



Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

A Multi-purpose Framework for Efficient Parallelized Execution of Charged Particle Tracking

dissertation submitted to Universität Hamburg
for the degree of Dr. rer. nat.

by

Georgiana Mania

Faculty of Mathematics, Informatics and Natural Sciences
Informatics Department

Hamburg, April 2023

Reviewers

Prof. Dr. Thomas Ludwig
Prof. Dr. Simon McIntosh-Smith

Date of disputation

12.07.2023

Abstract

Estimates for High Luminosity runs at the Large Hadron Collider show that hundreds of petabytes of experimental data are expected to be processed in real-time and stored for future analysis; comparable amounts of simulated data add to the unprecedented volumes. In the High-Performance Computing field, many supercomputers have already reached petascale and even exascale by adopting many-core architectures and accelerators. These upgrades improve not only the computation power and the memory throughput, but also the performance per Watt. Nevertheless, numerous scientific applications still do not efficiently make use of the powerful hardware currently available. Despite of a sustained effort for code parallelization and GPU porting, the development productivity is hindered by the need to use a different programming language for each platform in order to achieve the best runtime performance, while most of the code contributors are natural scientists without a deep knowledge of computer science nor continuity in their projects.

Using concepts from functional programming and different language extensions, we prototyped the *vecpar* framework. It decouples the scientific code from the parallelization and offloading techniques, and allows a single-source C++ algorithm to be compiled for multiple hardware platforms, therefore eliminating the need to maintain different repositories. Additionally, this approach allows targeting future architectures by only updating *vecpar*'s backends without interfering with the physics-related algorithms.

In this thesis, we introduce a series of research studies that motivate our technical decisions regarding the framework, which is discussed at length together with the evaluation of its potential. Tested on different open-source benchmarks and architectures, the framework ensured the same level of performance as native parallel implementations in most of the cases, but with the advantage of using plain C++ code; for the edge cases, we proposed and/or implemented methods to alleviate the overhead which were validated by preliminary results. As a consequence, the *vecpar* framework can be used for multiple purposes, either (a) as an environment for testing the parallelization potential of a given algorithm and/or implementation, or (b) as a production-ready solution for targeting CPU and GPU architectures. Despite being developed for track reconstruction code, the framework's applicability is extended to any problem that fits its abstractions. However, multiple performance optimizations and portability improvements are still possible, and they are therefore listed as future work.

Zusammenfassung

Schätzungen für High-Luminosity Experimente am Large Hadron Collider zeigen, dass Hunderte von Petabyte Daten in Echtzeit verarbeitet und für zukünftige Analysen gespeichert werden sollen; vergleichbare Mengen an Simulationsdaten kommen zu diesen noch nie dagewesenen Mengen hinzu. Im Bereich des Hochleistungsrechnens haben viele Supercomputer durch den Einsatz von Many-Core-Architekturen und Beschleunigern bereits PetaFLOPS oder sogar ExaFLOPS erreicht. Diese Ansätze verbessern nicht nur die Rechenleistung und den Speicherdurchsatz, sondern auch die Leistung pro Watt. Dennoch nutzen zahlreiche wissenschaftliche Anwendungen die derzeit verfügbare leistungsstarke Hardware noch immer nicht effizient aus. Trotz anhaltender Bemühungen wird die Entwicklungsproduktivität dadurch behindert, dass für jede Plattform eine andere Programmiersprache verwendet werden muss, um die beste Leistung zu erzielen. Dies ist problematisch, da die meisten Entwickler Naturwissenschaftler sind, die häufig über keine fundierten Informatikkenntnisse verfügen.

Mithilfe von Konzepten der funktionalen Programmierung und verschiedener Spracherweiterungen haben wir einen Prototypen des *vecpar*-Frameworks entwickelt. Es entkoppelt den wissenschaftlichen Code von den Parallelisierungs- und Offloading-Techniken und ermöglicht es, einen C++-Algorithmus aus einem einzigen Quellcode für mehrere Hardwareplattformen zu kompilieren. Darüber hinaus ermöglicht dieser Ansatz die Unterstützung künftiger Architekturen, indem die Backends von *vecpar* angepasst werden, ohne die physikbezogenen Algorithmen zu beeinträchtigen.

In dieser Arbeit stellen wir mehrere Forschungsstudien vor, die unsere technischen Entscheidungen für das Framework motivieren und diskutieren sowohl das Framework als auch seine Leistungsfähigkeit ausführlich. Bei Tests mit verschiedenen Benchmarks und Architekturen hat das Framework in den meisten Fällen dasselbe Leistungsniveau wie native parallele Implementierungen erreicht. Es bietet allerdings den Vorteil, dass einfacher C++-Code verwendet werden kann; für Spezialfälle haben wir Methoden vorgeschlagen und/oder implementiert, die den Overhead verringern können und diese durch vorläufige Ergebnisse validiert. Folglich kann das *vecpar*-Framework für mehrere Zwecke verwendet werden, entweder (a) als Umgebung zum Ausloten des Parallelisierungspotenzials eines bestimmten Algorithmus oder einer bestimmten Implementierung oder (b) als produktionsreife Lösung für die Nutzung von CPU- und GPU-Architekturen. Obwohl das Framework für Track-Rekonstruktions-Code entwickelt wurde, lässt es sich auf viele weitere Probleme anwenden. Verschiedene Leistungsoptimierungen und Portabilitätsverbesserungen sind möglich und werden als zukünftige Arbeiten aufgeführt.

Acknowledgements

I would like to thank DASHH Helmholtz Graduate School for the Structure of Matter and my supervisors, Prof. Dr. Thomas Ludwig and Prof. Dr. Krisztian Peters, for believing in me and for giving me this amazing opportunity. Additionally, I am so grateful to Jun.-Prof. Dr. Michael Kuhn and Dr. Nicholas Styles for their support, guidance and valuable feedback throughout my doctoral studies.

A sincerely thank you to the ACTS Parallelization Group at CERN and in particular to Dr. Xiacong Ai, Prof. Dr. Heather Gray, Dr. Attila Krasznahorkay, Dr. Andreas Salzburger and Dr. Beomki Yeo for the interesting discussions and fruitful collaboration. Many thanks to my colleagues in the ATLAS Group at Deutsches Elektronen-Synchrotron DESY, the Scientific Group at the University of Hamburg, and the Parallel Computing and I/O Group at the University of Magdeburg, while special thanks go to Kira and Petra. I would also like to acknowledge the contribution of the Bachelor students Yannik Koenneker and Henning Lindemann to the development of this work.

I am so thankful for my oldest and dearest friends: Adela, Alex B., Alex G., Anca, Andreea, Cristina B., Cristina S., Diana, Irina, Lillian, Lucia, Matthew, Ramona, Roxana D., Roxana S. and their families, who steadily kept in touch and supported me from different countries in the world. Particularly, I am beyond grateful to Alex G., without whom I wouldn't have taken this challenge nor be the same person I am today. Lastly, my deepest thanks go to my parents, Mioara and Petrica, and my grandparents, who have always supported me in all of my endeavours. Thank you!

Contents

1. Introduction	1
1.1. High Performance Computing	1
1.1.1. Heterogeneous Architecture	1
1.1.2. Present and Future Supercomputers	6
1.2. High Energy Physics	8
1.2.1. Large Hadron Collider	9
1.2.2. ATLAS Experiment	11
1.2.3. Track Reconstruction in ATLAS	14
1.3. Motivation	17
1.4. Contribution	20
1.5. Thesis Structure	20
2. State of the Art	21
2.1. Basic Concepts	21
2.2. C++ Standard Template Library	22
2.3. Multi-platform Standards and APIs	23
2.3.1. OpenMP	23
2.3.2. OpenACC	27
2.3.3. OpenCL	29
2.3.4. SYCL	29
2.4. Vendor-Proprietary Frameworks	31
2.4.1. NVIDIA CUDA	31
2.4.2. AMD HIP	36
2.4.3. Intel DPC++	39
2.5. Compilers for Heterogeneous Programming	40
2.6. Functional Programming in HPC	42
3. Research Studies	45
3.1. Track Reconstruction Software	45
3.1.1. A Common Tracking Software	45
3.1.2. Case studies: Evaluate the Parallelization Potential with ACTS	49
3.1.3. Case study: Evaluate Matrix Inversion GPU Implementations	61
3.1.4. Conclusions	65
3.2. Framework Specifications	65

3.3. Clang-offload Prototype	67
3.3.1. Design and Execution Flow	68
3.3.2. Implementation Details	70
3.3.3. Limitations and Conclusions	72
4. Technical Design	73
4.1. Conceptual Design	73
4.1.1. Mathematical Concepts	73
4.1.2. Vecpar Abstractions	75
4.2. High-level Design	77
4.2.1. Execution Flow	78
4.3. Low-level Design	78
4.3.1. Data Types and Memory Resources	79
4.3.2. Algorithms	80
4.3.3. Execution Backends	88
4.3.4. Lambda Functionality	100
4.3.5. Algorithm Chain	101
4.4. Compilation for Multiple Targets	103
4.4.1. Native Single-source Functionality	103
4.4.2. Ompt Functionality	104
4.4.3. Conclusion	105
4.5. Performance Optimizations	105
4.6. Automated Tests	106
4.7. Summary	106
5. Related Work	107
5.1. Heterogeneous Frameworks	107
5.1.1. Kokkos	107
5.1.2. RAJA	109
5.1.3. Alpaka	110
5.1.4. Charm++	111
5.1.5. OpenMP	112
5.2. Performance Studies Review	112
5.3. Automatic Parallelization and Offloading Tools	115
5.3.1. Source-to-source Translators	115
5.3.2. Futhark Programming Language	116
5.4. Parallel Frameworks for Track Reconstruction	117
5.4.1. Allen	117
5.4.2. Patatrack	118

6. Evaluation	121
6.1. Evaluation criteria	121
6.1.1. Performance	121
6.1.2. Portability	122
6.1.3. Productivity	122
6.2. Test Configuration Setup	122
6.3. Benchmarks	123
6.3.1. Vecpar Internal Benchmark	123
6.3.2. BabelStream Benchmark	129
6.4. Track Reconstruction – RKN Stepper	135
6.4.1. Simplified RKN Stepper	136
6.4.2. A More Realistic RKN Stepper	141
7. Conclusion and Future Work	145
7.1. Contribution Summary	145
7.1.1. Potential	145
7.1.2. Limitations	146
7.2. Future Work	147
7.3. Development Experience Report	148
7.4. Conclusion	148
Bibliography	149
Appendices	
A. Contributors to Vecpar	161
B. Code Samples	163
B.1. Clang-offload Generated Code Samples	163
B.2. Vecpar SAXPY/DAXPY Benchmark	165
B.3. Detray RKN Algorithms	167
C. Vecpar Backlog	171
C.1. Known issues	171
C.2. Productivity and Performance Optimizations	171
D. Reproducibility Artifacts	173
D.1. Vecpar Internal Benchmark	174
D.2. BabelStream Benchmark	174
D.3. Track Reconstruction – RKN Stepper	175
E. List of Publications	177
E.1. Published	177

E.2. Accepted for Publication	177
List of Figures	179
List of Tables	183

1. Introduction

The field of High Performance Computing offers the support for groundbreaking scientific research in different domains, with High Energy Physics being only one of the examples. In this chapter, a brief introduction into both of these fields is presented, while the focus lies on the collaboration between them, which is the main motivation for this thesis.

1.1. High Performance Computing

High performance computing (HPC) refers to computing systems with extremely high computational power and/or storage capacity that are able to solve complex problems [Ludwig, 2020]. To evaluate the performance of a system, one of the most widely used metrics is the number of floating point operations per second: FLOP/s or FLOPS¹. In the last 20 years, HPC facilities have achieved a huge performance leap by increasing the number of compute processing units and their individual efficiency but also by employing diverse hardware resources. While the most powerful supercomputer evaluated on the Linpack benchmarks in 1993 had 1024 Central Processing Units (CPU) and achieved a performance of 59.70 GFLOPS, supercomputers in 2022 reached the exascale era with the *Frontier* supercomputing achieving an impressive performance of 1,102 PFLOPS with its almost 9 million compute cores, both CPU and Graphics Processing Unit (GPU) [Dongarra and Luszczek, 2011, TOP500, 2022].

In this section, the developments that led to today's capabilities are investigated and the latest architectures are described in detail since their characteristics shape the motivation for the current work.

1.1.1. Heterogeneous Architecture

In a famous article published initially in the journal *Electronics* in 1965, and republished in 1998, Gordon Moore, one of the co-founders of Intel corporation, predicted that the number of transistors in a hardware chip would double every 24 months [Moore, 1998]. Known as *Moore's law*, this estimate proved to be accurate for the next few decades until it started to slow down due to CMOS transistors' limits and power consumption requirements. Additionally, while wall-clock speedups were automatically obtained with each new generation of hardware due to increasing clock speeds, researchers noticed that

¹Commonly used FLOPS multipliers are GigaFLOPS ($1 \text{ GFLOPS} = 10^9 \text{ FLOPS}$), PetaFLOPS ($1 \text{ PFLOPS} = 10^{15} \text{ FLOPS}$) and ExaFLOPS ($1 \text{ EFLOPS} = 10^{18} \text{ FLOPS}$)

there is an imbalance between memory bandwidth and processor speed by looking at the cache miss/hit ratio per average cycle per access, which they named the *memory wall* [Wulf and McKee, 1995]. This practically meant that the computational performance is limited by the speed of read/write accesses from/to the memory modules rather than the CPU's speed. Consequently, new approaches started to be investigated.

Defined in 1966 and extended in 1972, Flynn's taxonomy is a categorization of parallel computer architectures based on the way flows of instructions and data are handled [Flynn, 2011]. Therefore four classes are proposed:

- Single Instruction, Single Data (SISD) which corresponds to the von Neumann architecture where there is one control unit that loads the instruction and one bus that connects the processing unit to the memory;
- Single Instruction, Multiple Data (SIMD) where the same instruction is executed for several chunks of data; the GPU is a classic example of a SIMD architecture;
- Multiple Instruction, Multiple Data (MIMD) where different processors can execute an independent flow of instructions on different sets of data in parallel; an example is a recent CPU, which has several (independent) cores on the same card;
- Multiple Instruction, Single Data (MISD) where several execution units operate on the same data flow; this model has no practical use cases in HPC for now.

A major breakthrough came with the development of *multicore architecture*. In this case, a processor has multiple cores available on the same chip, each having its own local cache while being connected to a shared cache/global memory interface. Refining the same idea, the *manycore architecture* was proposed. This is an enhanced multicore where the chip contains tens to hundreds of simpler cores which allow massive parallel execution [Vajda, 2011]. Based on how the memory is accessed from the cores, several models are valid:

- Uniform Memory Access (UMA) model – the processors use a symmetrical architecture in respect to the memory module and therefore the latency and bandwidth are identical;
- Non-Uniform Memory Access (NUMA) – there are at least two different memory spaces, because the memory is physically distributed but logically shared. In this case the latency depends on whether the read/write call is made to a local or remote memory location, but within the same order of magnitude. The data is transferred from one node to another through high-speed Ethernet connections.
- cache-coherent Non-Uniform Memory Access (ccNUMA) – similar to NUMA but with the extent that the coherence ensures that modifying the data from one cache line which resides in several CPU caches will invalidate the caches of all the others.

Depending on the processing unit types and the memory access models, two designs were proposed: *homogeneous architectures*, where all the processing cores are of the same

type and share the same (logical) address space (UMA, NUMA, ccNUMA), and *heterogeneous architectures*, where two or more types of cores are available within the same computing node but with a distributed memory model (NUMA), as shown in Figure 1.1.

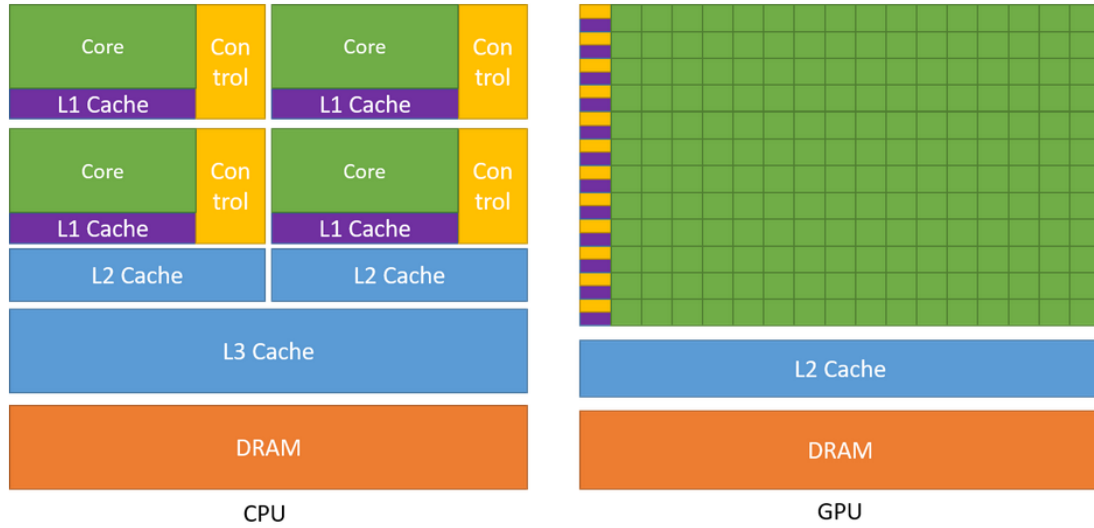


Figure 1.1.: CPU and GPU memory illustration by NVIDIA [NVIDIA Corporation, a]

A concrete example of a heterogeneous compute node with one CPU and four GPUs is shown in Figure 1.2. While GPUs are usually the common choice for *accelerators*²,

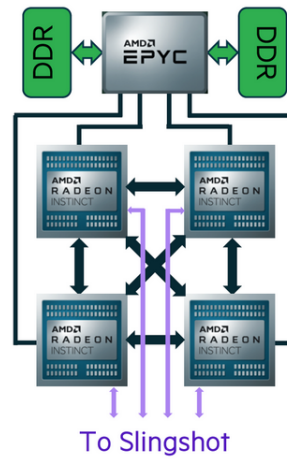


Figure 1.2.: AMD EPYC Processor with 64 cores is fully connected to the four AMD Radeon Instinct GPU and to the two high speed DDR cards in Heterogeneous node of the Frontier Supercomputer [Oak Ridge National Laboratory, 2022]

other cards can also deliver performance improvements. These alternatives include

²Accelerators are hardware processing units tailored to perform certain classes of tasks more efficiently than a CPU

co-processors³, Field-Programmable Gate Array⁴ (FPGA), and Application Specific Integrated Circuits⁵ (ASIC).

The Peripheral Component Interconnect Express (PCIe) connects the distributed memory locations as shown in Figure 1.3 [Stephen Jones, 2017]. It is one of the major contributors to performance penalties, but its impact is alleviated with every new generation that gets released. This is achieved by increasing the number of connection lines and ensuring simultaneous bi-directional throughput. For example, PCIe Generation 4 guarantees a bandwidth of 64Gb/s, which is double what PCIe Gen3 could offer.

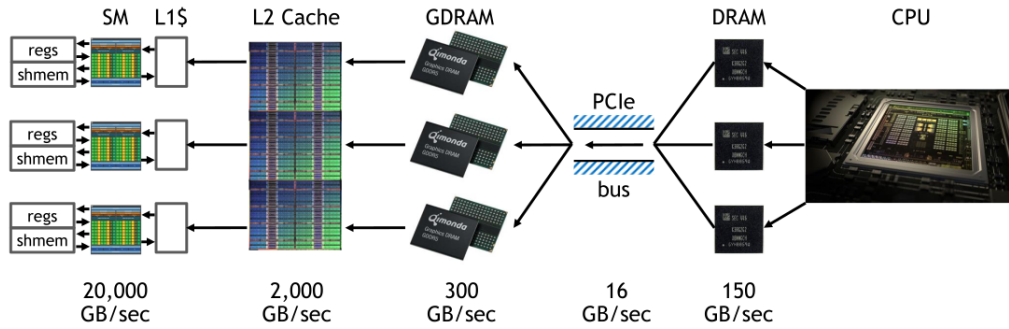


Figure 1.3.: NVIDIA memory bandwidth in 2017 [Stephen Jones, 2017]

To achieve top performance, state-of-the-art processors implement low-level optimization strategies and techniques [Hager and Wellein, 2011]. Some of the most widely used ones are mentioned below:

1. Development of instruction pipelines – instructions are executed in stages (e.g. read, fetch, jump) with pipelines overlapping for different instructions
2. Hardware dynamic instruction scheduling – the hardware is allowed to reorder instructions to exploit instruction-level parallelism, independently from the compiler's actions
3. Hyper-threading or Simultaneous Multi-Threading (SMT) – this provides better cache and instruction pipeline utilisation, while decreasing power consumption
4. SIMD instructions issue an identical operation on each element of an array of floating point operands (which are usually stored in registers). Also, the number of available registers and their sizes were extended to ensures support for data level parallelism. The x86 architecture went through several upgrades, each generation extending the instruction set further. The first addition was 8 MMX registers \times 64 bit. This was followed by a series of extensions — Streaming Single-Instruction-Multiple-Data Extensions (SSE, SSE2, SSE3, SSE4, SSE5) — which introduced 16 registers \times 128 bit. Now two operands fit in an SSE register and the notion of *packed*

³A co-processor has the same CPU structure but is focused on floating point compute units and less on interconnect or storage capabilities; A popular example is the Intel Xeon Phi

⁴FPGA are reconfigurable integrated circuits that can be tailored for simple logical operations

⁵Similar to FPGA in some sense, ASICs can target complex operations and cannot be reconfigured

operation as opposed to the existing *scalar* operations is introduced. Then Intel’s Advanced Vector Extensions (AVX and AVX2) doubled the size of the SSE registers to 256 bits while adding masking and shuffling operations. Finally, AVX512 provides $32 \text{ registers} \times 512 \text{ bit}$. An important point to mention here is that while CPU’s registers kept expanding, the GPU registers remain at the same values defined by the IEEE 754-2008 standard: 32 bit for single-precision and 64 bit for double-precision [IEEE, 2008]. This often leads to different results delivered by the arithmetic units from these two architectures.

5. Use a simplified instruction set – While the term *reduced instruction set computer* (RISC) was coined in 1980, the first implementations started to emerge in 1985, with *Advanced RISC Machine* (ARM) being one of them.

Nevertheless, just hardware advances are not enough to achieve top performance unless the software and middleware are also upgraded to leverage the physical capabilities. The interaction between all these components is illustrated in Figure 1.4. Parallel and

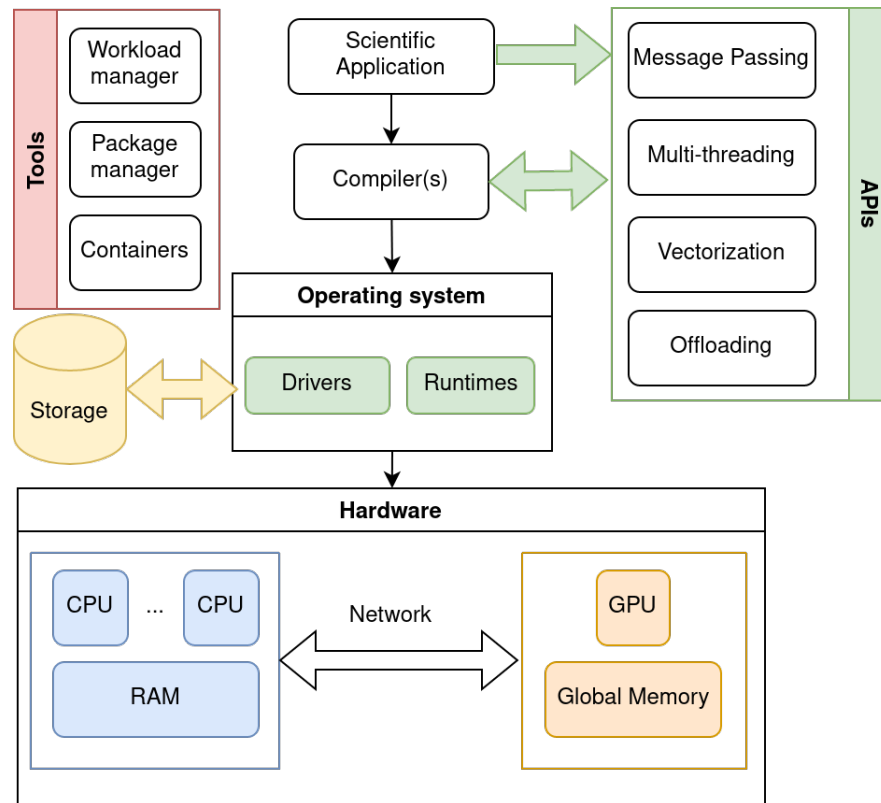


Figure 1.4.: Software stack of an HPC compute node

distributed file systems like *Lustre* [Braam, 2019] provide fast I/O operations when handling storages of petabytes. Package managers like *Spack* [Gamblin et al., 2015] ensure easy installation, configuration and update of software dependencies automatically and without requiring administrator rights, while *containers* pack all the dependencies together. Workload managers like *SLURM* [Yoo et al., 2003] provide an efficient scheduling

mechanism of the computing jobs and distribute the computations. Highly optimized compilers connect parallel programming techniques to the hardware through application programming interfaces (API) like message passing, multi-threading, vectorizations, acceleration, etc.

1.1.2. Present and Future Supercomputers

The most powerful supercomputers in the world are evaluated constantly and the results are published twice a year in a Top500 list [TOP500, 2022, Dongarra and Luszczek, 2011]. Commonly, two performance values are provided: the theoretical peak R_{peak} and the achieved maximum R_{max} . The former is an upper bound and represents the number of floating-point additions and multiplications (in full precision) that can be completed during a cycle; this depends on hardware factors like the clock frequency of the processor, the number of cores or the SIMD operation support. The latter is the actual number of floating point operations in a cycle measured given a specific benchmark. Linpack and Lapack benchmarks contain linear algebra routines, while a parallel implementation of Linpack called *High-Performance Linpack Benchmark for Distributed-Memory Computers*⁶(HPL) is used for ranking the supercomputers. HPL package provides algorithms that solve a dense system of linear equations in double precision with no restriction on the problem size or matrix element types but with constraints on the accuracy of the results.

Based on the statistics released in 2022, the ten highest ranked supercomputers use a variety of processing units from different vendors: seven of them have accelerators from either AMD or NVIDIA combined with many-core CPU from IBM, AMD and Intel, one is ARM based while one has a Sunway many-core CPU [TOP500, 2022]. A brief summary of their hardware is presented in Table 1.1.

Rank	Name	R_{max} (PFLOPS)	CPU model	GPU Model
1	Frontier	1 102.00	AMD EPYC	AMD MI250X
2	Fugaku	442.01	Fujitsu A64FX	-
3	LUMI	309.10	AMD EPYC	AMD MI250X
4	Leonardo	174.70	Intel Xeon 8358	NVIDIA A100
5	Summit	148.60	IBM Power9	NVIDIA V100
6	Sierra	94.64	IBM Power9	NVIDIA V100
7	TaihuLight	93.01	Sunway SW 26010	-
8	Perlmutter	70.87	AMD EPYC 7763	NVIDIA A100
9	Selene	63.46	AMD EPYC 7742	NVIDIA A100
10	Tiahe-2A	61.44	Intel Xeon E5-2692v2	-

Table 1.1.: Top 10 HPC Supercomputers in November 2022 [TOP500, 2022]

Intel and NVIDIA are dominating the market for CPU and GPU respectively, while AMD, Intel, Fujitsu or IBM have started to become significant competitors, as shown in Figure 1.5. Fujitsu ARM A64FX ensured *Fugaku's* supremacy in Top500 for two consecutive years and was only recently overtaken by an AMD-based cluster, *Frontier*. Also,

⁶<https://netlib.org/benchmark/hpl/>

many of the supercomputers use hybrid platforms, with CPU from one vendor and GPU from another. An example of using this approach is *Levante* supercomputer located at the German Climate Computing Center (DKRZ) in Hamburg, Germany, which uses two AMD 7713 CPU connected to four NVIDIA A100 GPU for the nodes in the GPU partition.

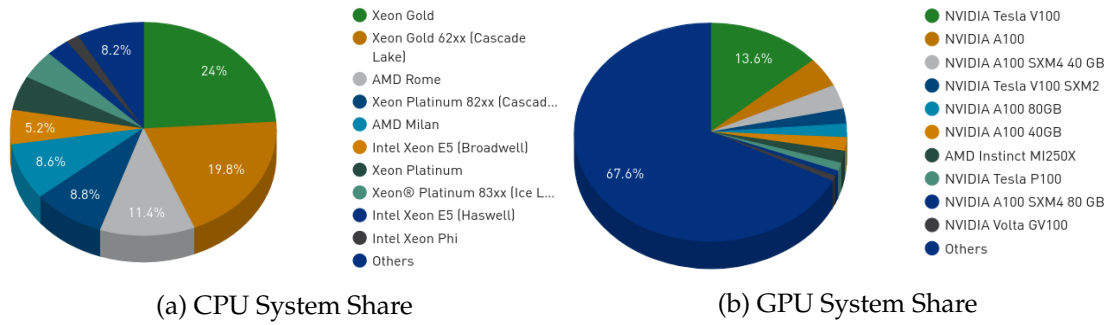


Figure 1.5.: CPU and GPU models shares in Top 500, November 2022 [TOP500, 2022]

The decision to use accelerators is not driven only by the desire to increase performance but stems also from the need to reduce energy consumption. GPUs can offer better FLOPS per Watts (FLOPS/Watt) and therefore ensure a more *energy efficient* resource utilization ratio. This metric is used to rank the supercomputers in GREEN500⁷. New additions to top 10 fastest supercomputers like *Frontier* and *LUMI* are among the top 10 most efficient too with more than 50 GFLOPS/Watt while older ones like *Fugaku* and *Summit* deliver around 15 GFLOPS/Watt. Nevertheless, focusing on graphic cards only, *the leap in performances is correlated with an increase in power consumption*⁸, as it is clearly shown in Table 1.2 [Fujitsu, 2020, NVIDIA Corporation, 2017, NVIDIA Corporation, 2021, AMD, 2020, AMD, 2021].

Name	FP32	FP32 Tensor	FP64	FP64 Tensor	Power
Fujitsu A64FX	6.8	-	3.4	-	170
NVIDIA V100 (PCIe)	14	112	7	-	250
NVIDIA V100 (SXM2)	15.7	7.8	125	-	300
NVIDIA A100-40GB-PCIe	19.5	156	9.7	19.5	250
NVIDIA A100-80GB-SXM2	19.5	156	9.7	19.5	400
AMD Instinct MI100	23.1	46.1	11.5	-	300
AMD Instinct MI250X	47.9	95.7	47.9	95.7	500

Table 1.2.: Computing capabilities of the floating point (FP) units and power consumption for the major graphic cards compared to a top CPU (Fugaku). The measurement units for performance and power are TFLOPS and Watts, respectively;

Therefore, less optimized scientific applications that do not take full advantage of the huge compute potential, might end up using more energy without speeding up their

⁷<https://www.top500.org/lists/green500/>

⁸The vendor-released specifications include the *tensor cores*’ performance too, which are special FP units that enable mixed-precision computing, dynamically adapting calculations to accelerate throughput while preserving accuracy. Commonly, FP8, FP16, FP32 and FP64 units are available in high-end graphic cards.

operations despite running on latest hardware.

Table 1.2 also shows that recent AMD GPUs matched and in some cases even outperformed NVIDIA's capabilities, and therefore qualify themselves as eligible solutions for future supercomputers. Examples already include *Frontier*, which is on its way to replacing *Summit*, and *El Capitan*, which is scheduled to replace *Sierra* in 2023.

Moreover, Intel has announced the *Xe Architecture "Ponte Vecchio"*, which is currently being installed in the new exascale supercomputer *Aurora* at Argonne National Laboratory, in the United States [Argonne National Laboratory, 2022]. In addition, the company is exploring new architectures for a high-performance GPU targeting AI and supercomputing, which is scheduled to be released in the following years.

The vendors diversity translates into a variety of instruction sets, one for each hardware resource. Moreover, while most of the top supercomputers are heterogeneous systems, 75% of the Top500 list are still using CPU as only compute resource. Also, Cloud Data Centres offer huge parallelization opportunities employing ARM-based CPUs like Graviton, dedicated ASICs like the Tensor Processing Unit (TPU) or GPUs.

In this context, scientific applications are required to ensure a high level of *portability*. This is the ability to execute a computer program on architectures with different low-level instruction set and memory handling routines. High-Energy physics is one of the exciting scientific domains that relies on massive scale parallelism to handle large amounts of real-time data that need to be processed and stored in less than a millisecond. Therefore, the physics software needs to be both portable and performant.

1.2. High Energy Physics

According to *Nature Journal*, Particle Physics or High-Energy Physics (HEP) "is the study of the elementary building blocks of matter and radiation and their interaction"⁹. The best understanding on how the fundamental particles interact with three out of the four fundamental forces is comprised in the Standard Model (SM).

Nevertheless, this theory is still unable to answer a number of very interesting questions like: Why does *gravity* seem to behave differently from the other fundamental forces, hence is missing from the SM? What are the properties and decays of the *Higgs boson*, whose existence was postulated in 1964 but its discovery was confirmed experimentally only years later, in 2012? What particles and phenomena are responsible for the dark matter and dark energy? How is the antimatter involved in the Universe? To search for answers, physicists design experiments to observe and measure the particle interactions in colliders, such as the Large Hadron Collider (LHC) at CERN.

⁹<https://www.nature.com/subjects/particle-physics>

$$\mathcal{L}_{int} = \int_0^T \mathcal{L} dt \quad (1.2)$$

While the LHC is constantly being upgraded with each run¹¹, a major turning point is scheduled for 2029, when the luminosity is set to be increased considerably and will mark the beginning of the *High-Luminosity Large Hadron Collider (HL-LHC) era*. The amount of data captured by the detectors will increase proportionally. Currently, the LHC produces 50 PB per year [CERN, 2017]. While at the end of 2018, 150 inverse femtobarns of data were recorded in total since the beginning of Run 1, HL-LHC is estimated to produce 250 femtobarns/year. Recent technical reports about data taking during Run 3 confirm this assumption and show that LHC already produces $1fb^{-1}$ a day, even before HL, as shown in Figure 1.7 [ATLAS Collaboration, 2022c].

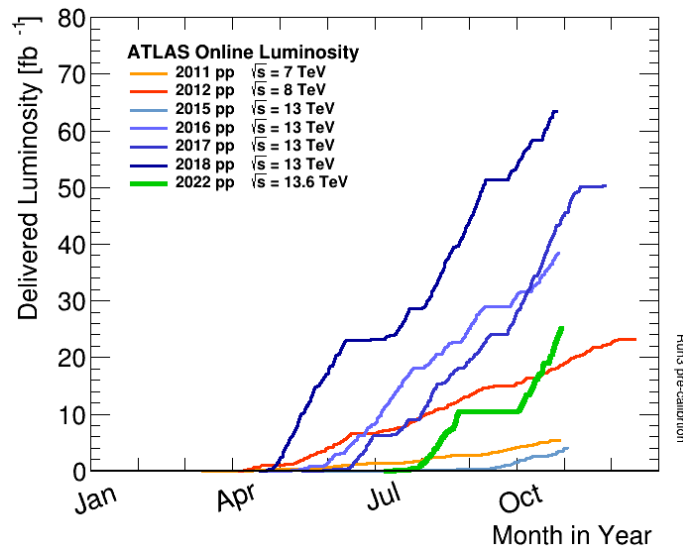


Figure 1.7.: Delivered Luminosity for 2011-2022 [ATLAS Collaboration, 2022c]

Worldwide LHC Computing Grid (WLCC) [Shiers, 2007] is the multi-tier infrastructure¹², which currently handles these large amount of data as follows:

- Tier 0 — a data centre with 73 000 cores located at CERN mostly used for storing raw-data, data aggregation, initial reconstruction and transfers to long storage;
- Tier 1 — 13 key data centres in Asia, North America and Europe focused on permanent storage, re-processing and data analysis;
- Tier 2 — 160 universities and scientific institutions around the world, which perform simulation and analysis;
- Tier 3 — department cluster or PC where end-user analysis is performed.

¹¹LHC has had three periods of data taking: Run 1 (2009-2013), Run 2 (2015-2018) and Run 3 (2022-2026). Between consecutive operational runs, the so called *Long Shutdown* periods are used to upgrade the detector.

¹²Source: <https://wlcg-public.web.cern.ch/tiers>, Accessed: October, 2022

In the current regime, the LHC produces one billion collisions per second. The particles resulting from a collision need to be counted and tracked by physicists. The charge, energy and momentum of a particle can be inferred from its trajectory — or in short, *track*. For example, high momentum particles travel almost in straight lines while low momentum ones have higher curvature. Therefore the *track reconstruction* process is of a great importance in both (a) validating the properties of a particle and (b) identifying potential new particles.

To capture the information leading to computing the tracks, detectors use different type of sensors as shown in Figure 1.8.

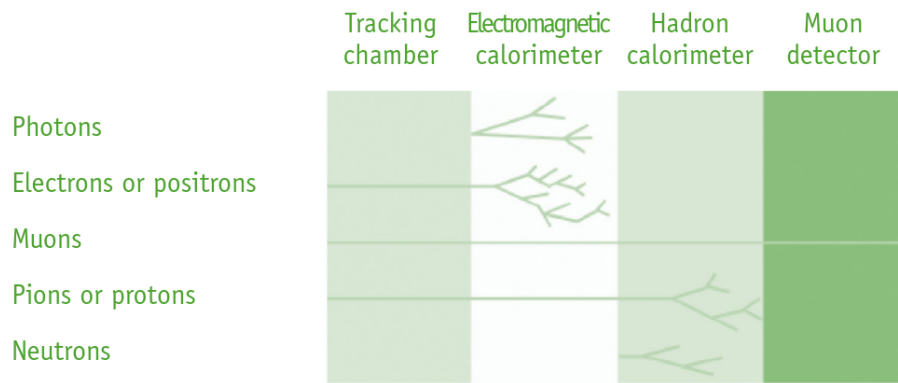


Figure 1.8.: Detectors based on particle type [LHC Education and Group, 2017]

Based on the technical design, sensors types and physics goals, there are four large experiments at CERN: *ATLAS* — A Toroidal LHC Apparatus is a general-purpose detector for "improving our understanding of the fundamental constituents of matter"¹³ by providing precise measurements and search for new physics beyond the standard model; *CMS* — Compact Muon Solenoid has a similar focus as *ATLAS* but the search is done with different technical solutions and magnet design; *ALICE* — A Large Ion Collider Experiment is a specialised detector, which studies the properties of quark-gluon plasma, a state of matter where quarks and gluons, under conditions of very high temperatures and densities, are no longer confined inside hadrons; *LHCb* — is a specialised detector focused on the symmetry between matter and antimatter present in interactions of the *b*-mesons¹⁴. While this work is intended to potentially be useful in a wide range of applications, the *ATLAS* detector is chosen as a particular use case and therefore it will be described in more details.

1.2.2. ATLAS Experiment

ATLAS is the largest volume particle detector ever built, measuring 46 meters long, 25 meters high and 25 meters wide. Its current structure consists of several layers with

¹³<https://atlas.cern/Discover/Physics>

¹⁴Particles containing the *b* quark

different types of sensors wrapped concentrically around the proton beam lines, as shown in Figure 1.9.

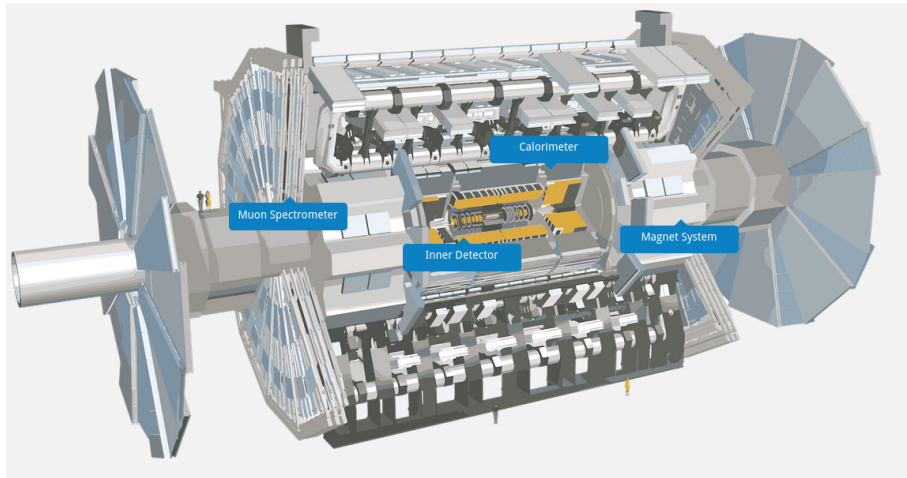


Figure 1.9.: The ATLAS detector at CERN [ATLAS Collaboration, 2022a]

A system of powerful magnets is employed to produce a magnetic field of 2T that bends the particles' trajectories so that their momenta can be computed. The hardware used for Run 2 includes three main components with different functions:

1. The inner detector — measures charge, direction and momentum using three different system of sensors in a magnetic field parallel with the beam axis:
 - Pixel detector – A grid of 92 million pixels, which provide just as many data acquisition channels;
 - Semiconductor Tracker (SCT) – A silicon microstrip tracker consisting of 4088 two-sided modules arranged in four barrels and two end-caps with nine wheels each. They capture over 6 million channels.
 - Transition Radiation Tracker (TRT) – A network of 350 000 straw tubes with 4mm diameter, in the centre a 0.03mm diameter gold-plated tungsten wire.
2. The calorimeter — measures the energy¹⁵ a particle loses as it passes through the detector. It has two components:
 - Liquid Argon (LAr) Calorimeter, which measures the energy of electrons, photons and hadrons. It features layers of metal (either tungsten, copper or lead) that absorb incoming particles, converting them into a “shower” of new, lower energy particles. It provides 110 000 readout channels.
 - Tile Hadronic Calorimeter, which measures the energy of hadronic particles, which do not deposit all of their energy in the LAr Calorimeter. It is made out of several layers of steel and plastic scintillating tiles. When a particle hits the

¹⁵In particle physics, the unit that is most frequently used for energy is the electronvolt (eV) and its derivatives Giga-electronvolt ($1\text{GeV} = 10^9\text{eV}$) and Tera-electronvolt ($1\text{TeV} = 10^{12}\text{eV}$)

layers of steel, a shower of new particle is created. The scintillators convert high energy radiation into photons which are later converted into an electric current with the intensity proportional to the original particle's energy.

3. The muon spectrometer — identifies and measures the momenta of muons (which pass through previous layers undetected, as shown in Figure 1.8). It has four major components:

- Thin Gap Chambers (440 000 readout channels)
- Resistive Plate Chambers (380 000 readout channels)
- Monitored Drift Tubes - measure curves of tracks using over 350 000 tubes
- Cathode Strip Chambers – measure precision coordinates at the ends of the detector (70 000 readout channels)

To summarize, over 10 million data acquisition channels record electric pulses from 10^9 collisions per second. Nevertheless, not all the collisions that take place within ATLAS can be stored on disk due to the high data volume, and neither are all the collisions "interesting" from the physics point of view. Thus, two filters, which decide upon keeping an event in under a millisecond, are employed: L1 — a hardware electronics trigger, which chooses approximately 1 in 10 000 events, and High-Level-Trigger (HLC) — a software-based one, which reconstructs the tracks and keeps 1 in 100 events.

In the shutdown period between Run 2 and Run 3, several upgrades were done for ATLAS detector: significant architecture improvements at LAr and calorimeter in the L1 trigger and a new *small wheel* in the muon spectrometer.

In preparation for HL-LHC, ATLAS will upgrade its geometry with a new all-silicon Inner Tracker (ITk), with an impressive number of readout channels: 5 billion in the pixel detectors and 50 millions in the strip detectors [Meng, 2021]. The chip's activity depends on the *hit*¹⁶ rate and the position of the module in the detectors and it is able to transmit data via 4 links with a speed of 1.28 Gbit/s [Meng, 2021]. A common data flow is depicted in Figure 1.10.

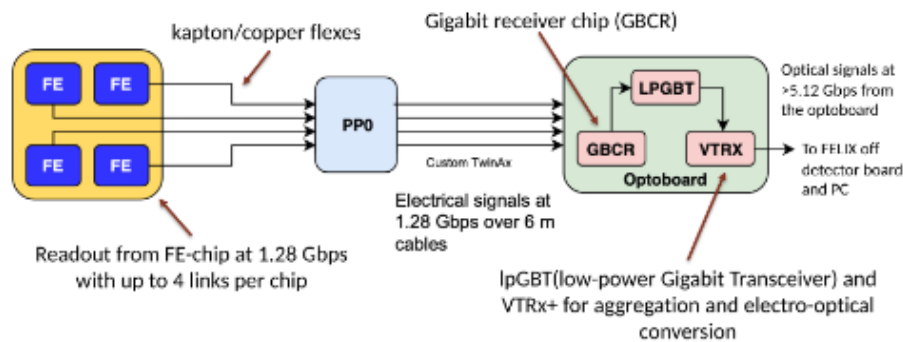


Figure 1.10.: Data flow in the ITk detector [Meng, 2021]

¹⁶A *hit* is the 3D location where a charged particle intersects a sensitive surface of a detector.

1.2.3. Track Reconstruction in ATLAS

In this section, the ATLAS reconstruction process is described, together with the main concepts required for understanding the algorithmic flow.

1.2.3.1. Concepts

To describe a trajectory, ATLAS uses cylindrical coordinate system around the beam line (z direction). The parameters that fully characterise the track are called *track parameters* and they can be defined as *bound track parameters*: $(l_1, l_2, \phi, \theta, q/p)$, and *free track parameters*: (\vec{r}, \vec{p}, q) . In these notations, l_1 and l_2 are the local coordinates in the measurement plane, q is the particle's charge, \vec{p} is the global momentum that is conserved in the plane perpendicular to the beam line¹⁷, ϕ and θ are the azimuthal and polar angle of the particle momentum vector direction at the particle's position, q/p is the *curvature* in the cylindrical system, and \vec{r} and \vec{p} are vector projections in 3D space of the Cartesian system. A particular case with a measurement surface perpendicular to the beam and defined at the point of closest approach to a reference point, like the beam (z) axis, is called the *perigee surface* and it is shown in Figure 1.11¹⁸. Now, l_1 and l_2 are replaced by d_0 and z_0 , which are called *transverse and longitudinal impact parameters*.

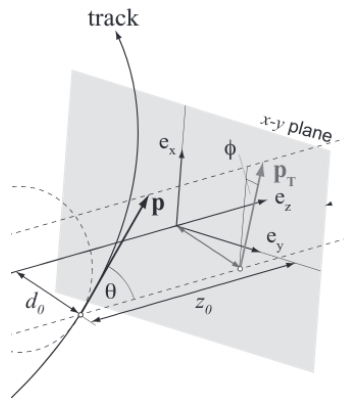


Figure 1.11.: Particle tracking inside ATLAS detector using cylindrical coordinate system (Source: Nicholas Style, EDIT2020)

As briefly mentioned before in this chapter, properties like the intensity and orientation of the electric or magnetic field (\vec{E} and \vec{B} respectively) are tightly connected to a particle's trajectory. When a uniform electromagnetic field is applied, the movement governed by the Lorentz force is described in Equation 1.3.

$$\vec{F} = q(\vec{E} + \vec{v} \times \vec{B}) \quad (1.3)$$

Since the electric field is negligible within ATLAS detector, the value of the Lorentz force becomes $F = qvB\sin(\theta)$, where θ is the angle between the magnetic field vector and the

¹⁷Therefore *transverse momentum* (p_T) is commonly used in computations.

¹⁸Excellence in Detector and Instrumentation Technologies (EDIT) 2020, Feb 17-18 2020, DESY, Hamburg

particle's velocity vector (\vec{v}). Now based on the vectors' orientation, three trajectories shapes can be identified:

- a straight line, if $\vec{v} \parallel \vec{B} \rightarrow \theta = 0^\circ$ or $\theta = 180^\circ \rightarrow F = 0$
- a circular trajectory, with radius r , if $\vec{v} \perp \vec{B} \rightarrow \theta = 90^\circ \rightarrow$ Lorenz force equals the centripetal force, which leads to computing the radius: $r = mv/qB$
- helical trajectory due to the fact that one component of the velocity vector is constant in magnitude and direction which implies straight-line motion, while the other component of the velocity vector is constant in speed but uniformly varies in direction, which implies circular motion.

When the magnetic field is non-uniform, which is the case for ATLAS detector, the equation of motion becomes more difficult to solve, as shown in Equation 1.4. s is the deviation from the straight line trajectory called *sagitta*¹⁹, $B(r)$ is a 3D vector containing the intensities of the magnetic field in every point of the detector and $\frac{d\vec{r}}{ds} = \vec{T}$ is the normalised tangent vector to the track.

$$\frac{d^2\vec{r}}{ds^2} = \frac{q}{p} \left(\frac{d\vec{r}}{ds} \times B(\vec{r}) \right) = \frac{q}{p} (\vec{T} \times \vec{B}(\vec{r})) \quad (1.4)$$

This equation can be solved numerically by using a 4th order Runge-Kutta-Nystrom (RKN) algorithm that is tailored for second order differential equations. An adaptive method based on RKN was developed by ATLAS scientists to reduce the error by adjusting the step size according to the error tolerance as shown in Figure 1.12 [Lund et al., 2009].

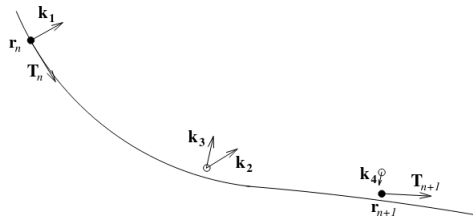


Figure 1.12.: Runge-Kutta-Nystrom stages [Lund et al., 2009]

The main steps are:

1. Evaluate the equation in 4 stages k_1 - k_4 using the RKN formulas
2. Compute the local error estimate $\epsilon = h^2(k_1 - k_2 - k_3 + k_4)$; where h is the current step size
3. Evaluate the quality of the solution based on acceptance criteria $|\epsilon| < 4\tau$
4. Evaluate the step size using user specified error tolerance $h_{n+1} = h_n \left(\frac{\tau}{|\epsilon|} \right)^{\frac{1}{q+1}}$, with q the order for the lower-order solution (which is 3 in this case)

¹⁹The sagitta of a circular arc is the distance from the centre of the arc to the center of its base

5. Trim the step size using a limitation criterion $\frac{1}{4}h_n \leq h_{n+1} \leq 4h_n$
6. Compute T and r using the RKN formulas with adjusted step h

The precision of track reconstruction parameters is sensitive to the amount of material of the tracking detector. The description of the material including the geometrical layout and atomic composition — or its *geometry* — is based on engineering design drawings of the detector, together with supporting measurements of the masses, dimensions and compositions of detector components. This is later complemented by studies of interactions sensitive to the detector material in data, which allow differences to the assumed material distribution to be identified and eventually corrected for. Computationally, these are encoded in a covariance matrix that has the derivatives of each track parameter with respect to those at the starting point of the propagation.

The estimates produced by the numerical integrator are combined with the measurements recorded by the sensors. A commonly used approach in particle physics is to use the Kalman Filter (KF) algorithm [Billoir, 1984, Murphy, 2012]. This is a statistical model that ensures better precision than when considering either measurements or estimates alone. Another benefit is that KF requires just the current state to compute the next state because the entire history is already aggregated into the current state. To increase the precision of an estimated trajectory, an additional *smoothing* step can be performed at the end; this will use information about all the measurements and will update the track states and their associated covariances.

1.2.3.2. Event Data Model

A generic reconstruction flow involves a series of steps as shown in Figure 1.13.

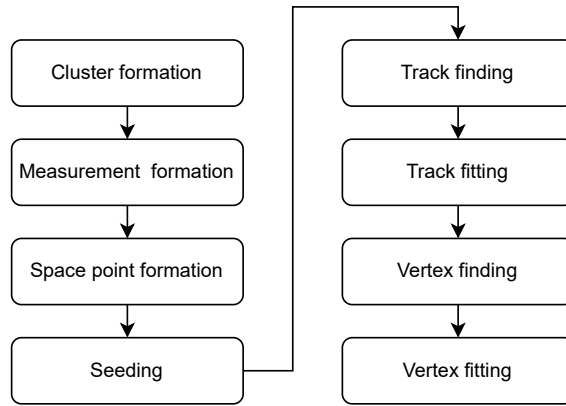


Figure 1.13.: Generic charged particle reconstruction flow

Data coming from detector's sensors are initially clustered together based on pattern matching algorithms. For each cluster, if the collected charge is above a specific threshold, a *measurement* can be constructed. This contains the position of a hit together with the covariance matrix associated with it. Then the measurement is filtered-by detector-specific knowledge and a *space point* is obtained.

The seeding algorithm matches three space points on different detector layers which could potentially be the source of a trajectory, as depicted in Figure 1.14. Since any three

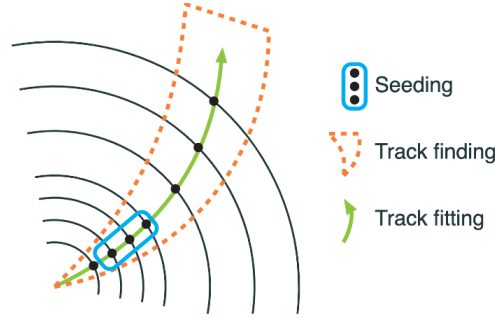


Figure 1.14.: Visual representation for seeding, track finding and fitting [Salzburger, 2022]

points within a specific detector's region described by $\Delta\phi$ in the xy -plane and $\Delta\theta$ in the rz -plane could belong to the same trajectory, a combinatorial approach is followed. Moreover, a space point can be part of several potential trajectories, creating (impossible) track candidates, which are filtered out as soon as possible to minimise unnecessary computational overheads.

The seeds are then fed into track finding and fitting algorithms, which could be done either simultaneous or as separate steps and which follow a potential trajectory starting from the seeds through the detector's adjacent layers and provide precise estimates for the track parameters and covariances by propagating them from one layer of the detector to the next one. This process is also called *propagation* or *extrapolation*. Usually, at this point, the track duplicates are removed²⁰ and the remaining viable candidates are processed further by vertex²¹ finding and fitting steps, which can employ machine learning algorithms to deliver optimum results.

In summary, track reconstruction algorithms are mostly sequential, with each step depending on the result of the previous one. They employ mathematic operations with a high-computational cost, like matrix multiplications, matrix inversion, partial derivatives calculations in covariances and jacobians, all in a combinatorial environment. When the amount of data grows for the HL-LHC, the execution times are expected to increase linearly, therefore posing a significant challenge on the computing infrastructure.

1.3. Motivation

Track reconstruction software is used when selecting the events in real-time (*online*) trigger and when performing the *offline* physics analysis as shown in Figure 1.15. The online reconstruction needs to run very fast in order not to miss any events and therefore some

²⁰Typically 20 000 seeds generate 2000 track candidates which lead to 1000 tracks.

²¹A *vertex* is the actual proton-proton interaction point in space. Distinctions are made between *primary vertex*, *pile-up vertex* and *secondary vertex*, which denote interactions with large momentum transfer, low momentum transfer and decay of long lived particles respectively.

concessions are made in terms of precision. Later, the offline reconstruction must deliver high-precision tracks, which in theory means the code is not constrained to fit within a fixed run time per event. Nevertheless, the highly-increasing computational costs in supercomputers pose restrictive time constraints on the software. The computing grid used for particle physics applications is distributed globally and includes some of the top 500 supercomputers, with *Horeka* at Karlsruher Institut für Technologie (KIT) and *Summit* at Oak Ridge National Laboratory (ORNL) being just a few contributors to WLCG tiers 1 and 2 respectively.

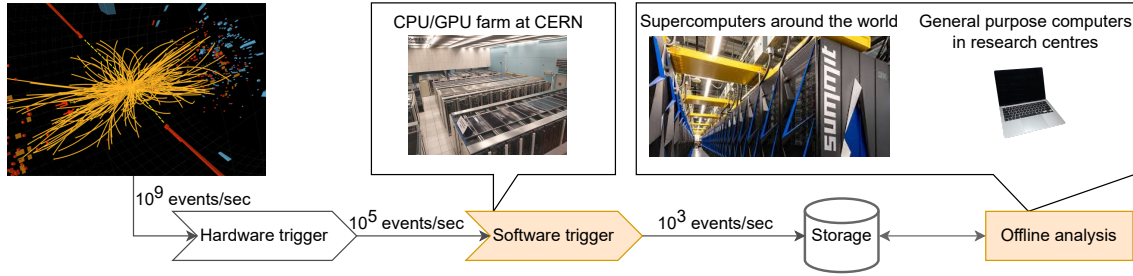


Figure 1.15.: Data acquisition and processing chain in ATLAS

Anticipating the increasing volumes of data at HL-LHC and taking into consideration the hardware developments in supercomputers, HEP Software Foundation (HSF) drafted a white paper to address the new computational challenges ahead [Albrecht et al., 2019]. One key recommendation was to explore heterogeneous architectures and in particular: the parallelism enabled by multicore CPU and manycore GPU. This effort started years ago, triggering software R&D projects within all the major experiments at CERN. Firstly, some of the software packages were upgraded to use multi-threading support. For ATLAS experiment, this meant the delivery of *AthenaMT*, the successor of the existing *Athena* framework, which brought thread-safe data structures, re-entrant and stateless algorithms, and the option to run algorithms in parallel [Leggett et al., 2017]. Similar developments were reported for CMSSW, CMS’s track reconstruction framework [Jones et al., 2015]. Secondly, the ability to target GPU platforms was prototyped by all major experiments: ALICE developed a GPU-accelerated track reconstruction in the High Level Trigger, which can run on NVIDIA and AMD GPU, and which facilitated important speedups translated into cost savings of 1.5 million Swiss francs [Rohr et al., 2017]; ATLAS ported a HEP Parameterized Calorimeter Simulation Code to NVIDIA and AMD GPUs [Dong et al., 2021]; CMS developed *Patatrack* workflow, which offloads parts of the computations to a GPU [Bocci et al., 2020]; LHCb wrote a heterogeneous framework named *Allen*, which accommodates both CPU and NVIDIA GPUs [Aaij et al., 2020].

