OpenCL implementations of the N-Body benchmark. . . . . . . . . . 101
4.15 Average complexity ratio of PHAST Library with respect to the other frameworks across all the considered metrics. . . . . . . . . . . . . . . . 102
4.16 Performance comparison between single-thread AES PRNGs on CPUs, measured in Gbps. . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
4.17 Performance comparison between AES PRNG implementations on NVIDIA GPUs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
4.18 Complexity metrics calculated on different AES PRNG implementations. 115
4.19 Parameters of the randomly generated benchmark applications. . . . . 118
4.20 Execution time ratio between SYCL and PHAST versions of the same task-DAGs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
4.21 Complexity metrics measured on PHAST, PHAST SA (single architecture), and SYCL implementations of various benchmarks. . . . . . . 131

# CHAPTER 1

# Introduction

## 1.1 The laws of technological progress

### 1.1.1 Moore's law

In 1965, Dr. Gordon E. Moore, then director at Research and Development Laboratories of Fairchild Semiconductor, a division of Fairchild Camera and Instrument Corp., wrote a famous article about the predicted growth rate of the number of components in Integrated Circuits (ICs) [109]. He observes that, in the period 1959-1965, the continuous shrinking of transistors was such that their number per chip roughly doubled every year. Thus, he predicted that the same rate would remain constant for the next ten years, at least, as showed in Figure 1.1. This prediction is known as *Moore's law*.

In the same article, he also states that the main sources of power consumption in a chip are the capacitances associated to it. Since these depend on its overall size, he observes that increasing the number of transistors would not impact power consumption as long as the chip area remains constant [109]. Thus, reducing the transistors' size to put more and more of them in a chip while keeping its size constant, would be a viable way to increase complexity without increasing energy needs [109]. Moreover, from the transistor shrinking would result also an increase in the operating speed [109], which is a fundamental property of ICs.

Moore's law, more than a *law* stricto sensu, is a fluid idea that changes its particular

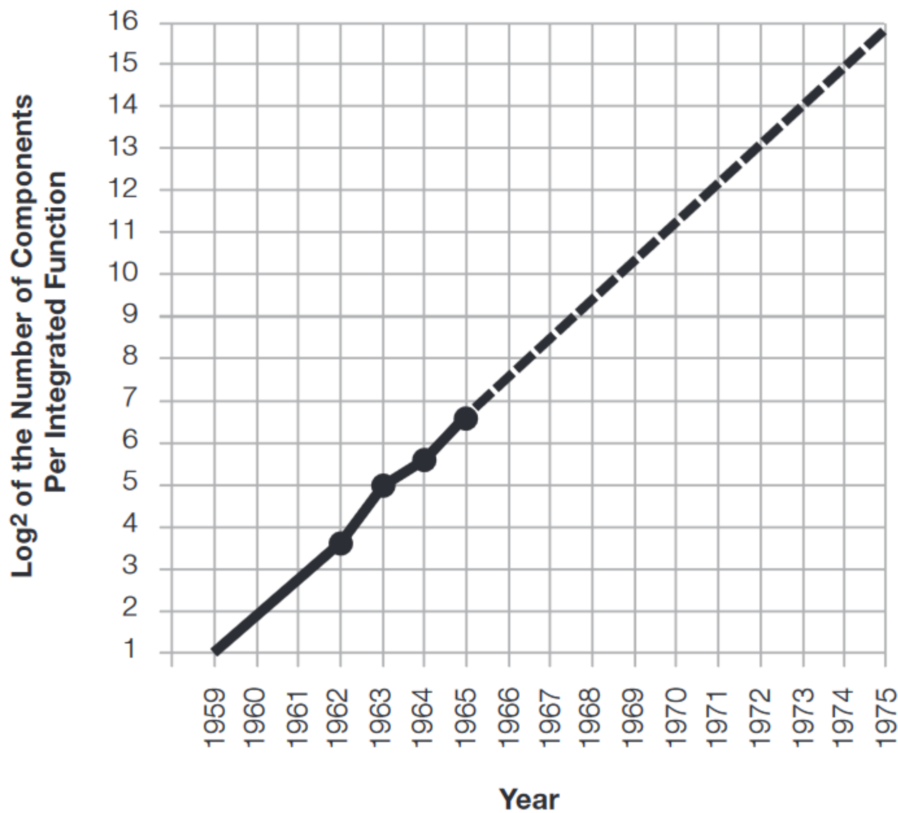



**Figure 1.1:** Growth rate of the number of components per IC as originally predicted by Moore's law.

nature over time [29], a self-fulfilling prophecy [108]. Its original formulation has undergone some updates during the years, with the most popular version indicating "18 months" as the period in which the transistor count per IC doubles [108]. Thanks to those updates, Moore's law's scope extended well over its original span, as it has been able to describe the evolution of ICs, and thus microprocessors, until the present day.

### 1.1.2 Dennard scaling

Moore's predictions, expressed *in words* in his original article, were better formalized in 1974 by Robert H. Dennard in the case of Metal-Oxide Semiconductor Field-Effect Transistors (MOSFETs) [32], the transistor technology that is still in use today. He shows that, by reducing the gate oxide's thickness ($t_{ox}$), the length ($L$), and the width ($W$) of transistors by a factor $k$, the delay time per circuit reduces too by the same factor $k$ without increasing the power density, as long as also the voltage and the current are scaled accordingly [32]. Table 1.1 shows these and other expressions, which are collectively known as *Dennard's scaling law*, or simply *Dennard scaling*. It is worth noting that "Delay time / circuit" is the inverse of speed, and thus the whole scaling law

| Device or circuit parameter | Scaling factor |
|---|---|
| Device dimension $t_{ox}$, $L$, $W$ | $1/k$ |
| Doping concentration | $k$ |
| Voltage | $1/k$ |
| Current | $1/k$ |
| Delay time / circuit | $1/k$ |
| Power dissipation / circuit | $1/k^2$ |
| Power density | 1 |

**Table 1.1:** Dennard's scaling's relations between MOSFET dimensions and other device parameters.

can be summarized as "as transistors get smaller, they can switch faster and use less power" [14].

Dennard scaling indicated a path to follow to the whole IC industry. Its scaling principles were adopted as a roadmap, thanks to their ability to provide systematic and predictable improvements [14]. As a consequence, in the period 1975-2006, the most part of IC research was focused on reducing transistor size. These ever shrinking transistors passed from being around 5 $\mu$m in 1975 [14] to 65 nm of Intel Core 2 Duo E6700 in 2006 [61]. Their number per chip went from being around one hundred to around 300 millions. The gate oxide's thickness shrunk at exponential rate as expected, even faster than expected starting with the 0.35 $\mu$m generation in the mid-1990s [14]. The operating frequencies have been increasing generation after generation, surpassing 3 GHz sooner than most experts could imagine ten years before [14]. In conclusion, in thirty years of Moore's law and Dennard scaling, microprocessors' speed and complexity grew at exponential rate, with tangible impacts on nearly every aspect of society as a whole.

## 1.2 The crisis of Dennard scaling

Dennard scaling is now considered no longer valid. Some authors advocate that it definitely ended in 2003 [29], others were more optimistic about it still in 2006 [51] and 2007 [14], even though they acknowledged its limits. Anyhow, the critical issues related to its continuation were evident to the whole IC industry. An extensive summary of them with a comprehensive bibliography can be found in [51]. The main issues can be categorized as:

### Material properties

Scaling did not affect (or affected only slightly) the permittivity constant of the gate insulator. Conversely, vertical fields grew, leading to a decrease in channel mobility.

Moreover, the ever shrinking gate's thickness was approaching dimensions where the contribution of the tunnel effect would have been significant [51].

### Energy dissipation

Active power was no longer the only major contribution to the power balance. Passive power's importance was growing, since sub-threshold voltage did not scale along with operating voltage as expected in the 1970s [14]. In order to contain leakage, the voltage scaling was slowed down, avoiding operating voltages below 1 V. This lead to tradeoffs between active and passive power that limited chip performance [51].

### On-chip interconnects

| Device or circuit parameter | Scaling factor |
|---|---|
| Line resistance | $k$ |
| Normalized voltage drop | $k$ |
| Line response time | 1 |
| Line current density | $k$ |

**Table 1.2:** Scaling effects on interconnections lines.

Line interconnects' response time depends on the time constant RC, the product between line resistance and line capacitance. Dennard showed that, since the scaling in line capacitance is balanced by an increase in line resistance, the time constant is not affected by scaling [32], as expressed in Table 1.2. This effect did not get much attention at start, because the line delay was a negligible fraction of clock at the time. With the growth of clock frequency and the microarchitecture approaching the nanometer scale, however, the delay role grew in importance [14]. Wire delays became responsible for latencies of many clock cycles, such that manufacturers had to hide them with longer and longer pipelines that ultimately led to the Intel Prescott processor design, which featured more than 30 pipeline stages [72].

All of these effects combined made clear that traditional processor design was not a viable solution any longer.

## 1.3 The advent of multi-core CPUs

With the end of Dennard scaling, increasing microprocessors' speed as done in thirty years became impossible. Hardware manufacturers had to find a way to continue technological progress, possibly without abandoning Moore's law.

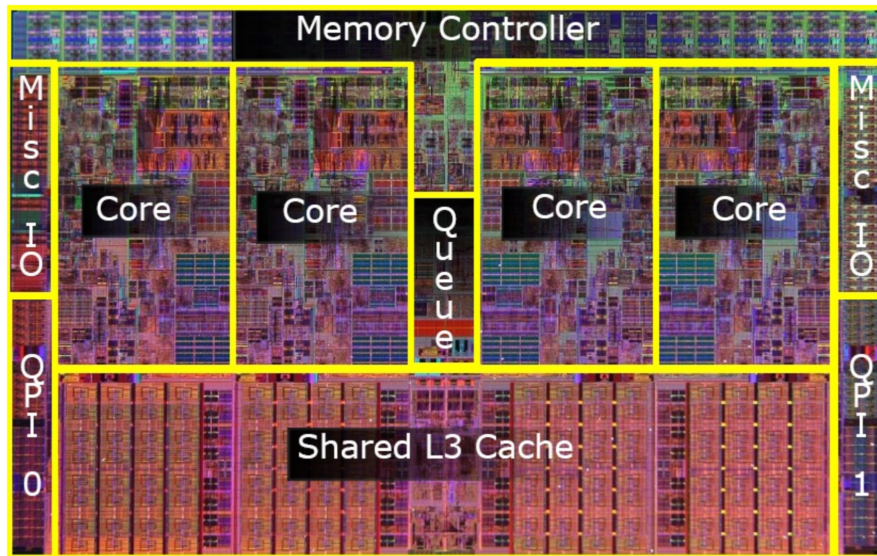



**Figure 1.2:** Callout of Intel Nehalem quad-core processor's die.

They found the answer in multi-core design: instead of keeping improving the performance of single processors, they started shipping microprocessors with multiple processors per chip [129]. These renowned components took the name of "multi-core microprocessors" or "multi-core CPUs", with the term *core* referring to each of the single processors that make up a multi-core CPU [129]. Figure 1.2 shows an example of such a multi-core: a quad-core Nehalem from Intel [60].

### 1.3.1 Multiprocessor architecture

Multiprocessors can be synthetically described as "computers consisting of tightly coupled processors whose coordination and usage are typically controlled by a single operating system and that share memory through a shared address space" [56]. Multi-core microprocessors are a special case of multiprocessors having all the cores on a single chip. They are also called Chip Multiprocessors (CMPs). The general case admits also multiple chips interconnected, each of them possibly being a multi-core. Each of the cores of a multiprocessor can be regarded as the evolution of a super-scalar uniprocessor: as such, they support all the architectural achievements of that field, like Instruction Level Parallelism (ILP), branch prediction, out-of-order execution, a long pipeline, vector units, and so on. In this section, the two terms are kept distinct with the meaning just expressed. In the rest of this work, the term "multi-core" is preferred instead to denote a generic processor with two or more cores. Since these are treated as logical entities, the belonging to the same or different chips does not affect the validity of the considerations.

Multiprocessors allow the execution of programs organized in multiple cooperat-

ing threads as well as more complex applications organized in relatively independent processes that may or may not communicate via Inter Process Communication (IPC) mechanisms. From a Flynn's taxonomy point of view [43], they have a Multiple Instruction Multiple Data (MIMD) execution model: processors are independent, and thus they can execute different instructions from different programs, operating on different data.

Multiprocessors can be further classified according to their memory organization. Symmetric MultiProcessors (SMPs) have a single centralized memory that is accessed the same way by all the processors [56]. For this reason, their architecture is also called Uniform Memory Access (UMA). The majority of multi-core CPUs fall into this category [56]. Distributed Shared Memory (DSM) multiprocessors have a distributed memory supported by a high-bandwidth interconnection network, generally a switch or a multidimensional mesh. Because of the topology of the memory, they are also called NonUniform Memory Access (NUMA) processors. In this latter case, the access time to data depends on the physical location of the data in memory [56]. In both cases, however, the shared address space makes every word in memory accessible by every core in the system, with data transfer, if necessary, performed by the circuitry without the need for special care in the software layer [56]. Of course, this *care* is not necessary to write correct programs, but it can help to achieve better performance when spent limiting data transfers and exploit data locality.

SMPs support caching of data that is managed in different ways depending on whether it is private or shared data [56]. *Private data* is migrated to the cache of the processor using it. *Shared data* may be replicated in the caches of the processors using that data. While modifications of private data do not involve the other cores, modifications to shared data should be propagated to those sharing the data in order to avoid accesses to outdated values and hamper the correctness of the program. Cache coherence protocols, both directory-based and snooping like MSI, MESI, or MOESI, are thus implemented to manage this situation and maintain *coherent* values across different caches [56].

DSMs have become the most common design in recent years [56]. With the every increasing multi-core count per multiprocessor, the bandwidth demand of a centralized memory system would incur in long access latency that would slow down the whole system. The distributed solution, due to its lesser bandwidth demand, is preferred instead. Cache coherence protocols are still necessary in DSMs, but they have to manage also the communication across chip. This is generally done by implementing a directory-based protocol, with the directories distributed across chips. On top of them, a snooping protocol can be added to manage coherence inside each multi-core [56].

Since a few years now, multi-core CPUs, generally, have two private caches (L1 and L2) for each core and a large shared Last-Level Cache (LLC) that is also used

to communicate between on-chip cores. Its growing capacity, due to both higher bandwidth needs and the availability of more bits per unit area, led to ever-increasing wire delays that challenged the classical cache design, then called Uniform Cache Architecture (UCA) [86]. It was characterized by a centralized decoder that drove physically partitioned sub-banks, with latencies that varied according to the distance between the processor and the bank containing the requested data. However, access time was mostly tuned to the worst-case of the farthest cache zones. In order to avoid the high latencies of the most distant data, Non-Uniform Cache Architecture (NUCA) design was proposed [86]. This design is characterized by a scalable infrastructure that connects the banks in a bank-partitioned cache [44]. NUCAs can be further classified as *static* or *dynamic*, according to the mapping policy which can assign blocks to banks statically or dynamically, respectively. In the latter case, blocks are migrated near the most frequently referring core, in order to reduce the average access latency [44].

Today's multiprocessors generally support vector registers and vector operations. Vector registers are large sequential registers that can host a fixed number of words. Vector operations like load, store, add, subtract, etc. are executed by special vector units that manipulate operands in vector registers. They can be executed by invoking vector instructions that are added to processors' Instruction Set Architecture (ISA) by means of SIMD extensions, referring to the Single Instruction Multiple Data (SIMD) execution model they provide.

| SIMD extension | Register width | Main capabilities |
|---|---|---|
| SSE | 128 bit | operations on single precision FP numbers |
| SSE2 | 128 bit | operations on double precision FP numbers and integers of 8, 16, 32, and 64 bit |
| SSE3 | 128 bit | horizontal arithmetics operations |
| SSSE3 | 128 bit | various operations like shuffle, negate, and other horizontal ones |
| SSE4 | 128 bit | compare between strings, multiplication between 32-bit integers, dot product and more |
| AVX | 256 bit | operations on eight single precision or four double precision FP numbers |
| AVX2 | 256 bit | operations on integers, vector shifts and the ability to load elements from non-contiguous locations |
| AVX-512 | 512 bit | many previous AVX operations extended to 512-bit registers plus exponentiations, prefetches, and more |

**Table 1.3:** SIMD extensions supported by modern x86 processors.

SIMD instructions are not a recent achievement of computer architecture. The first

supercomputer with SIMD instructions was Illiac IV, designed in 1966 [59]. Intel introduced MMX instruction set extension in 1996 mainly to boost the performance of multimedia applications [56], that are characterized by the repetitions of the same operations on many elements, and thus were the ideal candidates for SIMD operations. They worked with 64 bit registers. After Intel, AMD was also given the opportunity to support them in their processors. MMX evolved in SSE, introduced in Pentium III and originally called Internet Streaming SIMD Extensions [157], which manipulates 128-bit data registers. Further additions have been done over the years with the names SSE2, SSE3, SSSE3, and SSE4. Advanced Vector Extensions (AVX), then further enriched with AVX2 and AVX-512, is the most recent SIMD extension. AVX and AVX2 support 256-bit registers with the majority of SSE operations adapted to the new width. AVX-512 introduced around 250 new instructions for 512-bit registers [56]. It was initially supported by few Intel multiprocessors (mainly of the Xeon series) and Xeon Phi co-processors, and it is now being shipped more extensively. Also ARM processors have their SIMD extension as Neon technology, included in Cortex-A and Cortex-R series processors and supporting 128-bit registers [8].

Table 1.3 lists the SIMD ISA extensions that can be found in modern x86 multiprocessors with their main capabilities.

### 1.3.2 Multi-core and performance scaling

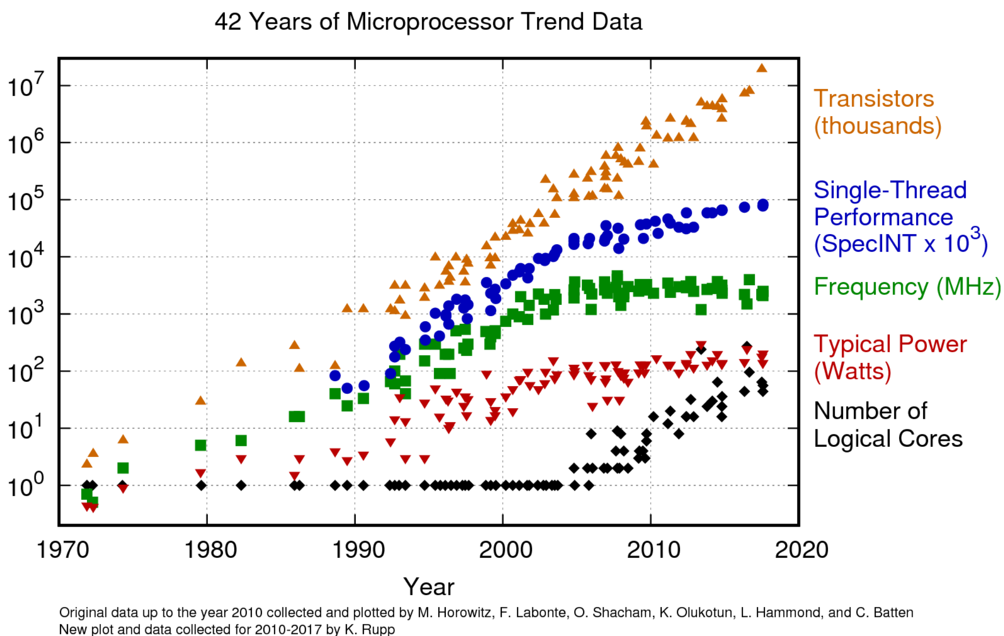



**Figure 1.3:** Microprocessor trend data up to 2017

Figure 1.3 shows that the multi-core advent did not cause the end of Moore's law.

Around 2006, single-thread performance scaling had a slowdown, frequency scaling stopped completely, but the transistor count continued to grow at the same rate, dragged by the growth in the number of cores [76].

Although Moore's law is still valid even now in the multi-core era, it does not mean what it used to mean for users and developers. In the past, they could expect that the doubling in transistor count every 18 months would have caused their programs' *response time* to be halved, without the need to change a line of code [129]. Nowadays, rather than the response time, it's the *throughput* that potentially benefits the most. Taking advantage of it needs particular care, that can be summarized as the challenges of *parallel programming*: programs must be coded in a way that partitions computations and deals with communication between a varying number of processors. A *varying* number, because if it were a *fixed* number, the program's performance would not gain from hardware upgrade as it used to do in the single-core era.

Even back then, however, the statement that "performance used to grow at the same rate as transistor count" was optimistic. A program's performance is due to many factors (compute intensity, memory latency, network latency, disk latency, ...), and not all of them gain from the sheer number of transistors the same way. Memory speed, for instance, never scaled at the same rate as processor speed [171]. As a simplification, we can think that *compute bound* programs used to double their performance every 18 months. In the multi-core era, however, even this is no longer true: the increase in the number of cores can only improve the fraction of the program that can be executed in parallel.

The impact of this difference is evident if we consider the speedup of a given program executed on a processor with $n$ cores with respect to one with $m$ cores as:

$$S(n, m) = \frac{T_m}{T_n} = \frac{1 - p + \frac{p}{m}}{1 - p + \frac{p}{n}}$$

Where $T$ is the execution time and $p$ is the fraction of the program that can be executed in parallel. By setting $m = 1$, i.e., considering a sequential processor as a baseline, we obtain the famous Amdahl's law [7]:

$$S(n, 1) = \frac{1}{1 - p + \frac{p}{n}}$$

Figure 1.4 shows such impact. It shows the effect on speedup of doubling the number of cores for various values of $p$.

A first thing to consider is that, if the program is intrinsically sequential, the speedup is always equal to 1: increasing the number of cores gives no advantage from a performance point of view. On the contrary, a program that has no sequential code has

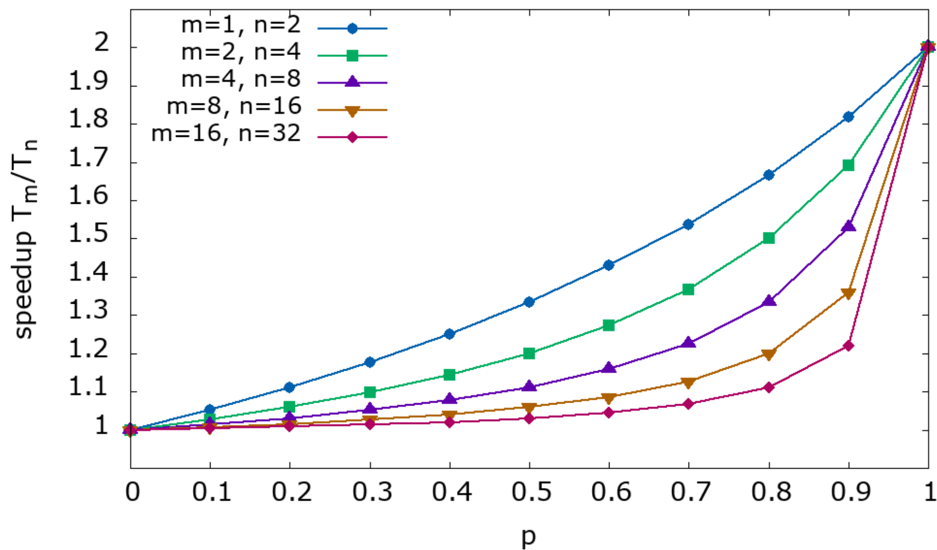



**Figure 1.4:** Various speedups obtained doubling the number of cores for a few values of $p$, the fraction of code that can be executed in parallel.

a speedup of 2 independently of the number of cores. All the cases in between show a situation that is far from being ideal: the performance gain that can be obtained by doubling the number of cores is higher for lower values of $m$ and $n$. The curves for higher values of $m$ and $n$ have low slopes that grow only for high values of $p$. It means that, when a number of tens of cores is reached, further increasing their number translates into substantially better performance only for those programs that execute parallel code for a significant fraction of their total execution time. Seeing this from another point of view, less and less already existing parallel programs will gain from increasing the number of cores as the multi-core generations come and go.

For instance, by taking into account the curve $S(16, 32)$ in Figure 1.4, it results that a speedup of at least 1.5 (i.e., a 50% improvement in performance) passing from 16 to 32 cores is achievable only if the fraction of code that can be executed in parallel $p$ is greater than about $97\%$. It is difficult to say *how many*, but it is probable that few real world programs have that degree of parallelism.

These considerations highlight the need to extract as much parallelism as possible from applications, even if the performance return is not comparable to what programmers used to achieve in the single-core era. Hardware parallelism is what nowadays gives performance and potential throughput. Parallel portions of the code are those that benefit the most from it. Programmers' work should be focused into identifying as many of them as they can. Applications must be coded with parallelism and concurrency in mind, also with novel advancements in algorithm design that stress the role of parallelism [56]. Doing so is not easy, but libraries and frameworks can help programmers to write correct

parallel code without losing productivity. The next chapter addresses the issue in-depth.

## 1.4 The advent of GPUs

Modern Graphics Processing Units (GPUs) are the evolution of graphics coprocessors that accompanied CPUs from the PCs' early days. Even at the time of the IBM PC, one of the first personal computers[1], there was the possibility to expand its graphics capabilities by the means of discrete graphics cards, like the Hercules Graphics Cards [148]. Initially, their use was to manage part of the graphics rendering, but they were not autonomous, as they still relied on the CPU. They evolved from managing 2D graphics only to 3D graphics in the second half of the 90s. The first of NVIDIA's GeForce cards, the GeForce 256, released in 1999, was called "the world's first GPU" [48]. For the first time, an external card was able to perform lightning and shading autonomously, without the intervention of the CPU. After that, in the last twenty years, the evolution of their graphics capabilities continued at a tremendous pace, as shown in Figure 1.5, and as anyone can see by comparing modern video-games to even the best examples from twenty years ago. However, the improvements in graphics rendering are not the only thing that GPU evolution made possible: GPUs are also the principal cause of a major leap in the field of parallel computing.

### 1.4.1 GPGPU

In the early 2000s, GPUs became palatable for their computing capabilities to researchers, scientists, and developers not directly involved in graphics development. GPUs were powerful and inexpensive, compared to the best processors available on the market. In 2006, for instance, NVIDIA's GeForce 7900 GTX was capable of delivering 51.2 GB/sec of memory bandwidth, and the ATI's counterpart, Radeon X1900 XTX, had a peak performance of 240 GFLOPS [128]. On the CPU side, a dual-core Intel Pentium Extreme Edition 965 @ 3.7 GHz had a memory bandwidth of 8.5 GB/sec and a peak performance of 25.6 GFLOPS, measured on its SSE units [128]. As for the prices, the dual-core had a "recommended customer price" of 999$ at launch (Q1 2006) [62], while GeForce's price was around 500$[2].

Today, this huge discrepancy in computing power is still there, as Figure 1.5 from NVIDIA [119] shows.

The effort of performing calculations on GPUs not related to graphics took the name of General Purpose computing on Graphics Processing Unit (GPGPU) [128]. Even the first researchers, scientists, and developers attempting GPGPU were aware that GPUs, with their highly parallel architecture, were not suitable for *every* type

[1] IBM PC was released in 1981.

[2] On the official NVIDIA website, back in 2006, a reviewer mentioned "GeForce 7900 GTX's 500$ price tag" [115].

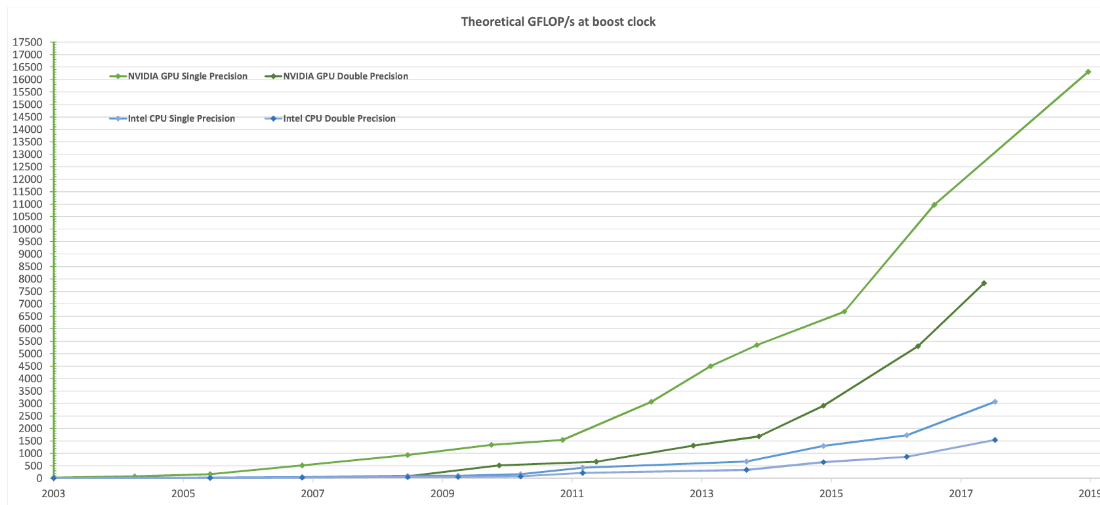



**Figure 1.5:** Evolution of Intel CPUs' and NVIDIA GPUs' peak performance in GFLOPS during the years.

of calculation. The problems that could be mapped on GPUs shared with graphics calculations the highly data-parallel nature: robot motion planning, artificial neural networks, convolution and wavelet transforms, physically-based simulations, signal and image processing, geometric computations, and more [128]. At first, all of these problems had to be expressed in terms of shaders. Since GPUs were made to generate images, even the best languages available in terms of expressiveness were designed with the special purpose nature of their target devices in mind. The best option programmers willing to do some GPGPU had, was to formalize and solve a *dual problem* that could be expressed using the graphics elements like textures, pixel shaders, vertex shaders, and so on [128]. GPGPU programmers were forced to express their data as pixel color values, write them into a pixel framebuffer, use it as a texture map and use the texture map as an input of the next computation step [56]. All of it, for instance, to solve partial differential equations.

In November 2006, the introduction of Compute Unified Device Architecture (CUDA) by NVIDIA [119] changed the world of GPGPU. It is a language explicitly designed to allow programmers to write general purpose programs on NVIDIA GPUs. It extends C++, and thus it does not share the special purpose nature with the languages used for GPGPU before its introduction. With CUDA, programmers suddenly passed from manipulating pixels and textures to launching thousands of parallel threads that operate on standard IEEE floating-point elements.

The execution model of modern GPUs recalls the SIMD execution model, but it is not exactly the same. NVIDIA calls it a Single Instruction Multiple Threads (SIMT) [119], because the same instruction can be executed by different CUDA threads. However, it can also be seen as a Single Program Multiple Data (SPMD) [107] execution model: the

available cores execute the same program, but they can be executing different parts of that program. It is, however, more flexible than most examples of SPMD [114].

Because of the importance of CUDA from a parallel and heterogeneous programming point of view, it will be analyzed in-depth in the following chapter.

### 1.4.2 GPU architecture

GPU architectures evolved during the years mainly to provide better capabilities in graphics rendering. Even though there can be differences between GPUs of different manufacturers, they all are designed and optimized to perform that task. Graphics rendering is characterized by highly parallel operations. Thus, GPUs need to "crunch" as many numbers as possible. The best way to achieve this is by using the majority of transistors in arithmetic calculations, rather than complex flow control or data caching [119]. Thousands of simple processors that lack the most sophisticated capabilities of modern processors (out-of-order execution, branch prediction, long pipelines) are the best fitting design to efficiently perform the task at hand.

**Figure 1.6:** NVIDIA TU102 full GPU block diagram.

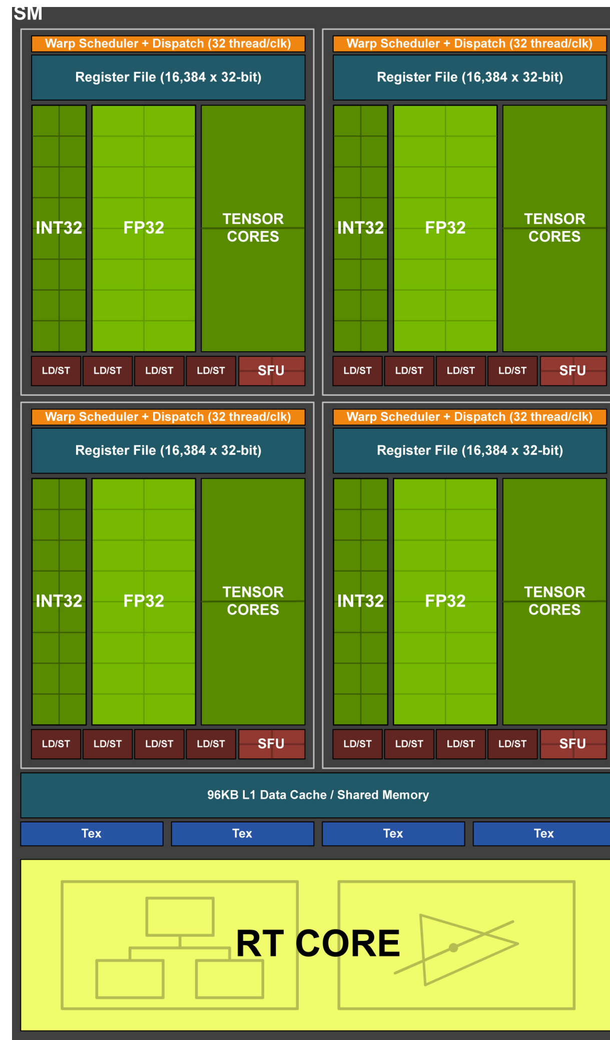



**Figure 1.7:** Particular of NVIDIA Turing Streaming Multiprocessor.

As an example, Figure 1.6 from [117] shows a block-diagram representation of NVIDIA's Turing TU102 chip, that can be found in GeForce RTX 2080 Ti cards.

From the Figure, it is evident that the whole GPU has a modular design. This model has 6 Graphics Processing Clusters (GPCs), which can be thought as full functioning GPUs on their own. A GPC has a Raster Engine and 6 Texture Processing Clusters (TPCs), each with a PolyMorph Engine and two Symmetric Multiprocessors (SMs). There are 12 32-bit memory controller, each connected to 8 Render Output Units (ROPs) and 512 KB L2 cache [117]. The L2 cache memory takes advantage of wired texture compression algorithms that reduce the data exchange between the cache and the GDDR6 main memory [117]. NVLink is an NVIDIA proprietary interconnection technology that connects two boards and allows each board to directly access the memory of the other, with a bidirectional channel that has a maximum bandwidth of 100 GB/s [117].

Figure 1.7 from [117] shows an enlargement of an SM. Each SM includes one Turing

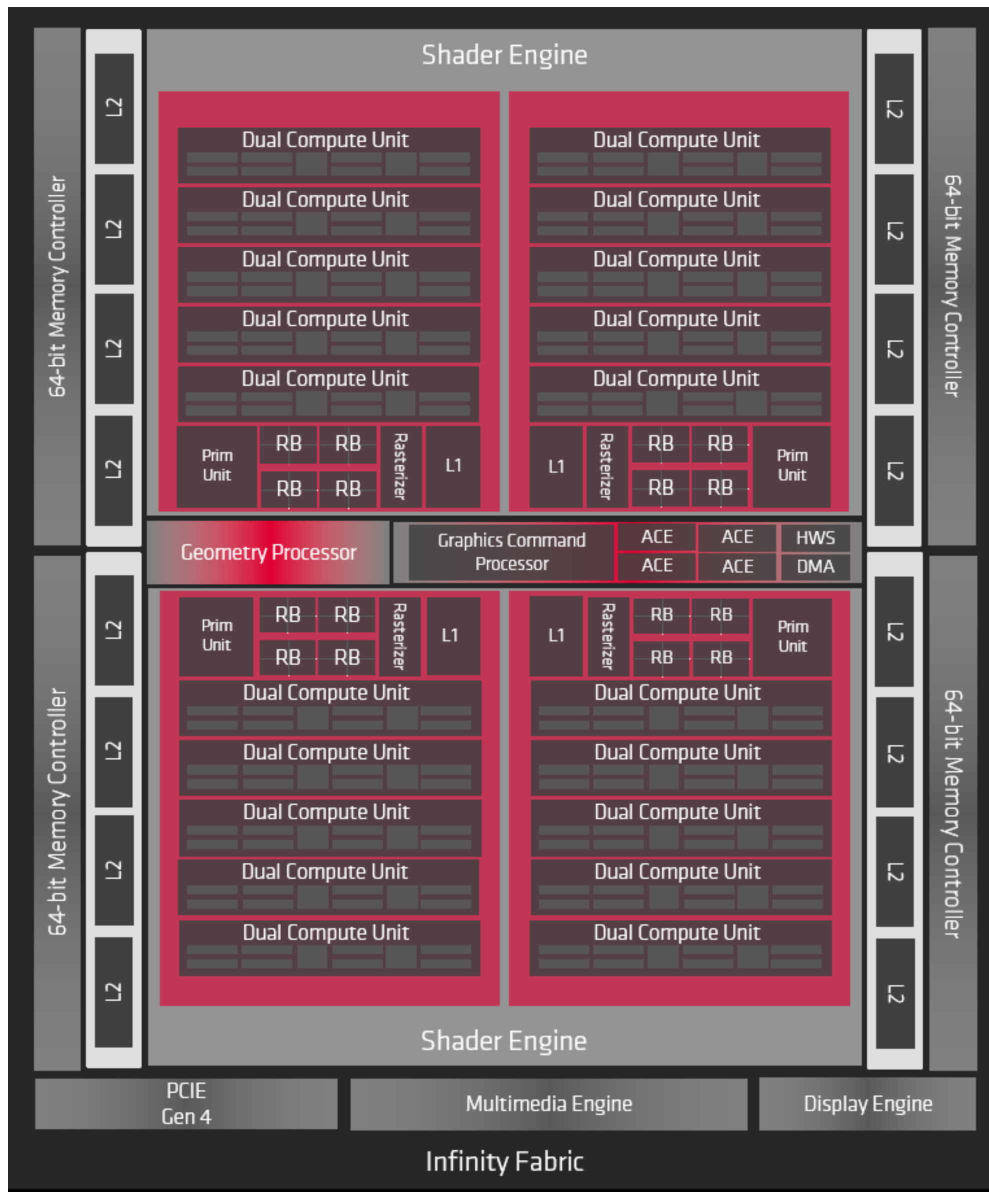



**Figure 1.8:** AMD Radeon RX 5700 XT full GPU block diagram.

RT core and 4 processing blocks, each with 16 FP32 cores for floating-point operations, 16 INT32 cores for integer operations, 2 mixed precision Turing Tensor Cores. The pair FP32 + INT32 can be seen as a single CUDA core. The two units have been displayed separately because Turing architecture introduces the possibility to execute integer and floating-point operations in two parallel execution paths [117]. Turing Tensor Cores are processors designed for Artificial Intelligence (AI) related calculations. They support INT4, INT8 and FP16 (half-precision floating-point) that can help that workload [117]. RT cores are used for real-time ray tracing, and have been introduced in this generation. Each processing block has also an L1 data cache that can be used also as shared memory,

a warp scheduler, and a dispatch unit that manage the execution of CUDA threads on CUDA cores. The CUDA related aspects will be discussed in the next chapter.

Figure 1.8 from [6] shows a block-diagram representation of AMD's Radeon RX 5700 XT. It is one of the first 7nm Navi series, the first one realized with the RDNA architecture [6].

The main part of this GPU are the two Shader Engines, where all the programmable logic takes place. They are connected to a shared L2 cache that stands between them and the memory controllers [6]. Each of them includes two shader arrays, which consist of five Dual Compute Units (CUs), an L1 cache, a primitive unit, a rasterizer, and four Render Backends (RBs). The primitive units are responsible for building triangles from vertices, the rasterizers emit pixels from triangles, and the RBs test, sample, and blend pixels. All these three components are part of the traditional graphics pipeline [6].

Figure 1.9 from [6] shows an enlargement of a CU, that hosts all the programmable resources. It comprises four SIMD units, each including 32 Arithmetic Logic Units (ALUs) that support operations on integers as well as half, single, and double precision floating-point data. They can operate on different data types, explicitly supporting various workloads such as scientific computing and AI related ones [6]. Various caches reduce the pressure on L2 cache and efficiently share data among units. L0 instruction cache can deliver 2-4 instructions per cycle to each of the SIMD units, which work independently [6].

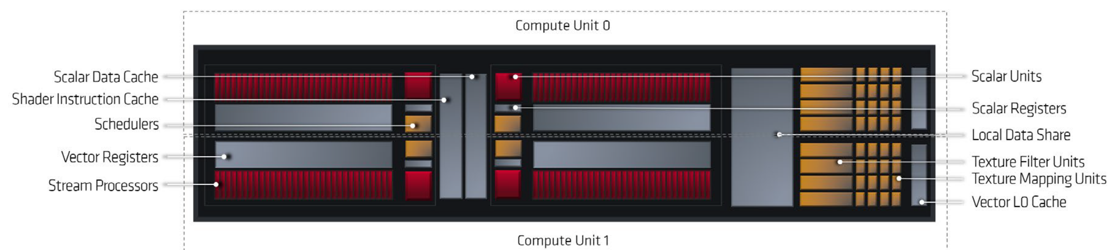



**Figure 1.9:** Particular of AMD RDNA architecture Dual Compute Unit.

The architectural discussion highlights the highly parallel nature of GPUs. These models, in particular, have respectively 72 SMs and 40 CUs, each with tens of cores. As a comparison, a typical desktop multi-core has around ten cores. So, GPUs can afford problems with a degree of parallelism up to two orders of magnitude higher with respect to a common desktop multi-core. Apart from that, another consideration can be done: while GPUs have units, pipelines, and engines specialized for graphics rendering, they are no longer limited to that task. After all these years of GPGPU, they can be legitimately perceived as *general purpose* co-processors: schedulers, integer and floating-point units of various precision, even tensor cores that are not part of the graphics rendering pipeline.

In conclusion, GPUs can be promising computing devices that can solve highly-parallel real-world problems. It is important that programmers can productively take advantage of them in their applications with easy-to-use and expressive programming models.

## 1.5 Heterogeneous computing

*Heterogeneous computing* can be defined as "a scheme in which the different computing nodes have different capabilities and/or different ways of executing instructions" [174]. The concept is not new [79], but it spread after the introduction of parallelism in microprocessor design with multi-core CPUs, when it became obvious that the sole replication of components might not be the best way to solve every computing problem.

Heterogeneous parallelism can be achieved by placing diverse computing resources side by side: general-purpose cores with different characteristics, special purpose cores, or even accelerators [71]. Thus, *computing nodes* in the definition above can refer to various resources, even located on the same chip. For instance, it is the case of ARM big.LITTLE technology, where fast and power-hungry processors are combined with slower and less demanding ones [9], or the case of AMD's Accelerated Processing Units (APUs), inaugurated with AMD Fusion technology to combine both CPU and GPU on the processor die [25].

However, the most natural form of heterogeneity is achieved by using the most common parallel architectures together: a multi-core processor and a GPU in the same system [26, 71]. It is a common configuration in desktop systems, and its importance is also testified by the fact that many supercomputers in the TOP500 list use this configuration. Among them, the most powerful and the second most powerful supercomputer in the November 2019 list are heterogeneous in this sense [161].

The importance of heterogeneous computing lies in both performance and energy efficiency [102, 139, 174]. Different computing devices have different architectures, optimized to solve problems with different characteristics. The sophisticated cores that can be found in multi-core processors are the best choice to run complex threads with complicated control flows and irregular memory access patterns [174]. The simple cores in GPUs, conversely, are optimized to manipulate large numerical datasets with few or no control flow divergences between threads. In this era of diverse tools, being able to use the right one for the right problem is fundamental.

The *ability* to use different devices for different problems translates mainly in their *programmability*. In its current state, programming heterogeneous systems is still hard [26, 174], and it poses several challenges that will be discussed in the next chapter. Various frameworks and libraries to program multi-core CPUs and GPUs exist. In the opinion of the author, none of them can be considered the definitive solution to

heterogeneous programming, even in the sub-domain of CPU+GPU systems. Some approaches are too low-level to be used productively, and thus force programmers to take care of many details that are related to the programming model rather than the logic of the problem at hand. Other offer a higher-level interface, but lack in other aspects like performance or performance portability between different architectures. These shortcomings have inspired the research activity that is herein presented.

