TECHNISCHE UNIVERSITÄT DRESDEN

DEPARTMENT OF COMPUTER SCIENCE
INSTITUTE OF COMPUTER ENGINEERING
CHAIR OF COMPUTER ARCHITECTURE
PROF. DR. WOLFGANG E. NAGEL

Master's-Thesis

for the acquisition of the academic degree
Master of Science

Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures

Benjamin Worpitz
(Born October 2nd, 1990 in Chemnitz)

Professor: Prof. Dr. Wolfgang E. Nagel
Tutor: Dr. Michael Bussmann, Dr. Guido Juckeland,
Dr. Andreas Knüpfer, Dr. Bernd Trenkler

Dresden, October 7, 2015

Aufgabenstellung

Zielstellung dieser Arbeit ist die Definition einer abstrakten Schnittstelle zur task- und datenparallelen Abarbeitung von Prozessen auf Vielkernarchitekturen in C++. Hierbei soll der Einsatz Policy-basierter Abstraktionen es erlauben, die Abarbeitung der Prozesse auf einzelnen Hardwarearchitekturen optimal abzubilden, sodass die Performanceoptimierung eines Simulationscodes keine Änderung des Quellcodes der Simulation erfordert, sondern eine Optimierung der entsprechenden Policy-Implementation. Konkret soll dieser Mechanismus an dem Plasmacode *PIConGPU* implementiert werden, indem die entsprechenden aktuell in *CUDA* geschriebenen Optimierungen transparent auf die abstrakte Schnittstelle abgebildet und mit einer Implementation auf einer Mehrkern-/Vielkern x86 Hardware strukturell und leistungsmäßig verglichen werden. Dabei sollen optimierte Abbildungen mittels *OpenMP* / *Boost.Fiber* oder anderer Methoden implementiert und per Leistungsmessungen verglichen und analysiert werden.

Task

The objective is to define an abstract interface for task- and data-parallel execution of processes on multi-core architectures in C++. In doing so, the use of policy-based abstractions should allow to optimally reflect the execution of processes on individual hardware architectures, with the result that the performance optimization of a simulation code does not require a change to the source code of the simulation, but only an optimization of the respective policy implementation. More specifically, this is to be implemented in the plasma code *PIConGPU* by transparently mapping the respective optimizations, currently written in *CUDA*, to the abstract interface and comparing them with an implementation on a multi-core / many-core x86 hardware structurally and in terms of performance. Optimized mappings using *OpenMP* / *Boost.Fiber* or other methods are to be implemented and compared and analyzed by performance measurements.

Statement of authorship

I hereby certify that the Master's-Thesis I submitted today to the examination board of the faculty of computer science with the title:

Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures

has been composed solely by myself and that I did not use any sources and aids other than those stated, with quotations duly marked as such.

Dresden, October 7, 2015

Benjamin Worpitz

Kurzfassung

Die *alpaka* Programmbibliothek definiert und implementiert ein abstraktes hierarchisches Parallelitäts-Modell. Dieses Modell nutzt Parallelität und Speicherhierarchien auf allen verfügbaren Ebenen eines Knotens mit aktueller Hardware aus. Dadurch ist es möglich Portabilität leistungsfähiger Codes über verschiedenste Arten von Beschleunigern hinweg zu erreichen, indem bestimmte nicht unterstützte Stufen ausgelassen, und nur die auf einer spezifischen Hardware verfügbaren Ebenen genutzt werden. Alle Typen von Hardware (Mehrkern- und Vielkern-CPUs, GPUs sowie andere Beschleuniger) werden gleich behandelt und können in einheitlicher Form programmiert werden. Die bereitgestellte C++ Template-Schnittstelle ermöglicht die einfache Erweiterung der Bibliothek um andere Beschleuniger oder die Spezialisierung der Interna zum Zwecke der Optimierung.

Abstract

The *alpaka* library defines and implements an abstract hierarchical redundant parallelism model. This model exploits parallelism and memory hierarchies on a node at all levels available in current hardware. This allows to achieve portability of performant codes across various types of accelerators by ignoring specific unsupported levels and utilizing only the ones supported on a specific accelerator. All hardware types (multi- and many-core CPUs, GPUs and other accelerators) are treated and can be programmed in the same way. The C++ template interface provided allows for straightforward extension of the library to support other accelerators and specialization of its internals for optimization.

Contents

1 Introduction

1.1 Motivation

PIConGPU is a relativistic Particle-in-Cell (PIC) code for graphics processing units (GPUs). With its outstanding performance *PIConGPU* was one of the finalists of the 2013s Gordon Bell Prize. Without conducting real experiments it allows to investigate the dynamics of a plasma. It is an open-source [1][2] project dual-licensed under the GPLv3 and LGPLv3 developed and maintained by the *Junior Group Computational Radiation Physics* at the *Institute for Radiation Physics* of the HZDR.
The PIC algorithm is one of the central tools in computational plasma physics. Based on the Maxwell equations it describes the dynamics of a plasma by computing the motion of electrons and ions (fig. 1.1). Electric and magnetic fields act on (macro-)particles that can move freely between cells via the Lorentz force. The new momenta and positions of the particle resulting from the integration of the equation of motion then result in currents on the mesh, which in turn are required to compute the new electric and magnetic fields.

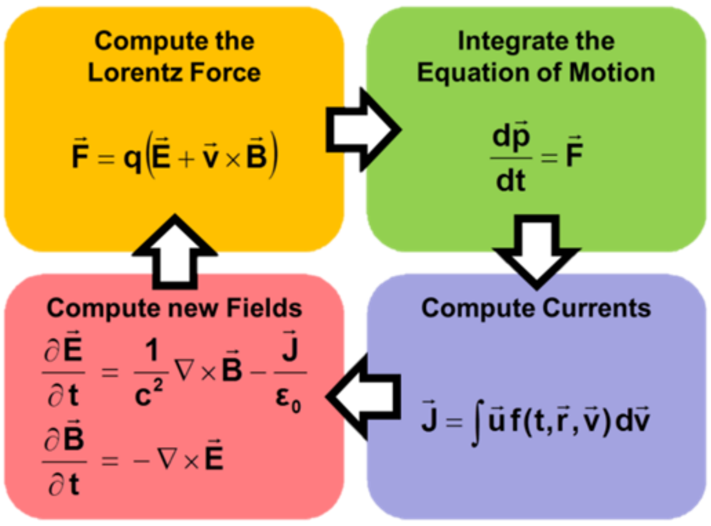
Compute the Lorentz Force
$\vec{F} = q(\vec{E} + \vec{v} \times \vec{B})$
Integrate the Equation of Motion
$\frac{d\vec{p}}{dt} = \vec{F}$
Compute new Fields
$\frac{\partial \vec{E}}{\partial t} = \frac{1}{c^2}\nabla \times \vec{B} - \frac{\vec{J}}{\varepsilon_0}$
$\frac{\partial \vec{B}}{\partial t} = -\nabla \times \vec{E}$
Compute Currents
$\vec{J} = \int \vec{u} f(t, \vec{r}, \vec{v}) d\vec{v}$

Figure 1.1: One time-step of the relativistic particle-in-cell algorithm.[5]

PIConGPU utilizes both, task and data parallelism. Task parallelism is exploited by running different kernels in parallel depending on the data already available and by overlapping computation and communication. Data parallelism requires the usage of special data structures. As can be seen in 1.2 the mesh of cells storing the electric and magnetic fields is further partitioned in so called super cells. All cells within a super cell are computed in parallel.
Because particles are moving freely between cells, it is not convenient to store them per cell. Some cells

[1] http://dx.doi.org/10.5281/zenodo.31121
[2] https://github.com/ComputationalRadiationPhysics/picongpu

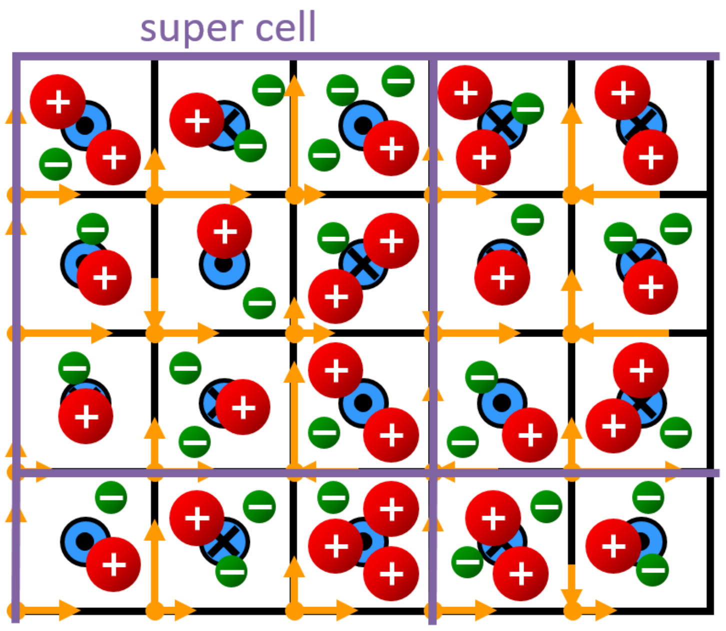

Figure 1.2: Schematic representation of a section of a 2D mesh of cells and particles for a PIC-simulation. Super cells combining multiple cells are marked with a purple edge[5]

could have no particles while others are crowded. This would lead to irregular computation times for individual cells and unpredictable execution times for a super cell. Furthermore, because particles are constantly migrating from one cell to another cell, they would have to be moved between the particle lists of the neighboring cells. By storing the particles per super cell, with each particle knowing its cell, these load imbalances are smoothed and the move overhead is reduced because the total number of moves between super cells is much smaller then the total number of moves between all individual cells. Figure 1.3 shows that each super cell has a corresponding list of frames containing the particles. All of the particles within a frame will be computed in parallel for all super cells. A frame stores the particle attributes as a structure of arrays so that, for example, all particle positions are aligned, close together and can be processed individually in a cache optimal manner. At the end of a time-step, particles that are marked invalid are filled with ones that have moved into this super cell or valid ones from the end of the frame list possibly removing empty or adding new frames. This resorting guarantees a continuously filled list of frames with minimal size that can be processed in parallel optimally.

This domain decomposition of the mesh of fields into super cells of cells and particles into a list of frames per super cell enables the code to scale across hundreds and thousands of GPUs. Even the roughly 18000 GPUs of the Titan supercomputer have been harnessed simultaneously to simulate the Kelvin-Helmholtz-Instability in a previously unmatched resolution [4]. However, what scales well on current hardware does not necessarily scale well on future architectures. The hardware landscape is always changing. In the past the big clusters have been CPU only. Today we see a change to accelerator supported computing. For example, GPUs, Intel Xeon Phis or other special purpose extension cards are extensively used. It is unpredictable what the next big step will be and how the Exaflop hardware will look like. It is not clear that GPUs will always be the best platform. Nevertheless, the underlying physical algorithms as well as the need for heterogeneous architectures will not change.