These are only a few examples out of all the results that were published, while some of the key outcome of these developments are summarised below:

- Most of the projects delivered considerable wall-clock speedups with different degrees of portability (either between CPU and GPU or between GPUs from different

vendors). This can be used as a proof-of-concept to validate the assumption that accelerators can be successfully used for HEP code.

- There is a huge development effort to learn new programming languages and their associated extensions to target different architectures using native implementations²², which is followed by the need to maintain different repositories. This is exacerbated by the fact that reconstruction code is mostly written by physicists, who sometimes lack (a) advanced software development training and (b) the motivation to spend most of their time writing code instead of doing actual physics analysis. Therefore development productivity was not ideal.
- The reconstruction code is very complex and therefore the parallelization effort was not trivial as this often required rewrites of the event data flows or the algorithms. Also several parallelization strategies needed to be employed in order to achieve the desired speedup. For example, reading event data cannot be done on GPU yet so it must be covered on the CPU. Then inherently sequential algorithms like RKN or KF cannot be directly parallelized, therefore the parallelization needs to happen at an upper level (i.e. propagate different tracks in parallel). Many of the above-mentioned results were based on trial and error approaches.

While elaborating the white paper for defining the software strategy for the next 10 years, special consideration was given to some specific topics, with *Software trigger* and *Event reconstruction* being two of them [Albrecht et al., 2018]. The initial estimates for compute requirements are revised (and updated accordingly) every year. Figure 1.16a and Figure 1.16b are part of the *technical report* [ATLAS Collaboration, 2022b] published in March, 2022, and show that to deliver the expected amount of work while fitting into a realistic budget model requires serious investments in R&D²³.

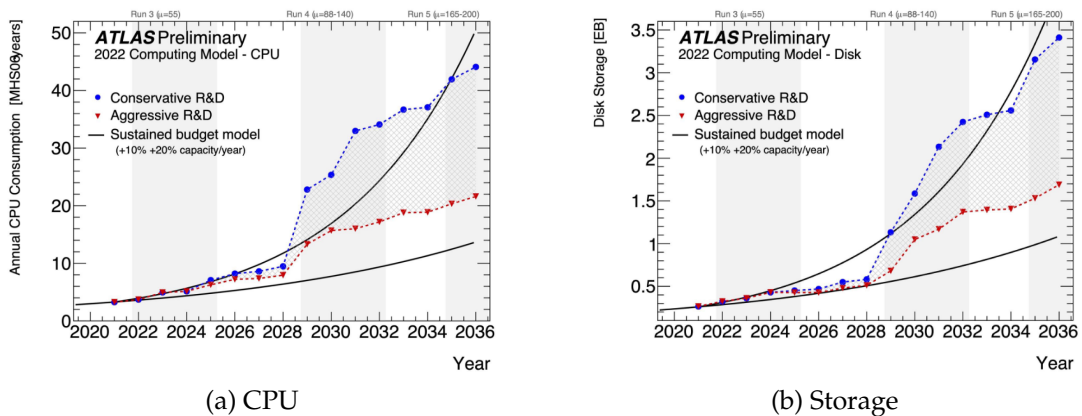


Figure 1.16.: ATLAS Computing Resource Estimates [ATLAS Collaboration, 2022b]

²²Examples detailed in the next chapters include C++ and OpenMP for CPU, and CUDA, OpenCL, HIP for GPU

²³A baseline model represents a state where R&D is kept to the minimum and even dropped altogether in some cases. The *conservative R&D* model assumes that both current person-power and technical expertise are maintained at the same level on the long term, while *aggressive R&D* assumes larger person-power, which can be achieved by either hiring new people or allocating more development time for the current collaborators.

As a consequence, several R&D projects prototyped experiment-independent track reconstruction using C++ algorithms wrapped in target-specific demonstrator chains (detailed in Section 3.1.1.1). Inline with this effort, the need for a multi-purpose framework that would allow a parallel and efficient execution of the algorithms while using only C++ code was identified. As the name suggests, the framework should contribute to improving performance, portability and productivity of the reconstruction software by providing a clear distinction between the scientific code (e.g. the physics-related algorithms) and the computational approach (e.g. multi-threading and/or offloading languages). This should alleviate the development effort by passing the responsibility for parallelization and/or GPU utilisation to a (hidden) backend maintained by computer scientists. While there will be a cost of porting current implementations to the new framework, this should be reduced to a minimum by providing C++ APIs and by allowing it to happen gradually. Moreover, the framework should also reduce the need to maintain several code repositories by allowing an algorithm implementation to be compiled for different architectures.

1.4. Contribution

The goal of this thesis is to study efficient ways of running track reconstruction code on supercomputing hardware, while maximizing developers' productivity. Our contribution is threefold. First, we explore state-of-the-art parallel approaches to get insights about the algorithms' characteristics and to setup a test bench for further evaluations; the outcome consists of parallel implementations in open-source physics applications and a series of performance studies. Second, we introduce *clang-offload*, a simple standalone prototype for C++ automatic source-to-source translation tool, that transforms sequential code to a parallel one and enables execution on different hardware. Third, we introduce the *vecpar framework*, our proposed solution designed to address the concerns expressed in the motivation section.

1.5. Thesis Structure

This thesis is structured as follows: Chapter 2 presents an overview of state-of-the-art parallel and distribute programming approaches in current HPC environments and introduces the main concepts and terminology of the field. Chapter 3 summarises our experimental work of (a) applying state-of-the-art libraries to particle physics examples extracted from the track reconstruction flow and (b) developing a trivial prototype tool; the results and observations define the functional and technical specifications for our proposed solution: the *vecpar framework*, which is detailed in Chapter 4. Chapter 5 covers related work and compares *vecpar* to existing approaches. Several metrics together with evaluation results obtained in selected experimental setups are presented in Chapter 6, while future developments are proposed in Chapter 7.

2. State of the Art

Dedicated programming languages and specific extensions were developed based on C/C++ language to ensure the best hardware exploitation through performance optimizations at all levels: application, compiler, driver, runtime and hardware. In this chapter, we summarise the state-of-the-art APIs and tools for parallelization and offloading targeting heterogeneous architectures, highlighting the key points that shaped our work.

2.1. Basic Concepts

It is important to note the difference between *concurrent* and *parallel* execution. The former refers to different actions that happen in the same time but not necessarily simultaneously; an example is a single-core CPU, which executes several processes at the same time; in practice, only one instruction (belonging to one of the processes) is executed at a time, but the scheduler mimics the parallelism by allocating a number of cycles when each process executes its code and then stops to give priority to the next one in line. The latter refers to dividing the work that can be done simultaneously; if the same code is executed on different chunks of data, we talk about *data parallelism*, while if the code is divided into smaller pieces, called *tasks*, which can be executed on the same or on different chunks of data, we talk about *task parallelism*.

As mentioned in Section 1.1, most of the current supercomputers use NUMA architectures composed of multiprocessors and accelerators within the same computing node. In this case, we talk about *distributed computing* or *offloading* rather than *parallelism* because data that resides in CPU memory must be made accessible to accelerator(s) at runtime, if computations are expected to happen on both devices¹. Memory transfers can be done *explicitly*, by invoking appropriate APIs or *implicitly*, by delegating this task to the compiler and driver for the GPU that offer the concept of *unified memory* or *shared memory*². The implicit way usually trades performance for usability.

Moreover, the memory capacity of a GPU is much smaller than the one of a CPU. Therefore, several constraints apply on both execution and data. First, there is typically no support for recursive function calls since the stack is limited. Second, memory cannot be allocated dynamically. Recently, different mechanisms that emulate the dynamic

¹CPU and GPU memories are often referred to *host* and *device* respectively, because a process usually starts on the host and then it can continue on the device, and not vice-versa.

²These terms can be misleading since they refer to a *common way of accessing memory* rather than a single shared physical card.

allocations were proposed and they will be discussed later in this chapter. Third, there is limited support for runtime object polymorphism and in some cases, virtual functions are not yet supported at all. Fourth, there is no support for C++ Standard Library functionality on the device (this includes both functions and containers).

2.2. C++ Standard Template Library

Traditionally, C++ has not offered dedicated support for parallelism or concurrency. However, in the last years, the C++ Standard Template Library (STL) has started to provide features in this direction as a part of an ongoing effort³ to setup a standardized approach to handle different memory resources of a NUMA system and different types of *hardware and logical threads* within the same application.

C++17 delivers *algorithm library*, which defines routines like searching, sorting and manipulating data sets and support for iterators. The notion of *execution policy* is defined for an algorithm and it expresses the way the operations are carried out. For example, *parallel policy* denoted by `std::execution::par` marks that the algorithm can be executed in parallel in a multi-threading setup while the *parallel unsequenced policy* denoted by `std::execution::par_seq` signals that the algorithm can be parallelized either through multi-threading or vectorization but the order of the results is not guaranteed. Important algorithms to mention in scope of this thesis are `for_each` and `transform`; the former applies a function on every iterator in a range while the latter has a similar functionality except for the fact that it returns the results stored in a different iterator. A trivial example is shown in Listing 2.1. Numeric operations like `reduce`, `inner product`, `accumulate` etc. provide overloaded implementations with execution policies to ensure the support for parallelism. Another important component is the new *polymorphic allocator*, which enables run-time polymorphism based on the memory resource that it is constructed with. Conceptually, this means that both CPU and GPU memory are handled similarly from the user code's perspective while the details concerning actual read/write operations are hidden through abstractions.

```

1  int a[N]; // initialize a ...
2  std::for_each(std::execution::par, std::begin(a), std::end(a),
3  [&](int& i) {
4      i++;
5  });

```

Listing 2.1: Increment the values from an array of size N in parallel using C++17

C++20 develops *algorithm library* further by adding *unsequenced execution policy*, which enables vectorization within the same thread. Additionally, *ranges library* is an extension and generalization of the algorithms and iterators exposed by *algorithm library*. With this

³Besides the input from the C++ community, there is also an important contribution from GPU vendors to the ISO C++.

new addition, the algorithms are now more easily *composable*, thus building pipelines becomes a trivial process. Furthermore, several extensions to the existing *concurrency support library* were introduced: new thread cancellation mechanisms, new synchronization objects like semaphores, latches and barriers, and support for atomic operations on non-atomic objects.

New features, meant for increasing the support for targeting heterogeneous architectures, were also made available. First, *concept library* is a collection of functions that support compile-time polymorphism. This has a critical importance for code compiled for GPU since this platform has limitations in handling polymorphism, as discussed previously. Second, *span* view is added to *container library*; it is a non-owning view over a contiguous sequence of objects, the storage of which is owned by some other object. Third, the memory resources can now support allocation/deallocation of raw aligned memory from the underlying resource; again, this can be a generic approach for any type of memory from a NUMA system if the compilers/drivers handle the particularities of the targeted platform.

While parallel policies proposed by ISO C++17 are already supported by most of compilers, new concepts like *thread pool executors* containing different types of resources (i.e. CPU and/or GPU threads) are proposed for C++23/C++26 and prototyped in some proprietary compilers [Larkin, 2022].

2.3. Multi-platform Standards and APIs

In this section, we investigate the standards and APIs driven by community effort, regardless whether they were designed having a heterogeneous model in mind, or they were initially used for shared-memory CPU architecture and adapted to include accelerators later on.

While an API might have several versions that cover more programming languages, we will focus only on C/C++ APIs since this is in scope for our domain application.

2.3.1. OpenMP

With the rise of the shared-memory parallel computers in the 1980s, there was the need for directives to let the compiler know how to execute the instructions in parallel on different nodes; while nowadays SMP are replaced by Massively Parallel Processor (MPP), the directives roles are more important than ever. Initially, the specifications were drafted by a group of hardware vendors in the 1990s and they were based on the work done by the *Parallel Computing Forum*, and more exactly a parallel loop for Fortran language [Chapman et al., 2008].

OpenMP API defines “a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer”⁴.

⁴<https://www.openmp.org/>

It is not a programming language, but a set of directives and utility functions, which instruct the compiler which instructions can be executed in parallel and how to handle the shared memory among threads. If used on supercomputer nodes with many cores and distinct memory locations, the compiler could receive information about how to (a) distribute threads to the available cores, (b) access memory spaces and (c) synchronize among workers. Each compiler that wants to support OpenMP standard must provide a custom implementation for the API.

First OpenMP specifications for C/C++ were published in 1998. In 2013, OpenMP added support for accelerators with the release of version 4.0. This was the point when the standard extended its portability to shared-memory CPU node and *heterogeneous (CPU and GPU) node*. The main C/C++ compilers fully support OpenMP 4.5 (2015) and partially support OpenMP 5.0 (2017), while the most recent standard specification is 5.2 (2021). Next, when mentioning OpenMP features we will be referring to 4.5 version unless stated otherwise.

The execution model of OpenMP is based on a *fork-join* mechanism: the initial thread executes sequentially until it reaches a *parallel region* when it creates a team of threads with the initial thread being the master. One (implicit⁵) *task* is created for each thread and one (implicit) *barrier* is set at the end of the parallel region to join the threads. One thread is bound to a specific place within the current *place partition*⁶ according to the default value of the *bind-var* internal control variable (initially set by the OpenMP implementation), but the distribution strategy is configurable through directives and environment variables. Similarly, many other execution-related configurations can be set in this way; a few examples include: the number of working threads and/or teams, the device to offload the computations to, the thread affinity policy for the nested regions, the visibility of a variable (shared, private, firstprivate), the reduction function that applies for the given loop, and the synchronization directives.

OpenMP uses a host-centric execution model when a GPU is required; this means that a thread starts executing on the host and it can offload the computations to a *target* device with the main thread expecting the task's completion on the device before exiting. While multiple threads can be started on both host and device, they cannot migrate from one platform to another.

The memory model is fairly straight-forward when only CPU are performing the execution since all the threads share the same address space. They also have private variables allocated in the thread's local memory. When GPU threads are required as well, the memory needs to be mapped between host and device unless *unified memory* is used. OpenMP 5.0 brings support for memory allocators (based on C++17 memory resource concepts) while OpenMP 5.2 extends it by adding dynamic allocators and unified memory allocators. Next, we briefly explain some the main OpenMP features that are in scope for the

⁵Additionally, OpenMP defines the API needed for the developer to setup tasks and barriers explicitly.

⁶The place partition defines the execution environment and contains information about how threads, cores and sockets could bind logically to OpenMP threads.

current work [van der Pas et al., 2017].

1. Work sharing

- Create a team of threads to execute the region

```
#pragma omp parallel [clause[ [,] clause] ... ] //new-line
    structured-block
```

- Create a league of teams with the initial thread in each team executing the region

```
#pragma omp teams [clause[ [,] clause] ... ] //new-line
    structured-bloc
```

- Specify the loops that are executed by the thread team

```
#pragma omp distribute [clause[ [,] clause] ... ] //new-line
    loop
```

- Specify the iterations that will be executed in parallel by threads from the configured teams

```
#pragma omp for [clause[ [,] clause] ... ] //new-line
    loop
```

2. Device execution directives

- Instruct the compiler that the following structured block can be executed on a device (if available)

```
#pragma omp target clause[ [ [,] clause] ... ] //new-line
    structured-bloc
```

- Allows the function or variable (declared or defined) after the directive to be executed on the device

```
#pragma omp declare target clause[ [,] clause] ... ] //new-line
    structured-bloc
```

3. Data mapping between host and device

- Map variables to a device environment; set map clause

```
#pragma omp target data clause[ [ [,] clause] ... ] //new-line
    structured-block
```

- Similar to the one above but divided into 2 explicit steps

```
#pragma omp target enter data [ clause[ [,] clause]... ] //new-line
...
#pragma omp target exit data [ clause[ [,] clause]... ] //new-line
```

- ### 4. Specialized execution⁷ – this is a run-time switch, which can choose a dedicated implementation for a specific architecture, sub-architecture or vendor.

⁷Available starting with OpenMP 5.0

- Specify multiple directive variants of which one may be conditionally selected to replace the metadirective based on the enclosing OpenMP context

```
#pragma omp metadirective [clause[ [,] clause] ... ] //new-line
    when(context-selector-specification: [directive-variant])
    default(directive-variant)
```

- Declare a specialized variant of a base function and specifies the context in which that specialised variant is used

```
#pragma omp declare variant(variant-func-id) clause //new-line
    match(context-selector-specification)
    function definition or declaration
```

The API allows that several features are combined, as shown in Listing 2.2.

```
1 // define the size N
2 int a[N], b[N], c[N];
3 // initialize vectors a,b,c ...
4 #pragma omp target teams distribute parallel for \
5     map(to:a[0:N], b[0:N]) map(from:c[0:N])
6 for (int i = 0; i < N; i++)
7     c[i] = a[i] + b[i];
```

Listing 2.2: Simple OpenMP example for adding together two vectors of size N

Also, it is important to note that an OpenMP program that contains *target* regions is compiled for all available⁸ platforms by default. Therefore the executable will run on a CPU as an automatic fallback mechanism if no accelerator is available at run-time. This behaviour can be configured through appropriate compilation flags.

To ensure the massive scale of parallelism that is required in supercomputers, OpenMP is usually used in conjunction with Message Passing Interface (MPI), which distributes the computations on different nodes to obtain a collaborative solution. MPI code is more difficult to implement and it usually requires the core to be rewritten in order to fit the paradigm, but it can exploit large-scale parallelism. On the contrary, OpenMP is easy to implement, can be done incrementally, but it was limited to shared-memory systems. There is also the hybrid approach (MPI + OpenMP), which exploits all levels of parallelism and decreases the need for network communication. A summary of this comparison is shown in Table 2.1. MPI is also extending its capabilities to support communications between GPU nodes besides the existing CPU ones, but this is beyond the scope of this thesis.

In summary, OpenMP has an important series of advantages. First, OpenMP 4.5 is supported by all C/C++ compilers [Huber et al., 2022] and can be enabled through `-fopenmp` flag. Second, there is a very active community, which contributes to its development including experts from hardware vendors who are involved in both (a) defining

⁸Available platforms are the ones which can be discovered by the compiler based on shared libraries on the build machine or the ones explicitly specified through compiler flags (like for example `-march`)

Criteria	OpenMP	MPI	Hybrid (OpenMP + MPI)
Address space	Shared	Distributed	Shared + Distributed
Implementation difficulty	Easy	Moderate complexity	Very complex
Incremental porting	Yes	No	No

Table 2.1.: OpenMP and MPI comparison

the standard further and (b) providing low level optimizations in vendor-proprietary and open-source compilers. Linear algebra libraries like for example *eigen* or *lapack* have also optimized their code to run safely and efficiently in a multi-threading environment using OpenMP [Anderson et al., 1999, Guennebaud et al., 2010]. Third, OpenMP offers a high degree of portability due to its vendor-agnostic approach, with the directives covering x86 CPU produced by all vendors, ARM CPU, GPU from NVIDIA and AMD, and FPGA. Fourth, from a single-source OpenMP code, one executable can be built to target several platforms.

2.3.2. OpenACC

*Open Accelerators*⁹ (OpenACC) is a programming model that uses high-level compiler directives to expose parallelism in the code with the help of compilers to target a variety of parallel accelerators. Launched in 2011, when there was no support for accelerators in existing parallelization standards like OpenMP, OpenACC was designed with the goal of closing this gap. Next, we will focus on features of version 3.1 released in 2020.

OpenACC defines an abstract model for accelerated computing with several levels of parallelism, similar to the OpenMP one: a coarse-grain one covered by *gangs*, a fine-grain level covered by *workers* and *vector* level for SIMD operations. For an offloading region, OpenACC might create one or more gangs, each with one or more workers, while each worker can have one or more vector lanes. Gang, worker and vector are the OpenACC naming for team, thread and SIMD concepts in OpenMP, although the execution model works a little bit differently: The gangs start executing in *gang-redundant* (GR) mode, which means that one worker in each gang executes the same vector lane. When a parallel region is encountered, the program executes in *gang-partition* (GP) mode, which partitions the iterations of a loop among gangs. At this point, there is still one worker per gang and one vector lane per worker. When an imbricated loop is encountered, the execution moves to *worker-partitioned* (WP) mode, which activates all threads inside a gang. Further, if a SIMD operation is required, the execution happens in *vector partitioned* (VP) mode, which means that all vector lanes of the worker are now active.

The memory model is handled through directives. Similar to OpenMP, the programmer is responsible for making sure that the data is accessible at run-time by explicitly mapping the memory between the devices.

⁹<https://www.openacc.org>

Some of the most common directives relevant for accelerator programming are summarized next:

1. Work sharing

- Create one or more gangs with one or more workers each

```
#pragma acc parallel [clause [[,] clause]...] //new-line
{ structured block }
```

- Distribute the computation to the appropriate parallelization layers as explained above; set gang / worker / vector clauses as needed

```
#pragma acc loop [clause [[,] clause]...] // new-line
loop
```

- Parallelizes the loops across gangs; it relies on the programmer to identify which loops are data-independent but leaves the decision on how to map the parallelism on the device to the compiler

```
#pragma acc parallel loop [clause [[,] clause]...] // new-line
loop
```

- Marks the loop that will be executed on the device

```
#pragma acc kernels [clause [[,] clause]...] // new-line
{ structured block }
```

2. Data mapping between host and device

- Maps a list of objects from host memory to device memory by specifying copy / copy_in / copy_out clauses

```
#pragma acc data [clause[[,] clause]...] //new-line
{ structured block }
```

- Similar data mapping but divided into two explicit steps

```
#pragma acc enter data [clause[[,] clause]...] //new-line
...
#pragma acc exit data [clause[[,] clause]...] //new-line
```

- ### 3. Specialized execution – the device_type clause can be passed to `parallel` or `kernel` directives to target a specific platform (e.g. `acc_device_nvidia` or `acc_device_radeon`).

In summary, OpenACC has a few strong points. First, it is supported by some of the C++ compilers but not all; currently, LLVM/clang [Fandrey, 2010] is working on building a front-end to compile ACC based on OpenMP backend and runtime. The compilers that support OpenACC usually enable it through the `-fopenacc` flag. Second, it offers some degree of portability, currently covering x86 architecture, and GPU from NVIDIA and AMD vendors. Nevertheless, using a CPU as an offloading device is currently supported only by vendor-proprietary compilers (e.g. NVIDIA HPC SKD compiler), while general C++ compilers like gcc (v.12) are still lacking this ability.

2.3.3. OpenCL

Open Computing Language(OpenCL) is “an open, royalty-free standard for cross-platform, parallel programming of diverse accelerators found in supercomputers, cloud servers, personal computers, mobile devices and embedded platforms”¹⁰. It was released in 2009 and the most recent version is 3.0 (2022).

The execution model of OpenCL involves one host and one or more devices. Each device can have one or more *compute units (CU)* with each CU having one or more *processing elements (PE)*. The host submits a kernel to the device and it is executed by CU and PE. If all the processing elements of a compute unit execute the same instruction then an efficient *converged work flow* is achieved, as opposite to a *diverged* one.

The computations offloaded to the accelerator inside a kernel is a *work-item*. A collection of work-items that are executed on a single compute unit form a *work-group*. The host interacts with the device(s) through *command queues*, which instruct the runtime to perform particular operations like registering a kernel for execution, transferring memory from host to device or vice-versa or setting explicit synchronization points. An entity that holds commands and work-groups from kernel-instances that are ready to execute form a *work-pool*. There is one work-pool associated with each device.

The OpenCL execution model defines three types of kernels. First, the *OpenCL kernels* are built and managed by OpenCL API; second, *native kernels* are accessed through host function pointers; the semantics is implementation specific and their support is optional; third, *built-in kernels* are specific to a particular device; they are not built from source-code and are usually used to expose hardware or middleware functionality specific to a device.

The OpenCL memory model gives explicit access to the regions of an accelerator memory. All work-items have read/write rights to *global memory* in a given context. The *constant memory* can only be allocated and instantiated by the host and the work-items have read-only rights to access it. All work-items in a work-group share the *local memory* while each of them has access to a *private memory*, which cannot be shared.

To compile OpenCL code, a specific compiler is needed since most of the C/C++ compilers cannot build OpenCL code. Nevertheless, the major advantage of OpenCL is its portability; it can currently target x86, ARM and GPU from NVIDIA, AMD and Intel.

2.3.4. SYCL

SYCL¹¹ is a C++ programming model for OpenCL based on C++17 and developed by Khronos. Similar to OpenMP and OpenACC, this is not just an API but it includes also a runtime. On the other hand, similar to OpenCL, SYCL requires a dedicated compiler. Initially proposed in 2014, SYCL has developed significantly over the years. In this section we will focus on the features of SYCL 2020, released in 2021.

¹⁰<https://www.khronos.org/opencv/>

¹¹<https://www.khronos.org/sycl/>

As a programming model designed for heterogeneous resources, SYCL provides abstractions for both memory handling and parallel computing scheduling, all controlled by the SYCL runtime. Unlike the previous APIs, SYCL has no host-device model, since one SYCL interface might have multiple backends.

The memory model is based on *buffer/accessor model* or on Unified Shared Memory (USM). For the former, a *buffer* is defined in host code, which is the owner of the data while *the accessor* requests access to data for different valid operations¹². For the latter, GPU's driver and runtime handle the shared pointers, so no further API calls are required.

The execution model is based on commands submitted to a queue, which is associated with a device. The commands can require data copy, kernel execution, synchronization, etc. Commands can be aggregated in *command groups*, which can later be composed together and submitted asynchronously to the scheduler; this can decide to either trigger an execution or to wait for appropriate preconditions. Nevertheless a command group can have only one kernel invocation as named or unnamed lambda function.

SYCL defines *nd-range* as n-dimensional layout of the iteration space, which handles the distribution of tasks per threads or group of threads. Different global, group and local identifiers are accessible within each thread, while the associated memory locations are read/written at different speeds and latencies.

While the SYCL API is very rich and complex, we mention the main entry point function for device execution: `parallel_for`. There are several overloaded implementations, templated on data types, kernel name, nd-ranges, etc.

A single-source program requires two compilation passes to produce an executable, as shown in Figure 2.1. The dedicated compiler produces either intermediate device

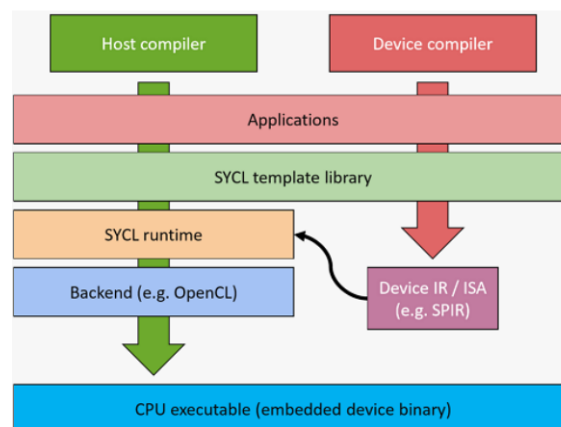


Figure 2.1.: SYCL compilation model [Codeplay Software, 2022]

representation (IR), like for example the Standard Portable Intermediate Representation (SPIR) used in OpenCL environment, or Instruction Set Architecture (ISA), for example, Parallel Thread Execution (PTX), which is the instruction set for NVIDIA hardware. To

¹²Some of the valid operations include read, write, read_write, discard_write, discard_read_write, atomic

build the executable, it requires a *host compiler* that handles the compilation for CPU architectures and links the appropriate dependencies.

At the time of this thesis, there are several SYCL compilers available as shown in Figure 2.2: *dpcpp* is the Intel’s compiler, *hipSYCL* is the AMD alternative, *computeCpp* is the compiler developed by Codeplay Software.

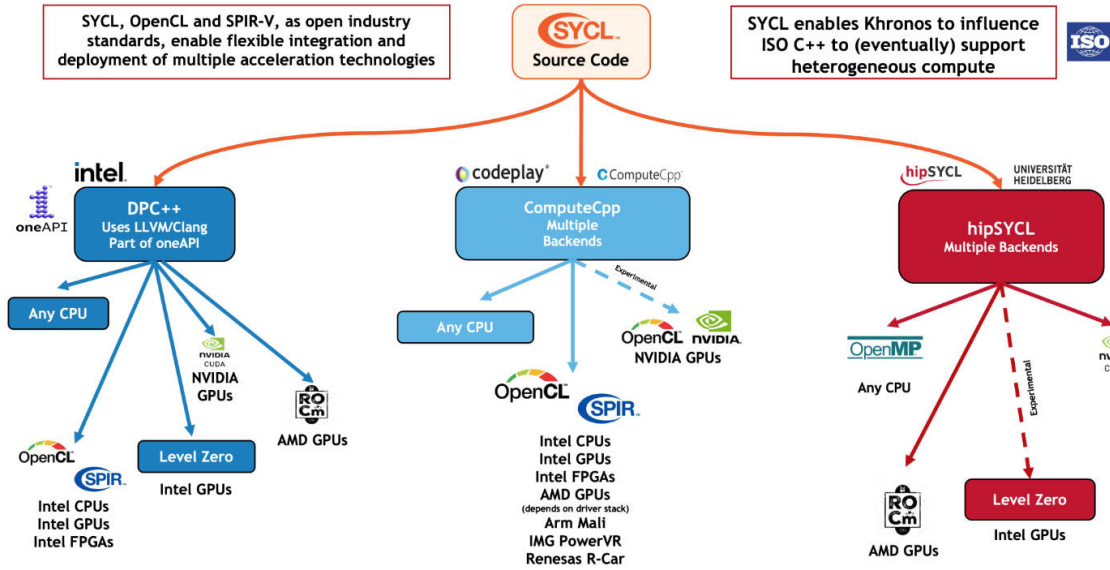


Figure 2.2.: SYCL supported backends and compilers [Khronos Group, 2022]

To summarize, SYCL’s strong point is the portability that it ensures. Currently, SYCL code can target any CPU, GPU from all major vendors and Intel FPGA. Moreover, SYCL oneAPI DPC++ compiler supports targeting multiple architectures in one executable. The main drawbacks are: the major open-source C++ compilers cannot build SYCL code yet and the verbosity of SYCL heterogeneous code.

2.4. Vendor-Proprietary Frameworks

In this section, we describe the heterogeneous frameworks proposed by the major hardware vendors: NVIDIA, AMD and Intel. These include language extensions, run-times and dedicated compilers, all being highly optimized for the targeted devices.

2.4.1. NVIDIA CUDA

NVIDIA corporation has released seven GPU architectures: Fermi, Kepler, Maxwell, Pascal, Volta, Turing and Ampere, with each of these having a particular hardware layout characterised by different features: number of compute units for single and double precision, memory size, connectivity options, the existence of tensor cores, etc. To exploit this knowledge, the generated binary code is architecture-specific, yet *backward compatible*¹³

¹³Support for Fermi, Kepler and Maxwell was dropped in the latest releases.

but with a lower degree of efficiency. The CUDA Toolkit provides a set of tools needed to write parallel code on NVIDIA GPUs: compilers, libraries, profiling and optimization tools, and last but not least, runtime support. CUDA HPC Software Development Kit (SDK) is the complete set of tools for developers [NVIDIA Corporation, c], as shown in Figure 2.3. The ones meaningful for the present work are described next.

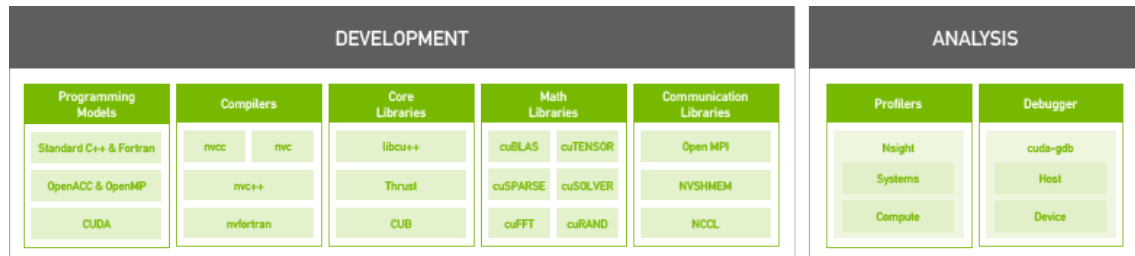


Figure 2.3.: NVIDIA SDK [NVIDIA Corporation, c]

Compute Unified Device Architecture (CUDA) is “the parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units” [NVIDIA Corporation, b]. Initially launched in 2007, CUDA is a C language extension, which has had eleven main releases and many updates throughout the years. The features described in this section focus on version 11.5 (released in 2021).

The execution model exposes several levels of parallelism; more *threads* are grouped in a *block*, while both of them can have a 1D, 2D or 3D layout over the hardware. Figure 2.4 shows an example of 1D threads and 2D blocks, allocated on a GPU with 4 Streaming Multiprocessors (SM); this means that 4 blocks can be executed simultaneously. More recently, several GPUs can be connected by high-throughput connectors, which ensure an extra layer of coarse-grained parallelism.

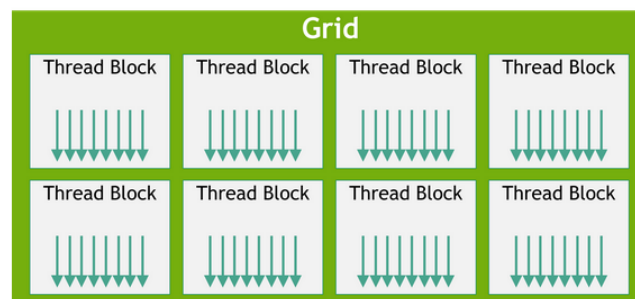


Figure 2.4.: CUDA execution model [NVIDIA Corporation, a]

Inside a block, the threads are grouped in *warps*; these are groups of 32 threads that are executed in parallel or in *lockstep*¹⁴. When not all the threads execute the same instruction, as it is the case of *if-then-else* statement for example, the warp *diverges*, which means that the threads are divided into two sub-groups, one for each branch, which execute

¹⁴The same instruction is executed by all the threads; it loads distinct operands from different memory locations, ideally within the same or adjacent cache line(s) to increase performance

sequentially. This is highly inefficient not only due to this induced serialization but also from the memory perspective because it no longer deals with consecutive addresses, therefore triggering unnecessary reads/writes in caches. While threads inside a block can be synchronized, each block is independent; regardless, the device can be synchronized, which means that the barrier waits for the completion of all threads (from all blocks).

Similar to all the other execution models that involve GPUs, the data needs to be copied over PCIe, from host to device memory card, before the computations can start. For NVIDIA devices, improved performance can be obtained by overlapping data transfers and execution using *streams*, which are the logical communication channel with the device¹⁵. Another approach to boost performance is to share the memory among threads within the same block if user data is aligned and if the algorithms require multiple reads from the same memory locations. Similar, improving the algorithm's data access patterns to ensure cache coherence can also bring important speedups.

A selection of the most relevant routines from CUDA C API is listed below, aggregated by functionality. `__host__` and `__device__` specify the context in which a function can be invoked.

1. Memory allocation of size bytes in device's global memory, as page-locked memory on the host and memory handled by the Unify Memory system respectively

```
__host__ __device__ cudaError_t cudaMalloc(void** devPtr, size_t size);

__host__ cudaError_t cudaMallocHost(void** ptr, size_t size);

__host__ cudaError_t cudaMallocManaged(void** devPtr, size_t size,
    unsigned int flags = cudaMemAttachGlobal);
```

2. Deallocate memory from device or page-locked memory on the host

```
__host__ __device__ cudaError_t cudaFree(void* devPtr);

__host__ cudaError_t cudaFreeHost(void* ptr);
```

3. Transfer the memory to/from device either synchronously or asynchronously

```
__host__ cudaError_t cudaMemcpy(void* dst, const void* src, size_t count,
    cudaMemcpyKind kind);

__host__ __device__ cudaError_t cudaMemcpyAsync(void* dst, const void*
    src, size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0);
```

4. Launch a device function and wait for the results

```
__host__ cudaError_t cudaLaunchKernel(const void* func, dim3 gridDim,
    dim3 blockDim, void** args, size_t sharedMem, cudaStream_t stream);

__host__ __device__ cudaError_t cudaDeviceSynchronize(void);
```

¹⁵Choosing the appropriate number of parallel streams that would hide the latency and therefore minimise the total wall clock time, depends both on the actual hardware and the problem's characteristics.

To compile CUDA code together with C sources, the `nvcc` compiler is provided. It handles the CUDA code while delegating the C sources to a so called *host compiler*. A fat binary is produced in the end by linking together the object files with different instruction sets.

*Basic Linear Algebra Subroutines (cuBLAS)*¹⁶ provides hardware accelerated implementations for the main mathematic operations using mixed and low precision. It is highly optimized for NVIDIA hardware and provides an extension for distributing computations on several GPUs in parallel. While cuBLAS is available in both Toolkit and HPC SDK, an advanced library targeting high-end GPUs is *cuTENSOR*¹⁷, which provides routines for exploiting the tensor cores available in HPC GPU.

Since the C++ STL is not portable to NVIDIA hardware, some of the parallelization features are supported through *libc++*¹⁸, which implements a heterogeneous implementation for utility functions and thread synchronization mechanism across devices. *Thrust library*¹⁹ extends the C++ STL support for NVIDIA devices, in terms of memory resources and models, parallel policies, executors, etc; while it was heavily extended in the last few years to cover most of the parallel operations, it lacks portability among GPU architectures since it only targets NVIDIA devices.

Recently, a different approach to heterogeneous C++20 standard has been developed with the release of *stdpar library* and its associated compiler, *nvc++*²⁰. It builds upon the parallel concepts defined in Section 2.2; the solution provides a pool of CUDA threads or OpenMP threads as executors for the parallel policies. Unlike CUDA language extension that required the code to adhere to NVIDIA abstractions, *stdpar* enables the user to use plain C/C++ code. No need to mark functions `__host__` / `__device__` if they are in the same compilation unit as the calling function; moreover, offloaded code in parallel loops is wrapped in lambda objects and passed on the GPU automatically. The memory is handled as unified memory if the allocations are done on the heap and in the same compilation unit. As all these facilitate the development productivity by removing the need to (a) learn CUDA C, (b) explicitly handle memory copies between host and device, and (c) maintain different repositories for CPU and GPU targets, there are a few drawbacks too. First, there is an impact on performance due to the usage of managed memory and therefore the performance does not match CUDA's yet. Second, the compilation: the GPU code needs to be compiled with *nvc++*, so that the functions are marked appropriately at compile time and hence using external libraries in device code becomes more complicated. Also, there is still poor support for *cmake* integration since both *nvc++* and *gcc* compilers are involved in compiling several pieces of the code. Third, while the result is a single-source heterogeneous code, the decision on which platform to target is done through

¹⁶<https://docs.nvidia.com/cuda/cublas/index.html>
¹⁷<https://docs.nvidia.com/cuda/cutensor/index.html>¹⁸<https://nvidia.github.io/libcudacxx/>¹⁹<https://docs.nvidia.com/cuda/thrust/index.html>²⁰https://docs.nvidia.com/hpc-sdk/archive/20.7/pdf/hpc207c++_par_alg.pdf

compilation flags, and therefore is currently no option to mark that some of the `stdpar` functions should be executed parallel on CPU while others should use the GPU. Fourth, the code, which is not open-source, is being actively developed with new compiler releases that add new functionality and address bugs happening almost monthly.

`Nvc++` is the CUDA C++ compiler provided by the HPC SDK. Beside compiling `stdpar` and CUDA code, it can also support OpenMP and OpenACC offload from C/C++ sources. The HPC SDK provides insightful profiling tools, like `nsight-systems`²¹ and `nsight-compute`²², which were extensively used in our research.

Looking at the future HPC architectures, NVIDIA has recently introduced their state-of-the-art heterogeneous superchip: *Grace Hopper*²³. As shown in Figure 2.5, it features a 72-cores ARM based CPU with 4 128-bit SIMD units per core, a high-performance GPU with 3x higher FP32 and FP64 throuput than the A100, a hardware-coherent interconnect between the CPU and GPU that enables Hopper GPU to address all Grace CPU memory (up to 608 GB RAM) and a link switch that connects up to 256 Grace Hopper superchips together. To program it, `std::par`, CUDA, OpenMP and OpenACC are a few supported platforms, together with the NVIDIA CUDA LLVM compiler.

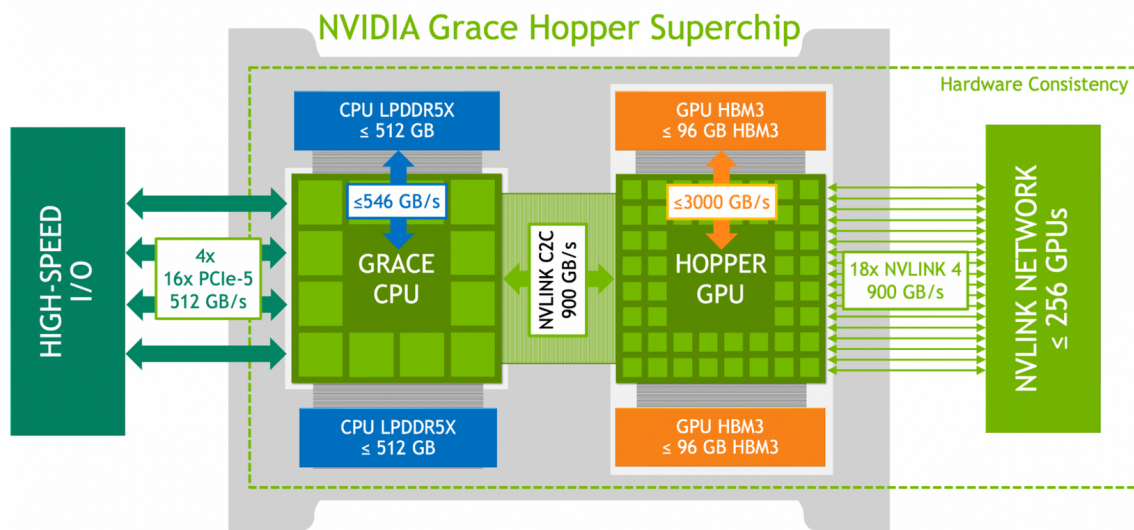


Figure 2.5.: NVIDIA Grace Hopper Architecture [Evans et al., 2022]

In summary, CUDA has a very important advantage: it guarantees the best performance out of NVIDIA devices. With *thrust*, *libc++*, *stdpar* and *nvc++*, NVIDIA contributes to the implementation of the C++20 parallel standard, opening the way for heterogeneous approaches. Nevertheless, currently, CUDA has a limited portability on heterogeneous devices as it can target x86 and ARM CPU and NVIDIA GPUs.

²¹<https://developer.nvidia.com/nsight-systems>

²²<https://developer.nvidia.com/nsight-compute>

²³<https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper>

2.4.2. AMD HIP

AMD is one of the leading vendors on the CPU market that has recently achieved outstanding performance with their GPU lines as well. They provide two main GPU architectures: Radeon-DNA (RDNA) and Compute-DNA (CDNA), as shown in Figure 2.6. The former is optimized for gaming to deliver high number of frames per second; to achieve this, the cards have hardware accelerated support for rasterization, tessellation, graphic caches, blending operations and even display engines. The latter is optimized for data centres with intense compute operations with the end goal of maximizing FLOP/s; it has less graphic capabilities but more compute units, wider registers, hierarchical memory structures with multiple layers of caches and fast AMD Infinity Fabric links.

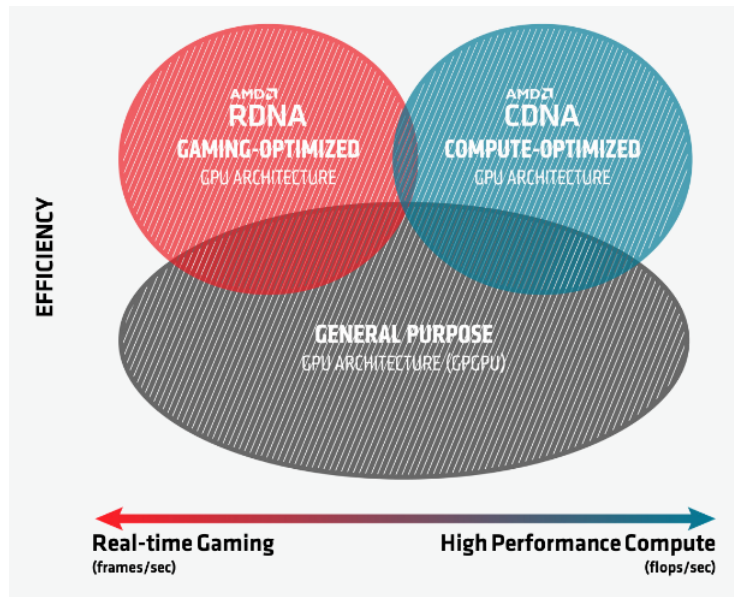


Figure 2.6.: AMD GPU Architectures [AMD, 2022a]

While several generations of RDNA and CDNA were made available, we would like to mention two GPU models relevant for the current work: firstly, the AMD Radeon RX 6600, which belongs to RDNA 2 and delivers up to 8.93 TFLOP/s single-precision performance at low power consumption²⁴ (132 W), and secondly, the AMD Instinct MI250 found in top supercomputers mentioned in Section 1.1.2, which belongs to the CDNA 2 architecture; this enables more GPUs tightly connected on the same chip, following a ccNUMA layout, to produce a Graphics Compute Die (GCD), which delivers 47.9 TFLOP/s peak double-precision vector FP64 throughput²⁵.

Radeon Open eCcosystem (ROCm) is the AMD open-source software development platform, which includes libraries (linear algebra, Fourier transforms, random number generators, etc), compilers and profiler, debugger and other tools, as shown in Figure 2.7. In this section, we discuss version 5.0.

²⁴<https://www.amd.com/en/products/graphics/amd-radeon-rx-6600>

²⁵<https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf>

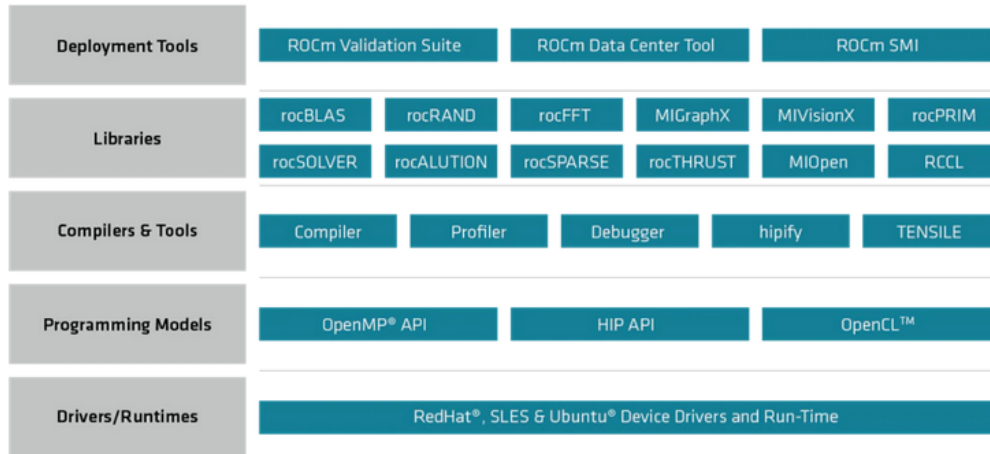


Figure 2.7.: AMD ROCm Platform [AMD, 2022b]

The execution model is similar to NVIDIA GPU, with some small differences in terms of naming: more *threads* are grouped in *blocks*, which are then mapped to a *grid* of 1D, 2D, 3D dimension. A *wavefront*²⁶ is a group of threads from the same block that are executed in lockstep; the only technical difference is that AMD uses 64 threads in a wavefront, while NVIDIA usually keeps the warp size at 32. For execution, each block is mapped to a *Compute Unit*²⁷, whereas the number of compute units gives a measure of the coarse-grain parallelism.

Regarding the memory model, ROCm enables two coherency options for host memory: *coherent memory* implies that an atomic operation that alters GPU memory is immediately visible to the CPU or to other GPU peers with the price of no caching mechanism, and *non-coherent memory*, which can be cached but cannot support synchronization while the kernel is running. The former is recommended for fined-grained parallelism while the latter is optimal for high-performance access when no fined-grained synchronization is required.

Heterogeneous-Compute Interface for Portability (HIP) is the AMD GPU programming environment; it contains a C++ language extension and an associated runtime, which allows creating portable applications. A selection of representative routines is presented below to demonstrate the similarity with the CUDA API described in the previous section.

1. Memory allocation of size bytes in device's global memory, as page-locked memory on the host and memory handled by the Unify Memory system respectively

```
hipError_t hipMalloc(void** devPtr, size_t size);
```

```
hipError_t hipMallocHost(void** ptr, size_t size);
```

²⁶A wavefront is equivalent to NVIDIA's warp

²⁷A Compute Unit is the equivalent to NVIDIA's Streaming Multiprocessor

```
hipError_t hipMallocManaged(void** devPtr, size_t size, unsigned int
    flags = hipMemAttachGlobal);
```

2. Deallocate memory from device or page-locked memory on the host

```
hipError_t hipFree(void* devPtr);
```

```
hipError_t hipFreeHost(void* ptr);
```

3. Transfer the memory to/from device either synchronously or asynchronously

```
hipError_t hipMemcpy(void* dst, const void* src, size_t count,
    hipMemcpyKind kind);
```

```
hipError_t hipMemcpyAsync(void* dst, const void* src, size_t count,
    hipMemcpyKind kind, hipStream_t stream = 0);
```

4. Launch a device function and wait for the results

```
hipError_t hipLaunchKernel(const void* func, dim3 gridDim, dim3 blockDim,
    void** args, size_t sharedMem, hipStream_t stream);
```

```
hipError_t hipDeviceSynchronize(void);
```

Similar to CUDA, hardware accelerated and highly optimized mathematical libraries are provided; a few examples include *rocBLAS*²⁸ a basic linear algebra routines package implemented in C++14 and HIP (over ROCm runtime) and *rocSPARSE*²⁹ the linear algebra routines for sparse matrices.

Besides mirroring the CUDA API, HIP offers support for automatic source-to-source translation from CUDA to HIP using a so called *HIPify operation*. This uses OpenCL to implement functionalities that cannot be translated in a straight-forward way. Nevertheless, if the users still want to use CUDA code, the `hipcc` compiler can act as a wrapper: it identifies the CUDA code and delegates it to `nvcc` while using the `hip-clang` to compile HIP code. OpenMP and OpenCL are also supported on AMD GPU, but a dedicated compiler³⁰ is required to leverage this feature.

In summary, HIP has two strong points: first, the language extension, compiler and runtime ensure optimized performance on AMD GPUs; second, it offers an increased level of portability on different GPU models. Currently, HIP can target x86 and ARM CPU, and NVIDIA and AMD GPUs. Nevertheless, while development productivity can be increased when using HIP in comparison to CUDA due to its portability, it might also be negatively influenced by the fact that AMD tools are not mature enough yet³¹.

²⁸<https://roclblas.readthedocs.io/en/rocm-5.3.0/>

²⁹<https://rocspare.readthedocs.io/en/rocm-5.3.0/>

³⁰AOMP compiler is part of ROCm toolbox

³¹From our experience, most of the ROCm tools are difficult to install on both clusters and development stations/laptops, with limited package manager support and many driver/operating system/libraries incompatibilities

2.4.3. Intel DPC++

In 2020, Intel launched *oneAPI* as their open-source, unified programming model for heterogeneous compute environments, which include CPU, GPU, FPGA and other accelerators, as shown in Figure 2.8. Distinct toolkits for developing applications that target diverse hardware were made public since then, with *oneAPI Base*³² and *HPC*³³ toolkits being the equivalent of NVIDIA ones³⁴, and consequently, in our focus. These contain Data Parallel C++ (DPC++) programming language, libraries, compilers, runtimes, profilers and migration tools from CUDA to DPC++. In addition, dedicated frameworks for Artificial Intelligence Analytics, Deep learning, Internet-of-Things and Visualizations are also made available to users.

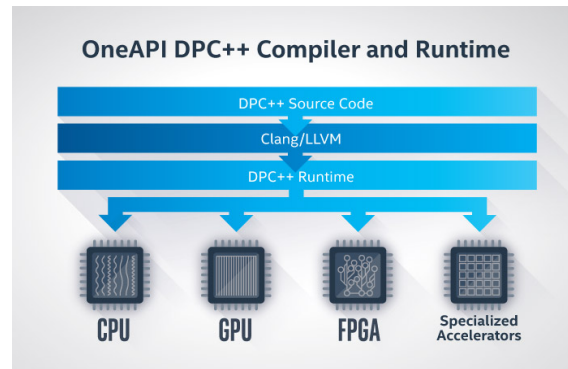


Figure 2.8.: Intel oneAPI [Intel Corporation, 2022]

DPC++ Language is based on C++17 and SYCL (see Section 2.3.4). The API provides implementations for parallel and vectorized execution policies for both host and device, together with iterators, algorithms and utility classes, following the STL definition described in Section 2.2. To achieve the heterogeneous behaviour, there are two preconditions that have to be met: first, use a compiler with OpenMP 4.5 support or add a dependency to Threading Building Blocks³⁵ (TBB) library to cover the host parallelism, and second, use a compiler with support for SYCL 2020 to cover the device parallelism.

Intel oneAPI DPC++/C++ Compiler satisfies the above requirements and is available in the Base toolkit. Moreover it has experimental support for NVIDIA CUDA and AMD HIP backends for a limited ranges of operating systems and devices.

In summary, Intel’s approach to heterogeneous computing has the following advantages: first, it enables developing single-source code that can be compiled for different instruction sets to target all Intel hardware (CPU, GPU, FPGA), ensuring a series of performance optimizations; second, it delivers a high degree of portability for other vendors’ accelerators by leveraging SYCL’s portability.

³²<https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html>

³³<https://www.intel.com/content/www/us/en/developer/tools/oneapi/hpc-toolkit.html>

³⁴I.e. the CUDA Toolkit and the CUDA SDK Toolkit

³⁵<https://spec.oneapi.io/versions/latest/elements/oneTBB/source/intro.html>

2.5. Compilers for Heterogeneous Programming

In this section, we provide an overview of the compiler support for targeting heterogeneous architectures, the interoperability between parallel APIs and different compilation chains and we focus on outstanding features exposed by some of the compilers.

*Gnu Compiler Collection*³⁶ (gcc) and g++ are a set of open-source tools, which contain front end for compiling C/C++ sources and provide implementation for different standards in the form of libraries, for example, C++ STL. Due to more than 30 years of development, gcc is one of the most robust and mature compilers, actively supported by the C++ community from both academia and industry.

*Low Level Virtual Machine*³⁷ (LLVM) is a distinct set of compiler toolchains, which can be used to develop front ends for any programming language and back ends for any *instruction set architecture* (ISA); its flexibility stems from the usage of a language-independent *intermediate representation* (IR), which is generated by the front ends³⁸ and serves as a portable high-level assembly language, which can be then converted into specific ISAs [Lattner, 2002].

Both gcc and clang support compilation for heterogeneous architecture by implementing OpenMP and OpenACC standards. Additionally, clang supports native CUDA code compilation to nvptx, while front ends for SYCL and HIP are currently discussed³⁹. Table 2.2 summarises the available support for all previously mentioned APIs in main compilers; it shows that currently OpenMP is the only parallel standard implemented by all major compilers.

Compiler	Parallel C++	OpenMP	OpenACC	OpenCL	SYCL	CUDA	HIP	DPC++
nvc++	Yes	Yes	Yes	No	No	Yes	No	No
hipcc	No	Yes ^I	No	No	No	Yes ^I	Yes	No
dpcpp	Yes	Yes	No	Yes	Yes	Yes ^E	Yes ^E	Yes
clang	Yes ^L	Yes	Yes	Yes	No	Yes	No ^S	No ^S
gcc	Yes ^L	Yes	Yes	Yes	No	No	No	No
cce	No	Yes	Yes	No	No	No	No	No

Table 2.2.: Heterogeneous API support in main compilers; *E* stands for *Experimental*; *I* stands for *Indirect* and means that the compiler cannot do the operation but it can delegate to an appropriate compiler, which can handle the compilation; *S* stands for *Scheduled*; *L* marks *Limited* support for CPU parallelism only

We would like to underline the support for ISO C++ parallel algorithms and execution policies mentioned in Section 2.2, since this is relevant for the present work. The parallel features are supported by gcc only for CPU through Intel TBB library. Similarly, LLVM/clang also currently provides only CPU support, but through a cross-platform im-

³⁶<https://gcc.gnu.org/>

³⁷<https://llvm.org/>

³⁸LLVM's C and C++ front ends are *clang* and *clang++* respectively.