CHAPTER 2

# Programming heterogeneity

When multi-core CPUs arrived on the market around fifteen years ago, programmers had to suddenly learn and master new parallel programming techniques. If sequential code was hard in the application-related aspects, like reasoning about program organization, algorithms, architectural patterns, and data structures, parallel programming presented the same difficulties enhanced with the necessity to manage parallelism at every level. Many new challenges are associated to this new programming model, the majority of them arising from the intrinsic asynchrony of the system: mainly, coordination of concurrent accesses to shared memory locations (data races), synchronizations between parallel activities (e.g., threads) and dealing with various sources of delay, which can be even orders of magnitude different from each other (cache misses, page faults, ...) [57]. Addressing these issues, programmers had to also become familiar with tools and concepts of parallel programming, like mutexes, locks, barriers, condition variables, atomics, and so on.

Heterogeneous programming is even harder than parallel programming. As previously specified, the main reason for attempting it is to obtain performance and/or power efficiency [139, 174], thanks to the coordinated exploitation of different architectures – having different features, strengths, and weaknesses – in the same system. To be able to have them, mainstream programmers need to address two issues [174]:

- Dealing with details of the underlying hardware;

- Dealing with the programming models and facilities of the devices involved.

The first is also an issue that arises in homogeneous parallel programming, or even sequential programming. However, when a variety of devices is taken into account, the hardware considerations that need to be done grow accordingly.

The second issue is related to the first, since the amount of hardware details exposed to the programmer depends on the specific programming model, tools, libraries, language extensions, etc. Yet, it has a broader extent: different devices may have different programming models, even *much* different between each other or with familiar approaches used in sequential programming. Each may need many months of experience in order to achieve the ability to write high-performance or energy-efficient code, depending on the complexity of the model but also on its *newness* with respect to familiar approaches.

In the following, the state-of-the-art of parallel programming frameworks and libraries will be analyzed. First, the device-specific ones for multi-core CPUs and GPUs. Then, heterogeneous approaches that target both families of architectures.

## 2.1 Multi-core programming

| Position | Programming Language | Ratings |
|---|---|---|
| 1 | Java | 17.78% |
| 2 | C | 16.33% |
| 3 | Python | 10.11% |
| 4 | C++ | 6.79% |
| 5 | C# | 5.32% |
| 6 | Visual Basic .NET | 5.26% |
| 7 | JavaScript | 2.05% |
| 8 | PHP | 2.02% |
| 9 | SQL | 1.83% |
| 10 | Go | 1.28% |

**Table 2.1:** Ten most popular programming languages according to March 2020 TIOBE index.

Multi-core CPUs can be programmed with a variety of programming languages. Since they are the evolution of the general purpose single-core processors, all the languages intended to write code for those platforms can also target multi-core CPUs. Table 2.1 shows the 10 most popular programming languages according to the March 2020 TIOBE index, which is a popular index updated monthly that takes into account various searching criteria [160]. Apart from SQL that is an interrogation language for Relational DataBase Management Systems (RDBMSs), all of these permit developing general purpose concurrent code on multi-core CPUs: Java includes a `Thread` class and various other constructs [127]; C and C++ have been extended around 2011 to natively

support concurrency [66, 67] after relying on external libraries like Pthreads [113], Win32 Threads [104], or Boost Threads [146] for years; Python has a **`threading`** module [140]; etc. Even PHP, which was born as a single-threaded language, has a **`parallel`** extension that enables concurrency from version 7.2 on [158].

Both concurrency and parallelism refer to running multiple tasks at the same time. The former focuses mainly on the separation of concerns, while the latter has a clear focus on performance [170]. Concurrency could also be obtained on a single-core system. Conversely, when speaking about *parallelism*, the presence of multiple cores in the system and the ability to fully utilize them are fundamental. Of course the two terms are quite similar and also overlapping in many aspects, yet their different nuances are still important. With those variations in mind, one can say that all the aforementioned languages may support concurrency, but not all of them are well-suited for parallelism.

It is a common opinion that some languages are not good for parallelism because some of their design principles may prevent the achievement of high-performance. For instance, the majority of Python virtual machines do not allow truly parallel code because they have a Global Interpreter Lock (GIL) that serializes execution [101]. Garbage collected languages like Java and C#, despite their ability to free programmers from explicitly managing memory, may be limited in performance by the garbage collection itself [58]. So, when dealing with parallelism on multi-core processors, C and C++, with their explicit memory management and *systems programming* [152] nature, are the languages that most likely are well-suited for the job. Also, heterogeneous approaches, which are the main focus of this work, are often designed as C/C++ extensions [81, 119], co-languages [125], or libraries [37, 84].

For the above reasons, the discussion of multi-core programming in this work is limited to C and C++ solutions.

### 2.1.1 C++ threads

The 2011 standardization of C++ marked the beginning of what some authors call "modern C++" [103]. It saw the introduction of various classes that permit parallel programming without using external libraries.

The main class can be considered **`std::thread`**, a class used to execute a task, i.e., "an activity potentially executed concurrently with other activities" [152]. C++ threads are mapped directly on the operating system's threads and can be executed on different cores. Threads share address space, and thus can communicate by accessing and modifying data in the same memory locations. From an interface point of view, a **`std::thread`** takes a task at construction in the form of a callable and its arguments and starts executing immediately, without the necessity to invoke explicitly a "start" operation [152]. Threads can be *joined* or *detached*: in the former case, the calling

thread blocks until the joined thread completes, in the latter, the **std::thread** object is separated from its underlying system thread so that it is not terminated when its owner is destructed and its management is left to the operating system.

Some common classes to manage synchronization and data-race avoidance are also provided. Between them, **std::mutex**es model the exclusive access to shared resources, **std::lock_guard**s and **std::unique_lock**s permit acquiring/releasing mutexes in a Resource Acquisition Is Initialization (RAII) flavor, and **std::condition_variable**s are used to let threads notify and signal the occurrence of specific conditions to other threads [152].

Simple lock-free operations that do not suffer from data-race problems may be implemented in terms of **std::atomic** variables. They are called *atomics* because the read/write operations they implement on most primitive types cannot be interrupted, or, in other words, they are sequentially consistent [92]. This property can be relaxed by explicitly specify the desired memory ordering, a feature that can be useful to implement high-performance drivers and libraries, but at the expense of code clarity [152].

The standard library contains also some classes and a function that facilitate the formalization of a common pattern in which an asynchronous operation is invoked at the sole purpose of producing a result. In these cases, spawning a thread to write some needed data in a shared variable and waiting for that data to be produced could be simplified by the means of futures and promises. A task can be launched with **std::async**, that produces a **std::future** that can be used to retrieve the produced value, which was stored into a promise. **std::async** can be invoked with two different policies that generate a separate thread (*async* policy) or emulate a lazy evaluation that is done synchronously when the future is asked for the stored value (*deferred* policy) [152].

Updates to the standard after 2011 introduced further additions to the concurrency capabilities of the language. C++14 [68] introduced the **std::shared_timed_mutex** to help to code patterns in which a writer thread and many reader threads compete to access a resource. C++17 [69] marked a bigger update by introducing the parallel algorithms into the standard. They are overloads of the classic, single-threaded versions with an additional parameter that selects the execution policy [170]. C++20 has not been published yet, but it is being finalized in these months and will further enrich the standard parallel programming facilities of C++ with advanced barriers and synchronizations, composable continuations, and coroutines, to cite a few.

### 2.1.2 C threads

Similarly to C++, also C was updated in 2011 with the addition of various concurrency facilities. They are located in **threads.h** and **stdatomics.h** headers [67].

The type **thrd_t** has been introduced to denote threads [67]. These are created,

terminated, and generally manipulated by many functions that accept **thrd_t** or **thrd_t*** arguments. C11 threads are analogous to C++11 threads in every aspect, apart from some limitations that depend on the nature of the C language. Beyond the obvious differences of creation and destruction, an important difference is the specific nature of the task, as opposed to the general nature of C++ tasks: in C, a task is a function with the signature **int(*)(void*)**. Its arguments are passed during creation as a **void*** argument, that can be casted to the needed type at runtime [67]. Using **void*** is a common solution in C to implement generic code, but it suffers from the lack of type-safety.

Mutual exclusion is obtained by means of mutexes, identified by the type **mtx_t** [67]. Locks and unlocks are performed by invoking the corresponding functions. Obviously, in C there are no special RAII classes like C++ locks.

Thread communication can be achieved via condition variables, identified by the type **cnd_t**. They support signal, broadcast, wait, and destroy operations that can be performed with the corresponding functions [67].

Many atomic types have been introduced to represent C integer types of various sizes. They can be manipulated by test-and-set, init, store, load, and other functions included in the standard. Apart from the **atomic_flag** type, not all of them are guaranteed to be lock-free. For each of those, a utility macro has been introduced to inform whether the referred type is never, sometimes, or always lock-free [67].

The standard has been further revised in C18 [70], which did not introduce any new features.

## 2.2 GPU programming

During the years, GPUs have evolved until becoming general purpose co-processors with thousands of cores optimized to execute different applications with a high degree of parallelism. GPU programming frameworks are designed to allow users to code with a formalism that is similar to CPU programming and departs from shader programming. Some common programming solutions are discussed in the following.

### 2.2.1 CUDA

CUDA [116, 119] was born as a parallel computing platform and programming model for NVIDIA GPUs, whose architecture is discussed in Subsection 1.4.2. It is designed as a standard C++ extension with the addition of a few keywords and a special syntax to invoke kernels. During the years, it evolved into a whole ecosystem featuring also libraries and tools.

The center of any CUDA program can be considered the kernel: a regular function annotated with the keyword **__global__** that is triggered by the controlling host and

executed on the GPU in many replicas by CUDA threads. Kernels can call C++ functions annotated with `__device__` or launch other kernels since the introduction of dynamic parallelism in the standard[1]. All the device code can be written in standard C++, albeit without some features such as virtual functions, function pointers, or exception handling [119].

CUDA threads are part of a hierarchical structure that includes a mono- , bi-, or three-dimensional grid of thread blocks, each being a mono- , bi-, or three-dimensional block of threads. Grid and block dimensions are specified at the kernel call site with a special syntax:

```
kernel<<<blocks_per_grid, threads_per_block>>>( ... );
```

When a kernel is invoked, it is executed blocks_per_grid×threads_per_block times by as many CUDA threads [119]. CUDA blocks have a 3D index to position them inside the grid, CUDA threads have a 3D index to position them inside the block. The unused dimensions are set to 0 in the index. Both indexes can be accessed inside the kernel code to calculate the global index of the thread in the grid [119].

The threads inside a block can communicate via shared memory (more on that later). They are grouped by the Symmetric Multiprocessors in groups of 32 threads, called *warps*, and executed. Each thread has its own instruction address counter, so it can execute instructions independently of the other threads in the warp. However, maximum efficiency is achieved when all the threads execute the same code: if, conversely, there are branches and divergences, the executions are serialized and the different code paths are taken separately. CUDA provides barrier synchronizations that involve the threads in a block or the threads in a warp [119].

| | Host code | | Device code | |
|---|---|---|---|---|
| Memory | Alloc/Dealloc | Access | Alloc/Dealloc | Access |
| Host | YES | YES | NO | NO |
| Device global | YES | NO | YES | YES |
| Device shared | NO | NO | YES | YES |
| Device texture | YES | NO | NO | YES |
| Device constant | YES | NO | NO | YES |

**Table 2.2:** CUDA memory types with their (de-)allocation and access capabilities by host and device.

CUDA distinguishes between a *host* and a *device*. The host is the CPU that executes the main program and coordinates the kernel execution on the device, i.e., the GPU. They have separate memory spaces: Table 2.2 summarizes the host and device capabilities in terms of allocation, deallocation, and access to them. The host memory space is

[1] This feature is limited to device of compute capability higher than 3.5.

basically the CPU RAM, while the device has different memory spaces. The main is the global memory, that lies in the GPU RAM [119]. In general, the workflow of a CUDA program is as follows: the host allocates space on the device global memory, it initializes it with data, if needed, copying them from the host memory, it launches one or more kernels that read and write those data, and finally it copies result data back to host.

The recent introduction of Unified Memory (UM) simplifies the memory management: data copies can be managed by the underlying system when needed, and the programmer is freed by explicitly calling the relevant functions [119]. However, letting the system transparently manage memory transfers may cause performance loss in some cases. The possibility to gain performance by overlapping computation and memory transfers is lost, since the transfers are not visible to the programmer. Also, GPUs of compute capability lower than 6.0 do not support fine-grained memory transfers: all the data is transferred before kernel invocations regardless of them being accessed or not [119].

Shared memory is a fast-access memory[2] located into the processing block's L1 cache, as detailed in Subsection 1.4.2, that is accessed inside kernels. It can be used to implement communication between threads in a block. A common pattern is to write data in shared memory, synchronize threads in the block with a block-wise barrier (`__syncthreads()`), and then proceed with the consuming of that data. Static data can be annotated with `__shared__` keyword to specify that it should be placed in shared memory. There is also the possibility to place data in shared memory dynamically, but a sufficient amount must be reserved to each block in advance by specifying the desired size at the kernel call site with the `<<<...>>>` syntax [119].

Registers, constant memory, and texture memory complete the list. Registers are the fastest memory, since accessing them costs zero clock cycles unless there are read-after-write conflicts [116]. Constant memory is a special memory for constant data that is cached in the constant cache. Finally, texture memory is used to cache data in the texture cache and is particularly optimized to access data that present 2D spatial locality [119].

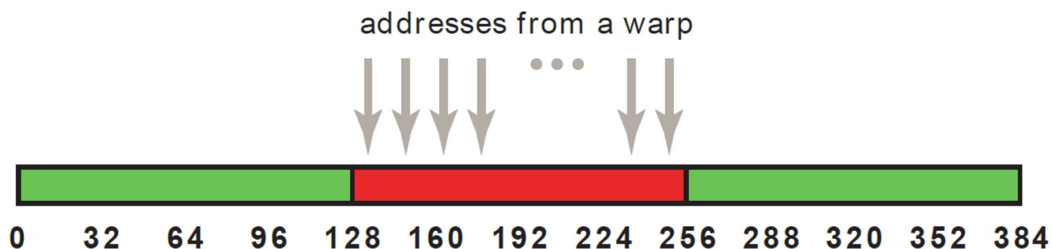



**Figure 2.1:** Coalesced access pattern to global memory.

[2]Registers and shared memory are the only on-chip memories [116].

The CUDA programming model introduced a new way to write massively parallel code. While it is not hard to write correct CUDA code, there are various aspects a programmer must take care of to write *efficient* CUDA code. The main aspect is the global memory access pattern of the threads [116]. The one that brings the best performance is the so-called coalesced access pattern, shown in Figure 2.1 from [116]: according to their index, consecutive threads in a warp should access consecutive memory locations. In the best case, 32 threads in a warp can read 32 4-byte words in a single coalesced 128-byte transaction [116]. Other memories need programmers to take care of other aspects to maximize bandwidth. For instance:

- shared memory accesses must be organized as to avoid bank conflicts;
- constant memory reaches maximum performance in broadcast accesses;
- texture memory takes advantage of 2D spatially local accesses;

Nowadays, the CUDA ecosystem features various libraries that help programmers in many situations: cuFFT [120], cuBLAS [118], cuRAND [121], Thrust [122], and so on. Thrust, in particular, is an attempt to bring the Standard Template Library (STL) formalism (containers, iterators, and algorithms) [149,152] into the CUDA ecosystem. It offers two containers: a **`host_vector`** and a **`device_vector`**. Users can manipulate data with the high-level formalism introduced in the STL, and thus simplify their code. Thrust, however, is limited in scope: with only a mono-dimensional container, many interesting problems that are not embarrassingly parallel or have a different grain of parallelism cannot be expressed. Moreover, the presence of a device and a host vector still force programmers to manage all the memory movements.

Despite the maturity of the CUDA ecosystem and the vast literature that focuses on various techniques to write efficient CUDA code [23,112,114,145], programming GPUs with CUDA is still challenging. The many architectural aspects that must be taken care of, and being quite low-level abstraction-wise too, may make other solutions preferable from the programmer's productivity point of view.

### 2.2.2 ROCm

Radeon Open Compute platform (ROCm) is a parallel computing environment intended for GPUs, released by AMD in 2016 [5]. It has a modular design that allows hardware vendors to release drivers that support its stack. It is not limited to a single solution, but it comprises various programming languages to choose.

It features four main software components:

- HCC;

- HIP;
- Anaconda;
- OpenCL.

Heterogeneous Compute Compiler (HCC) is an open source C++ compiler for heterogeneous devices. It can compile code written in Heterogeneous Compute (HC) format, a C++ dialect extended with facilities to invoke kernels and manage device memory [5]. However, HCC has been deprecated in June 2019 in favour of alternatives like HIP and OpenCL [5].

Heterogeneous-computing Interface for Portability (HIP) is a parallel programming language that can target AMD and NVIDIA GPUs. It has been designed to be as closer as possible to CUDA: by replacing calls to CUDA API with calls to HIP API, programmers can transform their code into a version that can target GPUs of both vendors. Table 2.3 shows some function, type, identifier, flag, and keyword correspondences between the two APIs [5, 12].

| CUDA | HIP |
|---|---|
| cudaMalloc | hipMalloc |
| cudaFree | hipFree |
| cudaMemcpy | hipMemcpy |
| cudaMemcpyHostToDevice | hipMemcpyHostToDevice |
| cudaMemcpyDeviceToHost | hipMemcpyDeviceToHost |
| cudaError_t | hipError_t |
| cudaEvent_t | hipEvent_t |
| threadIdx.x | hipThreadIdx_x |
| blockIdx.x | hipBlockIdx_x |
| blockDim.x | hipBlockDim_x |
| gridDim.x | hipGridDim_x |
| __device__ | __device__ |
| __host__ | __host__ |
| __global__ | __global__ |
| __shared__ | __shared__ |
| __constant__ | __constant__ |
| <<< ... >>> | hipLaunchKernel |
| __syncthreads | __syncthreads |

**Table 2.3:** Some correspondences between CUDA and HIP APIs.

The code transformation can even be automated with Hipify, a tool that is shipped with HIP [5]. It is able to perform much of the conversion and emit a log that notifies the user about the code portions it was not able to convert. For instance, since AMD cards cache textures in the L1 cache, Hipify is not able to convert CUDA code that explicitly takes advantage of the texture memory [5].

Similarly to CUDA, HIP is distributed with many libraries that automate application development in various contexts: rocFFT, rocBLAS, rocRAND, and so on [12].

Anaconda is an analytics platform that helps developing scientific computing and machine learning applications with Python (2.x and 3.x) and R on AMD devices like GPUs, CPUs, and APUs [5, 12].

OpenCL is an open-source royalty-free standard for heterogeneous programming. That included in ROCm is the implementation of the 1.2 specification for ROCm-enabled devices [81]. OpenCL will be analyzed in-depth in the next section.

## 2.3 Heterogeneous programming

Heterogeneous programming approaches permit writing code that is portable across different devices. Code portability is promising in terms of its impact on programmer's productivity: it permits writing, testing, debugging, and maintaining a single version of a program instead of one for each device. However, it is not guaranteed that this *single version* meets other possible constraints such as near-native performance on all the target devices or an *acceptable* programming effort. Depending on the characteristics of the adopted solution, for instance, the produced code could lack of *performance portability* across different devices or even across different generations of the same device [26]. Or, from another perspective, it could be so complex to partially cancel the advantages that its portability provides.

In general, it is possible to identify various desirable features that a heterogeneous programming approach should have. Those discussed so far can be summarized as:

1. code portability across different devices and different generations of the same device;
2. possibility to achieve near-native performance;
3. performance portability across different devices and different generations of the same device;
4. limited programming effort in terms of code complexity and abstraction level.

From a more fundamental point of view, heterogeneous programming solutions differ one another in the way they allow programmers to write parallel code for heterogeneous platforms. A possible classification is proposed in [131], and it is here slightly modified and reviewed. Three main categories can be identified:

1. Standalone framework;
2. Compiler-based;

3. High-level wrapper.

*Standalone frameworks* are comprehensive solutions that include various aspects. For instance, a Domain Specific Language (DSL) or an extension to an existing language with compiler, libraries, even drivers, and so on. They take care of all the aspects of heterogeneous development and are thus self-contained solutions. Interoperability with other approaches is not a fundamental aspect.

*Compiler-based solutions* rely on a compiler that is able to extract parallelism from the application code. The compiler is responsible for generating the low-level parallelization details, so that programmers are freed from taking care of them. The most notable solutions require that programmers annotate their code to give hints on how the code should be interpreted and what are the relations between the various components.

*High-level wrappers* are approaches (mainly libraries) that wrap one or more solutions lying in the first two categories. Their main purpose is to elevate the abstraction level by hiding some low-level details in the inner layers, those that directly communicate and interact with the wrapped approaches.

| **Standalone framework** | **Compiler-based** | **High-level wrapper** |
|---|---|---|
| OpenCL | OpenACC | SkePU |
| SYCL | OpenMP | SkelCL |
| | C++AMP | Kokkos |
| | | PHAST Library |

**Table 2.4:** Heterogeneous programming approaches. PHAST Library is the subject of this Ph.D. thesis work.

In the following, various heterogeneous programming solutions will be discussed. Table 2.4 shows how they can be classified according to the aforementioned categories.

### 2.3.1 OpenCL

OpenCL [81, 82] is an open standard for heterogeneous programming. When its design started, programmers were writing latency-focused programs for CPUs mainly in C and C++, and throughput-focused programs for GPUs in CUDA [73]. Apple proposed the creation of a new approach explicitly designed to support different devices like CPUs, GPUs, Field Programmable Gate Arrays (FPGAs), and more with a single abstraction. Various teams accepted the proposal, refined it, and in 2008 formalized it into the first version of Open Computing Language (OpenCL), which is currently developed and maintained by the Khronos Group [73].

The OpenCL specification has many similarities with CUDA, but also some fundamental differences. It is articulated in four main parts [73]:

- Platform model;
- Execution model;
- Kernel programming model;
- Memory model.

The OpenCL *platform* is a host connected to one or more devices. As in the CUDA model, a host is the coordinator that manages resources on the devices; devices are abstractions of processors with their separate memory spaces [95]. They are organized in compute units, which are further divided in processing elements. These abstract entities are mapped to physical resources in a way that depends on the vendor implementation of the OpenCL drivers [73]. The OpenCL specification provides C and C++ (from version 2.0 on) host-side API calls to query the set of available platforms, the associated devices, and their characteristics.

The execution model regulates how the parallel kernels are executed on the available devices. The central entity is the *context*, which is used to do many management activities: host-device coordination, memory object allocation/deallocation, program and kernel creation. Command-queues are used to send commands, i.e., data movements and kernel executions, to devices. These commands are executed asynchronously, and barrier synchronizations are used to wait for their completion. With version 2.0 of the standard [82], it is given also the possibility to send commands from the device execution space [73].

OpenCL kernels are analogous to CUDA kernels: they are functions that can be executed by many threads on the device in parallel. An important distinction with CUDA is that OpenCL kernel functions must be coded in extended C99 [65], instead of extended C++. Also in this case, the extension concerns the addition of a few keywords. Although OpenCL 2.2 brings the support for C++ kernel language and was finalized in 2016 [85], there are still no implementations available. OpenCL's concurrency model is also similar to CUDA's, but different terms are used: the kernel function is executed by work-items, which are grouped in work-groups, that are part of a bigger structure called N-Dimensional Range (NDRange). The correspondence of concepts is evident: work-items in a work-group can synchronize and access to a shared memory the same way as CUDA threads can do in CUDA blocks. NDRange is the OpenCL correspondent of a CUDA grid [73]. These correspondences are summarized in Table 2.5.

Finally, the OpenCL memory model defines its abstract objects that can be manipulated in code. The main one is the *buffer*, the equivalent of a C array, i.e., a collection of contiguous elements referred by a pointer, that can be accessed device-side. For their creation, they need a context to be associated with in order to map to the physical

| Framework | | | |
|---|---|---|---|
| CUDA | grid | block | thread |
| OpenCL | NDRange | work-group | work-item |
| SYCL | NDRange | work-group | work-item |
| OpenMP | league | team | thread |
| Kokkos | league | team | thread |

**Table 2.5:** Thread hierarchical organization terminology used in various programming frameworks.

memory of a particular device. Other examples are images, included in the standard as a native type, and pipes, special FIFO types that ease writing producer-consumer code [73]. Moreover, it distinguishes host memory from device memory, and divides the former into four regions [73]:

- global memory, which can be accessed by all the work-items and is the main target of memory transfers from the host;
- constant memory, designed for data that does not change during kernel execution;
- local memory, which is shared between work-items in a work-group;
- private memory, which is exclusive to an individual work-item.

OpenCL, in order to properly manage heterogeneity, demands to C++ or CUDA programmers an important deviation in their workflow. In OpenCL programs, the system is queried at runtime in order to discover what devices are available. Device vendors have to provide the implementation of a common runtime called Installable Client Driver (ICD) in order to make their devices discoverable [73]. After choosing the target architecture, the C99 kernel code, necessarily located in a different source file, is loaded as a character array, compiled with the OpenCL Just-In-Time (JIT) compiler into a program object, and then the program is used to obtain kernel objects. All these steps are performed at runtime, and any compilation error must be addressed during this phase. The particular binary representation produced while building the program is vendor specific and can vary a lot: it can be x86 instructions, AMD's GPU intermediate language, or NVIDIA's PTX instructions [73]. The kernel produced this way cannot be launched as a normal function, but needs API calls that manage the queuing of parameters and the execution of the kernel [73].

These aspects make OpenCL a very low level approach to program heterogeneity. It requires users to write C99 code, which lacks of many facilities of modern object oriented programming languages like RAII mechanisms, polymorphism, generic programming (e.g., C++ templates) or even function overloading; it is not single-source, since the

kernel functions must be read as character arrays; it requires *boiler-plate* code to query platforms and devices; it requires various instructions to build kernels at runtime and produce callable entities; it needs specific instructions to manage particular objects like contexts and command-queues. This low-levelness determines a programming style that hampers productivity, as OpenCL may require on average twice the lines of code of CUDA [102]. It produces portable, but not *performance portable* code [35, 50]: architecture-dependent fine-tuning is generally necessary to achieve high performance. Moreover, this performance is often inferior to what would have been achieved with native approaches [42, 89].

### 2.3.2 OpenMP and OpenACC

Some heterogeneous approaches prefer to let programmers specify parallelization details by the means of annotations. These take the role of a co-language that interleaves with the main programming language instructions, usually C or C++. It is the case of OpenMP [27] and OpenACC [123].

OpenMP was released in 1998 as a pioneering shared-memory programming standard, as opposed to the widespread Message Passing Interface (MPI) that dominated in those days [27]. It is "a set of compiler directives and callable runtime library routines" [27] that extends Fortran, C, and C++. It defines an API that is portable across architectures and is implemented in many compilers by different vendors.

OpenMP is based on *directives* given to the compiler. They can be declarative, that contain declarations only, or executable, that can also cause execution of user code. In the case of C and C++, directives are preprocessor instructions introduced by **`pragma omp`** that precede the statement, loop, or structured block they refer to. These are generally called *regions*. This code is what will be executed in parallel by a *team* of OpenMP threads whose number is determined at runtime [126]. With version 4.0 OpenMP made its transition to heterogeneity, with the addition of the possibility to offload code towards target devices like GPUs [125].

OpenMP begun supporting accelerators taking inspiration from OpenACC [153], which in turn was inspired by OpenMP itself, as its citation in OpenACC 1.0 API introduction shows [123]. OpenACC also supports C, C++, and Fortran and can target accelerators – even multi-core CPUs. The accelerator support is included from its first release in 2011 [123], and in fact a lot of emphasis is given to the host and device concepts. In this context, the main program is executed by a host-directed thread that orchestrates the offloading of parallel, kernel, or serial regions to an accelerator device [124]. OpenACC directives are different from OpenMP directives: they are marked by **`pragma acc`**, and the syntax is generally different. However, some research has been done to translate one formalism into the other [138, 153].

Both approaches have the advantage of allowing the parallelization of sequential applications with a reduced effort. Their nature of directive-based co-languages permits avoiding code re-writings and can be appealing to the scientific community that needs to port large legacy codebases to modern parallel architectures. In general, these solutions require fewer lines of code than competitors like CUDA or OpenCL [102]. However, these models usually work well with for loops, but are not able to parallelize C++ programs that use higher-level abstractions such as STL [149, 152] or user-defined libraries [96]. This aspect alone can be a limitation for programmers willing to parallelize pre-existing high-level code, and also for those willing to write new heterogeneous code with a high-level of abstraction.

### 2.3.3 SYCL

SYCL [84] is an open standard for heterogeneous programming developed and maintained by the Khronos Group. It builds on top of OpenCL concepts and its focus is to ease the programmability of heterogeneous architectures. Unlike OpenCL, it is a full modern C++ single-source solution that moves part of the complexity to the compilation process, defined a Single-source Multiple Compiler Passes (SMCP) design: host and device code are not physically separated, they are just compiled by different compilers in different passes [84].

Like OpenCL, SYCL can be fully described in terms of four aspects [84]:

- Platform model;
- Execution model;
- SYCL programming model;
- Memory model.

SYCL platform model is similar to OpenCL's: a host is connected to one or more OpenCL devices, to which it sends commands. It can search devices and retrieve information about their capabilities, such as resource limits or functionalities [84].

The execution model describes a SYCL application that executes on the host and launches kernels on the device. Kernels are created and submitted to a command queue together with their requirements in the form of *command group* objects. These requirements can concern the end of a previous computation or the access to some data. Access requirements are expressed through *accessors*, that are declared specifying the data buffer to access and also the access type: read-only, write-only, or read-write. Their declaration determines the construction of an implicit Directed Acyclic Graph (DAG) of dependencies that impacts the execution order of the kernels. Finally, inside them

the familiar concepts of work-item, work-group, and nd-range determine the thread indexing [84].

The SYCL programming model describes the conforming SYCL programs, which are C++ programs that mix host and device code. Kernels are declared as modern C++ function objects or lambdas. They can take advantage of many features of the language such as variadic templates and polymorphism, but not all of them: exceptions, virtual functions, function pointers, etc. are not supported, which are basically the same limitations imposed by the CUDA programming model [84, 119]. The host queues kernel executions up with **`parallel_for`** or **`single_task`** functions. There is also the possibility to take advantage of hierarchical parallelism by invoking nested parallel for loops [84].

Finally, the memory model can be split in application and device memory model. The former describes the memory objects (**`buffer`** and **`image`**) declared host-side and the accessors that allow access device-side. The latter specifies the same memory regions as OpenCL (global, constant, local, and private) which also retain the same meaning.

The Khronos Group provides the SYCL specification as an open standard, so to encourage implementations by different subjects. These can follow different criteria, as long as they conform to the specification. A list updated to March 2020 counts five SYCL implementations, not all of them complete [53]:

- triSYCL;
- ComputeCpp;
- hipSYCL;
- sycl-gtx;
- Intel oneAPI.

triSYCL [144] is an open-source implementation based on OpenMP, Threading Building Blocks (TBB) [142], and OpenCL mainly meant to let users experiment and send feedback about the SYCL standard. Codeplay's ComputeCpp [21] is compatible with architectures supporting SPIR-V intermediate language, which is required to implement the OpenCL 2.1 specification [83]. hipSYCL [2] wraps OpenMP for multi-core CPUs, CUDA for NVIDIA GPUs, and AMD HIP for AMD GPUs. sycl-gtx [137] is built on top of OpenCL 1.2, and thus can target the majority of heterogeneous devices. Intel oneAPI [64] is an ecosystem that comprehends a language, Data-Parallel C++ (DPC++), which supports the C++17 Core Language [69], and many libraries meant to create parallel applications.

SYCL is now a growing ecosystem. The recent appearance of Intel oneAPI seems to suggest that it is going to be the *de facto* standard for heterogeneous programming

as OpenCL has been in the recent years. SYCL alone is a powerful abstraction, but it needs libraries on top of it, such as SYCL Parallel STL [80], to improve programmers' productivity, especially in those cases that are not easily mappable on a parallel-for computation.

A limitation in terms of abstraction offered to the programmer is that, device-side, the framework does not hide the multi-threaded nature of the code: in kernel code, it is still necessary to access data depending on thread index, to reason about work-group size, to explicitly use a local memory with explicit synchronization primitives to share data among threads, and so on. In SYCL code, the programming patterns that gained popularity with CUDA and OpenCL are still there. This aspect is not necessarily a downside for programmers with heterogeneous/GPGPU background, but it may slow down or even prevent the adoption of this model by programmers without such a background. Also, it could hamper performance portability, as it will be shown with the DCT8x8 benchmark in Subsection 4.1.2.

### 2.3.4 C++ AMP

C++ Accelerated Massive Parallelism (C++ AMP) is a programming model designed by Microsoft to program data-parallel devices such as GPUs. It provides an annotation, some classes, and some functions that together permit writing code for accelerators [49, 105].

The annotation **`restrict(amp)`** can be used to annotate callables to inform the compiler that their execution can be done on an accelerator. The compiler checks that the usual restrictions that apply to C++ GPU code are met and, if so, it generates an executable kernel [49]. The C++ AMP classes include the **`accelerator`** class, that represents the device of execution, and other classes like multi-dimensional arrays and indexes that regulate data access. Kernel execution is launched by calling the **`parallel_for_each`** construct that takes an *extent* argument that determines the thread creation and an annotated callable that is compiled into a kernel and executed [105].

C++ AMP is released as an open specification. Although it is implemented in Visual Studio compilers, its release as a specification allows for other implementations, such as Intel Shevlin Park, based on OpenCL [147].

C++ AMP can be a rapid solution to parallelize for loops and offload execution on an accelerator. Doing so requires minimum changes to existing code. However, since it only provides **`parallel_for_each`** as a construct to build heterogeneous applications, it is rather limited and needs some higher abstractions to implement common patterns with a reduced effort (e.g., reduction).

### 2.3.5 SkePU and SkelCL

SkePU [41] and SkelCL [151] are skeleton-based approaches. Skeletons are pre-defined programming patterns [151] that express common computations with a high level of abstraction. They can be composed and combined in a functional programming fashion to express complex applications.

SkelCL wraps OpenCL as heterogeneous framework. It defines the **`Vector`** and **`Matrix`** classes that can point to the host and device memory regions and offers access from both execution spaces [150]. The computation is performed by applying various skeletons included in the library: **`Map`**, **`MapOverlap`**, **`Zip`**, **`Reduce`**, **`AllPairs`**, and **`Scan`** [150]. They are higher-order functions that need user-defined computations to be specified. These are passed as string arguments to the skeleton and compiled with the runtime compilation capabilities of OpenCL. SkelCL supports also multi-GPU programming by distributing data to the available GPUs in the system [151].

SkePU is built on top of CUDA, OpenCL, and OpenMP. The backend can be selected by defining the macros **`SKEPU_CUDA`**, **`SKEPU_OPENCL`**, and **`SKEPU_OPENMP`**, respectively. The only container defined is **`vector`**, which manages memory on both host and device and returns proxy elements on access that implement a lazy copy logic. The library provides five different skeletons in the form of functors: **`Map`**, **`Reduce`**, **`MapReduce`**, **`MapArray`**, and **`MapOverlay`**. User-defined functions can be declared with the help of macros that permit defining unary, binary, and ternary operations. SkePU also supports multi-GPU by distributing its vectors among the GPUs in the system [41].

Both these approaches are very similar in scope and realization. They offer higher order functional-like abstractions that could help formalize various kinds of data-parallel applications. There can also be ones that are not easy (or even not possible) to express using this formalism, which may need some effort to be mastered by programmers used to imperative languages. Finally, these abstractions are limited to the host-side code: because of their adoption of an OpenCL backend, the code that will be executed on the device must be expressed in C99.

### 2.3.6 Kokkos

Kokkos [19, 37, 38] is a high-level programming library that targets heterogeneous devices through various backends. Currently, it supports OpenMP [126], PThreads [113], and HPX [74] for multi-core CPUs, and CUDA [119] for NVIDIA GPUs.

Kokkos provides multi-dimensional arrays as C++ **`View`** classes. They can be instantiated by specifying the data type and, for each dimension, a compile-time expression if it is static or an asterisk if it is dynamic. The data-layout is flexible and it can be decided

at compile-time to accommodate various computation needs. **`View`**s are designed not to hide memory copies, so object copies are treated as shallow copies, instead of deep copies. Data deallocation is managed with reference counting, with a semantics that is similar to the C++11 **`shared_ptr`** semantics [38, 66].

Kokkos supports both data-parallel and task-parallel execution patterns [163]. Data-parallel execution can be performed by three functions that correspond to different execution patterns: **`parallel_for`**, **`parallel_reduce`**, and **`parallel_scan`**. **`parallel_for`** allocates a specified number of threads and applies the same computation on all the elements of a view, possibly in parallel. **`parallel_reduce`** is similar to **`parallel_for`**, but the results of the single computations are aggregated into a scalar element with an operation that has the requirement of associativity. The last, **`parallel_scan`**, produces as many elements as input values, each being the result of a partial reduction [37, 163]. Task-parallel computations are also possible, but they have a limited scope and other solutions offered by the framework are usually a better choice [163].

The computations are expressed in terms of functors (function objects) or lambdas. Since these define a callable that will be executed on the device, in order to work with the CUDA backend they must be recognized by the CUDA compiler as a device function or a kernel, i.e., they must be annotated with the **`__device__`** keyword or the **`__global__`** keyword, respectively. The solution Kokkos proposes is to annotate them with the **`KOKKOS_INLINE_FUNCTION`** that expands differently according to the selected backend [163].

The thread generation can be done with different strategies. Threads can be generated as a collection of "sibling" threads at the same level or as a *league* of threads organized in independent *teams*. League and team are terms defined in OpenMP [126] to indicate the canonical CUDA concepts of grid and block summarized in Table 2.5. Threads in the same team can share data via a "scratch pad" memory and synchronize with explicit synchronization primitives [163].

The Kokkos programming model is a powerful programming model that values expressiveness. However, like other models, it also requires programmers to be familiar with the heterogeneous/GPGPU programming style that requires the management and coordination of blocks of indexed threads. So, in the Kokkos case, the same considerations made for SYCL are still valid, included the impact of this programming style on performance portability that will be further clarified in the discussion of DCT8x8 in Subsection 4.1.2.

### 2.3.7 Other approaches

Apart from the frameworks and libraries mentioned above and analyzed in-depth, there are others that, for their specificity or limited scope have not been fully covered.

In the spirit of high-level wrappers, various approaches focus on offering a thin layer with limited capabilities around a more established approach, trying to smooth some corners and improve programmability. Some examples are EasyCL [136], that adds a reduced abstraction layer on top of OpenCL, and VexCL [31], that helps writing vector expressions in OpenCL and CUDA.

Others aim at improving programmability in specific contexts. For example, StarPU [11] proposes a runtime system that manages the scheduling of tasks in a heterogeneous system. These tasks can be declared to include various versions of the same function, implemented with a variety of approaches. Halide [141] is a programming language that helps writing heterogeneous code for image and array processing. ArrayFire [173] is an array-centric C++ library with a focus on matrix manipulation, image filtering, and mathematical computations.

Other solutions are intended to improve programmability in multi-node systems, generally with a High Performance Computing (HPC) focus. Many enrich existing approaches to target distributed systems: OmpSs [16] is an extension to OpenMP to support heterogeneous devices and clusters, libWater [47] extends the OpenCL runtime and programming model to distributed systems, Celerity [159] extends the SYCL programming model to accelerator clusters, StarPU-MPI [10] extends StarPU to clusters of heterogeneous nodes. There are also extensions to existing languages, like Charm++ [75], that extends C++ and provides a distributed runtime system.

### 2.3.8 Critical discussion

The analysis of the state-of-the-art so far highlights that different approaches have different strong and weak points, but none of them can be considered as the perfect fulfillment of all the requirements listed at the beginning of the section, that can be summarized as code portability, near-native performance, performance portability, and productivity. For sure, all the heterogeneous solutions meet the first requirement, but the same cannot be said for the other three. In fact, they may conflict with each other, and trying to design a framework to invest in one aspect can result in the lowering of another. For instance, the programming effort and code complexity can be limited with the adoption of high-level constructs and additional layers, but these may hide too many fundamental details of the underlying hardware and consequently limit performance. On the contrary, the exposition of fine-grained details would promote a *close-to-the-metal* programming style that may allow reaching peak performance, but at the expense of the level of abstraction and code complexity. Also, it could lead to over-tailored code that would limit performance portability.

Another important aspect is that programmer's productivity is also affected by two things: the programmer's background and the context in which they operate.

On the background side, for instance, programmers coming from GPGPU will have no difficulties to understand the role of thread blocks/work-groups and shared/local memory in frameworks such as Kokkos [37] and SYCL [84]. For programmers without such a background, however, being productive with these frameworks may be difficult and would need them to learn new concepts first. A similar reasoning may be done for SkelCL [151] and SkePU [41]: their skeleton-based approach may be attractive for programmers with functional programming experience, but it may need a mindset shift for the others.

On the context side, an example would be that of programmers dealing with refactoring of legacy code. Probably, they would find the OpenMP [27] and OpenACC [123] way an appealing solution for their task, but the same evaluations may not apply if they have to start a new project from scratch.

What emerges from this discussion is that there is still research to do in the heterogeneous programming field, as the *definitive* solution is yet to come. In the following chapter, it is presented the contribution of the author in form of a heterogeneous programming framework, designed and realized with the requirements listed above clear in mind.

CHAPTER 3

# PHAST Library

This chapter and the next are a summa of the contribution of the author in the field of heterogeneous programming. The majority of the topics here presented have already been discussed in various papers [130–133, 135]. In this case, however, the more relaxed space constraints compared to a scientific paper permit going deeper into the motivations that led to precise design choices and their implementation.

## 3.1 Design principles

Parallel Heterogeneous-Architecture STL-like Template (PHAST) Library is a modern-C++ library for *single-source* high-productivity programming of heterogeneous parallel architectures. It currently supports multi-core CPUs and NVIDIA GPUs. It offers an STL-like interface based on containers, iterators, algorithms, and functors [152]. All the phases of its realization have been guided by the desiderata expressed in the previous chapter. Namely:

**Portability**

The inner layers of the library are implemented in terms of modern C++ **`std::thread`**s [66, 68, 69], to run on multi-core CPUs, and in CUDA C++ [119], to run on NVIDIA GPUs. The implementation, and thus the platform of execution, can be statically selected at

compile-time via a single globally-defined macro[1] that can be put into the Makefile. Thus, the same application code is portable: it can be built for different architectures without the need to modify it.

### Performance

In the design and implementation of PHAST Library, the layers have been kept as thin as possible to limit the performance impact of the abstractions adopted. Various computations have been formalized in algorithms, which have been made available to the programmer as a target-architecture agnostic programming interface. In their underlying implementation, they are backed by specific tailored specializations for each architecture, expressed in native approaches, which have been chosen to have access to the best performance the hardware has to offer [42, 89, 102]. During their design, the state-of-the-art techniques have been researched.

### Performance portability

Performance portability is usually hampered by the need to fine-tune an application for different architectures, with different optimal data access patterns and unique micro-architectural features [26]. The resulting implementation is tailored on a single device, and thus can have poor performance on a different one. In PHAST Library, the fine-tuning can be done by the means of parallelization parameters that allow separation of application and tuning code. They can be used to set at runtime many parameters that drive code execution adjusting the behaviour of the architecture-specific implementations in the inner layers of the library. This also enables high flexibility to adapt to a variety of conditions and constraints (e.g., power operation modes, performance levels, different instances of target machines).

### Productivity

The main goal of PHAST Library is to raise the level of abstraction in heterogeneous programming via expressive high-level constructs. Containers are used to store data, offering a richer interface than arrays and buffers. They can be visited via iterators or accessed directly by index to reach individual elements as well as convenient container slices (e.g., matrix rows). Complex calculations can be expressed with algorithms, that operate on ranges of iterators and provide common computations and recurring programming patterns in the form of single functions. Functors execute on device and are used in various algorithms that offer a degree of freedom to the programmer (e.g., a predicate indicating which iterated elements to consider in the algorithm computation, like in a

[1] In its current state, `__PHAST_USING_MULTI_CORE` or `__PHAST_USING_CUDA`.