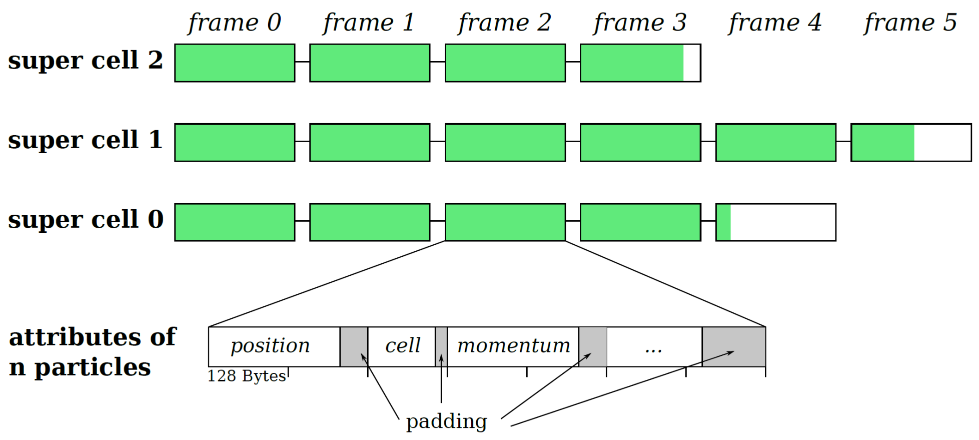

Figure 1.3: Particle frame list. Each super cell holds a list of frames with fixed width. Frames hold the attributes of multiple cells as a cache optimized structure of arrays. [8] p.36

Current highly parallel GPUs are optimized for throughput and hide latency and data dependencies by always keeping a ready pool of work. This allows to sustain the performance at a high percent of peak. CPUs in turn are designed to optimize the execution time of a single thread. Features like branch prediction, speculative execution, register renaming and many more *“[...] would cost far too much energy to be replicated for thousands of parallel GPU threads but [...] are entirely appropriate for CPUs.”* [3] Even more specialized architectures will appear and find their way into HPC. For example, after the acquisition of Altera by Intel and AMD filing patents [16] supporting it, we will be seeing FPGAs integrated into future processor generations.

“The essence of the heterogeneous computing model is that one size does not fit all. Parallel and serial segments of the workload execute on the best-suited processor delivering faster overall performance, greater efficiency, and lower energy and cost per unit of computation.” [3]

New hardware will not only allow to execute faster or calculate more. Systems like the pre-exascale *Summit* cluster with a fat node architecture combining *IBM POWER* CPUs with *NVIDIA* GPUs and a large coherent memory in between will furthermore enable the usage of new algorithms for a more precise simulation of additional algorithms such as models for atomic physics. For example, *PIConGPU* could be extended with an ionization kernel which would have to perform random searches for only a few values in a lookup table of up to hundreds of gigabytes. This would perfectly fit to those *IBM POWER* CPUs, while the rest of the simulation would still be running on the GPUs. Being able to express both of those parallel tasks in the same way would greatly enhance the productivity of the programmer and the clarity of the code.

Porting a complicated physics simulation code like *PIConGPU* from *CUDA* to x86 and possibly to other hardware architectures is a non-trivial task. A lot of developer time could be saved if this task would not have to be done repeatedly for every new hardware, but rather only once. Therefore, this work investigates the porting of highly scalable simulation codes on various multi-core architectures.

1.2 Problems in Porting Performant HPC Codes

Porting a highly performant code to a new architecture is a non-trivial task that poses many problems. Often it is a requirement to keep the simulation operative on the previous platform as well. This means that multiple hardware platforms have to be supported simultaneously. A great number of projects take the route that seems easiest at first and simply duplicate all the parallel algorithms and port them to the new back-end. All the specific API functions that have been used, have to be supplemented by the new pendants, possibly guarded by preprocessor macros to switch between the old and the new version. A switch of the back-end used in a simulation, for example, from *OpenMP* to *CUDA* often requires a near rewrite. For example, the *PIConGPU* simulation consists of 80000 lines of code, thereof 20000 lines of CUDA API and kernel code. Each newly supported platform would have to duplicate those nearly 20000 lines of code.
The following paragraphs will summarize problems that arise when performant HPC codes have to be ported:

Sustainability: Because the underlying HPC hardware is constantly changing, every new generation will require an adaption of the simulation. Even to deliver the performance reached on previous architectures is a tough task for programmers. Furthermore, nobody can guarantee the lifespan of the parallelization technique used. *OpenMP*, *CUDA*, *OpenACC* and all the other possibilities could be discontinued or get deprecated for any reason at any time. Therefore, an abstract interface is required that hides the particular back-end and allows to port the interface implementation and not the application using the interface itself.

Heterogeneity: Some parts of a simulation perfectly map to current GPUs while other parts are better computed on CPUs or other accelerators. Furthermore, by letting one part of the heterogeneous cluster hardware idle, a lot of computing power is wasted. It is essential, especially for future architectures, that those resources are utilized to reach the peak performance of the systems. This heterogeneous work division not only depends on the architecture but also on the number of available hardware resources, the workload and many other factors. Therefore, to reach good scaling across a multitude of systems, it is necessary to be able to dynamically decide where to execute which part of the simulation either at make-time, compile-time or at run-time. Currently this requires to duplicate the kernels and write specific implementations per back-end. Many projects only allow to switch the back-end of the whole simulation at once or possibly even per kernel at make-time. This will not be enough on future architectures where the ability to mix the back-ends is required to optimally utilize different cluster architectures or to dynamically load balance tasks across a diverse set of (possibly failing) accelerator devices. Therefore, an abstract interface unifying the abilities of all the back-ends is required to let the application express parallelism of the different back-ends in a unified algorithm that can then be mapped to the device currently in use.

Maintainability: Looking at the software engineering aspects, duplication is a bad solution because this leads to maintainability issues. In many projects such copies result in a large growth in the number of lines of code while only minimal new functionality is implemented. Most of the new code only executes things that have already been implemented for the initial platform. Developers having to change one of the algorithms additionally have to change all duplicates for all other back-ends. Depending on the similarity of the implementations, this can result in a doubling / multiplication of developer efforts in the worst-case scenario. Especially for open-source projects that rely on contributions from the community this raises the hurdle for new developers because they have to know not only one, but multiple different parallelization libraries. In the end good maintainability is what keeps a software project alive and what ensures a steady development progress. Therefore, an interface hiding the differences between all the back-ends is required to let the application express parallelism in a unified algorithm.

Testability: Code duplication, being the easiest way to port a simulation, exacerbates testing. Each new kernel has to be tested separately because different bugs could have been introduced into the distinct implementations. If the versions can be mixed, it is even harder because all combinations have to be tested. Often the tests (continuous integration tests, unit tests, etc.) have to run on a special testing hardware or on the production systems due to the reliance on the availability of special accelerators. For example, *CUDA* compile tests are possible without appropriate hardware but it is not feasible to execute even simple runtime tests due to the missing CPU emulation support. An interface allowing to switch between acceleration back-ends, which are tested for compatibility among each other, enables easy testing on development and test systems.

Optimizability: Even if the simulation code has encapsulated the APIs used, the optimal way to write performant algorithms often differs between distinct parallelization frameworks. It is necessary to allow the user to fine-tune the algorithm to run optimally on each different accelerator device by compile time specialization or policy based abstractions without the need to duplicate the kernel. Within the kernel there has to be knowledge about the underlying platform to adaptively use data structures that map optimally onto the current architecture. To ease this optimization work, libraries with data structures, communication patterns and other things hiding the differences between back-ends have to be implemented. This would allow to optimize the interface implementation and not the simulation itself.

In summary, it can be stated that all the portability problems of current HPC codes could be solved by introducing an abstract interface that hides the particular back-end implementations and unifies the way to access the parallelism available on modern many-core architectures.

1.3 Related Works

There are multiple other libraries targeting the (portable) parallel task execution within nodes. Some of them require language extensions, others pretend to achieve full performance portability across a multitude of devices. But none of these libraries can provide full control over the (possibly diverse) underlying hardware while being only minimal invasive. There is always a productivity-performance trade-off.
Furthermore, many of the libraries do not satisfy the requirement for full single-source C++ support. This is essential due to the *PIConGPU* simulation code relying heavily on template meta-programming for method specialization and compile time optimizations.

CUDA - Compute Unified Device Architecture *CUDA* [28] is a parallel computing platform and programming model developed by *NVIDIA* [3]. It is used in science and research as well as in consumer software to compute highly parallel workloads on GPUs starting from image and video editing up to simulations on high-performance computers. Such usage of graphics processing units not only for computer graphics, but also for tasks that have traditionally been handled by the CPU is called GPGPU (general-purpose computing on graphics processing units). A disadvantage of *CUDA* is that its application is bound to the usage of *NVIDIA* GPUs. Currently no other vendors provide accelerators that support *CUDA*. Additionally there is no supported free emulator allowing to execute *CUDA* code on CPUs.
The *CUDA* API is a higher level part of the programming model which allows to access and execute code on GPUs from multiple host languages including C++. The *CUDA* C/C++ language on the other hand is a mid level construct based on standard C++ with some extensions for accelerator programming and limitations in the supported constructs. For example, throwing and catching exceptions as well as runt-time type information (RTTI) are not supported. *CUDA* C/C++ is compiled to a low level virtual instruction set called PTX (Parallel Thread Execution). The PTX code is later compiled to assembler code by the GPU driver.
NVIDIA provides an extended C++ compiler based on the LLVM clang [4] compiler called nvcc that allows to mix host C++ code using the *CUDA* API with *CUDA* C/C++. The host part of the C++ code is compiled by the respective host system compiler (gcc, icc, clang, MSVC) while the GPU device code is separately compiled to PTX. After the compilation steps both binaries are linked together to form the final assembly.
CUDA defines a heterogeneous programming model where tasks are offloaded from the host CPU to the device GPU. Functions that should be offloaded to the GPU are called kernels. As can be seen in figure 1.4 a grid of such kernels is executed in parallel by multiple threads organized in blocks. Threads within a block can synchronize, while blocks are executed independently and possibly in sequential order depending on the underlying hardware.
The global device memory is the slowest but largest memory accessible by all threads. It can be accessed from host code via methods provided by the *CUDA* API. Global memory is persistent across kernel invocations. Threads within a block can communicate through a fast but small shared memory. Each thread has a set of very low latency registers similar to CPU threads. Additionally there are special purpose memory sections for constant and texture data.
The *CUDA* C/C++ language gives full control over memory, caches and the execution of kernels.