³⁹Efforts are currently in progress to support the integration of SPIR with LLVM IR and therefore to allow clang to support SYCL compilation eventually.

plementation with Intel TBB, OpenMP and Apple Grand Central Dispatch (GCD) back-ends [Lin et al., 2022]. Intel compiler is a private fork from the LLVM project, therefore it inherits the same support level; nevertheless, Intel extends the parallel executors with a SYCL backend and therefore ensures the ability to run on Intel GPUs as well. To summarize, the same C++ code (based on ISO C++ parallel policies) can be compiled with different compilers to ensure portability and represents a distinct approach to *single-source* heterogeneous programming.

Table 2.2 also shows that LLVM/clang is currently the most versatile compiler for single-source code targeting different platforms. This is also confirmed by the fact that in 2022, *Frontier* offers four compilers: gcc, LLVM, cce (HPE-Cray) and hipcc (AMD ROCm), the last two being LLVM-based. Therefore, LLVM's impact in HPC community is significant.

Focusing on compilation for heterogeneous resources, LLVM version 15.0.0 offers a series of features that are not (yet) available in other compilers. First, improved interoperability between OpenMP target and CUDA code; for example, it is now possible to call a CUDA function within a target region. Second, special OpenMP target optimizations in dedicated LLVM passes ensure improved wall-clock execution time for the applications. Third, assumptions and remarks provided to the OpenMP runtime at compile time can bring important performance benefits even further due to tailored optimizations that can be enabled. Fourth, compilation for several GPU types (even of different ISA like AMD and NVIDIA) within the same executable. Fifth, debugging options like using a virtual GPU [Patel et al., 2021, Doerfert et al., 2021].

Moreover, recent successful prototypes for remote GPU offloading demonstrate the potential for massive parallelism [Lu et al., 2022]. On-going research on *OpenMP as target independent intermediate layer* aims to convert device-specific code (e.g. CUDA) to OpenMP and therefore make it portable so that it can run on GPUs from any vendors [Doerfert et al., 2023], as shown in Figure 2.9.

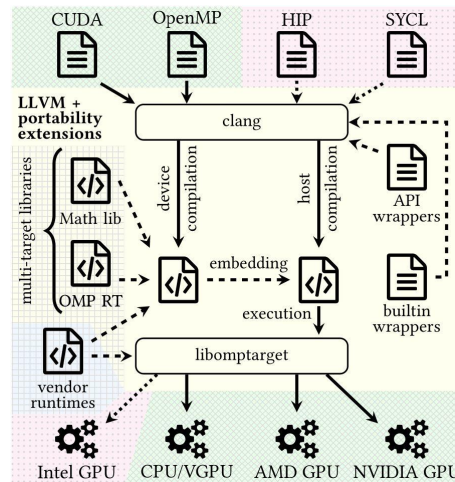


Figure 2.9.: OpenMP as target independent runtime layer; *green* color marks supported platform while *red* marks the unsupported ones [Doerfert et al., 2023]

In conclusion, there are many libraries, standards and frameworks that allow developers to express parallel code in different ways, using distinct approaches to target heterogeneous architectures. While some of them might be interoperable (e.g. CUDA and OpenACC) or interchangeable (e.g. CUDA code can be converted into HIP code), most of them are not compatible. Additionally, the C++ compilers are trying to accommodate the heterogeneous computing model by supporting these APIs at different levels.

The present work is based on features of OpenMP 4.5/5.0 and CUDA 11. While this leverages the advantages of the clang compiler, the code is not restricted to using it.

2.6. Functional Programming in HPC

The scientific community had been relying on procedural programming languages like C and Fortran for several decades, whereas many branches of science (including High Energy Physics) started moving towards object oriented languages like C++ and Python, while maintaining the same imperative paradigm.

With the rise of data science and machine learning, newer languages that combine imperative and declarative approaches like R⁴⁰, Julia⁴¹ or Scala⁴² are becoming more popular due to their ease-of-use, portability and availability in big data centers; they provide built-in parallel routines for numerical computations and hide this complexity from the user. Along with that, functional concepts like *lambdas*, *higher-order functions* or *immutability* started to make their way into C++ and Python in the last ten years. For C++, some of these were briefly mentioned in Section 2.2. Moreover, the *pipe* operator in C++20 is the implementation of the higher-order function concept. Also, the `std::ranges` and `std::functional` namespaces provide support for using functions as first-class citizens of the language, as shown in Listing 2.3⁴³.

```

1 #include <iostream>
2 #include <ranges>
3 int main() {
4     auto const ints = {0, 1, 2, 3, 4, 5};
5     auto even = [](int i) { return 0 == i % 2; };
6     auto square = [](int i) { return i * i; };
7     // the "pipe" syntax of composing the views:
8     for (int i : ints | std::views::filter(even) | std::views::transform(square))
9         std::cout << i << ' ';
10    // a traditional "functional" composing syntax:
11    for (int i : std::views::transform(std::views::filter(ints, even), square))
12        std::cout << i << ' ';
13 }
```

Listing 2.3: C++20 function composition example

⁴⁰<https://www.r-project.org>

⁴¹<https://julialang.org>

⁴²<https://www.scala-lang.org>

⁴³Source: <https://en.cppreference.com/w/cpp/ranges>

Merging these two programming paradigms offers an advantage over purely functional languages like Lisp⁴⁴, because the code does not diverge too much from the imperative style even though the functional features are being used; for example, the behaviour of a parallel function can easily be encapsulated into a C++ object, ideally stateless, by overloading the parenthesis-operator.

Additionally, GPU language extensions like CUDA move into the same direction of data parallelism by applying the same function (i.e. the global kernel) to different chunks of data, ideally independent and located at consecutive memory addresses. Also, a parallel region in OpenMP is just an anonymous function executed on the data segment assigned to it.

Nevertheless, while using functional concepts for expressing parallelism on independent data can be a natural choice, it has some limitations when a certain level of cooperation between executors is required. Undoubtedly, in this case, sharing the data and synchronizing the write accesses become less trivial. To achieve an optimal performance, especially on GPUs, (a) the entire functionality must be ported to a specific language and (b) computing experts tune the algorithms based on particular needs. Therefore, advanced knowledge together with a substantial development effort might be required, whereas natural scientists usually do not have a deep expertise in computer science.

Our contribution aims to simplify the parallelization approach by hiding it completely from the user, who can still develop the algorithm as a C++ class and call a function for its execution, in an imperative way.

The next chapter covers a series of research studies performed by the author in order to gather the specification for the framework. Since these studies focus on common use cases of the track reconstruction flow, our abstractions will not fit any problem type (e.g. stencil algorithms), and are mostly recommended for independent data parallel tasks with at most one synchronization point at the end. More details about this are provided in Chapter 4.

⁴⁴Lisp language was invented in 1960 [McCarthy, 1960], while more recent and popular dialects are CommonLisp (1994) and Clojure (2007)

3. Research Studies

In this chapter, an example of track reconstruction software is described together with our contributions to its development, driven by an R&D effort. The outcome provided us with important insights into ways of improving the performance and portability, thus serving as requirements and technical specifications for our advanced work, which are discussed later in this section. Details about the first attempt to transform the requirements into a framework conclude this chapter.

3.1. Track Reconstruction Software

Despite the fact that common track reconstruction principles are shared among experiments, their unique technical design and physics goals push the physicists of each experiment to develop their own reconstruction software stack. As hardware accelerators become more prominent as a powerful and efficient option for high performance computing, HEP Software Foundation decided to support the development of common but highly configurable solutions that could be reused by several experiments. In this section, we briefly describe one of these projects and our contribution to its development.

3.1.1. A Common Tracking Software

A *Common Tracking Software* (ACTS) is an experiment-independent toolkit for charged particle track reconstruction in high energy physics experiments implemented in C++ [Ai et al., 2022]. Initiated in 2015, ACTS is currently a stable framework used by several physics experiments like ATLAS, sPHENIX or ALICE for different parts of the reconstruction flow in production environments. It is an open-source project¹, which consist of the main reconstruction algorithms, *ACTS Core*, together with a list of plugins that extend the basic functionality, like the geometry description plugin *dd4hep*, the fast simulation tool *Fatras* or the visualisation tool *actSVG*. ACTS requires minimal build dependencies² and is covered by extensive automated tests, which ensure a high degree of both code quality and physics performance.

ACTS is designed for concurrency and ensures a thread-safe environment by providing stateless algorithms aligned to *const correctness*³ standard; the few exceptions where *const*

¹<https://github.com/acts-project/acts>

²Library dependency for ACTS Core are *cmake*, *eigen* and *boost*. Plugins might have additional requirements like *dd4hep* or *tgeo* packages for *geometry* description.

³In C++, passing an argument to a function as a constant reference or pointer marks it as *immutable* and any attempt to modify it leads to a compile-time error.

correctness is bypassed in ACTS are clearly explained in the online documentation. The framework makes use of C++17 templating features that allow user libraries to chain together several algorithms and to parametrize the reconstruction based on specific detector geometries. This feature allows not only multiple experiments to reuse the same code but also within the same experiment, the backward compatibility with past detector geometries is easily ensured.

The main algorithms exposed by ACTS and a valid way of chaining them together are depicted in Figure 3.1 where the reconstruction-related steps are being highlighted. ACTS provides several implementations for the main reconstruction steps. *Clusterization* and *Space point building* transform the data from the sensors to meaningful points in the detector coordinate system. *The seed finding* approach follows either the ATLAS logic—where points from the middle layers are identified first, then matched to points from bottom and top layers within specific regions and finally confirmed by a weighting score—or employs a k-d tree⁴, which divides the search space into smaller orthogonal regions and thus reduces the complexity to an average of $\mathcal{O}(\log n)$. Seeds are fed to the Combinatorial Kalman Filter (CKF), which handles both track finding and fitting in the same step and computes the fitted track parameters for all found tracks. The *Propagator* class in ACTS is templated on a *Navigator*, which provides the next candidate surface, and a *Stepper*, which is a module based on the RKN integrator that ensures the transport of track parameters and their covariances through the magnetic field.

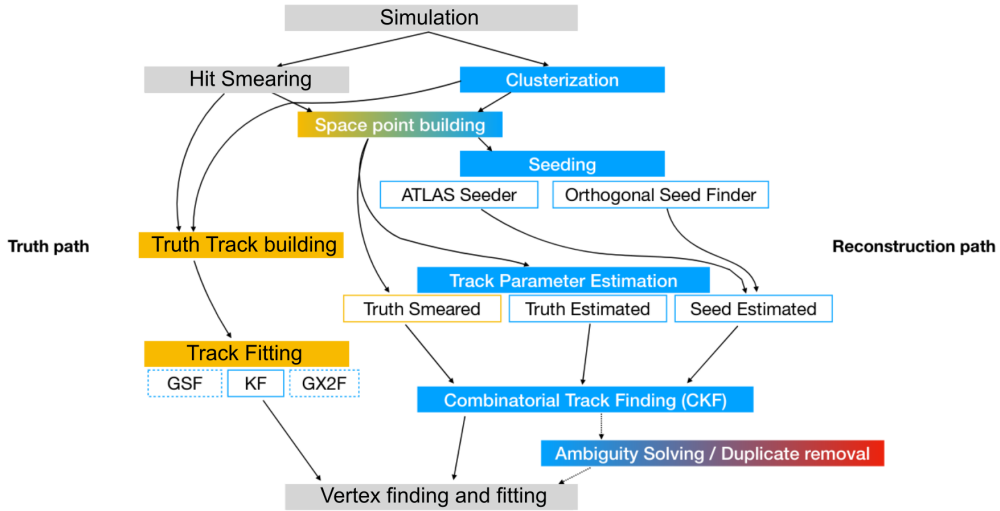


Figure 3.1.: ACTS Core reconstruction toolkit [Salzburger, 2022]

The track parameters description used in ACTS is based on the ATLAS model described in Section 1.2.3 with an extra parameter: time. Detectors with timing capabilities provide extra information for separating interactions that occur within the same event window, and hence have significant potential benefits for next-generation tracking detectors. Therefore, the ACTS track parameters are $(l_1, l_2, \phi, \theta, q/p, t)$ for bound state and $(\vec{r}, \vec{p},$

⁴A *k-d tree* is a k-dimensional generalisation of a binary search tree.

$q/p, t)$ for free state, respectively. Using one state representation or the other impacts the computations twofold. First, when using the free track parameters, there is the constant need to check the distance to the destination surface, which could require extra compute cycles. Second, the linear algebra operations handle matrices of sizes 6×6 , 6×8 , 8×6 and 8×8 when computing the transport jacobians for all the possible combinations of source and destination surfaces; the smoothing step of the KF alone employs the following matrix operations: five multiplications, one addition, three subtractions, two transpositions and one inversion, whose performance usually varies with the matrix size. Therefore, the execution time to reconstruct the trajectory of a particle using free track parameters is slightly larger than the case when bound surfaces are involved.

Despite being computationally intensive, most of the ACTS algorithms are memory-bound due to the large amount of memory that is required for computing each track (and that has to be accessible from the device as well). For the data volume estimated for HL-LHC (e.g. $t\bar{t}$ $\mu = 200$), the ATLAS magnetic field description uses 200MB, the CKF produces 3GB of track states and the KF used for truth tracking adds an extra 100MB [Gessinger, 2022]. In the light of these demanding computing requirements, several experimental projects based on ACTS code were developed with the goal to explore heterogeneous architectures. These will be briefly described next.

3.1.1.1. ACTS Heterogeneous Libraries

The R&D projects aimed to re-develop the ACTS core algorithms and data model having in mind the potential but also the limitations of a GPU and therefore to allow the heterogeneous hardware to share an implementation, when this is possible, or to provide highly specialized native code, otherwise. Nevertheless, the entry point that assembles the reconstruction chain is always written in a dedicated language: C++, CUDA or SYCL. This was considered to be as close to "single-source code" as possible without trading off performance. The outcome of this (ongoing) effort is summarized in Figure 3.2 and briefly explained below [Gessinger et al., 2023].

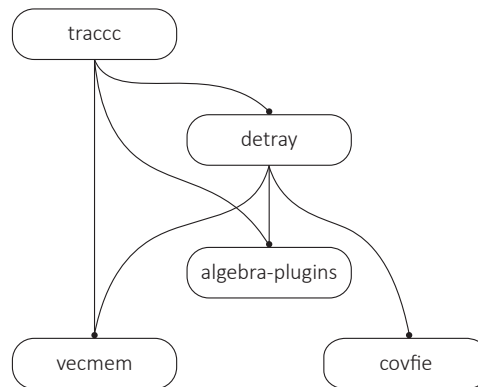


Figure 3.2.: ACTS R&D Heterogeneous Libraries [Gessinger et al., 2023]

⁵A top quark and top antiquark pair of particles with an average number of pileup interactions μ

Vecmem [Swatman et al., 2023] is a memory management library, which provides efficient data structures (e.g. `vecmem::vector` or `vecmem::jagged_vector`) and memory transfer mechanisms for handling both host and device storages. Based on the C++17 STL features described in Section 2.2, *vecmem* has a C++ core functionality that defines the heterogenous-friendly containers and the memory resources and allocators. Nevertheless, the implementations are done in native languages by each of the supported backends, which currently are C++, CUDA, HIP and SYCL.

Algebra-plugin is a linear algebra library with fast algorithms for common vector and matrix operations, including affine transformations. For each algorithm, a home-brew implementation (*cmath*) and a wrapper to an *eigen* function are provided. Also, the data structures can have different storage type, either based on `std::array`, `vecmem::vector` or `Eigen::Matrix`. When invoking *algebra-plugin*, a combination of the implementation backend and the storage backend has to be provided.

Detray [Salzburger et al., 2023] is a new concept for geometry description without layers that avoids runtime polymorphism; instead, flat data structures with lists of indexes to adjacent volumes steer the navigation. ACTS RKN stepper was ported and adapted to use *vecmem* data structures, and was integrated with the new geometry description and material interaction model. Regardless of the fact that *detray* is written in C++, the user libraries need to ensure that the detector is built on data structures with appropriate memory allocators so that it can be accessed on both host and device.

Covfie is a co-processor vector field library that optimises the layout and accessibility to n-space tensors. To deliver this, *covfie* relies heavily on category theory concepts to provide a generic, composable and extendable C++ header-only library, which allows the user to configure the storage location, the memory layout and the associate transformations based on provided functors. *Covfie* is currently being integrated into the *detray* project with the goal to provide fast access to read-only but very large tensor that describes the inhomogeneous magnetic field.

Tracc is the tracking chain demonstrator that employs all the other projects to develop algorithms for the reconstruction flow. To achieve the heterogeneity goal, *tracc* provides a common interface but with different backends for host and accelerator code, while striving for reusing as many functionalities as possible. At the moment, clusterization and track finding are supported in various degrees by the existing backends —C++, CUDA, SYCL and Futhark—, with the rest of the reconstruction steps currently being implemented.

Preliminary performance tests show considerable speedups in comparison to CPU code [Gessinger et al., 2023]. In parallel, other heterogeneous libraries like *NVIDIA stdpar* or *Kokkos* are currently being benchmarked using the R&D infrastructure.

3.1.2. Case studies: Evaluate the Parallelization Potential with ACTS

Before designing the ACTS heterogeneous libraries presented above, a series of *performance studies*⁶ were conducted to evaluate the parallelization potential of the reconstruction chain on different hardware platforms by either adding multi-threading capabilities or by porting to an accelerator language. While many scientists contributed to gather comprehensive feedback, the results described next cover only the projects where we had a direct contribution in developing the code, running the experiments and providing the interpretation.

3.1.2.1. Multi-threaded Kalman Fitter

This performance study was performed in collaboration with Dr. Xiaocong Ai (University of California, Berkeley), who supported this effort by running the experiments on the NERSC/Cori⁷ cluster.

Intel Threading Building Blocks (TBB) was used to parallelize the ACTS implementation of the *Kalman Fitter*⁸. The algorithm has two major loops: an outer loop that reads from the disk the available events and extracts useful information in local data structures, and an inner loop that iterates over the tracks inside each event, fits them and stores them in a local data structure ready for serialization. While both loops work on independent data, the results produced by the inner loop must be aggregated per event and written to an output file for validation (a Comma Separated Values (CSV) format in this case), therefore it involves a synchronization point at the end.

For the experimental part, three different CPU architectures from NERSC/Cori and UHH were used. Their configurations are described in Table 3.1.

System	Processor	Compute Units	Memory (GB)
Cori-Haswell	Intel Xeon E5-2698 v3	16 x 2	128
Cori-KNL	Intel Xeon Phi 7250	68 x 4	96
AMD Opteron	AMD Opteron 6168	48 x 1	128

Table 3.1.: CPU configuration *Compute Units* denotes the number of cores multiplied by the number of threads

We first looked at the fitting time for 10 000 tracks within the same event. This evaluation represents the inner loop described above and does not include any I/O operations. As explained in Section 1.2.3.1, the Kalman Filter algorithm relies on a numerical integrator that estimates the position and momentum of a particle inside the detector. Based on the magnetic field that is applied, three scenarios were investigated: (a) no magnetic field, (b) a magnetic field constant in z direction with the value of 1T, and (c) ATLAS inhomogeneous magnetic field⁹. The runtime of the algorithm was measured in 10 independent runs and

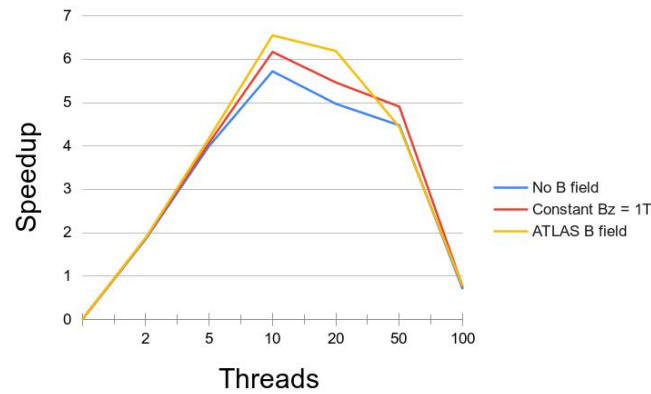
⁶These use some of the metrics detailed in Section 6.1.

⁷<https://docs.nersc.gov/systems/cori>

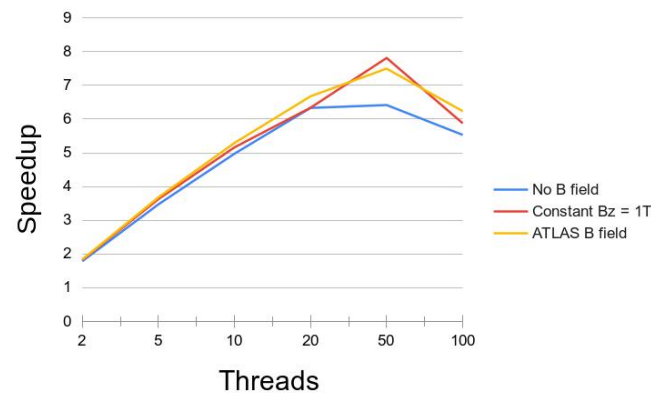
⁸This is a Kalman Filter implementation for Track Fitting step of the particle reconstruction flow.

⁹<https://gitlab.cern.ch/acts/acts-data>

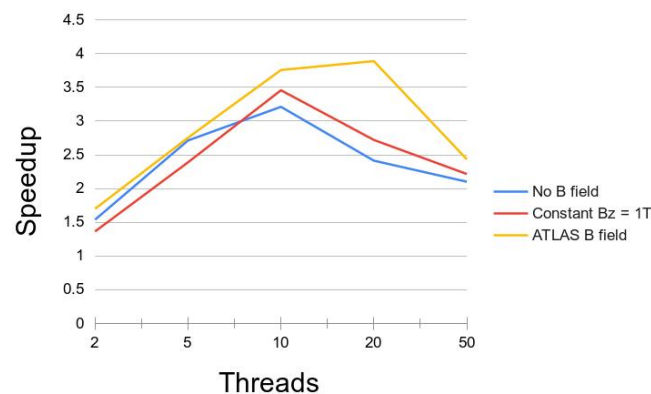
the mean value was used for the plots in Figure 3.3, which show speedup factors between 4 and 8 when using a multi-threading environment. This is lower than the number of executors, which according to Amdahl's law indicates that there are still parts of the code that are executed sequentially, so the code is not written entirely efficiently. Therefore adding more threads does not increase the benefits of the parallelization.



(a) AMD Opteron



(b) Intel Cori-KNL



(c) Intel Cori-Haswell

Figure 3.3.: Speedup of the parallel version over the sequential one when using different number of threads and different magnetic field values for fitting 10 000 tracks

Besides the values of the magnetic field, the particle's energy is another contributor

to the total wall clock time for the fitting algorithm because it influences the trajectory estimated by the numerical integrator, i.e. more or less integration steps could be required until convergence. Consequently we plotted the fitting time for different transverse momenta and number of threads in Figure 3.4.

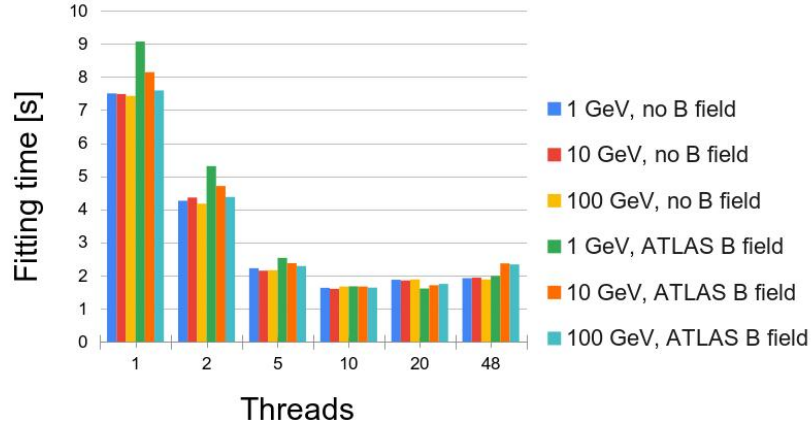


Figure 3.4.: Wall clock time in seconds for fitting 10 000 tracks at different momenta when using multi-threading configurations on AMD cluster

Secondly, we investigated the entire algorithm, which contains two levels of parallel loops and I/O operations. As shown in Figure 3.5, 65% of the time is spent in I/O operations when there is no parallelization available. Since read and write operations for different

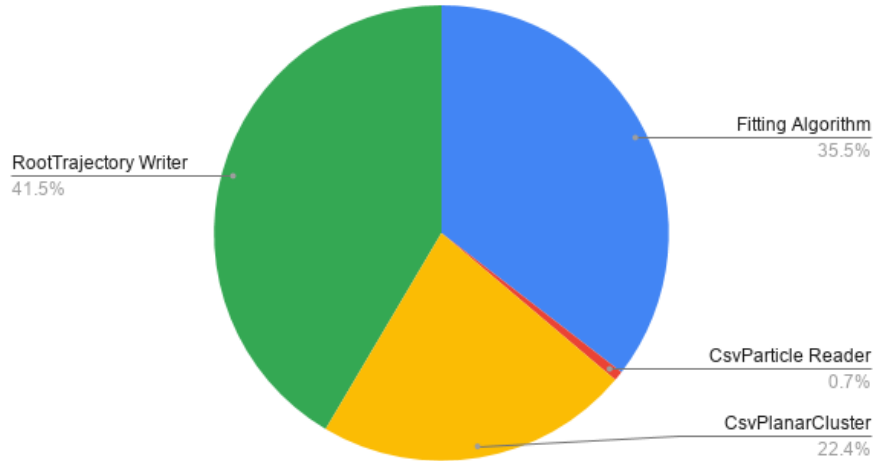
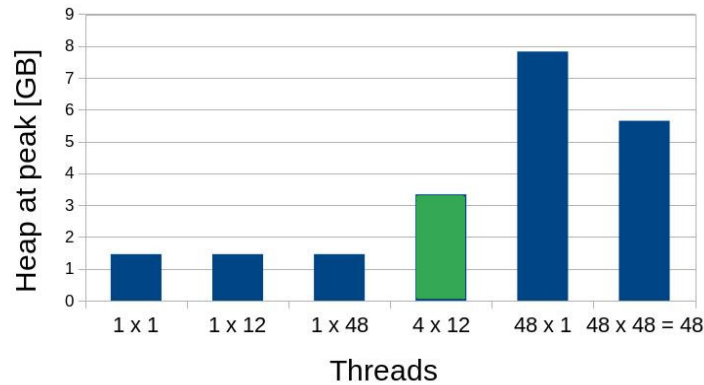


Figure 3.5.: Time distribution for fitting and I/O operations in sequential mode for fitting 10 events with 10 000 tracks each on AMD cluster

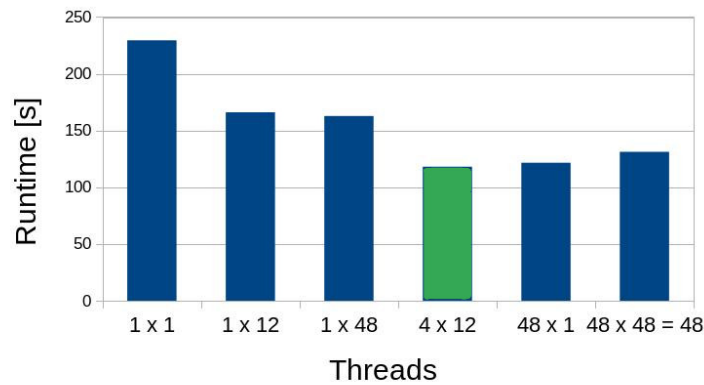
events are independent, they can be overlapped in multi-threading mode to hide some of the latency. By increasing the number of threads, the heap memory consumption also increases. An important point to mention is that the I/O operations are used only for algorithm validation and debugging. In production environment, these are not used since the fitting is an intermediate step of the reconstruction flow and the data is already

loaded in memory data structures.

Figure 3.6 shows that for this given scenario, an efficient parallelization strategy that would balance the execution time and the RAM memory required, involves a hierarchical scheme with two levels of 4×12 threads, on the given AMD system. The tests ran on the Intel nodes showed similar correlations between the number of available threads, the memory consumption and execution times. The last configuration with 48 threads per level uses 48 threads in total due to TBB policies. The observations are summarized in the Table 3.2.



(a) Heap memory at peak



(b) Wall clock time

Figure 3.6.: Memory consumption vs execution time trade-off when fitting $10 \times 10\,000$ particles using different number of threads on AMD cluster; the most efficient configuration is highlighted in green

Increase number of threads for / Impact on	Memory	Execution time per event	Total execution time
Event loop	Higher peak	No impact	Reduce considerably
Tracks loop	Small impact	Reduce considerably	Reduce marginally

Table 3.2.: TBB Parallelization empirical observations

When running the Kalman Fitter in a parallel environment, the order of both (a) the fitted tracks within events and (b) the results per event might differ from the solution

obtained by the sequential algorithm, but this does not have a *physical relevance*¹⁰. A sample file is listed in Figure 3.7. The correctness of the results was validated by running a comparison script on the output files after they were previously sorted by event identifier and internally, by particle identifier.

Parallel execution				Serial execution			
13:00:12	FittingAlgor	INFO	trajectories.size()=10	13:00:23	FittingAlgor	INFO	trajectories.size()=10
13:00:12	FittingAlgor	INFO	1 0 6 0 0	13:00:23	FittingAlgor	INFO	1 0 1 0 0
13:00:12	FittingAlgor	INFO	4503599728033792	13:00:23	FittingAlgor	INFO	4503599644147712
13:00:12	FittingAlgor	INFO	1 0 1 0 0	13:00:23	FittingAlgor	INFO	1 0 2 0 0
13:00:12	FittingAlgor	INFO	4503599644147712	13:00:23	FittingAlgor	INFO	4503599660924928
13:00:12	FittingAlgor	INFO	1 0 2 0 0	13:00:23	FittingAlgor	INFO	1 0 3 0 0
13:00:12	FittingAlgor	INFO	4503599660924928	13:00:23	FittingAlgor	INFO	4503599677702144
13:00:12	FittingAlgor	INFO	1 0 7 0 0	13:00:23	FittingAlgor	INFO	1 0 4 0 0
13:00:12	FittingAlgor	INFO	4503599744811008	13:00:23	FittingAlgor	INFO	4503599694479360
13:00:12	FittingAlgor	INFO	1 0 3 0 0	13:00:23	FittingAlgor	INFO	1 0 5 0 0
13:00:12	FittingAlgor	INFO	4503599677702144	13:00:23	FittingAlgor	INFO	4503599711256576
13:00:12	FittingAlgor	INFO	1 0 8 0 0	13:00:23	FittingAlgor	INFO	1 0 6 0 0
13:00:12	FittingAlgor	INFO	4503599761588224	13:00:23	FittingAlgor	INFO	4503599728033792
13:00:12	FittingAlgor	INFO	1 0 4 0 0	13:00:23	FittingAlgor	INFO	1 0 7 0 0
13:00:12	FittingAlgor	INFO	4503599694479360	13:00:23	FittingAlgor	INFO	4503599744811008
13:00:12	FittingAlgor	INFO	1 0 9 0 0	13:00:23	FittingAlgor	INFO	1 0 8 0 0
13:00:12	FittingAlgor	INFO	4503599778365440	13:00:23	FittingAlgor	INFO	4503599761588224
13:00:12	FittingAlgor	INFO	1 0 5 0 0	13:00:23	FittingAlgor	INFO	1 0 9 0 0
13:00:12	FittingAlgor	INFO	4503599711256576	13:00:23	FittingAlgor	INFO	4503599778365440
13:00:12	FittingAlgor	INFO	1 0 10 0 0	13:00:23	FittingAlgor	INFO	1 0 10 0 0
13:00:12	FittingAlgor	INFO	4503599795142656	13:00:23	FittingAlgor	INFO	4503599795142656
13:00:13	Sequencer	INFO	finished event 1	13:00:23	Sequencer	INFO	finished event 0

Figure 3.7.: Fitting results obtained when running the Kalman Fitter sequentially and in parallel mode for 10 tracks per event

3.1.2.2. Kalman Fitter porting from C++ to CUDA

This project was the work of several contributors and the results were published in the article *A GPU-Based Kalman Filter for Track Fitting* [Ai et al., 2021].

The C++ Kalman Fitter implementation available in ACTS was used as a starting point for the CUDA implementation. While the C++ code uses a generic detector geometry that can be configured for a specific detector, the CUDA prototype used a simplified and hardcoded telescope-like geometry with 10 planar surfaces perpendicular to the global x axis and placed equidistantly with 30 mm between two adjacent planes. The reasoning for this simplification was that a realistic detector description would have required increased resources, both memory- and computation-wise. The magnetic field was also simplified from a non-uniform 3D layout to a constant field of 2T along the z axis. The ground truth used to evaluate the correctness of the solution is a sample of Monte Carlo simulated muons and their associated hits smeared by Gaussian noise.

The CUDA implementation diverged from the C++ one due to numerous changes required for the GPU compilation and optimization. For example, Curiously Recurring Template Pattern (CRTP) was used to define the polymorphic classes that describe the geometry. Also, the detector's surfaces were allocated in page-locked memory on the host using `cudaMallocHost` in order to become easily accesible from the device at a higher bandwidth than pageable memory. While the C++ code relied heavily on linear

¹⁰The ACTS repository provides unit tests, which check that the results are within a given error tolerance. Also, a physicist also manually checked the results.

algebra library *eigen*, this had some limitations regarding matrix inversion algorithms on GPU at the time of our experiments. One of the issue was that some of the algorithms provided by this library were implemented as recursive functions, which are known to be incompatible with the small thread stack available on GPUs. Also, *cuBLAS* library, which provided routines for most of the linear algebra operation, had removed the support for calling the inversion kernel from another CUDA kernel. Therefore, we provided a custom iterative matrix inversion implementation in CUDA based on co-factor algorithm. Despite the fact that it was chosen due to its high degree of parallelism, it was only used in single-thread mode because of the technical limitations imposed by the actor-model of the Kalman Filter.

Several Intel CPUs and NVIDIA GPUs, described in Table 3.3, from NERSC and NAF facilities were used to benchmark the prototype.

System	Processor	Compute Units	Memory (GB)
Cori-Haswell-CPU	Intel Xeon E5-2698 v3	16 x 2	128
Cori-KNL-CPU	Intel Xeon Phi 7250	68 x 4	96
NAF-SL-CPU	Intel Xeon Gold 5115	20 x 2	376
Cori-V100-GPU	NVIDIA V100 SXM2	5120 2560	16
NAF-P100-GPU	NVIDIA P100 PCIe	3584 1792	16
NAF-V100-GPU	NVIDIA V100 SXM2	5120 2560	32

Table 3.3.: CPU and GPU configurations; *Compute Units* denotes the number of cores multiplied by the number of threads for the CPUs and the number of floating point arithmetic units in single and double precision for the GPUs

The experiments investigated some key points:

1. *the correctness and precision* of the results considering that the x86 floating point units use extended double precision registers (80-bit) while CUDA limits the size to 32-bit and 64-bit respectively as described in IEEE 754-2008 [IEEE, 2008];
2. *the wall clock time* when using different *parallelization strategies*
 - track-level parallelization – each OpenMP or CUDA thread handles the fitting for a different track; this is the straightforward case since the tracks are independent;
 - intra-track level parallelization – since the C objects that describe the Kalman Filter, the geometry and the magnetic field are constant but yet they have to be read for each track, this could make use of CUDA shared memory; therefore allocating one CUDA block to handle the fitting for one track and several CUDA threads within the block to boost the linear algebra operations;
3. the potential speedup of the CUDA implementation over a CPU multi-threaded implementation using OpenMP
4. the impact of different GPU architecture, block size, grid size, number of streams and registers on the wall clock time and *occupancy* when targeting the GPU

5. the impact on *distributing the work load* on two distinct GPUs and combining the results at the end; in this case, two OpenMP threads connect to two CUDA streams on different devices in parallel.

There are two observations that can be extracted from Figure 3.8. Firstly, the GPU starts

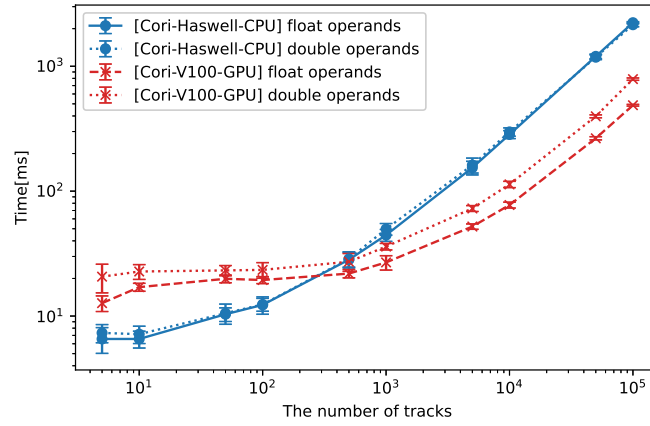


Figure 3.8.: The fitting time as a function of the number of tracks using float and double operands on Cori-Haswell-CPU and Cori-V100-GPU [Ai et al., 2021]

producing better wall-clock times than the CPU when the number of tracks exceeds 1000. Secondly, there is little impact on the CPU code when the precision of the operands is doubled, in contrast with the GPU where there difference is notable. In both scenarios, the number of OpenMP and CUDA threads was chosen to maximise performance.

Figure 3.9 shows that the runtimes on the V100 are better than on the P100 due to increased single/double precision performance and memory bandwidth.

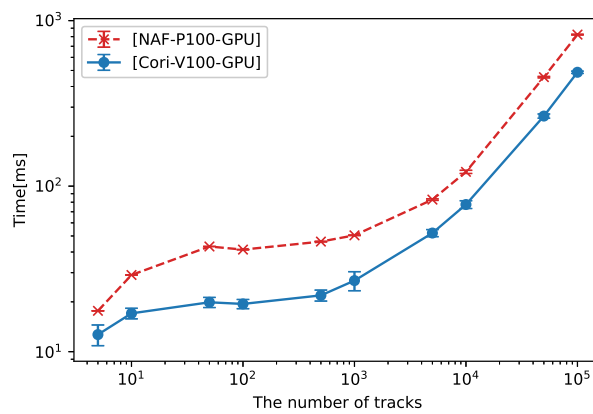


Figure 3.9.: The fitting time as a function of the number of tracks on NAF-P100-GPU and Cori-V100-GPU [Ai et al., 2021]

In terms of parallelization strategies, Figure 3.10 shows that intra-track one starts to bring benefits only when the number of tracks exceed 1000.

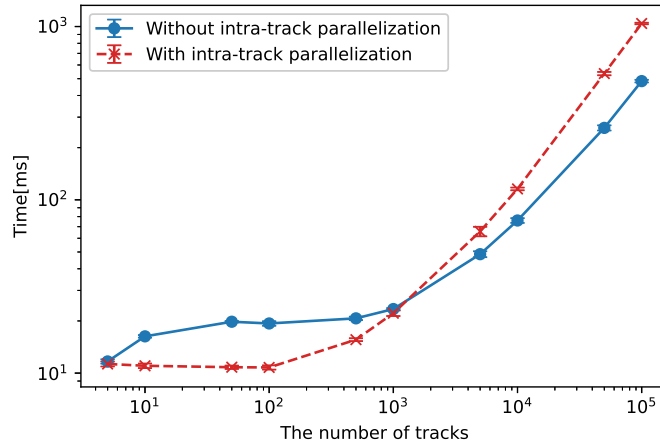


Figure 3.10.: The fitting time as a function of the number of tracks with linear grid of size $100\,000 \times 1$ with or without intra-track parallelization on Cori-V100-GPU [Ai et al., 2021]

Figure 3.11 and Figure 3.12 show that there is a trade-off between the execution time and the memory (more precisely the number of registers) used for each thread. While a 2D block of 32×32 threads and 64 registers per thread ensure an occupancy close to the theoretical maximum (50% in this case), it requires the largest runtime to perform the computations, regardless of the number of CUDA parallel streams or the grid's size. When there are 1024 threads per block, maximum 64 registers per thread are allowed.

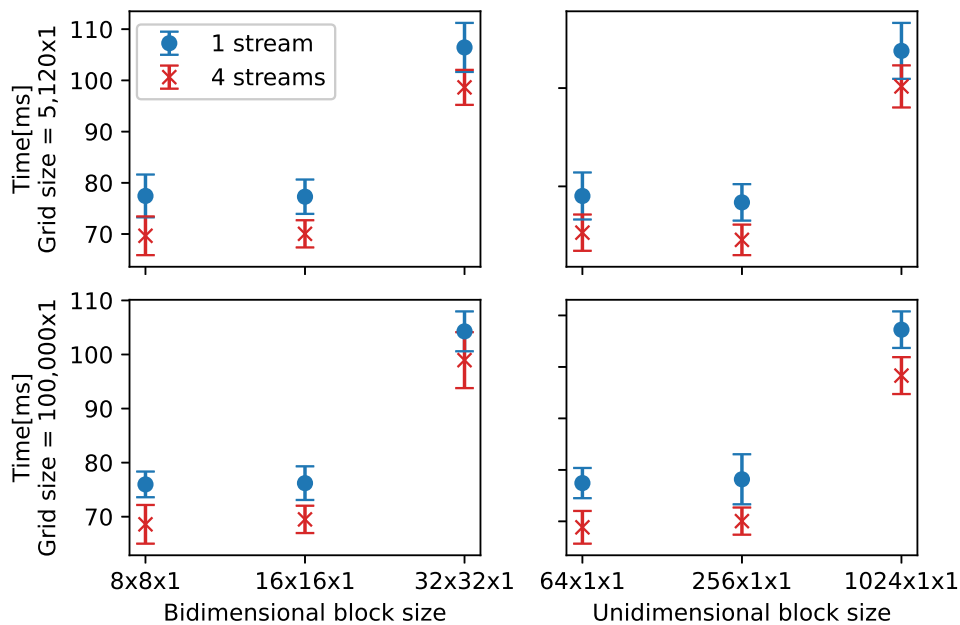


Figure 3.11.: The fitting time as a function of bidimensional (left) or unidimensional (right) block sizes with linear grids of sizes $5\,120 \times 1$ (top) and $100\,000 \times 1$ (bottom) on Cori-V100-GPU, with one or four streams per device [Ai et al., 2021]

Distributing the work load to two distinct GPUs using OpenMP threads brings little

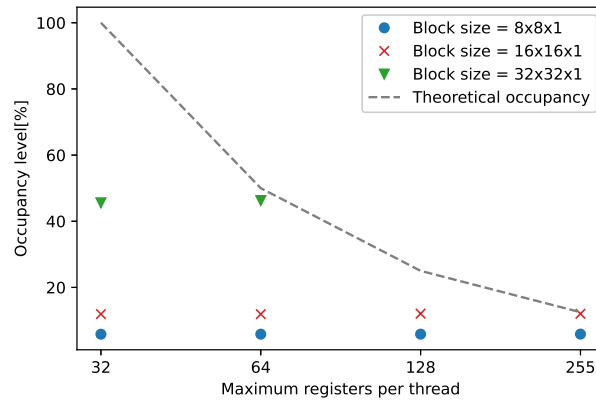


Figure 3.12.: Warp occupancy levels using various number of registers per thread and block sizes when using one stream with linear grid of size 5120×1 on Cori-V100-GPU [Ai et al., 2021]. The black dashed line is the theoretical warp occupancy.

benefits to the total wall-clock times as shown in Figure 3.13. MPI could probably be a more efficient alternative in this case, but this was left for future development.

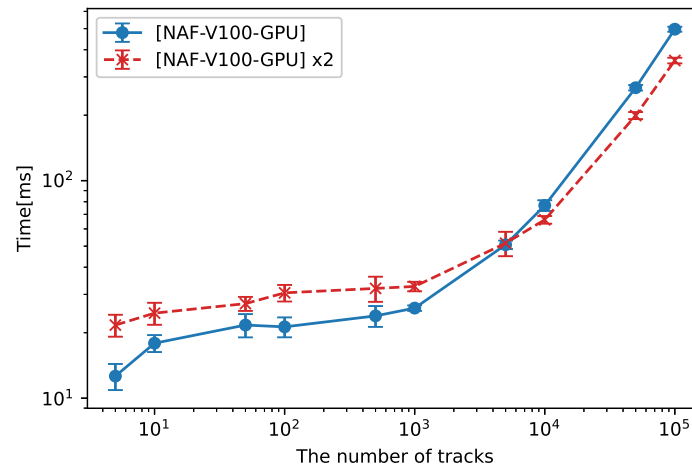


Figure 3.13.: The fitting time as a function of the number of tracks executed in one stream per device with linear grid of size 5120×1 and block size of $8 \times 8 \times 1$, when using one NAF-V100-GPU (solid blue) and two NAF-V100-GPU in parallel (dashed red) [Ai et al., 2021]

Finally, the implementation effort for porting the C++ version of the Kalman Fitter to CUDA was evaluated. Due to the fact that the CUDA example involved multiple physics simplifications¹¹, comparing the source lines of code (sloc) in the two examples might not be very relevant in this case. Nevertheless, we counted the sloc for main classes that handle the KF (e.g. `KalmanFilter.hpp`, `GainMatrixUpdater.hpp` and

¹¹Having a simplified hard-coded geometry and a constant magnetic field impact the numerical integration and surface intersection processes

GainMatrixSmoother.hpp) and the results showed 1295 sloc vs 1024 sloc for C++ and CUDA respectively¹². Additionally, the classes that handle the propagation and stepping were almost completely rewritten. From a subjective point of view, we asked the physicist, who was the main contributor to both C++ and CUDA implementations, to evaluate the effort, in three phases: (1) ramp-up process, (2) development, and (3) physics validation and performance improvements. First, C++ and ACTS knowledge required 3 months of study while learning basic CUDA skills added another 1.5 months. Second, assuming technical and domain knowledge is acquired, the C++ implementation took 2 weeks while the CUDA one took 2.5 months¹³. The final phase, took an equivalent amount of time in both cases¹⁴.

To conclude, this project demonstrated that there is a significant parallelization potential of this important step of the reconstruction flow. However, limitations imposed by CUDA language, incompatibilities between the ACTS data and actor execution models, and the porting effort were also substantial. These observations have had an important impact on the design of the new EDM in *traccc* and the new layer-less geometry in *detray*. A final point to mention is that, at the time of the prototype, an OpenMP target offloading implementation was also explored but we failed to deliver a compilable solution due to the lack of debugging tools and insufficient documentation.

3.1.2.3. Parallelization Strategies for traccc

In an early development stage for *traccc*, we conducted a performance study on the efficiency of several parallelization strategies for the first steps of the reconstruction chain, which produce the space points representation required for seed finding step, as shown in Figure 3.14, where F1, F2 and F3 are *cluster formation*, *measurement formation* and *space points formation* respectively.

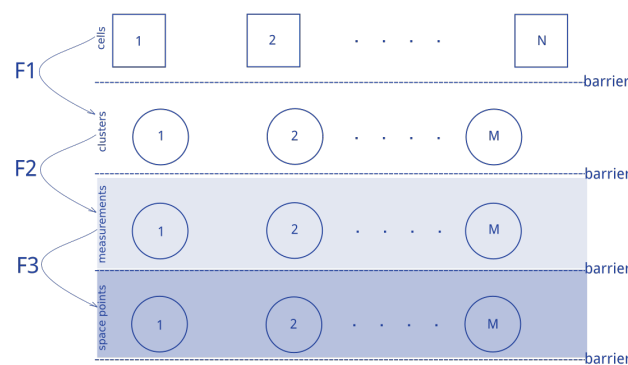


Figure 3.14.: Algorithm flow for transforming detector readouts into space point representation as implemented in *traccc*

¹²These values do not include code in utility functions like error handling.

¹³The initial development started during a hackathon supported by NVIDIA experts.

¹⁴Physics validation and performance improvement for C++ took 6 months for one developer while for CUDA, it took 3 months, but more people contributed to the outcome.

and *spacepoint formation* algorithms, N is the number of detector cells (from one event), M (with $M < N$) is the number of potential clusters/measurements/space points. The cluster and space points formation steps include reading the input data from a file and writing the results to another file, respectively.

Starting from this sequential implementation developed by the ACTS group (denoted with the name *seq*), the author provided three parallel alternatives, as follows: Firstly, *par* is a trivial OpenMP parallel loop with two imbricated critical regions for results aggregation. Secondly, *io_dec* is an approach based on *par* where the I/O operations are decoupled from the actual physics algorithms. This can be summarised in three steps: (1) read all the input data in parallel and store it in memory data structures; (2) run the algorithms in parallel and store the temporary results in memory; (3) write all the result files in parallel. Thirdly, *io_dec_dec* is based on *io_dec* but it provides a better granularity of the parallel regions, as shown in Figure 3.15; here $f1$, $f2$, $f3$ are the actions of algorithms $F1$, $F2$ and $F3$ applied to one entity of a collection. Once all the clusters are identified, we can run the measurement formation followed by the space point formation for each of them in parallel without the need to finalise all intermediate steps for all clusters in between. This removes the need of one of the barriers and reduces the size of the critical regions.

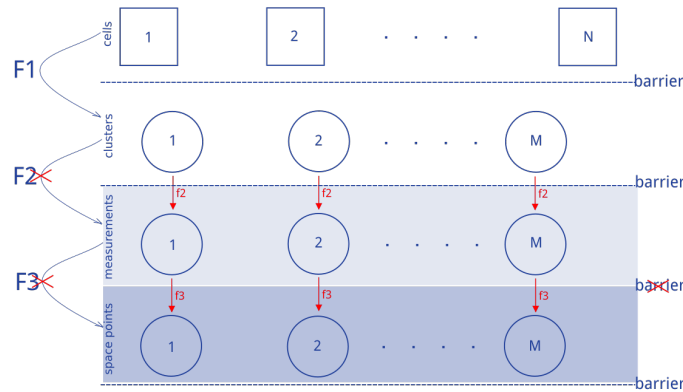


Figure 3.15.: Proposed algorithm flow for transforming detector readouts into space point representation as implemented in traccc.

The experiments used simulated data generated by the ACTS Fatras module, which contain muons in a constant magnetic field of 2T. The CPU used for the test was an Intel i7-1087H @2.2GHz, 8 physical cores, 16 threads. For each point, 5 measurements were taken and the average value was used for plotting.

To evaluate the impact of the I/O operations, we measured the time spent for reading and writing using *io_dec* implementation with 1 thread for 1000 events and when the events number variated. The results are shown in Figure 3.16a and 3.16b. More than 92% of the execution time is spent on I/O.

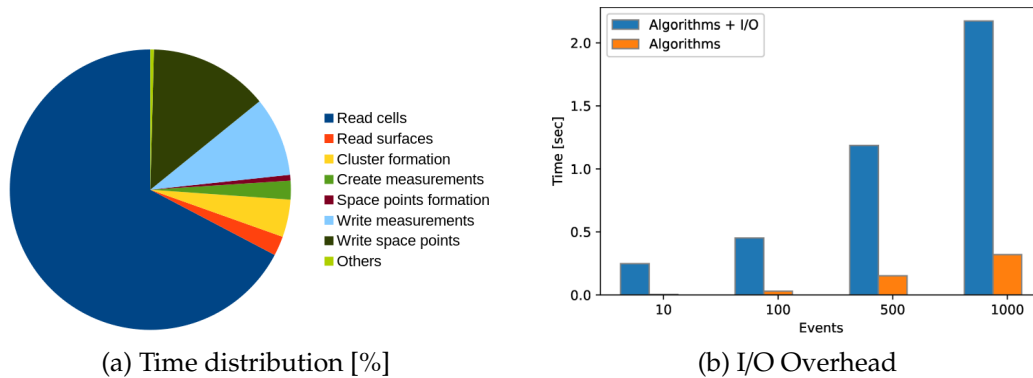


Figure 3.16.: Impact of the I/O operations when using one thread in *io_dec* implementation

Figure 3.17 shows that the parallel implementation scales well with an increasing number of events while keeping a constant number of OpenMP threads at 16. A speedup of $7\times$ is obtained by the *par* implementation when the number of events reaches 1000. The other two parallel implementation have a time penalty for creating and storing the intermediate in-memory data structures used for the algorithms. While decoupling I/O from the rest of the computations could be beneficial for evaluating the reconstruction algorithms in isolation, it is not the fastest solution. Nevertheless, it should be noted that the I/O operations are not part of a production use case since *space point formation* is an intermediate algorithm that assumes the data is already in memory.

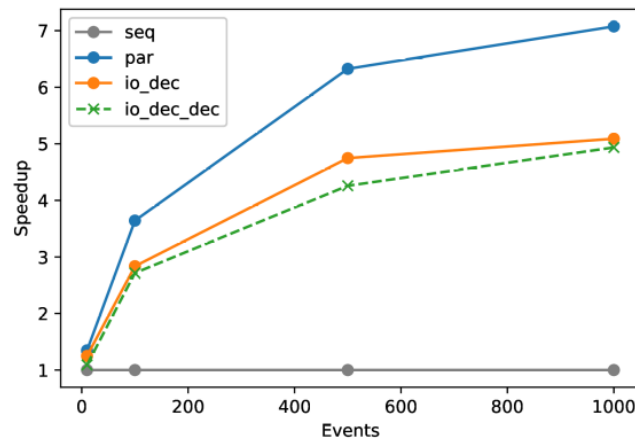


Figure 3.17.: Weak scaling analysis for the three parallel implementations in comparison with the sequential one

As a consequence of this work, *par* implementation was merged¹⁵ into *tracc* by the author and an effort to improve the I/O operations triggered by these observations is still on-going within the ACTS developer group at the time of this thesis.

¹⁵<https://github.com/acts-project/tracc/pull/40>

3.1.3. Case study: Evaluate Matrix Inversion GPU Implementations

One way to address the limitations described in Section 3.1.2.2 could be to improve and extend the GPU implementation for matrix inversion. In this case study, several approaches were investigated.

The co-factors method allows the computation of each element of the inverse matrix of size N independently, which translates into a trivially parallel algorithm with $2 \times N$ threads.

A challenge was to find an efficient way to allocate device memory for temporary matrices such as minors and cofactors. Dynamically allocating larger chunks in global memory proved to be highly inefficient. To address it, we used the information that the size of the matrix is known at compile time and therefore statically allocated this space in thread's stack in local memory.

Another optimization was to use shared memory to store the initial matrix since its elements are being used by all threads in the block. Finally, we replaced the eigen matrices and associated calls to eigen partitioning functions with C arrays and pointer arithmetic to ensure better data coalescence.

All the improvements brought a speedup of an order of magnitude between the CUDA kernel that inverts a matrix of 6×6 using a single CUDA thread and the optimized (shared-memory) kernel executed by a block of 6×6 threads.

Furthermore, a CUDA implementation for Gauss-Jordan algorithm was also provided. The advantage of this approach consists in a reduced amount of memory required for computations since the cofactors are no longer needed, but also comes with a downside: less parallelizable code. This stems from the fact that a set of operations (e.g. update and normalize) are applied to all matrix lines in each step. Also, the algorithm uses a temporary aggregated matrix of size $N \times 2N$ but which can be statically allocated and passed as a device pointer. To ensure full parallelism, the CUDA kernel should be invoked with a block of $N \times 2N$ threads. This solution, reduced the kernel time for inverting a 6×6 matrix by a factor of 20, to 2.3×10^{-5} seconds.

Starting from the above implementation, the student Yannik K  nneker and I investigated the performance of several implementations of Gauss algorithm using OpenMP, OpenACC and CUDA programming languages. While HIP and SYCL alternatives were also in focus, we decided to drop them due to the lack of debugging infrastructure available at the time of the project. The code is open-source¹⁶ and was forked from my private repository, which contained several CUDA implementations and a simple testing mechanism. The student extended the code with 6 new variations and wrote the evaluation framework. We ran the experiments on different machines and aggregated the results, which were published as part of his Bachelor Thesis [Koenneker, 2022].

In this new implementation form, the Gauss algorithm is done in 3 steps: normalization, Gaussian-elimination and update. For performance considerations, all implementations

¹⁶https://github.com/DoodleSchrack/matrix_inversion

feature distinct small kernels for each of the steps, which are offloaded to the device exploring the maximum level of parallelism available in the given language and supported by the associated compiler. For example, the key pragmas were `omp target teams distribute parallel for simd` and `acc parallel loop gang worker vector` for OpenMP and OpenACC respectively. Naturally, the execution configuration would use N blocks with N threads each, where N is the size of the matrix. While the launch bounds of the kernels are usually limited to 1024 threads regardless of the programming language and since the matrix to be inverted could potentially be larger than that, it was decided to partitioned it into N chunks. The CUDA implementation also made use of shared memory to optimize the access to elements of a given row, shared by all the threads in the block. The correctness of the GPU results was validated by (a) comparing them with the CPU results and (b) inverting the matrix twice and comparing the result with identity matrix. The performance was checked on two test environments described in Table 3.4. The following compilers were used: `gcc@9.3.0`, `clang@12.0.0`, `nvidia hpc++@21.9` in conjunction with libraries `cuda@10.2` and `eigen@3.4.0`.