`replace_if`). Their implementation bridges the expressiveness gap that usually exists between host and device code: after launching one of these algorithms on the host, data is partitioned and processed in functor code, that executes on device. Here, even the iterated partitions are presented as containers that can be accessed, iterated, and manipulated with a high-level formalism, including device-side algorithms. In other words, the data manipulated device-side have the same level of abstraction that can be found host-side.

PHAST Library has been designed in order to elevate the level of abstraction in heterogeneous data-parallel programming with respect to the state-of-the-art. It allows programmers to define high-level containers of various dimensionalities where data can be mapped conveniently, with the shape of the container that reflects the shape of the data. Containers can be accessed, processed, managed and, most importantly, iterated in *sections* of various shapes that can be processed in parallel, independently of each other, on various architectures with high-level, special algorithms. These abstract away the nature of the container to concentrate on the section they work on and the semantics of the operations that must be performed. Each operation, then, can be expressed with a high-level formalism, as the sections themselves are represented as containers with their own shape that can be dissected again into lower-level algorithms, analogous to their upper-level counterparts in expressiveness, abstraction level, and semantics. Moreover, the execution of the algorithms from both layers can be finely-tuned in order to intercept the best performance on each architecture of interest. Tuning, unlike most competitor approaches, is not invasive and can be done safely by setting some parallelization parameters outside of the code where the main logic is implemented. This whole organization and separation of concerns allows a decoupling between the data allocation, the semantics of the operations that must be performed on those data, the distribution of the work to the available hardware resources, and even its tuning and optimization.

PHAST Library's programmers can express their algorithmic semantics with a high level of abstraction, specifying their operations on multi-dimensional containers in a sequential-like fashion; relying on special algorithms that can process ranges of scalar and non-scalar sections, in a way that is agnostic of both the underlying device of execution and the nature of the iterated container. In this regard, PHAST Library represents a discontinuity with respect to the competitor approaches: these prefer iterations on sets of thread indexes, using those indexes to explicitly retrieve data from global array-like structures in device code, requiring programmers to write the code that translates thread indexes into data-structure indexes. In PHAST Library, programmers express their computations by specifying only the shape of the data they want to process (i.e., the container section), the operations to do on those data, and the range of sections

to process. The concept itself of thread can be left outside of PHAST Library code, since the code only takes care of the data structure, its sections and the operations that should be performed on them. The way these operations are expressed is the biggest improvement that PHAST Library brings to the heterogeneous programming field. Here lies its novelty and the single most important contribution of this Ph.D. thesis work.

To give a quick example of the novelty of the approach, the following code can be taken into consideration:

```
template <typename T, unsigned int policy =
    phast::get_default_policy()>
struct func : phast::functor::func_mat_scal<T, policy>
{
    _PHAST_METHOD void operator()(
        const phast::functor::matrix<T>& mat,
        phast::functor::scalar<T>& scal)
    {
        scal = this->accumulate(mat.begin_ij(), mat.end_ij()) /
            (mat.size_i() * mat.size_j());
    }
};
```

**Listing 3.1:** Example of a binary functor.

The code above describes a functor that calculates the average of all the elements of a matrix section and puts that value into a scalar element. This code can be executed on a device (multi-core CPU or NVIDIA GPU) in parallel, processing an unspecified number of matrix-scalar pairs. The processing it applies to the matrix is expressed with a high-level algorithm that iterates on the range of iterators of the matrix and performs the accumulation of the pointed elements. Its implementation is different on multi-core and NVIDIA GPU, with the latter that possibly takes advantage of a second axis of parallelism. Inside the body of the functor, no mention of threads or thread-indexes is done, and most importantly no mention of the two source containers that provide the matrix-scalar pairs. In fact, the same functor can be used in host-code in a variety of situations:

```
phast::grid<phast::matrix<type>> gr(mat, 4, 4);
phast::vector<type> avg1(gr.size_i() * gr.size_j());
phast::for_each(gr.begin(), gr.end(), avg1.begin(), func<type>());

phast::vector<type> avg2(cube.size_i());
phast::for_each(cube.begin_i(), cube.end_i(), avg2.begin(), func<type>());
```

In lines 1-3, a matrix object is partitioned in 4x4 sections and, for each section, the average element is calculated and saved into a vector that has as many elements as the number of 4x4 sections in the matrix. The section size can be changed at line 1 without modifying any of the rest of the code. This code could be used to divide an image matrix

into 4x4 blocks and calculate the average pixel value for each block. The size of the block can be easily adapted.

In lines 5-6, a cube object is partitioned in its faces and, for each face, the average element is calculated and saved into a vector that has as many elements as the cube has faces. This code could be used to calculate the average pixel of a set of frames from a video, where each face of the cube represents a frame.

In the code above, also the data type has been voluntarily left unspecified and a **`type`** placeholder has been used instead: the pixel type could be modeled as a **`float`** (range 0-1) or an **`unsigned int`** (range 0-255) for greyscale images/videos, or modeled with a PHAST Library vector type **`float3_t`** or **`uint3_t`** to store all the three RGB channels.

The code above could be slightly tuned with the use of parallelization parameters:

```
1 phast::custom::cuda::set_minor_block_size(4);
```

Since an algorithm is invoked inside the functor, this parameter set tells the library to allocate CUDA blocks with 4 threads on the minor axis, so to execute the accumulation in parallel with the help of 4 threads for each section. In this case, the 4 threads fetch one element each inside the **`accumulate`**, instead of letting one thread cycle over them.

The code here discussed presents various abstractions of PHAST Library: containers, iterators, algorithms, functors, hierarchical parallelism, and parallelization parameters. All these aspects will be discussed in this chapter in the following sections.

Currently, PHAST Library has been tested on the most common operating systems of the Linux family, multi-core CPUs of Intel, AMD, and ARM vendors, and NVIDIA GeForce desktop GPUs of the last three generations, plus the NVIDIA Jetson TX2. Its website [134] can be considered as the official source of information about it, with some simple examples and the development roadmap. There is also the possibility to register freely and download both a trial implementation of the library and the documentation, that currently has 166 pages.

## 3.2 Containers

PHAST Library containers are template classes that store collections of objects, with the common semantics introduced in the STL [149, 152]. There are three kinds of containers: **`vector`**, **`matrix`**, and **`cube`** (a generic *parallelepiped* of values), depicted in Figure 3.1. **`vector`** is the PHAST equivalent of the classic STL vector: it stores its data contiguously in memory and offers iterators and access methods that suggest a mono-dimensional layout. **`matrix`** and **`cube`** are the 2D and 3D generalizations of the vector concept. They also have contiguously stored data, but the associated iterators and access methods permit 2D and 3D indexing. They have been defined to allow programmers to express computations on entities that present a multi-dimensional structure in a more convenient

way, without the need to adapt them to 1D data structures and introduce the *noise* in the application code for managing the higher semantic dimensionality of the data structure within the mono-dimensionality of its programming handle (i.e., vectors).

It is crucial to provide the programmer with *native* data structures that can naturally and conveniently match the layout and dimensionality of application-level data. Multi-dimensional data-structures are a broad topic and the examples of applications that map well on them are abundant in literature [3, 28, 30, 34, 36, 46, 54, 91, 94, 98, 143, 166, 168, 172].

These kinds of data structures are a common feature in heterogeneous programming: for instance, C++AMP [49] and Kokkos [37] offer multi-dimensional arrays, SYCL [84] offers one-, bi-, and three-dimensional buffers, and SkelCL [151] offers **`Vector`** and **`Matrix`** classes.

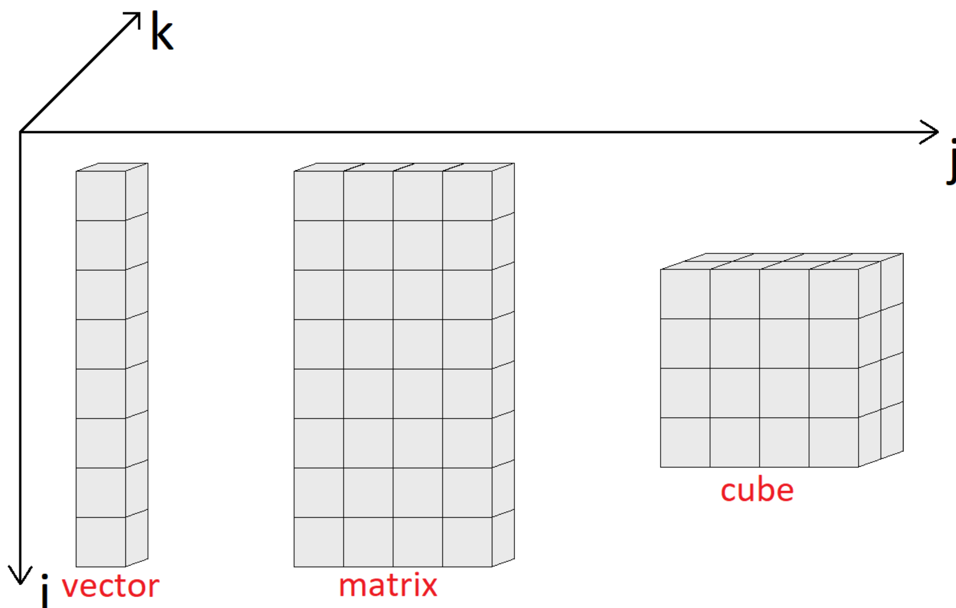



**Figure 3.1:** PHAST Library containers and the coordinate system adopted. From left to right: **`vector`**, **`matrix`**, and **`cube`**.

The data stored inside these containers is physically located on the device memory. In the multi-core CPU case, device and host coincide, and thus data is stored in the CPU RAM and is directly accessible from any part of the program. In the CUDA case, conversely, data is stored in the global memory. In order to give access to it from host code, there are two possible solutions:

- Taking advantage of CUDA Unified Memory Access (UMA) and let CUDA manage any data transfers;
- Not using UMA and managing data transfer in the library code.

In [93] Landaverde et al. study the impact of UMA on performance. They show that for all the considered dimensions but the smallest ones, the non-UMA approach performs better in the implemented benchmarks. Also, they observe that "the number of lines saved between the two versions of code (UMA vs non-UMA) is in the single digits for these benchmarks" [93], and thus that the performance loss of UMA is not balanced by a significant productivity improvement. For this reason, in the CUDA implementation of the PHAST containers, data is stored in the global memory in a non-UMA fashion.

Data can be accessed by invoking access methods on the containers (basically, **`at()`** and **`operator[]`**) or by dereferencing *scalar* iterators. The solution adopted in PHAST Library is similar to that adopted in SkePU [41]: instead of returning a direct reference to the data, a proxy object [45] is returned. In the SkePU case, this object is used to achieve lazy copying and retrieve data only when needed [41]. In PHAST case, data is loaded immediately and the proxy object mainly implements operator overloads to propagate any data modifications to the device memory.

In order to return to the user the right value, some coherency logic has been implemented. Inside containers there are two pointers: a device pointer, pointing to the data stored in the device, and a host pointer, pointing to a mirroring of the device data in the host memory that is not necessarily updated. If the container is updated, and thus the two pointers refer to coherent data, accessing an element in the container triggers the retrieval from the host pointer. Conversely, if the container is not updated, a **`cudaMemcpy`** of the single accessed element is invoked under the hood to retrieve it.

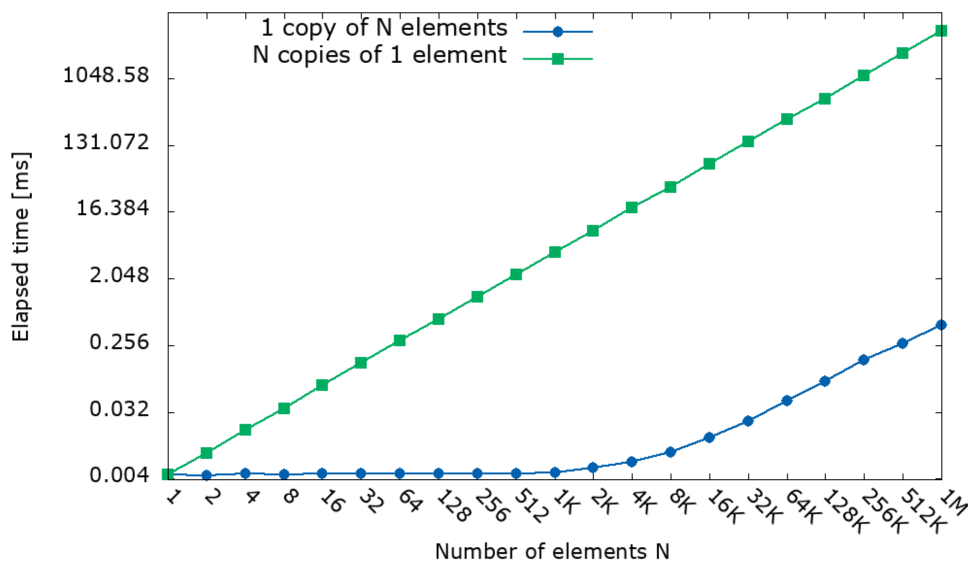



**Figure 3.2:** Comparison between **`cudaMemcpy`** elapsed times: one copy of N elements VS N copies of one element. The elements are 4 byte **`float`** values.

Since only the accessed element is copied, subsequent accesses to many elements

can degrade performance with unnecessary copies. For instance, a programmer willing to print the content of a container that stores $N$ elements, by accessing the elements one at a time invokes $N$ device-to-host copies of a single element. Figure 3.2 shows the "1 copy of N elements VS N copies of 1 element" elapsed time comparison. It is measured on an NVIDIA GPU GeForce RTX 2080 with 2944 CUDA cores clocking at 1.71 GHz maximum frequency and 8 GByte DDR6 RAM [117] and CUDA version 10.1. Elements there are 4 byte **`float`** values. It is not surprising that a single copy of $N$ elements is always more convenient than $N$ copies of 1 element. However, especially for large arrays, copying the whole data can be less convenient than copying only a few selected elements. For this reason, one access causes a copy of one element, but it is also given the possibility to copy the whole array in one step through the **`update()`** method, which is similar to SkePU's **`flush()`** [41]. It is up to the programmer to evaluate their access pattern and choose whether an **`update()`** can be convenient or not.

It must be noted that the aforementioned data transfers (implicit or explicit) are necessary only when data is read on the host. For instance, if two algorithms are launched in sequence on the same container, it is guaranteed that the second begins after the first finishes, processing the data that have been modified by the first algorithm. This happens because data normally stay on the device, they are accessed and modified coherently and no unnecessary transfers are performed.

When a container is updated, it remains so unless a modifying algorithm (e.g., **`fill`**, **`for_each`**, **`replace`**, and so on) is invoked with a range of iterators obtained from that container. Conversely, when a single value is modified invoking a modifying operator on a proxy element returned by an access method (e.g., $=$, $+=$, $-=$, $\ldots$), both the values on the host and the device are updated, so the two copies are kept coherent.

Since containers exist in a three-dimensional space, the library defines a coordinate system to locate and refer elements inside them. In Figure 3.1, such a system is shown. The axes are labeled **`i`**, **`j`**, and **`k`**, with **`i`** being the main axis. This nomenclature is consistent between the various parts of the library, and many objects and functions are named according to it. Figure 3.1 also shows how elements are contiguously stored in memory. Considering the **`cube`**, they are first arranged on the **`k`** axis, then **`j`**, then **`i`**. From another point of view, the storing order is the same as the lexicographic order that would be obtained by associating the label $(I_i, I_j, I_k)$ to each element, where $I_i$, $I_j$, and $I_k$ are the indexes on the **`i`**, **`j`**, and **`k`** axis, respectively.

As data types that can be stored in the containers, PHAST Library supports primitive types, vector types, and user-defined structures that meet the limitations imposed to device code in the CUDA programming model [119]. Vector types closely resemble CUDA vector types [119], but are richer than them for two reasons:

- PHAST types can be mapped on SSE types if available, general purpose registers

otherwise;

- PHAST types are shipped with arithmetic operator overloads and constructors.

Users can decide via a single globally-defined macro (`_PHAST_USING_SSE_TYPES`) whether or not the vector types should be mapped on SSE registers, if available. For instance, if the macro is set, `uint4_t` variables could wrap an `__m128i` variable on SSE-capable multi-core CPUs [157] or four `unsigned int`s variables on non-SSE-capable devices, like NVIDIA GPUs. Underlying operations would be invoked with SSE intrinsics. This way, PHAST programmers can express their computations on vector types via a natural arithmetic-resembling syntax, as they would do with primitive types, also taking advantage of the performance that special purpose hardware registers can bring.

Code automatic vectorization is a hot topic with a lot of research on compiler techniques [39, 77, 155] and the use of intrinsics can prevent it. However, in some cases relying on vectorization by hand can lead to better performance [90, 167]. For this reason, the choice of mapping PHAST vector types to SIMD registers or not is left to the programmer. The convenience of one solution over the other likely depends on the usage pattern.

## 3.3 Iterators

Iterators are a concept introduced in the STL [149, 152]. They are classes that *iterate* through the elements of a container, and can be considered as the generalization of pointers. They have in common with them the possibility of being dereferenced and the existence of an arithmetic that permit movements and calculation of reciprocal distances. In addition, they are C++ objects that expose many methods to change or query their state. Conceptually, they allow decoupling the structure of containers from the algorithms that need to *access* (i.e., read/write) their elements to perform a computation. This way, the algorithm *knows* only iterators and thus can neglect the specific details of the referred container. Stroustrup opens the Iterator chapter in [152] with a quote by Alex Stepanov, the inventor of the STL [149], that appropriately summarizes this separation of concerns: "the reason that STL containers and algorithms work so well together is that they know nothing of each other" [152].

This facet has been leveraged in PHAST Library: containers provide iterators that can be used in ranges inside algorithms; these do not know the nature of the iterated containers, but only their iterators. Moreover, algorithms also decouple the algorithmic semantics from the physical structure of the underlying architecture.

In PHAST Library, all the iterators have *random access* capabilities [152], and thus they can jump to elements at an arbitrary distance inside their range. They are listed in

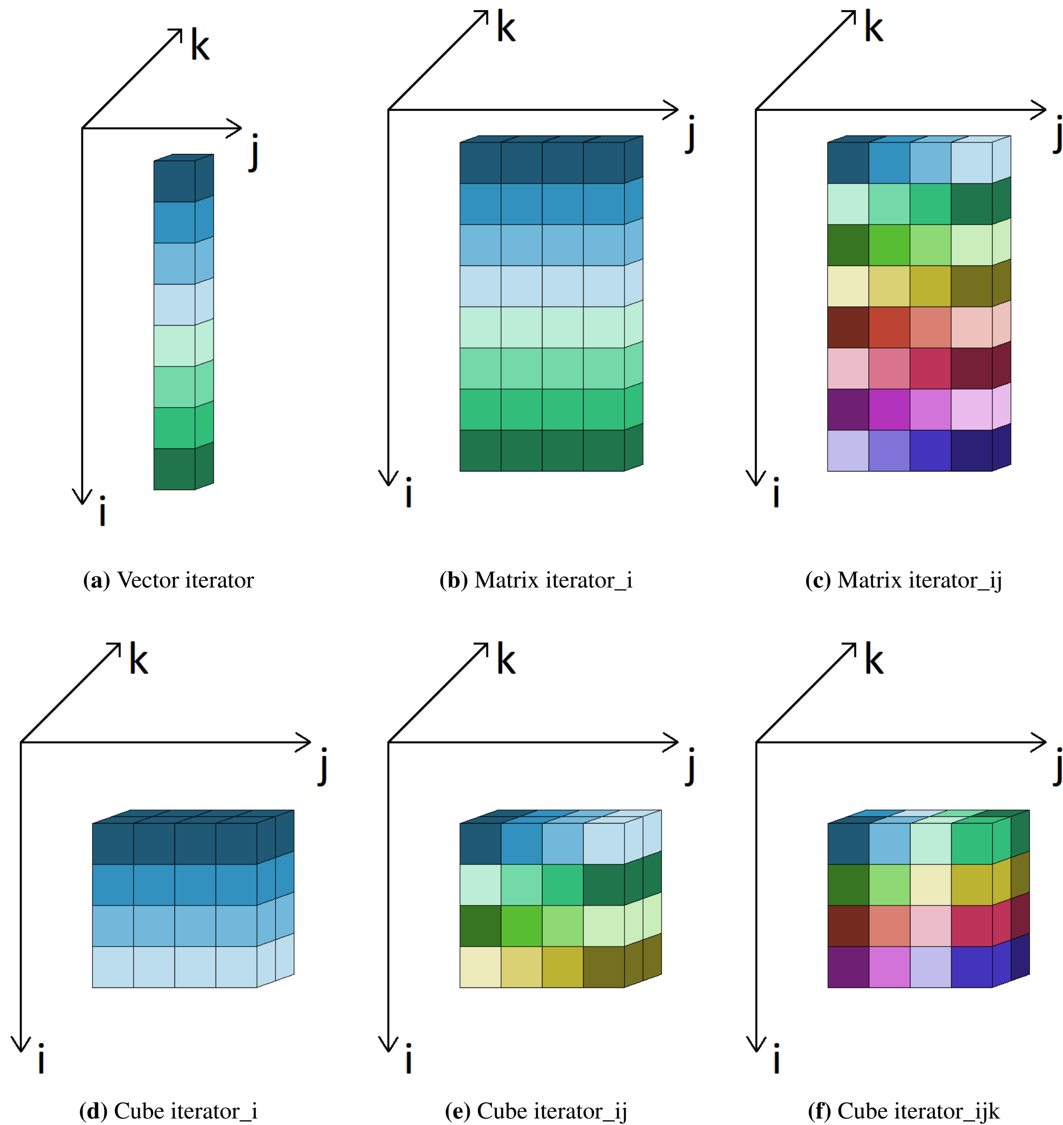
| Container | Iterator | Pointed element |
|---|---|---|
| vector | iterator | scalar |
| | const_iterator | scalar |
| matrix | iterator_i | vector |
| | const_iterator_i | vector |
| | iterator_ij | scalar |
| | const_iterator_ij | scalar |
| cube | iterator_i | matrix |
| | const_iterator_i | matrix |
| | iterator_ij | vector |
| | const_iterator_ij | vector |
| | iterator_ijk | scalar |
| | const_iterator_ijk | scalar |
| grid<vector> | iterator | vector |
| | const_iterator | vector |
| grid<matrix> | iterator | matrix |
| | const_iterator | matrix |
| grid<cube> | iterator | cube |
| | const_iterator | cube |

**Table 3.1:** List of PHAST Library iterators.

Table 3.1.

The term *element* has been used in a broad sense to denote where iterators point to. It is accurate in the case of the STL, since all the iterators point to individual, scalar elements inside the containers [152]. In PHAST Library, iterators can also point to *sections* of containers with specific topological properties: sections can be scalar elements, but also vectors, matrices, or cubes. They can be considered as slices, parts of the container with a specific shape, as sub-vectors in a **`vector`**, rows of a **`matrix`**, faces of a **`cube`**, and so on. Iterators can be classified according to the container they belong to, but more importantly according to the *shape* of the sections they point to. This particular aspect will be further discussed in the next section.

Figure 3.3 shows all the iterators that can be obtained from the three PHAST Library containers and the sections they refer to. These have the property of being contiguous in memory. Those obtained from **`matrix`** and **`cube`** objects have been named after the axes along which they are indexed, or, equivalently, the axes on which they can *move*. The dimensionality of the container sections they point to is the number of axes that *do not* figure in the name. For instance, **`matrix`**'s **`iterator_i`**s are indexed along the **`i`** axis, and thus point to vector sections that span the **`j`** axis, as shown in Figure 3.3b. Or, since **`cube`**s are three-dimensional containers, their **`iterator_i`**s point to matrix objects, each one being a **`size_j`** $\times$ **`size_k`** matrix, as shown in Figure 3.3d. The same naming convention is used to name **`begin`** and **`end`** methods that, respectively, return




(a) Vector iterator (b) Matrix iterator_i (c) Matrix iterator_ij

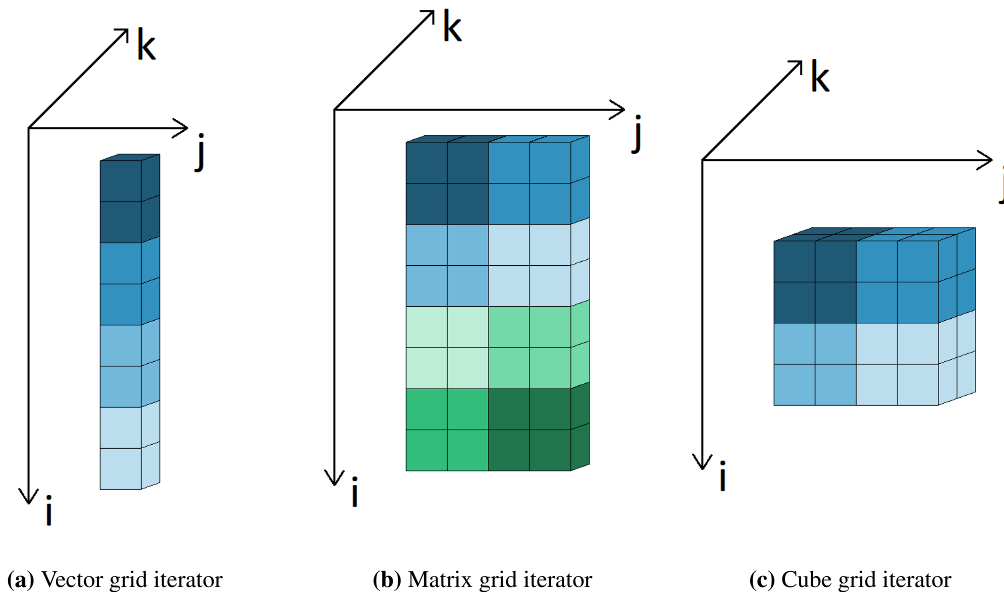
(d) Cube iterator_i (e) Cube iterator_ij (f) Cube iterator_ijk

**Figure 3.3:** PHAST Library iterators.

iterators that point to the first and the "one-beyond-the-last" [152] elements, with the common STL semantics.

Another type of iterator can be obtained from a `grid`, which is a view that can be instantiated on a container. As such, it does not manage data lifetime directly, but can be used to apply a tiling on the *gridded* container. It gives access to iterators that allow sub-vector iterations in `vector` objects, sub-matrix iterations in `matrix` objects, and sub-cube iterations in `cube` objects. The dimensions of the iterable sections can be chosen when the grid view is declared. Figure 3.4 shows some examples of grid iterators.

Every iterator of any kind is part of a *global* ordering. In general, each iterator has a




**(a)** Vector grid iterator **(b)** Matrix grid iterator **(c)** Cube grid iterator

**Figure 3.4:** PHAST Library grid iterators.

multi-dimensional position, with as many dimensions as the number of axes it moves on. The first iterator is always the one having all zeros in its position. By increasing it, its position is increased by 1 on the minor spanned axis, and so on until that value cannot be further increased. Then, the minor axis index is set to zero, and the second-minor is increased, and so on. In brief, iterators are ordered as the lexicographic order of their multi-dimensional positions.

In PHAST Library, non-scalar iterators have the same capabilities as scalar ones: they can be incremented or decremented, they can be used inside algorithms where their sections are processed in a way that abstracts away the container of origin. They can also be dereferenced, but instead of the proxy objects that point to single elements, they return *children* containers. In this case, there is a shared ownership of the data with the container that returned the then dereferenced iterator. A possible solution in this case would be to replace the internal pointers of the containers with **`shared_ptr`** objects [152]. When dereferencing the non-scalar iterator, the **`shared_ptr`** would be copied, increasing its reference count and causing data deallocation only when the last container is destroyed. However, a different solution has been implemented instead to spare the performance overhead due to the **`shared_ptr`** management [162]: the destructor of the child container does not deallocate data, only the parent container does it. If the latter is destroyed before the former, the child container is left with a dangling pointer. This last solution needs more care from the programmer, but it is the one that has the least impact on performance. Also, it is the solution commonly adopted in C++ with *orphan* iterators, as verified with micro-benchmarks compiled with g++

8.4.0, clang++ 10.0.0, and icc 19.0.4.

Grid iterators differ from the others because they may point to container sections that are non-contiguous in memory. The containers they would return when dereferenced would still have this property. Thus, non-contiguity should be managed in the container logic, with a consequent impact on performance when accessing and retrieving data and the necessity to abandon a strong guarantee on the data itself, i.e., its contiguity in memory. To avoid this, grid iterators have been made non-dereferenceable, limiting their use to algorithms only.

The presence of different kinds of iterators lets programmers express with a high level of abstraction various parallel computations that have a broader granularity than scalar elements. Countless examples from various domains can be found in literature: image processing [30, 168, 172], video compression [94, 143], fluid dynamics [3, 28], block-based ciphers [78, 97], linear algebra [34, 36], Finite Element Modeling (FEM) [46, 98], and more.

## 3.4 Algorithms

Algorithms are the tool that PHAST Library gives to its users to invoke parallel computations with a classic sequential-like syntax. They are another transposition of a famous STL concept. The majority of them accept sequences defined by pairs of iterators, the STL-way, while others inspired by linear algebra generally operate on entire containers. All of them have a parallel implementation, a solution that has been explored in other competitor approaches [80, 122, 156]. However, in this case, their semantics has been extended in many algorithms to operate on sections, as opposed to the canonical element-wise processing. So, what they do is to apply in parallel, using the capabilities of the selected architecture, the intended computation to the *sections* pointed by the iterators in the sequences passed as arguments. Table 3.2 lists the available algorithms and the sections they can process.

Every algorithm has been implemented for both the backends included in PHAST Library: `std::thread`s [66, 68, 69] for multi-core CPUs, and CUDA C++ [119] for NVIDIA GPUs. The execution is organized in three layers of functions:

- The first layer is made by the canonical algorithms that can be called from host code. Inside them, the ranges of iterators and containers received as parameters are checked. Then, they are "unpacked" and passed to a second layer that bifurcates depending on the selected backend;
- In the second layer, the information about the data to process, the number of elements, and the parallelization parameters is retrieved. The execution on the

| Algorithm | Works on |
|---|---|
| accumulate | scalar |
| accumulate_for_each | scalar, vector, matrix, cube |
| accumulate_prod_for_each | scalar, vector, matrix, cube |
| accumulate_prod | scalar |
| copy | scalar-scalar, vector-vector, matrix-matrix, cube-cube |
| count | scalar, vector, matrix, cube |
| count_if | scalar, vector, matrix, cube |
| dot_product | vector-vector |
| dot_product | matrix-vector |
| dot_product | matrix-matrix |
| fill | scalar, vector, matrix, cube |
| find | scalar, vector, matrix, cube |
| find_if | scalar, vector, matrix, cube |
| for_each | scalar, vector, matrix, cube |
| for_each | any combination of two sections of any shape |
| for_each | scalar-scalar-scalar |
| generate_normal | scalar |
| generate_uniform | scalar |
| max_element | scalar |
| min_element | scalar |
| replace | scalar, vector, matrix, cube |
| replace_if | scalar, vector, matrix, cube |
| reverse | scalar |
| sort | scalar |
| sort_desc | scalar |

**Table 3.2:** List of PHAST Library algorithms. For each of them, also the shapes of the sections it can process are specified.

device is set up and launched, being it a kernel executed by a grid of CUDA threads or a function executed by multiple OS threads;

- The third layer is the data-parallel device-side execution that actually performs the section-wise calculations expressed by the algorithm semantics.

Some algorithms, like reductions, need an aggregation of the data produced by the parallel workers. These are performed in the second layer and the results propagated to the first layer, where they are returned to the user. Users interact only with the first layer and have the possibility to define functors, which are discussed in the next section, that are directly invoked inside kernels and OS threads in the third layer.

The two solutions have different programming models and different requirements in terms of optimal data-thread mapping and memory access pattern, as briefly discussed in

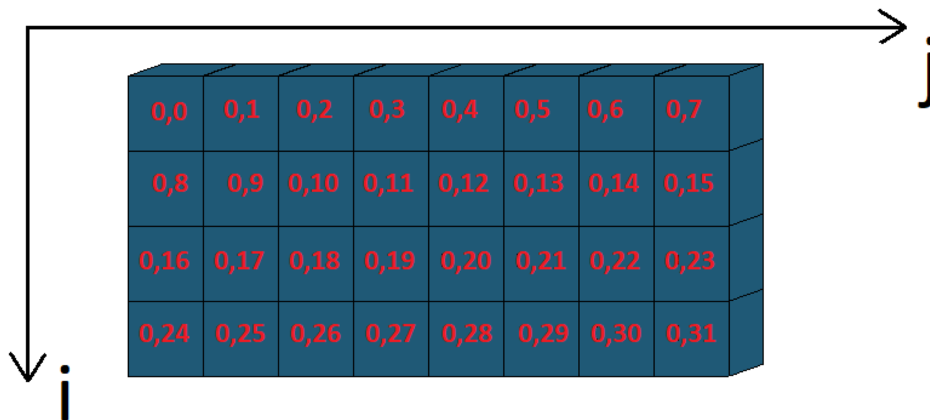
Subsections 2.1.1 and 2.2.1. Their peculiarities have been taken into account to permit reaching high performance on the supported architectures.

In the multi-core case, when an algorithm is executed, a number of threads that is determined by the **n_thread** parallelization parameter is generated. The workload is partitioned between these threads of execution in chunks of contiguous sections. Inside each thread, the single sections of the chunk are iterated and processed in order. This execution scheme exploits the spatial locality of the data, since the iterated sections are contiguous in memory[2]. This way, data can be prefetched, as the full cache line is generally used and the access latency hidden [56]. The calling thread joins the generated threads and, if needed, it aggregates the results. This is the general scheme that is applied to execute embarrassingly parallel algorithms (**for_each**, **replace**, etc.) and reductions (**accumulate**, **accumulate_for_each**, etc.), but it can vary. For instance, **sort** and **sort_desc** are implemented as a radix sort, which needs a number of steps that depends on the data-type size, each one articulated in three different phases [169].

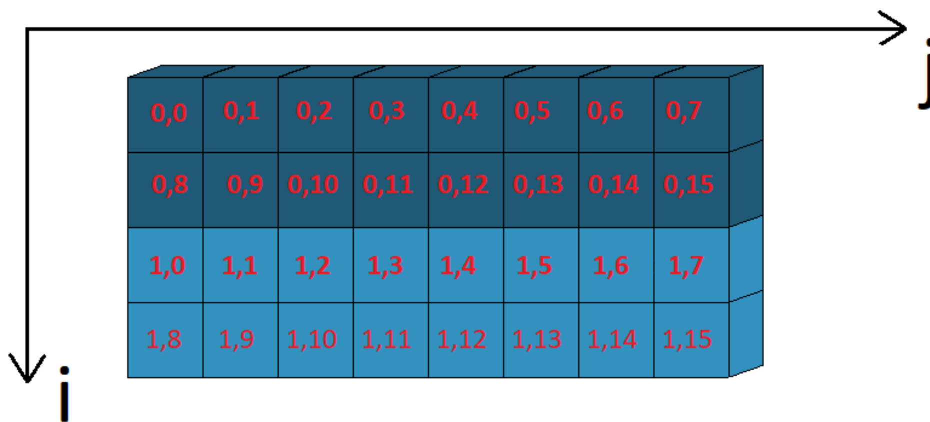
In the CUDA case, the mapping strategy is different, as the other solution would not permit reaching high performance. When the algorithm receives in input a range of scalar iterators, the mapping between CUDA threads and iterated elements follows the scheme described in Section 2.2.1. A number of CUDA blocks is generated, depending on the number of elements, the scheduling strategy adopted, and the selected **major_block_size**. The elements are assigned to the CUDA threads so to have a coalesced access to memory that permits fetching up to 128 bytes with a single transaction [116]. Contiguous blocks work on contiguous data. If the scheduling strategy adopted is **SATURATE**, these blocks cycle over the working set *jumping* ahead of the whole grid-size. Otherwise, they process the assigned elements and terminate, since there are enough blocks to cover all the working set.

Conversely, when non-scalar sections are iterated, the access pattern resembles that achieved when working on an Array-of-Structures (AoS): the contiguous sections can contain dozens of elements, and thus contiguous CUDA threads can access memory locations far away from each other. This aspect can be mitigated with a proper dimension of the **minor_block_size** parameter and the use of in-functor algorithms: in fact, the threads on the minor axis can work on contiguous elements inside the iterated section, realizing a coalesced access pattern. If the data retrieved from one section is greater or equal to 128 byte, there is coalesced access in every section. Otherwise, the total ordering of threads can still lead to a coalesced access across neighboring sections, given that no per-section conditional code limits this. Both cases are shown in Figure 3.5. This aspect will be further discussed in Section 3.7.

[2]It is possible that the elements that make up the section are not local in memory. This can happen with grid iterators that point to matrix and cube sections.




**(a)** Coalesced access with all the threads in the warp accessing contiguous elements in the same section.




**(b)** Coalesced access with the threads of the warp accessing contiguous elements of two contiguous sections.

**Figure 3.5:** Two examples of coalesced accesses to global memory. For each element, the numbers specified are the major and minor indexes of the CUDA thread accessing that element.

Also in this case, the mentioned scheme describes embarrassingly parallel algorithms and reductions. More complex algorithms are not described by it. Again, **`sort`** and **`sort_desc`** are examples of complex algorithms that are implemented with a more articulated logic. Some algorithms are implemented in PHAST Library as thin wrappers around calls to functions of the specialized libraries that are part of the CUDA ecosystem. For instance, **`generate_normal`** and **`generate_uniform`** are wrappers of cuRAND [121] functions. Or, **`dot_product`** algorithms are wrappers of cuBLAS [118] for the **`float`** and **`double`** data types. This choice has been done to take advantage of highly-optimized code included in any CUDA distribution (i.e., without limiting the library adoption) without giving up the PHAST Library abstraction layer.

## 3.5 Functors

Some algorithms have a degree of freedom that allows users to define special computations to customize and specialize the behaviour of an algorithm. This *degree of freedom* is always translated in the algorithm signature as a user-defined functor parameter.

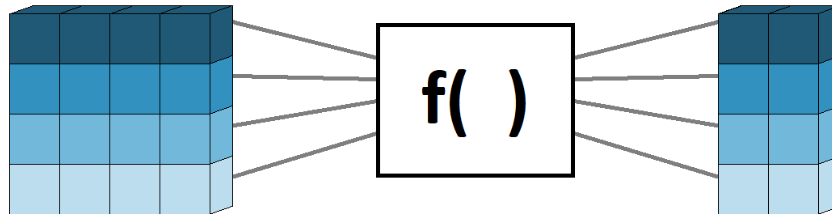



**Figure 3.6:** Binary **`for_each`** with a functor that processes, at each iteration, two vector sections.

PHAST Library allows its users to define many kinds of functors by deriving from library-provided *base-functors* and implementing the **`operator()`** method. There are unary, binary, and ternary functors. What kind must be used in a given computation depends on the algorithm at hand, as the functor -arity must match the number of ranges of iterators passed to the algorithm as arguments.

Another way to classify functors is by taking into account the dimensionality of the sections they process. This also is a constraint on the kind of functor that can be used in an algorithm invocation: each of the parameters in its **`operator()`** method must match the *shape* of the section pointed by the iterators in the corresponding range. For instance, a binary **`for_each`** invoked with two ranges of vector iterators must receive as functor argument one that accepts two vector sections in its **`operator()`** method, as depicted in Figure 3.6.

Functor declarations can be ideally divided in four parts:

1. Template types and name: every functor must have two template parameters, a **`typename`** that represents the data type of the iterated containers, and an **`unsigned int`** that represents a *policy*. The role of the second parameter is managed internally, as it will be discussed in the following sections, and must be set to a default policy by the user;

2. Type of the inherited pre-defined functor: the functor must inherit from a base-functor that defines the -arity and the shape of the parameters of its **`operator()`** method. It is also a template struct and must have the same template parameters as the functor that is being declared;

3. **operator()**: the body of the functor where the main computation occurs. It must be marked by **_PHAST_METHOD** macro, which expands in **__device__** in the CUDA-backend of the library;

4. Optional fields and methods. The latter must also be marked by **_PHAST_METHOD** macro.

The **operator()** method can return any type, but some algorithms have specific requirements on the return type. Algorithms needing a predicate, i.e., **find_if**, **count_if**, and **replace_if**, need a type that can be converted to **bool**, like any primitive type. Reduction algorithms, i.e., **accumulate_for_each** and **accumulate_prod_for_each**, need the return type to be the same as the template type. Finally, **for_each** ignores the return values, and thus any type would suffice.

Base-functors are used to identify the *nature* of the container that is being declared. Inside them, they have fields corresponding to special containers that permit accessing data device-side called *in-functor containers*. These are assigned and initialized inside algorithms to embody and represent the iterated sections. As an example, the binary functor mentioned above can be declared as follows:

```
1  template <typename T, unsigned int policy =
2      phast::get_default_policy()>
3  struct functor : func_vec_vec<T, policy>
4  {
5      _PHAST_METHOD void operator()(
6          phast::functor::vector<T>& vec1,
7          phast::functor::vector<T>& vec2)
8      {
9          // process two vector sections from
10         // two different ranges of iterators
11     }
12 };
```

The in-functor **vector** containers are used to represent two vector sections at each iteration.

Base-functors give also access to *in-functor algorithms*, that will be further discussed in Section 3.7. Their implementation is selected at compile time on the basis of the **policy** template parameter. Thus, they can only be provided as inherited methods and not as standalone functions. For this reason, the inheritance mechanism that regulates functor syntax is necessary in PHAST Library and cannot be eliminated. The major drawback is the current difficulty of supporting lambda functions [152] as a quick and compact way to declare anonymous functors. Although lambdas are not necessarily *better* than classic functors for programmers' productivity [165], they are an integral tool of modern C++ programming frameworks [37, 84, 122]. Their introduction in PHAST

Library, maybe with limited capacities (e.g., without in-functor algorithm support), will be studied as future work.

An important characteristic of PHAST Library functors is that they give access to the iterated sections by abstracting away both the nature of the iterated containers, as only their sections can be seen, and the iteration index. This is because the iteration is performed directly on the data structures that must be processed, rather than in a set of indexes as done in the majority of the heterogeneous frameworks [37, 84, 126]. In those cases, users must explicitly write code to retrieve the intended data using the index, even with complex expressions that involve geometric characteristics of the accessed data structures as dimensionality and strides. In PHAST Library, the index can still be retrieved with the **`get_index()`** method inherited from the base-functor, but its use is mostly intended as any other numeric data that can be used in the application logic. For instance, it could be used to assign incremental values to scalar elements as in the STL **`iota`** algorithm [152], but not to directly retrieve those elements.

## 3.6 Parallelization parameters

| Parameter | Architecture | Description |
|---|---|---|
| Number of threads | multi-core CPU | Number of multi-core threads |
| Affinity weights | multi-core CPU | Weights used to assign threads to logical cores |
| Major block-size | NVIDIA GPU | Number of threads on the primary dimension in CUDA blocks |
| Minor block-size | NVIDIA GPU | Number of threads on the secondary dimension in CUDA blocks |
| Tiling strategy | NVIDIA GPU | Assignment strategy of elements to CUDA blocks |
| Scheduling strategy | NVIDIA GPU | Block allocation on a CUDA grid |
| Shared pre-load | NVIDIA GPU | Whether data should be pre-loaded in shared memory or not |

**Table 3.3:** List of PHAST Library parallelization parameters. For each of them, the architecture it impacts and its role are specified.

The algorithm execution is regulated by architecture-dependent *parallelization parameters*. PHAST Library uses heuristics to infer their values, aiming at selecting the combination of parameters that would provide the highest performance for the problem at hand. Users have also the possibility to explicitly tune them via API calls in order to look for any sweet spots overlooked by heuristics.

Parallelization parameters are the tool to achieve code optimization and tuning in PHAST Library. The user is encouraged to write their code with high-level constructs that are implemented with some flexibility. This flexibility offers tunability from *outside* the application logic, in a way that does not pollute it with fine-grained optimizations and

preserves performance portability. In fact, thanks to the separation between multi-core and GPU backends, the algorithms are implemented twice, with each implementation taking advantage of the techniques that give the best performance on that architecture. Each implementation has additional flexibility that gives the possibility to tune the execution via parallelization parameters to increase the achieved performance on different devices of the same family. This organization increases also code readability, because the application logic and its tuning and optimization are kept decoupled and physically separated.

The available parallelization parameters are listed in Table 3.3. They are individually discussed in the following.