PGI CUDA-X86 [5] is a compiler technology that allows to generate x86-64 binary code from *CUDA* C/C++ applications using the *CUDA Runtime API* but does not support the *CUDA Driver API*. At runtime *CUDA* C programs compiled for x86 execute each *CUDA* thread block using a single host core, eliminating synchronization where possible. Multiple kernel threads are combined to be executed together via the CPUs SIMD (Single Instruction Multiple Data) capabilities for vectorized execution. The *PGI Unified Binary technology* allows to create a single binary that uses *NVIDIA* GPUs when available, or runs on multi-core CPUs else. The compiler is not always up-to-date with the latest *CUDA* versions and is not available for free. Furthermore, the compiler seems not to be developed actively since *NVIDIA*

[3] `http://www.nvidia.com/`
[4] `http://clang.llvm.org/`
[5] `https://www.pgroup.com/resources/cuda-x86.htm`

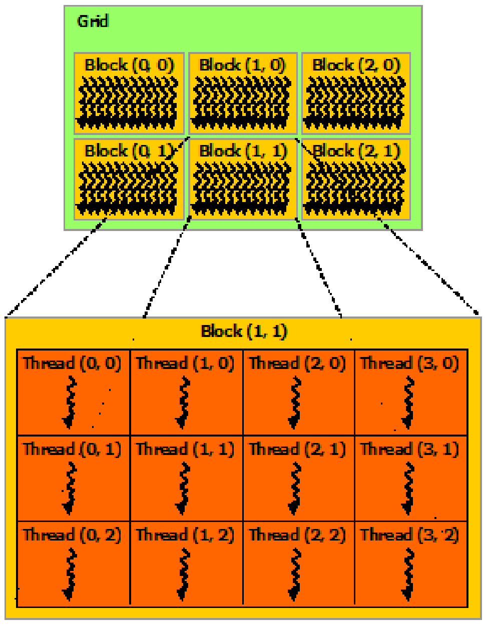

Figure 1.4: The grid of thread blocks defined by the *CUDA* programming model [22]. The execution order of blocks is undefined but possibly parallel while no synchronization between block is possible. In contrast, threads within a thread are allowed to synchronize and are executed in parallel.

acquired *PGI* in 2013. Since 2012 no news were published and nothing could be found in the yearly release notes of the *PGI* compiler suite.

GPU Ocelot [6] is an open-source dynamic JIT compilation framework. It allows to execute native *CUDA* binaries by dynamically translating the *NVIDIA PTX* virtual instruction set architecture to other instruction sets. It supports *NVIDIA* and *AMD* GPUs as well as multicore CPUs via a PTX to LLVM (Low Level Virtual Machine) translator. The project is not in active development anymore. It only supports PTX up to version 3.1 (current version is 4.2).

OpenMP [7] is an open specification for vendor agnostic shared memory parallelization. By adding annotations (pragmas in C/C++) to loops or regions, it allows to easily parallelize existing sequential C/C++/Fortran code in an incremental manner. Due to the nature of pragmas, these hints are ignored if the compiler does not support them or thinks they are inappropriate. This allows those programs to be compiled as sequential or parallel versions by only changing a compiler flag. In C/C++ the syntax for *OpenMP* directives is `#pragma omp` followed by multiple clauses. For example, with the directive `#pragma omp parallel for`, the compiler will automatically distribute the iterations of the directly following loop across the available cores. *OpenMP* 4.0 introduced support for offloading computations to accelerator devices, substantially improved the task support and extended the SIMD capabilities. By embedding code within a `#pragma omp target` block, the contained code will be executed on the selected device. *OpenMP* 4.0 is missing the ability for unstructured data movement and only implements

[6]http://gpuocelot.gatech.edu/
[7]http://openmp.org//

structured data movement from and to devices. The compiler directive `#pragma omp target data map(...)...` at the begin of a code block will define which data is copied to, copied back from and is created on the device. At the end of the code block the memory is copied back or gets deleted. There is no way to allocate device memory that is persistent between kernel calls in different methods because it is not possible to create a device data region spanning both functions in the general case. *OpenMP* 4.1, expected for the end of 2015, is likely to introduce `#pragma omp target enter data`, `#pragma omp target exit data` and other unstructured data movement directives that allow to pass and obtain pointers of already resident memory to and from offloaded kernels. Currently *OpenMP* does not provide a way to control the hierarchical memory because its main assumption is a shared memory for all threads. Therefore, the block shared memory on *CUDA* devices can not be explicitly utilized.

OpenACC [8] is a pragma based programming standard for heterogeneous computing. It is very similar to *OpenMP* and provides annotations for parallel execution and data movement as well as run-time functions for accelerator and device management. In contrast to *OpenMP* it allows limited access to *CUDA* block shared memory. Current compiler implementations support *NVIDA*, *AMD* and *Intel* accelerators. Only as of *OpenACC* 2.0 explicit memory management and tiling is supported. *OpenACC* does not support dynamic allocation of memory (`new`, `delete`) in kernel code. It is aimed to be fully merged with *OpenMP* at some point, but for now *OpenMP* 4.0 only introduced some parts of it.

OpenCL [9] is a programming framework for heterogeneous platforms. It is fully hardware independent and can utilize CPUs and GPUs of nearly all vendors. This is achieved by compiling the *OpenCL* kernel code (or the standardized *SPIR* intermediate representation) at run-time by the platform driver into the native instruction set. Versions prior to 2.1 (released in March 2015) did only support a C-like kernel language. Version 2.1 introduced a subset of C++14. *OpenCL* does not support single-source programming (combining C++ host code and accelerator code in a single file). This is a precondition for templated kernels which are required for policy based generic programming. It is necessary to note that *NVIDIA* seems to neglect their *OpenCL* implementation. Support for version 1.2 has just been added in April 2015 after only three and a half years after the publication of the standard. *OpenCL* does not support dynamic allocation of memory (`new`, `delete`) in kernel code.

SYCL [10] is a cross-platform abstraction layer based on *OpenCL*. The main advantage over *OpenCL* itself is that it allows to write single-source heterogeneous programs. It enables the usage of a single C++ template function for host and device code. As of now there is no usable free compiler implementation available that has good support for multiple accelerator devices.

C++ AMP (Accelerated Massive Parallelism) [11] is an open specification from *Microsoft* currently implemented on top of *DirectX 11*. It is an language extension requiring compiler support that allows to annotate C++ code that can then be run on multiple accelerators. *C++ AMP* requires the usage of the `array` data structure or the `array_view` wrapper responsible for copying data to and from the accelerator

[8] `http://www.openacc-standard.org/`
[9] `https://www.khronos.org/opencl/`
[10] `https://www.khronos.org/sycl/`
[11] `https://msdn.microsoft.com/en-us/library/hh265136.aspx`

devices. The `parallel_for_each` function is responsible for offloading the provided function object whose `operator()` has to be annotated with `restrict(amp)`. The threads can access shared memory and synchronize. The range of supported accelerator devices, plaforms and compilers is currently very limited.

KOKKOS [12] provides an abstract interface for portable, performant shared memory-programming. It is a C++ library that offers `parallel_for`, `parallel_reduce` and similar functions for describing the pattern of the parallel tasks. The execution policy determines how the threads are executed. For example, this influences the sizes of blocks of threads or if static or dynamic scheduling should be used. The library abstracts the kernel as a function object that can not have any user defined parameters for its `operator()`. Inconveniently, arguments have to be stored in members of the function object coupling algorithm and data together. *KOKKOS* provides both, abstractions for parallel execution of code and data management. Multidimensional arrays with a neutral indexing and an architecture dependent layout are available, which can be used, for example, to abstract the underlying hardwares preferred memory access scheme that could be row-major, column-major or even blocked.

Thrust [13] is a parallel algorithms library resembling the C++ Standard Template Library (STL). It allows to select either the *CUDA*, *TBB* or *OpenMP* back-end at make-time. Because it is based on generic `host_vector` and `device_vector` container objects, it is tightly coupling the data structure and the parallelization strategy. There exist many similar libraries such as *ArrayFire* [14] (*CUDA*, *OpenCL*, native C++), *VexCL* [15] (*OpenCL*, *CUDA*), *ViennaCL* [16] (*OpenCL*, *CUDA*, *OpenMP*) and *hemi* [17] (*CUDA*, native C++).

1.4 Distinction of the *alpaka* Library

In section 1.2 we saw that all the portability problems of current HPC codes could be solved with an abstract interface unifying the underlying accelerator back-ends. Section 1.3 showed that there is currently no project available that could solve all of the problems highlighted. Therefore the problem description given at the begin of the work can not be solved satisfactorily by directly porting *PIConGPU* to any other API but requires the definition of such an abstraction. As long as the implementation of this simple abstraction requires less then the 20000 lines of code, which would have to be changed within *PIConGPU*, this even leads to less work then the direct port itself. A proof-of-concept library showing the feasibility of this approach has been implemented in this work. This C++ interface library is called *alpaka* - *Abstraction Library for Parallel Kernel Acceleration*. The subsequent enumeration will summarize the purpose of the library:

***alpaka* is ...**

[12] https://github.com/kokkos
[13] https://thrust.github.io/
[14] http://www.arrayfire.com/
[15] https://github.com/ddemidov/vexcl/
[16] http://viennacl.sourceforge.net/
[17] https://github.com/harrism/hemi/

- an **abstract interface** describing parallel execution on multiple hierarchy levels. It allows to implement a mapping to various hardware architectures but **is no optimal mapping itself**.
- sustainably solving portability (50% on the way to reach full performance portability)
- solving the **heterogeneity** problem. An identical algorithm / kernel can be executed on heterogeneous parallel systems by selecting the target device.
- reducing the **maintainability** burden by not requiring to duplicate all the parts of the simulation that are directly facing the parallelization framework. Instead, it allows to provide a single version of the algorithm / kernel that can be used by all back-ends. All the accelerator depending implementation details are hidden within the *alpaka* library.
- simplifying the **testability** by enabling **easy back-end switching**. No special hardware is required for testing the kernels. Even if the simulation itself will always use the *CUDA* back-end, the tests can completely run on a CPU. As long as the *alpaka* library is thoroughly tested for compatibility between the acceleration back-ends, the user simulation code is guaranteed to generate identical results (ignoring rounding errors / non-determinism) and is portable without any changes.
- **optimizable**. Everything in *alpaka* can be replaced by user code to optimize for special use-cases.
- **extensible**. Every concept described by the *alpaka* abstraction can be implemented by users. Therefore it is possible to non-intrusively define new devices, streams, buffer types or even whole accelerator back-ends.
- **data structure agnostic**. The user can use and define arbitrary data structures.

alpaka is not ...

- an automatically **optimal mapping** of algorithms / kernels to various acceleration platforms. Except in trivial examples an optimal execution always depends on suitable selected data structure. An adaptive selection of data structures is a separate topic that has to be implemented in a distinct library.
- automatically **optimizing concurrent data accesses**.
- **handling** or hiding differences in arithmetic operations. For example, due to **different rounding** or different implementations of floating point operations, results can differ slightly between accelerators.
- **guaranteeing any determinism** of results. Due to the freedom of the library to reorder or repartition the threads within the tasks it is not possible or even desired to preserve deterministic results. For example, the non-associativity of floating point operations give non-deterministic results within and across accelerators.