Several types of matrices were pseudo-randomly generated: *normal* contain floats/doubles in $[-1.0, 1.0]$, *natural* contain integers in $[-100, 100]$ while *sparse* with floats/doubles $[-1.0, 1.0]$ on the main diagonal and the rest is filled with zeroes.

System	CPU	GPU	GPU Memory (GB)
1	Intel Xeon Gold 5218	NVIDIA Tesla V100	32
2	Intel i7 9700k	NVIDIA GTX 1080	8

Table 3.4.: Test environment setup

A selection of the resulting plots and conclusion is presented next. In most of the plots, the benchmark is the eigen implementation compiled for a CPU architecture. As per eigen's documentation¹⁷, multi-threading support is enabled by default when the code is compiled with `-fopenmp`. Then we can assume that eigen implementation is already parallelized, but there is no way of actually validating that. In case of a GPU execution, the final time measurements include the memory transfers from host to device and back.

Figure 3.18 shows the speedup of OpenMP and OpenACC implementations compiled with gcc versus the eigen CPU implementation for *normal* matrices of different sizes. As expected, the GPU offload implementation performs badly for small matrices since the memory transfers cost much more than the actual kernels. Nevertheless, they start to be faster than the benchmark once the matrix size exceeds 1024, reaching a speedup of 15× and 150× for OpenMP and OpenACC, respectively.

To evaluate the compiler's contribution when it comes to the parallel and/or offload implementations (with default configurations), we compiled the same code with gcc and clang using the same optimization level. The results are showed in Figure 3.19. Firstly, the measurements for inverting matrices larger than 1024 are missing for the executable

¹⁷<https://eigen.tuxfamily.org/dox/TopicMultiThreading.html>

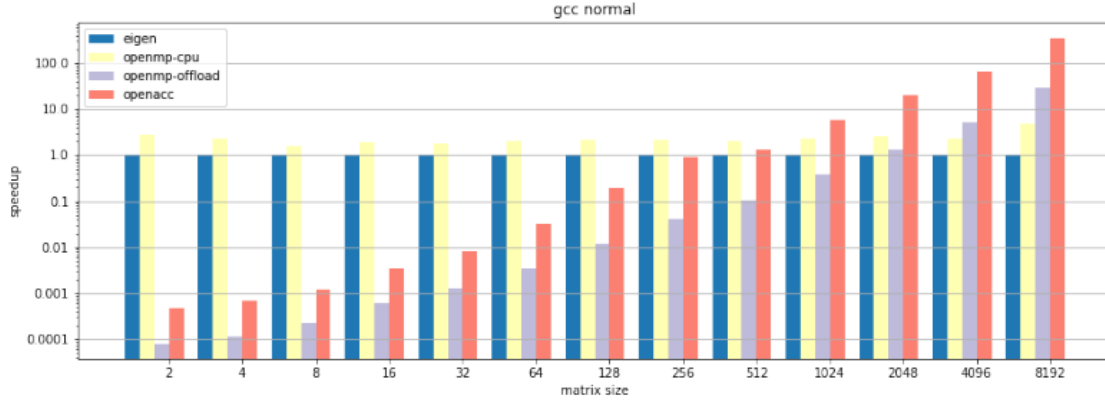


Figure 3.18.: Speedup over the eigen CPU implementation for matrices with different sizes when the executables were compiled with gcc or clang for host and device, on System 1 [Koenneker, 2022]

compiled with clang due to a runtime error. We assume that in this particular case, the compiler's mechanism, which ensures no more than 1024 threads are started simultaneously, does not work as expected, and throws an exception instead of automatically limiting the number to threads. Secondly, the executable compiled with clang show better runtimes than the ones compiled with gcc, up to 100× faster. The assumption here is that clang must use some extra performance optimization or different default values for the OpenMP runtime.

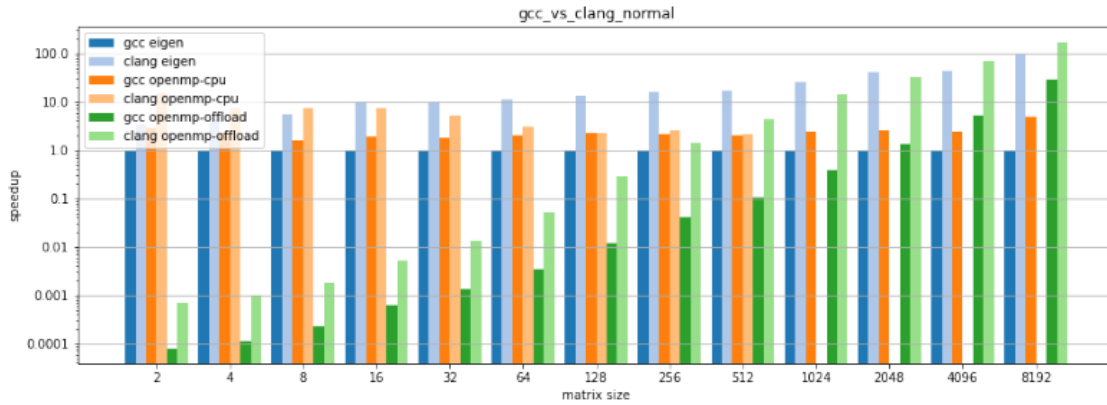


Figure 3.19.: Speedup over the eigen CPU implementation for *normal* matrices with different sizes containing floats when the executables were compiled with gcc or clang for host and device, on System 1 [Koenneker, 2022]

Figure 3.20 shows the performance comparison between several GPU implementations on System 2 and includes cuBLAS too. The nvcc-compiled openACC, CUDA and cuBLAS scale well with problem size up to a maximum speedup of $\approx 25\times$ over the OpenMP offloading one until the matrices are smaller than 512, but then the performance starts to degrade reaching $\approx 0.25\times$ for a matrix of size 2^{13} .

Since OpenACC offload outperforms OpenMP in all cases, NVIDIA Visual Profiler Tool (NVVP) was used to investigate the generated kernels' configurations on System 2.

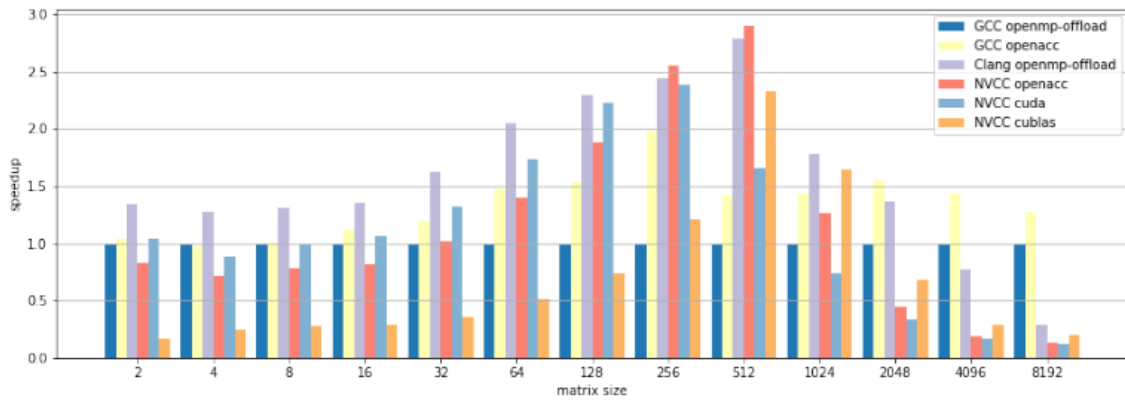


Figure 3.20.: Speedup over eigen GPU OpenMP offload implementation for *normal* matrices with different sizes containing floats compiled with gcc, clang and nvcc++, and cuBLAS on System 2 [Koenneker, 2022]

OpenACC allocates 1920 blocks with 32×24 threads for the Gauss elimination kernel, in comparison to 60 blocks with 32×8 threads for OpenMP. Thus the speedup is expected given the fact that OpenACC allocates $\approx 96\times$ more resources. Another interesting observation is that while offloaded implementations compiled with gcc and clang keep a fixed grid configuration regardless of the problem size, the nvcc++ versions factors it in: $(1 \times N \times 1)$ blocks with $(\max(2 \times N, 1024) \times 1, \times 1)$ threads for CUDA and $(N/4 \times 1 \times 1)$ blocks with $(32 \times 4 \times 1)$ threads for OpenACC.

Figure 3.21 shows that when using double precision the wall clock of the algorithms roughly doubles, which is to be expected. This holds true for all compilers and scenarios.

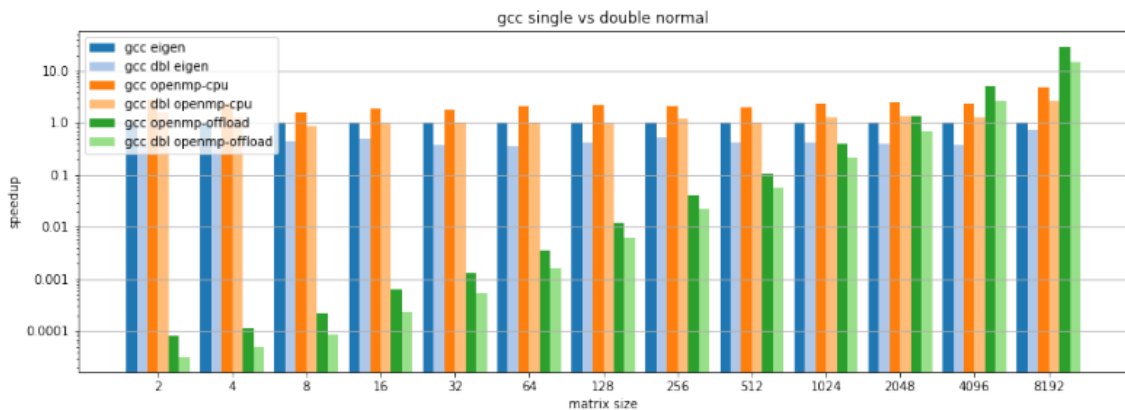


Figure 3.21.: Speedup over eigen CPU implementation for *normal* matrices with different sizes of floats and double compiled with gcc, clang and nvcc, on System 1 [Koenneker, 2022]

The error distribution of the implementations is also investigated. For example, for *sparse* matrices of size 1024, the errors are around 10^{-8} , with similar values for all the implementations, as showed in Figure 3.22.

In summary, we showed that offload techniques with OpenMP or OpenACC pragmas could bring important performance gains (up to $100\times$) when matrices required to be

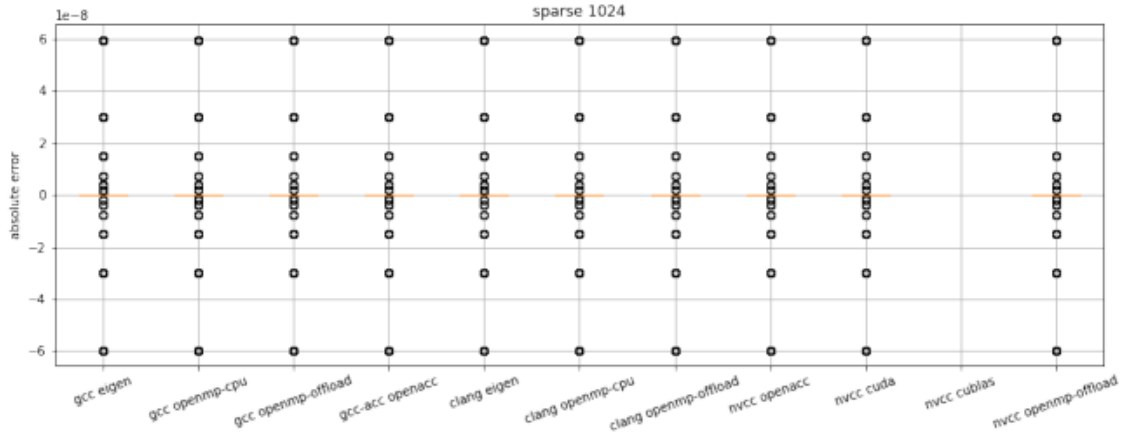


Figure 3.22.: Error distribution for several implementation when inverting a *sparse* matrix of size 1024 on System 1 [Koenneker, 2022]

inverted are larger than $2^{11} \times 2^{11}$. Since the particle reconstruction algorithm in our concern do not fall into this category, we decided to set aside this approach and still use the CUDA version. At the time of writing this thesis, there is an ongoing discussion about the integration of Gauss-Jordan implementation into CUDA backend of the *algebra-plugin*.

3.1.4. Conclusions

Several conclusions could be made based on the preliminary R&D efforts. First, all the case studies showed promising speedups when parallelization is applied, either using multi-threading on the CPU or CUDA threads on an NVIDIA GPU. Second, CUDA language constraints and execution limitations incompatible with the current ACTS design were identified and recommendations were provided to the developer forum. These contributed to the refinement of specifications for the ACTS heterogeneous libraries described in Section 3.1.1.1. Third, despite the ACTS libraries' potential to provide significant speedups when using GPUs, they require different backends to achieve this, thus forcing the developers to learn and then to maintain several implementations for the same algorithm, which decreases productivity.

Based on these observations, we derived the specification for a new framework, which will be detailed next.

3.2. Framework Specifications

Figure 3.23 summarizes the native programming languages and extensions for the major heterogeneous hardware currently available in supercomputers; the hashed surfaces denote that a language can be used to target a non-associate architecture but with external support from a specific compiler. For example C++ can be translated to NVPTX using either clang or nvc++ compilers but can also use OpenMP target features to run on AMD

GPU if the code is compiled with clang or gcc.

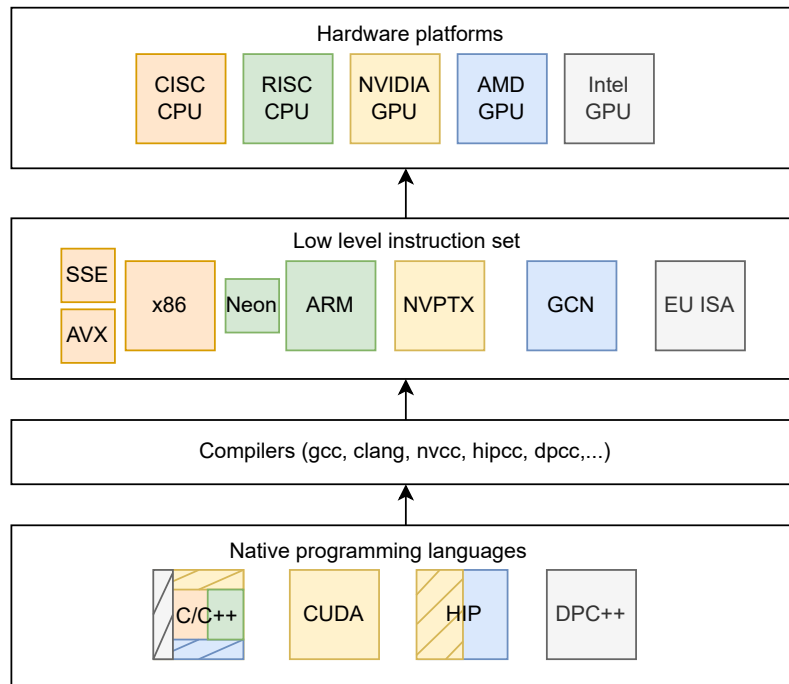


Figure 3.23.: Hardware platforms and their native programming languages

In general, all state-of-the-art approaches can achieve efficient execution on CPU and/or GPU, with different degrees of portability and productivity. CUDA ensures best wall-clock times on NVIDIA hardware but cannot target GPUs from other vendors. Similar to CUDA, HIP, OpenCL and SYCL are language extensions that require complete porting from C++, so more work for developers to learn them and to port the code. OpenMP and OpenACC are *portable* but they require changes to the existing implementations by adding compiler pragmas and do not always deliver the maximum performance on GPUs, as will be shown in the studies detailed in Section 5.2. C++ parallel standard ensures single-source code to target multiple architectures, but each executable has to be compiled with a different compiler, while not all parallel features are supported consistently. Also, for all of them except C++ STL, *parallelization and offloading code is tightly coupled with the domain algorithm*. This is the most important point we wanted to address with the current work: *separation of concerns* that will increase productivity by allowing domain scientists to focus on writing algorithms rather than keeping up with many language extensions that might offer speedups with the cost of portability, which later translates into the need of maintaining several implementations for modeling the same process. While C++ STL will probably close this gap in the future years, a viable solution is required now. In this context, we introduce *vecpar framework*, which aims to deliver the following:

1. *Increased development productivity* by providing

- *separation of concerns* – the particle physicists will only focus on the reconstruction algorithm and leave the computational decisions like parallelization

strategies or support for different runtimes to the backends maintained by computer scientists; this is achieved by abstracting the *parallel execution* notions;

- *straight-forward code development* – by offering a reduced set of operations as a Domain Specific Language (DSL) in C++; this eliminates the need to learn hardware specification (like memory layout concepts in targeted clusters) and software language extensions (like CUDA). Also, incremental porting to *vecpar* is also possible;
 - *code maintainability* – by reducing the need to maintain multiple repositories for different architectures; any new supported backend requires minimal or no changes in the user code.
2. *Increased portability* of the reconstruction code so that it can be executed efficiently on heterogeneous architectures of modern supercomputers; initially *vecpar* targets execution on a single cluster node with (a) shared-memory CPU, (b) NVIDIA or AMD GPUs, (c) Big Data Platforms (like Graviton-based clouds) but its architecture allows adding support for new platforms and/or extending to distributed memory nodes; more details about this are presented in the final chapter;
 3. A *prototyping environment* to validate the parallelization potential of a given algorithm on CPU and GPU while using a single-source implementation that targets different platforms; useful observations can later be fed back into other projects of the same nature (like for example the ACTS Parallelization libraries mentioned earlier in this chapter) or *vecpar* can directly be used as a production-ready solution.
 4. *Speedups* over sequential executions by (automatically) choosing the parallelization strategy adapted to the targeted platform.

The first implementation approach to achieve these goals was through the *clang-offload framework*, which served as a starting point for the final solution, the *vecpar framework*. However, some functionalities from the initial code base haven't been merged into the *vecpar* framework yet and are planned for future work. In the next section, the *clang-offload* prototype will be briefly introduced, while *vecpar*'s technical details are discussed at length in Chapter 4.

3.3. Clang-offload Prototype

The LLVM/clang compiler has the ability to expose an intermediate representation (IR) of a source code as an *abstract syntax tree* (AST) to outside caller libraries. This is achieved by three different methods depending on the desired outcome: first, by calling *libClang* which is an interface that can be invoked from any language; it is easy to integrate in code and provides read-only access to the AST; second, by implementing a new *clang plugin*, which enables full read/write access to the AST as part of the compilation step while the code reinterpretation has to be added to the LLVM repository; third, by using *libTooling* interface

to write *standalone tools*¹⁸ while having full access to the AST without the limitation of a specific clang version. Based on this last method, we designed a prototype framework named *clang-offload*, which contains a C++ header-only library used to mark the pieces of the code that should be parallelized, and a *source-to-source translation* tool, which converts C++ sequential code into different parallel implementations.

For this prototype, we had a simplified use case in mind: the scientific code would define a behaviour as a C++ function that is later invoked in a for-loop (potentially in a different C++ file), for each element of an input collection. Moreover, the function has one input and one output parameter. A particle reconstruction example would be: fit a track, for each set of track parameter objects in a list, in the context of a constant (and hard-coded) magnetic field.

3.3.1. Design and Execution Flow

Figure 3.24 shows an overview of the execution flow. The clang-offload tool is invoked on the source folder that needs to be parallelized. The user can provide a set of options and configurations, which include a desired architecture for the build, a specific *backend* (one from the three supported backends: OpenMP for CPU, OpenMP target and CUDA) and an *action* (translate the code with or without building an executable from the new sources, which is the result of running the tool).

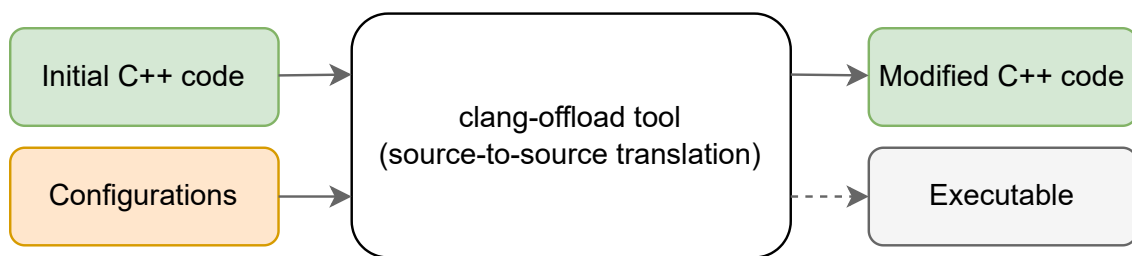


Figure 3.24.: Clang-offload execution flow

The *clang-offload* tool comprises four different tools, which are run in order as sequential steps for the main tool, as shown in Figure 3.25. The *FileGeneratorTool* duplicates the

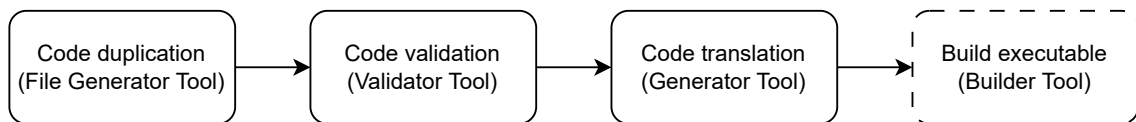


Figure 3.25.: The clang-offload modules; the ones in dashed line are optional for the execution flow

folder containing the source code; this is a precaution and stems from the fact that the code translation is a destructive process. The *ValidatorTool* makes sure that a series of preconditions are met in the context of a given backend; for example, for the CUDA

¹⁸Standalone refers to independent from of a specific LLVM repository.

backend, the code must not use any calls to the C++ standard library since these are not supported. The *GeneratorTool* is the key component of the entire flow because it rewrites the source code. Both the validator and the generator tools are implemented based on `clang::tooling` API, which allows the definition of AST matchers and associated callback functions that are invoked in different passes of the compilation process. Finally, the *BuilderTool* is called when the user specifies that an executable must be built after the source transformation. This is optional and the execution flow can end before this step. If one of these tools fails, an error is shown to the user and the entire flow stops.

To decide which parts of the source-code to translate, the tool uses input from the user, who will mark the code by replacing a for-loop with a call to the framework's mark API function `run_efficiently()` and include the appropriate marker header, as shown in Listing 3.1, while the signature of the API interface is shown Listing 3.2. When passed through the *clang-offload tool*, line 11 from Listing 3.1 gets replaced by a piece of generated code for each backend. The OpenMP for CPU version is shown in Listing 3.3 while the generated code for OpenMP target and CUDA backends is listed in Appendix B.1 since more changes are required in different files.

```

1  #include "functions.cpp"
2  #include "include/mark-offload/marker.hpp"
3
4  int main(int argc, char** argv) {
5      // int n = ..
6      int *h_data = (int *) malloc(sizeof(int) * n); // initialization ...
7      int *h_result = (int *) malloc(sizeof(int) * n); // initialization ...
8      // initial for-loop
9      // for (int i = 0; i < N; i ++)
10     //     functions::f(h_data[i], h_result[i]);
11     api::run_efficiently<int*, int, int*, int>(functions::f, h_data, h_result, n);
12 }

```

Listing 3.1: Sample user code using clang-offload API

```

1  // define function type
2  template<typename TDataItem, typename TResultItem>
3  using func_t = void (*)(TDataItem&, TResultItem&);
4
5  template<typename TData,          // TData<TDataItem>
6          typename TDataItem,
7          typename TResult,        // TResult<TResultItem>
8          typename TResultItem>
9  void run_efficiently(func_t<TDataItem, TResultItem> func,
10                      TData &data,          // input vector
11                      TResult &result,      // output vector
12                      int n);              // size of the vector(s)

```

Listing 3.2: Mark API

```

1 auto lambda = [<typename T, typename R>(T* data, R* result, int n) {
2
3     #pragma omp teams distribute parallel for
4     for (int idx = 0 ; idx < n; idx ++) {
5         functions::f(data[idx], result[idx]);
6     }
7 };
8
9 lambda(h_data, h_result, n);

```

Listing 3.3: Generated code for OpenMP CPU backend

The decision to generate a lambda function (when the code could have been generated as part of the original function) is due to the need of renaming some variables in the predefined templates with the names extracted while parsing the abstrac syntax tree.

3.3.2. Implementation Details

The implementation is composable and revolves around modules, which makes it flexible and maintainable. It defines the notion of a *ToolBox*, which is a placeholder for a set of tools that defines the behaviour of a backend, and which is required to provide implementations for the four tools. The tools are C++ classes with a default behaviour, which might be overridden for specific backends. The implementation relies heavily on polymorphism and virtual functions when needed¹⁹. For example, the *BuilderTool* extracts a build command from the *compilecommands* file generated by the *cmake* invocation on the original sources and extends it with the appropriate compilation flags and linked libraries as provided by each backend. Then the command is invoked using `std::system` API. Likewise, the *FileGenerator* tool is implemented based on STD API, however its behaviour is identical for all backends.

The most complex modules are the *ValidatorTool* and the *GeneratorTool*, which extend `clang::tooling::ClangTool` and have a similar structure: firstly, a virtual `addMatcher` function, which allows specific `MatchFinder`²⁰ objects to be added for each backend. Secondly, a `run_tool()` function that instructs the clang frontend to create a new compilation pass with the given matchers. Each matcher is basically an association between a condition that triggers the execution of a callback during compilation phase. An example of such condition is shown in Listing 3.4 and a sample callback function is shown in Listing 3.5. While the callback is rather generic, each backend defines its own notion of *included libraries* and uses a different template for the lambda generation. Lines 4 and 16 from Listing 3.5 identify the exact file location where `run_efficiently()` is being called and its arguments (e.g. variable names and types); the callback will replace the call with a template-based generated code (which includes a lambda and its invocation

¹⁹As a remark, this is the implementation of the tool and does not interfere with the execution of a backend, therefore the limitations imposed by a GPU backend do not apply here.

²⁰`clang::ast_matchers::MatchFinder` class allows finding matches over the Clang AST

call) defined by each backend. This callback can repeatedly be triggered if several calls are made throughout the user code.

```

1 static const StatementMatcher OffloadFnInvocationMatcher =
2   declRefExpr(
3     hasParent(
4       implicitCastExpr(
5         hasParent(
6           callExpr(
7             callee(
8               functionDecl(
9                 hasName(Constant::offloadFnName)))) // "run_efficiently"
10            .bind(Constant::offloadCall))))))
11 .bind(Constant::funcDecl);

```

Listing 3.4: Matcher statement for identifying the *run_efficiently()* call

```

1 virtual void run(const MatchFinder::MatchResult &Result) override {
2   ASTContext *Context = Result.Context;
3   SourceManager *SM = Result.SourceManager;
4   const CallExpr *CE = Result.Nodes.getNodeAs<clang::CallExpr>(Constant::
5     offloadCall);
6   if (CE) {
7     std::vector<std::string> params;
8     params.reserve(3);
9     for (int i = 1; i < CE->getNumArgs(); i++) {
10      const VarDecl* p = dyn_cast<VarDecl>(CE->getArg(i)->
11      getReferencedDeclOfCallee());
12      if (p)
13        params.emplace_back(p->getDeclName().getAsString());
14    }
15    // get the arguments from the "run_efficiently" call
16    const DeclRefExpr *DRE = Result.Nodes.getNodeAs<clang::DeclRefExpr>(Constant
17      ::funcDecl);
18    if (DRE && (hasNodeId(DRE->getID(*Context)))) {
19      m_offloadNodeIds.insert(DRE->getDecl()->getID());
20      StringRef location = SM->getFilename(Context->getFullLoc(DRE->getBeginLoc()
21      ));
22      // include <omp.h> or <cuda.h> ..
23      includeDriverLib(location);
24      // generate code that will be inserted into "location"
25      generateOffloadingLambda(CE, DRE, Context, SM, location);
26    }
27  }
28 }

```

Listing 3.5: Callback function for identifying offloading region registered in the GeneratorTool for the OpenMP Target backend

The code replacements are chunks of code that can include statements, OpenMP pragmas, function decorators (like adding `__device__` attribute to a function), or even functions all together in the form of C++ lambdas²¹. Also, the AST is searched in depth, so for example if a function is annotated with `__device__` attribute, then all the functions invoked by the current function are also annotated accordingly.

3.3.3. Limitations and Conclusions

While the tool works fine for this simple use case, we found it difficult to generalize it for more complex scenario, like the case when the threads need synchronization mechanisms. To address this, we noticed that *run_efficiently()* function must be (a) more flexible in defining the parameters and (b) more knowledgeable about what kind of operations are happening inside.

The code generation is based on some hard-coded templates for each backend. If the function becomes more generic and can accept more parameters, this model will no longer work. Also, we found different limitations of the tooling mechanism when identifying the functions as part of C++ classes/structures, which is a more probable case for a scientific code.

Also, since the parallelization happens at compile-time, only one backend can be active at a time; this means that addressing two platforms, the clang-offload tool will generate two source-folders. Nevertheless, since the transformations are deterministic, one can store only one implementation in a repository, while generating different versions only at compile time when the code is built for a specific target.

Based on this experience, we decided to design the vecpar framework as a library instead of a compile-time tool that allows extraction of more information from the user by fitting the code into a series of abstractions. This also removes the need for hard-coded templates and opens the possibility to compile the code with other compilers and is no longer limited to clang. Nonetheless, the vecpar framework could benefit from such an external tool to implement advanced performance optimizations, as detailed in the final chapter.

²¹Lambdas were chosen to match easier with the template code for each backend.

4. Technical Design

This chapter describes the vecpar framework, from its mathematical foundation to C++ implementation details.

4.1. Conceptual Design

Vecpar relies on the notations and calculus from functional programming for specifying and manipulating computable functions over lists. Section 4.1.1 briefly summarizes these concepts introduced by Richard Bird in [Bird, 1987], while the description of the vecpar abstractions concludes this section.

4.1.1. Mathematical Concepts

A *list* (or equivalent, a *sequence*) is a linearly ordered collection of values of the same type. The elementary operations are governed by the following operators:

- *Map* – The operator $*$ applies a function to each element of a list and it is defined by the Equations (4.1) and (4.2), where f is a function of type $\alpha \rightarrow \beta$, and $a_1, a_2, \dots, a_n \in \alpha$ while the result is another list with elements of type β , of the same length as the input one.

$$f * [a_1, a_2, \dots, a_n] = [f a_1, f a_2, \dots, f a_n] \quad (4.1)$$

$$* : (\alpha \rightarrow \beta) \times [\alpha] \rightarrow [\beta] \quad (4.2)$$

- *Filter* – The operator \triangleleft takes a predicate p and a list x and returns the list of elements of x that satisfy p . This is defined by Equations (4.3) and (4.4).

$$\triangleleft : (\alpha \rightarrow \text{Bool}) \times [\alpha] \rightarrow [\alpha] \quad (4.3)$$

$$p : \alpha \rightarrow \text{Bool} \quad (4.4)$$

Filter obeys the three laws described in Equation (4.5): *commutativity* states that filtering a list with the predicate q and then filtering the result with the predicate p gives the same answer as first filtering with p and then with q ; *idempotency* ensures that multiple filtering with predicate p give the same result as when p is applied once; and finally, *commutativity of map and filter* says that mapping with function f

followed by filtering with predicate p gives the same result as first filtering with $p \cdot f$ and then mapping with f .

$$\begin{aligned} p \triangleleft q \triangleleft x &= q \triangleleft p \triangleleft x \\ p \triangleleft p \triangleleft x &= p \triangleleft x \\ p \triangleleft f * x &= f * (p \cdot f) \triangleleft x \end{aligned} \tag{4.5}$$

- *Reduce* – The operator $/$ takes an operator \oplus on the left and a list x on the right and it inserts \oplus between adjacent elements of x , as shown in Equations (4.6) and (4.7). In order to avoid ambiguity, \oplus must be associative. If \oplus additionally has an identity element, then $/$ is a *homomorphism*¹.

$$\oplus / [a_1, a_2, \dots, a_n] = a_1 \oplus a_2 \oplus \dots \oplus a_n \tag{4.6}$$

$$/ : (\alpha \times \alpha \rightarrow \alpha) \times [\alpha] \rightarrow \alpha \tag{4.7}$$

- *Direct reductions* – The operators \Leftarrow (right-reduce) and \Rightarrow (left-reduce) are related with the reduction operator $/$ but there are several key differences, shown in Equations (4.8) and (4.9). Firstly, each takes three arguments: an operator \oplus , a value e and a list x ; secondly, in this case, \oplus does not need to be associative; lastly, the type of \oplus is not restricted to $\alpha \times \alpha \rightarrow \alpha$. For the right reduction, the result is determined incrementally from right to left, therefore the progress of the computation is *recursive*, which can be more time-efficient if the operator is *non-strict*² while the size of the intermediate expression grows proportionally to the length of the original list. In contrast, left reductions can be more efficient regarding the space required for the computation because in this case, the intermediate expression never grows beyond a constant amount.

$$\Leftarrow : ((\beta \times \alpha \rightarrow \beta) \times \beta) \rightarrow [\alpha] \rightarrow \beta \tag{4.8}$$

$$\Rightarrow : ((\alpha \times \beta \rightarrow \beta) \times \beta) \rightarrow [\alpha] \rightarrow \beta \tag{4.9}$$

Since every homomorphism can be expressed as a directed reduction, if \oplus is associative and has identity element e , then the undirected reduction can be expressed as a directed one as shown in Equation (4.10).

$$\oplus / = (\oplus \Rightarrow e) = (\oplus \Leftarrow e) \tag{4.10}$$

An important consequence of using homomorphisms is the *promotion lemma*, which states that for arbitrary function f , predicate p and associative operator \oplus , the statements

¹As an observation, we note that *map* and *filter* are also homomorphisms.

²A non-strict operator's value does not always depend on the full evaluation of the right-hand argument.

in (4.11) hold, where ++ is the concatenation operator. This means that rather than applying *map*, *filter* and *reduce* on a large sequence, one can divide the list into shorter ones, then *map*, *filter* and *reduce* each of these, and then collect the outcome.

$$\begin{aligned}
 (*\text{promotion}) \quad (f *) \cdot (\text{++} /) &= (\text{++} /) \cdot ((f *) *) \\
 (\leftarrow \text{promotion}) \quad (p \leftarrow) \cdot (\text{++} /) &= (\text{++} /) \cdot ((p \leftarrow) *) \\
 (/ \text{promotion}) \quad (\oplus /) \cdot (\text{++} /) &= (\oplus /) \cdot ((\oplus /) *)
 \end{aligned} \tag{4.11}$$

4.1.2. Vecpar Abstractions

Building on top of the equations and notations above, *vecpar* defines several types of abstractions to support the implementations for:

1. the operators *map*, *filter* and *reduce*, which are named *parallel_map*, *parallel_filter* and *parallel_reduce* respectively – These decide *how* a function will be executed with different *vecpar backends* providing different parallelization strategies. *Vecpar* extends the domain of these operators by adding additional information regarding the algorithm that needs to be executed, the storage location of the data and contextual information in some cases. For the *parallel_map* (and similarly for the *parallel_mmap*³) operator, *vecpar* allows up to *five*⁴ lists to be iterated in parallel, while for *parallel_reduce* and *parallel_filter* only one list is accepted. Details about function signatures will be presented in Section 4.3.

There is also the option to use composed operators *map-filter* and *map-reduce*, named *parallel_map_filter* and *parallel_map_reduce* in *vecpar*, which implement some performance optimizations when executed on GPUs.

Vecpar also exposes a generic *parallel_algorithm*, which delegates to one of the above for further execution and a mechanism for executing consecutive steps of an algorithmic chain by composing them together, in the mathematical sense, if they can be expressed as *vecpar* operators. Assuming the function chain depicted in Figure 4.1 is applied to a list x , the framework can produce the composed result shown in Equation (4.12).

$$\begin{aligned}
 (\text{reduce}_1(\text{filter}_2(\text{map}_3(\text{map}_2(\text{filter}_1(\text{map}_1(x))))))) &= \\
 (\text{reduce}_1 \circ \text{filter}_2 \circ \text{map}_3 \circ \text{map}_2 \circ \text{filter}_1 \circ \text{map}_1)(x)
 \end{aligned} \tag{4.12}$$

³*Parallel_mmap* is *parallel_map* that operates on *mutable* input data.

⁴This comes from the most complex use case in the reconstruction flow implemented in ACTS *detrax* project that handles the particle propagation through the detector's layers; for this, vectors containing the track parameters, the intersection candidates, the path lengths, the position data and the transport jacobians are needed together with the contextual description of the detector geometry in order to compute the final track states. Nevertheless, this number can be extended in the future.

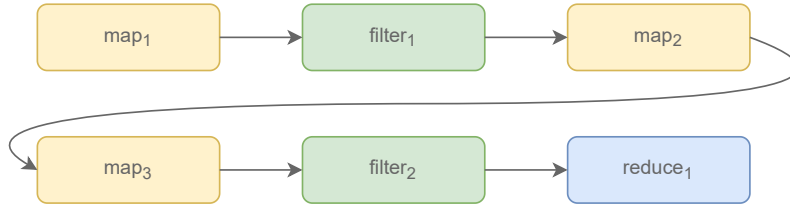


Figure 4.1.: Algorithm steps example

2. the function f , the predicate p and the direct reduction function \oplus , which are named *mapping_function*, *filtering_function* and *reducing_function* respectively.

Mapping_function extends the domain of f to include the context θ as shown in Equation (4.13).

$$\text{mapping_function} : \alpha \times \theta \rightarrow \beta \quad (4.13)$$

In the particular case when $\alpha = \beta$ and the list which f is applied to is expected to be mutable, (4.13) becomes (4.14):

$$\text{mapping_function} : \alpha \times \theta \rightarrow \alpha \quad (4.14)$$

Equation (4.4) remains unchanged, therefore *filtering_function* is defined as:

$$\text{filtering_function} : \alpha \rightarrow \text{Bool} \quad (4.15)$$

For reduction case, vecpar supports only homomorphic functions with $\alpha = \beta$ and can be defined as a left-reduction:

$$\text{reducing_function} : \alpha \times \alpha \rightarrow \alpha \quad (4.16)$$

Additionally, vecpar defaults the identity element for the reduction operation through the function defined by Equation (4.17). The user is required to overload this function only in the case when the identity element is different than the value provided by the default constructor of that type (i.e. for numerical types this is 0).

$$\text{identity_function} : \alpha \rightarrow \alpha \quad (4.17)$$

Technically, all these functions are wrapped in *vecpar::algorithm* classes or structures that provide a placeholder for the numerous templates required for an efficient execution, and are detailed in Section 4.3.

These abstractions are implemented through C++ structures and functions and they form the vecpar API, detailed later in this chapter.

4.2. High-level Design

Besides the higher-order algebra of program invariants such as the map-reduce list homomorphisms described above, which are inherently parallel, other concepts from functional programming are built into vecpar. Firstly, while the vecpar algorithms can be C++ functors, the desired behaviour is the one of a *stateless class/struct* with constant member functions; this ensures a GPU-friendly layout that allows both instantiation and copy of an algorithm to a device. Secondly, *immutability* is guaranteed by default (by obeying *const correctness principle*) unless the user explicitly requires otherwise; working with immutable data structures enhances the multi-threading capabilities by reducing potential race conditions or deadlocks, since each function works on its own data.

The vecpar framework is designed as a C++ *header-only* library that acts like a framework due to its *inversion of control* characteristic; this means that by implementing vecpar's abstract features, the user library releases the execution flow to the framework, which decides when and how to invoke the user code [Piętak and Kisiel-Dorohinicki, 2013]. The main goal of this approach is to ensure the decoupling between a scientific algorithm (which can be implemented by a domain expert), the way to compile and execute it on different hardware platforms (which is based on parallelization strategies developed by computer scientists) and the data storage location (which can be tailored for different scenarios), as shown in Figure 4.2⁵. Moreover, this separation of concerns contributes to a high degree of maintainability of the source code since (a) new parallel execution backends can be added without changing the client code that uses vecpar, and (b) within vecpar, adding new functionality is simplified due to the design's modularity.

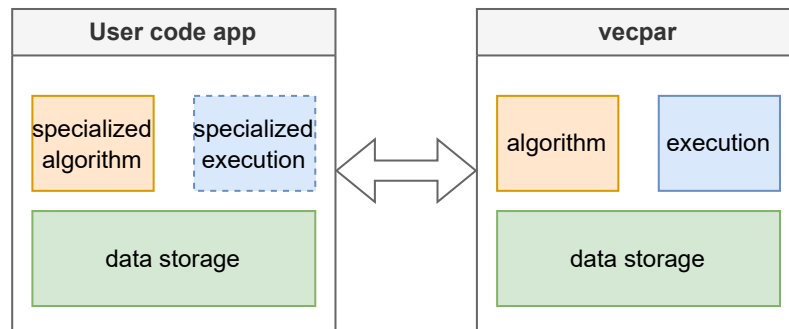


Figure 4.2.: High Level Design; the dashed line represents an optional module/choice

Vecpar has minimum dependencies: the vecmem library and the CUDA and OpenMP libraries and runtimes.

From the implementation perspective, vecpar relies on C++20 key features like *concepts* to avoid runtime polymorphism and replace it with a compile-time version of it whenever possible.

⁵This scheme is detailed later in Section 4.4 after all the modules are completely explained.

4.2.1. Execution Flow

Vecpar's execution flow is summarized in Figure 4.3. Assuming there is a user application that defines an algorithm by extending the *vecpar::algorithm* functionality, as suggested by the dashed arrow, run-time execution follows the numbered arrows. The entry point is the user code (1) that invokes *vecpar::parallel_algorithm*⁶ with a given algorithm implementation. The execution moves next into the *vecpar* API (2); since the algorithm is a *vecpar algorithm*, the framework handles parallel task distribution and memory transfers if needed (3), and then the execution moves back into the user code to perform the actual computations (4). The results are then available to the user code (5).

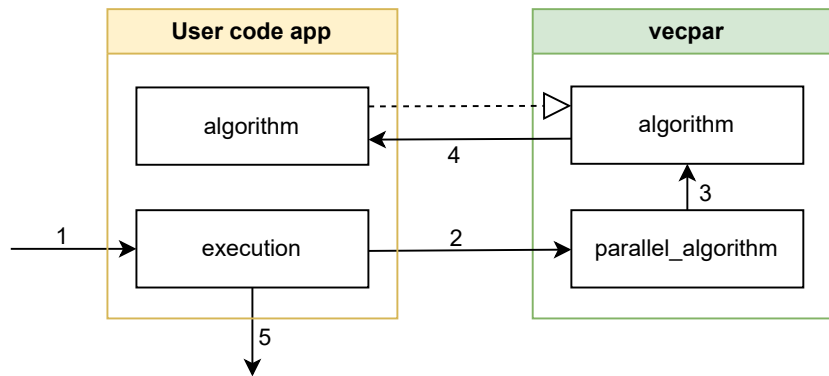


Figure 4.3.: Vecpar execution flow

4.3. Low-level Design

An in-depth view of the architecture based on the execution flow in Figure 4.3 is shown in Figure 4.4. The three key concepts here (the algorithm, the backend and the memory resource) and their interaction are exposed through the *vecpar* API, which will be covered next in this section.

Figure 4.4 also shows the automatic backend selection mechanism (detailed in Section 4.3.3.4). If the user does not explicitly require a specific backend by prefixing the function call accordingly, the decisional state machine passes through the depicted steps. Firstly, the *algorithm type*⁷ decides the data distribution mechanism between workers. Secondly, based on compilation flags, the *execution backend*⁸ employs the parallelization and/or offloading strategy. Lastly, in case a GPU backend was identified, the location of the data is checked to see if host-device transfers might be required; in case they are, *vecpar* will perform them automatically.

⁶A more specific invocation like `vecpar::parallel_map()` or `vecpar::cuda::parallel_algorithm()` can also be used for a specialized execution. These are detailed in the next section.

⁷Valid *vecpar::algorithms* are detailed in Section 4.3.2.

⁸The backends and the way to setup these configurations are detailed in Section 4.3.3.

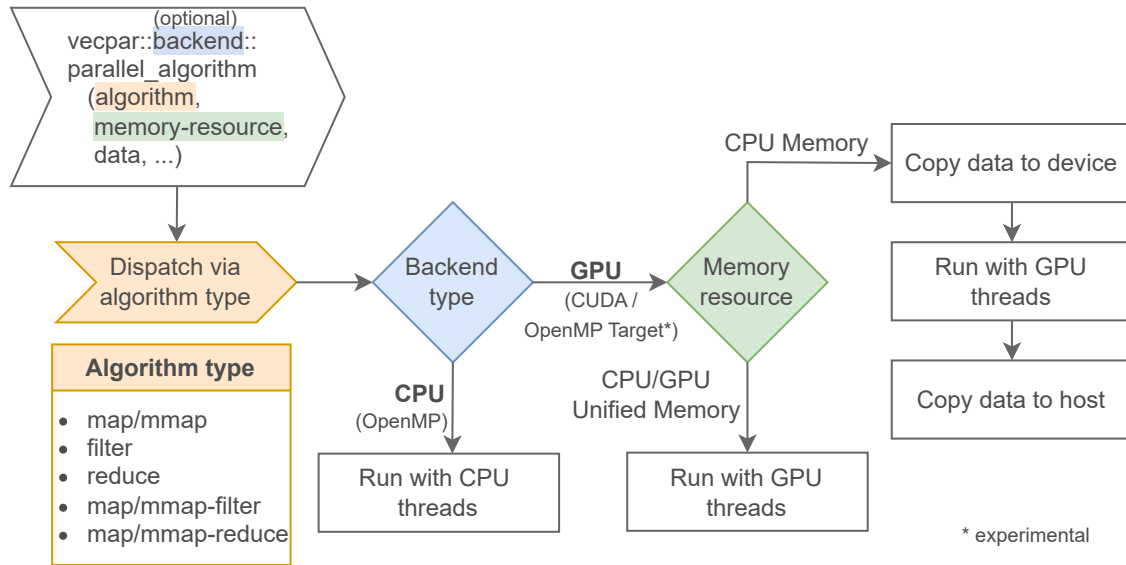


Figure 4.4.: Vecpar's components and decision flow

4.3.1. Data Types and Memory Resources

The list types that vecpar can parallelize across are the containers defined by the vecmem library, introduced in Section 3.1.1.1. `Vecmem::vector` extends `std::vector` so that it can be used in a GPU environment as well. `Vecmem::jagged_vector` is a 2-dimensional container currently being prototyped, which is an array of arrays of different sizes. Similar to the vectors exposed by the C++ Standard Library, vecmem containers are templated on the elements' type. To ensure a successful GPU allocation, the types must be default constructible.

The user can specify a *memory resource* for a vecmem container that is used to read the input data from and to store the result(s). This can be CPU memory or GPU memory, with the last one being handled either directly or through unified memory APIs. Vecmem wraps CUDA, HIP and SYCL allocators, and offers enriched proxies with extended functionality that even emulate dynamic memory allocations on device. Vecpar currently uses only the C++ and CUDA backends from vecmem, so the memory allocators that are used are represented by the `vecmem::host_memory_resource` class for the former, and `vecmem::cuda::managed_memory_resource` and `vecmem::cuda::device_memory_resource` for the latter. Vecmem also handles the memory deallocations automatically, when a variable goes out of scope.

The type `vecmem::device_vector` is the counterpart for a `std::vector` that allows modification of the vector's elements in device code; nevertheless it does not allow the vector to be resized nor to be allocated only on the device. Vecmem provides access to the data inside a container through objects defined in the `vecmem::data` namespace, which can either own the data (e.g. `vector_buffer` and `jagged_vector_buffer`) or not (e.g. `vector_view` and `jagged_vector_view`). Portability between host and device is

achieved using these notations as shown in Listing 4.1 and Listing 4.2⁹. While the former example is easier to implement, the second one most likely ensures better performance due to CUDA's way of handling read/write operations when managed memory is involved. Vecpar aims to hide all this complexity from the user and perform these steps automatically, behind the scene.

```

1 // init memory resource
2 vecmem::cuda::managed_memory_resource mm_mr;
3 vecmem::vector<int> host_vec(10, &mm_mr); // initialization ...
4 // get the view to the data
5 auto view = vecmem::get_data(host_vec);
6 // pass the view to the CUDA kernel
7 kernel<<<...>>>(view);
8 // inside the CUDA kernel code ...
9 vecmem::device_vector<int> device_vec(view);

```

Listing 4.1: CUDA kernel invocation with vectors allocated in CUDA managed memory

```

1 // init memory resource(s)
2 vecmem::host_memory_resource host_mr;
3 vecmem::cuda::device_memory_resource device_mr;
4 // copy tool which wraps cuda::memcpy calls
5 vecmem::cuda::copy copy;
6 vecmem::vector<int> host_vec(10, &host_mr); // initialization ...
7 // get the view to the data
8 auto view = vecmem::get_data(host_vec);
9 // copy host to device
10 auto vec_buffer = copy.to(view, device_mr, vecmem::copy::type::host_to_device);
11 // get the view to the buffer
12 auto vec_buff_view = vecmem::get_data(vec_buffer);
13 // pass the view to the CUDA kernel
14 kernel<<<...>>>(vec_buff_view);
15 // inside the CUDA kernel code
16 vecmem::device_vector<int> device_vec(vec_buff_view);
17 // copy device to host
18 copy(vec_buffer, host_vec, vecmem::copy::type::device_to_host);

```

Listing 4.2: CUDA kernel invocation with vectors allocated in CPU memory; data transfer to and from the device are mandatory in this case

For going further, the term *iterable* will be used to denote a `vecmem::vector` or a `vecmem::jagged_vector`.

4.3.2. Algorithms

Building on top of the equations defined in Section 4.1.2, vecpar exposes the notion of an *algorithm*, which wraps the *mapping_function*, the *filtering_function*, the *reducing_function*

⁹There is one important observation to note here: while lines 6 and 8 can be combined together, this is not possible for lines 8 and 10 due to the dependency in line 16.

and the optional *identity_function*. The available algorithms are briefly summarized in Table 4.1 and each will be detailed next in this section.

No	Simple	No	Composed
1	parallelizable_map	5	parallelizable_map_filter
2	parallelizable_mmap	6	parallelizable_mmap_filter
3	parallelizable_filter	7	parallelizable_map_reduce
4	parallelizable_reduce	8	parallelizable_mmap_reduce

Table 4.1.: Vecpar algorithm types

From the implementation perspective, these are C++ structures¹⁰ templated on several types that will be detailed for each algorithm next. The types are used to configure the behaviour for data distribution, parallelization and offloading strategies in the backends.

For notation purposes, the input data types will be denoted with *Iterable* < *Type*_{*i*} > with $i \in \overline{1, n}$ and their count is *n*, $n \leq 5$ and the result type is *Iterable* < *RType* > or *RType*; all iterables are colored in green, while parameters as their count is shown in yellow. The generic types that define the context are denoted *CType*_{*j*}, $j \in \overline{0, m}$, and colored in purple. In the function signatures, the **bold** notation stands for *mutable* parameters, while the others are considered constant. Also in the signatures, the qualifier `vecpar::TARGET` is a macro, which expands to different qualifiers based on the backend; a valid example is `__inline__` (or `__forceinline__` for CUDA), which can be added for performance considerations.

Since some of the eight algorithms are similar, they will be described together, in the following five sections.

4.3.2.1. Parallelizable_(m)map

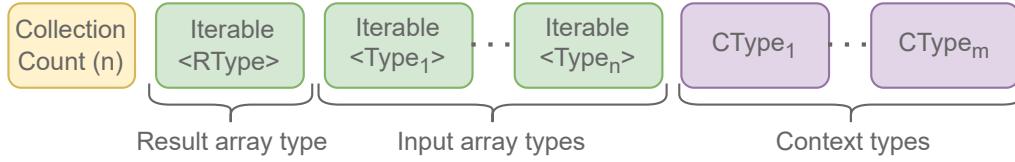
Parallelizable_map and *parallelizable_mmap* classes provide similar behaviour in the sense that they each define one member function, *mapping_function*, which will be invoked for every element of the input list(s) based on different parallelization strategies provided by the vecpar backends. The difference comes from the way they handle the data and store the result: while the former treats the input parameters as constant and returns a new iterable collection (allocated automatically by *vecpar*), the latter will treat the first collection as mutable and will therefore override it; the latter could be used as a performance optimization over the former in the case when the following conditions are simultaneously satisfied: (a) the return type of the map is the same as the type of the first input collection, and (b) the input data is safe to be modified (i.e. it will not generate race conditions for example).

The user code that uses these abstractions must extend *parallelizable_map* (or *paralleliz-*

¹⁰While for the basic map, filter and reduce operators the usage of a C++ structure could be redundant since each exposes only one member function, this design is needed for the composed operators map-filter and map-reduce, which inherit the behaviour from two different basic algorithms and therefore expose more member functions.

able_mmap) class and provide an implementation for its function¹¹.

This algorithm can support up to five¹² input collections (of identical size) that are iterated at the same time¹³. Also, these collections can be of different types (as long as they are *iterable*), as shown in Figure 4.5a and Figure 4.6a, which lead to the required signature for the *mapping_function* as shown in Figure 4.5b and Figure 4.6b.

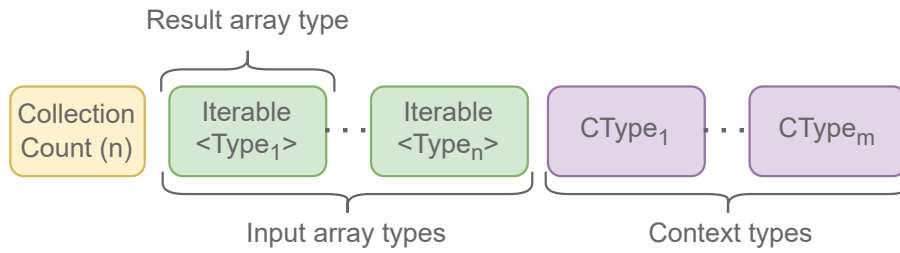


(a) Class template definition

TARGET **RType&** mapping_function(**RType&**, Type_1&, ..., Type_n&, CType_1&, ..., CType_m&);

(b) Function signature

Figure 4.5.: Parallelizable_map class templates and member function signature



(a) Class template definition

TARGET **RType&** mapping_function(**Type_1&**, ..., Type_n&, CType_1&, ..., CType_m&);

(b) Function signature

Figure 4.6.: Parallelizable_mmap class templates and member function signature

While the collection(s) type and their number are less important when the code targets a CPU resource, there are significant differences when it comes to transferring the data to/from a GPU; these are detailed in Section 4.3.3 since they are particular to each backend. Due to the need for further specialization, from the implementation's perspective, a different class is defined for each number of input collections, leading to having five base

¹¹In an early implementation of vecpar, the member function was marked *virtual*, but this restriction was later removed due to performance consideration when invoked on a GPU. Currently, if the implementing code does not obey the required signature, a compilation error is thrown. This observation of avoiding virtual functions is valid for the entire vecpar API, therefore it will be omitted while introducing the remaining algorithms.

¹²This is specified through the *collection count* parameter, which is implemented as an enumeration.

¹³A trivial yet very common use case, which uses several collections, is the vector addition example: $c[i] = a[i] + b[i]$, $i \in \overline{1, n}$.

templates. Nevertheless, this is hidden from the users of vecpar library as can be seen in Listing 4.3, which shows a simple implementation example.

```

1  template <class T>
2  struct vector_addition :
3  public vecpar::algorithm::parallelizable_mmap<
4      vecpar::collection::Three, // a,b,c
5      vecmem::vector<T>, // c
6      vecmem::vector<T>, // a
7      vecmem::vector<T>> // b
8  {
9      TARGET T& mapping_function(T& c_i, const T& a_i, const T& b_i) const {
10         c_i = a_i + b_i ;
11         return c_i;
12     }
13 };

```

Listing 4.3: Parallelizable_mmap implementation example

4.3.2.2. Parallelizable_filter

Parallelizable_filter algorithm allows the definition of a predicate (by providing an implementation to *filtering_function*), which is used to filter out elements of the input collection. Since the output result of filtering is another list of elements of the same type, the template required by the C++ classes is trivial, as shown in Figure 4.7. Filter never modifies the input collection and the results is always a new vector.



Figure 4.7.: Parallelizable_filter class templates and member function signature

An example of a trivial *parallelizable_filter* implementation that keeps only the even numbers in a list is shown in Listing 4.4.

```

1  struct vector_filtering :
2  public vecpar::algorithm::parallelizable_filter<
3      vecmem::vector<int> // input type
4  {
5      TARGET bool filtering_function(int& a_i) const {
6          return (a_i % 2) == 0;
7      }
8  };

```

Listing 4.4: Parallelizable_filter implementation example

4.3.2.3. Parallelizable_reduce

As the name suggests, *parallelizable_reduce* algorithm reduces the input list using the behaviour defined by a *reducing_function*. Similar to the previous case, only the list type is required for the parametrization since the result type can be inferred automatically. This is shown in Figure 4.8 below. If the identity element for the *reducing_function* is not 0 (which is the default for a number type), then the user should override *identity_function* as well and provide the correct value. Since the reducing operation is performed in parallel, the global result is protected by synchronization mechanism implemented in each backend to avoid race conditions; an example is the OpenMP implementation in Listing 4.10.