### 3.6.1 Multi-core

#### `n_thread`

It regulates the number of threads used to execute an algorithm. It is adjusted to assume the minimum between the number of iterated elements and the number of logical cores in the system. User can manually intervene to set it to a custom value, but it is never set to a value greater than the number of logical cores, since that would necessarily degrade performance, especially with the explicit affinity management.

#### Affinity weights

During installation, PHAST Library queries the system to achieve some information on the topology of the multi-core CPU. On Linux, it accesses **`/proc/cpuinfo`** to read the number of packages and physical cores and the presence or lack of hyper-threading. A Windows implementation is not yet complete, but in that case the **`GetLogicalProcessorInformation`** Windows API function could be invoked instead [106]. The logical cores are memorized in an array of structures that retain the topology information. This array is ordered on the basis of package, core, and hyper-thread number of each element, with a weighted sorting algorithm that weights each feature with user-adjustable weights. The ordering determines the affinity of the threads launched in an algorithm: the first thread is pinned to the first logical core in the ordering, the second thread to the second core, and so on. So, the weights are called *affinity weights*, and their values can be set in order to explore different affinity strategies. The default values are such that the threads can take as much cache space and resources as they can, in a way that resembles OpenMP's **`SCATTER`** policy [126].

### 3.6.2 NVIDIA GPUs

**major_block_size**

It regulates the number of CUDA threads on the primary axis of bi-dimensional CUDA blocks. It determines the number of sections that are mapped to each CUDA block, since exactly **major_block_size** different sections are processed for each CUDA block. Its value is constrained together with **minor_block_size** by the CUDA programming model, as their product cannot be greater than 1024 [119]. This parameter is overridden by the **ONE_ELEM_PER_BLOCK** tiling strategy.

**minor_block_size**

It determines the number of CUDA threads on the secondary axis of the CUDA blocks. These threads *see* the same iterated section inside algorithms and are used to cooperate inside in-functor algorithms to obtain a second level of parallelism. In this regard, the threads with the same major index and different minor indexes act like *sibling* threads. Its value is limited by the warp size, for synchronization necessities that will be detailed in Section 3.7. This parameter is overridden by the **ONE_ELEM_PER_BLOCK** tiling strategy.

**tiling_strategy**

It regulates how the container sections iterated in an algorithm are mapped to CUDA resources:

- **N_ELEM_PER_BLOCK** assigns **major_block_size** different iterated sections to each CUDA block. Threads with different indexes on the major axis work on different sections and do not share any data. Every element is shared between the **minor_block_size** threads with the same major index that cooperate in the code of in-functor algorithms;
- **ONE_ELEM_PER_BLOCK** assigns one iterated section to each CUDA block. It is shared with **minor_block_size** CUDA threads, with this value being forced to the number of elements of the iterated section. This tiling strategy cannot be used when working on scalar partitions or couples of sections with different numbers of elements. In those cases, **N_ELEM_PER_BLOCK** would be used as a fallback mode.

Of course, the **ONE_ELEM_PER_BLOCK** strategy removes the constraint of **minor_block_size** being necessarily less or equal to the warp size.

**scheduling_strategy**

It regulates how many CUDA blocks are instantiated and scheduled – i.e., the size of the CUDA grid. It can assume two possible values:

- **SATURATE** generates as many CUDA blocks as CUDA multi-processors can schedule at once, looping over the workload;
- **QUEUE** generates the *right* number of CUDA blocks, given the size of the problem and the **major_block_size** value, and it lets the hardware scheduler manage them.

In the first case, the maximum number of blocks that can execute simultaneously is retrieved by combining the maximum number of threads per SM, that can be obtained by calling a CUDA API function, with the chosen block-size and the amount of shared memory needed. These blocks cycle over the available work-load by jumping ahead of the whole grid once the processing on the current **major_block_size** sections is completed. In the second case, a number of blocks that can cover the whole working set is generated according to the following formula, where $N$ is the number of sections to iterate:
$\lceil N \,/\, \texttt{major_block_size} \rceil$
It can be calculated with **int** variables with the formula:
$(N + \texttt{major_block_size} - 1) \,/\, \texttt{major_block_size}$

#### shared_pre_load

It regulates how CUDA shared memory is used in algorithms where a functor is involved. If pre-load is enabled and the iterated sections fit in shared memory, they are stored there before the invocation of the functor's **operator()** method, and stored back to global memory after its execution. Otherwise, shared memory is used inside in-functor algorithms to store data when needed.

This parameter can be useful with functors that are articulated in many instructions that can hardly be mapped to in-functor algorithms: by enabling shared pre-load, all the accesses to global memory would be replaced with accesses to shared memory, and thus possibly gain in performance.

## 3.7 Hierarchical parallelism

Inside functors' **operator()** method, the iterated sections are seen as containers on their own, and thus they are called *in-functor containers*. Apart from **scalar** that can be considered a *degenerate* container, **vector**, **matrix**, and **cube** are analogous to their *main* counterparts that operate in host code. As them, they offer a high-level interface that permit accessing and modifying their data. However, this can be done directly with references to the data, and no proxy classes are needed: any instruction accessing the data is executed on the device, and the datum itself is located in that

memory space. Specifically, it could be in the global memory or in the shared memory if the parallelization parameter **`shared_pre_load`** has been set to **`true`**.

| In-functor container | In-functor iterator | Pointed element |
|---|---|---|
| vector | iterator | scalar |
| | const_iterator | scalar |
| matrix | iterator_i | vector |
| | const_iterator_i | vector |
| | iterator_ij | scalar |
| | const_iterator_ij | scalar |
| cube | iterator_i | matrix |
| | const_iterator_i | matrix |
| | iterator_ij | vector |
| | const_iterator_ij | vector |
| | iterator_ijk | scalar |
| | const_iterator_ijk | scalar |

**Table 3.4:** List of PHAST Library in-functor iterators.

An important feature of in-functor containers, is that they can be iterated via in-functor iterators. They also define **`begin()`** and **`end()`** methods that give access to many in-functor iterators, which are listed in Table 3.4. They also can be incremented, decremented, and dereferenced to return scalar elements as well as non-scalar sections.

In the same spirit of what can be done in host-code with iterators, in-functor iterators also can be passed in ranges to *in-functor algorithms* as arguments.

### 3.7.1 In-functor algorithms

In-functor algorithms are the device-side counterpart of the *main* algorithms. They are invoked inside the functors' **`operator()`** to process ranges of in-functor iterators from in-functor containers. They are defined as methods of the base-functors, and thus they can be accessed inside functors that are declared as derived classes of these. Table 3.5 lists all the in-functor algorithms provided by PHAST Library. The list will be likely expanded in the future.

Some in-functor algorithms (**`accumulate_for_each`**, **`accumulate_prod_for_each`**, and **`for_each`**) accept a functor as the last argument. These are different from the main ones, even though their declaration is similar to that described in Section 3.5. They are called *inner functors* in PHAST Library and can be considered as functors of a lower layer. Inside their **`operator()`** method, they manipulate the sections identified by the ranges of in-functor iterators passed as arguments to in-functor algorithms. An important distinction is that they do not give access to another layer of in-functor algorithms, and thus they are the lowest level of computation that can be achieved in PHAST Library.

| Algorithm | Works on |
|---|---|
| accumulate | scalar |
| accumulate_for_each | scalar, vector, matrix, cube |
| accumulate_prod_for_each | scalar, vector, matrix, cube |
| accumulate_prod | scalar |
| copy | scalar-scalar, vector-vector, matrix-matrix, cube-cube |
| count | scalar, vector, matrix, cube |
| dot_product | vector-vector |
| dot_product | matrix-vector |
| dot_product | matrix-matrix |
| fill | scalar, vector, matrix, cube |
| find | scalar, vector, matrix, cube |
| for_each | scalar, vector, matrix |
| max_element | scalar |
| min_element | scalar |
| replace | scalar, vector, matrix, cube |

**Table 3.5:** List of the in-functor algorithms included in PHAST Library.

As said, in-functor algorithms are invoked inside functors' `operator()` methods, which are called in the third layer of functions in which algorithm execution is organized, as explained in Section 3.4 – the one that happens device-side. In the second layer, the parallelization parameters are retrieved and the device-side execution is set up depending on their values. For this reason, parallelization parameters *affect* the way in-functor algorithms execute at the point that their implementation has to take care of their values and even bifurcate if necessary. This bifurcation is achieved with partial specializations of the base-functors that depend on the `policy` parameter. Users do not manage that parameter explicitly, but in the inner layers of PHAST Library the user-defined functors can be casted to functors with a different policy value in response to combinations of parallelization parameters that may require some special care. An example of parameter that causes this behaviour is `shared_pre_load`: when it is set to `true`, the iterated sections are loaded in shared memory just before `operator()` is invoked, and written back to global memory after its execution completes. Inside `operator()`, the in-functor algorithms that take advantage of the shared memory (e.g., `accumulate`, that calculates a reduction) must skip the pre-loading phase, since data is *already* in shared memory. For this reason, when `shared_pre_load` is set the functor is casted to a functor of the same type but with a different `policy` value: this inherits from a base-functor with *that* `policy` and the in-functor algorithms inside it are implemented not to use shared memory as a temporal, faster storage. Obviously, a similar behaviour could be obtained via in-functor algorithms that would adjust their execution in conditional code depending

on the values of parallelization parameters queried at runtime. However, compile-time choices lead to more efficient implementations, since they are made during compilation and thus cost no clock cycles during execution. The burden of this choice is that there is more *library* code to maintain, but it is all on the shoulder of the library's implementer. Users do not interact with that logic, as it is not part of the interface.

The main advantage of using in-functor algorithms instead of analogous solutions implemented by hand is twofold, as it pertains to productivity and performance. From a productivity standpoint, relying on library calls (usually one-liners when no functors are involved) instead of writing by hand code with cycles and conditionals is obviously a preferable solution. From a performance standpoint, in-functor algorithms are finely tuned implementations, specialized in accordance of the parallelization parameters. They are written to achieve high performance and are also able to take advantage of a further axis of parallelism when the hardware allows it.

To further clarify, the following code shows a functor equivalent to that presented in Listing 3.1, in Section 3.1. It has the same semantics, but the average value is not calculated with an in-functor algorithm, but by the means of a classic **`for`** cycle.

```
template <typename T, unsigned int policy =
    phast::get_default_policy()>
struct func_no_algo : phast::functor::func_mat_scal<T, policy>
{
    _PHAST_METHOD void operator()(
        const phast::functor::matrix<T>& mat,
        phast::functor::scalar<T>& scal)
    {
        T acc = T(0);
        for(int i=0; i<mat.size_i(); ++i)
            for(int j=0; j<mat.size_j(); ++j)
                acc += mat.at(i, j);
        scal = acc / (mat.size_i() * mat.size_j());
    }
};
```

As a first thing, it is evident how this version is longer in terms of lines of code than the other.

On the host, both the functors have been used the same way:

```
phast::cube<float> cube(100, 400, 400); // 100x400x400 cube
phast::vector<float> avg(cube.size_i());

phast::for_each(cube.begin_i(), cube.end_i(), avg.begin(),
    func<float>());
phast::for_each(cube.begin_i(), cube.end_i(), avg.begin(),
    func_no_algo<float>());
```

Both functors are used to calculate the average of 100 400×400 matrix sections of a **cube** in parallel, in lines 4-5 and 6-7. The execution time of the two **for_each** invocations has been measured on a GeForce RTX 2080. An extensive search of the **major_block_size** values identified 32 as the value that delivers the best performance in the non-algo case, which corresponds to an elapsed time of 3.28 ms. In the algo case, conversely, the **minor_block_size** also has a role: the sibling threads can cooperate in the **accumulate** in-functor algorithm. In fact, the best performance is achieved with **major_block_size** and **minor_block_size** set to 1 and 32, respectively. In that case, the elapsed time is 0.19 ms, which corresponds roughly to a 17x speedup with respect to the other version.

### 3.7.2 Synchronization

An aspect that is worth analyzing is the synchronization between sibling threads in PHAST Library. These cooperating threads can be just a few (e.g., 2, 4, etc.) per section, with the CUDA block consisting of many sections. Since the sections are processed independently of each other, it is possible that an in-functor algorithm is invoked inside conditional code that executes only in *some* of the sections in the block. In this situation, it is not possible to call block-wise synchronization primitives inside that in-functor algorithm: it would block because not all the threads in the block took that code path [119]. For this reason, two synchronization primitives have been defined in PHAST Library and are used inside the implementation of in-functor algorithms:

```
1  template <unsigned int policy>
2  __DEVICE void __phast_sync(typename std::enable_if<
3      (policy == 1) || (policy == 2)>::type* = nullptr)
4  {
5      __syncwarp(__activemask());
6  }
7
8  template <unsigned int policy>
9  __DEVICE void __phast_sync(typename std::enable_if<
10     (policy == 3) || (policy == 4)>::type* = nullptr)
11 {
12     __syncthreads();
13 }
```

The first one is a barrier synchronization that involves only the threads of the warp that executed that line of code. It is safe to use it inside conditional code, because the threads that did not took that code path are filtered away by the *active mask*. However, it is limited to the current warp. This is the synchronization barrier that is normally used inside in-functor algorithms, and it is also the reason why **minor_block_size** is limited by the warp size, as explained in Section 3.6.

The second is the classic block-wise synchronization barrier that involves the whole block. It is used in the implementations of PHAST Library in-functor algorithms that correspond to the **`ONE_ELEM_PER_BLOCK`** tiling strategy. In fact, in that case only one section per block is iterated, and all the threads in the block are sibling threads that cooperate on the same section. Thus, in the presence of conditional code, whether all the threads or none of them enters that code path – thus it is safe to synchronize them this way. For this reason, when **`ONE_ELEM_PER_BLOCK`** is selected as a tiling strategy, the limit of 32 sibling threads no longer holds.

### 3.7.3 Innovation and novelty

These aspects here described are unique to PHAST Library. The possibility to maintain basically the same level of abstraction even in device code, to use pre-defined high-level algorithms that, like the main ones, work on sections of various shapes, decoupling the semantics of the computation from the particular layout of the source (in-functor) container has no equivalent in the other heterogeneous programming frameworks. Moreover, they are implemented in various versions that are used transparently by the users that can select the implementation by setting runtime parallelization parameters. So, even if the parameters are calculated according to heuristics and user-defined criteria at runtime, the selection is done by a combination of casting and compile-time template parameters, so that it does not affect the execution time.

This high-level, layered, finely-tuned architecture that involves sophisticated data structures and flexibility can be considered as the main feature of PHAST Library and thus the main contribution to the heterogeneous field of this Ph.D. thesis. In the next chapter, its importance in terms of productivity will be made clear by an in-depth evaluation and comparison against existing solutions.

## 3.8 Interoperability and low-level optimizations

PHAST Library promotes an architecture-agnostic programming style that goes in the direction of code and performance portability across different architectures and across different devices of the same family of architectures. However, it is possible that, for some applications, small portions of code could benefit in performance from leveraging low-level architecture-specific features and optimizations. Well-known techniques like code replication on GPUs or a *surgical* use of intrinsics on multi-core CPUs are examples of this kind of low-level optimizations. The choice of one of these techniques could lead to a possible performance gain, and thus it should not be prevented.

In PHAST Library, there are two globally defined macros that can help to identify the device of execution: **`_PHAST_USING_CUDA`** for NVIDIA GPUs and **`_PHAST_USING_`**

**MULTI_CORE** for multi-core CPUs. They could be used with **#ifdef** pre-processor directives to define device-dependent scopes under which these low-level instructions can be safely placed. In Section 4.2 a sample use of this technique will be shown.

Another context in which this device-dependent scope could prove useful is to achieve interoperability with third party libraries and frameworks. These could provide particular, unique functionalities not included in PHAST Library, or simply being legacy code that is too expensive to rewrite and needs to live aside of modern PHAST Library code. They could be device-specific, and thus their use could need to be put under the scope of the mentioned **#ifdef** clauses not to break the build process of the other backend. Finally, in order to facilitate the access to the raw device data and permit true interoperability, all the containers have methods that return the underlying device pointers.

## 3.9 Task programming

The algorithms discussed so far are examples of programming techniques that expose data-parallelism [56]. They are characterized by the application of the same calculation to collections of data in parallel. Another form of parallelism is task-parallelism. It is a more relaxed form that admits the possibility to split a program in multiple parts (called *tasks*), with each of them possibly doing different calculations that can be performed concurrently in accordance with their relative dependencies.

### 3.9.1 task and stream_task

Usually, task-parallel problems expose specific relations of precedence between tasks that account for logical dependencies between operations to be performed. These can be represented through a DAG, where the nodes correspond to tasks and the directed edges account for the ordering between them, as shown in Figure 3.7. In many cases, the ordering requirement between two adjacent tasks is induced by the data dependency of the successor from the predecessor. Thus, the dependent task can start when its inputs are available, and so, when the preceding one finishes its computation and makes its result available.

The absence of a directed path between any couple of tasks must be interpreted as the absence of dependencies between them. If that is the case, they could be, in principle, executed concurrently. Figure 3.7 shows an example of task-DAG that depicts the dependencies between tasks. There, $task1$ and $task2$ are independent of $task3$, according to the given definition.

Two classes compatible with C++14 [68] and later provide task-DAG support in PHAST Library: **task** and **stream_task**. In the following, the term "task" can refer to

both of them. They are equipped with a **make_task** and **make_stream_task** factory functions that accept a *callable* [40] (free function, class method, lambda expression, or functor) and its arguments, and return a **task** or a **stream_task** object, respectively. The signatures of these functions are shown in the following.

```
template <typename Callable, typename... Args>
task<Callable, Args...> make_task(Callable&& callable,
    Args&&... args);

template <typename Callable, typename... Args>
stream_task<Callable, Args...> make_stream_task(Callable&& callable,
    Args&&... args);
```

In construction, the callable and its parameters are perfectly forwarded and saved into the object so to be later used, as it happens in standard modern C++ constructs for parallelism (e.g., **std::thread** and **std::async**). Both classes expose a **get()** method that executes synchronously the wrapped callable on its parameters.

Unlike the **get()** method of modern-C++ **std::future**s [66, 68, 69] that cannot be invoked more than once, both classes are equipped with a method that can be invoked many times. The main difference between the two classes is the way they behave when these methods are invoked:

- In the **task** case, the first invocation of **get()** calculates the result of the wrapped callable and caches the result, so that subsequent invocations just return the cached result;
- In the **stream_task** case, any invocation of **get()** executes the callable and returns the newly-calculated result, so that subsequent invocations cause multiple executions (e.g., on successive inputs of a data *stream* in streaming applications).

The two classes are meant to be used in different contexts. Because of the possibility to cache the result of the callable in the **task** case, it must return a non-movable copyable object, so that multiple invocations still return a valid result. This limitation does not apply to the **stream_task** class, as each invocation returns a different object.

In both cases, the invocation of the **get()** method is protected by a lock, so that thread safety is guaranteed and task execution relationships are enforced. This way, only the first dependent **task** triggers the execution of the preceding one before waiting for its result, while other dependent **task**s that execute the **get()** will immediately wait for the same result. Conversely, in **stream_task** the lock allows triggering a new execution of the preceding task and so passing a different result to every task executing the **get()**.

Dependencies between tasks are described through the invocation of **make_task** / **make_stream_task** task-factory functions. In fact, if an input parameter of a task B is

generated by a task A, exactly such task A can be syntactically specified as the parameter of task B. In other words, any input parameter of a task can be either a value or an expression, as in a normal function call, or a concurrent task that returns a value having the same type as the parameter. The signature of the callable, which can be considered as the *body* of a task, can have parameters with standard types (i.e., integers, floats, C-like structs, ...), without worrying about the possibility that some of them could be replaced with tasks. This guarantees an easy decomposability of the application parts during the coding process, and easy composability of the overall final application structure also from the concurrent behavior of its parts (tasks). As an example, let's consider two functions, **foo** and **bar**. **foo** has an integer parameter and returns an integer. **bar** has no parameters and returns an integer, as in the following:

```
1 int foo(int x) { /* ... */ }
2 int bar() { /* ... */ }
```

These functions can be used as building blocks of the tasks in many ways, of which the following are just a few:

```
1 auto task1 = make_task(foo, 5); // foo(5)
2 auto task2 = make_task(foo, task1); // foo(foo(5))
3 auto task3 = make_task(bar); // bar()
4 auto task4 = make_task(
5     [](int a, int b, int c){ return a * b + c; },
6     2, task2, task3); // 2 * foo(foo(5)) + bar()
7 auto task5 = make_task(foo, task3); // foo(bar())
```

**Listing 3.2:** Creation of a few simple tasks.

This sample code, whose task-DAG is shown in Figure 3.7, shows how task composition can be expressed in a natural and non-invasive way from a syntactic point of view.

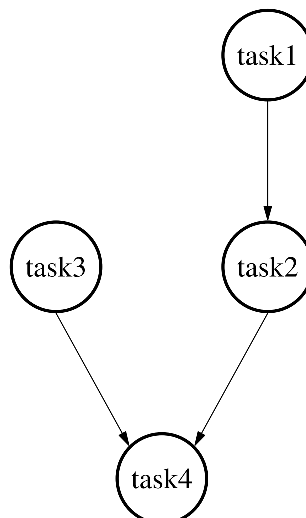



**Figure 3.7:** The task-DAG corresponding to **task4** in Listing 3.2.

This task representation has been designed to require as little code as possible. Compared to other task models [11], this solution is more expressive and allows programmers to declare tasks and specify dependencies in a more concise and natural way. However, it can be slightly less general: for example, since data-dependencies are modeled with callable parameters, any dependency must be expressed this way. To express a dependency from a task that has no return value, currently a dummy one must be added in order to fit in this scheme.

When invoking the **`get()`** method of a task $T_1$, all the **`get`**s of the tasks on which it depends are launched asynchronously and their results are awaited in **`std::future`**s, except the last one that is executed in place. They execute concurrently, and their result will be used as the effective parameter to invoke the callable of the considered task $T_1$.

In the code, this process is achieved in the following way:

- When a task is constructed, the passed arguments are saved into a tuple inside the task, the *parameter-tuple*;
- When the get function is invoked, the *parameter-tuple* is scanned and translated into the *future-tuple*, containing:
  - for each non-task element of type T, the same element;
  - for each task but the last one, the **`std::future`** returned by the invocation of **`std::async`** with the **`get()`** method of that task passed as an argument. The last one is forwarded as-is.
- Then, the *future-tuple* is scanned too and translated into the *result-tuple*, containing:
  - for each non-future element of type T, the same element;
  - for each future, the element obtained by its **`get()`** method, which corresponds to the return value of the callable in the task that originated that future;
  - for the only task, its **`get()`** method is invoked.
- The *result-tuple* contains the final parameters that can be used to invoke the callable.

The last task in the parameter-tuple is treated differently to save one thread spawning and re-use the current thread.

The following code shows how this behavior is achieved in the implementation of the **`task`** class. In the **`stream_task`** case, it is similar, apart from the lacking caching logic.

```
1 return_type get()
2 {
```

```
    return _get(std::index_sequence_for<Args...>());
}

template <std::size_t... Is>
return_type _get(std::index_sequence<Is...>)
{
    {
        std::unique_lock<std::mutex> lock(mut_);
        if(started_)
            ready_var_.wait(lock, [this]{ return ready_; });
        else
            started_ = true;
    }
    if (ready_)
        return output_;

    // here sub-tasks are launched in futures
    auto future_tuple = std::make_tuple(this->get_arg_or_future(
        std::get<Is>(parameter_tuple_))...);
    // for each future, its result is expected
    auto result_tuple = std::make_tuple(this->get_arg_or_result(
        std::get<Is>(future_tuple))...);

    // here the callable is invoked on its parameters
    output_ = callable_(std::get<Is>(result_tuple)...);
    ready_ = true;
    ready_var_.notify_all();
    return value_;
}
```

In the above code, the locking code is implemented so that the first task that invokes **get()** is the one that calculates the result. Any other task that wants to consume this value waits on a condition variable for the value to be produced. The waiting tasks are signaled so they can concurrently read the produced value.

Since the task invocation is done so that independent tasks are invoked before dependent tasks, and the latter are launched only when the former have finished execution, this scheme implicitly executes the task-DAG structure defined in the code relationships.

As Listing 3.2 and Figure 3.7 show, the task-DAG is constructed statically when the tasks are instantiated. However, although this is the proposed approach, it is also possible to construct dynamic task-DAGs. This can be achieved by explicitly constructing tasks inside callables and then invoking their **get()** methods. For instance, the following code depicts a modified mergesort: an array is sorted if its elements are less than a threshold, otherwise it is split in two halves, recursively processed the same way.

```
float* merge(float* ptr1, size_t size1, float* ptr2, size_t size2)
{
```

```
    // merges two contiguous arrays and returns the first pointer
}

float* sorter(float* ptr, size_t size, size_t thr)
{
    if(size < thr)
    {
        sort(ptr, size);
    }
    else
    {
        float* ptr1 = ptr;
        size_t size1 = size / 2;

        float* ptr2 = ptr + size1;
        size_t size2 = (size / 2) + (size & 0x1);

        auto t1 = make_task(sorter, ptr1, size1, thr);
        auto t2 = make_task(sorter, ptr2, size2, thr);

        auto tm = make_task(merge, t1, size1, t2, size2);
        tm.get();
    }

    return ptr;
}
```

In the code above, the two tasks **t1** and **t2** are responsible for sorting the two halves of an array recursively. The two tasks are passed to another task that merges two contiguous arrays together. Since they are passed as task arguments instead of arrays, the receiving merger task (**tm**) depends on them, and so it executes them concurrently via **std::async** calls before being able to call its callable. This way, the two sorting tasks are first executed concurrently, then they return the sorted arrays to the merger task, and finally they can be merged together. Of course, the sorting tasks, during execution, can also split in two halves and invoke the same logic recursively. Again, the task-DAG resulting from the execution of the code above is dynamic, since the number of nodes depends on the runtime values *threshold* and *size* of the vector.

### 3.9.2 Task- and data-parallelism

The **task** and **stream_task** classes seamlessly interoperate with the data-parallel core of PHAST Library: its API is standard modern C++, and thus its functions and methods can be invoked inside callables, in turn wrapped by tasks. The underlying idea is that data-parallel algorithms, which would be called in sequence in a PHAST Library program without tasks, can be embedded into callables, and become part of tasks

created from **`make_task`** or **`make_stream_task`** factory functions. This way, the task orchestration is always done by the CPU, which manages the *task-wrapper objects* dealing with dependencies among them. Then, each task can contain data-parallel *device* code that can be invoked and executed on CPU or GPU. This task orchestration, specifically, can expose and trigger execution concurrency between tasks, and task sub-chains, that are not linked by dependency relations, since the underlying threads launched by **`std::async`**s can execute concurrently.

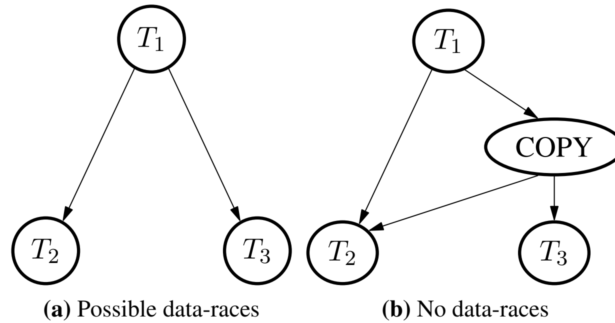
This setup allows the easy specification and coexistence of various kinds of parallelism in the same framework: data-parallelism and task-parallelism. Each task can contain sequential or data-parallel code, targeted for execution and exploiting the parallelism of multi-core CPUs or NVIDIA GPUs. Then, the data-driven orchestration of tasks execution (DAG constraints) allows exploiting both the concurrent usage of heterogeneous resources (e.g., tasks running on the CPU while others run on the GPU) and the concurrent execution of multiple tasks on the same resource.

Data (container) dependencies between tasks can be expressed as follows:

- the first task allocates the container on the heap;
- after its execution, it returns a pointer to the container;
- the dependent task awaits for such a pointer in one of its parameters;
- after the dependent task completes, it can delete the container, if no longer needed, or pass it to the next task.

The problem of determining whether a container is still needed by concurrent tasks and thus manage its deletion can be avoided using shared pointers. This solves the deletion problem and it is preferable to passing whole containers around: this way, they would be copied or moved at task boundaries. The former would have proven expensive, the latter would have not worked in the case of multiple read-only subsequent tasks. Also, in the case of **`task`** classes, moving containers would have caused problems with multiple invocations of the **`get()`** method, as explained before.

The management of data dependencies between tasks can be completely described by explaining the behaviour in case of one-to-one, one-to-many, and many-to-one relations between them. One-to-one and many-to-one do not pose any criticality, since there is no data sharing between subsequent tasks, i.e., the output of a task is not shared between more than one tasks. Conversely, the one-to-many case needs additional care: since the output of a task will be used by *many* tasks without any dependencies between them that would impose an ordering, the dependent tasks would be executed concurrently. If all of them have read-only access on the input data, no further care is necessary. Otherwise, if this data is a pointer to container, the risk of concurrent writing should be avoided. There are two possible solutions:




**(a)** Possible data-races **(b)** No data-races

**Figure 3.8:** Data races resolution strategy: data copying and fake dependency addition.

- Copying data as the first operation of each dependent task;
- Adding copy-tasks before all the *many* tasks but one, adding a fictitious dependency in the non-copying task on all the copy-tasks, in order to start modifications only when copy operations have completed, as shown in Figure 3.8.

### 3.9.3 Truly heterogeneous task-DAGs

In PHAST Library, the application code must be written only once for all the supported devices. The two backends, however, need necessarily two different compilation processes for the following reasons:

- One macro between **`_PHAST_USING_CUDA`** and **`_PHAST_USING_MULTI_CORE`** must be defined to specify the platform of execution. It must be visible to all the translation units to select the right implementation of the various components of the library. It is convenient to put it directly in the Makefile as a compiler flag;
- NVIDIA code must be necessarily compiled with **`nvcc`** version $\geq$ 9.0, while multi-core code can be compiled with any compiler supporting C++11 and later standards;
- The libraries to be linked are different: *libphastmulti.so* or *libphastmulti.a* in the multi-core case, and *libphastcuda.so* or *libphastcuda.a* in the NVIDIA GPU case.

The splitting of the compilation process can be done by writing a single Makefile with conditionals inside, or by splitting the Makefile in two different files, one per platform, in order to completely separate the two compilations. In the second case, in the opinion of the author, the separation of concerns can improve readability.

The globally defined macro and the linked library select one implementation over the other in the internal layers via **`ifdef`** statements. For this reason, the support of

both architectures (multi-core CPU and NVIDIA GPU) in the same program cannot be achieved with the the normal compilation process.

PHAST code is invoked inside callables executed by tasks. So, by having the possibility to call different versions of the same callable, one for each device supported, the multi-device support would be complete and would not need any further adjustments. The task orchestration would be left to the multi-core CPU, while various versions of callables would invoke the data-parallel code on different devices. In terms of productivity, however, this scheme would not be acceptable if the various callables would be obtained by implementing more than once the same callables. Instead, they should be implemented once, compiled once for each supported device, and used together in the code to construct tasks.

Fortunately, it is possible to do so by doing the operations described in the following steps. The names of the files and namespaces adopted are just used as placeholders to improve the readability.

- Write a source file **`core.cpp`** with the implementations of the callables;
- Write a header file **`core.h`** with the signatures of the callables;
- Write two source files **`core_cpu.cpp`** and **`core_gpu.cpp`** where **`core.cpp`** is included as part of a namespace, i.e., *inside* the opening and closing brackets – e.g., **`cpu_impl`** in the first file and **`gpu_impl`** in the second;
- Do the same with two header files, and include **`core.h`** inside them;
- Compile the two source files **`core_cpu.cpp`** and **`core_gpu.cpp`** as two shared objects with the following characteristics:
  - Compile the first shared object following the instruction for PHAST multi-core compilation, and the second following the instructions for PHAST CUDA compilation;
  - Include in each shared object PHAST static library with the *--whole-archive* flag, *libphastmulti.a* in the first case and *libphastcuda.a* in the second;
  - Limit the symbol visibility via a version-script: make the callables global and all the other symbols local.

By following these steps, two headers with the signatures of the callable functions in two different namespaces and two shared objects *that share no symbols* with their implementations can be produced. To gain access to both their versions (multi-core and NVIDIA GPU) and invoke them, it is sufficient to include the headers in the main source file and link the two shared objects. The selection of one of the implementations is done via *cpu_impl* or *gpu_impl* namespaces.

These callables can be used to construct tasks and task-DAGs. Of course, there can be dependencies between them in terms of data-flow, and so a task executed on a device could return a PHAST Library object to a task that will be executed on another device. In this situation, a conversion of that object from a platform to the other is necessary, and consists in a memory copy between devices because its memory structure is identical in both of them. This facility has been added to the library and it is triggered automatically by the inner logic of the task-DAG. In particular, each task knows its platform of execution and the platform of execution of the preceding task. By comparing them, each task can decide if the conversion should be triggered (current task and its predecessor execute on different devices) or not (both current and preceding tasks execute on the same device).

The conversion logic core is a function that takes as input the object to be converted from one domain (e.g., CPU) to the other (e.g., GPU), the source **`device`**, and the destination **`device`**. The need of a conversion is evaluated by comparing the two devices. Generally, a conversion requires a data movement from the memory of the source device to the memory of the destination device.

CHAPTER 4

# Evaluation

PHAST Library is a programming framework that aims at providing a high-level of abstraction and high performance to its users. In order to affirm whether it delivers significant steps in this direction or not, it must be thoroughly evaluated with respect to other state-of-the-art frameworks, from both performance and productivity points of view. The evaluation must involve various programs, as the more programs are modeled, the more the results are representative of the set of all the possible heterogeneous applications.

In this chapter, some applications of different categories are presented and discussed in-depth. First, the discussion focuses on four simple examples with different structural characteristics. Then, a complex AES-based Pseudo-Random Number Generator (PRNG) is analyzed as representative of real-world, highly optimized programs. Finally, a set of randomly generated task-DAGs are evaluated to properly position PHAST Library's task abstraction. In all the cases, the evaluation involves both performance and productivity.

## 4.1 Simple examples

This Section amplifies and renews the discussion presented in [130, 131]. The data shown here are those collected at the time of writing of the papers, mainly [131]. They

are still valid today, although the results would likely be somewhat different, numerically, on more modern hardware.

In the following, four applications implemented using PHAST Library are presented: Triad, DCT8x8, Black-Scholes, and N-Body. First, they are analyzed from a performance portability point of view: it is shown how their performance changes according to various parallelization parameters combinations, and thus the sensitivity of the benchmarks to them. This study shows how tuning and optimization is done in PHAST Library, what parameters are explored and how. The discussion also highlights that it is possible to fine-tune an application by intervening *only* on those parameters, thus leaving the code unaltered.

The benchmarks are then evaluated from both performance and productivity standpoints with respect to two well-established low-level parallel programming approaches, CUDA [119] and OpenCL [81], and two productivity-oriented high-level programming frameworks, SYCL [84] and Kokkos [37]. In particular, the chosen SYCL implementation is ComputeCpp 0.7.0 by CodePlay [21], Kokkos 2.0 [37], CUDA 8.0, and OpenCL 1.2.1 [81].

| | **meeseeks** | **maxi** |
|---|---|---|
| CPU | Intel Core i7-4790K | Intel Xeon E5-2650 v2 |
| CPU type | quad-core HT | dual octa-core HT |
| CPU frequency | 4.00 GHz | 2.60 GHz |
| RAM | 16.0 GB | 64.0 GB |
| OS | Ubuntu14.4 3.16.0 x86_64 | Debian 3.16.7 x86_64 |

| | **elwood** | **golia** |
|---|---|---|
| CPU | AMD Phenom II X6 1100T | Intel Core i7-6800K |
| CPU type | hexa-core | hexa-core HT |
| CPU frequency | 3.30 GHz | 3.60 GHz |
| RAM | 16.0 GB | 128.0 GB |
| OS | Debian 3.2.65 x86_64 | Debian 3.16.0 x86_64 |

**Table 4.1:** Multi-core based machines used to evaluate simple examples.

The multi-core CPUs used for performance measurements are listed in Table 4.1, while the NVIDIA GPUs are listed in Table 4.2.

PHAST Library is compared against four different approaches to give a comparison against the state-of-the-art that is as reliable as possible: the two low-level ones are a reliable reference from the performance standpoints, as they permit reaching cutting-edge performance, albeit with a remarkable programming effort. The two high-level ones are a reliable reference from the productivity standpoints, as they have been designed to explicitly reduce the programming effort in heterogeneous contexts.

| | **GTX970** | **GTX1080** |
|---|---|---|
| GPU | GeForce GTX 970 | GeForce GTX 1080 |
| Compute Capability | 5.2 | 6.1 |
| Global Memory | 4095 MB | 8113 MB |
| GPU frequency | 1.25 GHz | 1.85 GHz |
| Memory frequency | 3505 MHz | 5005 MHz |
| Number of CUDA cores | 1664 | 2560 |
| Host | meeseeks | golia |

**Table 4.2:** NVIDIA GPUs used to evaluate simple examples.

The five frameworks are always compared performance-wise in correspondence of the combination of parameters that provides the best performance. These parameters are *parallelization parameters*, presented in Section 3.6, in the case of PHAST Library, and other degrees of freedom in the others, such as thread number or block-size where they can be explicitly set.

The productivity comparison is based on the idea that, for the same application, a *complex* implementation requires more programming time than a *simple* one, with the former solution being less productive than the latter. Thus, measuring complexity can serve as a measurement of productivity. On this regard, some research has been done over the years, and various complexity metrics defined. In this work, the productivity comparison is done in terms of code metrics. Three of them have been adopted: Source Lines Of Code (SLOC), Halstead's Mental Discriminations (MEN D) [52], and McCabe's Total Cyclomatic Complexity (TOT CY) [99]. SLOC measures the sheer length of a program by counting the non-blank lines of the source-code. MEN D expresses the program complexity by counting the number of different symbols needed to express that program. Finally, TOT CY shows how many paths the program can take during its execution. All the metrics have been calculated with a freely available tool [20]. It is designed for C code, but the adopted metrics can safely capture the complexity of C++ code too.

Before calculating the metrics on the source codes, blank lines, comments, logging, performance measuring, and other portions of code that were not necessary to the correct execution of the benchmark have been removed. This way, the measurement is as fair as possible, since it does not take into account any line of code that is not strictly *necessary*.

All the benchmarks presented below have been implemented by hand except for the CUDA versions of DCT8x8, Black-Scholes, and N-Body, taken from the CUDA SDK, and the OpenCL versions of Black-Scholes and N-Body, taken from the OpenCL SDK. Also DCT8x8 was in the OpenCL SDK, but not in the matrix-matrix multiplication version. The one examined here is a porting of the CUDA implementation.

As discussed in Section 2.3, SYCL and Kokkos expose to the programmers concepts borrowed by OpenCL/CUDA paradigms: local/shared memory, NDRanges/grids of blocks/work-groups, blocks/work-groups of threads/work-items, intra-block/group synchronizations [37, 84]. The code has been structured to take advantage of them since they are necessary concepts to achieve good performance on NVIDIA GPUs. Moreover, as can be seen in the following by the very good performance achieved by OpenCL versions, they are not slowing down the multi-core systems. Code bifurcations by **`#ifdef`**s or similar approaches would have been a possibility to fine-tune the code for each target architecture. They have been avoided mainly for two reasons:

- A truly single-source implementation is what best represents the way Kokkos and SYCL are intended to be used;
- Code bifurcation would have approximately doubled the complexity metrics of kernel code.

All the measures in this section have been obtained by averaging 1000 runs of each benchmark to get robust results. Warmup executions have been performed before starting measuring so to elide any start-up costs. *elwood* and *maxi* support a direct management of the CPU frequency, so it has been set to the maximum frequency available on all the cores, in order to measure the performance achieved with the CPUs at their maximum capability. The other host machines do not have any frequency manager installed, but it was monitored during benchmark execution and, also thanks to the warmup done, the frequency reached its maximum in all the cases.

### 4.1.1 Triad

#### Description

Triad is a simple application from the STREAM benchmark suite [100]. It calculates every element of a vector as the sum of the elements of two other vectors, the second one being scaled by a constant factor. A sequential implementation of the *core* of Triad application could be as follows:

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

**`a`**, **`b`**, and **`c`** are **`n`**-element vectors or arrays of primitive types, and **`scal`** is a scaling factor. Since every element is calculated independently of the others, Triad is an *embarrassingly parallel* [57] application, and its simplicity makes it a good fit to also highlight the bare minimum components needed in a PHAST Library program. In order to give an idea of what these components could be, a possible Triad implementation in PHAST Library is presented in the following.

The API can be accessed by including the header **phast.h** into the main cpp file. In the **main** function, then, a declaration of the needed containers, at least as many as the above arrays must be done. Being it a mono-dimensional problem, **phast::vector**s are the most appropriate choice.

```
1 const int n = // ...
2 phast::vector<double> a(n);
3 phast::vector<double> b(n, 1.0);
4 phast::vector<double> c(n, 2.0);
```

In the code above three vectors of **n** doubles are declared. The elements of **a**, **b**, and **c** are initialized to **0.0**, **1.0**, and **2.0**, respectively.

Each iteration of the sequential for loop can be seen as the application of a ternary function on three scalar elements. In PHAST Library, this concepts translates to the application of a functor with three scalar input/output parameters.

Referring to the concepts exposed in Section 3.5, a proper definition of this functor could be as follows:

```
1  template <typename T, unsigned int policy =
2      phast::get_default_policy()>
3  struct triad_functor : phast::functor::func_scal_scal_scal<T, policy>
4  {
5      _PHAST_METHOD triad_functor(const T& scal) : scal_(scal) {}
6      _PHAST_METHOD void operator()(phast::functor::scalar<T>& a,
7          const phast::functor::scalar<T>& b,
8          const phast::functor::scalar<T>& c)
9      {
10         a = b + scal_ * c;
11     }
12
13     T scal_;
14 };
```

It can be seen that, in the body of the functor, the nature of the iterated sections is abstracted away and the elements are referred directly, with no mention of their indexes inside the containers. **a**, **b**, and **c** are in this regard scalar views on the elements of the containers. The constructor takes care of the setting of the **scal_** field.

Inside main, the functor can be instantiated as follows:

```
1 triad_functor<double> triad(3.0);
```

The parallel computation can be then launched by calling a **for_each** algorithm with three ranges of scalar iterators and a valid instance of a functor as arguments, i.e., **triad** in this case.

```
phast::for_each(a.begin(), a.end(), b.begin(), c.begin(), triad);
```

Before invoking **for_each**, various parallelization parameters discussed in Section 3.6 can be set, so to tune the application execution in search of the best performance. If no choice is made, heuristics are used to make an estimation of the parameter set that would minimize the execution time.

This code is compatible with both families of platforms supported so far by PHAST Library: multi-core CPUs and NVIDIA GPUs. The selection of the platform of execution can be made by adding a global define as a compiler flag, chosen between **_PHAST_USING_CUDA** and **_PHAST_USING_MULTI_CORE**.

#### Performance study

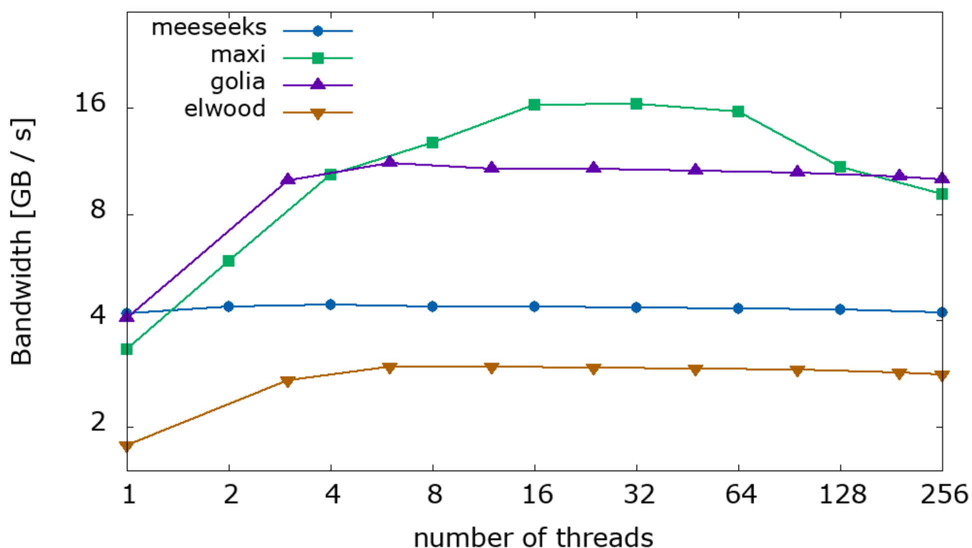



**Figure 4.1:** Triad bandwidth on multi-core systems, measured in GBps.

Triad is a simple memory-bound application. As such, it is an ideal candidate to show the performance cost of PHAST Library abstraction layers with respect to native approaches. Figure 4.1 summarizes the sensitivity of the bandwidth to the number of threads used on the considered multi-core CPU architectures from Table 4.1. A workload of $2^{25}$ doubles (256 MB) has been used.