The *alpaka* library is aimed at parallelization within nodes of a cluster. It does not compete with libraries for distribution of processes across nodes and communication among those. For these purposes libraries like MPI (Message Passing Interface) or others should be used. MPI is situated one layer higher and can be combined with *alpaka* to facilitate the hardware of a whole heterogeneous cluster. The *alpaka* library can be used for parallelization within nodes, MPI for parallelization across nodes.

1.5 Comparison

Table 1.1 summarizes which of the problems mentioned in section 1.2 can be solved by current intra-node parallelization frameworks and the proof-of-concept *alpaka* abstraction library.

Framework / API	Open-Source	Free	Single-Source C++	Port-ability	Hetero-genity	Maintain-ability	Test-ability	Optimiz-ability	Data structure agnostic
CUDA	✗	✓	✓	✗	✗	✗	✗	✓	✓
PGI CUDA-x86	✗	✗	✓	✓	❍	✓	✓	✗	✓
GPU Ocelot	✓	✓	✓	✓	❍	✓	✓	✗	✓
OpenMP	✓	✓	✓	✓	✓	✓	✓	✗	✓
OpenACC	✓	✓	✓	✓	✓	✓	✓	✗	✓
OpenCL	✓	✓	✗	✓	✓	✓	✓	✗	✓
SYCL	✓	(✓)	✓	✓	✓	✓	✓	(✓)	✓
C++AMP	✓	✓	✓	(✓)	✓	✓	✓	✗	✓
KOKKOS	✓	✓	✓	✓	❍	✓	✓	✗	❍
Thrust	✓	✓	✓	✓	❍	✓	✓	✗	✗
alpaka	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 1.1: Properties of intra-node parallelization frameworks and their ability to solve the problems in porting performant HPC codes. ✓: yes / fully solved, ❍: partially solved, ✗: no / not solved

1.6 Structure of this Thesis

In section 2 an abstraction layer will be defined which allows to utilize all levels of parallelism across various architectures. Within this chapter the structure of current hardware and their peculiarities will be explained and the influence on the final abstraction will be outlined.
During the course of this thesis a reference library called *alpaka* will be developed that implements the abstraction layer. The library structure and the implementation will be described in chapter 3.
To evaluate the design of the abstraction and to estimate the overhead introduced by the *alpaka* library, chapter 4 discusses measurements with artificial and real world codes. Native versions of DAXPY and DGEMM algorithms will be compared with implementations using the corresponding *alpaka* back-ends. Furthermore, an *alpaka* port of the *HASEonGPU* simulation on multiple CPU and GPU platforms will be compared to its native *CUDA* version.
Finally, chapter 5 summarizes the achievement and discusses future developments.

2 Abstraction

Parallelism and memory hierarchies at all levels need to be exploited in order to achieve performance portability across various types of accelerators. Within this chapter an abstraction will be derivated that tries to provide a maximum of parallelism while simultaneously considering implementability and applicability in hardware.
Looking at the current HPC hardware landscape, we often see nodes with multiple sockets/processors extended by accelerators like GPUs or Intel Xeon Phi, each with their own processing units. Within a CPU or a Intel Xeon Phi there are cores with hyper-threads, vector units and a large caching infrastructure. Within a GPU there are many small cores and only few caches. Each entity in the hierarchy has access to different memories. For example, each socket / processor manages its RAM, while the cores additionally have non-explicit access to L3, L2 and L1 caches. On a GPU there are global, constant, shared and other memory types which all can be accessed explicitly. The interface has to abstract from these differences without sacrificing speed on any platform.
A process running on a multi-socket node is the largest entity within *alpaka*. The abstraction is only about the task and data parallel execution on the process/node level and down. It does not provide any primitives for inter-node communication. However, such libraries can be combined with *alpaka*.
An application process always has a main thread and is by definition running on the host. It can access the host memory and various accelerator devices. Such accelerators can be GPUs, Intel Xeon Phis, the host itself or other devices. Thus, the host not necessarily has to be different from the accelerator device used for the computations. For instance, an Intel Xeon Phi simultaneously can be the host and the accelerator device.
The *alpaka* library can be used to offload the parallel execution of task and data parallel work simultaneously onto different accelerator devices.

2.1 Task Parallelism

One of the basic building blocks of modern applications is task parallelism. For example, the operating system scheduler, deciding which thread of which process gets how many processing time on which CPU core, enables task parallelism of applications. It controls the execution of different tasks on different processing units. Such task parallelism can be, for instance, the output of the progress in parallel to a download. This can be implemented via two threads executing two different tasks.
The valid dependencies between tasks within an application can be defined as a DAG (directed acyclic graph) in all cases. The tasks are represented by nodes and the dependencies by edges. In this model, a task is ready to be executed if the number of incoming edges is zero. After a task finished it‘s work, it is removed from the graph as well as all of it‘s outgoing edges,. This reduces the number of incoming edges of subsequent tasks.
The problem with this model is the inherent overhead and the missing hardware and API support. When

it is directly implemented as a graph, at least all depending tasks have to be updated and checked if they are ready to be executed after a task finished. Depending on the size of the graph and the number of edges this can be a huge overhead.

OpenCL allows to define a task graph in a somewhat different way. Tasks can be enqueued into an out-of-order command queue combined with events that have to be finished before the newly enqueued task can be started. Tasks in the command queue with unmet dependencies are skipped and subsequent ones are executed. The `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` property of a command queue is an optional feature only supported by few vendors. Therefore, it can not be assumed to be available on all systems.

CUDA on the other hand does currently (version 7.5) not support such out-of-order queues in any way. The user has to define dependencies explicitly through the order the tasks are enqueued into the queues (called streams in *CUDA*). Within a stream, tasks are always executed in sequential order, while multiple streams are executed in parallel. Streams can wait for events enqueued into other streams.

In both APIs, *OpenCL* and *CUDA*, a task graph can be emulated by creating one stream per task and enqueuing a unique event after each task, which can be used to wait for the preceding task. However, this is not feasible due to the large stream and event creation costs as well as other overheads within this process.

Therefore, to be compatible with a wide range of APIs, the interface for task parallelism has to be constrained. Instead of a general DAG, multiple queues of sequentially executed tasks will be used to describe task parallelism. Events that can be enqueued into the queues enhance the basic task parallelism by enabling synchronization between different streams, devices or the host threads.

2.2 Data Parallelism

In contrast to task parallelism, data parallelism describes the execution of one and the same task on multiple, often related data elements. For example, an image color space conversion is a textbook example of a data parallel task. The same operation is executed independently on each pixel. Other data parallel algorithms additionally introduce dependencies between threads in the input-, intermediate-, or output-data. For example, the calculation of a brightness histogram has no input-data dependencies. However, all pixel brightness values finally have to be merged into a single result. Even these two simple examples show that it is necessary to think about the interaction of parallel entities to minimize the influence of data dependencies.

Furthermore, it is necessary to respect the principles of spatial and temporal locality. Current hardware is built around these locality principles to reduce latency by using hierarchical memory as a trade-off between speed and hardware size. Multiple levels of caches, from small and very fast ones to very large and slower ones exploit temporal locality by keeping recently referenced data as close to the actual processing units as possible. Spatial locality in the main memory is also important for caches because they are usually divided into multiple lines that can only be exchanged one cache line at a time. If one data element is loaded and cached, it is highly likely that nearby elements are also cached. If the pixels of an image are stored row wise but are read out column wise, the spatial locality assumption of many CPUs is violated and the performance suffers. GPUs on the other hand do not have a large caching hierarchy but allow explicit access to a fast memory shared across multiple cores. Therefore, the best way to process individual data elements of a data parallel task is dependent on the data structure as well as the underlying hardware.

The main part of the *alpaka* abstraction is the way it abstracts data parallelism and allows the algorithm writer to take into account the hierarchy of processing units, their data parallel features and corresponding memory regions. The abstraction developed is influenced and based on the groundbreaking *CUDA* and *OpenCL* abstractions of a multidimensional grid of threads with additional hierarchy levels in between. Another level of parallelism is added to those abstractions to unify the data parallel capabilities of modern hardware architectures. The explicit access to all hierarchy levels enables the user to write code that runs performant on all current platforms. However, the abstraction does not try to automatically optimize memory accesses or data structures but gives the user full freedom to use data structures matching the underlying hardware preferences.

2.2.1 Thread

Theoretically, a basic data parallel task can be executed optimally by executing one thread per independent data element. In this context, the term thread does not correspond to a native kernel-thread, an *OpenMP* thread, a *CUDA* thread, a user-level thread or any other such threading variant. It only represents the execution of a sequence of commands forming the desired algorithm on a per data element level. This ideal one-to-one mapping of data elements to threads leads to the execution of a multidimensional grid of threads corresponding to the data structure of the underlying problem. The uniform function executed by each of the threads is called a kernel. Some algorithms such as reductions require the possibility to synchronize or communicate between threads to calculate a correct result in a time optimal manner. Therefore our basic abstraction requires a n-dimensional grid of synchronizable threads each executing the same kernel. Figure 2.1 shows an hypothetical hardware that could optimally execute this data parallel task. The threads are mapped one-to-one to the cores of the processor. For a time optimal execution, the cores have to have an all-to-all equal length connection for communication and synchronization.

The only difference between the threads is their positional index into the grid which allows each thread to compute a different part of the solution. Threads can always access their private registers and the global memory.

Registers All variables with default scope within a kernel are automatically saved in registers and are not shared automatically. This memory is local to each thread and can not be accessed by other threads.

Global Memory The global memory can be accessed from every thread in the grid as well as from the host thread. This is typically the largest but also the slowest memory available.

Individual threads within the grid are allowed to statically or dynamically allocate buffers in the global memory.

Prior to the execution of a task, the host thread copies the input buffers and allocates the output buffers onto the accelerator device. Pointers to these buffers then can be given as arguments to the task invocation. By using the index of each thread within the grid, the offset into the global input and output buffers can be calculated. After the computation has finished, the output buffer can be used either as input to a subsequent task or can be copied back to the host.