Figure 4.8.: Parallelizable_reduce class templates and member function signature

A trivial example of an algorithm, which adds together all the elements of a list, is shown in Listing 4.5. Here, the implementation of *identity_function* is optional. Similar to *parallelizable_filter*, *parallelizable_reduce* does not modify the input collection.

```

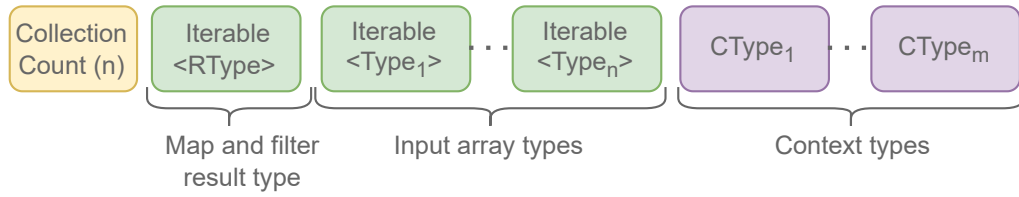
1 struct vector_reducing :
2 public vecpar::algorithm::parallelizable_reduce<
3   vecmem::vector<int> // input type
4 {
5     TARGET int* reducing_function(int* result, int& a_i) const {
6         *result += a_i;
7         return result;
8     }
9     // optional to override
10    TARGET int identity_function() const {
11        return 0;
12    }
13 };

```

Listing 4.5: Parallelizable_reduce implementation example

4.3.2.4. Parallelizable_(m)map_filter

Parallelizable_map_filter and *parallelizable_mmap_filter* extend either *parallelizable_map* or *parallelizable_mmap* class and *parallelizable_filter* class; therefore the implementing class inherits both templates and is required to provide implementation for two member functions, as shown in Figure 4.9 and Figure 4.10. Grouping a *map* and a *filter* together can bring performance optimization when the code is executed on a GPU, as detailed later in this chapter.



(a) Class template definition

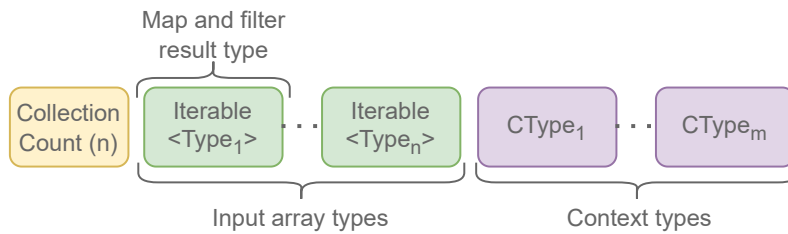
```

TARGET RType& mapping_function(RType&, Type_1&, ..., Type_n&, CType_1&, ..., CType_m&);
TARGET bool filtering_function(RType&);

```

(b) Functions signatures

Figure 4.9.: Parallelizable_map_filter class templates and member functions signatures



(a) Class template definition

```

TARGET Type_1& mapping_function(Type_1&, ..., Type_n&, CType_1&, ..., CType_m&);
TARGET bool filtering_function(Type_1&);

```

(b) Functions signatures

Figure 4.10.: Parallelizable_mmap_filter class templates and member functions signatures

A trivial example using map-filter behaviour is shown in Listing 4.6 where each element of an input vector of integers is multiplied by a double and only the even values are kept for the final result list.

```

1 struct vector_addition_filtering :
2 public vecpar::algorithm::parallelizable_map_filter<
3     vecpar::collection::One,
4     vecmem::vector<double>, // map result type
5     vecmem::vector<int> // input type
6 {
7     TARGET double& mapping_function(double& result_i, const int& a_i) const {
8         result_i = a_i * 1.0 ;
9         return result_i;
10    }

```

```

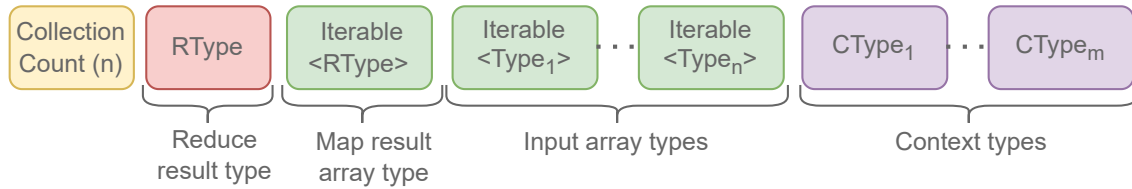
11
12     TARGET bool filtering_function(double& a_i) const {
13         return (a_i % 2) == 0;
14     }
15 };

```

Listing 4.6: Parallelizable_map_filter implementation example

4.3.2.5. Parallelizable_(m)map_reduce

Parallelizable_map_reduce and *parallelizable_mmap_reduce* extend either *parallelizable_map* or *parallelizable_mmap* class and *parallelizable_reduce* class; therefore the implementing class inherits both templates as shown in Figure 4.11a¹⁴ and Figure 4.12a. In this case, the user library must provide implementations for *mapping_function* and *reducing_function* (and *identity_function* if needed) as shown in Figure 4.11b and Figure 4.12b.



(a) Class template definition

```

TARGET RType& mapping_function(RType&, Type_1&, ..., Type_n&, CType_1&, ..., CType_m&);
TARGET RType* reducing_function(RType*, RType&);
TARGET Type_1 identity_function();

```

(b) Functions signatures

Figure 4.11.: Parallelizable_map_reduce class templates and member functions signatures

A sample implementation of a dot product operation between two vectors is shown in Listing 4.7.

```

1  template <class T>
2  struct vecpar_dot:
3      public vecpar::algorithm::parallelizable_map_reduce<
4          vecpar::collection::Two,
5          T, // reduction result
6          vecmem::vector<T>, // map result
7          vecmem::vector<T>, // a
8          vecmem::vector<T>> // b
9  {

```

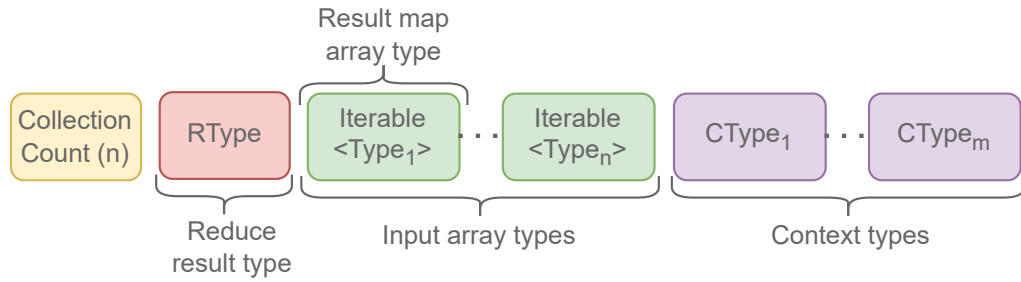
¹⁴Note that providing the map return type is mandatory since there is no way to identify whether the iterable is a 1D or a 2D vector since this is specified by the implementation.


```

10  TARGET T& mapping_function(T& result_i, const T& a_i, const T& b_i) const {
11      result_i = a_i * b_i;
12      return result_i;
13  }
14  TARGET T* reducing_function(T* result, T& crt) const {
15      *result += crt;
16      return result;
17  }
18  };

```

Listing 4.7: Parallelizable_map_reduce implementation example



(a) Class template definition

```

TARGET Type_1& mapping_function(Type_1&, ..., Type_n&, CType_1&, ..., CType_m&);
TARGET Type_1* reducing_function(Type_1*, Type_1&);
TARGET Type_1 identity_function();

```

(b) Functions signatures

Figure 4.12.: Parallelizable_mmap_reduce class templates and member functions signatures

4.3.2.6. Implementation Recommendations for Jagged Vectors

When implementing the *mapping_function*, *reducing_function* and *filtering_function*, the types in the functions' signatures are extracted from the lists template, as shown in the code snippets above. Nevertheless, when one of the input collection is a `jagged_vector<T>`, there is a technical challenge due to the fact that the parallelization goes one level in depth; in this case, the elements of a `vecmem::jagged_vector<T>` are of type `vecmem::vector<T>`. Since this is actually an `std::vector<T>` and C++ STL is not implemented for CUDA, the code will not compile with the default approach in Listing 4.8.

To work around this limitation while ensuring that the same C++ implementation works on both host and device, the recommended solution is to replace the type `vecmem::vector<T>` with `auto`; in this way, the function signature changes to the one in

Listing 4.9 (see parameter `zi`).

```

1  class jagged_algorithm :
2      public vecpar::algorithm::parallelizable_map<
3          Three,
4          /* result type of map */
5          vecmem::vector<double>,
6          /* input collections: */
7          vecmem::vector<double>,          // x
8          vecmem::vector<int>,             // y
9          vecmem::jagged_vector<float>,    // z
10         /* context parameters */
11         float > {
12     public:
13         TARGET double &mapping_function(double &result,
14             const double &xi,
15             const int &yi,
16             const vecmem::vector<float> &zi,
17             float &a) const {
18             result = a * xi + yi * zi[0];
19             return result;
20         }
21 };

```

Listing 4.8: `Parallelizable_map` implementation using a `jagged_vector` collection as one of the input lists; this will compile only for the CPU backend

```

1  TARGET double &mapping_function(double &result, const double &xi,
2      const int &yi, auto zi, float &a) const;

```

Listing 4.9: Function signature that ensures compilation for all backends

4.3.3. Execution Backends

As mentioned earlier, to make sure an algorithm is executed in parallel on a CPU or a GPU, a parallelization strategy is required. The strategies are implemented by different backends building on a common behaviour templated by the framework. Each backend must provide implementations for how to execute a `vecpar::algorithm`. For example, there are different strategies for executing a `parallel_map`, which works on independent and immutable data, and a `parallel_reduce`, which can compute partial results in parallel but which requires at least one global synchronization point at the end to aggregate the results; moreover, the CPU and the GPU reductions might have different approaches due to advantages and disadvantages of the hardware like the cache behaviour, the number of parallel threads or the available amount of memory per worker.

Following the concepts and algorithms introduced in Section 4.1.2 and in Section 4.3.2 respectively, each backend has to provide a way of executing each `vecpar` algorithm,

therefore there are 8 valid operators. For example, *parallel_map* provides a parallelization strategy for *parallelizable_map* algorithm which wraps the user-defined behaviour implemented in *mapping_function*. The parallel execution interface exposes the following functions (with the associated signatures):

1. *parallel_map* and *parallel_mmap*¹⁵ – This is the implementation of the *map* operator. The *Arguments* parameter can include more iterable collections and context information; the fact that the *algorithm* is a *parallelizable_map* or *parallelizable_mmap* is taken in consideration later on for a specific dispatch to the proper implementations.

```
template <class Algorithm, class MemoryResource,
          typename R = typename Algorithm::intermediate_result_t,
          typename T, typename... Arguments>
R &parallel_map(Algorithm &algorithm, MemoryResource &mr,
               /* optional */ vecpar::config config,
               T &data, Arguments &...args);
```

2. *parallel_filter* – This is the implementation of the *filter* operator.

```
template <class Algorithm, class MemoryResource, typename T>
T &parallel_filter(Algorithm &algorithm, MemoryResource &mr, T &data);
```

3. *parallel_reduce* – This is the implementation of the direct left-reduce operator.

```
template <class Algorithm, class MemoryResource, class R>
typename R::value_type &parallel_reduce(Algorithm &algorithm,
                                       MemoryResource &mr, R &data);
```

4. *parallel_map_filter* and *parallel_mmap_filter* – These are compound operations obtained by composing map and filter.

```
template <class Algorithm, class MemoryResource,
          class R = typename Algorithm::result_t,
          typename T, typename... Arguments>
R &parallel_map_filter(Algorithm &algorithm, MemoryResource &mr,
                     /* optional */ vecpar::config config,
                     T &data, Arguments &...args);
```

5. *parallel_map_reduce* and *parallel_mmap_reduce* – These are compound operations obtained by composing map and reduce.

```
template <class Algorithm, class MemoryResource,
          class R = typename Algorithm::intermediate_result_t,
          class Result = typename Algorithm::result_t,
          typename T, typename... Arguments>
Result &parallel_map_reduce(Algorithm &algorithm, MemoryResource &mr,
                          /* optional */ vecpar::config config,
                          T &data, Arguments &...args);
```

6. *parallel_algorithm* – A series of overloaded generic functions which dispatch to the correct implementation based on C++ concepts (at compile-time). There are eight

¹⁵*Parallel_mmap* is similar to C++20 STL function *std::for_each*, which works on iterators.

valid signatures corresponding to the eight vecpar operators described above. From the perspective of a scientific application which uses vecpar, this could be the *only* function needed; nevertheless, the above ones have to be provided by the backends.

```
template <class Algorithm, class MemoryResource,
         class T, typename... Arguments>
requires <concept>
R &parallel_algorithm(Algorithm algorithm, MemoryResource &mr,
/*optional*/ vecpar::config config,
T &data, Arguments &...args);
```

All these functions have similar parameters:

- a reference to the *vecpar::algorithm*
- a reference to the memory resource where the input data can be found (and where the result is going to be stored). At the moment, vecpar only supports *vecmem::host_memory_resource* and *vecmem::cuda::managed_memory_resource*; for the second one, a CUDA library must be accessible at compile and runtime.
- a configuration parameter which the user can specify tailored on the problem size. The *vecpar::config*¹⁶ parameter is a data structure containing numerical fields associated with the number of threads in a block (*m_blockSize*), the number of blocks in a grid (*m_gridSize*) and the size of the external shared memory allocated statically before a kernel execution on a GPU (*m_memorySize*). When the code targets a CPU, the number of OpenMP threads is computed by multiplying *m_blockSize* and *m_gridSize*. As a general recommendation, the user is advised not to use this option unless one has advanced information regarding the hardware configuration, the size of the problem in bytes and the data distribution patterns (which are particular to each vecpar backend).
- a number of references to the actual input data for the given algorithm. The reference count is variable but uniquely determined by the algorithm; for example, a *parallelizable_map* algorithm could receive up to five iterable collections and a variable number of contextual parameters while a *parallelizable_filter* algorithm can only receive one parameter, which is the iterable collection that needs to be filtered.

At the moment, vecpar fully supports two backends: OpenMP for CPU execution and CUDA for NVIDIA GPU. The OpenMP Target backend is experimental and provides limited functionality. There is also support for deciding at compile-time which backend to use based on specific flags set by the user. Next, we detail how these parallelization and offloading strategies are implemented in each of the backends and how the user can impact their execution.

¹⁶These are similar to CUDA's execution configuration parameters denoted by the <<< ... >>> notation.

4.3.3.1. OpenMP

For the *functions which implement the map operator*¹⁷, if the user passes a *valid configuration*¹⁸, the vecpar framework will try to start as many threads as the user requested. Nevertheless, the OpenMP runtime might limit this number to the value set by either an environment variable or to a compiler hard-coded limit (e.g. no more than 10 000 active threads). As discussed earlier in Section 3.1.2.1, starting more threads than the number of cores will likely bring no speedup (and can even induce slow-downs due to scheduling and context switching mechanisms). For the rest of the operators, there is no option to pass a configuration since filtering and reducing are more efficient when using a small number of parallel threads due to the linear dependency between the thread number and the count of critical regions or synchronization points, therefore this decision is left to the vecpar library.

The implementations for all the parallel strategies rely on the OpenMP pragmas described in Section 2.3.1. While *map* works on independent data which can be easily translated into a parallel loop, *filter* and *reduce* operations compute partial results in parallel, which are later aggregated inside critical regions. A short implementation of a core function used by the reduce operator is shown in Listing 4.10. Composed operators like map-filter and map-reduce are implemented as a chain of simple operators.

```

1  template <typename R, typename Function>
2  void offload_reduce(int size, R *result, Function f, vecmem::vector<R> &
   map_result) {
3      #pragma omp parallel
4      {
5          R *tmp_result = new R();
6          #pragma omp for nowait
7          for (int i = 0; i < size; i++)
8              f(tmp_result, map_result[i]);
9          #pragma omp critical
10         f(result, *tmp_result);
11     }
12 }
```

Listing 4.10: Implementation used by `vecpar::parallel_reduce` in OpenMP backend; *f* is the *reducing_function* provided by the user

Since the input data should be distributed to OpenMP threads, it is already directly accessible, no further transfers are required. The memory resource passed as argument is used to allocate the results, if needed¹⁹.

To access the parallel functions from the OpenMP backend, one must use the `vecpar::omp` prefix, as shown in Listing 4.11.

¹⁷`parallel_map`, `parallel_mmap`, `parallel_map_reduce`, `parallel_mmap_reduce`, `parallel_map_filter` and `parallel_mmap_filter`

¹⁸Not empty and non zero-ed

¹⁹In the case of mutable algorithms, this step is bypassed since the memory is already allocated.

```

1 vector_filtering algorithm;
2 vecmem::host_memory_resource mr;
3 vecmem::vector<int> vec(10, &mr);
4 // initialize vec ...
5
6 vecmem::vector<int> result =
7   vecpar::omp::parallel_algorithm(algorithm, mr, vec);

```

Listing 4.11: Example of using OpenMP backend assuming the *vector_filtering* algorithm defined in Listing 4.4

4.3.3.2. CUDA

Similar to the OpenMP backend, the CUDA backend offers the user the option to provide a configuration for the parallelization only for the operators which include a *map* operation. This configuration includes the number of threads in a block, the number of blocks in a grid and the size of the shared memory in bytes. If provided, vecpar will pass them to the CUDA kernel directly, therefore invalid or inefficient values translates into CUDA kernel launch errors or slow-running kernels, respectively.

While the CPU-targeting backend had a straight-forward implementation based on the OpenMP standard, the CUDA version is more complex, because besides providing the parallelization strategy, the operators must also ensure that the input data is accessible from the device at runtime and that the results are accessible to the host at the end.

In the current prototype, we assume the input data is either the CPU memory or the CUDA managed memory, while the results are expected to reside in the same location; this is achieved by implementing the parallelization strategies as common features but wrapped in two distinct modules²⁰, each handling different memory resources, while the dispatch decision is based on the reference type of the memory resource, at compile-time, as pictured in Figure 4.13.

The core functionality provides implementations for all vecpar operations, regardless of the memory location of the operands, by expecting `vecmem::data::views` instead of iterables input and output collections. These can come from extracting the data from either (a) vectors stored in managed memory or (b) buffers obtained after copying the data to the device.

There are 8 vecpar operations²¹, out of which 6 of them contain maps which could have up to 5 versions each (based on the number of the input iterables), and each iterable can be either vector or jagged_vector (which do not have a common C++ ancestor), and each function could receive a configuration or not, which leads us to 240 valid function signatures. Moreover, all have parameters packs already to allow customizable context arguments. To this number, we add the 4 valid reducers and filters (depending on the

²⁰Implemented as two C++ files: *managed_memory.hpp* and *host_memory.hpp*

²¹map, mmap, filter, reduce, map-filter, mmap-filter, map-reduce, mmap-reduce

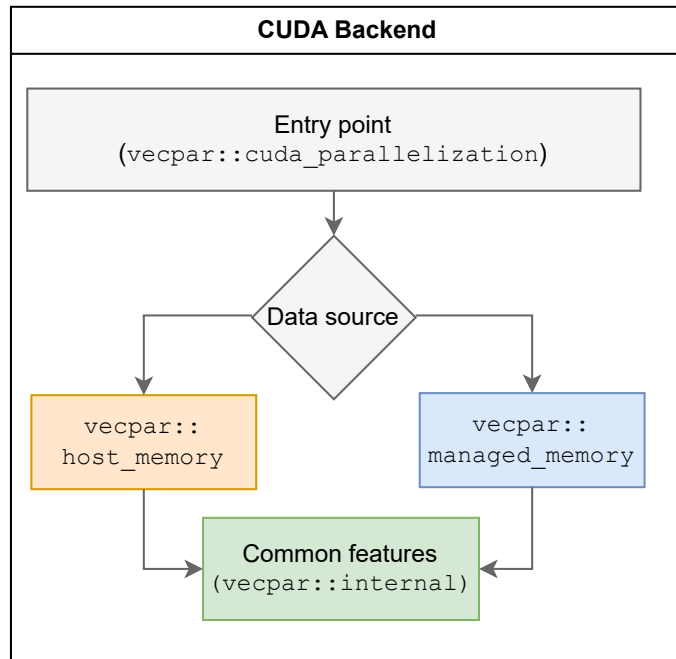


Figure 4.13.: CUDA backend implementation structure

iterable type). To avoid such a coding duplication effort, several wrapper functions were designed. The type of the iterable can be templated and automatically extracted at compile-time using member typedef types and concept features of the C++20 standard. A slightly more complicated scenario was to keep track of the memory copied to the device, by storing the references to the associated buffers in a local variable to avoid `vecmem` to release the memory before the kernel completion. The variable-arguments parsing functions are shown in Listing 4.12. The function at line 16 iterates through the incoming arguments (accepted by the matcher concepts defined above it) and returns either (a) a non-owning data container from a `vecmem::jagged_vector` (line 20), (b) a `vecmem::vector_buffer` obtained by copying the initial `vecmem_vector` to the device (line 25), or (c) the unchanged input argument if this is not an iterable (line 27).

```

1 // concept matcher for vecmem::vector<T> type
2 template <typename T>
3 concept Vector_type = std::same_as<T, vecmem::vector<typename T::value_type>>;
4 // concept matcher for vecmem::jagged_vector<T> type
5 template <typename T>
6 concept Jagged_vector_type = std::same_as<T,
7     vecmem::jagged_vector<typename T::value_type::value_type>>;
8
9 template <typename... T>
10 std::tuple<std::conditional_t<
11     (std::is_object<T>::value && Jagged_vector_type<T>),
12     vecmem::data::jagged_vector_data<value_type_t<T>>,
13     std::conditional_t<
14         (std::is_object<T>::value && Vector_type<T>),
15         vecmem::data::vector_buffer<value_type_t<T>>, T>>...>

```

```

16 get_buffer_of_copied_container_or_obj(T &...obj) {
17     return {([])(T &i) {
18         if constexpr (Jagged_vector_type<T>) {
19             auto data = vecmem::get_data(i);
20             return data;
21         } else if constexpr (Vector_type<T>) {
22             auto buffer = internal::copy.to(vecmem::get_data(i), internal::d_mem,
23                 vecmem::copy::type::host_to_device);
24             return buffer;
25         } else {
26             return i;
27         }
28     }(obj))...};
29 }

```

Listing 4.12: CUDA wrapper to handle a variable number of arguments of potentially different types

Having these in place, the common CUDA namespace implements 14 overloaded functions. Each one builds a lambda function that constructs a device vector from every `vecmem::data::view` and then it parallelizes across the iterables using the user-defined *mapping_function*, *filtering_function* or *reducing_function*. The lambda is passed on the device as a template parameter of a global kernel function similar to the one in Listing 4.13. After the execution, the results are then returned to the calling process on the host.

```

1 template <typename Function, typename... Arguments>
2 __global__ void kernel(const size_t size, const Function f, Arguments... args){
3     size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
4     if (idx >= size)
5         return;
6     f(idx, args...);
7 }

```

Listing 4.13: CUDA kernel example; f is the lambda function which wraps the user-defined behaviour

Similar to the OpenMP backend, the reduce and filter cases are more complex since there is dependency between threads. A code snippet from the CUDA implementation for reduce is shown in Listing 4.14. By default, the reduce configuration uses 32, 64 or 256 threads per block depending on the problem size, while the extern shared memory size (in line 6) depends on the size of R and the problem size. The for-loop on line 11 uses binary shifts operators which require the size of a block to be a multiple of 2 in order to perform the reduction correctly²². Synchronization mechanisms among the threads in a block and at the device's level are required to ensure the correctness of the result. The filter operation is implemented in a similar way with the difference that it uses an array for storing the temporary results for every block before the global synchronization point.

²²Since this is a mandatory requirement which comes with the implementation, the user does not have the option to configure the parameters in this case.

```

1  vecpar::cuda::rkernl<<<m_gridSize, m_blockSize, m_memorySize>>>(
2      lock, size,
3      [=] __device__(int *lock) {
4      vecmem::device_vector<R> partial_result(partial_result_view);
5
6      extern __shared__ char smem[];
7      R* temp = reinterpret_cast<R*>(smem);
8      size_t tid = threadIdx.x;
9      temp[tid] = partial_result[tid + blockIdx.x * blockDim.x];
10     size_t gidx = threadIdx.x + blockIdx.x * blockDim.x;
11     for (size_t d = blockDim.x >> 1; d >= 1; d >>= 1) {
12         __syncthreads();
13         /// for odd size and (larger) even number of threads, make sure
14         /// we do not read from outside of the array
15         bool within_array = ((tid + d) < blockDim.x) && (gidx + d < size);
16         if (tid < d && within_array) {
17             algorithm.reducing_function(&temp[tid], temp[tid + d]);
18         }
19     }
20     if (tid == 0) { /// only one thread in a block synchronizes the results
21         do {} while (atomicCAS(lock, 0, 1)); // acquire the lock
22         algorithm.reducing_function(result, temp[0]);
23         __threadfence(); // wait for write completion
24         atomicCAS(lock, 1, 0); // release the lock
25     }
26 }));

```

Listing 4.14: CUDA internal reducer

The composed operations *parallel_(m)map_reduce* and *parallel_(m)map_filter* implement a memory optimization if the initial data is in host memory, in the sense that the intermediate result (obtain by applying the first operation) is already accessible from the device and will not be re-transferred for the second operation.

From the point of view of a scientific application which calls *vecpar* to parallelize an algorithm, a sample invocation is shown in Listing 4.15. Here, the memory resource can also be replaced with CUDA managed memory, if the user wants to, but it is not mandatory since *vecpar* is able to transfer the memory behind the scenes.

```

1  vector_filtering algorithm;
2  vecmem::host_memory_resource mr;
3  vecmem::vector<int> vec(10, &mr);
4  /// initialize vec ...
5
6  vecmem::vector<int> result =
7      vecpar::cuda::parallel_algorithm(algorithm, mr, vec);

```

Listing 4.15: Example of using CUDA backend assuming the *vector_filtering* algorithm defined in Listing 4.4

Listing 4.15 is almost identical to Listing 4.11 with the only difference being the namespace invoked in line 7. If that is left out, a single-source implementation (which can run with either OpenMP threads or CUDA threads) is obtained; this is further detailed in Section 4.3.3.4.

4.3.3.3. OpenMP Target

In order to increase `vecpar`'s portability, the development of an OpenMP target backend has started, while contributions from people outside our research group was encouraged²³.

In comparison to the existing backends, the OpenMP target one is *experimental* since (a) a series of simplifications are assumed due to limitations in 3rd party libraries or compilers, (b) not the entire functionality is supported at the moment, and (c) some specific performance optimizations are prototyped for compilers that implement newer features of OpenMP 5.0 and 5.2. All these together with some technical implementation details are summarised next.

The OpenMP Target backend provides implementations for all the eight operators, but GPU performance optimizations are available only for a few *limited scenarios*²⁴ for the composed operators `parallel_(m)map_filter` and `parallel_(m)map_reduce`. This means that the framework will transfer the data to and from the device at the beginning and at the end of an algorithm, resulting in redundant copies which induce performance penalties.

A maximum of three input `vecmem::vectors` are supported as input/output containers for the `map`-related algorithms and operators. We are currently investigating ways of extending this without the need to duplicate the code. Also, no support for jagged vectors is implemented.

Due to the OpenMP target implementation in *some of the C++ compilers*²⁵ which requires that the number of threads in a block is a multiple of a warp²⁶ (or a wavefront²⁷), `vecpar` will not accept a configuration from the user for the map operations. The OpenMP runtime will decide the grid and block sizes for mapping, while similar to the previous backends, the filtering and reducing operations use a limited number of blocks (and a block size of 32 threads) for performance reasons.

Also, during implementation, we noticed many inconsistencies either between compilers or targeted platforms. For example, the `clang` compiler can automatically map non-trivial C++ objects to the GPU memory when used to build an executable for an NVIDIA GPU and it only shows a warning about it; whereas the `aomp` compiler (which is `clang`-based) will throw a compilation error requesting explicit mapping. A com-

²³The student Henning Lindemann has contributed to the development of OpenMP Target backend and its testing on AMD resources as part of his Bachelor project, under the author's supervision. Detailed contribution graph is available in Appendix A.

²⁴This includes a `parallel_map_reduce` operation with one input collection

²⁵For example: LLVM/clang and ROCm/AOMP

²⁶in NVIDIA terminology

²⁷in AMD terminology

mon ground was found by using the OpenMP basic functions like `omp_target_alloc`, `omp_target_memcpy` and `omp_target_free`. Nevertheless, by using this GPU-targeted features, the `vecpar` backend code can no longer be shared between CPU and GPU through the native fallback option of the `omp target` pragma. Despite the fact that the OpenMP 5.0 standard specifies mechanisms for customizing behavior based on architecture using variants and metadirectives (described in Section 2.3.1), most of the compilers do not support them yet. Therefore, we needed to rely on compilation flags provided by the user to choose between distinct implementations for the same operation; these are `COMPILE_FOR_DEVICE` and `COMPILE_FOR_HOST`. The first one is mandatory to be set in order to target a device while leaving it out or setting the second one will instruct `vecpar` to parallelize using OpenMP (CPU) threads (by delegating the execution to the OpenMP backend) even if a GPU is available. Nevertheless, the same C++ class of a scientific application, which invokes `vecpar::ompt::parallel_map` for example, can be part of two executables, each compiled with a different flag, as will be shown later in this chapter.

Building on clang's support for OpenMP 5.2 and more precisely on the shared memory allocators, `vecpar` has an optimized version of a *parallel_map* execution on GPUs when one iterable collection is provided; a code snippet is extracted in Listing 4.16. This kind of optimization proves its benefits when the input vector contains large objects, potentially misaligned in respect to the cache lines; using the shared memory will alleviate the uncoalesced accesses for read and write and therefore improve the performance. Nevertheless, the block size for this parallelization is defaulted to 32 threads.

```

1 Algorithm *d_alg = (Algorithm *)omp_target_alloc(sizeof(Algorithm), 0);
2 const int grid_size = (size + BLOCK_SIZE - 1) / BLOCK_SIZE;
3 // execution on device
4 #pragma omp target teams num_teams(grid_size) \
5   is_device_ptr(d_alg) \
6   map(to : d_data[0 : size]) \
7   map(from : map_result[0 : size])
8 {
9   value_type_t<T> inbuffer[BLOCK_SIZE];
10  #pragma omp allocate(inbuffer) allocator(omp_pteam_mem_alloc)
11  // thread 0 from each block loads data into shared memory
12  for (int i = 0; i < BLOCK_SIZE; i++) {
13    if (omp_get_team_num() * BLOCK_SIZE + i < size) {
14      inbuffer[i] = d_data[omp_get_team_num() * BLOCK_SIZE + i];
15    }
16  }
17  // allocate space in shared memory for the results
18  value_type_t<R> outbuffer[BLOCK_SIZE];
19  #pragma omp allocate(outbuffer) allocator(omp_pteam_mem_alloc)
20  // run in parallel
21  #pragma omp parallel num_threads(BLOCK_SIZE)
22  {
23    // all threads use the shared memory for computing the output result
24    if (omp_get_team_num() * BLOCK_SIZE + omp_get_thread_num() < size) {

```

```

25     d_alg->mapping_function(outbuffer[omp_get_thread_num()],
26         inbuffer[omp_get_thread_num()], rest...);
27 }
28 }
29 // thread 0 in each block copies the results from shared memory to global
    memory
30 for (int i = 0; i < BLOCK_SIZE; i++) {
31     if (omp_get_team_num() * BLOCK_SIZE + i < size) {
32         map_result[omp_get_team_num() * BLOCK_SIZE + i] = outbuffer[i];
33     }
34 }
35 }
36 // free the memory
37 omp_target_free(d_alg, 0);

```

Listing 4.16: Code snippet from the *parallel_map* implementation used when OpenMP 5.2 is supported

In short, line 1 allocates the algorithm pointer to the GPU memory; note that in this very particular case, there is no need for an explicit memory copy because the *vecpar::algorithm* is stateless. Lines 4-7 instruct the OpenMP runtime to create *grid_size* teams²⁸, to copy the input data *d_data* to the device, to allocate the space for the result *map_result* (and request a transfer to the host at the end of the target region) and to use the *d_alg* pointer which is already a device pointer so therefore no need for further transfers. Lines 9 – 10 and 18 – 19 ensure that temporary buffers are allocated in the team’s shared memory. Line 21 starts a parallel region of *BLOCK_SIZE* threads in each team, while the results are transferred from the shared buffers to the global memory in lines 30 – 34 (outside of the parallel region). While the shared memory is automatically deallocated when the variables go out of scope, the memory allocated explicitly must be released manually, as shown in the last line. Similar implementations are provided for the other parallel operations but are not listed here.

As it was shown above, the OpenMP target implementation can become complex and highly adapted to a specific platform in order to maximize the performance. Regardless of the backend complexity, calling *vecpar* from another project is still simple, as shown in Listing 4.17. Note the similarity between Listing 4.11 , Listing 4.15 and Listing 4.17.

```

1 vector_filtering algorithm;
2 vecmem::host_memory_resource mr;
3 vecmem::vector<int> vec(10, &mr);
4 // initialize vec ...
5
6 vecmem::vector<int> result =
7     vecpar::ompt::parallel_algorithm(algorithm, mr, vec);

```

Listing 4.17: Example of using OpenMP target backend assuming the *vector_filtering* algorithm defined in Listing 4.4

²⁸At this point, each team has only one thread

4.3.3.4. Backend Selection - Generic Dispatch

The option of explicitly invoking one of the backends is still important for the cases when the user knows that for a given problem, a specific implementation is either more efficient or the only one valid; an example could be reading files from the disk in a parallel loop; this only makes sense using OpenMP threads.

The vecpar API exposed for the single-source compilation for multiple backends (available in `vecpar/all/main.hpp`), provides two levels of delegation: first, overloaded versions of `vecpar::parallel_algorithm` dispatch the computation to the appropriate vecpar operators; second, the operator dispatches further to a given implementation (OpenMP CPU or NVIDIA CUDA) which is chosen at compile-time based on the associated compilation flags. A code snippet for each of these steps are shown in Listing 4.18 and Listing 4.19²⁹ respectively; similar implementations are provided for all the other (operator-backend) combinations.

```

1  template <class Algorithm, class MemoryResource,
2          class R = typename Algorithm::intermediate_result_t,
3          class T, typename... Arguments>
4  requires algorithm::is_map<Algorithm, R, T, Arguments...> ||
5          algorithm::is_mmap<Algorithm, R, Arguments...>
6  R &parallel_algorithm(Algorithm algorithm, MemoryResource &mr,
7          vecpar::config config,
8          T &data, Arguments &...args) {
9  return vecpar::parallel_map(algorithm, mr, config, data, args...);

```

Listing 4.18: Sample dispatch function based on algorithm type

```

1  template <class Algorithm, class MemoryResource,
2          class R = typename Algorithm::intermediate_result_t,
3          class T, typename... Arguments>
4  R &parallel_map(Algorithm &algorithm, MemoryResource &mr,
5          vecpar::config config,
6          T &data, Arguments &...args) {
7  #if defined(__CUDA__) && defined(__clang__)
8  return vecpar::cuda::parallel_map<Algorithm, R, T, Arguments...>(algorithm,
9          mr, config, data, args...);
10 #elif defined(_OPENMP)
11 return vecpar::omp::parallel_map<Algorithm, R, T, Arguments...>(algorithm,
12          mr, config, data, args...);
13 #endif

```

Listing 4.19: Sample dispatch function based on compilation flags

²⁹As mentioned in previous chapters, the LLVM/clang compiler can build NVPTX assembly from C++ natively; to be able to use this branch, a compiler with CUDA support has to be available and this is checked in line 7. Since most of the C++ compilers support OpenMP standard, there is no restriction on the compiler type for this backend, as shown in line 9.

The OpenMP target is not yet added to this dispatch system since it is still experimental. Nevertheless, once all the parallel operators are fully supported, to plug in a new backend is a trivial process and involves adding one `if-branch` statement which would delegate the call to the appropriate header file.

4.3.4. Lambda Functionality

By fitting a scientific problem into a `vecpar::algorithm`, the user passes the responsibility for parallelization and/or offloading to vecpar framework while no knowledge of multi-threading operations or GPU architecture is required. Nevertheless, if the user wants control over the way the memory is handled to potentially implement some optimizations that are customized for the given problem, vecpar offers the possibility to pass a lambda function which will be executed "as is" in a multi-threaded environment; this means that the user is responsible for memory transfers and synchronizations, if needed, therefore different lambdas need to be provided for CPU and GPU executors therefore losing the *single-source* benefit. An example of the vector addition problem mentioned earlier is shown in Listings 4.20-4.22. Since the CPU allocation and initialization for the arrays `*a`, `*b` and `*c` are identical in all cases, they are left out from the listings.

```
1 vector_addition<T> algorithm;
2 vecpar::parallel_algorithm(algorithm, memoryResource, *c, *a, *b);
```

Listing 4.20: Single-source user code when the algorithm is defined as a `vecpar::algorithm` in Listing 4.3; `memoryResource` is the CPU/C++ memory resource

```
1 using namespace vecmem::copy::type; // host_to_device / device_to_host
2 vecmem::cuda::device_memory_resource device_mr;
3 vecmem::cuda::copy copy;
4 // transfer data to device
5 d_a = copy.to(vecmem::get_data(*a), device_mr, host_to_device);
6 d_b = copy.to(vecmem::get_data(*b), device_mr, host_to_device);
7 d_c = copy.to(vecmem::get_data(*c), device_mr, host_to_device);
8 // offload lambda to CUDA kernel
9 vecpar::cuda::parallel_map(
10     array_size,
11     [=] __device__ (int idx,
12         vecmem::data::vector_view<T> &a_view,
13         vecmem::data::vector_view<T> &b_view,
14         vecmem::data::vector_view<T> &c_view) {
15         vecmem::device_vector<T> da(a_view);
16         vecmem::device_vector<T> db(b_view);
17         vecmem::device_vector<T> dc(c_view);
18         dc[idx] = da[idx] + db[idx] ;
19     },
20     vecmem::get_data(d_a),
21     vecmem::get_data(d_b),
22     vecmem::get_data(d_c));
```

```

23 // copy results to host
24 copy(d_a, *a, device_to_host);
25 copy(d_b, *b, device_to_host);
26 copy(d_c, *c, device_to_host);

```

Listing 4.21: CUDA user code for vector addition

```

1 vecpar::omp::parallel_map(array_size, [&] (int idx) {
2     c->at(idx) = a->at(idx) + b->at(idx);});

```

Listing 4.22: OpenMP user code for vector addition

4.3.5. Algorithm Chain

As mentioned earlier, a common use case in particle reconstruction software is to retrieve a data set (either simulated or read from the detector electronics) and pass it through consecutive steps of processing, transforming and filtering to obtain some meaningful physics results. Mathematically, this means composing together a series of functions and applying the resulting operator to the data set.

Vecpar implements this abstraction and provides a simplified way of *chaining* together multiple algorithms using a DSL-like API, through a *method chaining* approach based on the builder design pattern, where each function call returns the object class that it belongs to [Fowler, 2011]. The implementation uses the *FunctionComposition*³⁰ open-source project which builds on top of the C++19 functional and utility namespaces to compose an arbitrary number of `std::` functions.

The `vecpar::chain` class (partially listed in Listing 4.23) is templated on the memory resource, the input types for the first function in the chain, and the result type of the last function in the chain. The class exposes three functions which allow the user to provide (a) a configuration for the parallelization, (b) the list of algorithms to be chained, and (c) the input data for the execution.

```

1 template <class MemoryResource, class R, class T, class... Context>
2 class chain {
3     public:
4         // constructor
5         chain(MemoryResource&);
6         // optional method to provide a specific parallel configuration
7         chain& with_config(vecpar::config);
8         // method to provide algorithms (in a specific order)
9         template <typename First, typename... Rest>
10        chain& with_algorithms(First first_alg, Rest... rest_alg);
11        // execute the chain (run the composition on the data)
12        R execute(T& coll, Context &...rest);
13 };

```

Listing 4.23: Vecpar::chain class definition

³⁰<https://github.com/nestoroprysk/FunctionComposition>

Passing a `vecpar::config` is optional, otherwise `vecpar` will use the defaults for each backend. At least one `vecpar::algorithm` is mandatory for building the chain; `vecpar` will wrap the algorithm provided by the user in a `vecpar::parallel_algorithm` function call automatically which also handles memory transfers if needed; if the memory resource is the host one and the code is compiled for the CUDA backend, each algorithm will copy the input data to the device and copy the results to host. This means that there are some redundant copies of the intermediate results between the algorithm runs³¹. Calling the `execute` function on a `vecpar::chain` object triggers the invocation of the composed function on the data given as input arguments, and returns the result of the chain.

There are two restrictions. Firstly, the functions should be composable, namely the codomain of one function must match the domain of the other. Secondly, only the first function can support a variable number of context parameters while the other functions must accept as input only the output of the previous invocation. For example, a two-step chain made of a `parallelizable_map_filter` and `parallelizable_map_reduce`, each working on one `vecmem::vector<double>` argument, is a valid composition; on the other hand, just switching the order of the algorithms, an invalid chain is obtained (e.g. a map-filter cannot work on the result of a reduce operation which is a single object instead of an iterable) and therefore, a compilation error will be shown.

A sample code that uses the chain functionality which adds together two arrays, element by element, and then reduces the resulting array, is shown in Listing 4.24.

```

1  vecmem::host_memory_resource memoryResource;
2  // allocate the vectors
3  vecmem::vector<int> a(N, &memoryResource); // N = size of the vector
4  vecmem::vector<int> b(N, &memoryResource);
5  vecmem::vector<int> c(N, &memoryResource);
6  // ... initialize the vectors
7  vector_addition<int> addition_alg; // mmap c[i]=a[i]+b[i]
8  vector_reducing<int> reducing_alg; // reduce r=c[0]+..+c[N]
9  // define the chain
10 vecpar::chain<vecmem::host_memory_resource,
11      int, // result of the chain
12      vecmem::vector<int>, // c
13      vecmem::vector<int>, // a
14      vecmem::vector<int> // b
15      > chain(memoryResource);
16 // compose the algorithms and get the results
17 int result = chain
18     .with_config({10,32}) // optional
19     .with_algorithms(addition_alg, reducing_alg)
20     .execute(c,a,b);

```

Listing 4.24: Chain use case; *vector_addition* and *vector_reducing* algorithms are defined in Listing 4.3 and Listing 4.5

³¹A prototype that addresses this shortcoming is described in Section 4.3.5.1.

4.3.5.1. CUDA chain

As a performance optimization, `vecpar` provides a specific chain implementation for the case when the following preconditions are simultaneously satisfied: first, the input iterable for the chain is one `vecmem::vector` (and not `vecmem::jagged_vector` nor several vectors) allocated in the CPU host memory resource, and second, the code is compiled for NVIDIA GPU offloading using the `vecpar` CUDA backend.

In this case, a custom implementation of the `vecpar` operators is provided; it uses raw pointers³² allocated directly in the device memory as the result of each operation, which is passed as input further to the next function call. Consequently, no CPU-GPU transfers for intermediate results are done and only the final result is then copied to the host. Moreover, all the required data transfers are hidden from the users and are done automatically by the `vecpar` framework.

4.4. Compilation for Multiple Targets

There are two ways to compile a C++ code base (which uses the `vecpar` framework) for different architectures: using the (native) single-source functionality and using the `ompt` backend; these will be described next. Regardless of the approach, the `vecpar` library has to be installed locally. OpenMP and CUDA/ROCm runtimes must also be available. `Vecpar` provides easy integration with `cmake` system as shown in Listing 4.25. For the ARM platform, the `cmake` is instructed to use a different default search path for the system libraries.

```
1 find_package(vecpar REQUIRED 0.0.3)
```

Listing 4.25: Find local installation using `cmake`

4.4.1. Native Single-source Functionality

This approach means to define an algorithm as a `vecpar::algorithm` and to call a parallel executor by invoking `vecpar::parallel_algorithm`. Since at the moment only the `LLVM/clang` compiler can build C++ code into NVPTX, it is the only compiler that can be used.

Assuming the C++ files are defined as specified above (i.e. an algorithm as in Listing 4.4 passed to a `parallel_algorithm()` function like in Listing 4.18) and they are added to a `cmake` target named *example*, the Listing 4.26 and Listing 4.27 below show a sample configuration for targeting a CPU and an NVIDIA V100 GPU, with the same implementation. In the case of the CPU target, the dependency to `vecmem::cuda` is only needed if CUDA managed memory is the default memory resource used by the code. Line 3 (in both listings) shows the dependency to either the OpenMP runtime or the CUDA runtime respectively. Additionally, the GPU target requires the compilation options in lines 5-8, which instruct `clang` to generate NVPTX optimized for the V100 (SM_70).

³²The `vecmem::device_vector` is not used in this case.

```

1 target_link_libraries (example
2   vecmem::core vecmem::cuda
3   vecpar::all OpenMP::OpenMP_CXX)

```

Listing 4.26: CPU target

```

1 target_link_libraries (example
2   vecmem::core vecmem::cuda
3   vecpar::all CUDA::cudart)
4
5 target_compile_options (example
6   PUBLIC
7   $<$<COMPILE_LANGUAGE:CXX>:-x cuda
8   --offload-arch=sm_70>)

```

Listing 4.27: GPU target

4.4.2. Ompt Functionality

This approach requires the definition of an algorithm as a *vecpar::algorithm* and a call to *vecpar::ompt::parallel_algorithm*. Clang, gcc, icc, aomp, and hipcc compilers support OpenMP target features at different levels, so either of them can be used for this. The compiler must be built with offloading support, which usually requires several extra steps in the installation and configuration process. Nevertheless, some of them have automated this process through package managers like Spack³³.

Sample configurations when using the clang compiler to build executables for CPU x86_64 only, NVIDIA V100 and AMD Radeon 6600 are shown in Listing 4.28, Listing 4.29, and Listing 4.30 respectively³⁴.

```

1 target_compile_options(example PUBLIC $<$<COMPILE_LANGUAGE:CXX>:-fopenmp>)
2 target_compile_definitions(example PRIVATE COMPILE_FOR_HOST)
3
4 target_link_libraries(example vecpar_ompt OpenMP::OpenMP_CXX)
5 target_link_options(example PRIVATE -fopenmp -fopenmp-targets=x86_64)

```

Listing 4.28: CPU x86_64 target

```

1 target_compile_options(example PUBLIC
2   $<$<COMPILE_LANGUAGE:CXX>:-fopenmp -fopenmp-targets=nvptx64>)
3 target_compile_definitions(example PRIVATE COMPILE_FOR_DEVICE)
4
5 target_link_libraries(example vecpar_ompt OpenMP::OpenMP_CXX)
6 target_link_options(example PRIVATE -fopenmp -fopenmp-targets=nvptx64)

```

Listing 4.29: NVIDIA V100 target

```

1 target_compile_options(example PUBLIC
2   $<$<COMPILE_LANGUAGE:CXX>:-fopenmp-targets=amdgc -amd-amdhsa
3   -Xopenmp-target=amdgc -amd-amdhsa -march=gfx1030>)

```

³³The command to install the latest GCC compiler with OpenMP target offload for NVIDIA hardware: *spack install gcc +nvptx*. Nevertheless, there isn't such an easy installation path for AMD devices.

³⁴LLVM/clang 16 already prototypes an easier integration with cmake by automatically detecting the platform when setting *-offload-arch=native*, which would replace the need for using specific values for each platform.

```
4 target_compile_definitions(example PRIVATE COMPILE_FOR_DEVICE)
5
6 target_link_libraries(example vecpar_ompt OpenMP::OpenMP_CXX)
7 target_link_options(example PRIVATE -march=gfx1030
8     -fopenmp-targets=amdgcN-amd-amdhsa -Xopenmp-target=amdgcN-amd-amdhsa)
```

Listing 4.30: AMD Radeon 6600 target

In Listing 4.28, setting the flag `COMPILE_FOR_CPU` is redundant (but added here for more clarity); on the other hand, setting `COMPILE_FOR_GPU` flags in the other two listings is mandatory.

An important observation to make here is that different compilers use different names for similar flags. For example, to force OpenMP target to build only the CPU variants (without the GPU ones), the `-fopenmp-targets=x86_64` configuration has to be made for the clang compiler, while the gcc compiler requires `-foffload=disable`. This will be made easy in future clang releases, since version 16 already exposes an experimental `FindOpenMPTarget.cmake` file that is able to find and set the appropriate flags automatically by inferring the details about the platform.

4.4.3. Conclusion

The first approach has the advantage that it targets NVIDIA hardware using native code and therefore maximizes the performance, while the main disadvantage is that it has limited portability (e.g. x86, ARM and NVIDIA GPU). Whereas the second approach increases portability to include AMD GPU but might not ensure top performance since it is not using vendors' native APIs. Nevertheless, LLVM has ongoing development to support AMD and Intel GPUs through OpenMP target backend; when this functionality will be available, the scientific code that uses `vecpar` will just need to be recompiled with the latest compiler and the appropriate compilation flags for each platform, and the executables will be portable without any further code changes in the C++ files.

4.5. Performance Optimizations

The focus of the current work was to provide the optimizations that are connected to the library itself, while platform-specific optimizations are left for future work. While all optimizations are focused on reducing the execution time on GPU, some of them can be beneficial for CPU execution as well, but the impact is limited.

Firstly, `mmap` functionality reuses the memory allocated for one of the input collections as for the result; in this way, avoiding to allocate, deallocate and copy (bi-directionally) a chunk of memory could reduce the execution time of an algorithm substantially if the output list size in bytes is large enough. As mentioned in previous chapters, memory transfers are expensive due to much smaller bandwidths of the PCIe in comparison to on-board transfers; therefore, by avoiding some redundant ones, is highly desirable.

Secondly, the benefit of implementing an algorithm as a *parallelizable_map_reduce* instead of a *parallelizable_map* followed by a *parallelizable_reduce* is to avoid copying the intermediate results from the GPU to the CPU and then back to the GPU for the second algorithm. Similar to the previous case, this can speedup the total execution time by reducing the time needed for data transfers (since the data is already on device).

Thirdly, the chain functionality described above also brings a significant speedup when a series of algorithms that work on a list stored on the host is offloaded to a GPU. The potential of this optimization is detailed in Chapter 6.

Finally, using compile-time polymorphism wherever is possible means less decisions at runtime, which can also increase the performance.

4.6. Automated Tests

The vecpar framework currently has a few hundred automated tests written using GoogleTest³⁵ infrastructure. There are test cases that cover (a) APIs coming from a single backend (like the tests in the subfolders *omp*, *cuda* and *ompt*), (b) single-source cross-compilation (e.g. *single_source* subfolder covers different scenarios of *vecpar::algorithms* using generic backend with automatic dispatch for appropriate implementation) and (c) less common but accepted scenarios (e.g. *hybrid* subfolder tests an implementation of the *omp* backend using *vecmem::cuda::managed_memory_resource*). Different problem sizes are evaluated using vectors between ten and one million elements, in single and double precision, including testing edge cases like vectors of *prime numbers size*³⁶.

Benchmarks that compare native CUDA and OpenMP implementation of trivial examples are also implemented in order to evaluate the overhead of the library in comparison to native implementations. For the OMPT case, the GPU tests can run on either NVIDIA or AMD hardware, the only change required is the appropriate compilation flags and the availability of the vendor's runtime libraries.

More about our experimental setup and evaluation results are detailed in Chapter 6.

4.7. Summary

The vecpar framework offers a functional approach on parallelism by using functions as first class citizens; this allows easy composition and behaviour encapsulation. By leveraging C++ features like templated concepts and const-correctness, vecpar ensures compile-time polymorphism and thread-safety mechanisms, which are vital to concurrent execution, especially on GPUs. In addition, domain scientists can develop their code in C++ using vecpar abstractions without any previous knowledge of hardware architectures or other dedicated programming languages.

³⁵<https://github.com/google/googletest>

³⁶These are especially relevant for the complex CUDA reducers.

5. Related Work

Having the same single-source implementation to target multiple architecture is an approach explored in many ways. While the standards and APIs mentioned in Chapter 2 aim to achieve this through language extension and/or compiler support, there are also other possible ways. In this chapter, related work to `vecpar` is summarized and compared against the present solution. While the focus is on similar heterogeneous C++ APIs, other automatic parallelization and offloading solutions are briefly mentioned together with related heterogeneous software in particle reconstruction domain.

5.1. Heterogeneous Frameworks

Since `vecpar` is a framework expressed as a header-only library, it shares some features with other similar heterogeneous approaches like *Kokkos*, *RAJA*, *Alpaka*, *Charm++* and even *OpenMP*.

5.1.1. Kokkos

Kokkos is a “portability framework that provides abstractions for parallel execution in the form of a header-only library, a set of mathematical kernels, profiling and debugging tools”¹, available as open-source software [Edwards et al., 2014, Trott et al., 2022]. Initially released in 2014, Kokkos has evolved significantly in recent years through the effort of a large community mostly based around research facilities in the United States. Version 3 was released in 2021 and it is the one covered in this section.

Similar to `vecpar`, Kokkos provides a series of execution and memory abstractions to ensure performance portability. These are briefly described next.

Execution spaces provide a description of the location where the code will be executed, more precisely which core, NUMA node, GPU, etc. Supported backends include Serial, OpenMP, Cuda, HIP, OpenMPTarget, HPX, Threads and SYCL, with the remark that not all the functionalities are available for all backends. An application must decide which backend to choose at compile-time, by setting the variable `KOKKOS_DEVICES`. *Execution patterns* define independent units of work. Examples include `parallel_for`, `parallel_reduce` or `parallel_scan`, which can be called with functors or lambdas as arguments; `vecpar` has a similar feature but using an extended set of functions, which include groups like `map-filter` or `map-reduce`. These can bring an advantage when

¹<https://github.com/kokkos>

run on a GPU since two intermediate (and redundant) memory transfers² are avoided. Nevertheless, Kokkos release 3.0 introduced advanced reductions and improved locking mechanisms; while this brings performance improvements, the code becomes more verbose for programmers, as shown in Listing 5.1, which might decrease productivity.

```

1 double min; // the result
2 Kokkos::Min<double> min_reducer(min); // the pre-defined reducer
3 Kokkos::parallel_reduce( "MinReduce", N,
4   KOKKOS_LAMBDA (const int& x, double& lmin) {
5     double val = (1.0*x- 7.2) * (1.0*x- 7.2) + 3.5;
6     min_reducer.join(lmin, val);
7 }, min_reducer);

```

Listing 5.1: Compute the minimum value (*min*) from processed element of an array of size *N* using Kokkos pre-defined reducers; Source: Kokkos Programming Guide

Execution policies configure how a pattern is executed and connects the kernel with work items and execution spaces through scheduling mechanisms. Kokkos provides support for multidimensional ranges and hierarchical parallelism, which are not yet supported by *vecpar*.

Memory spaces is the abstraction for *memory resources*. Examples for NVIDIA hardware include *CudaSpace*, *CudaUVMSpace* and *CudaHostPinnedSpace*, which encapsulate the paged, unified and pinned resources respectively. *Memory layouts* provide the mapping between multidimensional array indices and the storage locations. Examples include *LayoutLeft* and *LayoutRight*, which describe the *column-major* and *row-major* representations. *Memory traits* add additional information about the memory; for example, if the access is atomic or not. *Views* connect all the memory abstractions together; they are templated on the (memory) space, layout and trait and behave as a multidimensional shared pointers. Kokkos is shipped with an entire library for containers. In comparison, *vecpar* uses the *vecmem*³ library, which has similar functionality but having the memory layout modeled after the EDM in particle physics. Kokkos views are more generic than *vecmem* views due to the fact that they can support n-dimensions, configured at compile-time.

To summarize, Kokkos provides more complex abstractions than *vecpar*, which allow a detailed configuration of computing and memory resources. Also Kokkos supports architecture-specific features like SIMD execution for CPU or CUDA execution graphs on NVIDIA GPU. Nevertheless, they come with two trade-offs: first, there is a much larger learning curve for Kokkos than for *vecpar* and second, highly configured executions are neither single-source nor portable. This is highlighted in Listing 5.2; here the data type is templated on the memory space, so this will not compile for an AMD GPU.

```
Kokkos::View<int **, Kokkos::LayoutLeft, Kokkos::CudaUVMSpace> A("A", 1, 2);
```

Listing 5.2: Define a 2D vector named "A", with 1 row and 2 columns, stored in CUDA unified memory in column-major representation

²The results from *map* are copied to the host and then copied back to the device for *reduce*

³*vecmem* library was introduced in Section 3.1.1.1.

5.1.2. RAJA

Another open-source alternative, called RAJA, was proposed by Lawrence Livermore National Laboratory [Beckingsale et al., 2019]: a C++ portability layer that ensures single-source compilation for multiple architectures. Similar to Kokkos and *vecpar*, RAJA is also an open-source project⁴.