On all platforms but *meeseeks*, the performance of PHAST implementation scales almost linearly up to the number of physical cores, and then saturates right after. The dual Xeon *maxi* shows a memory bandwidth smaller than a newer generation i7 for up to three cores and then takes advantage from the dual socket and multiple DRAM channels for higher core count.

On the NVIDIA GPUs of Table 4.2, the performance is stable on all the parameter set. The role of **minor_block_size** is not explored because the absence of in-functor

algorithms means that there is no opportunity for in-functor parallelism – thus, the default value of 1 is the most reasonable. Even the variations of **`major_block_size`** do not have a big impact on performance, especially on the GeForce GTX 1080. This may be due to the low requirements (both shared memory and registers) of the Triad application that permit having maximum GPU occupancy even in correspondence of different block-sizes. The only parameter that seem to have a measurable impact is the **`scheduling_strategy`**: **`QUEUE`** is the best on both cards, scoring on average about +2.5GBps and +2GBps on the GTX 970 and GTX 1080, respectively.

| **Version** | **Multi-core Systems** | | | | **APR** |
|---|---|---|---|---|---|
| | **meeseeks** | **maxi** | **elwood** | **golia** | |
| PHAST Library | 4.442 | 16.409 | 2.948 | 11.154 | 1.000 |
| SYCL | 1.563 | 4.263 | - | 2.152 | 0.268 |
| Kokkos | 4.308 | 15.394 | 2.907 | 10.399 | 0.957 |
| OpenCL | 4.301 | 14.443 | 2.966 | 11.251 | 0.966 |
| plain-for | 4.069 | 3.701 | 2.102 | 4.189 | 0.558 |
| C++ 1thr | 4.053 | 2.800 | 2.107 | 4.194 | 0.543 |

**Table 4.3:** Triad bandwidth comparison on multi-core systems, measured in GBps.

Table 4.3 shows the performance achieved on multi-core systems of various Triad implementations. The last column shows the Average Performance Ratio (APR) calculated between each implementation and PHAST Library across all the multi-core CPUs considered. In general, the lower the value, the better PHAST Library performs on average.

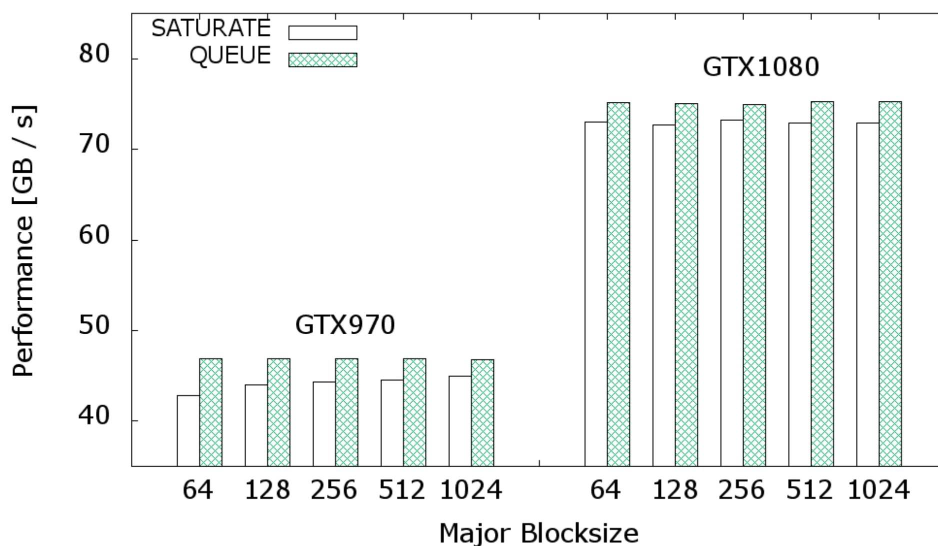



**Figure 4.2:** Triad bandwidth on NVIDIA GPUs, measured in GBps.

In this case, also two sequential implementations are shown: one correspondent to the sequential `for` loop previously discussed, and a 1 `std::thread` wrapper of the `for` loop. The former is shown because it is a straightforward implementation that requires a minimum programming effort, one that could be *tempting* because of its simplicity. The latter is shown to highlight the thread management cost that any parallel library will likely pay on multi-core architectures. The performance of SYCL implementation on *elwood* is not shown because a fatal error caused the termination of the program, maybe due to an incomplete support of that platform.

As can be seen, all the frameworks but SYCL lead to code that performs similarly, with PHAST Library being about 4 times as fast as that. It is possible that this is due to SYCL `parallel_for` or to the synchronization on the device queue being less efficient than the analogous instructions used in similar approaches. The last two rows show that even in a memory-bound application and despite the simplicity of the implementation *going parallel* is truly important, being it the only way to take advantage of the multiple memory channels and scale performance.

| Version | NVIDIA GPUs | | APR |
|---|---|---|---|
| | GTX970 | GTX1080 | |
| PHAST Library | 46.886 | 81.192 | 1.000 |
| SYCL | 13.163 | 13.596 | 0.224 |
| Kokkos | 46.593 | 72.020 | 0.940 |
| CUDA | 46.906 | 81.236 | 1.000 |
| OpenCL | 46.892 | 80.801 | 0.998 |

**Table 4.4:** Triad bandwidth comparison on NVIDIA GPUs, measured in GBps.

Table 4.4 shows the performance of Triad application on NVIDIA GPUs. All the implementations but SYCL reach similar performance, with PHAST Library performing about the same as a CUDA native implementation. This measurement highlights that the overhead of the library layers is negligible on NVIDIA GPUs.

SYCL, however, performs quite differently than the other implementations, and the same performance gap is present in all the benchmarks analyzed. It can be due to the fact that NVIDIA support by ComputeCpp, the SYCL reference implementation, was only experimental when this results have been collected [17] and it is still experimental today [22]. For this reason, PHAST Library performance against SYCL on NVIDIA GPUs will not be discussed.

#### Productivity study

Since Triad is a small application, it permits measuring the complexity baseline of each framework. Table 4.18 shows the relevant calculations.

| Implementation | Code Metrics | | |
|---|---|---|---|
| | SLOC | MEN D | TOT CY |
| PHAST Library | 38 | $2.61 \times 10^4$ | 2 |
| SYCL | 37 | $5.16 \times 10^4$ | 1 |
| Kokkos | 50 | $4.23 \times 10^4$ | 1 |
| CUDA | 41 | $8.14 \times 10^4$ | 4 |
| OpenCL | 77 | $2.14 \times 10^5$ | 4 |

**Table 4.5:** Complexity metrics calculated on PHAST, CUDA, SYCL, Kokkos, and OpenCL implementations of the Triad benchmark.

OpenCL and CUDA have the highest complexity metrics as expected, with OpenCL being the most complex because of the code that manages heterogeneity. In terms of SLOC, PHAST Library and SYCL have a similar value: the former has more setup code than the latter, since it needs to instantiate many objects even for simple programs: vectors, buffers, accessors, and device-queues even for the simplest program, but it can spare some lines of code thanks to the support of lambda functions. Kokkos is penalized by the fact that every Kokkos program needs an initialization and a finalization statement. Since every program can have various `return` points, the adopted solution is in conformity of the RAII principle [154], with the definition and instantiation of an object that initializes the framework on construction and finalizes it on destruction.

As for MEN D, PHAST Library is the less complex solution because the number of symbols to use is reduced with respect to the other two, especially in the case of SYCL and the various necessary objects. In fact, the vectors, the `for_each` algorithm, and the functor are all the PHAST-specific symbols needed.

As for TOT CY, the adopted measures calculates the single functions independently and adds them together in the cumulative complexity metric. In this case, PHAST Library pays the presence of a functor which adds 1 to the TOT CY value.

### 4.1.2 DCT8x8

#### Description

Triad application shows a basic usage of PHAST Library, but its simplicity does not permit to highlight many of the PHAST Library abstractions that allow to write high-level code in multi-dimensional, more complex applications. As an example, also the source code of the Discrete Cosine Transform (DCT) 8x8 is shown, an image transformation application that can be found in CUDA SDK and is the base of the JPEG compression algorithm [168]. It reads an image as an array of `float` values and divides it in square blocks of dimensions $8 \times 8$. For each block $A$, $C^T AC$ is calculated in parallel, where $C$ is an $8 \times 8$ matrix of constant values. A visual representation of this application is

given by Figure 4.3. In this case, the natural containers to be used are **phast::matrix**

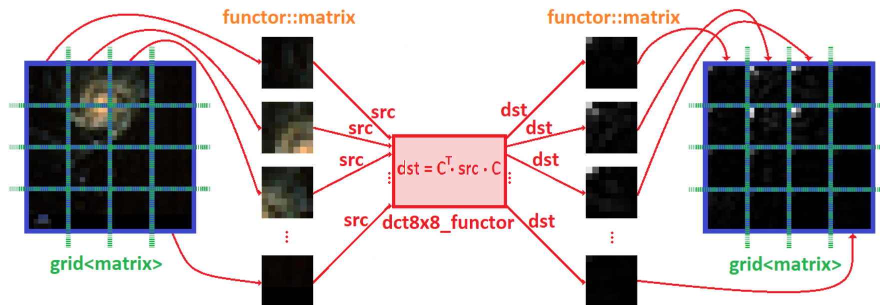



**Figure 4.3:** Visual representation of DCT8x8. The source image is read in a matrix object and a grid is applied to it so that each element of the grid is an $8 \times 8$ sub-matrix. These elements are processed in parallel and used to calculate the corresponding sub-matrices of the destination image.

objects of **float** values. Given **buffer** as a C-like array of **width**×**height** **float** elements representing the image, the two containers associated to source and destination images can be instantiated as follows:

```
1 phast::matrix<float> src_img(buffer, buffer + width*height,
2     height, width);
3 phast::matrix<float> dst_img(height, width);
4 phast::matrix<float> glob_tmp(height, width);
```

**glob_tmp** is a global matrix that will be used as temporary storage. Although it is used in functor code only, it cannot be allocated there because it must be shared between the cooperating threads, and thus all of them must see the same data structure. In the following lines its use will be made clear.

Since the iteration must be performed over $8 \times 8$ sub-matrices, two grids must be defined with that size for their cells:

```
1 phast::grid<phast::matrix<float> > src_grid(src_img, 8, 8);
2 phast::grid<phast::matrix<float> > dst_grid(dst_img, 8, 8);
```

Then, the constant matrices as C-like arrays and the DCT8x8 functor (**dct8x8_functor**) can be defined.

```
1 _PHAST_CONSTANT float C[] = {/*64 float values*/};
2 _PHAST_CONSTANT float Ct[] = {/*64 float values*/};
3
4 template <typename T, unsigned int policy =
5     phast::get_default_policy()>
6 struct dct8x8_functor : phast::functor::func_mat_mat<T, policy>
7 {
8     _PHAST_METHOD dct8x8_functor(phast::matrix<T>& glob_tmp)
```

```
9	    {
10	        tmp.link(glob_tmp);
11	    }
12	    _PHAST_METHOD void operator()(
13	        const phast::functor::matrix<T>& src,
14	        phast::functor::matrix<T>& dst)
15	    {
16	        this->set_partition(tmp, 8, 8);
17	        this->template dot_product_mm_const<T, 8, 8>(Ct, src, tmp);
18	        this->template dot_product_mm_const<T, 8, 8>(tmp, C, dst);
19	    }
20	
21	    phast::functor::matrix<T> tmp;
22	};
```

In the remainder of the **main** function, the functor is instantiated and the parallel computation is invoked as a canonical **for_each** algorithm:

```
1	dct8x8_functor<float> dct8x8(glob_tmp);
2	phast::for_each(src_grid.begin(), src_grid.end(), dst_grid.begin(),
3	    dct8x8);
```

The functor inherits from a **func_mat_mat** struct, so **src** and **dst** are coherently defined as references to **phast::functor::matrix** objects. **tmp** is declared as a field of **dct8x8_functor**, and, as such, it is initialized in the constructor. Its **link** method sets its internal state to act as a view on the whole **glob_tmp** object, thus having its same dimensions and referring to its data. This technique can be used to access whole containers as "global" data in functor code[1]. However, in this case, **glob_tmp** should be further partitioned between the groups of *sibling threads*: the threads of execution that work on the same iterated elements. These groups are as many as the iterators in the ranges passed to **for_each**. So, by executing the instruction

```
this->set_partition(tmp, 8, 8);
```

each group of sibling threads gets its own $8 \times 8$ **phast::functor::matrix** to use as temporary storage out of a bigger **height**$\times$**width** **tmp** matrix.

**dot_product_mm_const** is an in-functor algorithm that calculates the dot-product between a **phast::functor::matrix** and a C-like array whose dimensions are specified as template integers. It is similar to **dot_product_mm**, which calculates the dot-product between two **phast::functor::matrix** objects.

Also in this case, parallelization parameters can be set before calling **for_each**.

[1] On the contrary, each of the parameters of an **operator()** method will be an exclusive view on a section of the iterated container

**Performance study**

The measurements in this section have been done with an image of dimensions $3200 \times 16384$ as input, with pixels encoded as **`float`** values.

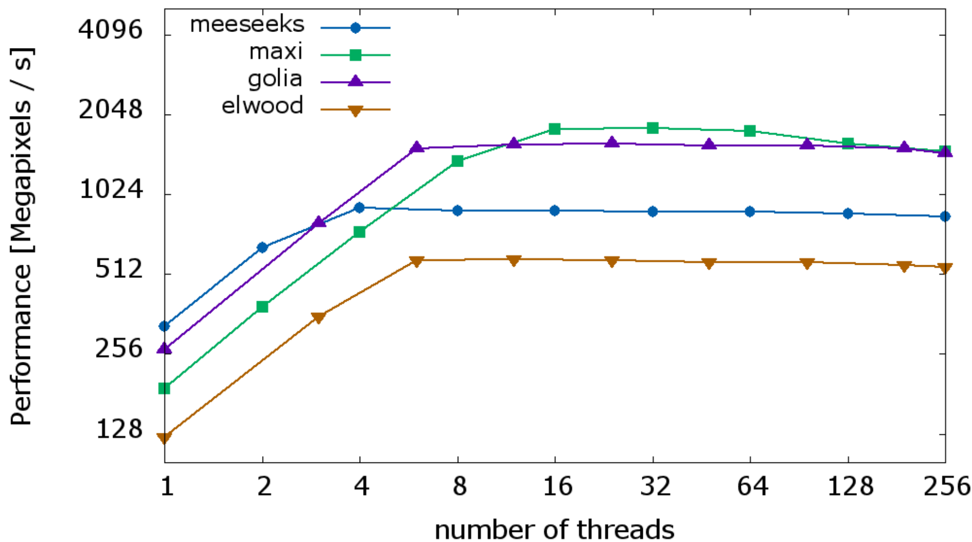



**Figure 4.4:** PHAST DCT8x8 performance on multi-core systems, measured in Megapixels per second.

Performance for multi-core CPUs scales almost linearly as in the previous case, up to the number of physical cores. A further improvement is gained when hyper-threading is taken into account on *maxi* and *golia*.

On NVIDIA GPUs there are more parallelization parameters to set, and performance has been split into two figures. In Figure 4.5, tiling strategy **`ONE_ELEM_PER_BLOCK`** has been used, so a fixed value of 64 for **`minor_block_size`** and 1 for **`major_block_size`** has been forced by that choice. In this case, all the threads in a CUDA block cooperate in the **`dot_product_mm_const`** in-functor algorithms. The histograms on the two graphic cards employed are similar, with the best parameter combination being the same on both cards. The maximum performance is achieved with pre-load of data in shared memory turned off, i.e., it is used on-demand inside in-functor algorithms. **`SATURATE`** is the best performing **`scheduling_strategy`**: this strategy is characterized by the generation of as many blocks needed to saturate the SMs and looping through the workset, as opposed to **`QUEUE`**, that generates lots of blocks and let the scheduler launch their execution. It is possible that the better performance is achieved because the different iterations of the loop on the workset takes advantage of cached data.

Figure 4.6 shows the achievable performance with tiling strategy set to **`N_ELEM_PER_BLOCK`**. This leads to better performance then **`ONE_ELEM_PER_BLOCK`**, with little margin for GTX970 and up to 25% for GTX1080. In this case, also **`major_block_`**

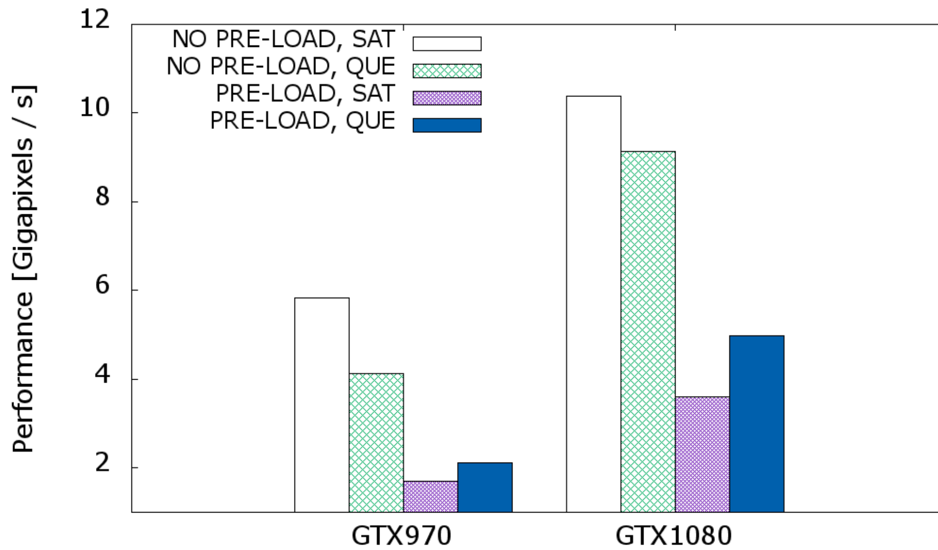



**Figure 4.5:** DCT8x8 performance on NVIDIA GPUs, with **`ONE_ELEM_PER_BLOCK`** tiling strategy, measured in Gigapixels per second.

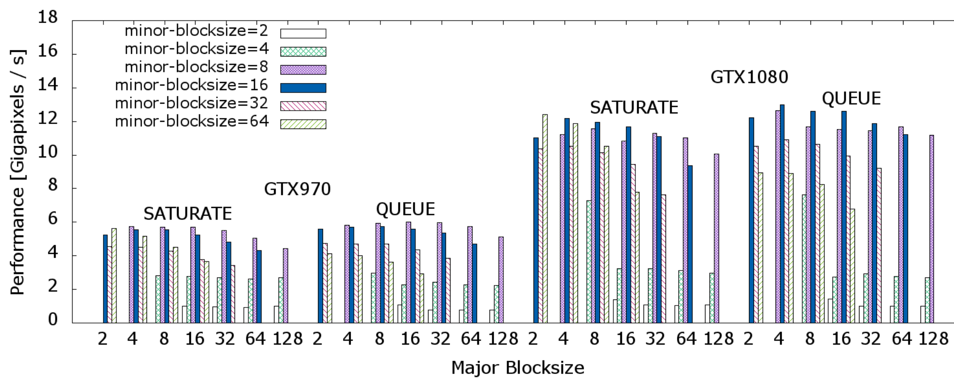



**Figure 4.6:** DCT8x8 performance on NVIDIA GPUs, with **`N_ELEM_PER_BLOCK`** tiling strategy and no shared pre-load, measured in Gigapixels per second.

**`size`** and **`minor_block_size`** can be varied, so various combinations of these parameters are explored. Some combinations that lead to low performance or that does not make sense in CUDA are missing. These include all the combinations with shared-pre-load on, or others with the total number of CUDA threads per block being less than 32 or greater than 1024. The two cards have a different best parameter set: 16, 8 on GTX970 and 4, 16 on GTX1080 for **`major_block_size`** and **`minor_block_size`**, respectively, with **`QUEUE`** scheduling strategy for both.

Table 4.6 shows DCT8x8 performance on multi-core systems. PHAST Library is, on average, about twice as fast as both SYCL and Kokkos. This difference is due to the fact that both SYCL and Kokkos single-source benchmarks have been coded to

| Version | Multi-core Systems | | | | APR |
|---|---|---|---|---|---|
| | meeseeks | maxi | elwood | golia | |
| PHAST Library | 0.908 | 1.811 | 0.580 | 1.587 | 1.000 |
| SYCL | 0.589 | 1.054 | 0.246 | 0.842 | 0.546 |
| Kokkos | 0.559 | 1.365 | 0.268 | 0.649 | 0.560 |
| OpenCL | 1.219 | 2.077 | 0.034 | 2.186 | 0.981 |

**Table 4.6:** DCT8x8 performance comparison on multi-core systems, measured in Gigapixels per second.

have good performance for GPUs as explained at the beginning of this Section. In this application, the GPU paradigm imposes to copy into the shared/local memory the 8x8 sub-matrix, pre- and post-multiply it and then copy it back to global memory. All the threads in a team collaborate in these operations and explicit synchronization calls are needed after each operation. Since Kokkos is a high-level wrapper of OpenMP, all these operations are translated in OpenMP thread code, hampering multi-core performance. A better approach for them would have had one thread working on one sub-matrix at a time and no local copy, but this would have been very bad for NVIDIA GPUs. These considerations highlight that SYCL and Kokkos, despite giving a high level of abstraction host-side, require programmers to take care of many low-level architecture-specific details device-side, making programs not performance portable.

SYCL, also, does not seem to elide these operations in its compilation phase, and so it performs similarly. PHAST Library, conversely, hides these details inside the implementations of the `dot_product_mm_const`: GPU implementation copies data in shared memory, synchronizes the threads in the block, and takes advantage of this programming paradigm. The multi-core implementation, conversely, does not need this, since the multiple levels of cache already implement a similar logic.

Finally, OpenCL performs better than PHAST on *meeseeks*, *maxi*, and *golia*, with a maximum difference of 37.74% on *golia*. An investigation into the generated assembly code led to the identification of the source of this difference. It depends on the Intel implementation of OpenCL, which aggressively vectorizes the kernel code thanks to the Implicit CPU Vectorization Module [63]. It is a *horizontal* auto-vectorization, with instructions from different work-items being executed on different SIMD lanes [63].

Since PHAST wraps modern C++ `std::thread`s, this mechanism is not currently available. In-functor algorithms are vectorized, thus applying a *vertical* vectorization of the code. This aspect of PHAST Library can be improved, and its improvement is left as future work. On the other hand, on *elwood*, OpenCL by AMD vendor has been employed, being *elwood* an AMD multi-core. In this case, PHAST is about 17 times faster, probably also due a sub-optimal compilation strategy of OpenCL by AMD

vendor.

| Version | NVIDIA GPUs | | APR |
|---|---|---|---|
| | GTX970 | GTX1080 | |
| PHAST Library | 5.996 | 12.986 | 1.000 |
| SYCL | 1.420 | 2.514 | 0.215 |
| Kokkos | 5.022 | 8.194 | 0.734 |
| CUDA | 4.413 | 9.260 | 0.725 |
| OpenCL | 4.397 | 8.940 | 0.711 |

**Table 4.7:** DCT8x8 performance comparison on NVIDIA GPUs, measured in Gigapixels per second.

Table 4.7 shows the performance of DCT8x8 benchmark on NVIDIA GPUs. PHAST Library performs significantly better on both cards than Kokkos, CUDA, and OpenCL. Looking at the last column, PHAST Library performs around +30% with respect to the others, on average. However, PHAST Library is a wrapper of CUDA, it generates code that is potentially feasible also in CUDA alone and so, in principle, it cannot be faster than CUDA. The main reason for the performance difference between these versions here lies in the slightly different algorithm PHAST Library uses and in the tuning of the parallelization, which can easily explore the parameter space without altering the source code. Specifically, PHAST Library uses more constant memory, but it performs better, i.e., it allocates both $C$ and $C^T$ matrices instead of only one accessed by-rows and by-columns, as done in the CUDA code. Kokkos too has been coded to take advantage of this. Also, the **`tiling_strategy`** adopted in PHAST code that leads to the best performance is **`N_ELEM_PER_BLOCK`**, but the standard implementation of this benchmark (i.e., the one provided by CUDA SDK) uses a mapping that resembles **`ONE_ELEM_PER_BLOCK`**. It is a more intuitive mapping that also leads to more concise code, so it is the one used in Kokkos implementation too.

#### Productivity study

In the DCT8x8 case, PHAST Library is by far the less complex implementation. It is around 19% shorter than SYCL, and even more with respect to the other frameworks. Again, this is due to the fact that the invocations of **`dot_product_mm_const`** in-functor algorithms implement device-side matrix multiplications with just a single line of code. In the other cases, there are no equivalent functions to rely on, and the necessary operations have been implemented by hand and are included in SLOC calculation.

Also from a MEN D point of view, PHAST Library provides the less complex code. In this case, the Kokkos and SYCL implementations, which are the most productivity-oriented frameworks, are also the most complex: they are 3.95x and 3.31x as complex

| Implementation | Code Metrics | | |
|---|---|---|---|
| | SLOC | MEN D | TOT CY |
| PHAST Library | 90 | $3.59 \times 10^5$ | 5 |
| SYCL | 111 | $1.19 \times 10^6$ | 8 |
| Kokkos | 148 | $1.42 \times 10^6$ | 11 |
| CUDA | 136 | $3.93 \times 10^5$ | 11 |
| OpenCL | 203 | $6.17 \times 10^5$ | 12 |

**Table 4.8:** Complexity metrics calculated on PHAST, CUDA, SYCL, Kokkos, and OpenCL implementations of the DCT8x8 benchmark.

as PHAST Library, respectively. Evidently, the high-level abstractions of the host code are not sufficient to counterbalance the lack of them in device code.

Also in the TOT CY case PHAST Library code is the less complex. Again, the reason has to be found in the high-level algorithms and in-functor algorithms it provides: since the implementation logic is wrapped by them, any conditional code and helper functions invoked do not appear in application code.

### 4.1.3 Black and Scholes

#### Description

Black-Scholes is a financial benchmark. It calculates the price of $N$ call and $N$ put options taking stock price, strike price, and time to expiration as inputs for each option. Since the option final prices can be calculated independently by applying a closed-form formula to the inputs, it is an embarrassingly parallel application. As such, it is particularly suitable for parallelization. In PHAST Library it has been coded as a **`for_each`** that takes a **`func_scal_scal`** functor as argument and two ranges of scalar iterators from the *call* vector and the *put* vector, respectively, where the final prices are stored.

#### Performance study

The measurements in this Section have been done with 8 million options, represented by **`float`** values.

Figure 4.7 shows the obtained performance of the PHAST Library implementation on the multi-core CPUs from Table 4.1. All the curves are almost linear for a number of threads less than or equal to the number of physical cores is used. A further performance gain is given by the hyper-threaded cores on *meeseeks*, *maxi*, and *golia*. The maximum gain is achieved on *maxi* with 32 threads, equal to a 54.50% improvement with respect to 16 threads.

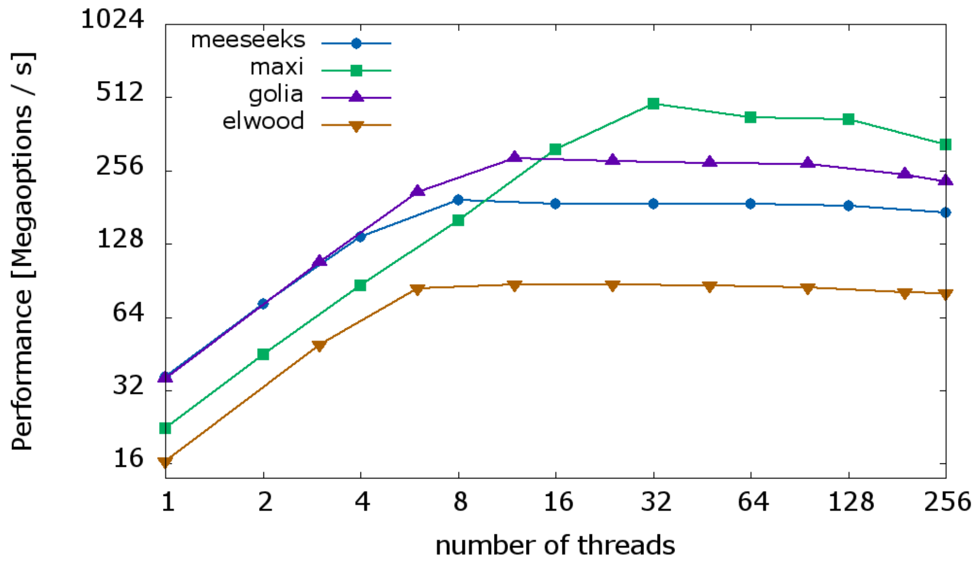



**Figure 4.7:** Black-Scholes performance on multi-core systems, measured in Megaoptions per second.

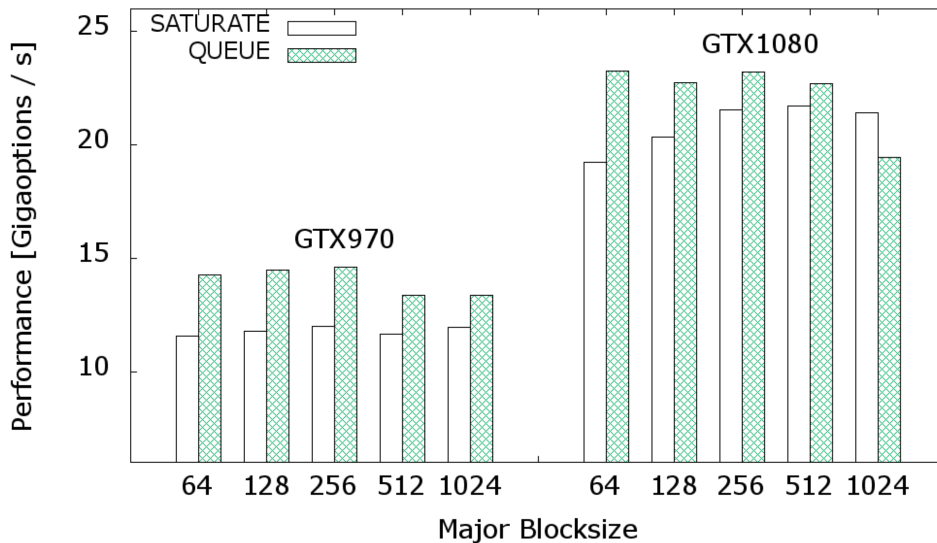



**Figure 4.8:** Black-Scholes performance on NVIDIA GPUs, measured in Gigaoptions per second.

Figure 4.8 shows the achieved performance on the graphic cards listed in Table 4.2. The study takes into account only the role of the **`major_block_size`** and **`scheduling_strategy`** parameters. **`shared_pre_load`** has no impact on performance because there is only a single write to global memory for each option. **`tiling_strategy`** and **`minor_block_size`** must be forced to **`N_ELEM_PER_BLOCK`** and 1, respectively, because the elements processed in parallel are scalar values and there is no

opportunity for exploiting in-functor parallelism.

The performance histograms are similar on the two cards. The maximum performance is obtained in both cases with the same configuration, i.e., `scheduling_strategy` set to `QUEUE` and `major_block_size` set to 256.

| Version | Multi-core Systems | | | | APR |
|---|---|---|---|---|---|
| | meeseeks | maxi | elwood | golia | |
| PHAST Library | 0.195 | 0.482 | 0.087 | 0.287 | 1.000 |
| SYCL | 1.201 | 2.301 | 0.046 | 1.902 | 4.522 |
| Kokkos | 0.101 | 0.349 | 0.067 | 0.192 | 0.670 |
| OpenCL | 0.926 | 1.982 | 0.076 | 1.036 | 3.336 |

**Table 4.9:** Black-Scholes performance comparison on multi-core systems, measured in Gigaoptions per second.

Table 4.9 shows the performance achieved by Black-Scholes benchmark on the multi-core systems shown in Table 4.1. PHAST Library performs 33% faster than Kokkos, on average. Part of this is due to Black-Scholes taking advantage of a fast *reverse square root* mathematical function that is provided by CUDA, OpenCL, SYCL, and also PHAST Library frameworks. In PHAST Library, in particular, this function is provided as `phast::math::rsqrt` that is part of a mathematical library. On NVIDIA GPUs, it wraps the CUDA implementation; on multi-core CPUs, it has been implemented based on the fast inverse square root algorithm [110]. Conversely, in the Kokkos case, $1/sqrt$ has been used on multi-core systems and $rsqrt$ on NVIDIA GPUs.

OpenCL and SYCL implementations perform better than both PHAST Library and Kokkos on Intel systems, with SYCL performance being 4.5x the performance of PHAST Library on average. The reason is the following: in this benchmark, all the calculations are done on `float` values via standard arithmetic operations. This application is thus particularly eligible for horizontal vectorization across different threads of execution, and apparently both OpenCL and SYCL take advantage of that. PHAST Library, conversely, cannot soften this performance difference via in-functor algorithms: only scalar elements are manipulated in-functor, so there is no opportunity for them. However, a different approach to vectorize the code has been taken: the vectors of `float`s representing option prices have been changed into vectors of `float8_t`s, thus expressing data as vectors of groups of 8 prices each. This way, on *meeseeks* the performance has growth to 1.180 Goption/s: 24.7% faster than Intel-OpenCL and only 1.75% slower than SYCL.

Finally, on *elwood*, OpenCL and SYCL performance are 12.64% and 47.13% slower than PHAST Library, respectively. This witnesses the negative overall overhead of PHAST Library if no other factors (e.g., auto-vectorization) come into play.

| Version | NVIDIA GPUs | | APR |
|---|---|---|---|
| | GTX970 | GTX1080 | |
| PHAST Library | 14.590 | 23.252 | 1.000 |
| SYCL | 0.477 | 0.477 | 0.027 |
| Kokkos | 14.588 | 23.290 | 1.001 |
| CUDA | 14.820 | 23.674 | 1.017 |
| OpenCL | 12.973 | 23.038 | 0.940 |

**Table 4.10:** Black-Scholes performance comparison on NVIDIA GPUs, measured in Gigaoptions per second.

Table 4.10 shows the performance obtained by various implementations of Black-Scholes benchmark on NVIDIA GPUs. This is a straight-forward benchmark, an embarrassingly parallel computation on scalar elements that does not need any shared memory or cooperation between threads. In fact, Kokkos and PHAST Library, two high-level wrappers of CUDA, perform similarly to it: they are slightly slower because of the additional library layers.

However, OpenCL code is slower than PHAST Library code, the latter being 12.46% faster on GTX970 and 0.93% faster on GTX1080 with respect to the former. On average, the four frameworks perform similarly.

**Productivity study**

| Implementation | Code Metrics | | |
|---|---|---|---|
| | SLOC | MEN D | TOT CY |
| PHAST Library | 75 | $1.44 \times 10^5$ | 3 |
| SYCL | 94 | $3.19 \times 10^5$ | 3 |
| Kokkos | 90 | $1.91 \times 10^5$ | 5 |
| CUDA | 105 | $2.46 \times 10^5$ | 7 |
| OpenCL | 164 | $6.08 \times 10^5$ | 12 |

**Table 4.11:** Complexity metrics calculated on PHAST, CUDA, SYCL, Kokkos, and OpenCL implementations of the Black-Scholes benchmark.

Also in this case, PHAST Library's implementation of the Black-Scholes benchmark is the less complex, according to all the considered complexity metrics. Its length spans from being around 83.33% of Kokkos implementation to 45.73% of OpenCL implementation, that also in this case scores as the most verbose programming approach.

Its MEN D value is similar to Kokkos' (75% of its value) and considerably less complex than SYCL, being less than a half. Since this benchmark is rather compact and the device-side code is similar in all the cases, the majority of this difference comes

from the host-side code, that requires the instantiation of various classes on the SYCL side, as discussed before.

Finally, the TOT CY value is the same as SYCL, but it is less than all the other frameworks analyzed, even 1/4th than OpenCL's value.

### 4.1.4 N-Body

#### Description

N-Body simulates the gravitational interaction between $N$ bodies by calculating all the $N^2$ interactions at every step. To store the relevant data of the bodies, two vectors are used: one for positions and one for velocities. Each element is modeled as a **`float4_t`** variable, a PHAST Library vector type, analogous to the CUDA **`float4`** type [119]. It is used to model a vector in the 3D-space (position or velocity) and 1 **`float`** for the body mass.

The core of the PHAST Library implementation are two **`for_each`** invocations. The first one operates on a range of scalar iterators from the *velocity* vector. In its functor, it calculates an acceleration value by applying an **`accumulate_for_each`** in-functor algorithm that iterates over all the positions and calculates, for each of them, a body-body interaction that is formalized into an inner-functor. This acceleration is multiplied by the time interval of the simulation to obtain a new velocity value, which is written into the vector. Then, the second **`for_each`** updates the positions based on the updated velocity values.

#### Performance study

The measurements in this Section have been on a system of 16384 randomly initialized bodies.

In this case too, performance scales almost linearly up to the number of physical cores, with a further performance gain on all the systems when hyper-threading capabilities are used. *maxi* is the multi-core system with the bigger gain, that amounts to 16.08%.

Figure 4.10 shows the performance reached on the GPUs. In this case, **`N_ELEM_PER_BLOCK`** tiling strategy only has been used, since the iteration on scalar elements forces this choice. For the same reason, **`minor_block_size`** has been set to 1. **`shared_pre_load`** has been set to false since it caused a worsening of the performance in all the cases. In fact, as explained in Section 3.6, when this flag is set, the library fills shared memory with the iterated data at functor-body boundaries and does not use it inside in-functor algorithms. Since for every body all the other bodies must be iterated in the main in-functor algorithm, it is more efficient to fetch groups of bodies when needed and share them between the CUDA threads, instead of pre-fetching only one body per

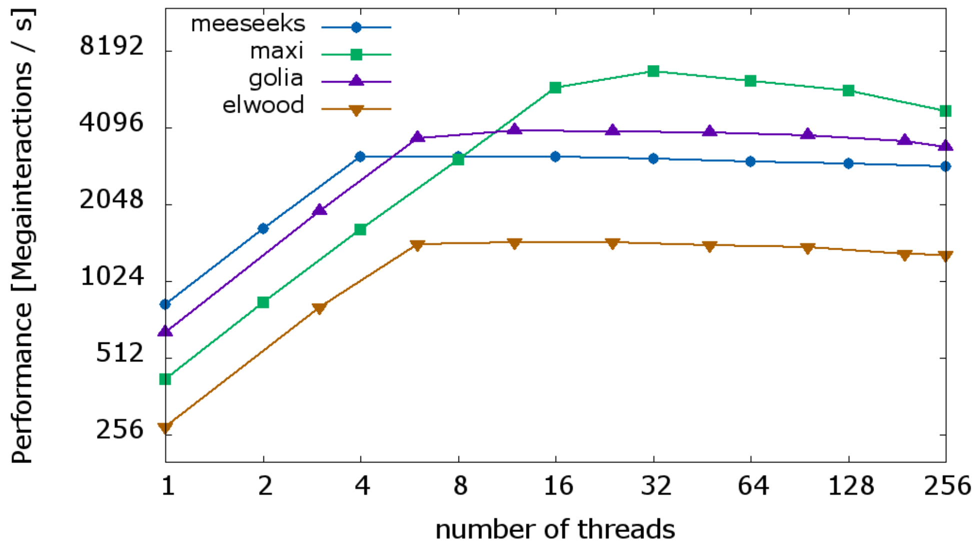



**Figure 4.9:** N-Body performance on multi-core systems, measured in Megainteractions per second.

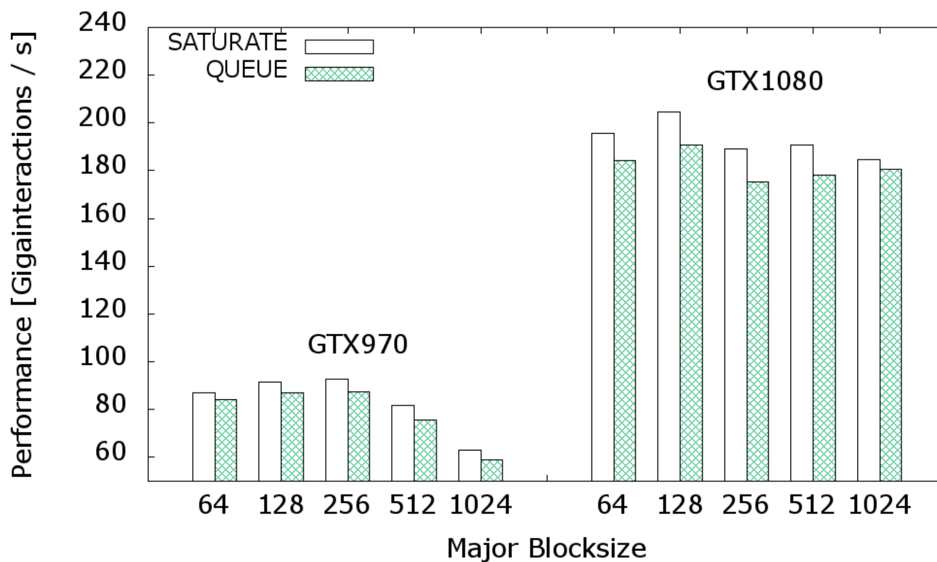



**Figure 4.10:** N-Body performance on NVIDIA GPUs, measured in Gigainteractions per second.

calculation.

Figure 4.10 shows that the best performance is achieved for different **`major_block_size`**s on the two graphic cards: 256 on GTX970 and 128 on GTX1080, respectively. **`SATURATE`** is the best scheduling strategy for both, presumably for its capacity of taking advantage of cached data.

Table 4.12 shows the N-Body performance on multi-core CPUs. Also in this case, PHAST Library performs better than Kokkos. It is, on average, about five times as fast as

| Version | Multi-core Systems | | | | APR |
|---|---|---|---|---|---|
| | meeseeks | maxi | elwood | golia | |
| PHAST Library | 3.143 | 6.787 | 1.456 | 3.973 | 1.000 |
| SYCL | 4.155 | 10.200 | - | 5.850 | 1.432 |
| Kokkos | 0.686 | 1.505 | 0.248 | 0.785 | 0.202 |
| OpenCL | 11.983 | 19.048 | 0.385 | 14.068 | 2.606 |

**Table 4.12:** N-Body performance comparison on multi-core systems, measured in Gigainteractions per second.

Kokkos. A consistent part of this difference is due to the **`rsqrt`** use, even more evident in this case: in fact, this function is used in *every* body-body interaction calculation, and so it is invoked $N$ times for each of the $N$ bodies.

On *elwood*, the same error had in the Triad case occurred, so no performance has been calculated on that machine.

Also in this case, OpenCL and SYCL perform better than PHAST Library, with the former having a performance ratio of about 2.61x and the latter 1.43x on average. It is not trivial to understand how OpenCL achieves that performance: the single elements representing the bodies are structs of four **`float`**s, in a layout that is sometimes called AoS. The peculiarity of this layout is that the instructions performed on the same elements of different structures cannot be easily vectorized, since these elements are not contiguous in memory. A possible workaround is to manually reorder data by changing the layout in a Structure-of-Arrays (SoA). By analyzing the generated assembly code of Intel OpenCL, it is evident that the compiler does just that: it reorders data, it vectorizes the execution of instructions on the same elements from different structures (also, in different work-items), and then it reorders data back to match the original layout. It is possible that also SYCL takes advantage of similar techniques.

The integration of this capability in PHAST Library will be investigated as future work, along with the automatic vectorization used in the Black-Scholes case, using well-known data-layout transformation strategies [88, 164]. Most probably, PHAST Library itself will need an extension with new backends with vectorization capabilities. A good candidate is OpenCL 2.2 [85], but it need a suitable implementation first. Also SYCL, being standard modern C++, could be used as a PHAST Library backend.