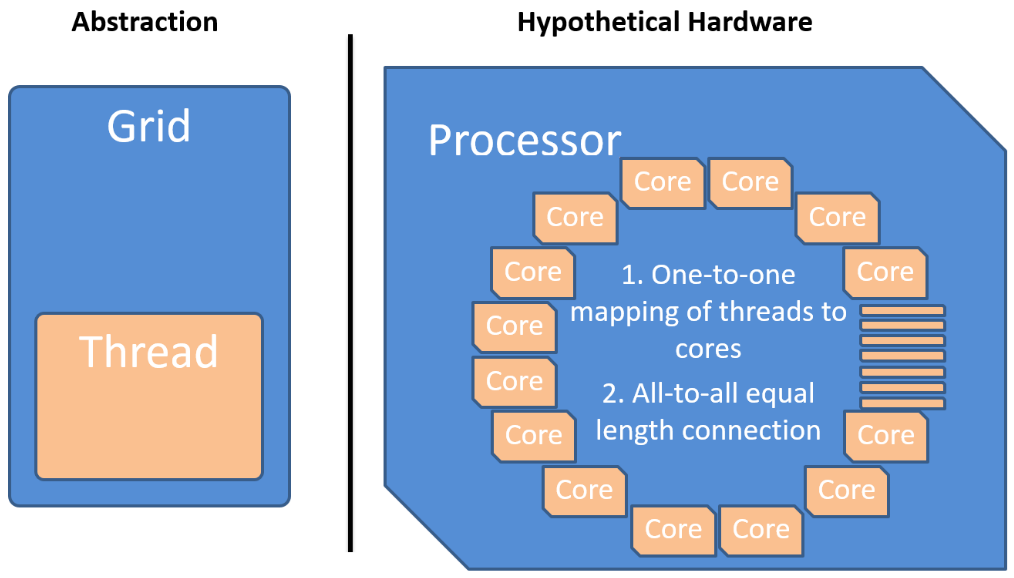
Abstraction
Hypothetical Hardware
Grid
Thread
Processor
Core
1. One-to-one mapping of threads to cores
2. All-to-all equal length connection

Figure 2.1: On the left: The abstraction hierarchy with a grid of threads. On the right: A hypothetical hardware that could execute data parallel tasks with the basic grid-thread abstraction optimally by providing a one-to-one mapping of per data element threads to cores and an equal length connection between all cores.

2.2.2 Block

Building a processor with possibly thousands of cores where all cores have an equal length connection for fast communication and synchronization is not viable. Either the processor size would have to grow exponentially with the number of cores or the all-to-all communication speed would decrease so much that computations on the processor would be impractical. Therefore, the communication and synchronization of threads has to be limited to sizes manageable by real hardware.
Figure 2.2 depicts the solution of introducing a new hierarchy level in the abstraction. A hypothetical processor is allowed to provide synchronization and fast communication within blocks of threads but is not required to provide synchronization across blocks. The whole grid is subdivided into equal sized blocks with a fast but small shared memory. Current accelerator abstractions (*CUDA* and *OpenCL*) only support equal sized blocks. This restriction could possibly be lifted to support future accelerators with heterogeneous block sizes.
There is another reason why independent blocks are necessary. Threads that can communicate and synchronize require either a one-to-one mapping of threads to cores, which is impossible because the number of data elements is theoretically unlimited, or at least a space to store the state of each thread. Even old single core CPUs were able to execute many communicating and synchronizing threads by using cooperative or preemptive multitasking. Therefore, one might think that a single core would be enough to execute all the data parallel threads. But the problem is that even storing the set of registers and local data of all the possible millions of threads of a task grid is not always viable. The blocking scheme solves this by enabling fast interaction of threads on a local scale but additionally removes the necessity to store the state of all threads in the grid at once because only threads within a block must be executed in parallel. Within a block of cores there still has to be enough memory to store all registers of all contained threads. The independence of blocks allows applications to scale well across diverse

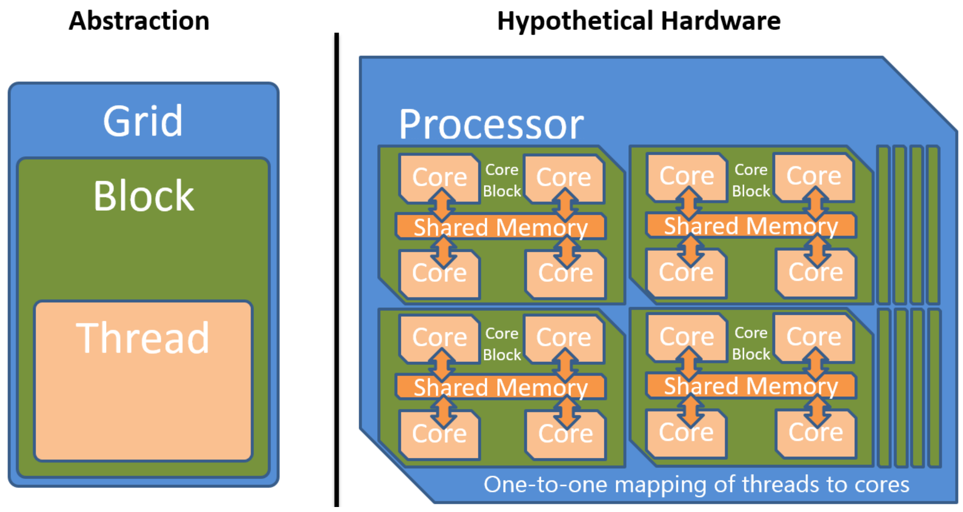

Figure 2.2: On the left: The abstraction hierarchy with a grid of blocks of threads. On the right: A hypothetical hardware that could execute data parallel tasks by providing a one-to-one mapping of per data element threads to cores and fast synchronization and communication between threads within a block.

devices. As can be seen in figure 2.3, the accelerator can assign blocks of the task grid to blocks of cores in arbitrary order depending on availability and workload.

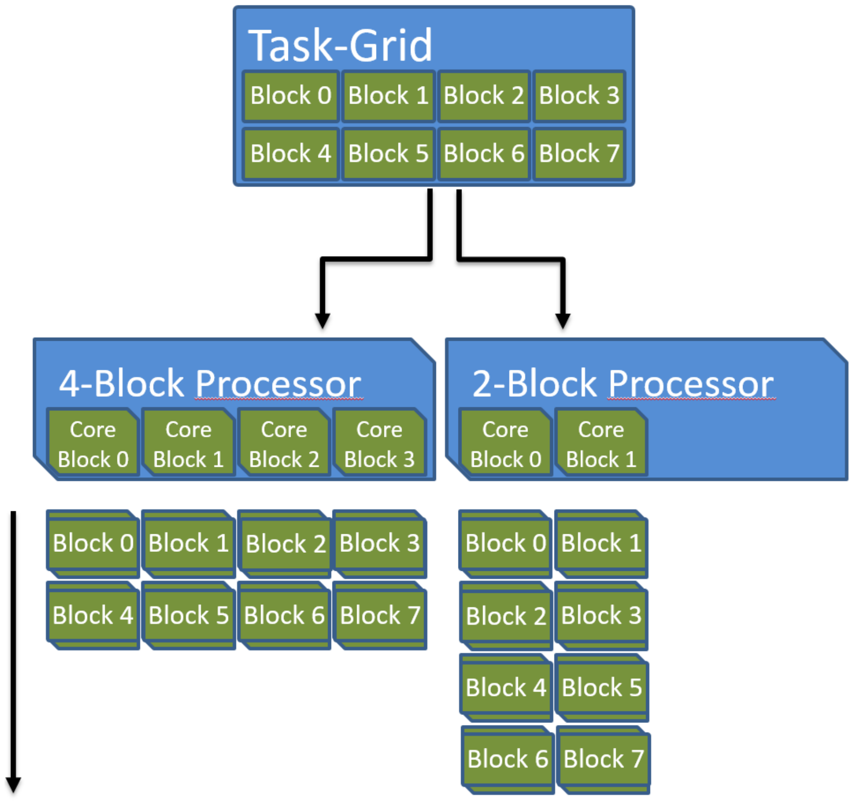
Task-Grid
Block 0 Block 1 Block 2 Block 3
Block 4 Block 5 Block 6 Block 7
4-Block Processor
Core Block 0 Core Block 1 Core Block 2 Core Block 3
2-Block Processor
Core Block 0 Core Block 1
Block 0 Block 1 Block 2 Block 3
Block 4 Block 5 Block 6 Block 7
Block 0 Block 1
Block 2 Block 3
Block 4 Block 5
Block 6 Block 7

Figure 2.3: Mapping of a grid of threads subdivided into independent blocks onto processors with varying number of blocks.

Shared Memory Each block has its own shared memory. This memory can only be accessed explicitly by threads within the same block and gets discarded after the complete block finished its calculation. This memory is typically very fast but also very small. No variables are shared between kernels by default.

2.2.3 Warp

With the current abstraction only independent parallelism via blocks and synchronizable parallelism via threads can be expressed. However, there are more variants of parallelism in real hardware. Because all threads in the grid are executing the same kernel and even the same instruction at the same time when ignoring divergent control flows, a lot of chip space can be saved. Multiple threads can be executed in perfect synchronicity, which is also called lock-step. A group of such threads executing the same instruction at the same time is called a warp (fig 2.4). All threads within a warp share a single instruction pointer (IP), and all cores executing the threads share one instruction fetch (IF) and instruction decode (ID) unit.

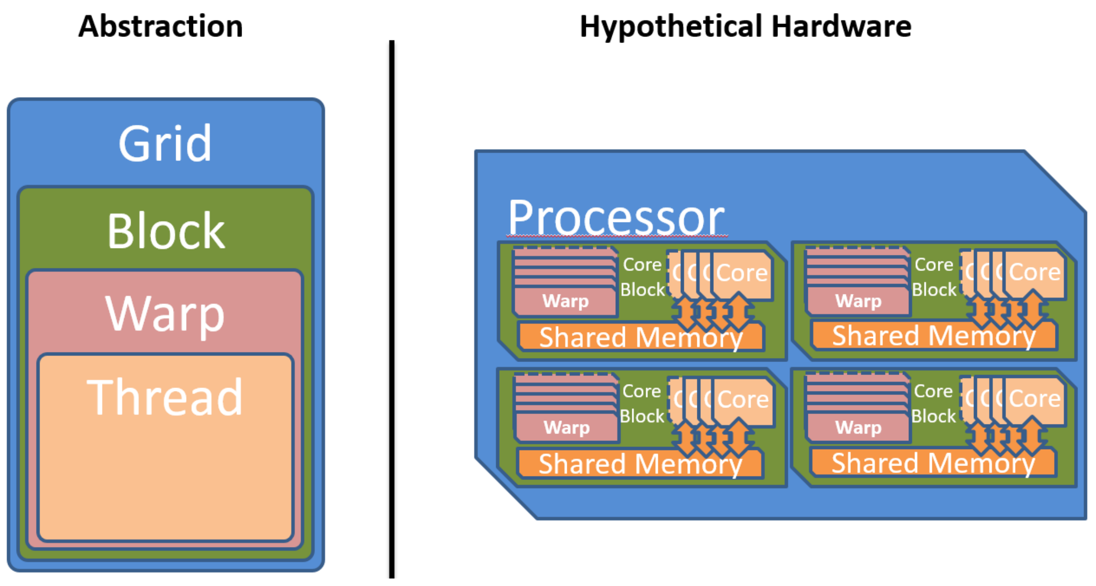

Figure 2.4: On the left: The abstraction hierarchy with a grid of blocks of warps of threads. On the right: A hypothetical hardware that could execute data parallel tasks by providing a one-to-one mapping of threads to data elements, combining multiple threads into independent blocks but saving chip space by executing multiple threads in lock-step on simplified cores.