There are a few features that RAJA and *vecpar* share: first, RAJA allows incremental adoption with platform specific data and parallelization strategies isolated from the scientific code; second, RAJA was designed with the goal of increasing productivity by ease-of-use; third, the portability is covered through platform-specific backends. At present, supported backends include *sequential*, *SIMD*, *OpenMP* for CPU platforms and *CUDA* and *HIP* for GPU, with *TBB* and *OpenMP_target* as experimental⁵ projects. The user code must decide which backend to use at compile-time.

Similar to Kokkos, RAJA defines its own abstractions. *Execution policies* configure how a loop will run. Each policy defines 3 components: a *policy type*, which is the execution backend: sequential, OpenMP or CUDA; a *launch category*, which configures how the code is executed, whether synchronously or asynchronously; and an *execution platform*, which defines where the code is executed in terms of memory spaces. Hierarchical parallelism is ensured by nested policies. Moreover, the user can also define custom policies if the default ones are not covering all the use cases. *Iteration spaces* describe the data distribution model and have two potential approaches: *segments* containing a set of loop indices to be executed in a unit or *index set* a container of arbitrary segments used mostly for hierarchical parallelism. The latter is also an optimization setup for sparse data sets. *Execution templates* define an operation on a kernel based on an execution policy and an iteration space. Examples include *forall*, *kernel*, *scan*, 5 custom reductions and atomics. *Views* represent device-agnostic data abstractions and multi-dimensional access patterns. RAJA views do not manage memory allocations, covering only pointer arithmetic. The user-code is responsible to make sure that a pointer is accessible on the device at runtime. Therefore, the developers still need to be familiar with CUDA and/or HIP memory management API. This is one major difference from *vecpar*, which handles memory operations automatically and hides the complexity from the user. RAJA tries to address this gap by providing additional libraries⁶ to wrap specific implementations into more generic and portable versions. Nevertheless, calling this intermediate layer is also the user's responsibility.

The authors published evaluation results using RAJA Portability suite, which is a set of software abstractions that enable portable parallel numerical kernel execution for physics use cases ranging from hundred of thousands to over one million lines of code that use MPI for distributing computations accros the nodes and RAJA for fine-grained parallelism

⁴<https://github.com/LLNL/RAJA>

⁵Not all the features are currently supported

⁶An example is CHAI library: <https://github.com/LLNL/CHAI>

within the nodes [Beckingsale et al., 2019]. Three backends (sequential, OpenMP for CPU and CUDA for GPU) were compared with native implementations. In 55% of the cases, RAJA performs within a 10% variation from the baselines, while in 48% of the cases, it performs better at least on one backend. Overall, speedups of $9\times$ – $17\times$ were observed for specific scenarios.

To conclude, RAJA API, which has been evolving for 10+ years, can provide portability and productivity with some known penalty costs in performance.

5.1.3. Alpaka

An Abstraction Library for Parallel Kernel Acceleration (ALPAKA) is a C++ header-only library initially released in 2016, which is still being developed and extended today as an open-source project⁷ [Zenker et al., 2016]. Moreover, the main repository offers an automatic translation tool from CUDA to Alpaka, called *cupla*⁸.

Similar to CUDA, Alpaka has four parallelization levels: *grid*, *block*, *thread* and *element*, which map to three memory types: *global*, *shared* and *register*. Algorithms expressed as functors or lambdas can be offloaded to hierarchical parallelization levels on different architectures.

Currently supported backends include OpenMP, TBB, `std::thread`, `Boost::fiber` for CPU and OpenMP target, CUDA and HIP for GPU; also, Alpaka has experimental support for OpenACC. In contrast to Kokkos, it allows multiple backends simultaneously in the same source file while the final decision about the execution platform can be delayed until runtime.

Alpaka provides a set of abstractions that allow a single-source implementation to be compiled for different targets, but in a distinct way when compared to *vecpar*. First, since Alpaka is inspired from CUDA programming syntax, it exploits parallelism through abstractions like *kernel indexing*, *work sharing* or *work queues*, which therefore are very different to the abstractions defined by *vecpar*. Second, while the user-code is responsible for data availability on device at runtime, *vecpar* hides this complexity in order to increase productivity. Similar to RAJA, external libraries like *llama*⁹ can be used to overcome this limitation. Third, to ensure both task and data parallelism, Alpaka does not automatically decompose the algorithmic execution domain or data domain, but it is the user-code's responsibility to do so; while this offers increased flexibility for the developers, which could translate into better runtime performance, it requires more GPU knowledge. The required code complexity can be seen from a simplified example¹⁰ of invoking a "Hello World" kernel, shown in Listing 5.3; here the backend is initially chosen in line 2, the communication queue with the device is set to be a *blockingqueue* in lines 3–4, the distribution of the data is handled by the *WorkDivMembers* defined in line 6 without vectorization

⁷<https://github.com/alpaka-group/alpaka>

⁸<https://github.com/alpaka-group/cupla>

⁹<https://github.com/alpaka-group/llama>

¹⁰<https://github.com/alpaka-group/alpaka/blob/develop/example/helloWorld>

support (line 10) and fully configured in lines 12 – 14; the execution is triggered in line 17 while the results are available after line 18, which waits for the queue’s completion. Finally, while one C++ file can be compiled for different architectures, Alpaka requires distinct kernel implementation and distinct memory accessibility patterns for each platform; otherwise, the performance is not guaranteed: “It is possible to write a single source kernel that performs well on all tested Alpaka backends without a drop in performance compared with the native implementation. In order to reach this performance, the developer needs to abstract the access to data, optimize the work division and consider cache hierarchies” [Zenker et al., 2016]. In the performance tests to evaluate the overhead in the same paper, two kernels were implemented, one to mirror OpenMP (for CPU) and the other one for CUDA (on GPU); when reusing the same kernel, the performance was very poor.

```

1  using Idx = std::size_t;
2  using Acc = alpaka::AccCpuSerial<Dim, Idx>;
3  using QueueProperty = alpaka::Blocking;
4  using Queue = alpaka::Queue<Acc, QueueProperty>;
5  using Vec = alpaka::Vec<Dim, Idx>;
6  using WorkDiv = alpaka::WorkDivMembers<Dim, Idx>;
7
8  auto const devAcc = alpaka::getDevByIdx<Acc>(0u);
9  Queue queue(devAcc);
10 Vec const elementsPerThread(Vec::all(static_cast<Idx>(1)));
11 Vec const threadsPerGrid(Vec::all(static_cast<Idx>(8)));
12 WorkDiv const workDiv = alpaka::getValidWorkDiv<Acc>(devAcc,
13     threadsPerGrid, elementsPerThread, false,
14     alpaka::GridBlockExtentSubDivRestrictions::Unrestricted);
15
16 HelloWorldKernel helloWorldKernel;
17 alpaka::exec<Acc>(queue, workDiv, helloWorldKernel);
18 alpaka::wait(queue);

```

Listing 5.3: Configure serial execution on CPU, using a blocking communication queue with the accelerator (CPU in this case), in a vectorized execution

In summary, while Alpaka offers an increased level of portability, the development productivity is highly impacted by (a) the need to handle memory management, (b) complex abstractions and (c) required kernel duplication to match native performance.

5.1.4. Charm++

Charm++ is an “object-oriented asynchronous message passing parallel programming paradigm” [Acun et al., 2014], which gets shipped with a dedicated runtime and set of tools to enable C++ developers to easily write parallel code. While it has no offloading support, it is worth mentioning it here due to its novel mechanism of approaching automatic parallelizations: a program is divided into logical collections of objects that interact

with each other, called "chares". A chare is a stateful object that can: (a) contain data, (b) send and receive asynchronous messages to other chares, and (c) execute a task. The user has no need to configure parallelization strategies, since the Charm++ Runtime makes all the decisions regarding resource allocation and scheduling.

5.1.5. OpenMP

Through the accelerator support introduced 10 years ago, OpenMP also qualifies as an API that allows a single C++ implementation to be compiled for different platforms. While the details were already covered at length in Section 2.3.1, there are two observations to note here: first, sometimes specific pragmas or data decomposition strategies are required to improve the performance for one of the platforms; this moves away from the single-source concept. Second, there is no clear separation between the scientific code and the parallelization strategy.

5.2. Performance Studies Review

In this section, a few significant performance studies that analyse the main GPU offloading APIs are reviewed. Since all libraries are actively maintained and improved, the results are presented in chronological order, which also shows the progress in portability and performance.

A performance study published by Gayatri et al. in 2018 [Gayatri et al., 2018] investigated OpenMP (4.5), OpenACC and CUDA for Intel Xeon/Xeon Phi and NVIDIA P100/V100 architectures. The authors implemented a material science kernel that computes the electron self-energy using general plasmon pole approximation through tensor contractions for double-precision arrays followed by a reduction to a 3×3 matrix. The overall memory footprint is 2GB, due to its 4 imbricated loops: the first two can be collapsed and distributed to threads on CPU and thread blocks on GPU; the 3rd loop can be vectorised on CPU and executed by parallel threads on GPU; 4th loop is small can be unrolled, targeting SIMD operations. The results are aggregated in Figure 5.1 and can be summarized as follows: First, at the time of the tests, OpenMP/OpenMP target was portable but not performance-portable; to obtain good execution times, an optimization was required, which made the GPU code to diverge from the CPU one. Second, OpenACC required several algorithm changes to parallelize the code on CPU and vectorization was still not possible with the pgi compiler, version 18.4; the OpenACC code ran on CPU is 4× slower than OpenMP as shown in Figure 5.1a. Third, CUDA implementation targets GPU only, is not portable, but it shows the best GPU occupancy from the three implementations (50%) while providing comparable wall clock times; an optimized CUDA version, which uses 32 threads per block (instead of 64), can achieve increased performance, as shown in Figure 5.1b.

In an in-depth study of Kokkos (vers 2.7), RAJA (0.6.0), OpenMP (for CPU only),

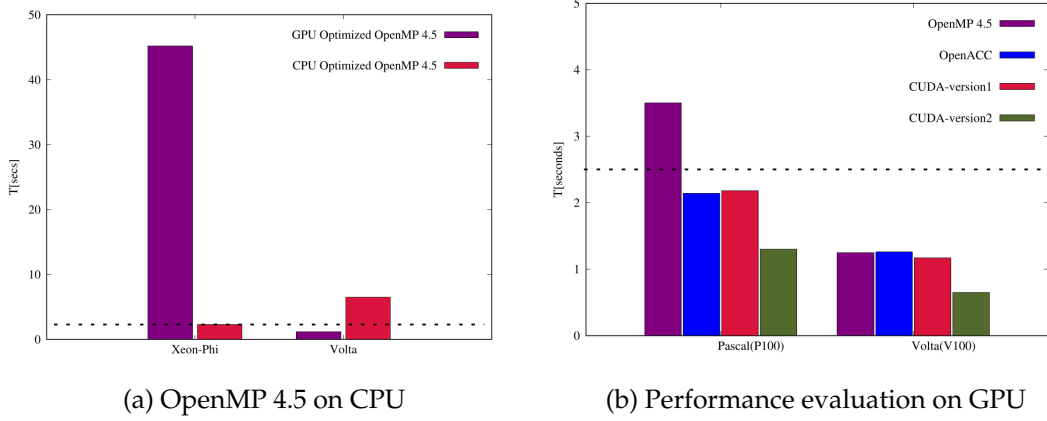


Figure 5.1.: Performance and portability results for OpenMP, OpenACC and CUDA implementations [Gayatri et al., 2018]

OpenACC and CUDA published in 2019, Artigues et al evaluate the performance and portability of these APIs on particle-in-cell method used to study the evolution of a species of plasma in its self-consistent and external electromagnetic field [Artigues et al., 2020]. This is modeled by a subset of the Vlasov-Maxwell equations that identify the position of a particle in the grid, evaluate the magnetic field intensity at the particle position, update the position and velocity, and accumulate the first component of the current density. From a coding perspective, this is a parallel loop followed by a reduction. The results are shown in Figure 5.2 and are summarized next.

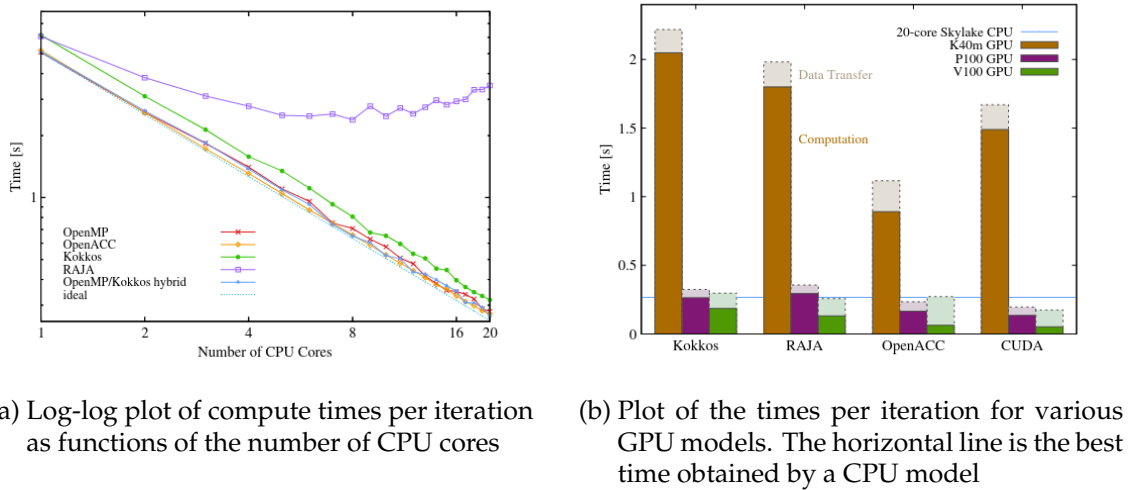


Figure 5.2.: Performance results for OpenMP, Kokkos, OpenACC, RAJA and CUDA in comparison to C++ code on CPU [Artigues et al., 2020]

First, when using RAJA, the user handled memory allocations and transfers, which resulted in two code branches, one for CPU and one for GPU; moreover, RAJA showed

*false sharing*¹¹, which was partially addressed through cache alignment and padding. Second, OpenMP and OpenACC required fewer and more straight-forward code changes, while CUDA, Kokkos and RAJA needed a significant rewrite of the code, which led to a lower productivity and code clarity. Third, for Kokkos and RAJA, the generated code was more difficult to debug since it is not accessible for inspection. Finally, from the performance perspective, there are different observation for each platform: for the CPU, OpenACC, OpenMP and Kokkos (with OpenMP backend) scale nearly ideally, while Kokkos is 1.21x slower and RAJA is far worse due to memory issues listed above; for the GPU, on NVIDIA P100 and V100, CUDA implementations is the fastest, with Kokkos and RAJA approximately 2 – 3.15× slower.

A more extensive study [Deakin et al., 2020] published in 2020 used the *performance portability metric*¹² defined by Pennycook to evaluate OpenMP, OpenCL, OpenACC, CUDA, Kokkos and SYCL implementations for two memory-bound kernels from the Babel Stream Benchmark [Deakin et al., 2018]. The studied kernels are *Dot*, which computes the sum of element-wise product of two arrays and it is implemented as a parallel loop with a reduction operation, and *Triad* or *DAXPY*, which computes $a \cdot \vec{x} + \vec{y}$, with double precision operands and which is an embarrassingly parallel loop with independent data. Having used cards from Intel, ARM, AMD, NVIDIA and Fujitsu, the experiments cover all major vendors.

The results for Triad are shown in Figure 5.3. OpenMP and Kokkos provide highest scores for performance portability metric with OpenMP being the only solution covering all the platforms from all vendors with the note that the implementation required specialization for CPU and GPU. While CUDA ensures the best wall clock times on NVIDIA GPUs, it provides the lowest performance portability score due to the lack of support on many architectures and/or vendors. OpenCL and SYCL show good results for GPU platforms but the portability on CPU resources is still limited to 30.8% and 36.1% respectively.

Platform set	P by programming model					
	OpenMP	Kokkos	OpenACC	CUDA	OpenCL	SYCL
All platforms	75.1	0	0	0	0	0
All non-zero platforms	75.1	75.4	27.3	86.1	46.6	47.4
Supported CPUs	77.9	71.6	35.9	0	30.8	36.1
Supported GPUs	72.2	81.2	22.8	86.1	81.4	81.7
Large input non-zero platforms	57.9	69.5	27.2	85.9	44.8	55.1

Figure 5.3.: BabelStream *triad* performance portability results for arrays of length 2^{25} FP64 elements [Deakin et al., 2020]

¹¹False sharing happens when more threads simultaneously update distinct locations from the same line of cache, which therefore is marked as invalid, and the threads are forced to retrieve it again; this happens because the granularity of the cache is at the line level rather than on the element’s level and induces overhead due to increased (redundant) communication.

¹²Performance portability is “defined as the harmonic mean of efficiencies over a set of platforms, or else zero if one of more platforms in the set is not supported. The metric represents the expected performance of the application on the set of platforms” [Pennycook et al., 2016].

On Dot kernel, lower efficiencies than Triad were observed due to reductions, as shown in Figure 5.4. OpenMP is the only one that covers all the architectures, with Kokkos being the most efficient one; OpenMP for CPU is fast but reductions on GPU could be improved. Similar to Triad, OpenCL and SYCL show good efficiency on GPU while performing bad or not at all on CPU.

Platform set	Φ by programming model					
	OpenMP	Kokkos	OpenACC	CUDA	OpenCL	SYCL
All platforms	9.3	0	0	0	0	0
All non-zero platforms	9.3	60.5	15.6	86.1	7.4	0.7
Supported CPUs	79.7	57.1	42.0	0.0	3.5	0.4
Supported GPUs	4.6	65.7	10.3	86.1	78.3	74.6

Figure 5.4.: BabelStream *dot* performance portability results for arrays of length 2^{25} FP64 elements [Deakin et al., 2020]

In summary, these APIs prove that performance portability can be achieved with a single-source implementation compiled for different architectures; nevertheless, for improved results, specialized backends have to be provided.

5.3. Automatic Parallelization and Offloading Tools

5.3.1. Source-to-source Translators

While we briefly mentioned *HIPify* in Section 2.4.2 as a tool that increases portability among GPU vendors, here we focus on tools that perform automatic parallelization and offloading starting from C++ code bases. This is related to *clang-offload* prototype described in Section 3.3.

ToGPU [Marangoni and Wischgoll, 2016] is a research project at the Wright State University, which prototyped a source-to-source translation tool that converts C++ to CUDA. It is based on *the Matchers* concept of LLVM/clang plugin mechanism. While the initial paper showed promising results in comparison to native CUDA implementations, the project doesn't seem to be maintained anymore and is not open-source. *OP2-Clang* [Balogh et al., 2018] is another LLVM/Clang-based translator capable of generating target parallel code based on SIMD, OpenMP, CUDA and their combinations with MPI. The particularity of this project is that it was designed for the OP2 DSL, which targets unstructured mesh computations. While the project is open-source¹³, it isn't currently maintained anymore. Another project that aims to offload computations to NVIDIA and AMD GPUs is *clad*¹⁴ [Ifrim et al., 2022]; nevertheless, it is limited to automatic differentiation operations. Similar to the previous examples, it manipulates the AST of a C++ file with the help of Clang tools.

¹³<https://github.com/OP-DSL/clang-op-translator>

¹⁴<https://github.com/vgvassilev/clad>

While all the above mentioned projects are targeted to a particular use case, more generic approaches started to be explored recently. *Future Offload Target Virtualization (FOTV)* [Vázquez and Sánchez, 2021] is a proposed prototype tool that converts OpenMP code to OpenCL and OpenMP target regions.

To summarize, the R&D efforts on source-to-source translators that enable automatic offload support are just starting to emerge; only limited use-cases approaches have been proved to be efficient so far.

5.3.2. Futhark Programming Language

Futhark [Henriksen et al., 2017] is a purely functional, statically typed, data-parallel array programming language designed to generate efficient OpenCL code for GPU execution. It is based on high-level invariants as second-order array combinators (SOAC), which include *map*, *reduce* and *scan*. These operators are used to define programs through mathematical composition. Nested parallelism is also built in the language and trivially expressed as shown in Listing 5.4, where the `main` function receives a 32-bit floating point matrix and returns a tuple containing the updated matrix by adding 1.0 to each element and a vector that represents the sums for each row in the initial matrix.

```
fun main (matrix : [n][m]f32): ([n][m]f32, [n]f32) =
  map (λ row : ([m]f32, f32) →
    let row_1 = map (λ x : f32 → x+1.0) row
    let s = reduce (+) 0 row
    in (row_1,s))
  matrix
```

Listing 5.4: Map-reduce example in Futhark [Henriksen et al., 2017]

An important emphasis is on the data shapes that provide useful information for compile-time optimization [Henriksen et al., 2014], which is achieved in two steps. First, the input program is *desugared*¹⁵ and translated into an intermediate representation (IR), which is an internal dialect of the language. This is required because the arrays of tuples cannot be efficiently represented in memory and therefore they are recursively converted to tuples containing arrays of the original tuple components. The external SOACs are also converted to tuple-less SOACs. While this conversion is reversible, the result is guaranteed to be equivalent but not necessarily identical to the external representation. Second, several optimizations are applied to the internal IR, which include: aggressive inlining of the non-recursive functions, tuple expression normalization, *copy propagation*¹⁶ and *constant folding*¹⁷, *hoisting*¹⁸, common subexpression elimination (CSE)¹⁹, loop fusions, etc. The compiler pipeline is summarized in Figure 5.5.

¹⁵Desugaring is converting the simplified syntax (denoted as syntactic sugar) back to the expanded/base versions.

¹⁶Copy propagation is the process which eliminates bindings that are merely copies of existing variables.

¹⁷Constant-folding is the process of evaluating a constant expression at compile time

¹⁸Hoisting is the movement of loop-invariant expressions out of a loop

¹⁹CSE identifies identical expressions and replaces them with a variable holding the computed value

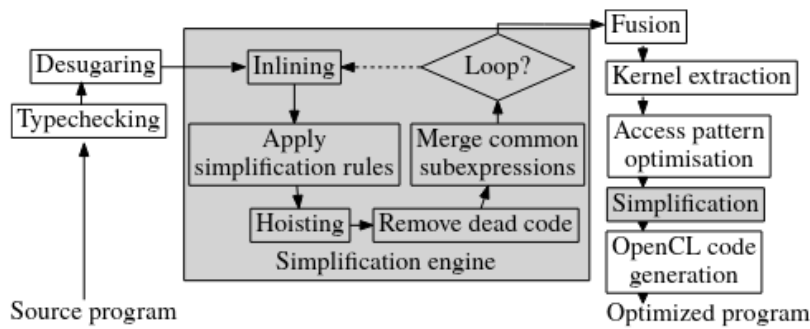


Figure 5.5.: Compiler pipeline in Futhark [Henriksen et al., 2017]

While *vecpar* offers similar invariants, the syntax is closer to C++ rather than functional programming languages like Haskell. This also means that a main stream compiler like *clang* can be used to build the executables as opposed to *Futhark*, which requires a dedicated compiler; nevertheless, Futhark compiler provides advanced optimizations based on knowledge extracted from the internal IR, which cannot be easily mirrored in general C++ compilers.

Futhark speedup ranges from 0.6× to 16× when evaluated on several benchmarks using NVIDIA and AMD GPUs. While Futhark code is slightly slower in some cases when compared to hand-optimized code, the benefits of hiding the GPU complexity from the user is expected to surpass the minimal performance loss.

5.4. Parallel Frameworks for Track Reconstruction

All CERN experiments face the same computational challenges for track reconstruction due to higher luminosity that leads to an increasing number of tracks and vertices. This makes the pattern recognition and hit classification a harder combinatorial problem. Before starting the R&D effort towards a detector-agnostic software toolkit, each experiment tackled this problem independently, leveraging information about specific setup conditions, like detector geometry, intensity of the magnetic field or the size of incoming raw detector data. Particularly, LHCb and CMS implemented GPU workflows within the existing data processing frameworks; the resulting solutions are called Allen and Patatrack, respectively.

5.4.1. Allen

In LHCb’s case, there are two software high-level triggers (HLT) that filter interesting events for long-term storage: HLT1 and HLT2. Allen Framework’s goal is to lower the workload on existing CPU nodes, by implementing the HLT1 in CUDA and therefore, leveraging the GPU farm, as shown in Figure 5.6. A memory manager based on CUDA streams and a custom scheduler ensure that data is efficiently transferred between the

resources and that the computing power is fully utilized by parallelizing over fully-independent and low size events. Each CUDA block handles one event at a time while intra-event parallelism is used process raw-data from a readout unit, to evaluate hits combination during pattern matching, to fit a track and then to combine them together to find vertices.

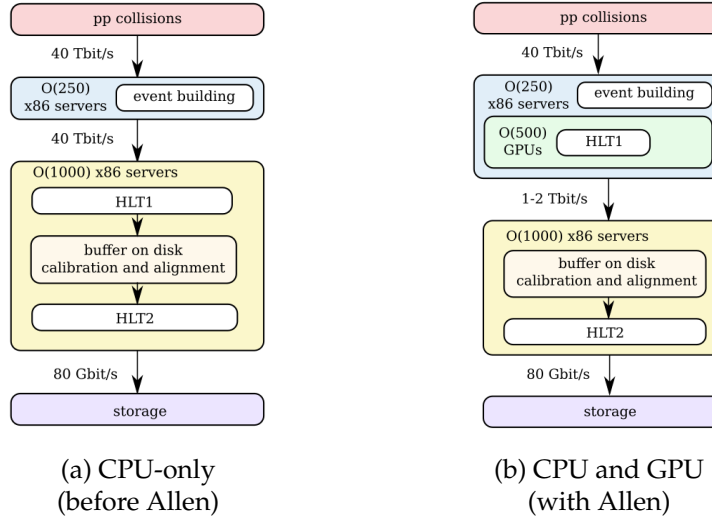


Figure 5.6.: LHCb workflow proposals [Aaij et al., 2020]

Allen proved to deliver high *efficiency*²⁰ and decreased HLT1 event rates of up to 30×, while reaching 14TFLOPS 32-peak performance for a throughput of 80kHz on a NVIDIA V100 card. At the time of this thesis, Allen framework is being extended with a HIP backend.

5.4.2. Patatrack

CMS experts proposed a GPU reconstruction flow as shown in Figure 5.7 [Bocci et al., 2020]. While all the steps run on the GPU, intermediate results are transferred on the CPU and converted to legacy formats for validation purposes. There are several parallelization strategies used in this workflow. The *digitization* step is parallelized on two levels: digitized detector data above a signal-over-noise threshold coming from different modules is processed in parallel by blocks of threads while each *digi*²¹ within a module is assigned to a thread. *Clustering* is done in parallel, with each digi assigned to a thread while a global atomic counter is used to mark the seeds. Clusters are linked together to form *n-tuples*; this process starts with creating the doubles and connecting them into a direct acyclic graph (DAG) starting from a root doublet; a DAG is then assigned to a thread that perform a depth-first-search algorithm; finally, fitting combines the results together, us-

²⁰Efficiency is defined as the fraction of simulated tracks having produced at least three hits in the pixel detector that have been associated with at least one reconstructed track.

²¹A digi represents a pixel with the charge above the signal-over-noise threshold and contains information about the local coordinates within the grid

ing one thread per n-tuplet, to obtain the track parameters and their covariance matrices. The vertex finding is parallelized in one dimension, while the reconstruction ends with sorting the vertices by the sum of p_T^2 of the contributing tracks.

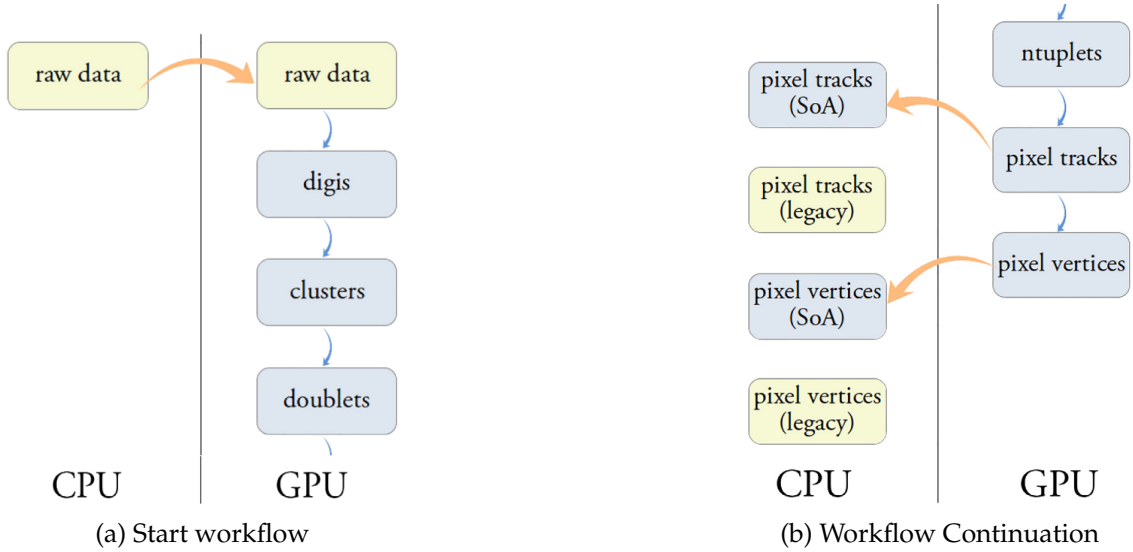


Figure 5.7.: Reconstruction flow proposed by Patatrack [Bocci et al., 2020]

In comparison to previous CMS solutions, this approach has improved *efficiency*, *fake rate*²², *duplicate*²³ *rejection* and *resolution*²⁴, while the events throughput increased up to 3×, when no CPU transfer nor data conversion is performed.

While the approaches described in the current chapter allow the code to run on both CPU and GPU platforms, they are either (a) generic and efficient but too complex to use, or (b) tailored for a specific problem and therefore difficult to reuse in similar scenarios. Vecpar could be an easy-to-use alternative for simple parallel flows without complex data access patterns while maintaining a single C++ code repository.

²²Fake rate is defined as the fraction of all the reconstructed tracks coming from a reconstructed primary vertex that are not associated uniquely to a simulated track.

²³A duplicate track is a reconstructed track matching to a simulated track that itself has been matched to at least two times.

²⁴The resolution of the estimations is a measurement of the accuracy of the fitted value in comparison to the true value

6. Evaluation

In this chapter, the evaluation criteria are initially presented and then they are used to compare the vecpar framework to other state-of-the-art and/or related APIs using different benchmarks, including a vecpar implementation of the most computationally intensive step of the track reconstruction chain: the numerical integrator. Details regarding the reproducibility of the results are available in Appendix D and are inline with the latest guidelines¹ on publications proposed by the Association for Computing Machinery.

6.1. Evaluation criteria

Runtime performance, platform portability and development productivity are the main metrics used for the evaluation; these are briefly summarized next.

6.1.1. Performance

One straight-forward criteria is the wall clock execution time and the derived metrics that can be computed based on it. For example, for the track reconstruction flow, one can measure the time to fit 10 000 particles² or how many particles can be processed during a time interval, like a millisecond.

In computer science, common metrics to evaluate the performance of an application that employs a certain level of parallelism, are *speedup* ($S(p)$) and *efficiency* ($E(p)$), defined by Equation 6.1 and Equation 6.2; here $T(1)$ and $T(p)$ are the wall clock time for optimal serial execution and parallel execution using p processors respectively [Ludwig, 2020]. While the speedup shows how a parallel implementation can increase the execution speed of an algorithm, the efficiency is a normalization over the number of processors.

$$S(p) = \frac{T(1)}{T(p)} \quad (6.1)$$

$$E(p) = \frac{S(p)}{p} \quad (6.2)$$

There are two types of scaling that can be used to evaluate a parallel implementation and they are governed by different laws. First, *strong scaling* refers to the speedup obtained when the problem size is kept constant while the number of workers p is increased.

¹<https://www.acm.org/publications/artifacts>

²This is the volume estimated for the HL-LHC

Amdahl's law provides the upper bound of this speedup using the Equations 6.3 where f is the fraction of serial code [Amdahl, 1967]. The dependency of the maximum theoretical speedup S_{max} on f and p , is summarized in Equations (6.4), which show that adding more processors without reducing the amount of sequential code (i.e. $f \rightarrow 0$) will heavily impact the ideal speedup, which is already limited to the number of processors. Second, *weak scaling* refers to the notion of *scaled speedup*, which means keeping the same workload per processor, so increasing both the number of workers p and the problem size, like in Equation 6.5 [Gustafson, 1988, Gustafson, 2011].

$$S \leq \frac{1}{f + \frac{1-f}{p}} \quad (6.3)$$

$$S_{max} = \lim_{p \rightarrow \infty} \underbrace{\frac{1}{f + \frac{1-f}{p}}}_{\frac{1}{f}} \leq \lim_{f \rightarrow 0} \underbrace{\frac{1}{f + \frac{1-f}{p}}}_p \quad (6.4)$$

$$S = f + (1 - f)p \quad (6.5)$$

6.1.2. Portability

An implementation is considered portable when the same code can be executed on different platforms. For the purpose of this thesis, a solution that ensures the same C++ source file can be compiled for x86_64 CPU and NVIDIA GPU is acceptable given the hardware resources currently available at CERN. Nevertheless, the option of extending the portability to other GPU vendors is highly desirable.

6.1.3. Productivity

To evaluate this, we take in consideration multiple factors, both objective, like the number of lines of code needed to develop an application, and subjective, like the effort to (a) learn new programming languages, language extensions or DSLs, (b) gather the knowledge about hardware resources (e.g. compute units, memory layouts, interface interconnects, etc) required to achieve the maximum performance, and (c) maintain one implementation for each targeted platform, if needed.

6.2. Test Configuration Setup

Several hardware resources were used for the evaluation. Their configurations are listed below:

- **Environment 1 (Env1)** – Mainstream laptop with Intel Core i7-10870H (8 cores × 2 hyperthreads) @2.2GHz CPU, 32 GB of RAM memory and an NVIDIA GeForce

RTX 3060Ti GPU (8 GB GDDR); the CUDA driver was 510.47.03, which supports CUDA library versions up to 11.6.

- **Environment 2 (Env2)** – HPC Cluster node with Intel Xeon Gold 5115 @2.4GHz CPU (2 sockets \times 10 cores \times 2 hyperthreads) and NVIDIA Tesla V100 GPU (32 GB GDDR), with CUDA driver 510.47.03 and CUDA library 11.5
- **Environment 3 (Env3)** – Raspberry Pi cluster node with an ARM Cortex A72 CPU, with 4 symmetrical cores
- **Environment 4 (Env4)** Intel Core i5-8600K CPU @3.60GHz, AMD Radeon RX6750 XT and ROCm 5.3.3

Unless stated otherwise, the above hardware environments had the following configurations: all used googletest 1.10.0, eigen 3.4.0 and vecmem 0.22.0; for compilation and parallelization support we used clang 14.0.0 and CUDA 11.6 on Env1 and Env2, gcc 12.1.0 on Env3, ROCm aomp 16.0.3 on Env4. All compilers provide OpenMP 4.5 support. Each test is repeated 20 times and the first run is always discarded as considered a warm-up test.

6.3. Benchmarks

The vecpar framework is evaluated using two different benchmarks. Firstly, as part of the vecpar repository, beside the automated tests for validating the API, we developed a number of benchmarks, which allow the evaluation of several features in a comparative manner; these will be introduced in Section 6.3.1. Secondly, the BabelStream Benchmark [Deakin et al., 2018] briefly mentioned in the previous chapter, is used to evaluate not only the runtime but also the memory bandwidth of different kernels and the results are discussed in Section 6.3.2.

6.3.1. Vecpar Internal Benchmark

Each of the tests in the internal benchmark generates single and/or double precision values using the C++ Standard Mersenne Twister Engine (from the C++ Standard Library) in a uniform real distribution between 0 and 1. This mechanism, together with the fact that distinct arrays are allocated for the benchmark and the vecpar case ensure that there should not be any caching impact in repeating the tests several times. Beside recording the wall clock time (which includes the host-device transfers in case of running on a GPU), the tests also validate the results of executing the algorithms using the EXPECT_EQ macro defined by the googletest infrastructure. To reproduce the results, check Appendix D.1.

6.3.1.1. SAXPY and DAXPY

Single-precision $A \times X + Y$ (SAXPY) and *double-precision* $A \times X + Y$ (DAXPY) are two common trivial kernels used for benchmarking numerical applications and defined by Equa-

tion (6.6).

$$y_i = A \times x_i + y_i, \forall i \in \overline{1, N}, A = ct \quad (6.6)$$

For the evaluation, only parallel implementations were investigated. The benchmark is composed of three native implementations: the first using OpenMP pragmas and the remaining two using CUDA (one that handles the memory allocations and copies explicitly, and the other using CUDA managed memory), while the test chooses one over the other using built-in macros to identify the platform. The associated code listings are available in Appendix B.2. Unlike these native approaches, we implemented the vecpar SAXPY/DAXPY kernel as a templated `vecpar::parallelizable_mmap` algorithm, shown in Listing 6.1, which is passed to the vecpar API as shown in the code sample in Listing 6.2³ compiled for host and device as detailed in Section 4.4. In conclusion, the implementations show that there is no need for explicit memory transfers or parallelization hints (like OpenMP pragmas or CUDA indexing mechanism), therefore vecpar could increase *productivity* by allowing developers who are not HPC experts to easily write scalable code.

```

1 #include "vecpar/core/algorithms/parallelizable_map.hpp"
2 template <typename T>
3 class axpy :
4     public vecpar::algorithm::parallelizable_mmap<
5         Two, // X and Y vectors
6         vecmem::vector<T>, vecmem::vector<T>, T // input data types
7     > {
8     public:
9         TARGET T &mapping_function(T &yi, const T &xi, T &a) const {
10             yi = a * xi + yi;
11             return yi;
12         }
13 };

```

Listing 6.1: SAXPY/DAXPY vecpar algorithm class definition

```

1 axpy<float> saxpy;
2 // invocation for the vecpar native OpenMP/CUDA backends
3 vecpar::parallel_algorithm(saxpy, mr, y, x, a);
4 // invocation for the vecpar OpenMP target backend
5 vecpar::ompt::parallel_algorithm(saxpy, mr, y, x, a);

```

Listing 6.2: Vecpar SAXPY invocation code samples

Since vecpar is expected to work within libraries that already use heterogeneous memory allocators (like the ones exposed by the vecmem library for example), we evaluate the performance and scalability on SAXPY and DAXPY kernels when the initial data is stored in CPU memory with or without CUDA unified memory support. For a CPU platform,

³At the time of the tests, the clang compiler cannot accept both OpenMP target code and C++ code compiled for CUDA in the same compilation unit; therefore, the user library should not mix the two GPU approaches. Consequently the calls in lines 3 and 5 cannot belong to the same compilation unit. Moreover, executing vecpar OpenMP target backend functions on data using CUDA unified memory and compiled for a GPU is not a valid scenario since this would mean mixing the two approaches.

when the data is allocated through the CUDA unified memory allocator, no transfer to the GPU (if one exists in the system) is actually done, but there is a small overhead from initializing the CUDA runtime; whereas for a GPU platform, the memory transfers are done automatically by the CUDA runtime based on the page-fault mechanism.

Figure 6.1 investigates the *performance* of the SAXPY/DAXPY kernels when executed on a CPU. All the measurements show that both vecpar implementations (using the OpenMP and the *OpenMP Target*⁴ backends) are in the same order of magnitude as the OpenMP native implementations, with vecpar being identical or slightly faster in 14 out of the 20 *test cases*⁵; nonetheless, the time improvement/overhead is within 10^{-6} seconds, which can be due to operating system interruptions.

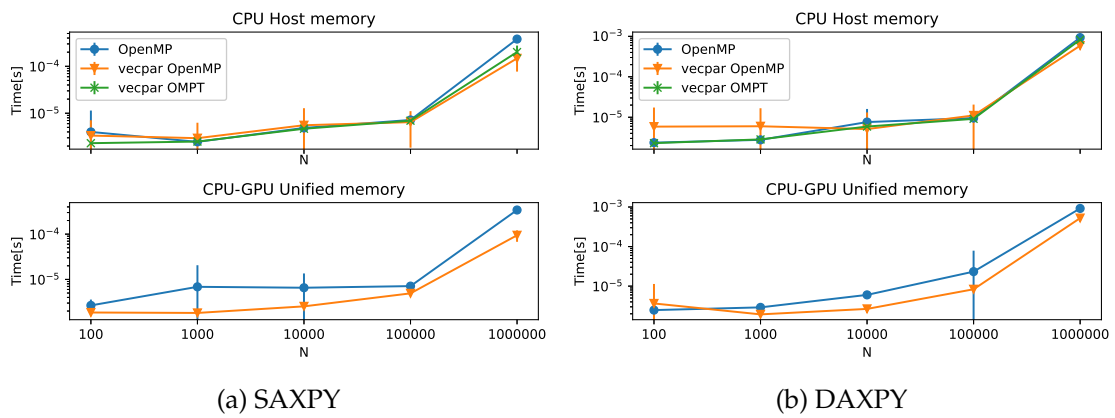


Figure 6.1.: Mean execution time and standard deviation of vecpar OpenMP and OpenMP target backend implementations in comparison to a native OpenMP version, on a logarithmic scale, when using input data vectors of size N , on x86_64 platform, Env1

Similar to the CPU results, the GPU measurements, plotted in Figure 6.2, show that vecpar is in the same order of magnitude with CUDA kernels, with the vecpar CUDA and OpenMP target backends being identical or slightly faster than the native code in 15 and respectively 14 test cases out of 20. The time improvement/overhead is within 10^{-5} seconds.

To illustrate the *portability* using vecpar, we compiled the vecpar algorithm in Listing 6.1 using the two vecpar GPU backends (CUDA and OpenMP target) for NVIDIA and AMD devices, while the results are shown in Figure 6.3. While the AMD GPU seems slower for smaller problem sizes, it proves to be faster for vectors of one million elements in both single and double precision. We credit this speedup to vendor-specific optimizations done by the ROCm aomp compiler.

⁴As a reminder, the OpenMP target backend will delegate the execution to the OpenMP backend when compiled for a CPU platform.

⁵The 20 cases = 5 values for $N \times 2$ memory allocators \times 2 kernels (SAXPY/DAXPY)

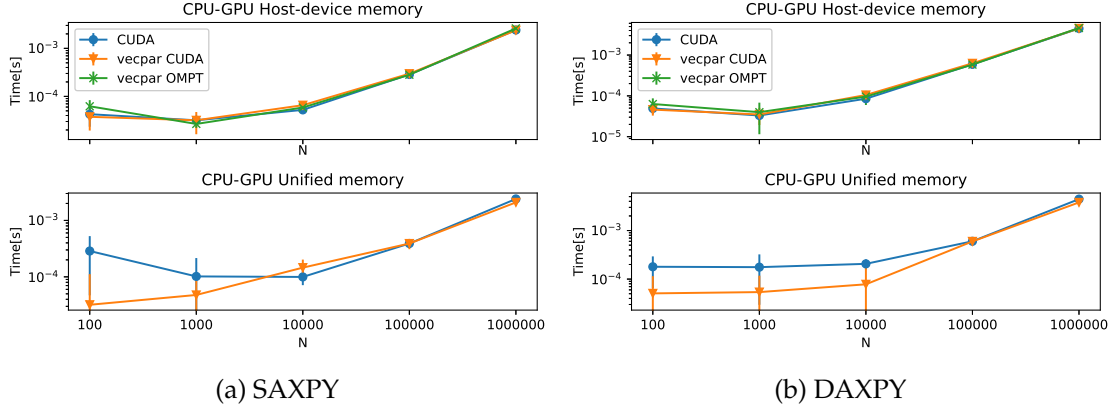


Figure 6.2.: Mean execution time and standard deviation of vecpar CUDA and OpenMP target backend implementations in comparison to a native CUDA version, on a logarithmic scale, when using input data vectors of size N , on an NVIDIA platform, Env1

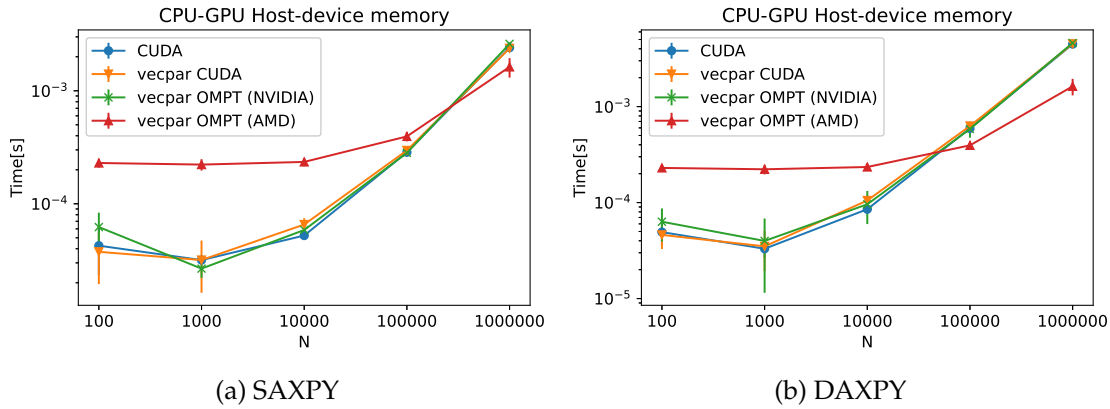


Figure 6.3.: Mean execution time and standard deviation of the same vecpar implementation compiled for different platforms, on a logarithmic scale, when using input data vectors of size N , on Env1 and Env4

6.3.1.2. Algorithm chaining

The vecpar chain functionality introduced in Section 4.3.5 was designed to improve development productivity by allowing a simplified way of calling a sequence of steps from a more complex use case and to increase performance when using an NVIDIA GPU by making the chain aware of the entire flow, so that intermediate results are no longer copied to the host for performance reasons. The main goals of these experiments are to (a) identify the potential overhead in comparison to the common way of function invocation, and (b) evaluate the impact of the performance optimization for the vecpar CUDA backend when the user data is initially in host memory.

The experiments consist in defining two vecpar algorithms, f and g , which are invoked either as the default `parallel_algorithm(g, parallel_algorithm(f,...))` or through a vecpar chain with `_algorithms(f,g)` composition function while the data is in either host memory or CUDA unified memory. The same C++ files are then compiled

for CPU and GPU using the appropriate compilation flags. The first algorithm, f , is a `parallelizable_map_filter` that performs a double precision multiplication between an integer and a double constant and then keeps only the even numbers. The second algorithm, g , is a `parallelizable_mmap_reduce` that performs a double precision multiplication for each element of a collection and then sums up the results. The size of the input data array varies from ten to one million elements.

Firstly, we want to motivate the need for the performance optimization we had in place for the GPU chain. Initially, we compared running `parallel_algorithm(g, parallel_algorithm(f, ...))` on NVIDIA GPU with an input vector of one million single-precision elements stored either in host memory or managed memory. It was expected that the former is faster than the latter because the read/write access to the data is faster⁶ but the impact needed to be quantified. For this, Nsight Compute and Nsight Systems tools from NVIDIA Toolkit are used to visualize more information about the device execution. Each of them have 4 kernels: $f_{mapping}$, $f_{filtering}$, $g_{mmaping}$ and $g_{reducing}$, which are independently traced by the profilers. Table 6.1 shows the duration, the compute throughput and the memory throughput⁷ for these kernels. f is executed with $3907 \text{ blocks} \times 256 \text{ threads}$ and while g uses $1954 \text{ blocks} \times 256 \text{ threads}$. While the grid configuration is chosen by `vecpar`, the low level features like the register count is decided by the CUDA runtime; in this case, all kernels use 16 registers, except for $f_{filtering}$, which uses 40. Table 6.2⁸ zooms even further into the $f_{mapping}$ kernel. The numbers show that, for this scenario, there is a great benefit of explicitly transferring the data between host and device. Nevertheless, these operations can increase the total execution time, which is investigated further.

Kernel	Duration [msec]	Compute throughput [%]	Memory throughput [%]
$f_{mapping}$	0.04 (-98.72%)	32.84 (+8473%)	85.27 (+3070%)
$f_{filtering}$	2.81 (-42.11%)	13.43 (+72.02%)	15.41 (+65.30%)
$g_{mmaping}$	0.02 (+1.89%)	25.98 (+2.52%)	87.22 (+3.51%)
$g_{reducing}$	1.24 (-5.65%)	11.18 (+0.10%)	7.04 (+0.12%)

Table 6.1.: Performance metrics for the CUDA kernels when the initial data is in host memory benchmarked against the same kernels using data stored in unified memory on Env1

Figure 6.4 shows a comparison between the execution time using default mechanism and the chain API. While for most of the cases the chain is faster, there is an overhead of up to 37% (3×10^{-4}) for the GPU case when managed memory is used, when the problem size is smaller than 10 000 elements.

⁶In the case of managed memory, the data is copied to the device based on page faults, when it is needed. Therefore, this can increase the total execution time.

⁷For the duration, smaller is better; for the throughput, a larger percent is better

⁸These are extracted from the reports produced by Nsight Compute.

Metric	Value
Memory throughput [Gbyte/sec]	255.62 (+2786%)
L1 Hit Rate [%]	0 (0%)
L2 Hit Rate [%]	66.74 (+10.85%)
Theoretical occupancy [%]	100 (0%)
Achieved occupancy [%]	80.29 (-19.37%)
SM busy [%]	35.07 (+9070%)
Avg. Active threads per warp	32 (0%)

Table 6.2.: Performance metrics for $f_{mapping}$ on one million floats to produce one million doubles, for the CUDA kernel using host-device memory benchmarked against the same kernel using unified memory on Env1

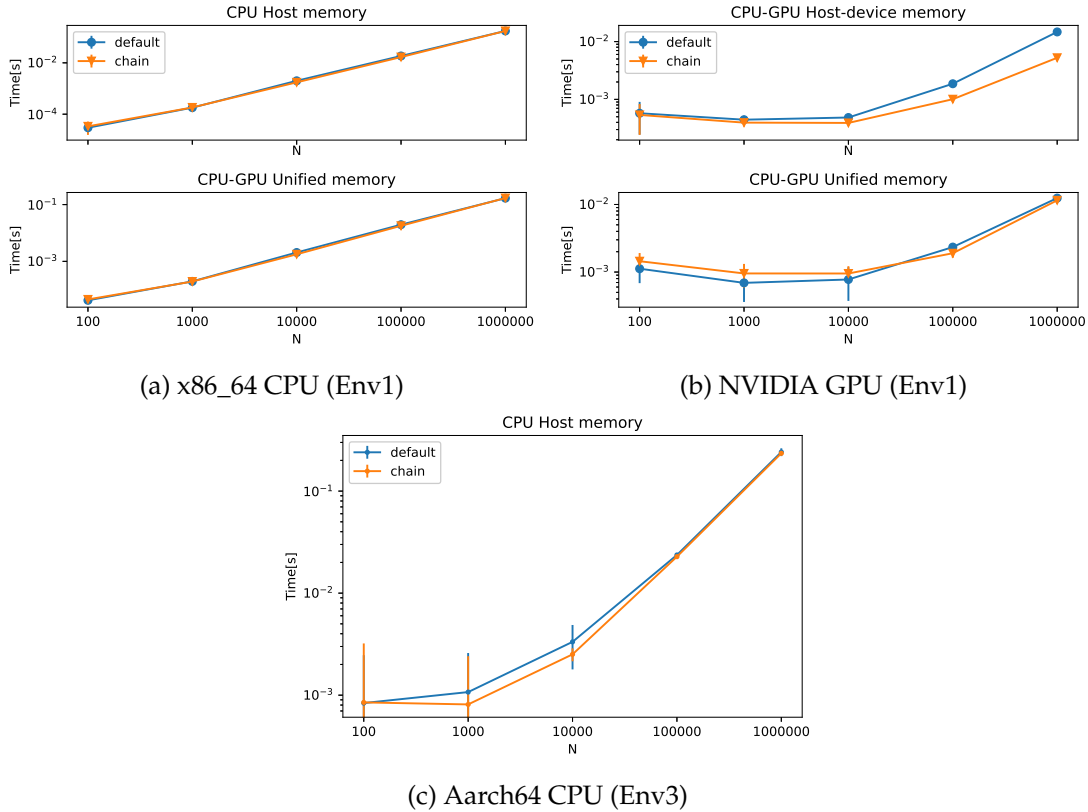


Figure 6.4.: Mean execution time and standard deviation of vecpar default parallel_algorithm function calls vs vecpar chain invocation on different architectures, on a logarithmic scale, when using input data vectors of size N

Moreover, when the input vectors have one million elements, the slowest GPU configuration (using host-device intermediate transfers) is still $\approx 10\times$ faster than the CPU parallel alternative, while the chain optimized GPU execution is $\approx 30\times$ faster. For the CPU platforms in Figure 6.4a and Figure 6.4c, the vecpar dispatch system uses the OpenMP backend and runs the code with 16 and 4 threads respectively, whereas it uses the CUDA backend (with $\min(N, 256)$ threads per block) for the plots in Figure 6.4b.

The GPU run in Figure 6.4b is analysed further on Env1, by looking into speedups when vectors of different lengths are used and by analysing several device execution

parameters when `vecpar`'s performance optimization is employed. Therefore, Figure 6.5 shows the comparison between the wall clock time of the default execution flow and the chain flow when the data is initially in host memory, plotted on a logarithmic scale. The time spent for transfers between host and device are included in the total time plotted here. This shows that the chain API is always faster than the default one for this use case, with a speedup factor between 1.24 and 3.19 for an input vector of 2^{10} and 2^{25} elements respectively.

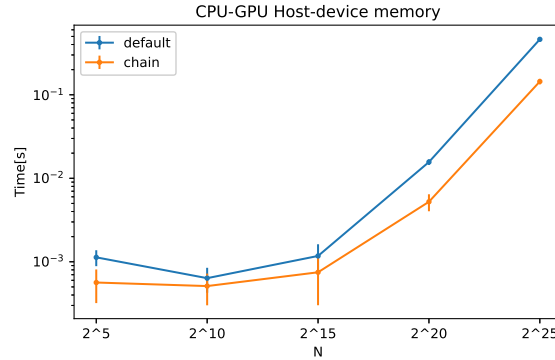


Figure 6.5.: Mean execution time and standard deviation for `vecpar` default `parallel_algorithm` function calls vs `vecpar` chain, when using input data vectors of size N , on a logarithmic scale, using an NVIDIA platform, Env1

6.3.2. BabelStream Benchmark

The BabelStream [Deakin et al., 2018] is an implementation of the McCalpin STREAM benchmark with adaptation for HPC: heap memory is used instead of stack memory, the size of arrays is known at runtime, etc. It provides an infrastructure to evaluate several kernels defined by the formulas in Equation (6.7), in single and double precision using different streams. Beside validating the correctness of a result computed by a chosen backend, it also evaluates the average memory bandwidth and the execution runtime of a given kernel, by rerunning them 100 times. To reproduce the results, check Appendix D.2.

$$\begin{aligned}
 \text{triad} : \quad & a_i = b_i + \text{scalar} * c_i \\
 \text{copy} : \quad & c_i = a_i \\
 \text{mul} : \quad & b_i = \text{scalar} * c_i \\
 \text{add} : \quad & c_i = a_i + b_i \\
 \text{dot} : \quad & \text{sum} = \text{sum} + a_i * b_i
 \end{aligned} \tag{6.7}$$

We extended the BabelStream with a `vecpar` stream in an open-source *repository*⁹, forked from the main project. The valid usecases (controlled through optional build flags `MEM`, `VECPAR_BACKEND` and `OFFLOAD`) and their appropriate `vecpar` implementations to guarantee meaningful measurements are shown in Table 6.3; the highlighted ones are used for

⁹<https://github.com/wr-hamburg/BabelStream>

an in-depth analysis further in this section¹⁰.

No.	Memory	Backend	Offload	Implementation
1	DEFAULT	NATIVE	OFF	single-source
2	DEFAULT	NATIVE	ON	lambda
3	DEFAULT	OMPT	OFF	single-source
4	DEFAULT	OMPT	ON	single-source
5	MANAGED	NATIVE	OFF	single-source
6	MANAGED	NATIVE	ON	single-source
7	MANAGED	OMPT	OFF	-
8	MANAGED	OMPT	ON	-

Table 6.3.: Valid BabelStream scenarios using Vecpar Stream; the measurements extracted from the highlighted test cases are the basis of the current evaluation

Vecpar backend NATIVE means targeting GPU using CUDA backend, as opposed to using OMPT. Despite the fact that a single-source implementation is possible for all the valid scenarios, we chose a lambda offloading approach for the GPU-CUDA case in order to avoid counting the memory allocation and transfer time twice¹¹. Code samples extracted from the VecparStream implementation that illustrate the algorithms were already shown in Listing 4.3 (on page 83) and Listing 4.7 (on page 86), while the single-source/lambda approaches were shown in Listing 4.20 and Listing 4.21 (both on page 100). The first three listings show that no previous knowledge of parallelization and/or offloading is required to implement the single-source vecpar versions, which is an important advantage over the other approaches shown in Listing 6.3¹²; the lines 12 – 18 have to be duplicated for each array, which means 3× in this case. Also, the Kokkos library requires extra calls to initialize and tear down the parallel environment.

```

1 // define the Kokkos vector addition kernel
2 template <class T>
3 void KokkosStream<T>::add() {
4     Kokkos::View<T*> a(*d_a);
5     Kokkos::View<T*> b(*d_b);
6     Kokkos::View<T*> c(*d_c);
7     Kokkos::parallel_for(array_size, KOKKOS_LAMBDA (const long index) {
8         c[index] = a[index] + b[index];
9     });

```

¹⁰When the OMPT backend is compiled for CPU, vecpar will delegate the execution (through compile-time pragmas) to the OMP backend therefore making case 3 identical to 1. Test cases 5 and 6 do not have a correspondent using a Kokkos implementation; while the GPU targeting versions of vecpar and CUDA using managed memory could be compared against each other, the preliminary results showed that the latter is twice faster than the former but further investigation is required to validate the assumption. Test cases 7 and 8 are invalid since the OMPT backend is designed to support pointers allocated in host memory only; if future production use case show the need for a different approach, this can easily be extended.