Table 4.13 shows the N-Body performance on NVIDIA GPUs. On both cards the CUDA implementation performs better than the others, being 1.78% and 7.77% faster than PHAST Library implementation on GTX970 and GTX1080, respectively. Though, PHAST implementation is 2.72% faster than OpenCL on GTX970, but 6.42% slower on GTX1080. Also, PHAST is slightly slower than Kokkos on GTX1080 (0.04%), but faster on GTX970 (6.7%). On average, the performance of PHAST Library, Kokkos, CUDA,

| Version | NVIDIA GPUs | | APR |
|---|---|---|---|
| | GTX970 | GTX1080 | |
| PHAST Library | 92.836 | 204.676 | 1.000 |
| SYCL | 14.170 | 13.926 | 0.110 |
| Kokkos | 86.957 | 205.588 | 0.971 |
| CUDA | 94.493 | 220.577 | 1.048 |
| OpenCL | 91.995 | 218.722 | 1.030 |

**Table 4.13:** N-Body performance comparison on NVIDIA GPUs, measured in Ginteractions per second.

and OpenCL implementations are similar, with differences that lie in the 2.9-4.8% range.

**Productivity study**

| Implementation | Code Metrics | | |
|---|---|---|---|
| | SLOC | MEN D | TOT CY |
| PHAST Library | 211 | $5.67 \times 10^5$ | 19 |
| SYCL | 192 | $5.53 \times 10^5$ | 20 |
| Kokkos | 234 | $6.79 \times 10^5$ | 23 |
| CUDA | 442 | $1.13 \times 10^6$ | 43 |
| OpenCL | 618 | $2.60 \times 10^6$ | 43 |

**Table 4.14:** Complexity metrics calculated on PHAST, CUDA, SYCL, Kokkos, and OpenCL implementations of the N-Body benchmark.

N-Body's code is the longest of all the considered simple examples. It is longer in PHAST Library than SYCL, because of the use of functors and inner-functors in the former that need more SLOC than SYCL lambdas. However, the MEN D and TOT CY values are similar, with PHAST Library scoring +2.5% and -1, respectively. All the other approaches are significantly more complex than PHAST Library.

### 4.1.5 Summary

From a performance point of view, PHAST Library proves competitive on both NVIDIA GPUs and multi-core CPUs.

On NVIDIA GPUs, it reaches near-native performance in all the benchmarks considered. It even surpasses it in the DCT8x8 case, thanks to its high-performance implementation of various in-functor algorithms and the ability to select a non-trivial *scheduling strategy* for the CUDA blocks with just a single runtime parameter.

On multi-core systems, its performance is competitive with the other considered approaches in two cases out of four. In the OpenCL and SYCL cases, in fact, it resulted

slower because of their auto-vectorization capabilities that include, in the case of OpenCL only, also AoS to SoA transformations. The performance gap that derives from them would be bridgeable in PHAST Library integrating the same data-layout transformations and auto-vectorization capabilities, which is left as future work.

| Implementation | Code Metrics | | |
|---|---|---|---|
| | SLOC | MEN D | TOT CY |
| PHAST Library | 1 | 1 | 1 |
| SYCL | 0.934 | 0.571 | 1.144 |
| Kokkos | 0.776 | 0.615 | 0.970 |
| CUDA | 0.695 | 0.580 | 0.456 |
| OpenCL | 0.434 | 0.290 | 0.402 |

**Table 4.15:** Average complexity ratio of PHAST Library with respect to the other frameworks across all the considered metrics.

Table 4.15 shows the average complexity ratio of PHAST Library with respect to SYCL, Kokkos, CUDA, and OpenCL across all the considered metrics. Overall, PHAST Library's results with respect to CUDA and OpenCL ones are not surprising, as PHAST Library is a high-level library designed to increase programmers' productivity, unlike the other two that are more focused on performance. They achieve this by giving *full control* to their users, thus limiting the abstraction level. OpenCL, moreover, has to manage the additional complexity deriving from its cross-platform nature.

Table 4.15 shows that PHAST Library has a reduced complexity with respect to *productive* frameworks too. It is, on average, more concise than both SYCL (93.4%) and Kokkos (77.6%). It is also less complex in the sense of MEN D than SYCL (57.1%) and Kokkos (61.5%). Even if, unlike these two frameworks, PHAST Library cannot take advantage of the expressiveness of lambda functions at the moment, it lacks of other sources of verbosity that are present in the other two, like the aforementioned Kokkos' initialization and a finalization instruction or SYCL's many objects that must be present in every program.

However, the main reason for the reduced length and complexity of PHAST Library code is given by what PHAST provides to its users: high-level containers and algorithms, both in host code and device code. Unlike Kokkos views or SYCL accessors, the containers can be iterated in sections of various shapes inside the algorithms, and not only accessed with indexes. These sections have a high level container-like interface that enables their manipulation in a concise way, hiding details relative to the thread-data mapping (i.e., thread indexes) and the nature of the global data-structure that is being accessed. PHAST Library in-functor algorithms, in particular, are a high-level formalization of common operations that can be achieved also in Kokkos and SYCL via their forms of nested/hierarchical parallelism, but not with the same conciseness,

expressiveness, and generality. Also, the presence of the parallelization parameters permit fine-tuning that does not mix with application-code and does not require code re-writings.

The only complexity metric where PHAST is behind is TOT CY in comparison with SYCL. This metric, however, is biased by the Triad benchmark, where PHAST Library's TOT CY value, which is 2, is twice SYCL's TOT CY value, which is 1. In that case, PHAST Library is *twice* as complex as SYCL. Considering the sum of the TOT CY scores of PHAST Library and SYCL, in fact, the former scores 29 and the latter 32.

In conclusion, PHAST abstraction mechanisms significantly decrease the resultant code complexity, even comparing it to high-level productive data-parallel frameworks. In this aspect, PHAST Library provides a measurable advancement of the state-of-the-art.

## 4.2 A real-world application: AES-based PRNG

The application in this section and its PHAST Library implementation have been presented in [135].

### 4.2.1 AES as PRNG

PRNGs are algorithmic ways to generate sequences of numbers that can be *reasonably* considered random in specific application contexts. As Donald Knuth observes, these sequences are important in many kinds of applications, such as simulation, sampling, numerical analysis, computer programming, decision making, cryptography, aesthetics, and recreation [87]. The Advanced Encryption Standard (AES) is a cryptographic algorithm that has been standardized in 2001 [111]. It has many properties such as speed, nonlinearity, and portability that make it a good fit for a high-quality PRNG [55].

Many AES implementations have been proposed so far. They are equivalent from a statistical point of view, but can differ in quality by other means. A common requirement of a *good* AES implementation is the immunity from side-channel attacks. Since many implementations make use of lookup-tables for performance reasons, this is not a common feature. However, recent implementations proved immune to cache-timing attacks by replacing lookup-tables and data-dependent branches with equivalent operations in Galois fields [78]. For instance, the so-called *SubBytes* step is an inversion in GF(256) [78].

Rewriting the entire algorithm as a series of Galois field operations means implementing AES as a sequence of atomic boolean instructions [18]. In this case, the most natural representation of AES bytes would be as 8 boolean variables each. Obviously, the performance loss would make the whole procedure impractical. A solution comes from a technique known as *bitslicing* [78] [97]. It is achieved by rearranging the bits of

many AES blocks in groups, so that equivalent bits of multiple bytes are packed in the same register. This way, the single atomic boolean operations can be performed between registers of a convenient size depending on the underlying architecture. For instance, in [78] Käsper et al. describe a fast AES implementation that takes advantage of 128-bit wide XMM registers and SSE instructions. Bit-slicing is, in the end, a technique to achieve bit-level parallelism and to process more AES blocks at once.

Another source of parallelism in AES can come from its modes of operation. Since it is a block cipher, it has many modes of operation that regulate how multiple blocks are encrypted. The most appropriate mode for pseudo random number generation is Counter (CTR) [55]. Electronic Code Book (ECB) mode is similar to CTR and it also can be used to implement a PRNG [97]. Both CTR and ECB modes require AES blocks to be encrypted independently, so both of them expose an intrinsic parallelism. Programmers can take advantage of it, for instance by using the multiple cores of a multi-core CPU or those of a GPU.

Various implementations can be found in the literature. However, the best implementations from a performance point of view are that presented by Käsper et al. in [78] for multi-core CPUs, and that presented by Lim et al. in [97] for NVIDIA GPUs.

In [78] Käsper et al. implement a cache-timing-attack resistant bitsliced AES encryption in counter mode for 64-bit Intel processors. Their implementation is 25% faster than the best of the previous ones, with 7.59 cycles per byte on an Intel Core 2. Their implementation has been implemented in assembly and QHASM [13] and is freely available on the authors' websites.

In [97] Lim et al. describe a deterministic AES-based PRNG written in CUDA C++. Their implementation is a cache-timing-attack resistant bitsliced AES in ECB mode. It achieves 78.6 Gbps on a GeForce GTX 480, 31-62% faster than the fastest previous implementation on similar devices. In their code, the authors addressed many common problems like coalesced accesses to main memory, shared memory conflict avoidance, fine register allocation to avoid spilling and maximize occupancy, and so on.

Both implementations are low-level, deeply optimized in the respective domains, and achieve great performance. In the following, a general implementation of a PRNG based on AES in ECB mode through PHAST Library is presented. It is compatible with both multi-core CPUs and NVIDIA GPUs. The implementation is then compared against the two specialized state-of-the-art implementations from performance and productivity standpoints.

### 4.2.2 Implementation

In AES block-cipher, data encryption is achieved by applying four basic operations ten to fourteen times, depending on the desired cryptographic strength required, on a data block.

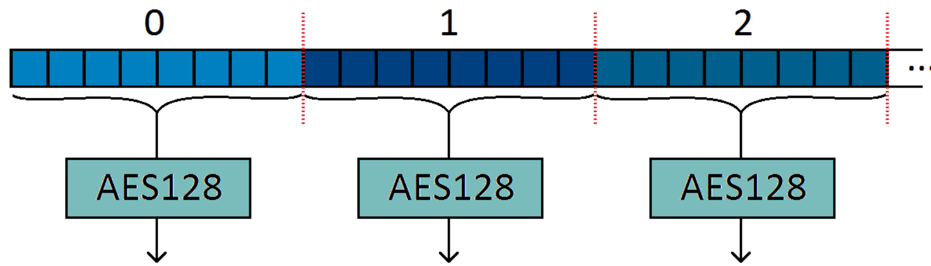



**Figure 4.11:** A grid of eight elements is applied to vector data, so to iterate over chunks of eight elements.

Such operations are commonly known as SubBytes, MixColumns, AddRoundKey, and ShiftRows [111].

In PHAST Library implementation, the application core is a **`for_each`** algorithm that works in parallel with chunks of eight variables at once as described in the cited papers. To achieve this, a **`grid`** of eight elements is applied to the **`phast::vector`** of the data and then iterated as shown in Figure 4.11 and in the following snippet.

```
1  // declare data-vector
2  phast::vector<uintN_t> data(n);
3  // define a grid of sub-vectors of size 8
4  phast::grid<phast::vector<uintN_t> > data_grid(data, 8);
5  // aes128_func is applied to each sub-vector in parallel
6  phast::for_each(data_grid.begin(), data_grid.end(), aes128_func);
```

**`aes128_func`** is an instance of the user-defined functor **`aes128`**. The 128-bit version of AES is considered here, therefore comprising 10 rounds of the basic operations. The functor can be declared as follows:

```
1  _FUNCTOR_HEAD(aes128)
2  template <typename T, unsigned int policy =
3      phast::get_default_policy()>
4  struct aes128 : phast::functor::func_scal<T, policy>
5  {
6      _PHAST_METHOD aes128(phast::matrix<T>& round_keys)
7      {
8          round_keys_.link(round_keys);
9      }
10
11     _PHAST_METHOD void operator()(phast::functor::vector<T>& data)
12     {
13         uintN_t x0, x1, x2, x3, x4, x5, x6, x7;
14         // data from phast::functor::vector data are bitsliced and
15         // saved in x0...x7 variables
16         bitslice(data, x0, x1, x2, x3, x4, x5, x6, x7);
17
18         add_round_key(x0, x1, x2, x3, x4, x5, x6, x7, round_keys_, 0);
```

```
19          // round 1-9
20          for(int round = 1; round <= 9; ++round)
21          {
22              sub_bytes(x0, x1, x2, x3, x4, x5, x6, x7);
23              shift_rows(x0, x1, x2, x3, x4, x5, x6, x7);
24              mix_columns(x0, x1, x2, x3, x4, x5, x6, x7);
25              add_round_key(x0, x1, x2, x3, x4, x5, x6, x7,
26                  round_keys_, round);
27          }
28          // round 10
29          sub_bytes(x0, x1, x2, x3, x4, x5, x6, x7);
30          shift_rows(x0, x1, x2, x3, x4, x5, x6, x7);
31          add_round_key(x0, x1, x2, x3, x4, x5, x6, x7,
32              round_keys_, 10);
33
34          // x0...x7 values are retrieved and stored back in
35          // phast::functor::vector data
36          ibitslice(data, x0, x1, x2, x3, x4, x5, x6, x7);
37      }
38
39      phast::functor::matrix<uintN_t> round_keys_;
40  };
```

In the functor code the four AES phases are clearly visible. As an example, also the implementation of the ShiftRows phase is shown below.

```
1   _PHAST_FUNCTION void shift_row(uint2_t& data)
2   {
3       uint2_t tmp(data);
4       data.at(0) = (tmp.at(0) & 0x0000ffffu)
5           | ((tmp.at(0) >> 4) & 0x0fff0000u)
6           | ((tmp.at(0) << 12) & 0xf0000000u);
7       data.at(1) = ((tmp.at(1) << 8) & 0x0000ff00u)
8           | ((tmp.at(1) >> 8) & 0x000000ffu)
9           | ((tmp.at(1) >> 12) & 0x000f0000u)
10          | ((tmp.at(1) << 4) & 0xfff00000u);
11  }
12
13  _PHAST_FUNCTION void shift_row(uint4_t& data)
14  {
15      uint4_t tmp(data);
16      // data.at(0) = tmp.at(0);
17      data.at(1) = (tmp.at(1) >> 8) | (tmp.at(1) << 24);
18      data.at(2) = (tmp.at(2) >> 16) | (tmp.at(2) << 16);
19      data.at(3) = (tmp.at(3) >> 24) | (tmp.at(3) << 8);
20  }
21
22  _PHAST_FUNCTION void shift_rows(uintN_t& x0, uintN_t& x1,
23      uintN_t& x2, uintN_t& x3, uintN_t& x4,
```

```
24     uintN_t& x5, uintN_t& x6, uintN_t& x7)
25 {
26     shift_row(x0);
27     shift_row(x1);
28     shift_row(x2);
29     shift_row(x3);
30     shift_row(x4);
31     shift_row(x5);
32     shift_row(x6);
33     shift_row(x7);
34 }
```

Function overloading permits the definition of two **shift_row** functions, the right one being called depending on the resolution of the **uintN_t** alias, which expands differently on different platforms: **uint2_t** on NVIDIA GPUs, a vector type wrapping two unsigned integers, and **uint4_t** on multi-cores, a vector type wrapping four unsigned integers, as in the following.

```
1 #ifdef _PHAST_USING_CUDA
2 using uintN_t = uint2_t;
3 #else
4 using uintN_t = uint4_t;
5 #endif
```

The introduction of **uintN_t** is not a necessary choice in terms of code-correctness, since both **uint4_t** and **uint2_t** are fully supported on multi-core processors and NVIDIA GPUs. All the application code could be implemented in terms of any of them without hampering heterogeneity, but performance would have suffered by this choice. In fact, looking at the code of the two state-of-the-art implementations, different kinds of variables are used: 128-bit integer SSE vector (i.e., **__m128i**) in Käsper et al. implementation, that naturally maps on **uint4_t**, and **vec2** in Lim et al. implementation, a class wrapping a CUDA **uint2** vector type with constructors and operator overloads that naturally maps on **uint2_t**.

ShiftRows is a straightforward function that does not require much effort to be implemented. AddRoundKey is also quite simple, requiring only eight XOR operations between the bitsliced variables and the round keys. These operations are expressed using classical C++ syntax, taking advantage of the operation overloading on PHAST Library vector types. MixColumns bitsliced implementation can also be expressed via common XOR operations as can be seen in appendix A in [78]. SubBytes, being the only phase that is not linear in GF(256), is maybe the most studied AES phase and many implementations can be found in the literature [18] [15] [97] [78]. In particular, in this work SubBytes has been implemented in two different versions: one using Boyar et al. work, clearly described in [15], and one based on Lim et al. work, thanks to the source code the authors made available for this work. Both these implementations, as well as

the other AES phases, are platform-agnostic and can be run on GPUs and multi-cores. For the performance and productivity analysis the second one has been adopted.

ShiftRows phase is a fitting example of the possibility to optimize PHAST Library code by using low-level architecture-specific constructs. In fact, this operation can take advantage of low-level instructions on SSE3-compatible processors (**_mm_shuffle_epi8**) and on CUDA devices (**__byte_perm**). In the following, *ISA versions* is used to denote the implementations that take advantage of architecture-specific optimizations, while *non-ISA versions* or *plain versions* are used to denote the others.

In the following code snippet, both **shift_row** functions (one for each managed vector type) have been optimized to take advantage of architecture-specific instruction sets. For each of them, the optimizations have been put under the scope of the *if* branch of an **ifdef** clause that checks what architecture is being used, using the technique described in Section 3.8. Conversely, the non-architecture-specific version has been maintained under the scope of the *else* branch. This way, both these functions are still multi-platform in their nature and can be executed on multi-cores and NVIDIA GPUs, but they also give the possibility to execute optimized code when executed on a specific architecture.

```
1  _PHAST_FUNCTION void shift_row(uint2_t& data)
2  {
3  #if defined(_PHAST_USING_CUDA)
4      data.at(0) = __byte_perm(data.at(0), data.at(0) >> 4, 0x7610u)
5          | ((data.at(0) << 12) & 0xf0000000u);
6      data.at(1) = __byte_perm(data.at(1), data.at(1) >> 4, 0x6701u)
7          | ((data.at(1) << 4) & 0x00f00000u);
8  #else
9      uint2_t tmp(data);
10     data.at(0) = (tmp.at(0) & 0x0000ffffu)
11         | ((tmp.at(0) >> 4) & 0x0fff0000u)
12         | ((tmp.at(0) << 12) & 0xf0000000u);
13     data.at(1) = ((tmp.at(1) << 8) & 0x0000ff00u)
14         | ((tmp.at(1) >> 8) & 0x000000ffu)
15         | ((tmp.at(1) >> 12) & 0x000f0000u)
16         | ((tmp.at(1) << 4) & 0xfff00000u);
17 #endif
18 }
19
20 _PHAST_FUNCTION void shift_row(uint4_t& data)
21 {
22 #if defined(_PHAST_USING_MULTI_CORE) && defined(_PHAST_SSSE3)
23     static const uint4_t mask( 0x03020100u , 0x04070605u ,
24         0x09080b0au , 0x0e0d0c0fu );
25     data = _mm_shuffle_epi8(data.data, mask.data);
26 #else
27     uint4_t tmp(data);
```

```
28     // data.at(0) = tmp.at(0);
29     data.at(1) = (tmp.at(1) >> 8) | (tmp.at(1) << 24);
30     data.at(2) = (tmp.at(2) >> 16) | (tmp.at(2) << 16);
31     data.at(3) = (tmp.at(3) >> 24) | (tmp.at(3) << 8);
32 #endif
33 }
```

In conclusion, PHAST Library is able to support the development of a performance-critical complex application like this AES-based PRNG. It allows to target it automatically on different architectures that, until now, were efficiently addressable with two different implementations, and permits also low-level *surgical* architecture-dependent optimizations, reducing the lines of code where this happens to just a few.

### 4.2.3 Performance study

PHAST Library code can run on both multi-core CPUs and NVIDIA GPUs. The performance study has been done on the multi-core CPUs listed in Table 4.1 and the NVIDIA GPUs listed in Table 4.2.

#### Multi-core CPUs

PHAST Library implementation is inspired by Käsper et al. [78] for the MixColumn step, which is described in the appendix of their paper [78]. For SubBytes, the pseudo-code described in [15] has been preferred, because it is one of the fastest and it is general. Käsper et al. is probably the fastest implementation, but it is not general as it explicitly requires SSSE3 ISA support.

For multi-core CPUs, as an overall reference, also the highly tuned and architecture-specific bitsliced AES encryption in counter mode described in [78] (Käsper-Schwabe in the following) is reported. It constitutes a landmark in terms of absolute performance for CPUs, despite not being usable on different architectures (i.e., multi-core CPUs without the required x86 SSE ISA support and GPUs). Its source code available at the authors' site is a thin C function wrapper that invokes assembly code. It is a thread-safe function, but the whole benchmark is a sequential implementation and no work-partition or thread management is done. For this reason, Table 4.16 shows a performance comparison on different multi-core CPUs with PHAST implementations launched with thread-number parameter set to 1.

Then, PHAST Library code has been implemented in two slightly different versions: one that takes advantage of low-level architecture-specific SSE instructions (ISA version) and one that does not (plain version), to witness the library capability to reach the machine specific features when needed.

Furthermore, the reference code has been implemented using the exact same versions of all the algorithms employed in the PHAST Library implementation, but without the

| | no-lib plain | PHAST plain | no-lib ISA | PHAST ISA | *Käsper-Schwabe* |
|---|---|---|---|---|---|
| meeseeks | 1.85 | 1.82 | 3.30 | 3.26 | *5.09* |
| maxi | 1.37 | 1.31 | 2.42 | 2.33 | *3.99* |
| elwood | 0.56 | 0.59 | - | - | - |
| golia | 1.53 | 1.46 | 2.65 | 2.56 | *4.18* |

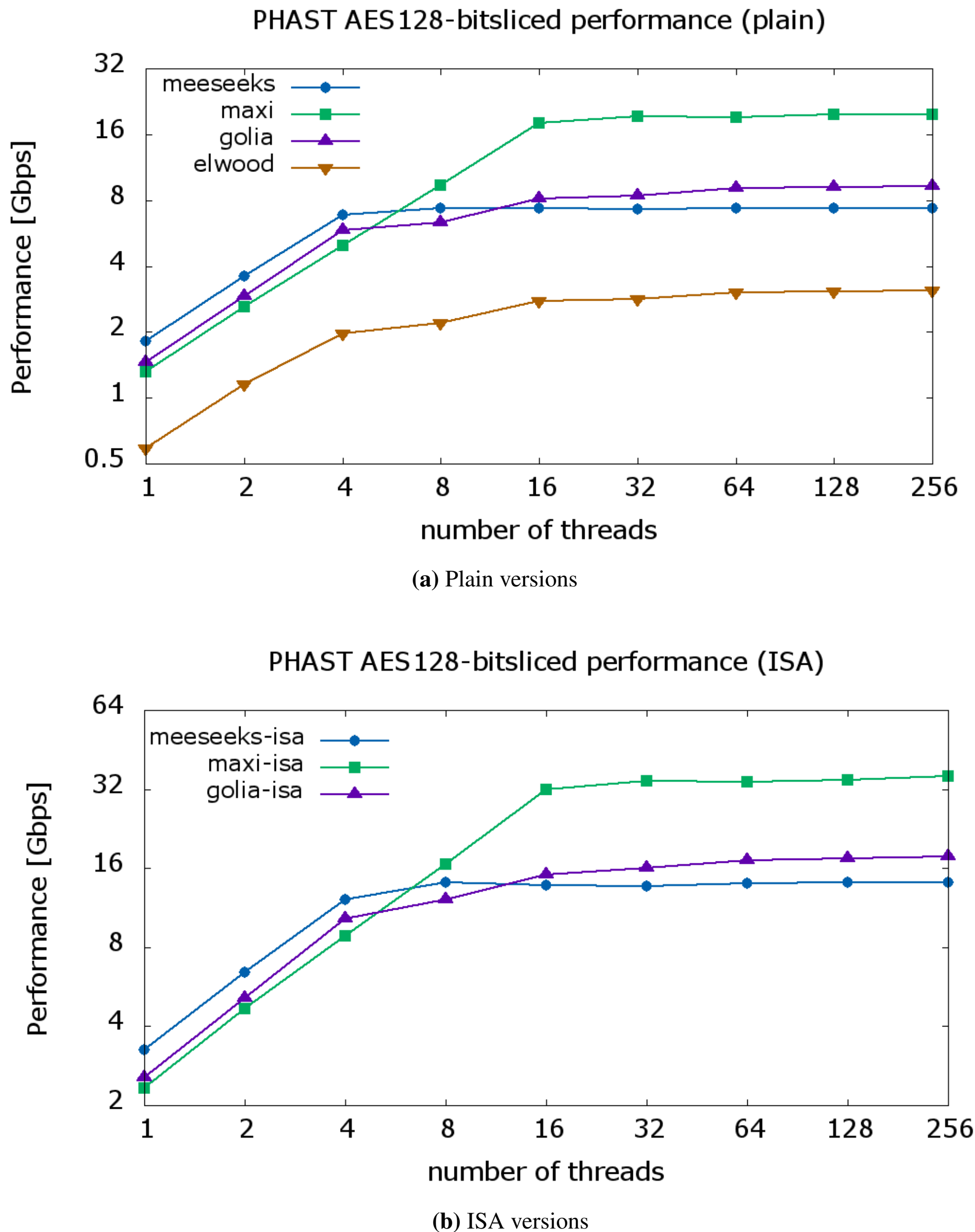
**Table 4.16:** Performance comparison between single-thread AES PRNGs on CPUs, measured in Gbps. Käsper-Schwabe employs a different algorithm for SubBytes AES step.

parallel support of the library. Also in this case with two minor variations, adopting SSE extensions (no-lib ISA) or not (no-lib plain). This way, it is possible to assess the exact overhead induced by the abstractions and heterogeneous parallelization facilities provided by PHAST Library.

It is important to underline that *elwood* is not an SSE3-compatible processor, and thus the only version that can run on it is the plain PHAST Library version. In fact, it does not use architecture-specific instructions and it is guaranteed to be portable across various multi-core CPUs.

Table 4.16 shows that in single-thread conditions on the same code, the maximum library overhead amounts to 4.79%, measured on *golia*. That number can be regarded as a measure of the performance impact of PHAST Library on single-core applications. Both ISA-enhanced and plain versions highlight a similar limited overhead across different CPUs. The Table highlights also a quite big difference in performance between Käsper-Schwabe and PHAST Library, with Käsper-Schwabe being 1.7x faster than PHAST ISA in the worst case. This performance difference is mainly due to the SubBytes algorithm, which in the case of Käsper-Schwabe is extremely tuned in assembly for full exploitation of the specific SSE3 ISA support available in some CPUs.

Furthermore, one of the main advantages of PHAST Library is the ability to automatically exploit the aggregate computational power of multi-core CPUs using more than a single thread. This approach requires that the underlying hardware has multiple processing units (e.g., cores in CPUs) and that all of them have *enough* work to do to hide the costs of thread allocation and management. PHAST Library can determine such hardware parallelism automatically, or the user can explicitly call an API function to force the desired number of runtime threads. Figure 4.12 shows the performance variation when varying the number of threads used. Each of the multi-core CPUs used achieves an almost linear performance scaling up to the number of available *physical* cores, with a further smaller gain in processors having hyper-threading technology (all but *elwood* machine, that mounts an AMD Phenom II processor). From the methodological viewpoint, the amount of data each thread works with has been kept constant so to avoid 'noise' due to caching effects in the cores.




**(a)** Plain versions



**(b)** ISA versions

**Figure 4.12:** AES performance on multi-core CPUs using PHAST Library in correspondence of different numbers of threads.

The intrinsic support of parallel execution with limited overhead allows PHAST Library implementation to automatically reach far better performance than the Käsper-Schwabe single-thread version on all the considered architectures. In fact, taking advantage of all the cores available, PHAST Library performance reaches around 4x Käsper-Schwabe performance on *maxi*. The same implementation permits to trivially harness the increasing overall computational power available in successive generations of processors, since it scales nicely with the number of cores and it would take advantage

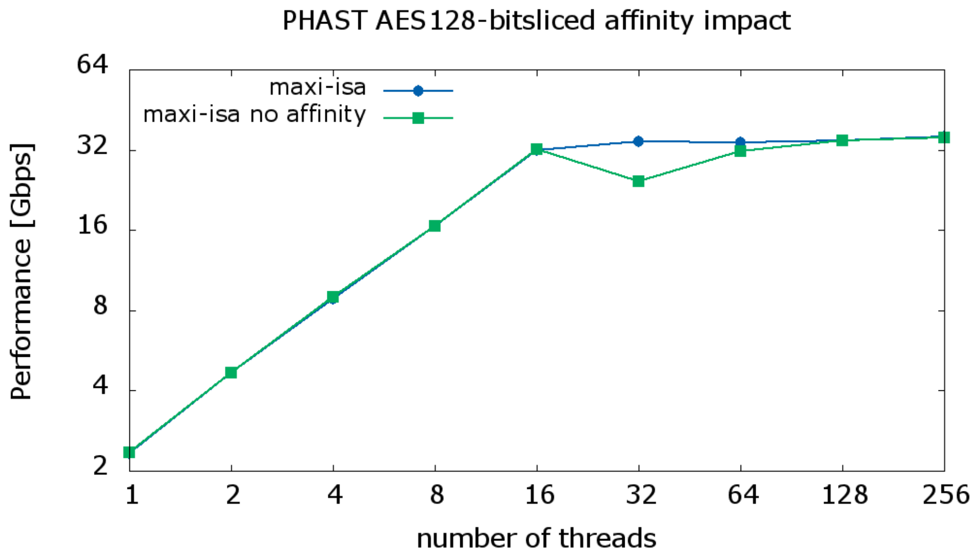



**Figure 4.13:** The impact of affinity management on multi-core performance.

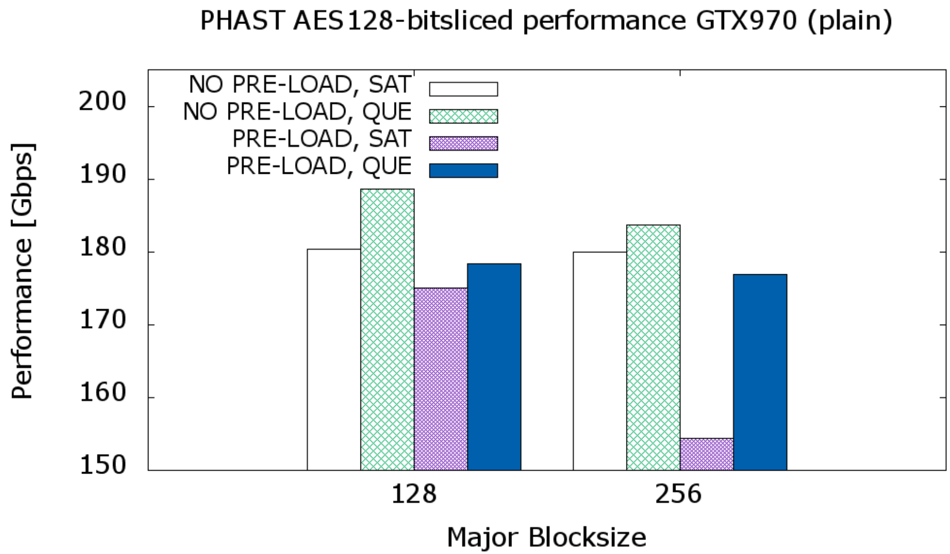
also of newer processors with an even higher degree of parallelism.

As previously cited in Section 3.6, PHAST Library also manages thread affinity. It is another user-adjustable parameter, but default behavior is usually the best choice. It is set to spread the available threads across the physical packages, then across the physical cores, and uses hyper-threading only as a last resource, in a way that is similar to the *scatter* policy of OpenMP [126]. This way, thread locality is minimized and the maximum amount of cache space and functional units is assigned to every thread. The impact on performance of affinity can be seen in Figure 4.13, where the green curve has been obtained with affinity management shut down. It can be seen that for AES, despite affinity management impact on performance is negligible when few threads are used, it becomes critical when all the logical cores are occupied.

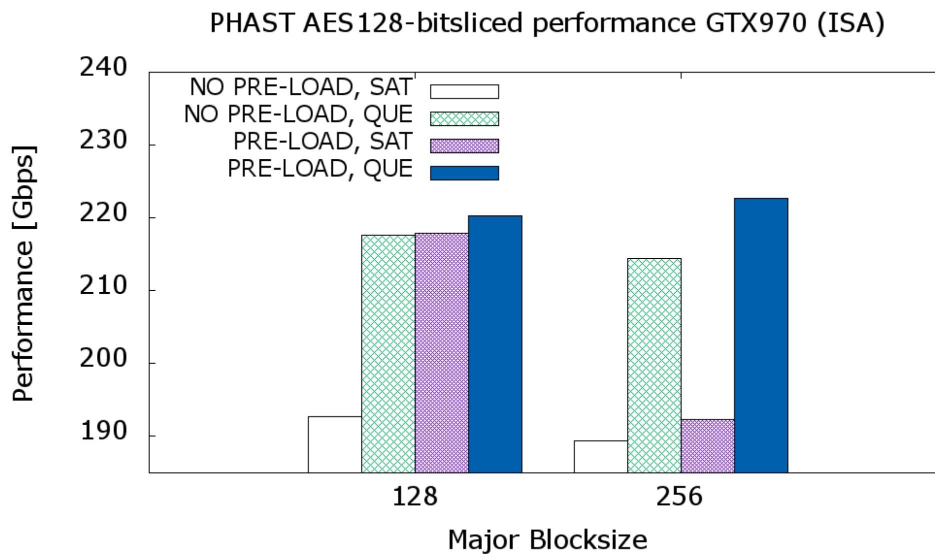
### NVIDIA GPUs

The GPUs used are summarized in Table 4.2. In this case, PHAST Library provides many user-adjustable parameters that reflect the degrees of freedom present in the CUDA programming model [119]. As explained in Section 3.6, all of these parameters can be varied without altering the structure of the source code.

Apart from **`minor_block_size`**, set to 1 since there is no in-functor parallelism to be exploited in this case, the optimum choice of the other parameters is not trivial and some fine tuning is needed. Figures 4.14 and 4.15 show the performance achieved in correspondence of different parameters. The best **`major_block_size`** values have been experimentally identified as 128 and 256, so no other values are showed in the figures for brevity. It can be seen that the best configurations are the same across the




**(a)** GTX970 plain version

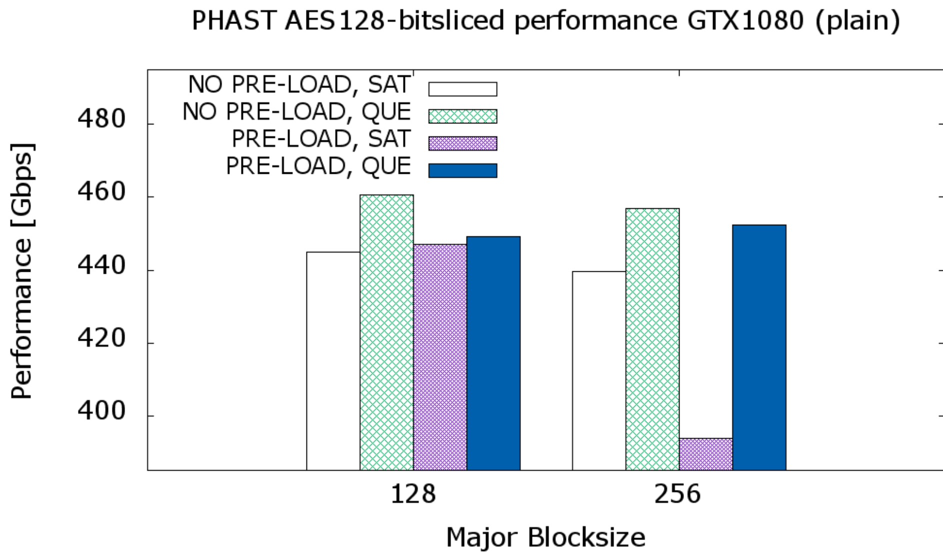



**(b)** GTX970 ISA version

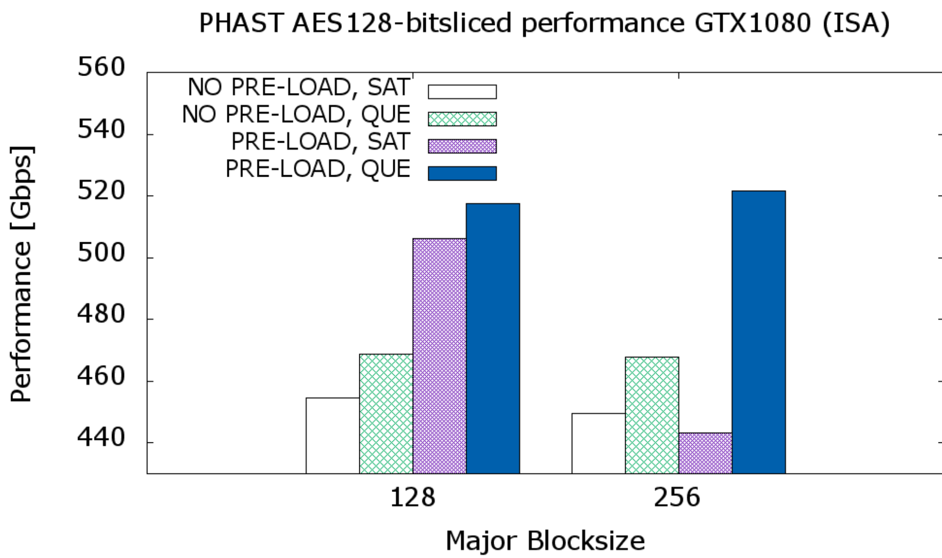
**Figure 4.14:** AES performance on GeForce GPU GTX970 in correspondence of different combinations of **`major_block_size`**, **`shared_pre_load`** and **`scheduling_strategy`**.

two considered GPUs, but not the same across plain – ISA versions. The figures also highlight that all the available parameters can have a significant impact on performance, so they must be carefully managed.

The reference implementation on NVIDIA GPUs is the bitsliced AES PRNG in ECB mode described in [97] (Lim-Petzold-Koç in the following). The SubBytes phase described there is a better fit for GPUs with respect to the one contained in [15], giving a maximum performance gain of 2%. Lim-Petzold-Koç has some degrees of freedom




**(a)** GTX1080 plain version




**(b)** GTX1080 ISA version

**Figure 4.15:** AES performance on GeForce GPU GTX1080 in correspondence of different combinations of `major_block_size`, `shared_pre_load` and `scheduling_strategy`.

too: block-size and grid-size. The optimum parameters in its case are: block-size set to 512 on both GPUs, and grid-size set to 512 on the GeForce GTX970 and to 8192 on the GeForce GTX1080.

Table 4.17 shows the achieved performance on the GTX970 and GTX1080 in correspondence of the respective optimum configurations. The performance loss of PHAST Library with respect to Lim-Petzold-Koç amounts to 11.04% on the GTX970, and 9.45% on the GTX1080. Considering that PHAST Library code is common to

| | Lim-Petzold-Koç | PHAST plain | PHAST ISA |
|---|---|---|---|
| GTX970 | 247.31 Gbps | 188.57 Gbps | 222.72 Gbps |
| GTX1080 | 570.72 Gbps | 452.36 Gbps | 521.45 Gbps |

**Table 4.17:** Performance comparison between AES PRNG implementations on NVIDIA GPUs.

both CPUs and GPUs, that it automatically accommodates different CPU and GPU architectures, and that it is expressed at a higher level of abstraction, such an overhead does not appear to be a limiting factor in PHAST adoption.

### 4.2.4 Productivity study

| | SLOC | MEN D | TOT CY | supported target |
|---|---|---|---|---|
| PHAST plain | 475 | $5.990 \times 10^6$ | 8 | CPU, GPU |
| PHAST ISA | 500 | $6.984 \times 10^6$ | 8 | CPU, GPU |
| no-library plain | 542 | $9.668 \times 10^6$ | 11 | CPU |
| no-library ISA | 567 | $1.068 \times 10^7$ | 11 | CPU |
| Käsper-Schwabe | 226 | $3.412 \times 10^5$ | 11 | CPU |
| assembly | 8200 | | | CPU |
| Lim-Petzold-Koç | 839 | $1.281 \times 10^7$ | 68 | GPU |

**Table 4.18:** Complexity metrics calculated on different AES PRNG implementations: Source Lines Of Code (SLOC), Halstead's Mental Discriminations (MEN D), and McCabe's Total Cyclomatic Complexity (TOT CY).

Table 4.18 shows some metrics calculated on the source files of the different AES versions discussed so far. The adopted metrics are SLOC, MEN D, and TOT CY, already described in Section 4.1. Käsper-Schwabe has been split in two since no metrics can be calculated on assembly source files, apart from SLOC. In the case of PHAST Library and *no-library*, both plain and ISA versions are analyzed, in order to show how much complexity increases when low-level optimizations are added. In this case, the addition of low-level optimizations acts similarly on the two versions: it increases SLOC of 25 in both cases, and it increases MEN D of $9.940 \times 10^5$ in PHAST implementation and $1.012 \times 10^6$ in no-library implementation.

Considering the programs where all the metrics can be calculated, both PHAST versions score the best, even better than any no-library version, which are sequential CPU-only programs. It must be also considered that both PHAST versions are compatible with multi-core processors and NVIDIA GPUs, unlike the others that are platform-specific

implementations. In this regard, each PHAST Library program can be seen, after all, as being *two programs* in one.

It is important to notice that these metrics are calculated on programs written by experts, and so the resulting differences can be considered intrinsic of the languages, APIs and libraries used.

In fact, assembly is well-known for its low-level nature that inevitably leads to verbose code, and the 8200 lines of code in Käsper-Schwabe implementation testify just that. Similar evaluations can be done on the many complex lines of code in Lim-Petzold-Koç implementation: CUDA has many details and many possible low-level optimizations to take care of, and all of them must be addressed to achieve cutting-edge performance. By these means, the use of PHAST Library can improve programmers' productivity by allowing them to write smaller, simpler, and more expressive code.

### 4.2.5 Summary

A challenging real-world application, a bitsliced cache-timing-attack resistant AES-based PRNG, has been implemented with PHAST Library. The comparison with state-of-the-art architecture-specific solutions highlights that PHAST Library code is smaller and simpler, thanks to its high-level constructs and its capacity of managing parallelization parameters with a limited programming effort. For instance, PHAST Library source length is around -40% with respect to the reference CUDA implementation and around -12% with respect to the sequential C++ CPU version.

PHAST Library performance resulted comparable to native versions of the same algorithm on both the families of architectures considered. Specifically, it delivers only a 5% overhead in case of single-threaded CPU version and about 10% for different NVIDIA GPU boards. Moreover, PHAST CPU version is able to automatically scale almost linearly up to the number of available cores, easily surpassing the single-core fine-tuned assembly implementation.

In brief, PHAST Library is mature enough to be used to develop a complex application that has been refined and optimized in-depth, leading to two state-of-the-art implementations for two different families of devices. PHAST Library is able to provide a single-source implementation that is shorter and less complex than any of the other two. It also performs well on both multi-core CPUs and NVIDIA GPUs and limits the necessity to express device-specific code to a few lines.

## 4.3 Random task-DAGs

This Section presents results that have been already discussed in a paper accepted for publication, currently in pre-production phase. That paper, in turn, is the extension of a

previous work [133]. It contains the discussion of the details of the task-programming facilities of PHAST Library: there, a family of randomly generated task-based applications is evaluated with respect to hipSYCL [2], a state-of-the-art SYCL implementation, from both performance and productivity standpoints.

### 4.3.1 Benchmark application

PHAST Library abstractions that allow task-parallelism can be evaluated by taking into account several applications that can be modeled as task-DAGs with different sizes and structures. The functions executed by the tasks can expose data-parallel calculations, in order to show the integration between the two kinds of parallelism supported.

Since the PHAST Library tasks aim to model *general* task-based parallel applications, their evaluation must be representative of a class of applications that is as broad as possible. A common approach in this regard is to randomly generate synthetic workloads [24, 33] and conduct evaluations on them. Random task-DAGs allow exploring a variety of different graphs, in a controlled way, that can be representative of different applications, both current and future.