Even threads with divergent control flows can be executed within one warp. *CUDA*, for example, solves this by supporting predicated execution and warp voting. For long conditional branches the compiler inserts code which checks if all threads in the warp take the same branch. For small branches, where this is too expensive, all threads always execute all branches. Control flow statements result in a predicate and only in those threads where it is true, the predicated instructions will have an effect.

Not only *CUDA* GPUs support the execution of multiple threads in a warp. Full blown vector processors with good compilers are capable of combining multiple loop iterations containing complex control flow statements in a similar manner as *CUDA*.

Due to the synchronictiy of threads within a warp, memory operations will always occur at the same

time in all threads. This allows to coalesce memory accesses. Different *CUDA* devices support different levels of memory coalescing. Older ones only supported combining multiple memory accesses if they were aligned and sequential in the order of thread indices. Newer ones support unaligned scattered accesses as long as they target the same 128 byte segment.
The ability of very fast context switches between warps and a queue of ready warps allows *CUDA* capable GPUs to hide the latency of global memory operations.

2.2.4 Element

To use the maximum available computing power of, for example, a modern x86 processor, the computation has to utilize the SIMD vector registers. Many current architectures support issuing a single instruction that can be applied to multiple data elements in parallel.
The original x86 instruction set architecture did not support SIMD instructions but has been enhanced with MMX (64 bit width registers), SSE (128 bit width registers), AVX (256 bit width registers) and AVX-512 (512 bit width registers) extensions. In varying degree, they allow to process multiple 32 bit and 64 bit floating point numbers as well as 8, 16, 32 and 64 bit signed and unsigned integers.
CUDA capable GPUs do not have vector registers where multiple values of type `float` or `double` can be manipulated by one instruction. Nevertheless, newer *CUDA* capable devices implement basic SIMD instructions on pairs of 16 bit values and quads of 8-bit values. They are described in the documentation of the PTX instruction set architecture chapter 8.7.13 [24] but are only of any use in very special problem domains, for example for deep learning.
It would be optimal if the compiler could automatically vectorize our kernels when they are called in a loop and vectorization is supported by the underlying accelerator. However, besides full blown vector processors, mainstream CPUs do not support predicated execution or similar complex things within vector registers. At most, there is support for masking operations which allow to emulate at least some conditional branching. Therefore, this missing hardware capability has to be circumvented by the compiler. There are scientific research projects such as the work done by Ralf Karrenberg et al [17] [18] [19] building on the *LLVM* compiler infrastructure supporting such whole-function vectorization. However, as will be shown in section 4.2.1, current mainstream compilers do not support automatic vectorization of basic, non trivial loops containing control flow statements (`if`, `else`, `for`, etc.) or other non-trivial memory operations. Therefore, it has to be made easier for the compiler to recognize the vectorization possibilities by making it more explicit.
The opposite of automatic whole function vectorization is the fully explicit vectorization of expressions via compiler intrinsics directly resulting in the desired assembly instruction. A big problem when trying to utilize fully explicit vectorization is, that there is no common foundation supported by all explicit vectorization methods. A wrapper unifying the x86 SIMD intrinsics found in the `intrin.h` or `x86intrin.h` headers with those supported on other platforms, for example ARM NEON (`arm_neon.h`), PowerPC Altivec (`altivec.h`) or *CUDA* is not available and to write one is a huge task in itself. However, if this would become available in the future, it could easily be integrated into *alpaka* kernels.
Due to current compilers being unable to vectorize whole functions and the explicit vectorization intrinsics not being portable, one has to rely on the vectorization capabilities of current compilers for primitive loops only consisting of a few computations. By creating a grid of data elements, where multiple elements are processed per thread and threads are pooled in independent blocks, as it is shown in figure

2.5, the user is free to loop sequentially over the elements or to use vectorization for selected expressions within the kernel. Even the sequential processing of multiple elements per thread can be useful depending on the architecture. For example, the *NVIDIA cuBLAS* general matrix-matrix multiplication (GEMM) internally executes only one thread for each second matrix data element to better utilize the registers available per thread.

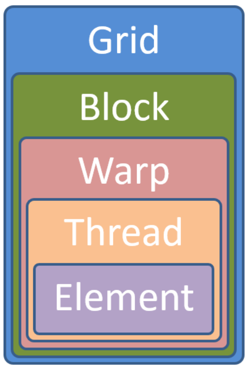

Figure 2.5: The abstraction hierarchy consisting of a grid of elements, where multiple elements are processed per thread, multiple threads are executed in lock-step within a warp and multiple warps form independent blocks.

2.2.5 Summary

In the further course of the work the abstraction will be called *Redundant Hierarchical Parallelism*. This term is inspired by the paper *The Future of Accelerator Programming: Abstraction, Performance or Can We Have Both?* [32]. It investigates a similar “concept of copious parallel programming” reaching 80%-90% of the native performance while comparing CPU and GPU centric versions of an *OpenCL* n-body simulation with a general version utilizing parallelism on multiple hierarchy levels.
The *CUDA* or *OpenCL* abstractions themselves are very similar to the one designed in the previous sections and consists of all but the Element level. However, as has been shown, all five abstraction hierarchy levels are necessary to fully utilize current architectures. By emulating unsupported or ignoring redundant levels of parallelism, algorithms written with this abstraction can always be mapped optimally to all supported accelerators. The table 2.1 summarizes the characteristics of the proposed hierarchy levels.

Hierarchy Level	Parallelism	Synchronizable
grid	sequential / parallel	✗/ ✓
block	parallel	✗
warp	parallel	✓
thread	parallel / lock-step	✓
element	sequential	✗

Table 2.1: Abstraction hierarchy levels, their different types of parallelism and their ability to synchronize.

Depending on the stream a task is enqueued into, grids will either run in sequential order within the same stream or in parallel in different streams. They can be synchronized by using events. Blocks can not be synchronized and therefore can use the whole spectrum of parallelism ranging from fully parallel up to fully sequential execution depending on the device. Warps combine the execution of multiple threads in lock-step and can be synchronized implicitly by synchronizing the threads they contain. Threads within a block are executed in parallel warps and each thread computes a number of data elements sequentially.

3 Implementation

This chapter describes how the abstraction defined in chapter 2 can be mapped to real devices (sec. 3.1). Furthermore, the implementation of the library in C++, especially the way C++11 allows to define the abstract concepts and to take advantage of the zero-overhead compile-time polymorphism is explained in section 3.2.

3.1 Mapping *Redundant Hierarchical Parallelism* onto Specific Hardware Architectures

By providing an accelerator independent interface for kernels, their execution and memory accesses at different hierarchy levels, *alpaka* allows the user to write accelerator independent code that does not neglect performance.

The mapping of the decomposition to the execution environment is handled by the *alpaka* library. A computation that is described with a maximum of parallelism can not be mapped one to one to any existing hardware. GPUs do not have vector registers for `float` or `double` types. Therefore, the element level is often omitted on *CUDA* accelerators. CPUs in turn are not (currently) capable of running thousands of threads concurrently and do not have equivalently fast inter-thread synchronization and shared memory access as GPUs do.

A major point of the *redundant hierarchical parallelism* abstraction is to ignore specific unsupported levels and utilize only the ones supported on a specific accelerator. This allows a mapping to various current and future accelerators in a variety of ways enabling optimal usage of the underlying compute and memory capabilities.

In the following sections the grid level is always mapped to the whole device being in consideration. The scheduler can always execute multiple kernel grids from multiple streams in parallel by statically or dynamically subdividing the available resources. However, this will only ever simplify the mapping due to less available processing units. Furthermore, being restricted to less resources automatically improves the locality of data due to spatial and temporal locality properties of the caching hierarchy.

3.1.1 CUDA GPUs

Mapping the abstraction to GPUs supporting *CUDA* is straightforward because the hierarchy levels are identical up to the element level. So blocks of warps of threads will be mapped directly to their *CUDA* equivalent.

The element level will be supported through an additional run-time variable containing the extent of elements per thread. This variable can be accessed by all threads and should optimally be placed in constant device memory for fast access. Additionally, it could be fully optimized away by template specialization if exactly one element is calculated per thread. This would allow the compiler to completely remove the sequential element loops due to the compile-time constant of exactly one element per thread.

3.1.2 x86 CPUs

There are multiple possible ways to map the *alpaka* abstraction to x86 CPUs. Figure 3.1 shows a node with two sockets and two cores each. Through symmetric multithreading (Hyper-Threading) each core represents two processing units.

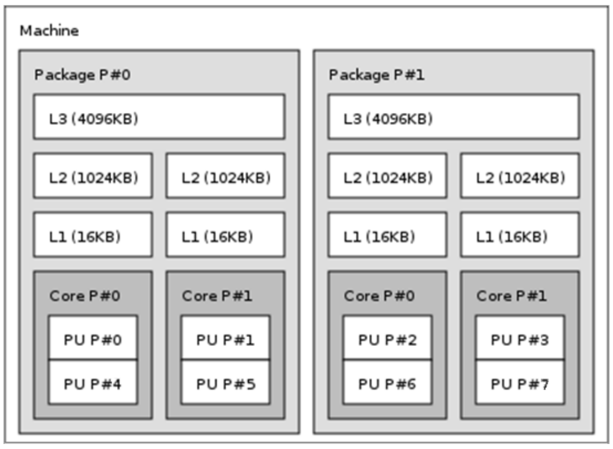

Figure 3.1: Compute and memory hierarchy of a dual-socket (Package) node with dual-core CPUs and symmetric multithreading (Hyper-Threading).

3.1.2.1 Thread

Mapping the thread level directly to the processing units is the most trivial part of the assignment of hierarchy levels to hardware units. However, the block and warp levels could be mapped to hardware components in different ways with varying advantages and disadvantages.

3.1.2.2 Warp

Even though a warp seems to be identical to a vector register, because both execute a single uniform instruction on multiple data elements, they are not the same. As has been described in section 2.2.3, warps can handle branches with divergent control flows of multiple threads. There is no equivalent hardware unit in a CPU supporting this. Therefore, the warp level can not be utilized on CPUs leading to a one-to-one mapping of threads to warps which does not violate the rules of the abstraction.

3.1.2.3 Block

One Block Per Node By combining all threads/PUs of all processors on the node into one block, the number of synchronizing and communicating threads is enlarged. This high possible thread count would simplify the implementation of some types of algorithms but introduces performance issues on multi-core nodes. The shared memory between all cores on a node is the RAM. However, the RAM and the communication between the sockets is far too slow for fine-grained communication in the style of *CUDA* threads.