¹¹The memory handling step is hidden by the vecpar API (in order to relieve the user from handling it explicitly) and it is one of the last steps in the vecpar's state machine, as shown in Figure 4.4. Therefore, a clear separation of the data transfer time from the function execution was not straight-forward for the CUDA backend as it is against the library's design.

¹²Code extracted from <https://github.com/UoB-HPC/BabelStream/blob/main/src/kokkos>

```

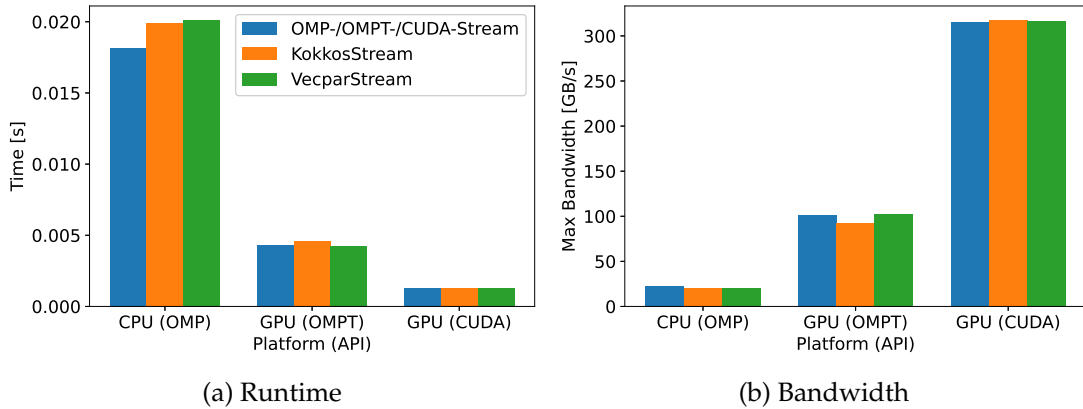
10 Kokkos::fence();
11 }
12 // define the device side array
13 typename Kokkos::View<T*>* d_a;
14 d_a = new Kokkos::View<T*>("d_a", ARRAY_SIZE);
15 // map the host array to the device array
16 typename Kokkos::View<T*>::HostMirror* hm_a;
17 hm_a = new typename Kokkos::View<T*>::HostMirror();
18 *hm_a = create_mirror_view(*d_a);
19 // transfer the memory from device to host
20 deep_copy(*hm_a, *d_a);

```

Listing 6.3: BabelStream Kokkos code samples

BabelStream allows the evaluation of the runtime and the bandwidth obtained by the VecparStream in comparison to other four streams: OpenMP (OMPStream), OpenMP target (OMPTStream), CUDA (CUDASTream) and Kokkos v.3.7.1 (KokkosStream); the first three are considered the benchmarks for state-of-the-art solutions targeting CPU and GPU, while KokkosStream is the benchmark that represents the closest similar library in terms of single-source approaches. For both vecpar and kokkos frameworks, the backends OMP, CUDA and OMPT are compared. In terms of hardware platform (and the associated targeted API), there are several test cases: CPU (OMP), GPU (CUDA) and GPU (OMPT), corresponding to test cases 1, 2 and 4 from Table 6.3.

Figure 6.6 and Figure 6.7 show the results for executing the *triad* kernel implemented using either (a) native solutions (OMPStream, OMPTStream, CUDASTream), (b) KokkosStream or (c) VecparStream, on different platforms, in single and double precision respectively. The plots show that vecpar adds an overhead between 9% and 11% over the native OMP CPU implementation while it varies with $\pm 0.8\%$ in comparison to the Kokkos implementation. For the GPU, when the default memory model is used, vecpar is within $\pm 1\%$ of the native approaches. The bandwidth is inversely proportional to the runtime, so the minimum execution time is associated with the highest bandwidth; due to this dependency, the bandwidth plots for the remaining kernels are omitted.

Figure 6.6.: Runtime and bandwidth for the *triad* kernel with vectors of 2^{25} FP32 elements, on Env1

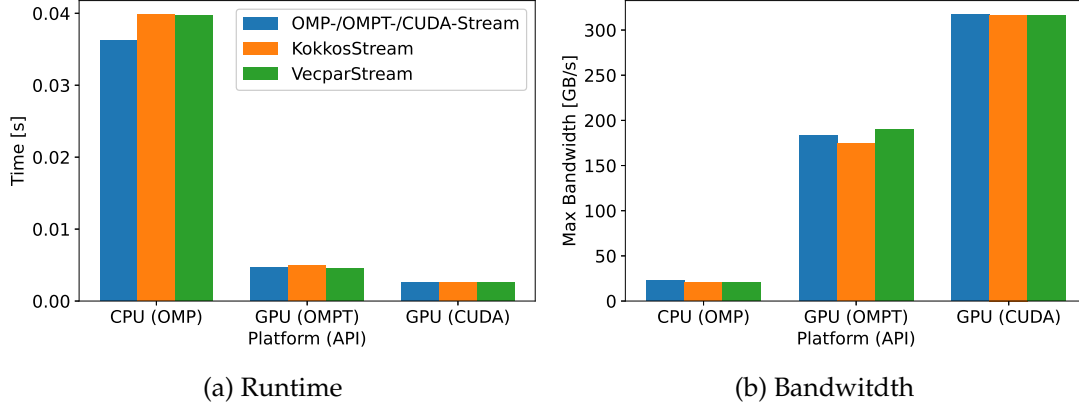


Figure 6.7.: Runtimes and bandwidth for the *triad* kernel with vectors of 2^{25} FP64 elements, on Env1

Figure 6.8, Figure 6.9 and Figure 6.10 show the runtimes for the kernels *mul*, *add* and *copy* of the BabelStream, executed in single and double precision. The vecpar OMP backend adds an overhead of up to 12% and 1% in comparison to the OMPStream and KokkosStream respectively. For the GPU (CUDA) test case, vecpar is between -5% and $+1\%$ when compared to CUDASTream and between 0% and 1% compared to KokkosStream. Finally, for the GPU (OMPT) case, VecparStream is consistently faster than both native and kokkos implementations for all the configurations, with speedups of up to 3% over the former and 9% over the latter.

As a general observation when running the experiments on NVIDIA hardware, the CUDA implementations are roughly $4\times$ and $2\times$ faster than OpenMP target implementations for single-precision and for double-precision respectively. Newer LLVM/clang versions that support *link-time optimizations*¹³ showed improved performance in preliminary measurements and are expected to minimize this gap.

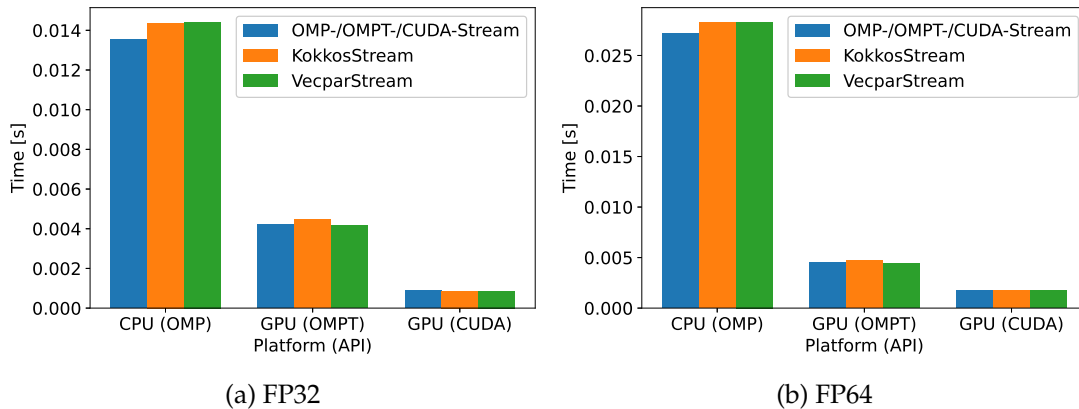


Figure 6.8.: Runtimes for the *mul* kernel with vectors of 2^{25} elements in single and double precision, on Env1

¹³Feature enabled using `-foffload-l1` to flag.

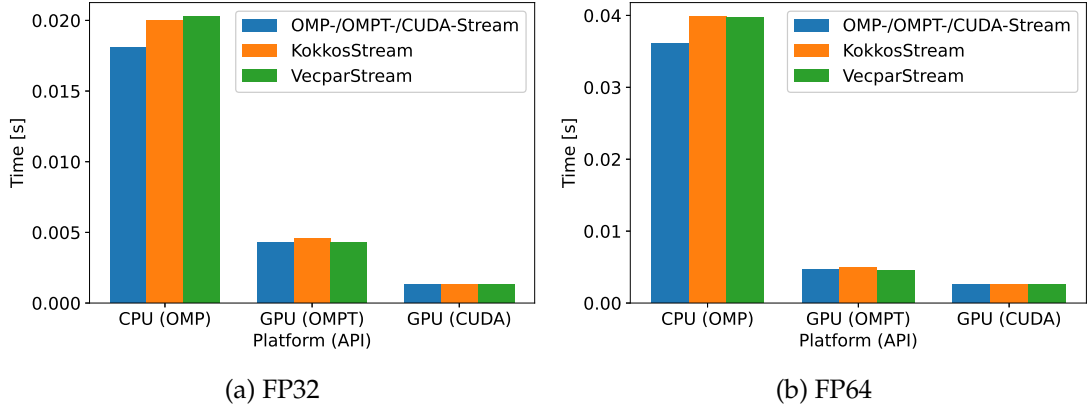


Figure 6.9.: Runtimes for the *add* kernel with vectors of 2^{25} elements in single and double precision, on Env1

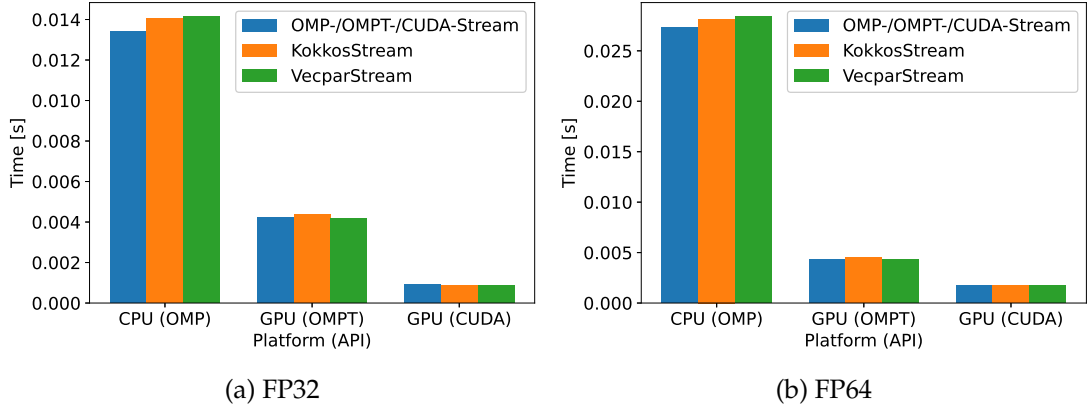


Figure 6.10.: Runtimes for the *copy* kernel with vectors of 2^{25} elements in single and double precision, on Env1

Lastly, we investigated the *dot* kernel, which is the only one that involves reductions. A general observation is that all of the OMPT implementations are slower than native CUDA-based approaches. Figure 6.11 shows that VecparStream is much slower than the benchmarks, with a factor of 6 – 8 \times for the OMP and OMPT APIs and 32 – 64 \times for CUDA. The cause of this poor performance is the fact that vecpar allocates an intermediate array to store the result of the *map* operation, which is later reduced; this induces an overhead because of (a) allocating 2^{25} float or double values before starting the parallel loop (which also justifies why the CUDA backend is slower than the CPU one), and (b) going through a loop of length 2^{25} twice. To alleviate this problem, we prototyped¹⁴ a new approach for executing the *parallel_map_reduce*, marked with *VecparStream-Opt* in the OMPT backend which demonstrates a performance improvement which could allow vecpar to be competitive with other solutions, while being 2 \times faster than OMPStream and KokkosStream (using OMP backend) on NVIDIA hardware. The generalization of this approach to the other overloaded implementations in all the vecpar backends is left

¹⁴This optimization is in place only when one or two iterable collections are passed to the operators

for future work.

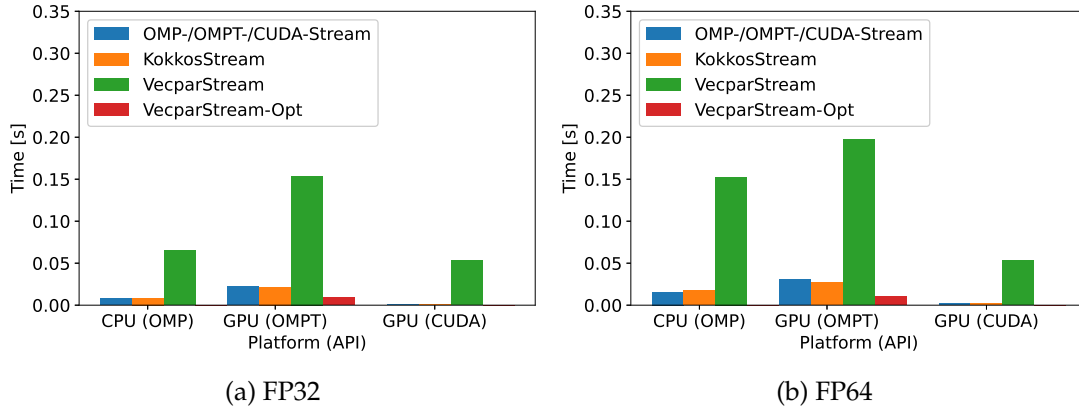


Figure 6.11.: Runtimes for the *dot* kernel with vectors of 2^{25} elements in single and double precision, on Env1

To evaluate the *portability*, we tested OMPTStream, KokkosStream (with the OMPT backend) and VecparStream on an AMD GPU as shown in Figure 6.12. For all the kernels that use independent data, in both single and double precision, *vecpar* is 1 – 3.3% faster than Kokkos, while both frameworks are slightly slower than the native OMPTStream. Similar to the NVIDIA results above, for the *dot* kernel, *VecparStream*¹⁵ is roughly 3× slower than Kokkos.

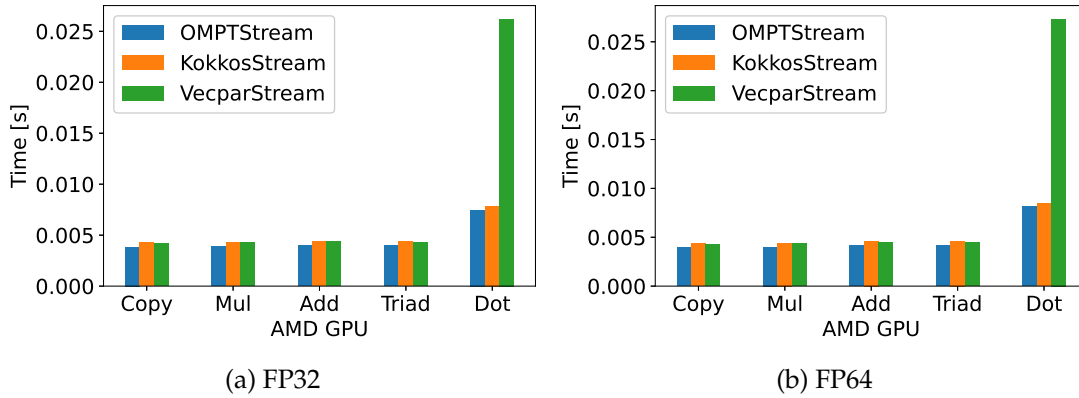


Figure 6.12.: Runtimes for the BabelStream kernels with vectors of 2^{25} elements in single and double precision, on Env4

To summarize this section, we can conclude that for the parallel cases when no reductions are involved, the *vecpar* framework ensures comparable performance to native or similar approaches, with the GPU backends being faster in most of the cases or induce an overhead of under 1% otherwise. Nevertheless, the GPU offloading code was constantly faster than the CPU code, with a speedup factor between 5 and 15 depending on the BabelStream benchmark and the *vecpar* backend (OMPT/CUDA). We identified the problems

¹⁵Note that this test already uses the optimized functionality for *parallelizable_map_reduce* but for the AMD case, the impact is limited

behind the lack of performance in case reductions are used and proposed an adaptive algorithm, which shows promising results in preliminary tests using the BabelStream *dot* kernel with the OMPT backend for the NVIDIA hardware; an in-depth investigation is required to fully explain the minimal improve in performance for the AMD hardware.

6.4. Track Reconstruction – RKN Stepper

Section 1.2.3 summarized the track reconstruction flow, which requires several steps, while the parallelization potential lies in reconstructing multiple tracks in parallel and aggregating them in vertices belonging to the same event at the end. One of the most computationally intensive steps is the numerical integrator used to estimate the position and momentum of a charged particle. As detailed in Section 3.1.1, the ACTS R&D project *detray* provides an implementation of the adaptive RKN method that uses the linear algebra API exposed by the *algebra-plugin* library through its two backends: *cmath* and *eigen*. This is encapsulated into an algorithm named *RKN stepper*, which repeatedly calls the integrator (i.e. `RKN_Stepper.step(...)` function) to compute the estimated track states. The computational cost comes mostly from working with arrays and matrices to model track parameters, 3D projections, propagation error estimations, etc. While these operations can be performed in single precision for the online reconstruction, for the offline case they require increased precision.

In this section, we evaluate two versions of algorithms working with the RKN stepper in *detray*. The first one is a simplified use case that computes the track states without taking into consideration the material interaction with the detector surfaces and their potential impact on final solutions; these are accounted for and modeled through covariance matrices in the second version. Nevertheless, both versions assume the motion happens in a constant magnetic field, which is a simplification from a frequent scenario in experiments, where magnetic fields are often non-homogenous.

For these two versions, several implementations are compared: `seq_cpu` – C++ sequential version, which runs on CPU, `omp_cpu` – C++ with OpenMP pragmas, `cuda_nvcc` – CUDA implementation exploiting managed memory support, and finally, `vecpar_cpu` / `vecpar_gpu` – single-source C++/vecpar implementation compiled for CPU and GPU. The `vecmem` library (v.0.12.0) is used to store the data and to handle CPU-GPU transfers. While the C++ and CUDA code was written by domain experts, our contributions are: the OpenMP parallelization of the (existing) RKN stepper working scenario and its porting to `vecpar`. The code is open-source on github¹⁶ and it is pending review in order to be merged into the main repository¹⁷. To reproduce the results, check Appendix D.3.

¹⁶<https://github.com/georgi-mania/detray>, the *vecpar* branch

¹⁷<https://github.com/acts-project/detray>

6.4.1. Simplified RKN Stepper

This simplified version computes the global position (using free track parameters) and momentum by integrating the equation using a fixed number of integration steps. Unless stated otherwise, each test estimate 10 000 track parameters in 100 integration steps.

The development productivity is investigated by comparing the CUDA and the vecpar implementations in the *detray repository*¹⁸ in Listing 6.4 and Listing 6.5 respectively. While the former requires knowledge of CUDA, by computing the index based on the grid layout, the latter uses only C++ concepts. The *parallelizable_mmap* algorithm abstracts away all the complexity (including the thread indexing, memory copies and device synchronization). The highlighted areas mark the code that instantiates the RKN stepper and computes the estimates by performing integration steps in forward and backward directions, and it is identical in both implementations.

```

1  __global__ void rk_stepper_test_kernel (
2      vecmem::data::vector_view<free_track_parameters> tracks_data,
3      const vector3 B) {
4      int gid = threadIdx.x + blockIdx.x * blockDim.x;
5      vecmem::device_vector<free_track_parameters> tracks(tracks_data);
6      // Prevent overflow
7      if (gid >= tracks.size()) {
8          return;
9      }
10     // Get a track
11     auto& traj = tracks.at(gid);
12     // Define RK stepper
13     rk_stepper_type rk(B);
14     // Index for stepping
15     unsigned int i_s=0;
16     // Forward direction
17     rk_stepper_type::state forward_state(traj);
18     for (i_s=0; i_s<rk_steps; i_s++) {
19         rk.step(forward_state);
20     }
21     // Backward direction
22     traj.flip();
23     rk_stepper_type::state backward_state(traj);
24     for (i_s=0; i_s<rk_steps; i_s++) {
25         rk.step(backward_state);
26     }
27 }
```

Listing 6.4: A CUDA kernel that uses the RKN stepper

¹⁸The code snippet is extracted from a stable version of the project from April 2022; the repository has been continuously evolving due to a large number of contributors. While the same vecpar abstractions are used, the physics related part is enhanced with some extra function calls.

```

1 struct rk_stepper_algorithm :
2 public vecpar::algorithm::parallelizable_mmap<free_track_parameters, vector3>{
3     TARGET free_track_parameters& mapping_function(free_track_parameters& traj,
4         vector3 B) override {
5         // Define RK stepper
6         rk_stepper_type rk(B);
7         // Index for stepping
8         unsigned int i_s=0;
9         // Forward direction
10        rk_stepper_type::state forward_state(traj);
11        for (i_s=0; i_s<rk_steps; i_s++) {
12            rk.step(forward_state);
13        }
14        // Backward direction
15        traj.flip();
16        rk_stepper_type::state backward_state(traj);
17        for (i_s=0; i_s<rk_steps; i_s++) {
18            rk.step(backward_state);
19        }
20        return traj;
21    };

```

Listing 6.5: A C++/vecpar algorithm using the RKN stepper

To evaluate the performance portability, several experiments are run on Env1, Env2 and Env3, using vecmem 0.12.0 and clang 13.0.0 as host compiler for CUDA.

Figure 6.13 shows that regardless of the test environment when using double precision operands, the vecpar implementation is as fast or slightly faster than the OpenMP one while being up to 12× and 30× faster than the sequential version that runs on a single processor when using either algebra-plugin backend on Env1 and Env2 respectively; this is more clearly shown in Figure 6.14, where speedup-ups of 27×-32× are recorded for Env2.

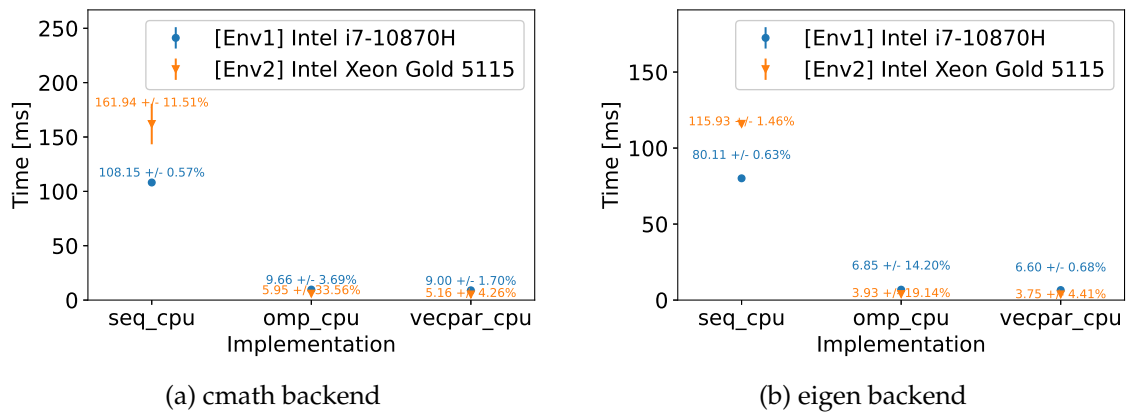


Figure 6.13.: Mean and standard deviation for sequential, OpenMP and vecpar implementations using the RKN stepper, in double precision, running with 40 and 16 threads, on Env1 and Env2 respectively

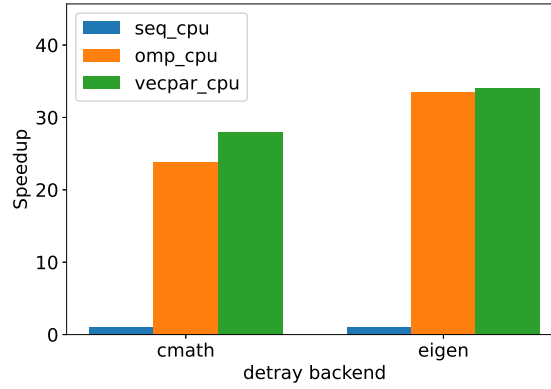


Figure 6.14.: Speed-up factors over initial sequential version for the multi-threading OpenMP (`omp_cpu`) and vecpar (`vecpar_cpu`) implementations using `cmath`/`eigen` backends, in double precision, running with 40 threads, on Env2

Figure 6.15 shows the results for the tests focusing on single precision, which validate the above observations: both OpenMP and vecpar versions provide close to ideal strong scaling performance.

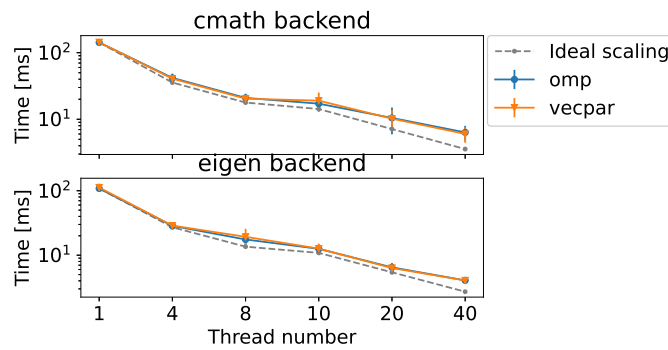


Figure 6.15.: Strong-scaling evaluation: RKN stepper multi-threading implementations in single precision for `cmath`/`eigen` backends, on Env2

The GPU tests assume the initial track data is stored in CUDA unified memory in both `cuda_nvcc` and `vecpar_gpu`. Additionally, we take in consideration advanced compiler support of using hardware intrinsic replacement for some mathematical operations. Both the `nvcc` and the `clang` compiler offer this performance optimization by enabling different `fastmath`-related flags. The impact is investigated on NVIDIA GPU when the computations use single and double precision. The CUDA implementation compiled with `nvcc` (using `clang` as host compiler) is compared with the vecpar single-source implementation compiled with `clang`. Figure 6.16 and Figure 6.17 show that the vecpar implementation is 7 – 21% faster than the native CUDA in both `cmath` or `eigen` backends for each NVIDIA GPU that we tested. Figure 6.18 and Figure 6.19 confirm the above observation, in this case with a speedup of 8 – 26%, regardless of the operands' precision.

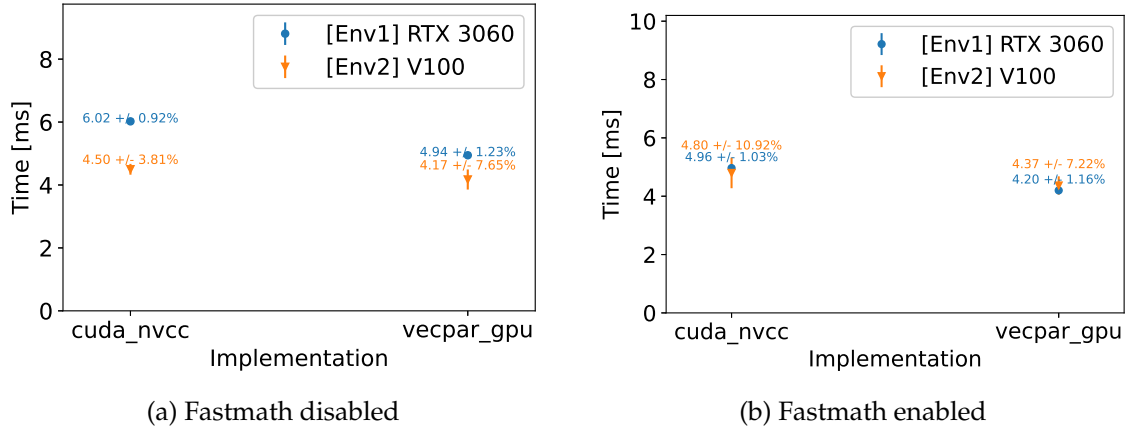


Figure 6.16.: Mean and standard deviation when using the RKN stepper and *cmath* backend on different NVIDIA GPU with grid configuration: 157×64 CUDA threads

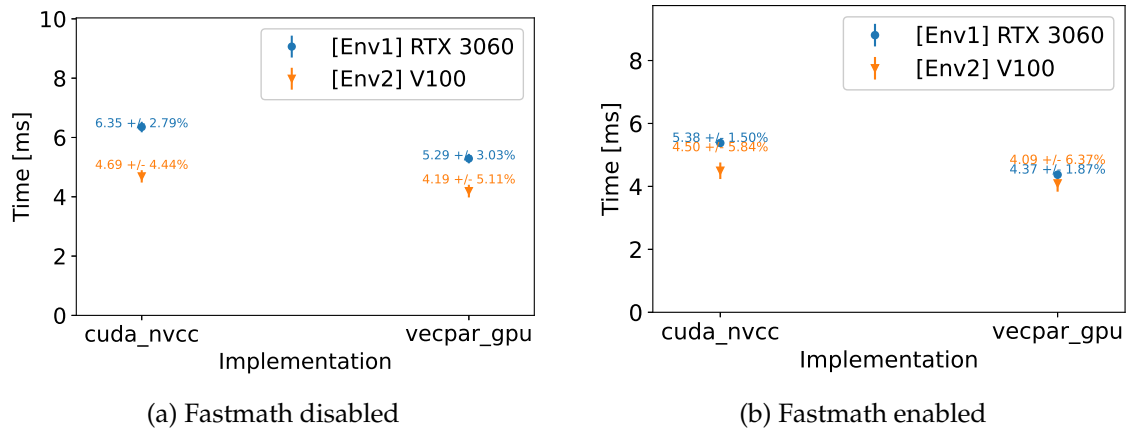


Figure 6.17.: Mean and standard deviation when using the RKN stepper and *eigen* backend on different NVIDIA GPU with grid configuration: 157×64 CUDA threads

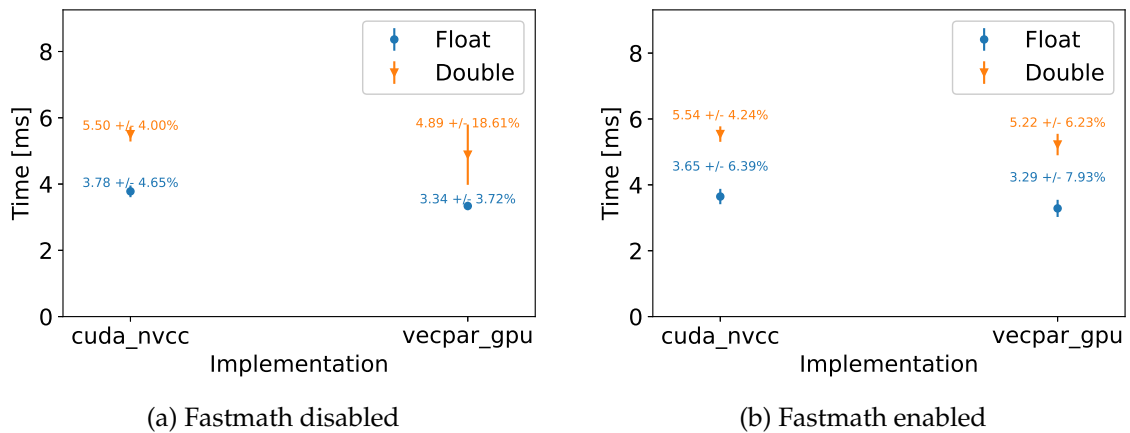


Figure 6.18.: Mean and standard deviation when using the RKN stepper and *cmath* backend, in single and double precision, with grid configuration 157×64 CUDA threads, on Env2

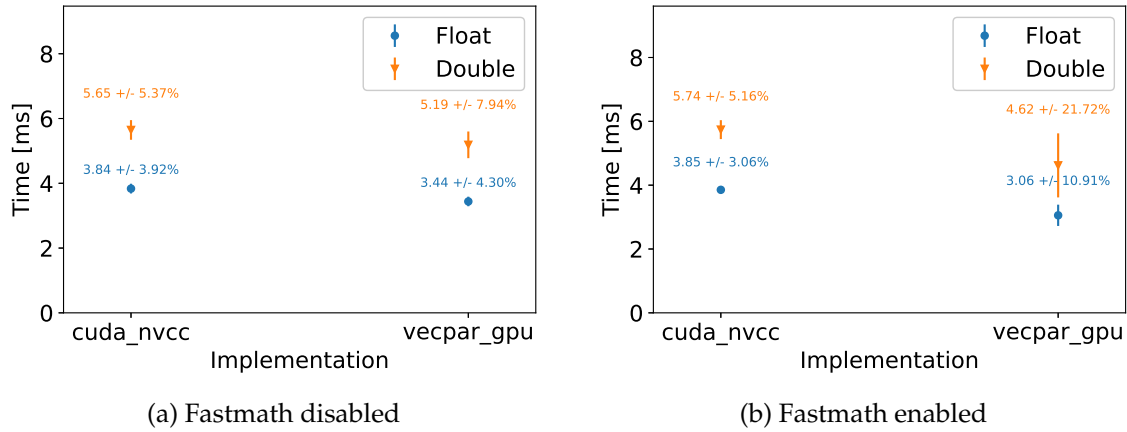


Figure 6.19.: Mean and standard deviation when using the RKN stepper and *eigen* backend, in single and double precision, with grid configuration 157×64 CUDA threads, on Env2

The speedup diagram in Figure 6.20 shows that `vecpar` implementation (`vecpar_gpu`) is slightly faster than the native CUDA version (`cuda_nvcc`) compiled with `nvcc`. This advantage is assumed to be due to NVPTX optimizations done by `clang` since compiling the same RKN kernel with `clang` already shows a slight speedup over the executable produced by `nvcc`.

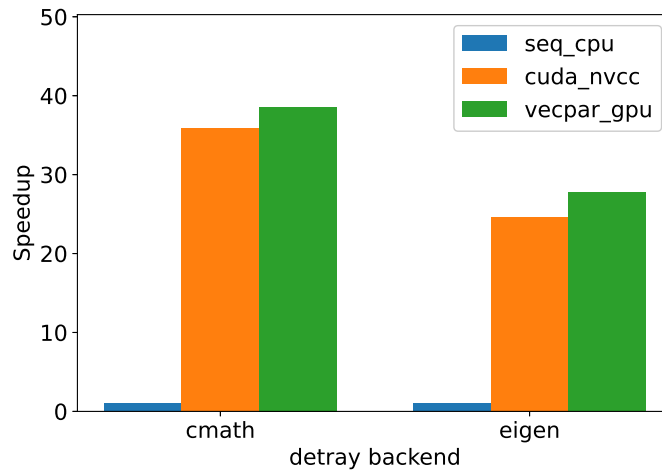


Figure 6.20.: Speed-up factors over initial sequential version (`seq_cpu`) for GPU CUDA (`cuda_nvcc`) and `vecpar` (`vecpar_gpu`), using `detray` `cmath`/`eigen` backends in double precision, with fastmath disabled, on Env2

Figure 6.21 summarizes these experiments with the conclusion that for this given scenario, the `vecpar` single-source implementation ensures an order of magnitude speedup over the initial sequential implementation, comparable to native OpenMP and CUDA implementations.

Next, we evaluated the performance of the `vecpar` implementation in an extreme-load test using one million tracks instead of just 10 000. The results in Figure 6.22 show that `vecpar` implementation is comparable with native OpenMP and CUDA, with the GPU

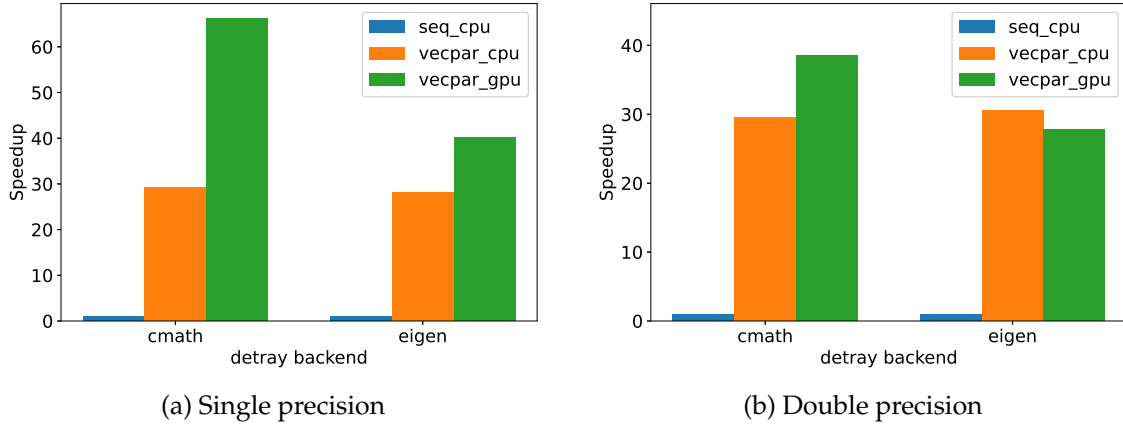


Figure 6.21.: Speedup for vecpar implementation over the base (sequential) CPU implementation, using cmath/eigen backends, in single/double precision, on Env2

implementations showing a speedup factor up to 70× over the sequential one; the speedup on CPU is limited by the number of threads (which is 16 in this case) while further runs with more threads showed no visible time improvement.

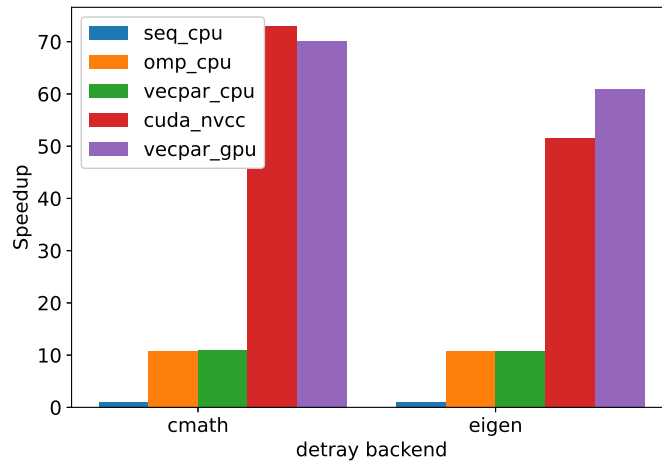


Figure 6.22.: Speedup diagram for simplified RKN stepper using cmath/eigen backends, in single precision, one million tracks, with fastmath disabled, on Env1

6.4.2. A More Realistic RKN Stepper

A more recent implementation¹⁹ of a realistic RKN stepper in *detray* provides functionality to propagate both track parameters (either using free or bound states), and their covariances. The full listings for these two algorithms (the free and bound ones) are available in Appendix B.3. While the former implements a `parallelizable_rmap` algorithm, which updates the input track states array with the newly computed values, the latter uses a `parallelizable_map` and does not change the input array. Moreover, when using bound parameters, an extra context parameter is required for the coordinate transformations

¹⁹*Detray* repository version from July 2022

from 3D space to a 2D plane and vice-versa.

Due to the requirement of computing the partial derivatives of the track parameters, the complexity is increased further. Since the double precision use case is more realistic for a production environment, we focus the experiments on this level of precision, while disabling fastmath support. Again, we assume the motion happens in a constant magnetic field.

For the ARM CPU on Env3, Figure 6.23a shows that the vecpar implementations are within $\pm 9\%$ of the native OpenMP ones, while being $3.55\times$ and $3.73\times$ faster than the sequential ones for *cmath* and *eigen* backends. This is close to the maximum theoretical speedup of $4\times$, which is limited by the number of OpenMP threads. For an *x86_64* platform, the vecpar implementations are within $\pm 3\%$ when compared with the OpenMP one while being $15\times$ and respectively $19\times$ faster than the sequential one for the *cmath* and *eigen* backend respectively, as shown in Figure 6.23.

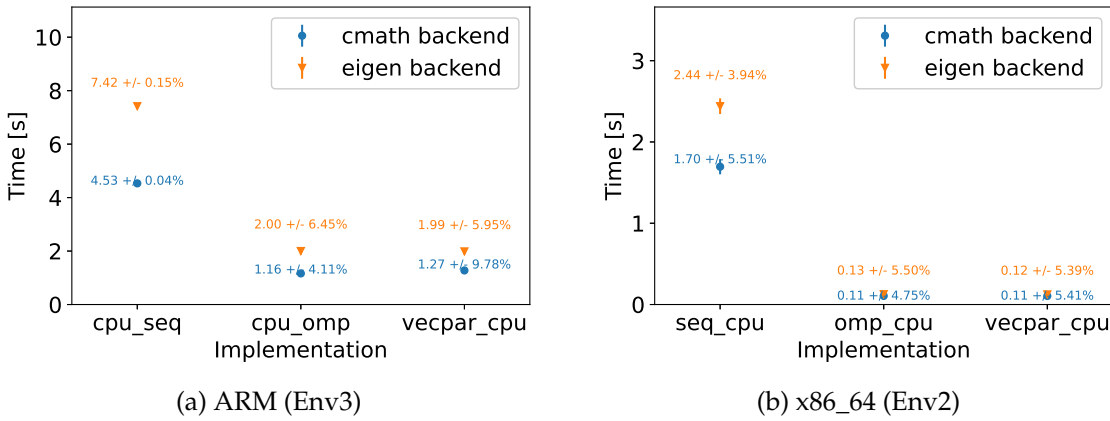


Figure 6.23.: Mean and standard deviation when using the RKN stepper (with error propagation) for 10 000 free track parameters, in double precision, on CPU Env2 and Env3

For the GPU platform, the existing CUDA implementation uses managed memory; vecpar can also support this, but additionally, the arrays can be initially stored in CPU memory relying on the framework to copy them to the device if/when needed. Therefore Figure 6.24 shows the runtimes for the benchmark (`mng_cuda_nvcc`), and two vecpar implementations (`mng_vecpar_gpu` and `host_vecpar_gpu`), which use vecmem’s host memory allocator and managed memory allocator respectively. Despite much better memory and computation throughput, there are several uncoalesced global accesses due to unnecessary duplication of the offloaded algorithmic code, which seem to make vecpar slower than the native approach. This is expected to be alleviated when using the new `__grid__constant` feature of CUDA 11.7, which allows storing read-only objects in constant cache²⁰. This will relieve thread’s local memory from the need to load the vecpar algorithm object. This also explains why this behaviour was not seen in the previous

²⁰The code is already implemented in vecpar, but clang 14 does not support CUDA 11.7 so we could not perform the validation

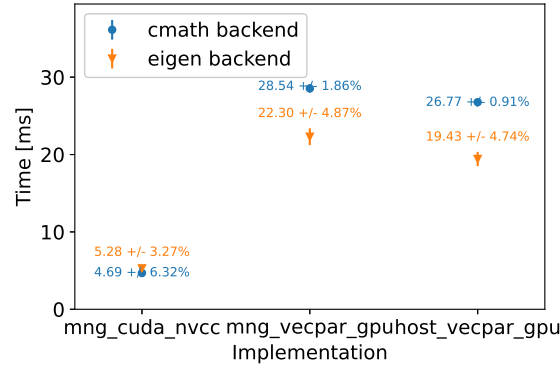


Figure 6.24.: Mean and standard deviation for RKN stepper (with error propagation) for 10 000 free track parameters, data in host/managed memory, in double precision, on Env2

experiments when the algorithm was less complex and required less memory.

Nevertheless, there is still a benefit of having a single parallel implementation that can be executed on different platforms as it is shown in Figure 6.25. Speedup factors up to 19 and 108 can be observed for CPU and GPU respectively when compared to the sequential version.

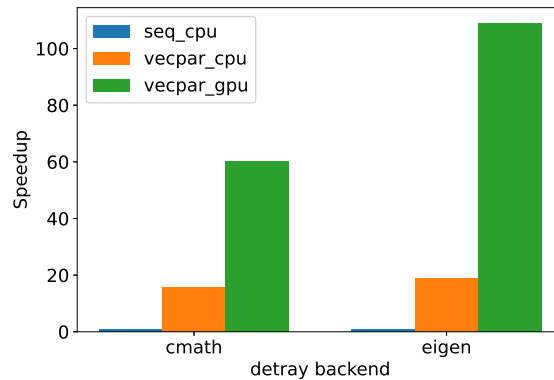


Figure 6.25.: Speedup of the vecpar implementation for RKN stepper (with error propagation) for 10 000 tracks over the sequential implementation for cmath and eigen backends, in double precision, on Env2

Additionally, we can also generate a track with the appropriate physics parameters (local position, momentum and charge) and a 2D plane that the trajectory would intersect in a first estimation step. In this setup, the RKN stepper will use bound track parameters instead of free track parameters. Figure 6.26 shows almost identical times for vecpar implementation vs CUDA native implementation in case one track state is updated. A test case that works with more planes, would require a geometry description in order to identify the intersected surfaces; at the time of our tests, this infrastructure was pending development in the *detray* project.

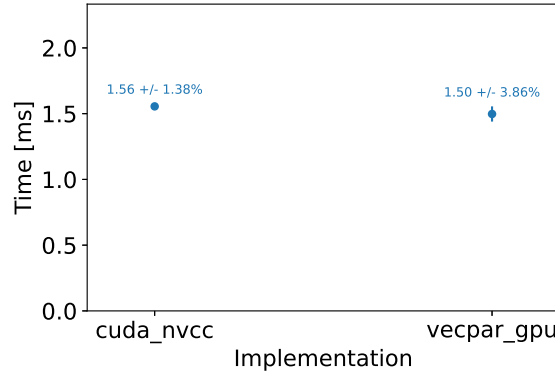


Figure 6.26.: Mean and standard deviation when using RKN stepper (with error propagation) for one bound track parameter stored in managed memory, in double precision, using cmath backend on Env2

To evaluate `vecpar`'s OpenMP target backend, the call to `vecpar::parallel_algorithm` was replaced by `vecpar::ompt::parallel_algorithm`. Figure 6.27 shows the comparison between the runtime for estimating 10 000 tracks on a CPU using OpenMP threads and on an AMD GPU using `vecpar`. The initial parameters are stored in CPU memory, so when running on the GPU, the `vecpar` OpenMP target backend handles the memory transfers in both directions. This should be taken in consideration when evaluating the performance on the GPU, which in this case, is comparable to a multi-threaded CPU but not better. Nevertheless, this experiment shows that `vecpar` reached the expected portability level, while having one C++ implementation (the one listed in Appendix B.3).

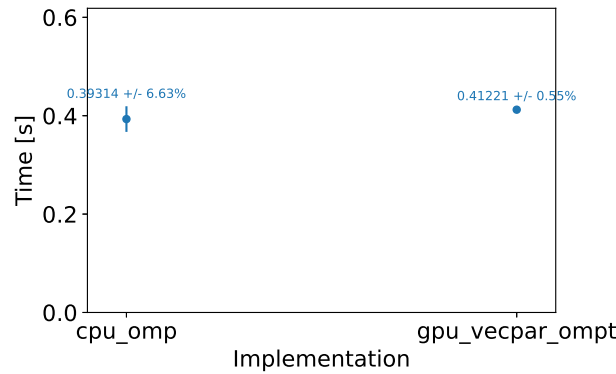


Figure 6.27.: Mean and standard deviation when using RKN stepper (with error propagation) for 10 000 free track parameters, in double precision, using cmath backend on Env4

To summarize this chapter, we showed that the `vecpar` framework is as efficient as other competitive APIs while keeping the domain specific code separated from the parallelization strategies. For the GPU platform, there are two corner cases for which `vecpar` currently underperforms, but ongoing developments on improving (a) the memory coalescing for CUDA and (b) the parallel patterns for reductions, are expected to minimize the overhead.

7. Conclusion and Future Work

In this chapter, the current work is summarized while highlighting its potential and limitations, together with a series of ongoing and future developments.

7.1. Contribution Summary

The parallelization coding tasks together with the performance evaluation reports using the ACTS framework described in Section 3.1.1 contributed to the development of the heterogeneous R&D projects that aim to redesign the data flow and to rewrite the reconstruction code so that it can make use of parallel execution capabilities on CPU and GPU. Using the lessons learned through this step, we proposed two different approaches to automatic parallelization in order to relieve physics experts from the burden to maintain different repositories targeting different hardware platforms. Firstly, the *clang-offload* source-to-source translator introduced in Section 3.3 proved its strength by having the ability to modify and/or rewrite the abstract syntax tree of a C++ file based on predefined templates; while the tool generates different sources for CPU and GPU, the repository can only hold the C++ sequential implementation and generate the appropriate executables when needed. Nevertheless, this solution proved difficult to generalize when more complex parallelization patterns (like reductions) were in place. Secondly, and the most important contribution, is the development of the *vecpar* framework, introduced in Chapter 4. This allows a C++ file (written using the *vecpar* abstractions) to be compiled for multiple targets with close-to-native runtimes as shown in Chapter 6. We identified a few corner cases in which *vecpar* induces a larger overhead and discussed and/or prototyped solutions to alleviate this. Additionally, we ensure the reproducibility of our results by (a) having all the code publicly available on github and (b) describing the test methodology in Appendix D. The framework’s potential and limitations are summarized next.

7.1.1. Potential

Even though the *vecpar* framework was designed having the particle reconstruction use cases in mind, it is not limited to this application domain. Any problem that can be described in terms of a `vecpar::algorithm` and `vecpar::parallel_algorithm(...)`, can be parallelized on CPU and GPU. As detailed in Section 2.6, the scientific world is adopting more functional programming concepts lately with the increasing number of use cases for data parallelism.

Additionally, by having platform-specific implementations, *vecpar* extracts the complexity of addressing multiple architectures and therefore it can simplify the development of a scientific application by requiring only a single-source platform-agnostic implementation. The domain experts can provide the algorithms in C++ and by calling `vecpar::parallel_algorithm()` they pass the responsibility for parallelization and potential data transfers to the framework. Knowing that *vecpar* adds a minimum overhead to native implementations, it can speedup the process of benchmarking to evaluate whether there are significant advantages to using a specific technology choice. In case there is, scientists can then decide to (a) implement the algorithms using OpenMP or CUDA, or (b) use *vecpar* as a production-ready solution (which eases the ramp-up process for the junior developers since no additional languages need to be learned). Also, adding new *vecpar* backends to target new future architectures can be done transparently for the *vecpar* developers and without any impact on the domain scientific code that is decoupled from the parallelization strategies.

One of the major benefits of having the code written in a standardized API (like OpenMP) is the guarantee for long-term support in all major compilers. Moreover with the development of the OpenMP target runtime in LLVM/clang mentioned in Section 2.5, a scientific application will soon be able to target GPUs from any vendor, just by recompiling the code that calls the `vecpar::omp::parallel_algorithm` backend with the help of the new compiler. No further code changes will be required to ensure portability. Therefore, coding productivity is maximized by reducing the development effort.

7.1.2. Limitations

In order to use *vecpar*, besides the code porting to fit the above mentioned abstractions, the data structures must also be converted to *vecmem* container types¹. Moreover, the size of a collection must be known at compile-time and the object types must be *default constructible*; these are due to the GPU static memory allocation restriction.

There is currently limited support for hierarchical parallelism. By default, each backend can parallelize over one direction; however, there is the option to imbricate the algorithms as follows: an `vecpar::omp::parallel_algorithm` could use an algorithm that calls `vecpar::cuda::parallel_algorithm`. Nevertheless, the execution is not expected to be optimal since each OpenMP thread will use the same (default) CUDA stream to communicate with the GPU and therefore create a potential bottleneck.

Also, at the moment, the OpenMP target backend does not fully support all the functionality of *vecpar*. We prototyped efficient versions of a *map-reduce* algorithm but this still has to be generalized to all use cases.

Another point would be regarding the concept of using "code as data", which is highly exploited in functional programming. For the CUDA backend, *vecpar* sends the algorithm

¹Nevertheless, since these are extensions of the `std::vector`, the conversion is not expected to introduce notable time delays.

as an argument to the global kernel function, which is then stored on the stack. When its size is limited (like in the SAXPY/DAXPY or BabelStream examples), there is no real impact on the overall performance. But when this is much larger in size (like in the track reconstruction use case), it occupies most of the registers, creating uncoalesced memory reads due to the need to constantly use both local and global memory. We estimate this side-effect will be alleviated by storing the algorithm in CUDA's constant cache, which should free the other caches for the rest of the arguments. This still has to be tested with a compiler that supports the `__grid_constant__` descriptor (CUDA 11.7).

Lastly, there are several limitations with respect to the compilation process. First, since `vecpar` relies on features from the C++20 standard, the compilation must be performed by a compiler that supports this; examples of such compilers and their minimum versions include LLVM/clang 10.x (2020), GNU/gcc 10.x (2020) and NVIDIA nvcc 12.x (2022); therefore, while many C++ compiler versions can be used, the limitation is more on the GPU side, when newer compilers are required. Second, as mentioned previously, LLVM/clang is the only C++ compiler that is able to build all the `vecpar` backends; the alternative is to use any C++ compiler² for `vecpar::omp` and `vecpar::ompt` backends and an nvidia compiler for `vecpar::cuda` backend. Third, for performance purposes, a compiler with support for newer OpenMP standards is preferred; a notable example is LLVM/clang 15.x.

7.2. Future Work

While an exhaustive list of development tasks for `vecpar` is presented in Appendix C, we would like to draw the attention on a few significant changes. Firstly, the optimizations prototyped for various edge-cases will be extended to all the `vecpar` backends; examples of these include using shared memory in CUDA blocks for all the operations and fusing composed operators (e.g. map-reduce and map-filter) in order to avoid an extra pass over the input array(s). These should bring important speedups for GPUs. Secondly, make use of latest OpenMP 5.2 specifications to write custom implementations for each accelerator family; this will allow the compilers to do an extra set of optimizations. Thirdly, adding new capabilities like the support for imbricated parallelism would increase the framework's readiness to be used in production-like environments. Finally, making the installation easier on any architecture by providing a spack package, would increase the number of developers/projects interested in testing it.

Since the main motivation for this work resides in the domain of particle reconstruction, a next milestone would be to port more steps of this workflow so that `vecpar` could compete with other approaches that are currently being evaluated.

²For the OpenMP Target, the compiler has to be built and configured to provide offloading support.

7.3. Development Experience Report

Based on the development effort conducted for this thesis, we would like to summarize a few observations with respect to our experience. Firstly, using the Equations (4.11) to optimize the operations most likely will not guarantee bitwise reproducibility due to the non-associative nature of the floating point operations. Secondly, the OpenMP target ecosystem (e.g. compilers, tools and profilers) has improved significantly in the last 3 years, with support for NVIDIA GPU being much better than the one for AMD ones. Additionally, the level of OpenMP support is different in every compiler so one is required to have multiple implementations to ensure the best performance. Thirdly, there are many inconsistencies throughout the compilers regarding the ways to configure a build for a GPU; the flags are named differently, the expected values are sometimes ambiguous, and the documentation regarding the correlation between them is usually sparse or is missing all together. For example, when using the LLVM/clang compiler, setting `-fno-fast-math` flag without setting `-ffp-contract` to `OFF`, will still instruct the compiler to use aggressive optimizations for floating point operations; the GNU/gcc compiler does not have a description of the flags' behaviour in their documentation.

7.4. Conclusion

The *vecpar* framework provides *performance-portable* C++ abstractions that allow code execution on multiple platforms with no prerequisite knowledge about hardware architectures from the domain scientists.