In [24] Cordeiro et al. discuss different techniques to randomly generate task-DAGs for evaluating scheduling algorithms. They dissect various techniques, as to highlight which one can best match any given context and its constraints. They also present the GGen tool to generate DAGs. It is freely available and can be used in multiple contexts where an extensive number of different task-DAGs needs to be randomly generated.

The GGen tool's Fan-in / Fan-out technique is here used to generate the random-graphs of the benchmark set. The graphs generated this way have nodes with a varying number of edges, constrained on the maximum number that enter each node (fan-in) and the maximum number that leave each node (fan-out). The Fan-in / Fan-out method "emulates the scatter/gather phases of parallel applications" [24] and can be used to model plausible graphs that arise in various applications, like for example Quicksort or FFT [33]. Moreover, since the graphs are generated with a freely available tool and the parameters used for the generation are specified, these setups are easy to reproduce.

In the adopted DAG model, as explained in Section 3.9, an edge connecting two nodes represents a data-flow dependency between the two: the value returned by the callable executed in the source node, when ready, will be used as a parameter in the callable executed in the destination node. This determines a relationship between the number of arguments of a node's callable function $N_a$ and the number of edges $N_e$ entering that node that is simply $N_a \geq N_e$. Thus, the choice of the function executed inside each node is made so to actually use the arguments received from predecessor tasks.

The maximum fan-in value determines the number of callables that will be invoked

inside the task associated to the node. More precisely, it is sufficient to implement up to as many callables as the fan-in possible values. Inside these callables, data-parallel algorithms are invoked, in order to evaluate the mixing between data-parallel and task-parallel code.

Overall, DAGs with 4, 8, 16, 32, 64, 128, 256, and 512 nodes are generated, having 1, 2, or 3 as maximum fan-in and fan-out values. As for the size of the data processed by each task, 32×32, 128×128, and 512×512 matrices of **double** values are used. In total, the benchmark set comprises 216 different applications. Table 4.19 summarizes these setups.

| Parameter | Values |
|---|---|
| Number of nodes | 4, 8, 16, 32, 64, 128, 256, 512 |
| Fan-in | 1, 2, 3 |
| Fan-out | 1, 2, 3 |
| Data size | 32×32, 128×128, 512×512 |

**Table 4.19:** Parameters of the randomly generated benchmark applications. Every combination of values has been considered.

The considered applications therefore comprise task-based elaborations where individual steps possibly expose data-parallel algorithms. These tasks can be modeled by the **task** as well as the **stream_task** classes. In the first case, the whole benchmark would execute on an input and produce an output. In the second case, the whole calculations could be applied to a stream of inputs that would produce a stream of outputs. As an explanatory example and for the current evaluation, the first case only is discussed.

In the following, some source code of the callable functions employed is shown, and thus the data-parallel computations performed, in the nodes of the randomly generated graphs.

Since the admitted values of fan-in are 1, 2, or 3, callables with 1, 2, or 3 parameters have been defined. Also, another one with no parameters that is used in the nodes without predecessors.

#### Callable 'callable0()'

```
1 std::shared_ptr<phast::matrix<double>> callable0(const int n)
2 {
3     phast::matrix<type>* p_mat = new phast::matrix<double>(n, n);
4     phast::fill(p_mat->begin_ij(), p_mat->end_ij(), 2.0);
5
6     return std::shared_ptr<phast::matrix<double>>(p_mat);
7 }
```

**callable0** is used in nodes without predecessors in the task-DAG. It allocates a new

square **matrix** with a side read as argument, it fills it with 2.0 and then returns it by wrapping it in a **shared_ptr**. The returned smart pointer will be read by the subsequent tasks.

#### Callable 'callable1()'

```
template <typename T, unsigned int policy =
    phast::get_default_policy()>
struct scaler : phast::functor::func_scal<T, policy>
{
    _PHAST_METHOD scaler(T factor) : factor_(factor) {}
    _PHAST_METHOD void operator()(phast::functor::scalar<T>& scal)
    {
        scal *= factor_;
    }
    T factor_;
};

std::shared_ptr<phast::matrix<double>> callable1(
    std::shared_ptr<phast::matrix<double>> sh_mat1)
{
    const double scal_factor = get_scal_factor();

    phast::matrix<double>* p_mat =
        new phast::matrix<double>(*sh_mat1.get());
    phast::for_each(p_mat->begin_ij(), p_mat->end_ij(),
        scaler<double>(scal_factor));

    return std::shared_ptr<phast::matrix<double>>(p_mat);
}
```

**callable1** is associated to every node in the generated task-DAG that has only a single preceding node. It copies an input matrix into a newly allocated one and scales it by a factor **scal_factor** that is retrieved via a helper function. The copy is done because the input matrix could be an input to another concurrent task in the task-DAG and, thus, modifying it could lead to data corruption.

#### Callable 'callable2()'

```
std::shared_ptr<phast::matrix<double>> callable2(
    std::shared_ptr<phast::matrix<double>> sh_mat1,
    std::shared_ptr<phast::matrix<double>> sh_mat2)
{
    phast::matrix<double>* p_mat = new phast::matrix<double>(
        sh_mat1.get()->size_i(), sh_mat1.get()->size_j());

```

```
8       if(rand() & 0x1)
9       {
10          phast::for_each(sh_mat1.get()->begin_ij(),
11              sh_mat1.get()->end_ij(), sh_mat2.get()->begin_ij(),
12              p_mat->begin_ij(), phast::plus<double>());
13      }
14      else
15      {
16          phast::dot_product(*sh_mat1.get(), *sh_mat2.get(), *p_mat);
17      }
18
19      return std::shared_ptr<phast::matrix<double>>(p_mat);
20  }
```

**callable2** receives two matrices from two preceding threads as input. It allocates a new matrix and, with a probability of 50%, it uses it to store the sum or the dot-product between the two input matrices.

#### Callable 'callable3()'

**callable3** is similar to **callable2**: it receives three matrices in input. It also can sum or calculate the dot-product between the first two input matrices and store it in a newly allocated matrix. Then, it sums this matrix to the one received as third parameter.

#### Code in main function

In order to show how the task-DAG definition code looks like in **main.cpp**, a sample snippet that is representative of all the benchmarks is shown instead. It is relative to benchmark 8-1-2, where 8 is the number of nodes in the task-DAG, 1 is the maximum fan-out value, and 2 is the maximum fan-in value. Figure 4.16 shows the associated task-DAG. Looking at the DAG, it is evident that the constraints have been met in its generation. For instance, every node has maximum 2 preceding nodes.

```
1  auto task0 = make_task(callable0, n);
2  auto task1 = make_task(callable0, n);
3  auto task2 = make_task(callable1, task1);
4  auto task3 = make_task(callable0, n);
5  auto task4 = make_task(callable2, task0, task3);
6  auto task5 = make_task(callable0, n);
7  auto task6 = make_task(callable2, task2, task4);
8  auto task7 = make_task(callable2, task5, task6);
9  task7.get();
```

The non-task arguments can be clearly seen in the code. They are **n**, the number that will be used as the side of the instantiated square matrices.

In this code is also evident what explained in Section 3.9: any of the parameters

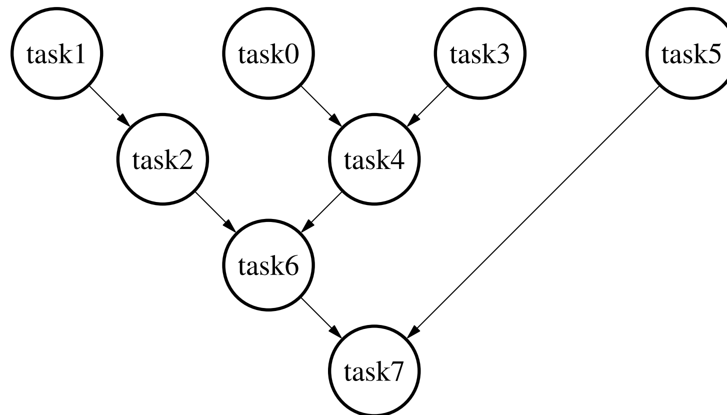



**Figure 4.16:** Example of the task-DAG corresponding to benchmark 8-1-2.

of the callable can be replaced with a task wrapping another callable that returns the needed value. For instance, **`callable1`** has been shown as a callable function that accepts a shared pointer to matrix parameter. Instead of passing such shared pointer during construction, a task that will output that shared pointer is passed instead, and so a precedence relation is defined between the two. Conversely, **`task0`** is an independent task that can directly process its integer input, and thus will be executed before the others that depend on it. So, the **`task`** abstraction class is able to express dependencies between tasks and also to manage the link between them so that output data is generated before being consumed by subsequent tasks.

### 4.3.2 Evaluation

To evaluate PHAST Library's task-related capabilities, its performance and productivity is compared against a SYCL platform for the benchmarks here described.

SYCL [84] is a heterogeneous, productivity-oriented open standard, which is getting increasing attention from the research and industrial communities. As briefly analyzed in Subsection 2.3.3, it supports the generation of a DAG of dependencies, even though tasks are not fully heterogeneous because the nodes in the implicitly generated task-DAGs are limited to execute on the same device. Of the various existing SYCL implementations, hipSYCL [2] has been chosen because it natively supports multi-core CPUs and NVIDIA GPUs. As previously explained, it implements the SYCL specification by wrapping OpenMP on the multi-core side [126], CUDA on the NVIDIA GPU side [119], and also AMD HIP [4].

The productivity comparison is done in terms of the complexity metrics described in Section 4.1: Source Lines Of Code, i.e., the sheer length of a program, Halstead's Mental Discriminations [52], which expresses the program complexity via the number of symbols needed to express that program, and McCabe's Total Cyclomatic Complexity [99],

which shows how many paths the program can take during its execution.

Also the performance of the PHAS Library implementation is evaluated with respect to the hipSYCL implementation in the case of all the 216 generated benchmarks summarized in Table 4.19. For each of them, the median of the execution time of 20 runs is adopted as the measure.

In order to execute each benchmark, the tasks must be assigned to the available devices, as each of them can have a callable where PHAST Library algorithms are executed on a multi-core CPU or NVIDIA GPU. There are no restrictions on the task assignment as any assignment is supported: any node, in fact, can be subsequent or predecessor of tasks that execute on its same device or on a different one. Any data returned by a predecessor would be converted automatically if the devices associated to the two tasks do not match. However, hipSYCL supports task assignments only to one kind of architecture at a time. For this reason, the following evaluation will consider either all tasks mapped on a multi-core CPU, or all tasks assigned to an NVIDIA GPU.

In the SYCL specification, in fact, the implicit task-DAG is generated by submitting command groups to a queue object [84]. Inside these command groups, accessors to buffers are declared and marked as read-only, write-only, or read-write. Depending on the access mode specified, the submitted command group can execute concurrently with respect to other command groups or be enqueued to wait for their completion if there are data dependencies. This structure is an implicit task-DAG, and it is constructed by invoking the submit method of the queue. However, as can also be seen in the SYCL API specification [84], a queue is associated to a single device that is decided at construction site. For this reason, it is not possible to express a graph of multiple command groups that execute on different devices.

#### Target architecture

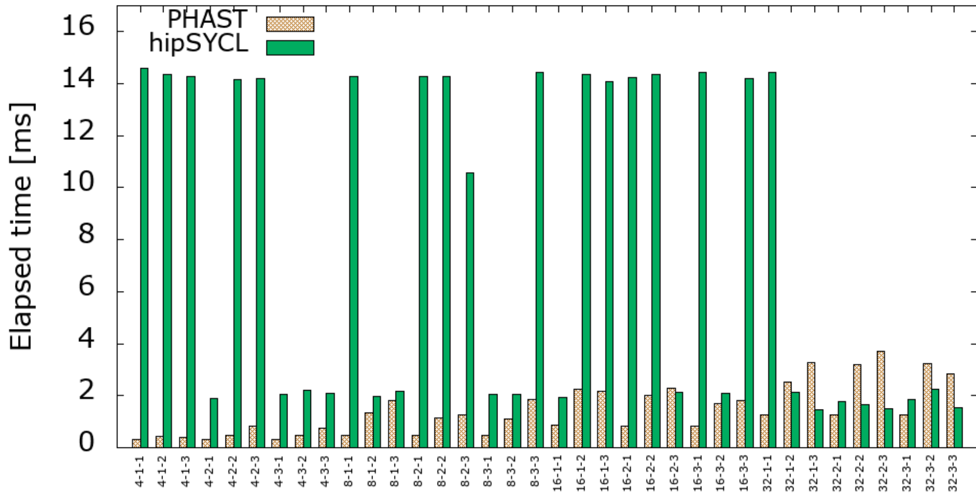
The experiments have been conducted on a high-performance workstation mounting an Intel i9 9900K at 3.60 GHz with 8 cores hyper-threaded, and an NVIDIA GPU GeForce RTX 2080 with 2944 CUDA cores clocking at 1.71 GHz maximum frequency and 8 GByte DDR6 RAM.

### 4.3.3 Discussion

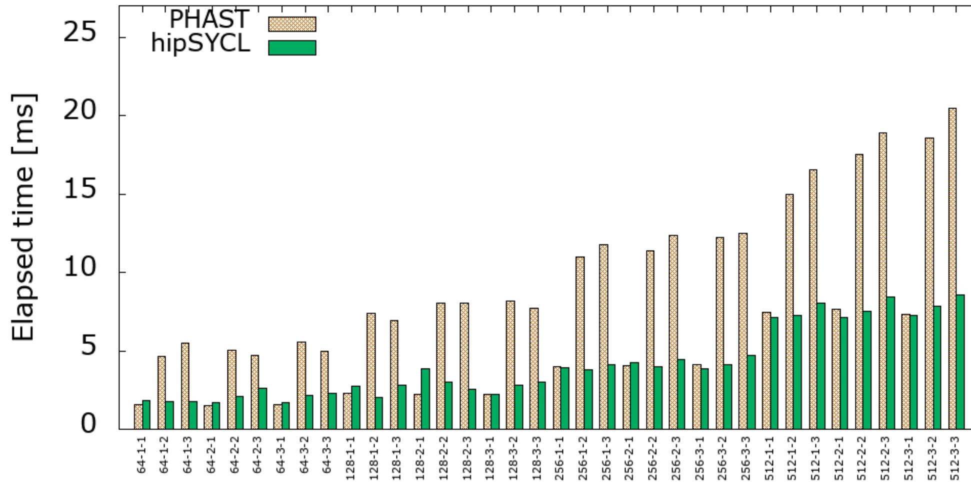
#### Performance analysis in case of multi-core CPU target

Figures 4.17, 4.18, and 4.19 illustrate the performance of PHAST Library compared to SYCL in case of a range of benchmarked task-DAGs for three levels of *computation* load in the tasks: light, medium and high, corresponding to the elaboration of 32x32, 128x128, and 512x512 matrices. For example, for 32x32 matrices, Figures 4.17a and

4.17b show results for task-DAGs with a smaller (4 to 32) and bigger (64 to 512) number of nodes, respectively. Similarly, Figures 4.18a-4.18b and Figures 4.19a-4.19b do the same for 128x128 and 512x512 matrices, respectively.




**(a)** Nodes 4, 8, 16, 32




**(b)** Nodes 64, 128, 256, 512

**Figure 4.17:** Comparison of elapsed times of task-DAG benchmarks on multi-core CPU with workload of 32x32 matrices. Labels are N-Fo-Fi: N is the number of nodes, Fo is the maximum fan-out, Fi is the maximum fan-in.

Figure 4.17 highlights that for small task-DAGs, i.e., for a number of nodes between 4 and 16, PHAST outperforms SYCL. On average, the smaller the task-DAG, the bigger the speedup achieved by PHAST Library. In particular, the majority of SYCL results are stuck to around 14 ms independently of the task-DAG features. SYCL probably hits some baseline overhead that shadows the computation-dependent execution time.

Here PHAST Library delivers speedups from +20% up to 35x on average, except the configuration 16-2-3 where it slows down about 7%.

No clear trend can be seen in the relationship between execution time and fan-in / fan-out in SYCL for smaller DAGs. Conversely, PHAST Library seems to have a clear correlation with the fan-in value that becomes more evident with bigger task-DAGs: the higher the fan-in, the bigger the advantage of SYCL. This effect has been investigated starting with an observation: as explained in the previous chapter, when fan-in is greater than one, tasks calculate their parameters by launching **`std::async`** calls. In order to assess the role of thread spawning in the slowdown, they have been removed by integrating a thread pool library into the task class and replaced **`std::async`** calls with pushes in the pool. Also, the data-parallel algorithms have been modified so to spawn 1 less thread by re-using the calling threads. After this intervention, all the benchmarks with fan-in greater than 1 in Figure 4.17b improved. For instance, The improvement is 55.17% in 512-3-3, 53.59% in 512-3-2, 45.79% in 256-3-2, and 39.11% in 256-3-3, resulting in performance in line with SYCL.

This preliminary study demonstrates that a thread-pool can bridge the performance gap where PHAST Library was showing some slowdown compared to SYCL. A few thread pool libraries have been tested and, despite gathering these encouraging indications, unfortunately they presented some instabilities due to the interaction between the thread pool and the task-DAG structure. It appears additional investigation is required and, most probably, a slightly tailored thread-pool architecture matching the specific usage pattern. This research and development path will be followed as future work.

Table 4.20 shows the geometric mean of the ratio between the execution time of SYCL and PHAST Library versions of the benchmarks with 32x32 matrices as workload. It indicates that PHAST Library is 1.56x faster than SYCL, overall. The table shows that the speedup is higher for smaller task-DAGs (about 13x, 6x, and 5x for 4, 8, and 16 nodes respectively). Then, for 32 nodes performance are similar and for bigger task-DAGs (i.e., from 64 to 512 nodes) SYCL is between 1.73x to 2.0x faster than PHAST Library due to the thread-spawning effect described above.

Figure 4.18 shows that also for 128x128 matrix workloads, PHAST Library is on average quite faster than SYCL for the range of small task-DAGs of the benchmark set (i.e., from 4 to 32 nodes), with speedups reaching up to 41x (case 4-3-1) and with decreasing advantage as the DAG size increases. SYCL seems to expose a baseline overhead also in this case when the DAG is small as a number of benchmarks execute in about 12-17 ms. As Table 4.20 highlights, PHAST Library goes from being about 5x faster for 4, 8 and 16 nodes, down to 2.9x for 32 nodes and then SYCL results faster for bigger task-DAGs.

Also in this case, the observed results highlight a slowdown due to thread spawning

| work size | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | Total |
|---|---|---|---|---|---|---|---|---|---|
| 32x32 | 13.61 | 6.08 | 4.94 | 0.96 | 0.58 | 0.54 | 0.50 | 0.58 | 1.56 |
| 128x128 | 5.16 | 5.03 | 4.79 | 2.90 | 0.69 | 0.65 | 0.74 | 0.80 | 1.77 |
| 512x512 | 2.06 | 2.63 | 1.32 | 1.54 | 1.15 | 1.03 | 1.02 | 0.96 | 1.38 |
| Total | | | | | | | | | 1.56 |

**(a)** multi-core CPU

| work size | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | Total |
|---|---|---|---|---|---|---|---|---|---|
| 32x32 | 2,39 | 3,08 | 1,64 | 1,24 | 1,12 | 1,19 | 1,49 | 1,50 | 1,61 |
| 128x128 | 2,39 | 2,73 | 1,86 | 1,97 | 2,04 | 1,90 | 2,34 | 2,62 | 2,21 |
| 512x512 | 3,45 | 4,68 | 5,41 | 5,34 | 5,24 | 4,85 | 5,27 | 5,90 | 4,96 |
| Total | | | | | | | | | 2.60 |

**(b)** NVIDIA GPU

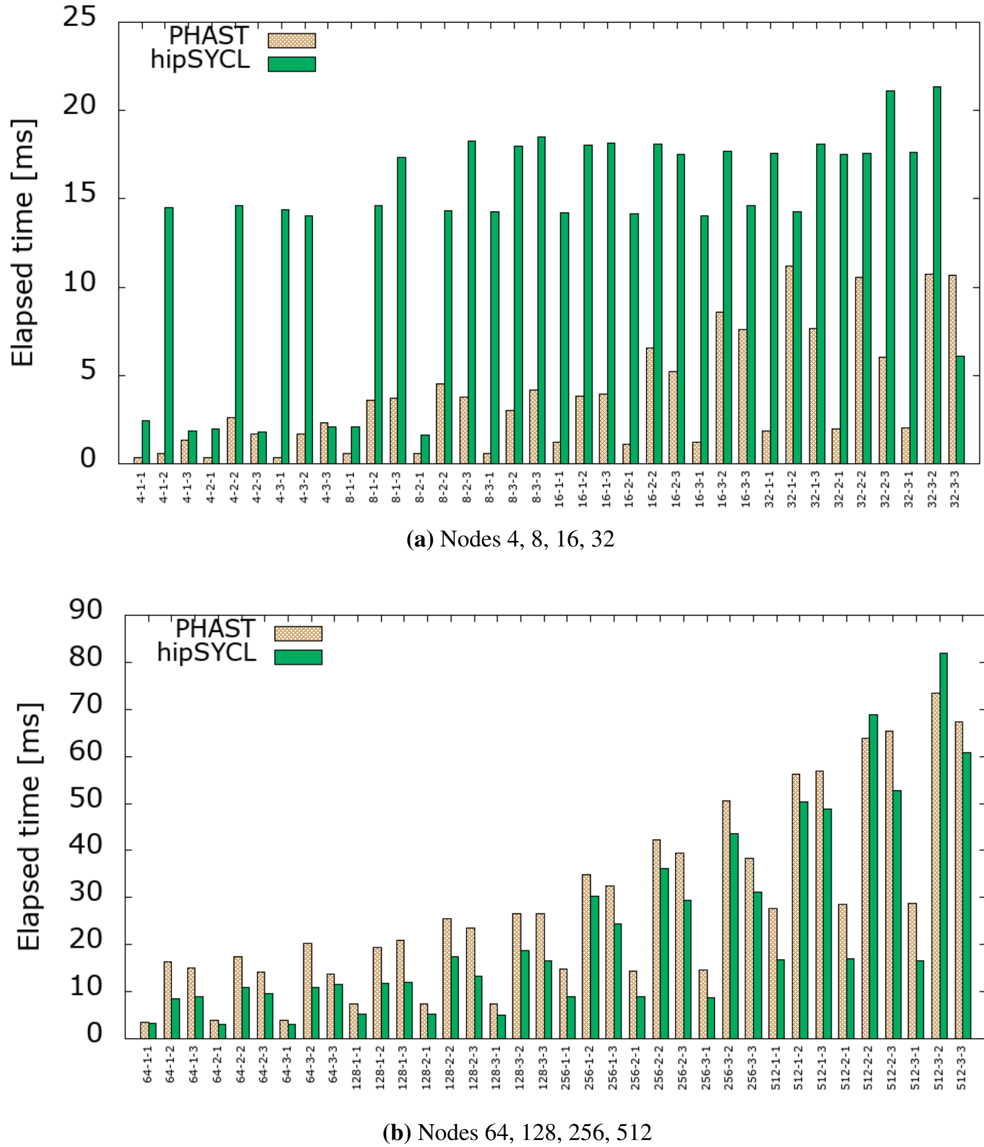
**Table 4.20:** Execution time ratio between SYCL and PHAST versions of the same task-DAGs, aggregated through geometric mean, by number of nodes in the columns and by the size of the matrices managed by each node in the rows. The overall average ratio across all nodes and matrix sizes is indicated as well. The top sub-table shows the ratio in case of a multi-core CPU, while the bottom one shows results for an NVIDIA GPU.

that is evident with bigger task-DAGs. Conversely, when computation dominates, PHAST Library has better execution time due to a more efficient implementation of the data-parallel algorithms invoked into the tasks' bodies. This effect is even more evident with bigger workloads.

Figure 4.19 highlights that, in case of heavier tasks (512x512 matrices), the base overhead observed for SYCL for small task-DAGs is diluted in the higher execution time. Furthermore, there are far more cases where PHAST Library is faster or close to SYCL across all the spectrum of task-DAG sizes. In particular, also Table 4.20 highlights that PHAST is about 2x faster than SYCL for 4 and 8 node DAGs, between 1.15x and 1.54x in the range from 16 up to 64 nodes, and slightly better than SYCL for 64, 128, and 256 nodes, and 4% worse than SYCL for 512. On average, on 512x512 matrices and across all considered random DAGs PHAST is 1.38x faster than SYCL. Essentially, similarly to the results for 32x32 and 128x128, starting from a certain DAG size, the average performance ratio starts converging to a quite stable value.

In conclusion, for multi-core targets, PHAST Library results substantially faster than SYCL where computation dominates on task management. Otherwise, with quick computations and big DAGs, thread management in PHAST dominates execution time and SYCL is faster.

Overall, the geometric average on all considered performance ratios indicates that PHAST is 1.56x faster than SYCL on the excution for multi-core CPUs.




**(a)** Nodes 4, 8, 16, 32

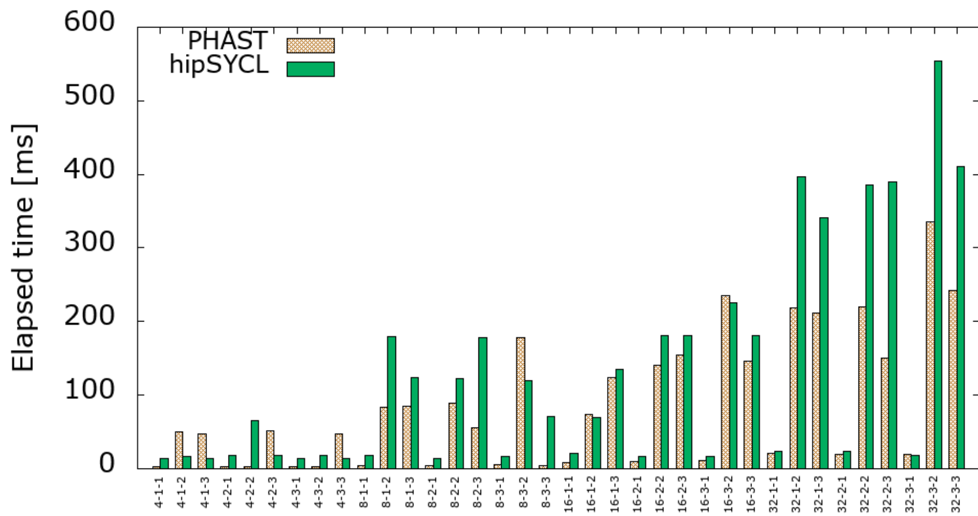
**(b)** Nodes 64, 128, 256, 512

**Figure 4.18:** Comparison of elapsed times of task-DAG benchmarks on multi-core CPU with workload of 128x128 matrices. Labels are N-Fo-Fi: N is the number of nodes, Fo is the maximum fan-out, Fi is the maximum fan-in.

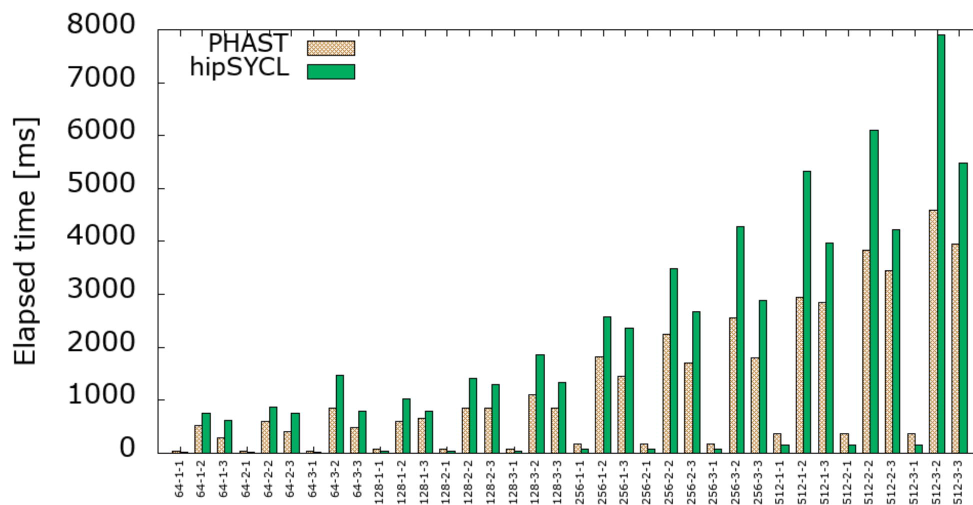
#### Performance analysis in case of NVIDIA GPU target

In this section, the NVIDIA GPU architecture results are discussed for the same benchmarks as before. Specifically, performance of the three benchmark-sets working on small (32x32), medium (128x128), and large (512x512) matrices inside the tasks of the DAGs are shown, respectively, in Figures 4.20, Figures 4.21, and Figures 4.22, for both PHAST Library and SYCL.

Figure 4.20 shows that for 32x32 matrices, and for the majority of smaller task-DAGs PHAST Library is significantly faster than SYCL (about 2.4x, 3.1x and 1.64x for 4, 8




**(a)** Nodes 4, 8, 16, 32

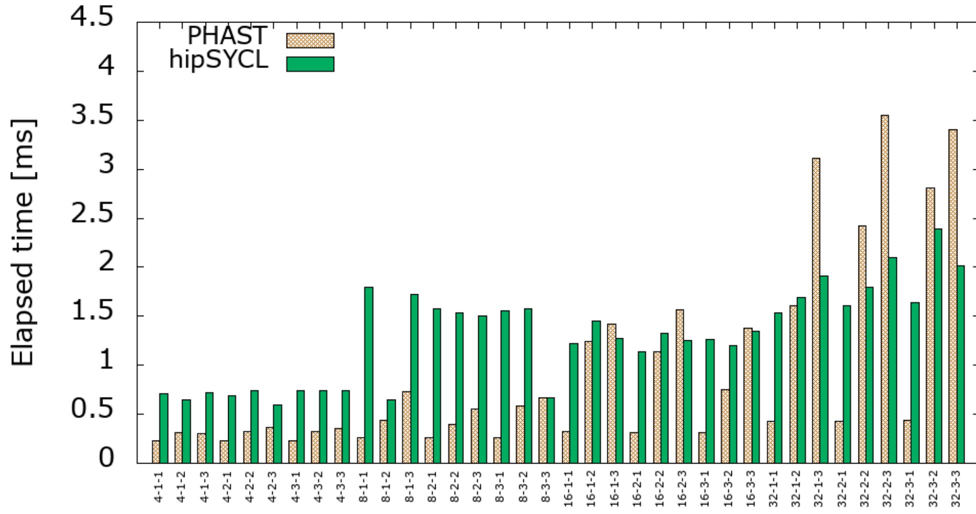



**(b)** Nodes 64, 128, 256, 512

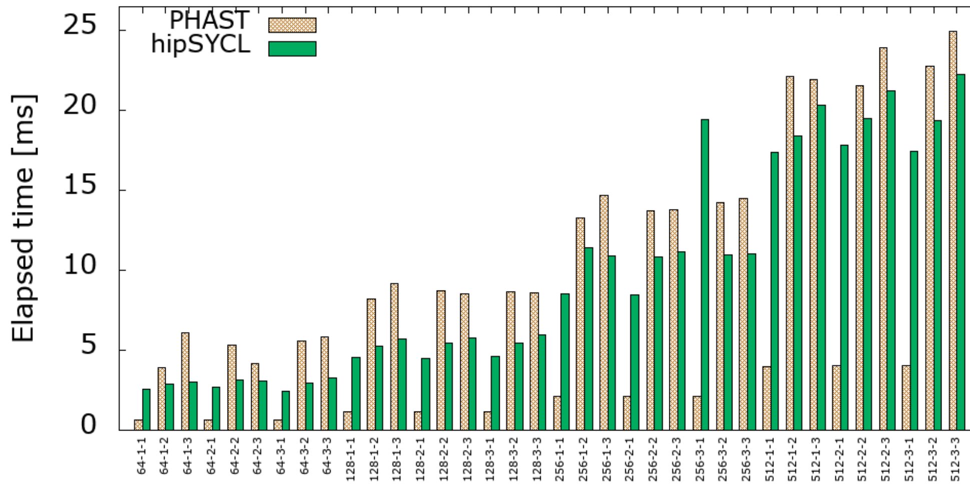
**Figure 4.19:** Comparison of elapsed times of task-DAG benchmarks on multi-core CPU with workload of 512x512 matrices. Labels are N-Fo-Fi: N is the number of nodes, Fo is the maximum fan-out, Fi is the maximum fan-in.

and 16 nodes respectively), then the advantage is slightly smaller for 32 node DAGs (1.24x). For 64 or more nodes, the geometric average over the considered fan-ins and fan-outs, shown in Table 4.20, indicates that PHAST Library keeps on being quite faster than SYCL.

The observed situation is similar to the multi-core case. However, the performance differences between the two approaches are less evident because the data-parallel algorithms are executed on the GPU, and there is no contention with the threads allocated for the task.




**(a)** Nodes 4, 8, 16, 32

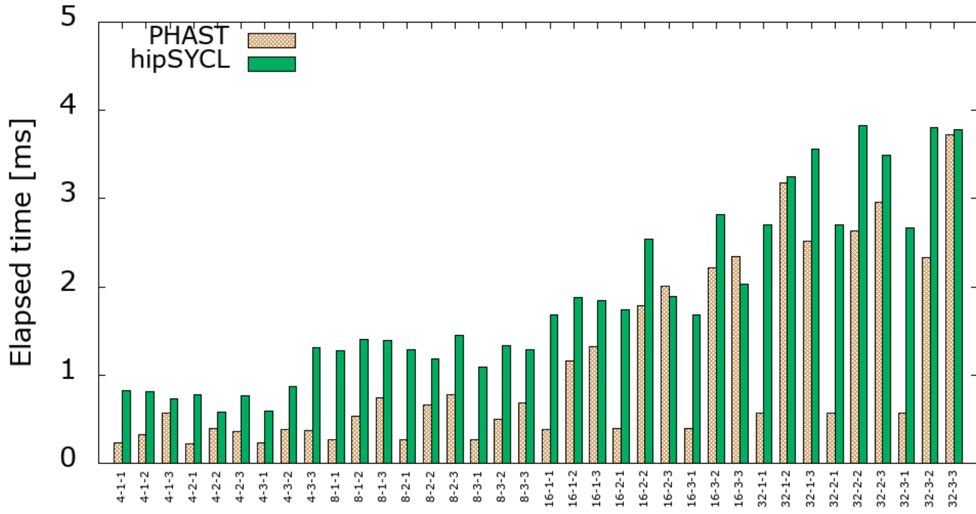



**(b)** Nodes 64, 128, 256, 512

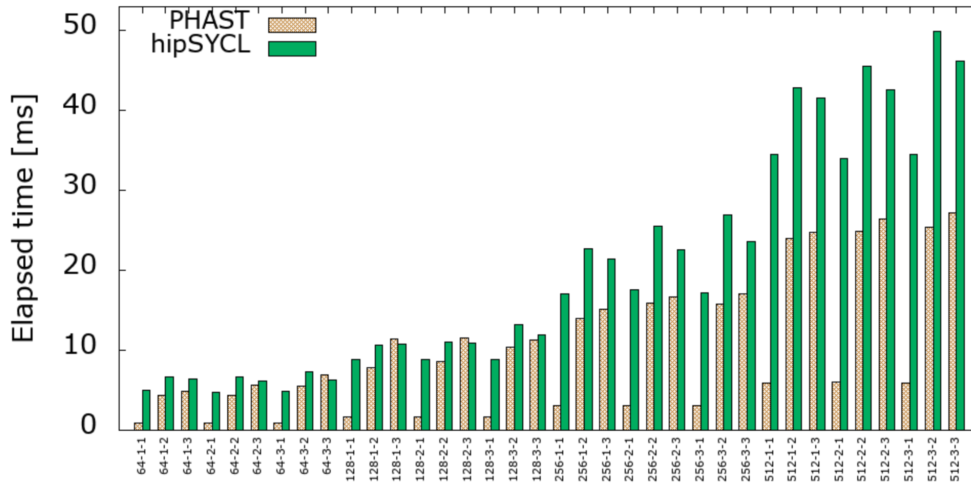
**Figure 4.20:** Comparison of elapsed times of task-DAG benchmarks on NVIDIA GPU with workload of 32x32 matrices. Labels are N-Fo-Fi: N is the number of nodes, Fo is the maximum fan-out, Fi is the maximum fan-in.

In any case, Table 4.20 shows that for every node-count, and overall, in case of small matrices PHAST is faster than SYLC.

Figure 4.21 describes the relative performance of the techniques for 128x128 matrices. PHAST Library appears to be consistently faster across all the considered task-DAGs apart from really a few cases (e.g., 16-3-3 and 128-2-3) where the slowdown is never more than 15%. Conversely, Table 4.20 indicates that the average speedup over SYCL goes from around 2.5x for 4 and 8 nodes, to around 2x up to 128 nodes and then raising again to about 2.5x for bigger task-DAGs. Again, the greatest speedups are




**(a)** Nodes 4, 8, 16, 32




**(b)** Nodes 64, 128, 256, 512

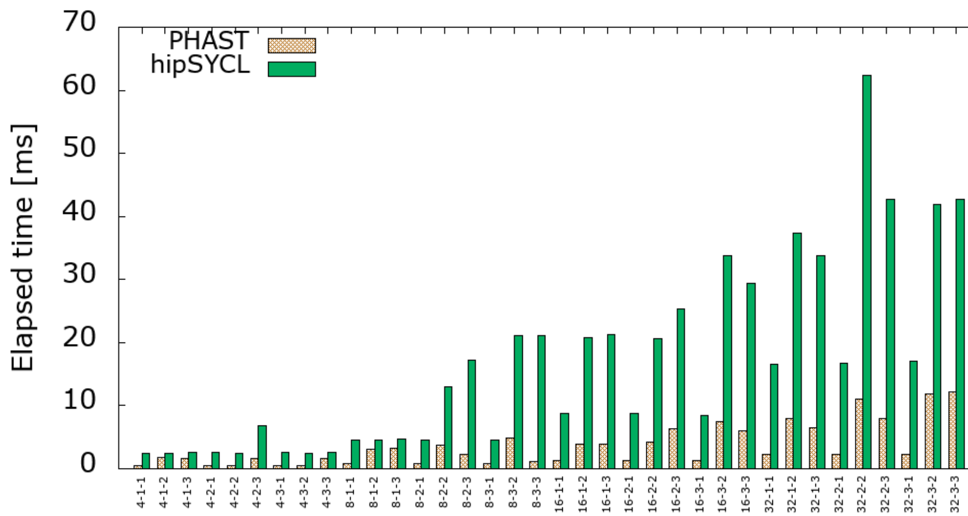
**Figure 4.21:** Comparison of elapsed times of task-DAG benchmarks on NVIDIA GPU with workload of 128x128 matrices. Labels are N-Fo-Fi: N is the number of nodes, Fo is the maximum fan-out, Fi is the maximum fan-in.

observed for smaller fan-ins.

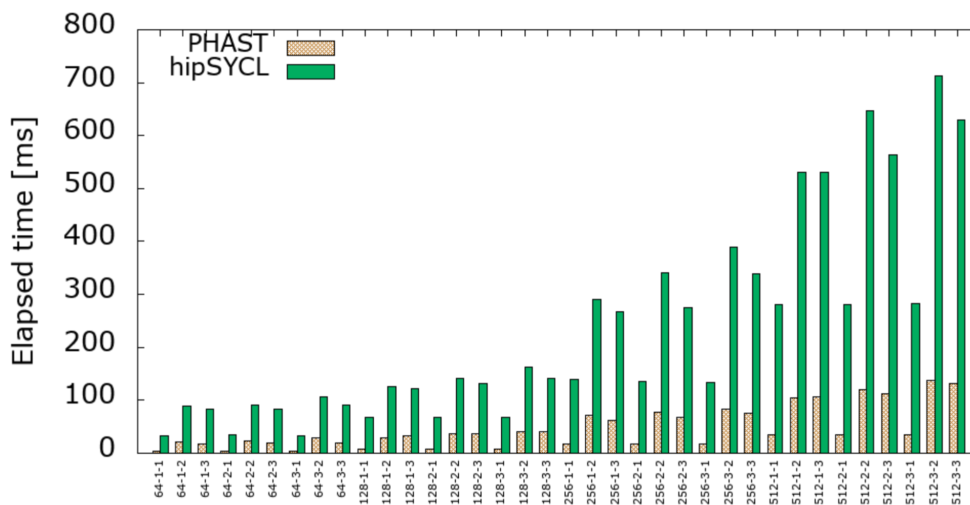
Evidently, a more efficient implementation in the data-parallel algorithms makes PHAST Library faster than SYCL even for a medium workload.

Lastly, Figure 4.22 indicates that PHAST Library is constantly faster than SYCL across the considered benchmark-set for computations on 512x512 matrices in the tasks. Table 4.20 highlights that the speedup more or less increases from about 3.5x up to 5.9x, going from 4 nodes to 512 nodes. Then, the overall average improvement is 4.96x.

This case confirms the faster implementation of data-parallel algorithms that domi-




**(a)** Nodes 4, 8, 16, 32




**(b)** Nodes 64, 128, 256, 512

**Figure 4.22:** Comparison of elapsed times of task-DAG benchmarks on NVIDIA GPU with workload of 512x512 matrices. Labels are N-Fo-Fi: N is the number of nodes, Fo is the maximum fan-out, Fi is the maximum fan-in.

nates execution time for big workloads.

Confronting results for different matrix sizes, i.e., amount of parallel work done in the GPU in each of the nodes, as the load increases PHAST Library improves its advantage, going from having a few configurations where it exhibits some slowdown, up to being always faster than SYCL for bigger matrices. This can be quite interesting from the user-perspective because smaller computations are less likely to get significant advantage from being offloaded to the GPU because of a) the intrinsic offloading overhead, and b) the limited parallelism exploitable for the GPU on small data structures.

Overall, Table 4.20 highlights that averaging also across the various matrix sizes, PHAST results 2.6x faster than SYCL.

**Productivity analysis**

| | SLOC | MEN D | TOT CY |
|---|---|---|---|
| PHAST | 146 | $2.413 \times 10^5$ | 15 |
| PHAST SA | 95 | $2.256 \times 10^5$ | 11 |
| SYCL | 142 | $5.952 \times 10^5$ | 11 |

**(a)** Benchmark 4-1-1

| | SLOC | MEN D | TOT CY |
|---|---|---|---|
| PHAST | 206 | $2.809 \times 10^5$ | 15 |
| PHAST SA | 155 | $2.653 \times 10^5$ | 11 |
| SYCL | 202 | $7.706 \times 10^5$ | 11 |

**(b)** Benchmark 64-1-1

| | SLOC | MEN D | TOT CY |
|---|---|---|---|
| PHAST | 654 | $7.359 \times 10^5$ | 15 |
| PHAST SA | 603 | $7.203 \times 10^5$ | 11 |
| SYCL | 650 | $2.884 \times 10^6$ | 11 |

**(c)** Benchmark 512-1-1

| | SLOC | MEN D | TOT CY |
|---|---|---|---|
| PHAST | 146 | $2.413 \times 10^5$ | 15 |
| PHAST SA | 95 | $2.256 \times 10^5$ | 11 |
| SYCL | 142 | $5.952 \times 10^5$ | 11 |

**(d)** Benchmark 4-3-3

| | SLOC | MEN D | TOT CY |
|---|---|---|---|
| PHAST | 206 | $2.948 \times 10^5$ | 15 |
| PHAST SA | 155 | $2.792 \times 10^5$ | 11 |
| SYCL | 202 | $7.984 \times 10^5$ | 11 |

**(e)** Benchmark 64-3-3

| | SLOC | MEN D | TOT CY |
|---|---|---|---|
| PHAST | 654 | $8.984 \times 10^5$ | 15 |
| PHAST SA | 603 | $8.828 \times 10^5$ | 11 |
| SYCL | 650 | $3.243 \times 10^6$ | 11 |

**(f)** Benchmark 512-3-3

**Table 4.21:** Source Lines Of Code (SLOC), Halstead's Mental Discriminations (MEN D), and McCabe's Total Cyclomatic Complexity (TOT CY) metrics measured on PHAST, PHAST SA (single architecture), and SYCL implementations of various benchmarks. Each benchmark is labeled as N-Fo-Fi, where N is the number of nodes, Fo and Fi are the maximum fan-out and fan-in values, respectively.