One Block Per Socket If each processor on each socket would concurrently execute one block, the L3 cache would be used as the fast shared memory. Although this is much better then to use the RAM, there is still a problem. Regions of the global memory and especially from the shared memory that are accessed are automatically cached in the L1 and / or L2 caches of each core. Not only the elements which are directly accessed will be cached but always the whole cache line they lie in. Cache lines typically have a size of 64 Bytes on modern x86 architectures. This leads to, for example, eight double precision floating point numbers being cached at once even though only one value really is required. As long as these values are only read there is no problem. However, if one thread writes to a value that is also cached on other cores, all such cache copies have to be invalidated. This results in a lot of cache and bus traffic. Due to the hierarchical decomposition of the grid of threads reflecting the data elements, neighboring threads are always combined into a common block. By mapping a block to a socket, threads that are executed concurrently always have very close indices into the grid. Therefore, the elements that are read and written by the threads are always very close together within the memory and will most probably share a cache line. This property is exploited on *CUDA* GPUs, where memory accesses within a warp are combined into one large transaction. However, when multiple threads from multiple CPU cores write to different elements within a cache line, this advantage is reversed into its opposite. This pattern which is shown in figure 3.2 non-intuitively leads to heavy performance degradation and is called false-sharing.

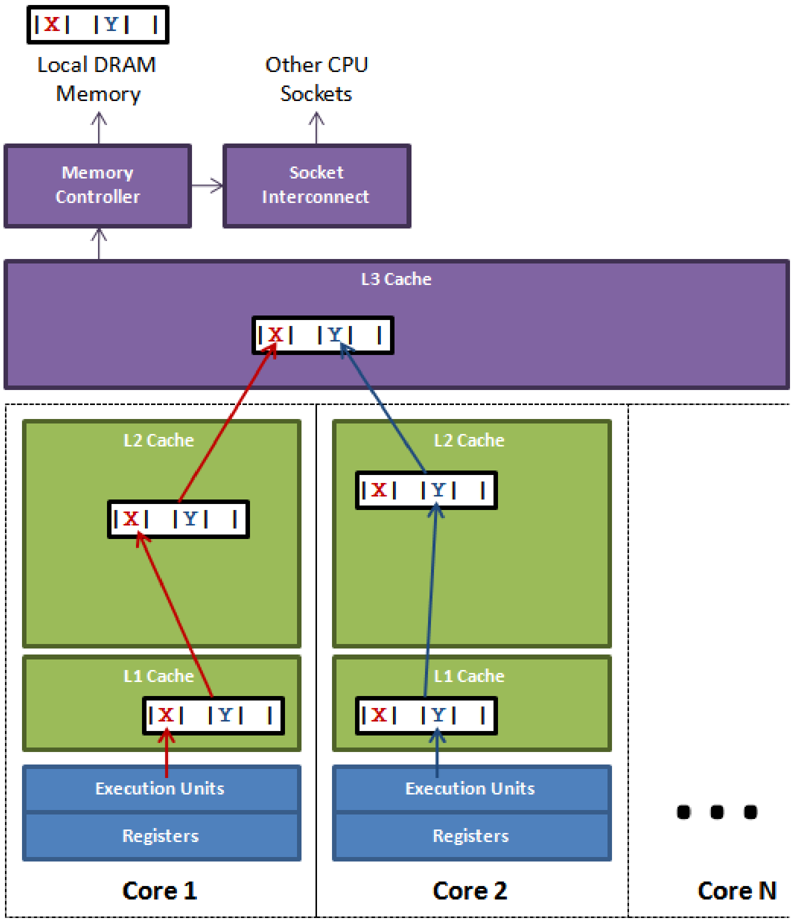

Figure 3.2: Multiple threads on different cores accessing distinct values in the same cache-line lead to performance degradation due to false-sharing. [20]

One Block Per Core The best compromise between a high number of threads per block and a fast communication between the threads is to map a block directly to a CPU core. Each processing unit (possibly a Hyper-Thread) executes one or more threads of our hierarchical abstraction while executing multiple elements locally either by processing them sequentially or in a vectorized fashion. This is illustrated in figure 3.3.

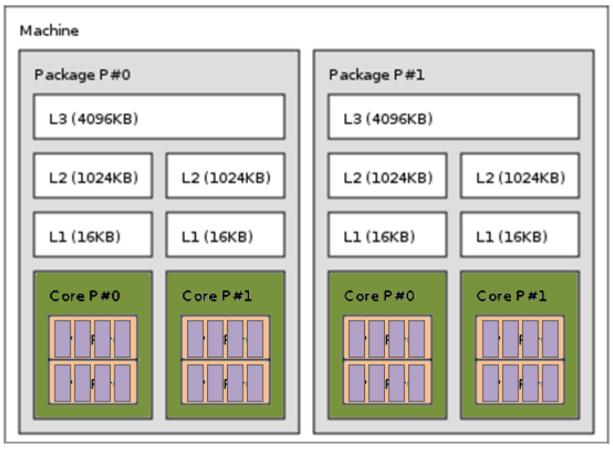

Figure 3.3: Possible mapping of blocks, threads and elements to the compute and memory hierarchy of a dual-socket node with dual-core CPUs and symmetric multithreading. Blocks are mapped to cores, threads to processing units and multiple elements are executed per thread.

One Block Per Thread If there is no symmetric multithreading or if it is desired, it is also possible to implement a mapping of one block with exactly one thread for each processing unit. This allows to completely remove the synchronization overhead for tasks where this is not required at all.

3.1.2.4 Threading Mechanisms

The mapping of threads to processing units is independent of the threading mechanism that is used. As long as the thread affinity to cores can be set correctly, *OpenMP*, *pthread*, *std::thread* or other libraries and APIs can be used interchangeably to implement various *alpaka* back-ends. They all have different advantages and disadvantages. Real operating system threads like *pthread*, *std::thread* and others have a high cost of thread creation and thread change because their default stack size amounts to multiple megabytes. *OpenMP* threads on the other hand are by default much more lightweight. However, they are arbitrarily limited by the runtime implementation in the maximum number of concurrent threads a machine supports. All of the previous methods have non-deterministic thread changes in common. Therefore it is not possible to decide the order in which threads within a block are processed, which could be a good optimization opportunity.
To allow blocks to contain more threads then the number of processing units each core provides, it is possible to simply start more threads then processing units are available. This is called oversubscription. Those threads can be bound to the correct cores and by relying on the operating system thread scheduler, they are preemptively multitasked while sharing a single cache and thereby avoiding false-sharing. However, this is not always beneficial because the cost of thread changes by the kernel-mode scheduler should not be underestimated.

Fibers To remove the overhead of the kernel mode scheduler as well as to enable the usage of deterministic thread context-switches, fibers can be used. A fiber is a user-space thread with cooperative context-switches and extends the concept of coroutines. A coroutine is basically a function that can be suspended and resumed but does not necessarily have a stack. In contrast, functions within most programming languages represent subroutines and not coroutines because they can neither be suspended in the mid of execution nor resumed exactly at the place they were suspended without losing values on the functions local stack.

Multiple fibers can be executed within one operating system thread, which allows to simulate multiple threads per block without kernel-mode multithreading. This was not possible without fibers because only coroutines allow the kernel functions to be suspended at synchronization points and resumed when all fibers reached it. Each time an operating system thread executing a function would wait for an other thread or a resource, an equivalent fiber just switches to the next fiber within the executing host thread. Due to the context changes happening at user-level, the cost is much lower. Additionally, fiber context changes are deterministic and it is even possible to implement an user-level scheduler. An advantage of a user level scheduler over the operating system thread scheduler is the possibility to optimally utilize the caches by taking into account the memory access pattern of the algorithm. Furthermore, fibers reduce the number of locks and busy waits within a block because only one fiber is active per operating system thread at a time.

There are multiple C++ Standards Committee Papers (N3858, N3985, N4134) discussing the inclusion of fibers, awaitable functions and similar concepts into C++.

3.1.2.5 *Intel* Xeon Phi

With it‘s 60 and more cores and 4 Hyper-Threads per core it is much larger then an ordinary CPU but conceptually they are very similar. As can be seen in figure 3.4, which is showing a section of the compute and memory hierarchy, the only difference is the amount of cores and Hyper-Threads. What can not be seen in the figure is the size of the vector registers, which are 512 bits wide and thereby larger then those of most conventional CPUs.

Because of the similarity, the same mapping that has been used for CPUs can also be used for the many-core *Intel* Xeon Phi. A possible mapping of blocks to cores with the L1 and L2 caches as shared memory and threads being mapped to processing units is shown in figure 3.5. Additionally, multiple elements can be computed per thread which enables the utilization of vectorization depending on the type of the elements.

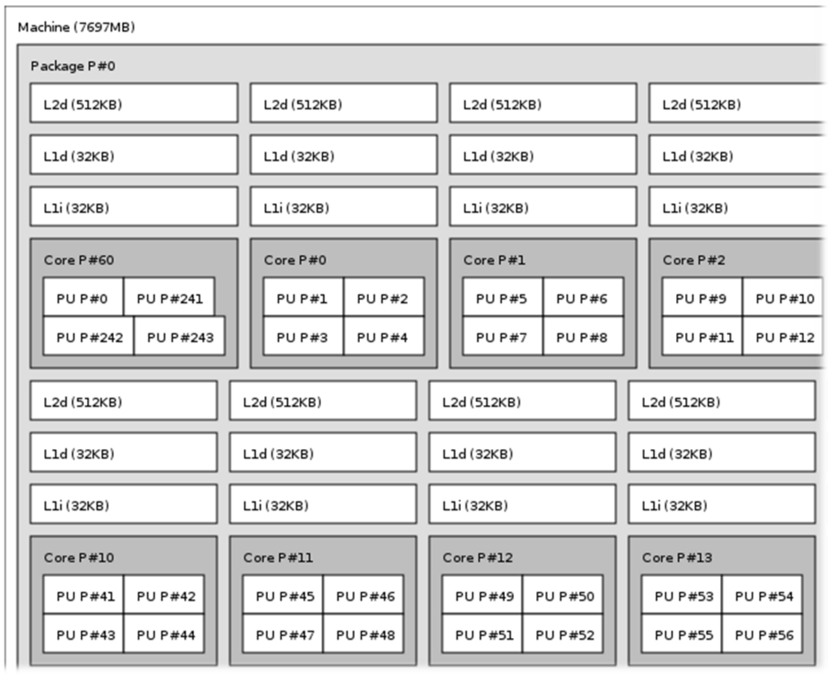

Figure 3.4: A section from the compute and memory hierarchy of a *Intel* Xeon Phi.