Bibliography

- [Aaij et al., 2020] Aaij, R., Albrecht, J., Belous, M., Billoir, P., Boettcher, T., Rodríguez, A. B., vom Bruch, D., Pérez, D. H. C., Vidal, A. C., Craik, D. C., Declara, P. F., Funke, L., Gligorov, V. V., Jashal, B., Kazeev, N., Santos, D. M., Pisani, F., Pliushchenko, D., Popov, S., Quagliani, R., Rangel, M., Reiss, F., Mayordomo, C. S., Schwemmer, R., Sokoloff, M., Stevens, H., Ustyuzhanin, A., Cardona, X. V., and Williams, M. (2020). Allen: A high-level trigger on GPUs for LHCb. *Computing and Software for Big Science*, 4(1). (Cited on pages 18, 118, and 181)
- [Acun et al., 2014] Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., Wesolowski, L., and Kale, L. (2014). Parallel programming with migratable objects: Charm++ in practice. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 647–658. (Cited on page 111)
- [Ai et al., 2022] Ai, X., Allaire, C., Calace, N., Czirkos, A., Ene, I., Elsing, M., Farkas, R., Gagnon, L.-G., Garg, R., Gessinger, P., Grasland, H., Gray, H. M., Gumpert, C., Hrdinka, J., Huth, B., Kiehn, M., Klimpel, F., Krasznahorkay, A., Langenberg, R., Leggett, C., Niermann, J., Osborn, J. D., Salzburger, A., Schlag, B., Tompkins, L., Yamazaki, T., Yeo, B., Zhang, J., Mania, G., Kolbinger, B., Moyse, E., and Rousseau, D. (2022). A common tracking software project. *Computing and Software for Big Science*. (Cited on page 45)
- [Ai et al., 2021] Ai, X., Mania, G., Gray, H. M., Kuhn, M., and Styles, N. (2021). A GPU-Based Kalman Filter for Track Fitting. *Computing and Software for Big Science*. (Cited on pages 53, 55, 56, 57, and 180)
- [Albrecht et al., 2019] Albrecht, J., Alves, A. A., Amadio, G., Andronico, G., Anh-Ky, N., Aphecetche, L., Apostolakis, J., Asai, M., Atzori, L., and et al. (2019). A Roadmap for HEP Software and Computing R&D for the 2020s. *Computing and Software for Big Science*, 3(1). (Cited on page 18)
- [Albrecht et al., 2018] Albrecht, J., Bloom, K., Boccali, T., Boveia, A., De Cian, M., Doglioni, C., Dziurda, A., Farbin, A., Fitzpatrick, C., Gaede, F., George, S., Gligorov, V., Grasland, H., Grillo, L., Hegner, B., Kalderon, W., Kama, S., Koppenburg, P., Krutelyov, S., Kutschke, R., Lampl, W., Lange, D., Moyse, E., Norman, A., Petric, M., Polci, F., Potamianos, K., Ratnikov, F., Raven, G., Ritter, M., Rizzi, A., Rodrigues, E., Rousseau, D., Salzburger, A., Kennedy, L. S., Sokoloff, M. D., Stewart, G., Ustyuzhanin, A., Viren,

- B., Williams, M., Winklmeier, F., and Wuerthwein, F. (2018). Hep community white paper on software trigger and event reconstruction. (Cited on page 19)
- [AMD, 2020] AMD (2020). AMD Instinct MI100 accelerator. Technical report. Accessed on October 2022. (Cited on page 7)
- [AMD, 2021] AMD (2021). AMD INSTINCT MI200 SERIES ACCELERATOR. Technical report. Accessed on October 2022. (Cited on page 7)
- [AMD, 2022a] AMD (2022a). AMD CDNA Whitepaper. Online, <https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf>, Accessed on June 2022. (Cited on pages 36 and 179)
- [AMD, 2022b] AMD (2022b). AMD ROCm Open Ecosystem. Online, <https://www.amd.com/en/graphics/servers-solutions-rocm>, Accessed in October 2022. (Cited on pages 37 and 179)
- [Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring)*, page 483–485, New York, NY, USA. Association for Computing Machinery. (Cited on page 122)
- [Anderson et al., 1999] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition. (Cited on page 27)
- [Argonne National Laboratory, 2022] Argonne National Laboratory (2022). AURORA. Online, <https://alcf.anl.gov/aurora>. Accessed on October 2022. (Cited on page 8)
- [Artigues et al., 2020] Artigues, V., Kormann, K., Rampp, M., and Reuter, K. (2020). Evaluation of performance portability frameworks for the implementation of a particle-in-cell code. *Concurrency and Computation: Practice and Experience*, 32(11):e5640. e5640 cpe.5640. (Cited on pages 113 and 180)
- [ATLAS Collaboration, 2022a] ATLAS Collaboration (2022a). ATLAS Experiment - Detector and Technology. Online <https://atlas.cern/Discover/Detector>, Accessed in June 2022. (Cited on pages 12 and 179)
- [ATLAS Collaboration, 2022b] ATLAS Collaboration (2022b). ATLAS Software and Computing HL-LHC Roadmap. Technical report, CERN, Geneva. (Cited on pages 19 and 179)
- [ATLAS Collaboration, 2022c] ATLAS Collaboration (2022c). Public ATLAS Luminosity Results for Run-3 of the LHC. Technical report. Accessed on October 2022. (Cited on pages 10 and 179)
-

-
- [Balogh et al., 2018] Balogh, G., Mudalige, G., Regul, I., Antao, S., and Bertolli, C. (2018). Op2-clang: A source-to-source translator using clang/llvm libtooling. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 59–70. (Cited on page 115)
- [Beckingsale et al., 2019] Beckingsale, D., Scogland, T. R. W., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., Pearce, O., Robinson, P., and Ryujin, B. S. (2019). RAJA: portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC, P3HPC@SC 2019, Denver, CO, USA, November 22, 2019*, pages 71–81. IEEE. (Cited on pages 109 and 110)
- [Billoir, 1984] Billoir, P. (1984). Track Fitting With Multiple Scattering: A New Method. *Nucl. Instrum. Meth.*, A225:352–366. (Cited on page 16)
- [Bird, 1987] Bird, R. S. (1987). An introduction to the theory of lists. In Broy, M., editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42, Berlin, Heidelberg. Springer Berlin Heidelberg. (Cited on page 73)
- [Bocci et al., 2020] Bocci, A., Kortelainen, M., Innocente, V., Pantaleo, F., and Rovere, M. (2020). Heterogeneous reconstruction of tracks and primary vertices with the cms pixel tracker. (Cited on pages 18, 118, 119, and 181)
- [Braam, 2019] Braam, P. (2019). The lustre storage architecture. (Cited on page 5)
- [CERN, 2017] CERN (2017). CERN Brochure. Online, https://home.cern/sites/default/files/2018-07/CERN-Brochure-2017-002-Eng_0.pdf. (Cited on page 10)
- [Chapman et al., 2008] Chapman, B. M., Jost, G., and van der Pas, R. (2008). *Using OpenMP - portable shared memory parallel programming*. Scientific and engineering computation. MIT Press. (Cited on page 23)
- [Codeplay Software, 2022] Codeplay Software (2022). Online, <https://github.com/codeplaysoftware/syclacademy>, Accessed on June 2022. (Cited on pages 30 and 179)
- [Deakin et al., 2020] Deakin, T., Poenaru, A., Lin, T., and McIntosh-Smith, S. (2020). Tracking performance portability on the yellow brick road to exascale. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 1–13. (Cited on pages 114, 115, and 180)
- [Deakin et al., 2018] Deakin, T., Price, J., Martineau, M., and McIntosh-Smith, S. (2018). Evaluating attainable memory bandwidth of parallel programming models via Babel-Stream. *International Journal of Computational Science and Engineering*, 17(3):247–262. Special Issue on Novel Strategies for Programming Accelerators. (Cited on pages 114, 123, and 129)
-

- [Doerfert et al., 2021] Doerfert, J., Huber, J., and Cornelius, M. (2021). *Advancing OpenMP Offload Debugging Capabilities in LLVM*. Association for Computing Machinery, New York, NY, USA. (Cited on page 41)
- [Doerfert et al., 2023] Doerfert, J., Jasper, M., Huber, J., Abdelaal, K., Georgakoudis, G., Scogland, T., and Parasyris, K. (2023). Breaking the vendor lock: Performance portable programming through openmp as target independent runtime layer. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT '22*, page 494–504, New York, NY, USA. Association for Computing Machinery. (Cited on pages 41 and 179)
- [Dong et al., 2021] Dong, Z., Gray, H., Leggett, C., Lin, M., Pascuzzi, V. R., and Yu, K. (2021). Porting hep parameterized calorimeter simulation code to gpus. (Cited on page 18)
- [Dongarra and Luszczek, 2011] Dongarra, J. and Luszczek, P. (2011). *TOP500*, pages 2055–2057. Springer US, Boston, MA. (Cited on pages 1 and 6)
- [Edwards et al., 2014] Edwards, H. C., Trott, C. R., and Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. (Cited on page 107)
- [Evans et al., 2022] Evans, J., Andersch, M., Sethi, V., Brito, G., and Mehta, V. (2022). Online, <https://developer.nvidia.com/blog/nvidia-grace-hopper-superchip-architecture-in-depth/>, Accessed on November 2022. (Cited on pages 35 and 179)
- [Fandrey, 2010] Fandrey, D. (2010). Clang/LLVM Maturity Report. Moltkestr. 30, 76133 Karlsruhe - Germany. Computer Science Dept., University of Applied Sciences Karlsruhe. (Cited on page 28)
- [Flynn, 2011] Flynn, M. (2011). *Flynn's Taxonomy*, pages 689–697. Springer US, Boston, MA. (Cited on page 2)
- [Fowler, 2011] Fowler, M. (2011). *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley. (Cited on page 101)
- [Fujitsu, 2020] Fujitsu (2020). FUJITSU Processor A64FX. Technical report. Accessed on October 2022. (Cited on page 7)
- [Gamblin et al., 2015] Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., and Futral, S. (2015). The spack package manager: Bringing order to hpc software chaos. In *Proceedings of the International Conference for High Performance*
-

- Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA. Association for Computing Machinery. (Cited on page 5)
- [Gayatri et al., 2018] Gayatri, R., Yang, C., Kurth, T., and Deslippe, J. (2018). A case study for performance portability using openmp 4.5. In Chandrasekaran, S., Juckeland, G., and Wienke, S., editors, *Accelerator Programming Using Directives - 5th International Workshop, WACCPD 2018, Dallas, TX, USA, November 11-17, 2018, Proceedings*, volume 11381 of *Lecture Notes in Computer Science*, pages 75–95. Springer. (Cited on pages 112, 113, and 180)
- [Gessinger, 2022] Gessinger, P. (2022). Event data model: Upcoming developments. Online, https://indico.cern.ch/event/1184037/contributions/5053609/attachments/2516564/4326715/2022-09-27-acts-ws-edm_v2.pdf. (Cited on page 47)
- [Gessinger et al., 2023] Gessinger, P., Grasland, H., Gray, H., Joubé, S., Kusiak, K., Krasznahorkay, A., Leggett, C., Mania, G., Niermann, J., Salzburger, A., Styles, N., Swatman, S. N., and Yeo, B. (2023). CTD2022: traccc - GPU Track reconstruction demonstrator for HEP. In *7th International Connecting the Dots Workshop, Princeton, USA, Proceedings*. Zenodo. (Cited on pages 47, 48, and 179)
- [Guennebaud et al., 2010] Guennebaud, G., Jacob, B., et al. (2010). Eigen v3. <http://eigen.tuxfamily.org>. (Cited on page 27)
- [Gustafson, 1988] Gustafson, J. L. (1988). Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533. (Cited on page 122)
- [Gustafson, 2011] Gustafson, J. L. (2011). *Amdahl's Law*. Springer US, Boston, MA. (Cited on page 122)
- [Hager and Wellein, 2011] Hager, G. and Wellein, G. (2011). *Introduction to High Performance Computing for Scientists and Engineers*. Chapman and Hall / CRC computational science series. CRC Press. (Cited on page 4)
- [Halkiadakis, 2010] Halkiadakis, E. (2010). Proceedings for tasi 2009 summer school on "physics of the large and the small": Introduction to the lhc experiments. (Cited on page 9)
- [Henriksen et al., 2014] Henriksen, T., Elsmann, M., and Oancea, C. E. (2014). Size slicing: A hybrid approach to size inference in futhark. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC '14*, pages 31–42, New York, NY, USA. ACM. (Cited on page 116)
- [Henriksen et al., 2017] Henriksen, T., Serup, N. G. W., Elsmann, M., Henglein, F., and Oancea, C. E. (2017). Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. In Cohen, A. and Vechev, M. T., editors, *Proceedings of*
-

- the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 556–571. ACM. (Cited on pages 116, 117, and 180)
- [Huber et al., 2022] Huber, T., Pophale, S., Baker, N., Carr, M., Rao, N., Reap, J., Holsapple, K., Davis, J. H., Burnus, T., Lee, S., Bernholdt, D. E., and Chandrasekaran, S. (2022). Ecp solve: Validation and verification testsuite status update and compiler insight for openmp. (Cited on page 26)
- [IEEE, 2008] IEEE (2008). IEEE 754-2008 Standard for Floating-Point Arithmetic. (Cited on pages 5 and 54)
- [Ifrim et al., 2022] Ifrim, I., Vassilev, V., and Lange, D. J. (2022). GPU accelerated automatic differentiation with clad. *CoRR*, abs/2203.06139. (Cited on page 115)
- [Intel Corporation, 2022] Intel Corporation (2022). OneAPI Specification - Software Architecture. Online, <https://spec.oneapi.io/versions/latest/architecture.html>, Accessed in October 2022. (Cited on pages 39 and 179)
- [Jones et al., 2015] Jones, C. D., Contreras, L., Gartung, P., Hufnagel, D., and Sexton-Kennedy, L. (2015). Using the CMS threaded framework in a production environment. *Journal of Physics: Conference Series*, 664(7):072026. (Cited on page 18)
- [Khronos Group, 2022] Khronos Group (2022). Online, <https://www.khronos.org/sycl/>, Accessed on June 2022. (Cited on pages 31 and 179)
- [Koenneker, 2022] Koenneker, Y. (2022). Performance study on GPU offloading techniques using the Gauß matrix inverse algorithm. Online https://wr.informatik.uni-hamburg.de/_media/research:theses:yannik_koenneker_performance_study_on_gpu_offloading_techniques_using_the_gauss_matrix_inverse_algorithm.pdf. (Cited on pages 61, 63, 64, 65, and 180)
- [Larkin, 2022] Larkin, J. (2022). Developing HPC Applications with Standard C++, Fortran, and Python. Online <https://www.nvidia.com/en-us/on-demand/session/gtcfall122-a41087/>. NVIDIA GTC. (Cited on page 23)
- [Lattner, 2002] Lattner, C. (2002). LLVM: An Infrastructure for Multi-Stage Optimization. Master’s thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. <http://llvm.cs.uiuc.edu>. (Cited on page 40)
- [Leggett et al., 2017] Leggett, C., Baines, J., Bold, T., Calafiura, P., Farrell, S., van Gemmeren, P., Malon, D., Ritsch, E., Stewart, G., Snyder, S., Tsulaia, V., and and, B. W. (2017). AthenaMT: upgrading the ATLAS software framework for the many-core world with multi-threading. *Journal of Physics: Conference Series*, 898:042009. (Cited on page 18)

-
- [LHC Education and Group, 2017] LHC Education, C. and Group, O. (2017). Online, https://home.cern/sites/default/files/2018-07/CERN-Brochure-2017-002-Eng_0.pdf, Accessed in June 2022. (Cited on pages 11 and 179)
- [Lin et al., 2022] Lin, W.-C., Deakin, T., and McIntosh-Smith, S. (2022). Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems held in conjunction with Supercomputing (PMBS)*. IEEE. in press. (Cited on page 41)
- [Lopienska, 2022] Lopienska, E. (2022). The CERN accelerator complex, layout in 2022. Complexe des accélérateurs du CERN en janvier 2022. General Photo. (Cited on pages 9 and 179)
- [Lu et al., 2022] Lu, W., Shan, B., Raut, E., Meng, J., Araya-Polo, M., Doerfert, J., Malik, A. M., and Chapman, B. (2022). Towards efficient remote openmp offloading. In Klemm, M., de Supinski, B. R., Klinkenberg, J., and Neth, B., editors, *OpenMP in a Modern World: From Multi-device Support to Meta Programming*, pages 17–31, Cham. Springer International Publishing. (Cited on page 41)
- [Ludwig, 2020] Ludwig, T. (2020). Lecture notes on Scientific Computing. Online, https://wr.informatik.uni-hamburg.de/teaching/wintersemester_2019_2020/hochleistungsrechnen. (Cited on pages 1 and 121)
- [Lund et al., 2009] Lund, E., Bugge, L., Gavrilenko, I., and Strandlie, A. (2009). Track parameter propagation through the application of a new adaptive runge-kutta-nyström method in the ATLAS experiment. *Journal of Instrumentation*, 4(04):P04001–P04001. (Cited on pages 15 and 179)
- [Marangoni and Wischgoll, 2016] Marangoni, M. and Wischgoll, T. (2016). Paper: Toggpu: Automatic source transformation from C++ to CUDA using clang/llvm. In Kao, D. L., Wischgoll, T., and Zhang, S., editors, *Visualization and Data Analysis 2016, San Francisco, California, USA, February 14-18, 2016*, pages 1–9. Ingenta. (Cited on page 115)
- [McCarthy, 1960] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195. (Cited on page 43)
- [Meng, 2021] Meng, L. (2021). Atlas itk pixel detector overview. (Cited on pages 13 and 179)
- [Moore, 1998] Moore, G. E. (1998). Cramming more components onto integrated circuits. *Proc. IEEE*, 86(1):82–85. (Cited on page 1)
- [Murphy, 2012] Murphy, K. P. (2012). *Machine learning - a probabilistic perspective*. Adaptive computation and machine learning series. MIT Press. (Cited on page 16)
-

- [NVIDIA Corporation, a] NVIDIA Corporation. CUDA C Programming Guide. Online <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Online, Accessed on June 2022. (Cited on pages 3, 32, and 179)
- [NVIDIA Corporation, b] NVIDIA Corporation. CUDA Zone. Online <https://developer.nvidia.com/cuda-zone>. Online, Accessed on June 2022. (Cited on page 32)
- [NVIDIA Corporation, c] NVIDIA Corporation. HPC-SDK. Online, <https://developer.nvidia.com/hpc-sdk>. Online, Accessed on June 2022. (Cited on pages 32 and 179)
- [NVIDIA Corporation, 2017] NVIDIA Corporation (2017). TESLA V100 PCIe GPU Accelerator. Technical report. Accessed on October 2022. (Cited on page 7)
- [NVIDIA Corporation, 2021] NVIDIA Corporation (2021). NVIDIA A100 TENSOR CORE GPU. Technical report. Accessed on October 2022. (Cited on page 7)
- [Oak Ridge National Laboratory, 2022] Oak Ridge National Laboratory (2022). Frontier. Online, <https://www.olcf.ornl.gov/frontier/>. Online, Accessed on June 2022. (Cited on pages 3 and 179)
- [Patel et al., 2021] Patel, A., Tian, S., Doerfert, J., and Chapman, B. (2021). *A Virtual GPU as Developer-Friendly OpenMP Offload Target*. Association for Computing Machinery, New York, NY, USA. (Cited on page 41)
- [Pennycook et al., 2016] Pennycook, S. J., Sewall, J. D., and Lee, V. W. (2016). A metric for performance portability. *CoRR*, abs/1611.07409. (Cited on page 114)
- [Piętak and Kisiel-Dorohinicki, 2013] Piętak, K. and Kisiel-Dorohinicki, M. (2013). *Agent-Based Framework Facilitating Component-Based Implementation of Distributed Computational Intelligence Systems*, pages 31–44. Springer Berlin Heidelberg, Berlin, Heidelberg. (Cited on page 77)
- [Rohr et al., 2017] Rohr, D., Gorbunov, S., and and, V. L. (2017). GPU-accelerated track reconstruction in the ALICE high level trigger. *Journal of Physics: Conference Series*, 898:032030. (Cited on page 18)
- [Salzburger, 2022] Salzburger, A. (2022). Acts : Project overview and status. Online, <https://indico.cern.ch/event/1184037/contributions/4993021/attachments/2515482/4324609/2022-09-25-WS-ACTS-Status.pdf>. (Cited on pages 17, 46, and 179)
- [Salzburger et al., 2023] Salzburger, A., Niermann, J., Yeo, B., and Krasznahorkay, A. (2023). Detray: a compile time polymorphic tracking geometry description. *Journal of Physics: Conference Series*, 2438(1):012026. (Cited on page 48)
-

-
- [Shiers, 2007] Shiers, J. (2007). The worldwide lhc computing grid (worldwide lcg). *Computer Physics Communications*, 177(1):219–223. Proceedings of the Conference on Computational Physics 2006. (Cited on page 10)
- [Stephen Jones, 2017] Stephen Jones (2017). Cuda optimization tips, tricks and techniques. Online <https://on-demand.gputechconf.com/gtc/dc/2017/presentation/dc7112-stephen-jones-cuda-optimization-tips-tricks-and-techniques.pdf>. NVIDIA GTC. (Cited on pages 4 and 179)
- [Swatman et al., 2023] Swatman, S. N., Krasznahorkay, A., and Gessinger, P. (2023). Managing heterogeneous device memory using c++17 memory resources. *Journal of Physics: Conference Series*, 2438(1):012050. (Cited on page 48)
- [TOP500, 2022] TOP500 (2022). Top500 list. Accessed in November 2022. (Cited on pages 1, 6, 7, 179, and 183)
- [Trott et al., 2022] Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., and Wilke, J. (2022). Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817. (Cited on page 107)
- [Vajda, 2011] Vajda, A. (2011). *Multi-core and Many-core Processor Architectures*. Springer US, Boston, MA. (Cited on page 2)
- [van der Pas et al., 2017] van der Pas, R., Stotzer, E., and Terboven, C. (2017). *Using OpenMP - The Next Step: Affinity, Accelerators, Tasking, and SIMD*. MIT Press. (Cited on page 25)
- [Vázquez and Sánchez, 2021] Vázquez, J. L. and Sánchez, P. (2021). FOTV: A generic device offloading framework for openmp. In McIntosh-Smith, S., de Supinski, B. R., and Klinkenberg, J., editors, *OpenMP: Enabling Massive Node-Level Parallelism - 17th International Workshop on OpenMP, IWOMP 2021, Bristol, UK, September 14-16, 2021, Proceedings*, volume 12870 of *Lecture Notes in Computer Science*, pages 170–182. Springer. (Cited on page 116)
- [Wulf and McKee, 1995] Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24. (Cited on page 2)
- [Yoo et al., 2003] Yoo, A. B., Jette, M. A., and Grondona, M. (2003). SLURM: simple linux utility for resource management. In Feitelson, D. G., Rudolph, L., and Schwiegelshohn, U., editors, *Job Scheduling Strategies for Parallel Processing, 9th International Workshop*,
-

JSSPP 2003, Seattle, WA, USA, June 24, 2003, Revised Papers, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. Springer. (Cited on page 5)

[Zenker et al., 2016] Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W. E., and Bussmann, M. (2016). Alpaka - an abstraction library for parallel kernel acceleration. IEEE Computer Society. (Cited on pages 110 and 111)

Appendices

A. Contributors to Vecpar

Vecpar repository is open-source <https://github.com/wr-hamburg/vecpar>. The main contributors until the time of this thesis are detailed in Figure A.1.

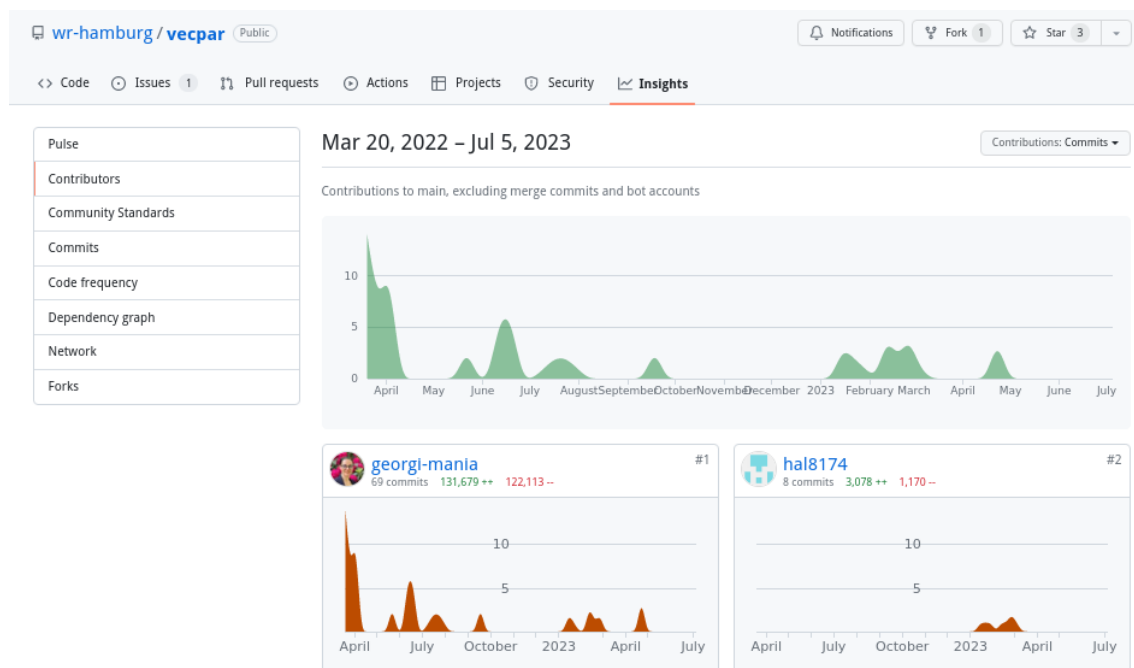


Figure A.1.: Vecpar's contributors

B. Code Samples

B.1. Clang-offload Generated Code Samples

When the OpenMP target backend is used, starting from the same file in Listing 3.1, both the main C++ file and the included header file are rewritten, as shown in Listing B.1 and Listing B.2. In the former, the parallel pragmas are added while the data is moved on device; in the latter, the functions invoked from a target region are marked accordingly so that an assembly version for both CPU and GPU is generated at compile time. Note that the functions are recursively included while the ones not needed in device code are not affected (e.g. function *h* in Listing B.2).

```

1  #include "include/marker.hpp"
2  #include "functions.hpp"
3
4  int main(int argc, char** argv) {
5  // int n = ...
6  int *h_data = (int *) malloc(sizeof(int) * n);
7  int *h_result = (int *) malloc(sizeof(int) * n);
8  // init arrays ...
9
10 auto lambda = [<typename T, typename R>(T* data, R* result, int n) {
11
12 T* d_data = &data[0];
13 R* d_result = &result[0];
14
15 #pragma omp target enter data map(to:d_data[0:n], d_result[0:n])
16 #pragma omp target teams distribute parallel for
17 for (int idx = 0 ; idx < n; idx++) {
18     functions::f(d_data[idx], d_result[idx]);
19 }
20 #pragma omp target exit data map(from:d_result[0:n])
21 };
22
23 lambda(h_data, h_result, n);

```

Listing B.1: Translated code for the main file using OpenMP target

```

1  namespace functions {
2      int h(int i) {
3          i = i - 5;
4          return i;
5      }

```

```

6
7  #pragma omp declare target
8  int g(int i) {
9      i = i + 100;
10     return i;
11 }
12 #pragma omp end declare target
13
14 #pragma omp declare target
15 void f(int& data_i, int& result_i) {
16     result_i = g(data_i) + 10;
17 }
18 #pragma omp end declare target
19 }

```

Listing B.2: Translated code for the functions namespace using OpenMP target

Similarly to the OpenMP target backend, when using the CUDA backend, both initial C++ files are modified as shown in Listing B.3 and Listing B.4. In the former, the memory is allocated and transferred to the device, a kernel is being called and the results are copied to the host; also, CUDA headers are being included. In the latter, the functions called by the CUDA kernel are marked with appropriate specifiers and the function pointer is copied to the device.

```

1  #include <cuda.h>
2  #include "include/mark-offload/marker.hpp"
3  #include "functions.hpp"
4
5  int main(int argc, char** argv) {
6      // int n = ...
7      int *h_data = (int *) malloc(sizeof(int) * n);
8      int *h_result = (int *) malloc(sizeof(int) * n);
9      // init arrays ...
10
11     auto lambda = []<typename T, typename R>(T* data, R* result, int n) {
12
13         T* d_data;
14         R* d_result;
15         api::func_t<T,R> h_f;
16
17         cudaMalloc(&d_data, n*sizeof(data[0]));
18         cudaMalloc(&d_result, n*sizeof(result[0]));
19
20         cudaMemcpy(d_data, data, n*sizeof(data[0]), cudaMemcpyHostToDevice);
21         cudaMemcpy(d_result, result, n*sizeof(result[0]), cudaMemcpyHostToDevice);
22
23         // copy function pointer to the device
24         cudaMemcpyFromSymbol(&h_f, functions::redirect<T,R>,
25             sizeof(api::func_t<T,R>));
26

```

```

27     api::kernel<<<1,n>>>(h_f, d_data, d_result, n);
28     // wait for device execution
29     cudaDeviceSynchronize();
30
31     cudaMemcpy(result, d_result, n*sizeof(result[0]), cudaMemcpyDeviceToHost);
32
33     cudaFree(d_data);
34     cudaFree(d_result);
35 };

```

Listing B.3: Translated code for the main file using CUDA

```

1 namespace functions {
2     int h(int i) {
3         i = i - 5;
4         return i;
5     }
6     __host__ __device__ int g(int i) {
7         return i;
8     }
9     __host__ __device__ void f(int& data_i, int& result_i) {
10         result_i = g(data_i) + 10;
11     }
12     template<typename TData, typename TResult>
13     __device__ api::func_t<TData&, TResult&> redirect = f;
14 }

```

Listing B.4: Translated code for the functions namespace using CUDA

B.2. Vecpar SAXPY/DAXPY Benchmark

The input/output for SAXPY/DAXPY are `vecmem::vector` container allocated using the memory resources exposed by the `vecmem` library (e.g. host memory and CUDA managed memory). We leave out the test setup and initialization part since this is identical for both the benchmark and `vecpar` use cases. Header files includes are also left out for space purposes; nevertheless the code can be checked online in the github repository.

The native OpenMP implementation is shown in Listing B.5.

```

1 template<class T>
2 void benchmark(vecmem::vector<T> &x, vecmem::vector<T> &y, T a) {
3     #pragma omp parallel for
4     for (size_t i = 0; i < y.size(); i++) {
5         y[i] = x[i] * a + y[i];
6     }
7 }

```

Listing B.5: OpenMP SAXPY/DAXPY kernel implementation

For the CUDA part, the global kernel is shown in Listing B.6, and it is called from a function which either explicitly handles the memory copies or assumes the data is accessible through CUDA managed memory mechanism; these are shown in Listing B.7 and Listing B.8 respectively. In these listings, a `vecpar::config` variable is set in order to make sure the same default¹ number of threads and blocks are used for all the tests. Also, the `CHECK_ERROR` macro is defined by the `vecpar` library inline with the CUDA documentation to check and log device errors.

```

1  template <class T>
2  __global__ void kernel(vecmem::data::vector_view<T> x_view,
3  vecmem::data::vector_view<T> y_view, T d_a) {
4      vecmem::device_vector<T> d_x(x_view);
5      vecmem::device_vector<T> d_y(y_view);
6      size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
7      if (idx >= d_x.size())
8          return;
9      d_y[idx] = d_x[idx] * d_a + d_y[idx];
10 }
```

Listing B.6: SAXPY/DAXPY CUDA kernel

```

1  vecmem::cuda::device_memory_resource d_mem;
2  vecmem::cuda::copy copy;
3  using vecmem::copy::type;
4
5  template <class T>
6  void benchmark(vecmem::vector<T> &x, vecmem::vector<T> &y, T a) {
7      // copy x vector to the device
8      auto x_buffer = copy.to(vecmem::get_data(x), d_mem, host_to_device);
9      auto x_view = vecmem::get_data(x_buffer);
10     // copy y vector to the device
11     auto y_buffer = copy.to(vecmem::get_data(y), d_mem, host_to_device);
12     auto y_view = vecmem::get_data(y_buffer);
13     // set the kernel configuration
14     vecpar::config c = vecpar::cuda::getDefaultConfig(x.size());
15     // invoke the CUDA kernel
16     kernel<T><<<<c.m_gridSize,c.m_blockSize, c.m_memorySize>>>>(x_view, y_view, a);
17     // synchronize and check errors
18     CHECK_ERROR(cudaGetLastError());
19     CHECK_ERROR(cudaDeviceSynchronize());
20     // copy result (y) back to host
21     copy(y_buffer, y, device_to_host);
22 }
```

Listing B.7: CUDA function which handles memory allocation and invokes the SAXPY/DAXPY kernel in Listing B.6

¹If vector's size is smaller than 256, then it is used as block size; otherwise, the block size is set to 256 and the number of blocks are computed to cover the problem size.

```

1  template <class T>
2  void benchmark(vecmem::vector<T> &x, vecmem::vector<T> &y, T a) {
3      // get the non-owning view of the data
4      auto x_view = vecmem::get_data(x);
5      auto y_view = vecmem::get_data(y);
6      // set the default configuration
7      vecpar::config c = vecpar::cuda::getDefaultConfig(x.size());
8      // invoke the CUDA kernel
9      kernel<<<c.m_gridSize, c.m_blockSize, c.m_memorySize>>>(x_view, y_view, a);
10     // synchronize and check potential errors
11     CHECK_ERROR(cudaGetLastError());
12     CHECK_ERROR(cudaDeviceSynchronize());
13 }

```

Listing B.8: CUDA function which uses vectors in managed memory and invokes the SAXPY/DAXPY kernel in Listing B.6

B.3. Detray RKN Algorithms

At the moment, we ported two algorithms to vecpar. These use either free or bound parameters to estimate the next track states and are listed in Listing B.9 and Listing B.10 respectively.

```

1  #ifndef DETRAY_RK_FREE_ALG_HPP
2  #define DETRAY_RK_FREE_ALG_HPP
3
4  #include "common.hpp"
5
6  namespace detray {
7
8      struct rk_stepper_free_algorithm
9      : public vecpar::algorithm::parallelizable_mmap<
10         vecpar::collection::One,
11         vecmem::vector<free_track_parameters>,
12         const vector3> {
13
14         TARGET free_track_parameters& mapping_function(
15             free_track_parameters& track,
16             const vector3& B) const {
17
18             free_track_parameters c_traj(track);
19
20             // Define RK stepper
21             rk_stepper_t rk_stepper(B);
22             crk_stepper_t crk_stepper(B);
23
24             // RK Stepping into forward direction
25             prop_state<rk_stepper_t::state, nav_state> propagation{

```

```

26     rk_stepper_t::state{track}, nav_state{};
27     prop_state<crk_stepper_t::state, nav_state> c_propagation{
28         crk_stepper_t::state{c_traj}, nav_state{};
29
30     crk_stepper_t::state& crk_state = c_propagation._stepping;
31
32     nav_state& n_state = propagation._navigation;
33     nav_state& cn_state = c_propagation._navigation;
34
35     crk_state.template set_constraint<step::constraint::e_user>(
36         0.5 * unit_constants::mm);
37     n_state._step_size = 1. * unit_constants::mm;
38     cn_state._step_size = 1. * unit_constants::mm;
39
40     for (unsigned int i_s = 0; i_s < rk_steps; i_s++) {
41         rk_stepper.step(propagation);
42         crk_stepper.step(c_propagation);
43         crk_stepper.step(c_propagation);
44     }
45
46     // Backward direction
47     // Roll the same track back to the origin
48     n_state._step_size *= -1. * unit_constants::mm;
49     cn_state._step_size *= -1. * unit_constants::mm;
50
51     for (unsigned int i_s = 0; i_s < rk_steps; i_s++) {
52         rk_stepper.step(propagation);
53         crk_stepper.step(c_propagation);
54         crk_stepper.step(c_propagation);
55     }
56
57     return track;
58 }
59 }; // end algorithm
60
61 } // namespace detray
62
63 #endif

```

Listing B.9: RKN stepper using free track parameters

```

1  #ifndef DETRAY_RK_BOUND_ALG_HPP
2  #define DETRAY_RK_BOUND_ALG_HPP
3
4  #include "common.hpp"
5
6  namespace detray {
7
8      struct rk_stepper_bound_algorithm
9      : public vecpar::algorithm::parallelizable_map<

```

```

10     vecpar::collection::One,
11     vecmem::vector<bound_track_parameters>,
12     vecmem::vector<bound_track_parameters>,
13     const vector3,
14     const transform3> {
15
16     TARGET bound_track_parameters& mapping_function(
17         bound_track_parameters& out_param,
18         const bound_track_parameters& in_param,
19         const vector3& B,
20         const transform3& trf) const {
21
22         mag_field_t mag_field(B);
23         prop_state<crk_stepper_t::state, nav_state> propagation{
24             crk_stepper_t::state(in_param, trf), nav_state{}};
25         crk_stepper_t::state& crk_state = propagation._stepping;
26         nav_state& n_state = propagation._navigation;
27
28         // Decrease tolerance down to 1e-8
29         crk_state.set_tolerance(rk_tolerance);
30
31         // RK stepper and its state
32         crk_stepper_t crk_stepper(mag_field);
33
34         // Path length per turn
35         scalar S = 2. * std::fabs(1. / in_param.qop()) / getter::norm(B) * M_PI;
36
37         // Run stepper for half turn
38         unsigned int max_steps = 1e4;
39
40         for (unsigned int i = 0; i < max_steps; i++) {
41             crk_state.set_constraint(S - crk_state.path_length());
42             n_state._step_size = S;
43             crk_stepper.step(propagation);
44             if (std::abs(S - crk_state.path_length()) < 1e-6) {
45                 break;
46             }
47         }
48
49         // Bound state after one turn propagation
50         out_param = crk_stepper.bound_state(propagation, trf);
51         return out_param;
52     }
53
54 }; // end algorithm
55
56 } // namespace detray
57 #endif

```

Listing B.10: RKN stepper using bound track paramters

C. Vecpar Backlog

Ongoing and future work address the known issues and optimizations listed below.

C.1. Known issues

1. Results are incorrect when a *parallel_reduce* algorithm whose identity element is not 0 is executed using the native CUDA backend. This is the motivation for adding the *identity_function* to the algorithm's definition. While the OpenMP and OpenMP Target backends are already using this approach, the CUDA backend still needs to be adapted.
2. The BabelStream benchmark shows an error of 10^{-4} for float operands when using the optimized prototype version of *parallel_map_reduce* in the OpenMP target backend, while double precision operations are within the accepted tolerance. While this is assumed to be due the non-associativity of the floating point operations, it needs further investigation before a similar implementation is generalized to the other backends.
3. There is an edge case when the vecmem library is prevented from releasing the GPU memory because vecpar references are not properly deleted. This can be reproduced only when using the CUDA backend, with large arrays stored in host-memory. To solve this but also to slightly improve the performance, we plan to use shared pointers as input and output for arrays in future *vecpar* versions.

C.2. Productivity and Performance Optimizations

1. Add context parameters for *parallelizable_filter* and *parallelizable_reduce*. This would allow easy porting of some other reconstruction steps like clustering.
 2. Add CUDA implementations for vecpar operators that could handle device pointers directly. At the moment, *vecpar* only supports input data stored in CPU memory or CUDA managed memory. By extending this to device memory as well, the algorithms can be easier and more efficiently composed in execution chains. This is correlated to the bugfix related to the usage of shared pointers.
 3. Extend the portability by adding a new SYCL-based backend using the SYCL support from the vecmem library.
-

4. Enhance chain functionality to support `vecmem::jagged_vectors` as well.
5. Support a user-specified number of iterable collections as input parameters to vecpar operations; currently this number is 5 for the OpenMP and CUDA backends, and 3 for the OpenMP target.
6. Hierarchical parallelism – Not only support this in an efficient way (using a pool of streams that communicate with the GPU(s)) but also provide an automatic detection of invalid usecases (e.g. no I/O operation should happen in a code that is supposed to be executed on a GPU).
7. Support massive parallelism by adding an MPI backend based on 3rd party libraries, like for example *ompcluster*¹.
8. Integrate clang-offload tool into the vecpar framework; this can trigger optimizations around the AST rooted in the vecpar function calls.
9. Use new heuristics and machine learning techniques to estimate an optimal configuration for a vecpar algorithm based on a given platform using 3rd party libraries like hwloc².
10. Extend the prototype approaches for *parallel_map* and *parallel_map_reduce* from the OpenMP target backend to other overloaded versions of them from the same and from the other backends.
11. Improve productivity by providing a Spack package for an easy installation and Python bindings.

¹<https://ompcluster.gitlab.io/>

²<https://www.open-mpi.org/projects/hwloc/>

D. Reproducibility Artifacts

The vecpar framework is open-source on github: <https://github.com/wr-hamburg/vecpar>. The only required dependency is the vecmem library. To compile all the vecpar backends (including the CUDA one), a version of LLVM/clang compiler with offloading support is required and CUDA libraries have to be installed; otherwise, gcc with offloading support can only build the OpenMP and OpenMP target backends. For testing on an AMD GPU, ROCm package has to be reachable.

The recommended setup can be created using spack package manager as shown in Listing D.1. Note that the latest version of CUDA will be automatically installed together with LLVM. For downloading a specific version (compatible to an existing driver), explicitly add this to the install command.

```
# install clang with offloading support
spack install llvm@14.0.0 +all_targets +cuda cuda_arch=<XY>
# install vecmem
spack install vecmem@0.22.0
# setup the support for the test infrastructure
spack install googletest
```

Listing D.1: Install dependencies and the compiler

For compiling on an AMD GPU, download and install the AOMP compiler as documented online¹.

To download, build and install vecpar follow the steps in Listing D.2. Disable the CUDA backend if the compilation happens either on a machine which does not have an NVIDIA GPU or the compiler is not clang. Disable the OpenMP Target (OMPT) backend if the compiler does not have offloading capabilities.

```
# download the sources
git clone --recurse-submodules https://github.com/wr-hamburg/vecpar.git
# compile
cmake -DCMAKE_BUILD_TYPE=Release -S vecpar -B vecpar-build
cmake --build vecpar-build \
  -DVECPAR_BUILD_OMP_BACKEND=On \
  -DVECPAR_BUILD_CUDA_BACKEND=On \
  -DVECPAR_BUILD_OMPT_BACKEND=On \
  -DVECPAR_BUILD_TESTS=On
# local installation default location is /usr/local/lib64 and /usr/local/
# include (sudo rights might be required)
```

¹<https://github.com/ROCm-Developer-Tools/aomp>

```
cmake --install vecpar-build
```

Listing D.2: Setup vecpar framework

D.1. Vecpar Internal Benchmark

To run the performance tests, invoke the shell script `saxpy_and_chain.sh` from the vecpar repository:

```
sh vecpar/test/scripts/saxpy_and_chain.sh $1 $2 $3 $4
```

with the following arguments: \$1 is the number of repetition for each test, \$2 is the number of OpenMP threads, \$3 is the fully qualified path to the location of `vecpar-build/test/` and \$4 is the fully-qualified path to the location where to store the CSV files generated by the tests. The script loads the specific versions of the libraries using `spack`, runs the test executables and then unloads the packages. The name of the result files use the format: `platform_kernel_memoryallocator.csv`, e.g. `gpu_saxpy_mmm.csv`; the results produced by the OpenMP target executable have `ompt` in the file name.

D.2. BabelStream Benchmark

The vecpar implementation is part of an open-source repository on github². To run the vecpar tests, see the following steps³:

1. Choose a vecpar setup by configuring the build

- VecparStream using the OMP backend on CPU

```
cmake -Bbuild/ -H. -DCMAKE_BUILD_TYPE=Release
      -DMODEL=vecpar
      -DOFFLOAD=OFF
      -DVECPAR_BACKEND=NATIVE
```

- VecparStream using the CUDA backend, arrays allocated in host memory, on NVIDIA GPU with `sm_XY` compute capability

```
cmake -Bbuild/ -H. -DCMAKE_BUILD_TYPE=Release
      -DMODEL=vecpar
      -DOFFLOAD=NVIDIA:sm_XY
      -DVECPAR_BACKEND=NATIVE
      -DMEM=DEFAULT
```

- VecparStream using the OMPT backend, arrays allocated in host memory, on NVIDIA GPU with `sm_XY` compute capability

²<https://github.com/wr-hamburg/BabelStream>, checkout the *vecpar* branch.

³The BabelStream authors recommend removing an existing build folder before changing the test setup for a new one.

```
cmake -Bbuild/ -H. -DCMAKE_BUILD_TYPE=Release
      -DMODEL=vecpar
      -DOFFLOAD=NVIDIA:sm_XY
      -DVECPAR_BACKEND=OMPT
```

- VecparStream using OMPT backend, arrays allocated in host memory, on AMD card with architecture gfx1031

```
cmake -Bbuild/ -H. -DCMAKE_BUILD_TYPE=Release
      -DMODEL=vecpar
      -DOFFLOAD=AMD:gfx1031
      -DVECPAR_BACKEND=OMPT
      -DCXX_EXTRA_FLAGS=-fopenmp-targets=amdgc-n-amd-amdhsa -Xopenmp-
target=amdgc-n-amd-amdhsa -march=gfx1031
```

2. Build the code

```
cmake --build build
```

3. Run different scenarios by setting the variables PRECISION and BENCHMARK control. If PRECISION is left unset, the tests will allocate double-precision arrays; to request single-precision, set it to --float. By default, all the benchmarks are being executed when BENCHMARK is not set; to run only the triad benchmark, set --triad-only.

```
./build/vecpar-stream $PRECISION $BENCHMARK --csv > vecpar.csv
```

For the comparison plots, Kokkos 3.7.1 was installed locally and was built by the BabelStream together with the associated tests by setting the flag `-DKOKKOS_IN_TREE`.

To check the performance optimization proposed for the *map_reduce* case, run the executable created by the automated tests: `vecpar-build/test/ompt/benchmark_ompt_gpu`. This will generate a CSV file with the same name which contains entries for running the tests with different vector sizes; the column "lib_total" states the time for running the code with the initial implementation which was `parallel_reduce(parallel_map(...))`, while the column "lib_grouped" states the time for the optimized version.

D.3. Track Reconstruction – RKN Stepper

To run the tests, a local installation of vecpar is mandatory. The code is available open-source on github⁴. Since our first evaluation tests, the code has been improved by the ACTS team. Nevertheless, for the simplified RKN stepper test, commenting out line 281 from the file `/core/include/detray/propagator/rk_stepper.ipp` which computes the transport jacobians. No additional code change has to be done for the realistic. The cmake file which configure the tests⁵ should be adapted to use the architecture for the

⁴<https://github.com/georgi-mania/detray>, *vecpar* branch, revision number `e3cebcbab47d02e4732336d4e88744dcafd3f7`

⁵`detray/tests/unit_tests/vecpar/CMakeLists.cmake`

GPU. Then the last step, is to configure the build to use a specific precision (float or double), in release mode. An example is shown below:

```
cmake -DCMAKE_BUILD_TYPE=Release \
      -DDETRAY_CUSTOM_SCALARTYPE=float \
      -S detray \
      -B detray-build
cmake --build detray-build/ -j 12
```

On an environment with CUDA support and an NVIDIA GPU, the compilation will produce the following executables:

- `detray_test_array_cuda` and `detray_test_eigen_cuda` – CUDA code with algebra backends `array`⁶ or `eigen` compiled with `nvcc`
- `detray_test_vecpar_array_cpu`, `detray_test_vecpar_array_gpu`, `detray_test_vecpar_eigen_cpu`, `detray_test_vecpar_eigen_gpu` – The same C++/vecpar code with algebra backends `array` or `eigen`, compiled for host or device (NVIDIA GPU).
- `detray_test_vecpar_array_ompt_gpu` – C++/vecpar implementation using vecpar OpenMP target compiled for NVIDIA/AMD GPU

For a system with an AMD GPU, only the executables using OpenMP and OpenMP target backends will be generated. Special compilation flags might be required for AMD GPU.

A couple of shell scripts⁷ were written to setup the test environment and to easily produce the csv files with the runtimes. Similar to the script in Section D.1, the input parameters are the number of test repetitions, the number of CPU OpenMP threads, the fully-qualified location of the build folder and the location to store the resulting files. Nevertheless, the executables can also be invoked manually like in the following example:

```
./detray_test_vecpar_array_gpu --gtest_repeat=$N \
  --gtest_filter=rk_stepper_vecpar.free_state_host_mr_time
```

⁶The naming *array* is interchangeable with *cmath* for the algebra backend developed by the ACTS team

⁷The scripts are available under `detray/tests/unit_tests/vecpar/scripts`

E. List of Publications

E.1. Published

Ai, X., Mania, G., Gray, H.M., Kuhn, M. and Styles, N. A GPU-Based Kalman Filter for Track Fitting. In *Computing and Software for Big Science Journal* Vol 5, No. 20 (2021). <https://doi.org/10.1007/s41781-021-00065-z>

Duwe, K., Luettgau, J., Mania, G., Squar, J., Fuchs, A., Kuhn, M., Betke, E. and Ludwig, T. State of the Art and Future Trends in Data Reduction for High-Performance Computing. In *Supercomputing Frontiers and Innovations Journal* Vol. 7, No. 1 (2020). <https://doi.org/10.14529/jsfi200101>

E.2. Accepted for Publication

Gessinger, P., Grasland, H., Gray, H., Joubé, S., Kusiak, K., Krasznahorkay, A., Leggett, C., Mania, G., Niermann, J., Salzburger, A., Styles, N., Swatman, S. N., and Yeo, B. ACTS GPU Track Reconstruction Demonstrator for HEP. In *7th International Connecting the Dots Workshop, Princeton, USA, Proceedings*.

Mania, G., Styles, N., Kuhn, M., Salzburger, A., Yeo, B. and Ludwig, T. Vecpar – A Framework for Portability and Parallelization. In *International Conference on Computational Science, Prague, Czech Republic, Proceedings*

List of Figures

1.1. HPC: CPU and GPU memory illustration [NVIDIA Corporation, a]	3
1.2. HPC: Heterogeneous Node at Frontier Supercomputer [Oak Ridge National Laboratory, 2022]	3
1.3. HPC: GPU memory bandwidth [Stephen Jones, 2017]	4
1.4. HPC: Software stack on HPC nodes	5
1.5. HPC: CPU and GPU models share [TOP500, 2022]	7
1.6. HEP: CERN collider infrastructure [Lopienska, 2022]	9
1.7. HEP: LHC delivered luminosity for 2011-2022 [ATLAS Collaboration, 2022c]	10
1.8. HEP: Particle detector types [LHC Education and Group, 2017]	11
1.9. HEP: The ATLAS detector [ATLAS Collaboration, 2022a]	12
1.10. HEP: Data flow in the ITk detector [Meng, 2021]	13
1.11. HEP: ATLAS coordinate system	14
1.12. HEP: Adapted Runge-Kutta-Nystrom numerical integrator [Lund et al., 2009]	15
1.13. HEP: Charged particle reconstruction flow	16
1.14. HEP: Seeding, track finding and fitting in ACTS [Salzburger, 2022]	17
1.15. HPC & HEP: Data aquisition in ATLAS	18
1.16. HPC & HEP: Computing Resources Estimates [ATLAS Collaboration, 2022b]	19
2.1. SYCL compilation model [Codeplay Software, 2022]	30
2.2. SYCL supported backends and compilers [Khronos Group, 2022]	31
2.3. NVIDIA SDK [NVIDIA Corporation, c]	32
2.4. CUDA execution model [NVIDIA Corporation, a]	32
2.5. NVIDIA Grace Hopper Architecture [Evans et al., 2022]	35
2.6. AMD GPU Architectures [AMD, 2022a]	36
2.7. AMD ROCm Platform [AMD, 2022b]	37
2.8. Intel oneAPI [Intel Corporation, 2022]	39
2.9. OpenMP as target independent runtime layer [Doerfert et al., 2023]	41
3.1. ACTS Core reconstruction toolkit [Salzburger, 2022]	46
3.2. ACTS R&D Heterogeneous Libraries [Gessinger et al., 2023]	47
3.3. TBB Kalman Fitter: Speedup diagram	50
3.4. TBB Kalman Fitter: Wall-clock time diagram	51
3.5. TBB Kalman Fitter: I/O vs computations	51
3.6. TBB Kalman Fitter: Memory consumption vs execution time	52

3.7. TBB Kalman Fitter: Fitting logs	53
3.8. GPU Kalman Fitter: Wall-clock time CPU vs. GPU [Ai et al., 2021]	55
3.9. GPU Kalman Fitter: Wall-clock time for NVIDIA GPUs [Ai et al., 2021]	55
3.10. GPU Kalman Fitter: Wall-clock time with inter- and intra-track paralleliza- tion [Ai et al., 2021]	56
3.11. GPU Kalman Fitter: Wall-clock time for different grid layouts [Ai et al., 2021]	56
3.12. GPU Kalman Fitter: Warp occupancy for GPU [Ai et al., 2021]	57
3.13. GPU Kalman Fitter: Wall-clock time when using one or two GPU simulta- neously [Ai et al., 2021]	57
3.14. Traccc: Transforming readouts to traccc EDM	58
3.15. Traccc: Proposed flow for transforming readouts to traccc EDM	59
3.16. Traccc: I/O vs computations	60
3.17. Traccc: Weak scaling analysis for proposed approach	60
3.18. Matrix inversion: Speedup diagram, different APIs, CPU [Koenneker, 2022]	63
3.19. Matrix inversion: Speedup diagram, different compilers [Koenneker, 2022]	63
3.20. Matrix inversion: Speedup diagram, different API, GPU [Koenneker, 2022]	64
3.21. Matrix inversion: Speedup diagram CPU vs GPU [Koenneker, 2022]	64
3.22. Matrix inversion: Error distribution [Koenneker, 2022]	65
3.23. Hardware platforms and their native programming languages	66
3.24. Clang-offload: Execution flow	68
3.25. Clang-offload: The modules	68
4.1. Vecpar: Algorithmic chain	76
4.2. Vecpar: High-level design	77
4.3. Vecpar: Execution flow	78
4.4. Vecpar: Components and decision flow	79
4.5. Vecpar: Parallelizable_map	82
4.6. Vecpar: Parallelizable_mmap	82
4.7. Vecpar: Parallelizable_filter	83
4.8. Vecpar: Parallelizable_reduce	84
4.9. Vecpar: Parallelizable_map_filter	85
4.10. Vecpar: Parallelizable_mmap_filter	85
4.11. Vecpar: Parallelizable_map_reduce	86
4.12. Vecpar: Parallelizable_mmap_reduce	87
4.13. Vecpar: CUDA backend structure	93
5.1. Performance portability review [Gayatri et al., 2018]	113
5.2. Performance portability review [Artigues et al., 2020]	113
5.3. Performance portability review (BabelStream triad) [Deakin et al., 2020]	114
5.4. Performance portability review (BabelStream dot) [Deakin et al., 2020]	115
5.5. Compiler pipeline in Futhark [Henriksen et al., 2017]	117

5.6. The Allen Framework [Aaij et al., 2020]	118
5.7. The Patatrack Framework [Bocci et al., 2020]	119
6.1. Evaluation: Vecpar saxpy/daxpy benchmark on CPU	125
6.2. Evaluation: Vecpar saxpy/daxpy benchmark on GPU	126
6.3. Evaluation: Portability on different GPUs	126
6.4. Evaluation: Vecpar chain internal benchmark	128
6.5. Evaluation: Vecpar default vs chain calls on GPU	129
6.6. Evaluation: BabelStream <i>triad</i> kernel single-precision	131
6.7. Evaluation: BabelStream <i>triad</i> kernel double-precision	132
6.8. Evaluation: Runtimes for the BabelStream <i>mul</i> kernel	132
6.9. Evaluation: Runtimes for the BabelStream <i>add</i> kernel	133
6.10. Evaluation: Runtimes for the BabelStream <i>copy</i> kernel	133
6.11. Evaluation: Runtimes for the BabelStream <i>dot</i> kernel	134
6.12. Evaluation: Runtimes for the BabelStream kernels on AMD	134
6.13. Evaluation: RKN wall-clock time on different CPU	137
6.14. Evaluation: RKN speedup diagram on CPU	138
6.15. Evaluation: RKN strong-scaling on CPU	138
6.16. Evaluation: RKN cmath on different NVIDIA GPU	139
6.17. Evaluation: RKN eigen on different NVIDIA GPU	139
6.18. Evaluation: RKN cmath single/double precision	139
6.19. Evaluation: RKN eigen single/double precision	140
6.20. Evaluation: RKN speedup on NVIDIA GPU	140
6.21. Evaluation: RKN vecpar speedup summary on CPU and NVIDIA GPU	141
6.22. Evaluation: RKN stepper with extreme load use case	141
6.23. Evaluation: RKN stepper on CPU	142
6.24. Evaluation: RKN stepper on NVIDIA GPU	143
6.25. Evaluation: RKN stepper speedup summary for CPU and GPU	143
6.26. Evaluation: RKN stepper with bound parameters on NVIDIA GPU	144
6.27. Evaluation: RKN stepper with free parameters on AMD GPU	144

List of Tables

1.1. HPC: First ten supercomputers [TOP500, 2022]	6
1.2. HPC: CPU/GPU Computing capabilities and power consumption	7
2.1. OpenMP and MPI comparison	27
2.2. Heterogeneous API support in main compilers	40
3.1. TBB Kalman Fitter: Environment setup	49
3.2. TBB Kalman Fitter: Observations	52
3.3. GPU Kalman Fitter: Environment setup	54
3.4. Matrix inversion: Environment setup	62
4.1. Vecpar: Algorithm types	81
6.1. Evaluation: Vecpar analysis on GPU with host vs unified memory	127
6.2. Evaluation: Vecpar mapping kernel analysis	128
6.3. Evaluation: Vecpar BabelStream Scenarios and Approaches	130

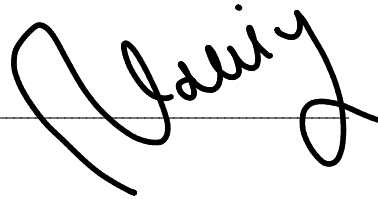
Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den 13. April 2023

Unterschrift:

A handwritten signature in black ink, appearing to read 'J. J. J.', written over a horizontal line.