Table 4.21 shows the comparison in terms of complexity metrics of PHAST Library and SYCL implementations of six benchmarks. They have been chosen between all the implemented ones as representative of small (4 nodes), medium (64 nodes), and big (512 nodes) task-DAGs with a simple (fan-in and fan-out set to 1) and complex (fan-in and fan-out set to 3) structure.

The metrics have been calculated on benchmark source code deprived of all the instructions that were not strictly necessary to the correct execution, such as blank lines, comments, logging, performance measuring, etc. It has been done to be as fair as possible and also to focus the analysis on the *relevant* code only.

The adopted metrics are: Source Lines of Code (SLOC), for the sheer length of the program, Halstead's Mental Discriminations (MEN D), for the complexity in terms of number of symbols used in the code, and McCabe's Total Cyclomatic Complexity (TOT CY), for the number of paths that the program can take. All of them describe in different manners the complexity of the program, and thus can be regarded as a measure of productivity: the greater the complexity of a program, the more difficult it will be to write it, and therefore, the lower the productivity of a programmer.

In Table 4.21, PHAST Library has been split in two different rows: PHAST and PHAST SA (where SA stands for *single architecture*). In the first case, the code has been written to allow the coexistence of multi-core and GPU code in the same executable, as explained in Sub-section 3.9.3. In the second case, it has been written the classical way where the device of execution is decided at compile-time and all the data-parallel algorithms inside the tasks are executed on the same device. It is important to underline that SYCL code, as previously explained, supports only homogeneous task-DAGs, like PHAST SA and unlike PHAST, due to the nature of its **queue** object that can schedule kernels on a single device only. Therefore, PHAST SA values constitute the fair comparison to SYCL.

The values in the Table show that PHAST code is slightly more complex than SYCL code in terms of SLOC and TOT CY, both 4 more than SYCL in all the benchmarks. Conversely, in terms of MEN D, PHAST code is less complex than SYCL code: from 59.46% up to 74.83% less complex with respect to SYCL. This last metric takes into account the fact that even simple SYCL applications need the allocation of a queue, arrays or vectors, buffers, accessors, command groups, and kernels expressed as functors or lambdas. PHAST, conversely, even with explicit task management, generally needs fewer symbols (a container and an algorithm invocation at its minimum).

PHAST SA's SLOC and TOT CY values highlight that part of the complexity (51 SLOC and 4 TOT CY) is due to the management of the coexistence of different architectures, being it the only difference between the two implementations. PHAST SA, as specified before, is more similar to SYCL than PHAST in the functionalities it

offers. Thus, comparing these two implementations is the fair approach. In this regard, we can see that PHAST SA is up to 33.10% shorter than the latter and up to 75.02% less complex in terms of MEN D. From a TOT CY point of view, the two implementations have the same complexity across all the benchmarks. In short, we can say that PHAST SA code is less complex than SYCL code.

Overall, the task-DAG formalism adopted in PHAST Library proves to be a valid choice to code task-parallel heterogeneous applications from a productivity perspective, even compared to a high-level productivity-oriented solution.

### 4.3.4 Summary

PHAST Library capabilities to write task-parallel code have been evaluated in the case of a family of randomly-generated task-based applications with different sizes and complexities. The comparison of the PHAST Library implementation against a SYCL one from both performance and productivity standpoints. On the performance side, PHAST Library scores 1.56x and 2.60x speedups on average on multi-core CPUs and NVIDIA GPUs, respectively. On the former, some work can be done to improve the thread spawning mechanism.

PHAST Library's productivity is significantly higher both in terms of SLOC and MEN D metrics, when considering uniform (CPU or GPU) task-DAGs, the only ones supported by the reference SYCL implementation. Then, PHAST Library version supporting heterogeneous tasks in the same DAG is only slightly more complex than the SYCL version supporting only one architecture.

The study also highlights the performance advancements that could result from the integration of a more sophisticated runtime. In particular, a thread-pool mechanism would be able to eliminate the thread spawning cost and consequently deliver better performance. The current implementation relies on **`std::async`**, but it could be improved this way. Also, the inclusion of a runtime would give the opportunity to tackle other aspects that have not been discussed so far, like the centralized management of memory allocation and deallocation that would limit, if not *eliminate*, the possible memory oversubscription that can happen in this concurrent context. Moreover, its thread-pool could be used to manage also multi-core side data-parallel algorithms, limiting the interference between OS threads that serve different purposes.

CHAPTER 5

# Conclusions and future work

Due to the difficulties found in continuing to improve the single-core microprocessor design driven by Moore's law and Dennard scaling, hardware manufacturers decided to switch to a multi-core layout that integrates various cores on a single chip and let them communicate through shared memory mechanisms. On a different technological path, GPUs improved to the point that they became fully fledged general purpose co-processors with specialized memories and thousands of simple cores that are able to deliver unprecedented performance in data-parallel calculations. This two architectures led the parallel revolution that, in recent years, allowed parallel architectures to penetrate in every field of computing, from wearable mobile devices to giant supercomputers.

Multi-core CPUs and GPUs can often be found in heterogeneous systems, giving to the programmers the possibility to solve their problems with the help of the best fitting architecture for the problem at hand, using the capabilities and peculiarities of devices with different strengths and weaknesses in terms of latency, bandwidth, power consumption, etc.

Being able to code for different parallel devices is a skill that, at the present state, is hard to master: every device can have its programming language, set of libraries, frameworks, programming patterns, and architectural details, which are often fundamental to optimize and tune an application. While some frameworks focus mainly on the sheer performance, others aim at combining both performance and programmer's productivity,

generally intended as the possibility to write code with a reduced complexity. However, the *definitive* solution is yet to come, as these frameworks still demand their programmers to lower the level of abstraction with respect to classic, well-known sequential techniques and to reason about parallel execution, architectural peculiarities, and other details that make the coding process difficult.

During these Ph.D. years, my research focused on studying parallel heterogeneous systems and their programming techniques with the main purpose of improving their programmability and bridging the gap that still exists between sequential and parallel heterogeneous programming. The way to help programmers in the difficult task of writing parallel heterogeneous code soon took the form of a single-source programming framework whose name is PHAST Library.

PHAST Library is built around the principles of code portability, near-native performance, performance portability, and productivity. The first one is achieved by wrapping two native approaches for multi-core CPUs and NVIDIA GPUs, that are used as backends. The other three are the product of many design choices that intersect in many ways and together constitute the structure of the library. Broadly, it permits expressing high-level calculations on multi-dimensional containers with a formalism that resembles and extends that defined and made famous by the STL [149, 152]. In PHAST Library, containers can also be iterated in sections, not only in scalar elements: these are slices, partitions with a well-defined shape. These sections are also responsible for the coarser grain of parallelism, achieved when they are processed in ranges inside tens of predefined algorithms that take inspiration from the STL itself and linear algebra. Some of them allow programmers to specify functors, that constitute the *degree of freedom* where general computations operating on sections can be expressed. Inside functors, the adopted formalism permits writing code with a high-level of abstraction that only considers the shape and data-type of the sections, represented as *in-functor* containers, and abstracts away many details concerning the source container, the OS or CUDA thread indexing, the architectural details of the device of execution, and so on. The calculations that can be applied on these in-functor containers can be synthetically expressed by the means of device-side algorithms that exploit their high-level nature and, in particular, their possibility to be iterated section-wise as the main containers. Both the main and the device-side in-functor algorithms have a flexible implementation that offers some *knobs* that can be regulated from outside with the use of the parallelization parameters. This way, algorithmic semantics, parallel execution, and tuning are decoupled and even physically separated in different portions of the code.

The library is mainly data-parallel in its design and purpose. However, it also gives the possibility to write task-parallel code. This can be used as a standalone feature to have task-parallelism on the multi-core only, or to wrap data-parallel computations that

can be offloaded also on NVIDIA GPUs. The chosen formalism allows to easily express task-DAGs regulated by data-flow dependencies.

PHAST Library as a whole is my main contribution to the field of heterogeneous programming, as it can help to write efficient, high-level code that can run on NVIDIA GPUs and multi-core CPUs. However, its device-side (i.e., in-functor) programming abstractions constitute the main novelty from a research point of view. No other framework permit expressing various and diverse high-level calculations on multi-dimensional data-structures, abstracting away the nature of the source containers, the parallel resources that execute the code as well as their identity and layout, the architectural details of the underlying hardware, and all of this with high performance and the possibility to tune and optimize the execution from outside with runtime parameters. These features push in the direction of productivity, for the expressiveness of the formalism adopted, and performance portability, for the separation between application code and fine-tuning that allows programmers to avoid rewriting and re-implementing the same application for different devices.

The value and maturity of PHAST Library is testified by the results of the thorough evaluation conducted and presented in this work. Moreover, it is now being used at University of Murcia by the computer architecture group to port the Caffe deep learning framework to a single-source version. The development is still in an early phase, but it will be continued in the next months.

From a productivity standpoint, the results here discussed highlight that the design choices done in terms of formalism adopted pay because they lead to code that is consistently less complex than that obtained with concurrent state-of-the-art approaches. So, they testify that PHAST Library can deliver a measurable improvement in terms of programmability.

From a performance standpoint, the evaluations centered on data-parallel applications highlight that PHAST Library is usually able to achieve near-native performance on NVIDIA GPUs. On the multi-core side, however, there is sometimes a non-negligible performance gap with respect to frameworks that use sophisticated horizontal vectorization techniques [63], like OpenCL by Intel vendor [81] and Codeplay's implementation of SYCL [21]. These permit translating operations on primitive types from neighbour threads of execution into SIMD operations on vector registers, and thus achieve a speedup for those operations that can amount to the number of SIMD lanes. In order to bridge this performance gap, PHAST Library will need to support similar techniques, that also involve AoS to SoA transformations, by ad-hoc implementations inside its layers or by adding the support for these approaches as further backends.

On this regard, a possible way to make PHAST Library evolve could be by wrapping the SYCL [84] programming model as an additional backend. This would solve the

performance problem above and also add the support of a variety of devices. Wrapping SYCL would be *strategic* also because there is a growing attention on that programming model and the recent debut of Intel OneAPI [64] will likely attract many application and library developers in the near future.

Another aspect that can be improved has been highlighted by the evaluation of the task-parallel capabilities of the library. The results for random task-DAGs with fan-in greater than one highlighted that the absence of a runtime can be penalizing from a performance point of view on multi-core CPUs. The current solution that spawns threads when needed adds some overhead that can be avoided or mitigated with the help of a thread-pool. Apart from supporting a thread-pool, a runtime could also take care of various aspects that would constitute an improvement: it could manage the execution of the data-parallel algorithms on the multi-core side, or it could manage the memory allocation in task-DAGs in a centralized way, in order to avoid problems related to oversubscription.

A trial version of PHAST Library is currently downloadable from the website [134]. There, a development roadmap is published with the purpose of informing the community about the ongoing and future developments. Alongside the development directions mentioned above, there are others that can be interesting to follow. Some concern the design and structure of the library, while others are relative to its domain of belonging and the classes of applications it can model.

On the design side, there are three possible development directions. All of them would help the library to evolve on the productivity path already traced.

The first concerns the enhancement of the existing heuristics that make an estimation of the parallelization parameters, in order to reduce the necessity of explicitly specifying them. That possibility would not be removed from the library, so to keep providing to the programmers full control on the algorithm execution and the optimization criteria, but the estimation process should evolve to make it as less necessary as possible. The research could focus on estimation techniques that involve neural networks, tentative runs, or code reflection mechanisms.

The second design direction is the definition of generic multi-dimensional containers that would enrich the current set of mono-, bi-, and three-dimensional ones. As these can be used to express some problems in a natural way, giving the possibility to match the application-data layout with the container layout, they present obvious limitations to model data that naturally map on structures of 4 or more dimensions. In this case, the research effort would concentrate on the way to express, both syntactically and semantically, a generic multi-dimensional container with many types of iterators that point to multi-dimensional sections of various shapes. Keeping the design choices done so far, in fact, a generic N-dimensional container would need to give access to N+1

different types of iterators.

The last design direction is the addition of the support of lambda functions. This would need to abandon – at least in that case – the inheritance from base-functors that adds to user-defined functors the possibility to invoke various finely-tuned implementations of in-functor algorithms inside their body. On a first approximation, they could lack the support for in-functor algorithms and thus being attractive only in simple examples that would not benefit from those. In the perspective of making them a drop-in replacement of any functor, however, some research is needed to find a more sound solution.

On the library domain side, apart from the addition of a SYCL backend discussed above, two directions can be taken. Both of them would expand the possibilities of PHAST Library, bringing it to new fields.

The first is the addition of the support for multi-GPU programming on single node. In the CUDA programming model, in fact, there is the possibility to select the device of execution between those installed in the system [119]. If no explicit choice is done, the device seen as *device 0* is picked [1]. So, the possibility to support more than a single GPU would pose several challenges as choosing the device to use for a particular operation (data allocation, algorithm execution, etc.) and managing communication between them.

The second possibility is to add the support for heterogeneous clusters to PHAST Library and make it a framework suitable for HPC problems. The addition of the cluster dimension to PHAST Library poses many challenges that need research and important design choices to be addressed: how to manage communication, load balancing, scheduling, work partition, fault tolerance, and so on. It is a really challenging direction, but it would significantly increase the possibility of the library and also bring its contribution in terms of productivity, tunability, and performance to a field that commonly addresses some of the most complex and urgent problems of industry, economy, health and, in one word, humanity itself.

Summarizing, this Ph.D. work presents PHAST Library, a mature framework that helps writing productive, high-performance parallel heterogeneous code and constitutes a measurable improvement of the state-of-the-art in the direction of the productivity. It provides also high performance capabilities, even if there is room for improvement, mainly on multi-core CPUs. Starting from here, there are countless possibilities of evolution that will be explored as future work and continue improving both the framework and the state-of-the-art of parallel heterogeneous programming.

# Bibliography

[1] Professional CUDA C Programming, 1st edn. Wrox Press Ltd., GBR (2014)

[2] Aksel Alpay: hipSYCL - an implementation of SYCL over NVIDIA CUDA/AMD HIP (2019). URL https://github.com/illuhad/hipSYCL

[3] Altazin, T., Ersoy, M., Golay, F., Sous, D., Yushchenko, L.: Numerical Investigation of BB-AMR Scheme Using Entropy Production as Refinement Criterion. Int. J. Comput. Fluid Dyn. **30**(3), 256–271 (2016). doi: 10.1080/10618562.2016.1194977

[4] AMD: Hip : C++ heterogeneous-compute interface for portability (2016). URL https://gpuopen.com/compute-product/hip-convert-cuda-to-portable-c-code/

[5] AMD: ROCm, a New Era in Open GPU Computing (2016). URL https://rocm.github.io/index.html

[6] AMD: RDNA Architecture (2019). URL https://www.amd.com/system/files/documents/rdna-whitepaper.pdf

[7] Amdahl, G.M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California. IEEE Solid-State Circuits Society Newsletter **12**(3), 19–20 (2007). doi: 10.1109/N-SSC.2007.4785615

[8] ARM: Arm Architecture Reference Manual (2019). URL https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf

[9] ARM: ARM big.LITTLE (2020). URL https://www.arm.com/why-arm/technologies/big-little

[10] Augonnet, C., Aumage, O., Furmento, N., Namyst, R., Thibault, S.: Starpu-mpi: Task programming over clusters of machines enhanced with accelerators. In: J.L. Träff, S. Benkner, J.J. Dongarra (eds.) Recent Advances in the Message Passing Interface, pp. 298–299. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[11] Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurr. Comput. : Pract. Exper. **23**(2), 187–198 (2011)

[12] Bandyopadhyay, A.: Hands-On GPU Computing with Python. Packt Publishing (2019)

[13] Bernstein, D.: QHASM: tools to help write high-speed software (2017). URL `http://cr.yp.to/qhasm.html`

[14] Bohr, M.: A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. IEEE Solid-State Circuits Society Newsletter **12**(1), 11–13 (2007). doi: 10.1109/N-SSC.2007.4785534

[15] Boyar, J., Peralta, R.: A New Combinational Logic Minimization Technique with Applications to Cryptology, pp. 178–189. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

[16] Bueno, J., Martinell, L., Duran, A., Farreras, M., Martorell, X., Badia, R.M., Ayguade, E., Labarta, J.: Productive Cluster Programming with OmpSs. In: Proceedings of the 17th International Conference on Parallel Processing - Volume Part I, Euro-Par'11, pp. 555–566. Springer-Verlag, Berlin, Heidelberg (2011)

[17] Burns, R.: Adding Experimental PTX Support To ComputeCpp For NVIDIA Hardware. URL `https://www.codeplay.com/portal/12-06-17-adding-experimental-ptx-support-to-computecpp-for-nvidia`

[18] Canright, D.: A Very Compact S-box for AES. In: Proceedings of the 7th International Conference on Cryptographic Hardware and Embedded Systems, CHES '05, pp. 441–455. Springer-Verlag, Berlin, Heidelberg (2005)

[19] Carter Edwards, H., Trott, C.R., Sunderland, D.: Kokkos. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). doi: 10.1016/j.jpdc.2014.07.003

[20] Castillo, A.D., Cortazar, D.I., Poo-Caamaño, G., Gonzalez-Barahona, J.M., Cañas-Díaz, L., Dueñas, S.: CMetrics. URL `https://github.com/MetricsGrimoire/CMetrics`

[21] Codeplay: ComputeCpp - Accelerate Complex C++ Applications on Heterogeneous Compute Systems using Open Standards. URL `https://www.codeplay.com/products/computesuite/computecpp`

[22] Codeplay: Targeting NVIDIA PTX. URL `https://developer.codeplay.com/products/computecpp/ce/guides/platform-support/targeting-nvidia-ptx`

[23] Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2012)

[24] Cordeiro, D., Mounié, G., Perarnau, S., Trystram, D., Vincent, J.M., Wagner, F.: Random Graph Generation for Scheduling Simulations. In: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, SIMUTools '10, pp. 60:1–60:10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium (2010). URL `http://dl.acm.org/citation.cfm?id=1808143.1808219`

[25] Daga, M., Aji, A.M., Feng, W.c.: On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In: Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing, SAAHPC ’11, p. 141–149. IEEE Computer Society, USA (2011). doi: 10.1109/SAAHPC.2011.29

[26] Daga, M., Tschirhart, Z.S., Freitag, C.: Exploring parallel programming models for heterogeneous computing systems. In: 2015 IEEE International Symposium on Workload Characterization, pp. 98–107 (2015). doi: 10.1109/IISWC.2015.16

[27] Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering **5**(1), 46–55 (1998)

[28] Deane, A., Brenner, G., Ecer, A., Emerson, D.R., McDonough, J., Periaux, J., Satofuka, N., Tromeur-Dervout, D.: Parallel Computational Fluid Dynamics 2005: Theory and Applications. Elsevier Science Inc., USA (2006)

[29] DeBenedictis, E.P.: It’s Time to Redefine Moore’s Law Again. Computer **50**(2), 72–75 (2017). doi: 10.1109/MC.2017.34

[30] Demirel, H., Anbarjafari, G.: Image resolution enhancement by using discrete and stationary wavelet decomposition. IEEE Transactions on Image Processing **20**(5), 1458–1460 (2011)

[31] Denis Demidov: VexCL (2018). URL `https://github.com/ddemidov/vexcl`

[32] Dennard, R., Gaensslen, F., Rideout, V., Bassous, E., LeBlanc, A.: Design of ion-implanted mosfet’s with very small physical dimensions. Solid-State Circuits, IEEE Journal of **9**(5), 256–268 (1974). doi: 10.1023/A:1008373903657. URL `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=1050511&isnumber=22538`

[33] Dick, Robert P. and Rhodes, David L. and Wolf, Wayne: TGFF: Task Graphs for Free. In: Proceedings of the 6th International Workshop on Hardware/Software Codesign, CODES/CASHE ’98, pp. 97–101. IEEE Computer Society, USA (1998)

[34] Dongarra, J.J., Duff, L.S., Sorensen, D.C., Vorst, H.A.V.: Numerical Linear Algebra for High Performance Computers. Society for Industrial and Applied Mathematics, USA (1998)

[35] Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-Platform GPU Programming. Parallel Comput. **38**(8), 391–407 (2012). doi: 10.1016/j.parco.2011.10.002

[36] Dumas, J.G., Pernet, C., Sultan, Z.: Computing the Rank Profile Matrix. In: Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC ’15, p. 149–156. Association for Computing Machinery, New York, NY, USA (2015). doi: 10.1145/2755996.2756682

[37] Edwards, H.C., Trott, C.R.: Kokkos: Enabling Performance Portability Across Manycore Architectures. In: 2013 Extreme Scaling Workshop (xsw 2013), pp. 18–24 (2013)

[38] Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202 – 3216 (2014). doi: https://doi.org/10.1016/j.jpdc.2014.07.003. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing

[39] Eichenberger, A.E., Wu, P., O'Brien, K.: Vectorization for SIMD Architectures with Alignment Constraints. SIGPLAN Not. **39**(6), 82–93 (2004). doi: 10.1145/996893.996853

[40] en.cppreference.com: C++ named requirements- Callable (2018). URL `en.cppreference.com/w/cpp/named_req/Callable`

[41] Enmyren, J., Kessler, C.W.: SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In: Proc. of the Int. Workshop on High-level Par. Progr. and Applications, HLPP '10, pp. 5–14. ACM, New York, NY, USA (2010)

[42] Fang, J., Varbanescu, A.L., Sips, H.: A Comprehensive Performance Comparison of CUDA and OpenCL. In: Proc. Int. Conference on Parallel Processing, pp. 216–225 (2011)

[43] Flynn, M.J.: Some computer organizations and their effectiveness. IEEE Transactions on Computers **C-21**(9), 948–960 (1972). doi: 10.1109/TC.1972.5009071

[44] Foglia, P., Panicucci, F., Prete, C.A., Solinas, M.: Analysis of performance dependencies in nuca-based cmp systems. In: 2009 21st International Symposium on Computer Architecture and High Performance Computing, pp. 49–56 (2009). doi: 10.1109/SBAC-PAD.2009.12

[45] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., USA (1995)

[46] Ganesan, S., John, V., Matthies, G., Meesala, R., Shamim, A., Wilbrandt, U.: An object oriented parallel finite element scheme for computations of PDEs: Design and implementation. In: 2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW), pp. 106–115. IEEE (2016)

[47] Grasso, I., Pellegrini, S., Cosenza, B., Fahringer, T.: LibWater: Heterogeneous Distributed Computing Made Easy. In: Proc. of the Int. Conference on Supercomputing, ICS '13, pp. 161–172. ACM, New York, NY, USA (2013)

[48] Greengard, S.: Gpus reshape computing. Commun. ACM **59**(9), 14–16 (2016). doi: 10.1145/2967979

[49] Gregory, K., Miller, A.: C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++. O'Reilly, Sebastopol, CA, USA (2012)

[50] Grewe, D., Wang, Z., O'Boyle, M.F.P.: Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 1–10 (2013)

[51] Haensch, W., Nowak, E.J., Dennard, R.H., Solomon, P.M., Bryant, A., Dokumaci, O.H., Kumar, A., Wang, X., Johnson, J.B., Fischetti, M.V.: Silicon CMOS devices beyond scaling. IBM Journal of Research and Development **50**(4.5), 339–361 (2006). doi: 10.1147/rd.504.0339

[52] Halstead, M.H.: Elements of Software Science (Operating and Programming Systems Ser.). Elsevier Science Inc., New York, USA (1977)

[53] Hammond, J.R., Kinsner, M., Brodman, J.: A Comparative Analysis of Kokkos and SYCL as Heterogeneous, Parallel Programming Models for C++ Applications. In: Proceedings of the International Workshop on OpenCL, IWOCL'19. Association for Computing Machinery, New York, NY, USA (2019). doi: 10.1145/3318170.3318193

[54] Hasan, K.M.A., Islam, K., Islam, M., Tsuji, T.: An extendible data structure for handling large multidimensional data sets. In: 2009 12th International Conference on Computers and Information Technology, pp. 669–674 (2009)

[55] Hellekalek, P., Wegenkittl, S.: Empirical Evidence Concerning AES. ACM Trans. Model. Comput. Simul. **13**(4), 322–333 (2003)

[56] Hennessy, J.L., Patterson, D.A.: Computer Architecture, Sixth Edition: A Quantitative Approach, 6th edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2017)

[57] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)

[58] Hertz, M., Berger, E.D.: Quantifying the performance of garbage collection vs. explicit memory management. SIGPLAN Not. **40**(10), 313–326 (2005). doi: 10.1145/1103845.1094836

[59] Hord, R.M.: Parallel Supercomputing in SIMD Architectures. CRC Press, Inc., USA (1990)

[60] Intel: The First Nehalem Processor. URL `http://download.intel.com/pressroom/kits/corei7/images/Nehalem_Die_callout.jpg`

[61] Intel: Intel Core 2 Duo Processor E6700 (2006). URL `https://ark.intel.com/content/www/us/en/ark/products/27251/intel-core-2-duo-processor-e6700-4m-cache-2-66-ghz-1066-mhz-fsb.html`

[62] Intel: Intel® Pentium® Processor Extreme Edition 965 (2006). URL `ark.intel.com/content/www/us/en/ark/products/27615/intel-pentium-processor-extreme-edition-965-4m-cache-3-73-ghz-1066-mhz-fsb.html`

[63] Intel: Autovectorization in Intel OpenCL SDK 1.5 (2011). URL `https://software.intel.com/en-us/blogs/2011/09/26/autovectorization-in-intel-opencl-sdk-15`

[64] Intel: oneAPI Specification (2020). URL `https://spec.oneapi.com/versions/latest/oneAPI-spec.pdf`

[65] ISO: ISO/IEC 9899:1999 – Information technology – Programming languages – C. Standard, International Organization for Standardization, Geneva, CH (1999)

[66] ISO: ISO/IEC 14882:2011 – Information technology – Programming languages – C++. Standard, International Organization for Standardization, Geneva, CH (2011)

[67] ISO: ISO/IEC 9899:2011 – Information technology – Programming languages – C. Standard, International Organization for Standardization, Geneva, CH (2011)

[68] ISO: ISO/IEC 14882:2014 – Information technology – Programming languages – C++. Standard, International Organization for Standardization, Geneva, CH (2014)

[69] ISO: ISO/IEC 14882:2017 – Information technology – Programming languages – C++. Standard, International Organization for Standardization, Geneva, CH (2017)

[70] ISO: ISO/IEC 9899:2018 – Information technology – Programming languages – C. Standard, International Organization for Standardization, Geneva, CH (2018)

[71] Iyer, R., Tullsen, D.: Heterogeneous Computing [Guest editors' introduction]. IEEE Micro **35**(4), 4–5 (2015). doi: 10.1109/MM.2015.82

[72] Jagannathan, A., Honghua Yang, H., Konigsfeld, K., Milliron, D., Mohan, M., Romesis, M., Reinman, G., Cong, J.: Microarchitecture evaluation with floorplanning and interconnect pipelining. In: Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005., vol. 1, pp. I/8–I15 Vol. 1 (2005). doi: 10.1109/ASPDAC.2005.1466114

[73] Kaeli, D.R., Mistry, P., Schaa, D., Zhang, D.P.: Heterogeneous Computing with OpenCL 2.0, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2015)

[74] Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: HPX: A Task Based Programming Model in a Global Address Space. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS '14. Association for Computing Machinery, New York, NY, USA (2014). doi: 10.1145/2676870.2676883

[75] Kale, L.V., Bhatele, A.: Parallel Science and Engineering Applications: The Charm++ Approach, 1st edn. CRC Press, Inc., Boca Raton, FL, USA (2013)

[76] Karl Rupp: 42 Years of Microprocessor Trend Data (2018). URL `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`

[77] Karrenberg, R.: Automatic SIMD Vectorization of SSA-Based Control Flow Graphs. Springer Vieweg (2015)

[78] Käsper, E., Schwabe, P.: Faster and Timing-Attack Resistant AES-GCM. In: Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems, CHES '09, pp. 1–17. Springer-Verlag, Berlin, Heidelberg (2009)

[79] Khokhar, A.A., Prasanna, V.K., Shaaban, M.E., Wang, C.L.: Heterogeneous computing: Challenges and opportunities. Computer **26**(6), 18–27 (1993). doi: 10.1109/2.214439

[80] Khronos OpenCL Working Group: SyCL Parallel STL. URL `https://github.com/KhronosGroup/SyclParallelSTL`

[81] Khronos OpenCL Working Group: The OpenCL Specification, version 1.2 (2012). URL `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`

[82] Khronos OpenCL Working Group: The OpenCL Specification, version 2.0 (2015). URL `https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf`

[83] Khronos OpenCL Working Group: The OpenCL Specification, version 2.1 (2018). URL `https://www.khronos.org/registry/OpenCL/specs/opencl-2.1.pdf`

[84] Khronos OpenCL Working Group: SYCL Provisional Specification, version 1.2.1 (2019). URL `https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf`

[85] Khronos OpenCL Working Group: The OpenCL Specification, version 2.2 (2019). URL `www.khronos.org/registry/cl/specs/opencl-2.2.pdf`

[86] Kim, C., Burger, D., Keckler, S.W.: An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. SIGARCH Comput. Archit. News **30**(5), 211–222 (2002). doi: 10.1145/635506.605420

[87] Knuth, D.E.: The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)

[88] Kofler, K., Cosenza, B., Fahringer, T.: Automatic Data Layout Optimizations for GPUs, pp. 263–274. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)

[89] Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., Kobayashi, H.: Evaluating Performance and Portability of OpenCL Programs. In: The Fifth International Workshop on Automatic Performance Tuning (2010)

[90] Kretz, M., Lindenstruth, V.: Vc: A C++ Library for Explicit Vectorization. Softw. Pract. Exper. **42**(11), 1409–1430 (2012). doi: 10.1002/spe.1149

[91] Kumar, A.: G-tree: a new data structure for organizing multidimensional data. IEEE Transactions on Knowledge and Data Engineering **6**(2), 341–347 (1994)

[92] Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, p. 197–201. Association for Computing Machinery, New York, NY, USA (2019). doi: 10.1145/3335772.3335935

[93] Landaverde, R., Tiansheng Zhang, Coskun, A.K., Herbordt, M.: An investigation of unified memory access performance in cuda. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6 (2014)

[94] Lee, J.B., Kalva, H.: The VC-1 and H.264 Video Compression Standards for Broadband Video Services, 1 edn. Springer Publishing Company, Incorporated (2008)

[95] Lee, J.H., Nigania, N., Kim, H., Patel, K., Kim, H.: OpenCL Performance Evaluation on Modern Multicore CPUs. Sci. Program. **2015** (2016). doi: 10.1155/2015/859491

[96] Liao, C., Quinlan, D.J., Willcock, J.J., Panas, T.: Extending Automatic Parallelization to Optimize High-Level Abstractions for Multicore. In: Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism, IWOMP '09, p. 28–41. Springer-Verlag, Berlin, Heidelberg (2009). doi: 10.1007/978-3-642-02303-3_3

[97] Lim, R.K., Petzold, L.R., Koç, Ç.K.: Bitsliced High-Performance AES-ECB on GPUs. In: A.P.Y. Ryan, D. Naccache, J.J. Quisquater (eds.) The New Codebreakers: Essays Dedicated to David Kahn on the Occasion of His 85th Birthday, pp. 125–133. Springer, Berlin, Heidelberg (2016)

[98] Marras, S., Kelly, J.F., Moragues, M., Müller, A., Kopera, M.A., Vázquez, M., Giraldo, F.X., Houzeaux, G., Jorba, O.: A Review of Element-Based Galerkin Methods for Numerical Weather Prediction: Finite Elements, Spectral Elements, and Discontinuous Galerkin. Archives of Computational Methods in Engineering **23**(4), 673–722 (2016). doi: 10.1007/s11831-015-9152-1

[99] McCabe, T.J.: A Complexity Measure. IEEE Trans. Softw. Eng. **2**(4), 308–320 (1976)

[100] McCalpin, J.D.: Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Comput. Soc. Tec. Comm. Newsl. Comput. Archit. (TCCA) (59), 19–25 (1995)

[101] Meier, R., Gross, T.R.: Reflections on the Compatibility, Performance, and Scalability of Parallel Python. In: Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2019, p. 91–103. Association for Computing Machinery, New York, NY, USA (2019). doi: 10.1145/3359619.3359747

[102] Memeti, S., Li, L., Pllana, S., Kołodziej, J., Kessler, C.: Benchmarking opencl, openacc, openmp, and cuda: Programming productivity, performance, and energy consumption. In: Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing, ARMS-CC '17, p. 1–6. Association for Computing Machinery, New York, NY, USA (2017). doi: 10.1145/3110355.3110356

[103] Meyers, S.: Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, 1st edn. O'Reilly Media, Inc. (2014)

[104] Microsoft: Multithreading with C and Win32. URL https://msdn.microsoft.com/en-us/library/y6h8hye8.aspx

[105] Microsoft: C++ AMP Overview (2018). URL https://docs.microsoft.com/it-it/cpp/parallel/amp/cpp-amp-overview

[106] Microsoft: GetLogicalProcessorInformation function (2018). URL https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getlogicalprocessorinformation

[107] Miller, R., Stout, Q.F.: Algorithmic Techniques for Networks of Processors. In: M.J. Atallah (ed.) Algorithms and Theory of Computation Handbook, 2nd edn., chap. 46, pp. 46:1–46:18. CRC Press, Boca Raton, FL, USA (1999)

[108] Mollick, E.: Establishing Moore's Law. IEEE Annals of the History of Computing **28**(3), 62–75 (2006). doi: 10.1109/MAHC.2006.45

[109] Moore, G.E., et al.: Cramming more components onto integrated circuits (1965)

[110] Moroz, L.V., Walczyk, C.J., Hrynchyshyn, A., Holimath, V., Cieliski, J.L.: Fast Calculation of Inverse Square Root with the Use of Magic Constant Analytical Approach. Appl. Math. Comput. **316**(C), 245–255 (2018). doi: 10.1016/j.amc.2017.08.025

[111] National Institute of Standards and Technology (NIST): FIPS PUB 197: Announcing the ADVANCED ENCRYPTION STANDARD (AES). National Institute for Standards and Technology, Gaithersburg, MD, USA (2001)

[112] Nguyen, H.: Gpu Gems 3, first edn. Addison-Wesley Professional (2007)

[113] Nichols, B., Buttlar, D., Farrell, J.P.: Pthreads Programming - A POSIX Standard for Better Multiprocessing. O'Reilly, Sebastopol, CA, USA (1996)

[114] Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable Parallel Programming with CUDA. Queue **6**(2), 40–53 (2008). doi: 10.1145/1365490.1365500

[115] NVIDIA: NVIDIA® GeForce® 7900 GTX – Reviews & Editorials (2006). URL www.nvidia.com/page/7900_reviews

[116] NVIDIA: CUDA C Best Practices Guide (2018). URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[117] NVIDIA: NVIDIA Turing GPU Architecture (2018). URL https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[118] NVIDIA: cuBLAS Library (2019). URL
https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf

[119] NVIDIA: CUDA C Programming Guide (2019). URL
docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[120] NVIDIA: cuFFT Library User's Guide (2019). URL
https://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf

[121] NVIDIA: cuRAND Library (2019). URL
https://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf

[122] NVIDIA: Thrust Quick Start Guide (2019). URL
https://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf

[123] OpenACC-Standard.org: The OpenACC Application Programming Interface (2011). URL
https://www.openacc.org/sites/default/files/inline-files/OpenACC_1_0_specification.pdf

[124] OpenACC-Standard.org: The OpenACC Application Programming Interface (2019). URL
https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf

[125] OpenMP Architecture Review Board: OpenMP Application Program Interface (2013). URL
www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[126] OpenMP Architecture Review Board: OpenMP Application Programming Interface (2018). URL
https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[127] Oracle: Java Language Specification - Chapter 17. Threads and Locks (2020). URL
https://docs.oracle.com/javase/specs/jls/se13/html/jls-17.html

[128] Owens, J.D., Luebke, D.P., Govindaraju, N.K., Harris, M.J., Krüger, J.H., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. In: Eurographics (2005)

[129] Patterson, D.A., Hennessy, J.L.: Computer Organization and Design, Fifth Edition: The Hardware/Software Interface, 5th edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2013)

[130] Peccerillo, B., Bartolini, S.: Phast library - enabling single-source and high performance code for gpus and multi-cores. In: 2017 International Conference on High Performance Computing Simulation (HPCS), pp. 715–718 (2017). doi: 10.1109/HPCS.2017.109

[131] Peccerillo, B., Bartolini, S.: PHAST - A Portable High-Level Modern C++ Programming Library for GPUs and Multi-Cores. IEEE Transactions on Parallel and Distributed Systems **30**(1), 174–189 (2019). doi: 10.1109/TPDS.2018.2855182

[132] Peccerillo, B., Bartolini, S.: Single-source library for enabling seamless assignment of data-parallel task-dags to cpus and gpus in heterogeneous architectures. In: Proceedings of the 10th and 8th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms, PARMA-DITAM 2019, pp. 3:1–3:4. ACM, New York, NY, USA (2019). doi: 10.1145/3310411.3310416

[133] Peccerillo, B., Bartolini, S.: Task-dag support in single-source phast library: Enabling flexible assignment of tasks to cpus and gpus in heterogeneous architectures. In: Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'19, pp. 91–100. ACM, New York, NY, USA (2019). doi: 10.1145/3303084.3309496

[134] Peccerillo, B., Bartolini, S.: PHAST Library website (2020). URL https://www.phast-library.com

[135] Peccerillo, B., Bartolini, S., Koç, Ç.K.: Parallel bitsliced aes through phast: a single-source high-performance library for multi-cores and gpus. Journal of Cryptographic Engineering pp. 1–13 (2017)

[136] Perkins, H.: EasyCL - Easy to run kernels using OpenCL (2016). URL https://github.com/hughperkins/EasyCL

[137] Peter Žužek: sycl-gtx - Implementation of the SYCL specification (2016). URL https://github.com/ProGTX/sycl-gtx

[138] Pino, S., Pollock, L., Chandrasekaran, S.: Exploring translation of openmp to openacc 2.5: lessons learned. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 673–682 (2017)

[139] Prongnuch, S., Wiangtong, T.: Heterogeneous Computing Platform for data processing. In: 2016 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS), pp. 1–4 (2016). doi: 10.1109/ISPACS.2016.7824762

[140] Python Software Foundation: The Python Standard Library, Concurrent Execution, threading – Thread-based parallelism (2020). URL https://docs.python.org/3.9/library/threading.html

[141] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. SIGPLAN Not. **48**(6), 519–530 (2013)

[142] Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media, Inc., Sebastopol, CA, USA (2007)

[143] Richardson, I.E.: Video Codec Design: Developing Image and Video Compression Systems. John Wiley and Sons, Inc., USA (2002)

[144] Ronan Keryell: triSYCL - Generic system-wide modern C++ for heterogeneous platforms with SYCL from Khronos Group (2016). URL https://github.com/triSYCL/triSYCL

[145] Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming, 1st edn. Addison-Wesley Professional (2010)

[146] Schäling, B.: The Boost C++ Libraries, 2nd edn. XML Press, Laguna Hills, CA, USA (2014)

[147] Sharlet, Dillon and Kunze, Aaron and Junkins, Stephen and Joshi, Deepti: Shevlin Park: Implementing C++ AMP with Clang/LLVM and OpenCL (2012). URL http://llvm.org/devmtg/2012-11/Sharlet-ShevlinPark.pdf

[148] Singh, G.: The ibm pc: The silicon story. Computer **44**(8), 40–45 (2011). doi: 10.1109/MC.2011.194

[149] Stepanov, A., Lee, M.: The standard template library, vol. 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304 (1995)

[150] Steuwer, M.: SkelCL Documentation (2015). URL `https://skelcl.github.io/doc/index.html`

[151] Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, IPDPSW '11, pp. 1176–1182. IEEE Computer Society, Washington, DC, USA (2011)

[152] Stroustrup, B.: The C++ Programming Language, 4th edn. Addison-Wesley Professional (2013)

[153] Sultana, N., Calvert, A., Overbey, J.L., Arnold, G.: From OpenACC to OpenMP 4: Toward Automatic Translation. In: Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale, XSEDE16. Association for Computing Machinery, New York, NY, USA (2016). doi: 10.1145/2949550.2949654

[154] Sutter, H., Alexandrescu, A.: C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series). Addison-Wesley Professional (2004)

[155] Tanaka, H., Ota, Y., Matsumoto, N., Hieda, T., Takeuchi, Y., Imai, M.: A new compilation technique for simd code generation across basic block boundaries. In: 2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 101–106. IEEE (2010)

[156] Tanase, G., Buss, A., Fidel, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Thomas, N., Xu, X., Mourad, N., Vu, J., Bianco, M., Amato, N.M., Rauchwerger, L.: The STAPL Parallel Container Framework. SIGPLAN Not. **46**(8), 235–246 (2011). doi: 10.1145/2038037.1941586

[157] Thakkur, S., Huff, T.: Internet streaming simd extensions. Computer **32**(12), 26–34 (1999). doi: 10.1109/2.809248

[158] The PHP Group: PHP Manual, Function Reference, Process Control Extensions, parallel (2020). URL `https://www.php.net/manual/en/intro.parallel.php`

[159] Thoman, P., Salzmann, P., Cosenza, B., Fahringer, T.: Celerity: High-level C++ for Accelerator Clusters. In: International European Conference on Parallel and Distributed Computing (Euro-Par), pp. 291–303 (2019)

[160] TIOBE Software BV: TIOBE Index for March 2020 (2020). URL `https://www.tiobe.com/tiobe-index/`

[161] TOP500.org: TOP500 - The List, November 2019 (2019). URL `https://www.top500.org/lists/2019/11/`

[162] Tripp, C., Hyde, D., Grossman-Ponemon, B.: FRC: A High-Performance Concurrent Parallel Deferred Reference Counter for C++. In: Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management, ISMM 2018, p. 14–28. Association for Computing Machinery, New York, NY, USA (2018). doi: 10.1145/3210563.3210569

[163] Trott, C.R.: Kokkos: The C++ Performance Portability Programming Model (2018). URL `https://github.com/kokkos/kokkos/wiki`

[164] Tseng, Y.Y., Huang, Y.H., Lai, B.C.C., Lin, J.L.: Automatic Data Layout Transformation for Heterogeneous Many-Core Systems, pp. 208–219. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

[165] Uesbeck, P.M., Stefik, A., Hanenberg, S., Pedersen, J., Daleiden, P.: An Empirical Study on the Impact of C++ Lambdas and Programmer Experience. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, p. 760–771. Association for Computing Machinery, New York, NY, USA (2016). doi: 10.1145/2884781.2884849

[166] Velho, L., Frery, A.C., Gomes, J.: Image Processing for Computer Graphics and Vision, 2nd edn. Springer Publishing Company, Incorporated (2008)

[167] Veras, R., Popovici, D.T., Low, T.M., Franchetti, F.: Compilers, Hands-off My Hands-on Optimizations. In: Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing, WPMVP '16. Association for Computing Machinery, New York, NY, USA (2016). doi: 10.1145/2870650.2870654

[168] Wallace, G.K.: The JPEG Still Picture Compression Standard. Commun. ACM **34**(4), 30–44 (1991). doi: 10.1145/103085.103089

[169] Wassenberg, J., Sanders, P.: Engineering a multi-core radix sort. In: European Conference on Parallel Processing, pp. 160–169. Springer (2011)

[170] Williams, A.: C++ Concurrency in Action, 2nd edn. Manning Publications (2019)

[171] Wulf, W.A., McKee, S.A.: Hitting the memory wall: Implications of the obvious. SIGARCH Comput. Archit. News **23**(1), 20–24 (1995). doi: 10.1145/216585.216588

[172] Xin Li, Shawmin Lei: Block-based segmentation and adaptive coding for visually lossless compression of scanned documents. In: Proceedings 2001 International Conference on Image Processing (Cat. No.01CH37205), vol. 3, pp. 450–453 vol.3 (2001)

[173] Yalamanchili, P., Arshad, U., Mohammed, Z., Garigipati, P., Entschev, P., Kloppenborg, B., Malcolm, J., Melonakos, J.: ArrayFire - A high performance software library for parallel computing with an easy-to-use API (2015). URL `https://github.com/arrayfire/arrayfire`

[174] Zahran, M.: Heterogeneous computing: Here to stay. Queue **14**(6), 31–42 (2016). doi: 10.1145/3028687.3038873