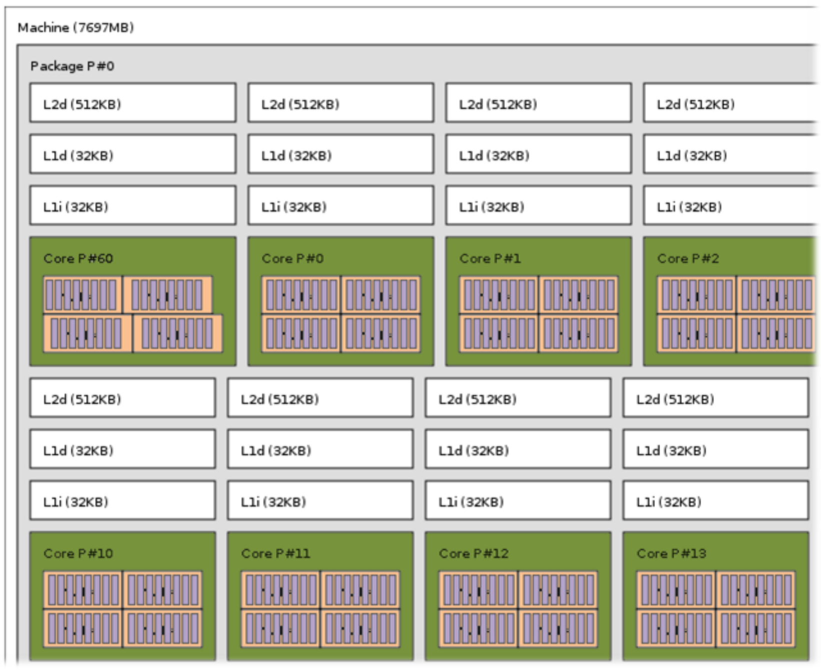

Figure 3.5: Possible mapping of blocks, threads and elements to the compute and memory hierarchy of a *Intel* Xeon Phi. Blocks are mapped to cores, threads to processing units and multiple elements are executed per thread.

3.2 Structure and Implementation of the *alpaka* Interface

To examine if the abstraction can accomplish it‘s design goals, a proof-of-concept library has been implemented. It is called *alpaka*, an acronym for ‘Abstraction Library for Parallel Kernel Acceleration‘. The source code is published[37] as open-source under the *LGPLv3*. It can be used for free in open-source as well as commercial projects. The structure of the library is explained in section 3.2.1.
As described in Chapter 2, the general design of the library is very similar to *CUDA* and *OpenCL* but extends both by some points, while not requiring any language extensions. General interface design as well as interface implementation decisions differentiating *alpaka* from those libraries which are described in section 3.2.2. It uses C++ because it is one of the most performant languages available on nearly all systems. Furthermore, C++11 allows to describe the concepts in a very abstract way that is not possible with many other languages. The *alpaka* library extensively makes use of advanced functional C++ template meta-programming techniques. The implementation details of the C++ library and the way it provides extensibility and optimizability is discussed in section 3.2.3.

3.2.1 Structure

The *alpaka* library allows offloading of computations from the host execution domain to the accelerator execution domain, whereby they are allowed to be identical.
In the abstraction hierarchy the library code is interleaved with user supplied code. This is depicted in figure 3.6. User code invokes library functions, which in turn execute the user provided thread function (kernel) in parallel on the accelerator. The kernel in turn calls library functions when accessing accelerator properties and methods. Additionally, the user can enhance or optimize the library implementations by extending or replacing specific parts.

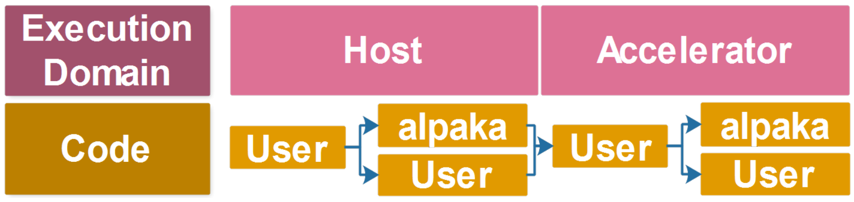

Figure 3.6: Correlation between the execution domain and the code originator.

The *alpaka* abstraction itself only defines requirements a type has to fulfill to be usable with the template functions the library provides. These type constraints are called concepts in C++.

> *A concept is a set of requirements consisting of valid expressions, associated types, invariants, and complexity guarantees. A type that satisfies the requirements is said to model the concept. A concept can extend the requirements of another concept, which is called refinement.*[1]

Concepts allow to safely define polymorphic algorithms that work with objects of many different types. The *alpaka* library implements a stack of concepts and their interactions modeling the abstraction defined in the previous chapter. Furthermore, default implementations for various devices and accelerators modeling those are included in the library. The interaction of the main user facing concepts can be seen in figure 3.7.

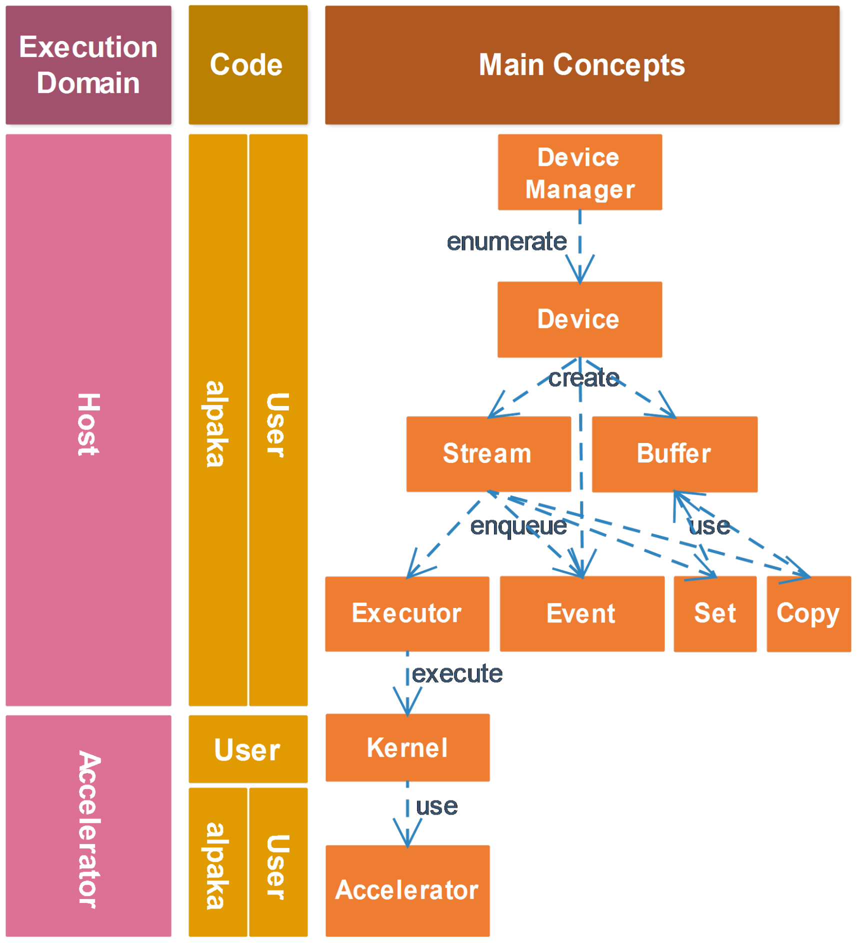

Figure 3.7: Interaction of the main concepts.

For each type of `Device` there is a `DeviceManager` for enumerating the available `Devices`. A `Device` is the requirement for creating `Streams` and `Events` as it is for allocating `Buffers` on the respective `Device` . `Buffers` can be copied, their memory be set and they can be pinned or mapped. Copying and setting a buffer requires the corresponding `Copy` and `Set` tasks to be enqueued into the `Stream`. An `Event` can be enqueued into a `Stream` and its completion state can be queried by the user. It is possible to wait for (synchronize with) a single `Event`, a `Stream` or a whole `Device`. An `Executor` can also be enqueued into a `Stream` and will execute the `Kernel` if all previous tasks in the stream have been completed. The `Kernel` in turn has access to the `Accelerator` it is running on. The `Accelerator` provides the `Kernel` with its current index in the block or grid, their extents or other data as well as it allows to allocate shared memory, execute atomic operations and many more.

3.2.2 User Interface

3.2.2.1 Thread Function

Many parallelization libraries / frameworks do not fully support the separation of the parallelization strategy from the algorithm itself. *OpenMP*, for example, fully mixes the per thread algorithm and the parallelization strategy. This can be seen in the source listing 3.8 showing a simple AXPY computation.

```
 1 template<
 2     typename TSize,
 3     typename TElem>
 4 void axpy(
 5     TSize const n,
 6     TElem const alpha,
 7     TElem const * const X,
 8     TElem * const Y)
 9 {
10     #pragma omp parallel for
11     for (i=0; i<n; i++)
12     {
13         Y[i] = alpha * X[i] + Y[i];
14     }
15 }
```

Figure 3.8: AXPY with *OpenMP*.

Only one line of the function body, line 13, is the algorithm itself, while all surrounding lines represent the parallelization strategy. In *OpenACC* the parallelization and the algorithm are similarly combined.
CUDA, *OpenCL* and other libraries allow, at least to some degree, to separate the algorithm from the parallelization strategy. They define the concept of a kernel representing the algorithm itself which is then parallelized depending on the underlying hardware. The AXPY *CUDA* kernel source code shown in figure 3.9 consists only of the code of one single iteration.

```
 1 template<
 2     typename TSize,
 3     typename TElem>
 4 __global__ void axpy(
 5     TSize const n,
 6     TElem const alpha,
 7     TElem const * const X,
 8     TElem * const Y)
 9 {
10     TSize const i(blockIdx.x*blockDim.x + threadIdx.x)
11     if(i < n)
12     {
13         Y[i] = alpha * X[i] + Y[i];
14     }
15 }
```

Figure 3.9: AXPY with *CUDA*.

On the other hand the *CUDA* implementation is bloated with code handling the inherent blocking scheme. Even if the algorithm does not utilize blocking, as it is the case here, the algorithm writer has to calculate the global index of the current thread by hand (line 10). Furthermore, to support vectors larger then the predefined maximum number of threads per block (1024 for current *CUDA* devices), multiple blocks have to be used. When the number of blocks does not divide the number of vector elements, it has to be assured that the threads responsible for the vector elements behind the given length, do not access the memory to prevent a possible memory access error.
By using the kernel concept, the parallelization strategy, whether all elements are executed in sequential order, in parallel or blocked is not hard coded into the algorithm itself. The possibly multidimensional nested loops do not have to be written by the user. For example, six loops would be required to emulate

the *CUDA* execution pattern with a grid of blocks consisting of threads.
Furthermore the kernel concept breaks the algorithm down to the per element level. Recombining multiple kernel iterations to loop over lines, columns, blocks or any other structure is always possible by changing the calling code and does not require a change of the kernel. In contrast, by using *OpenMP* this would not be possible. Therefore the *alpaka* interface builds on the kernel concept, being the body of the corresponding standard for loop executed in each thread.

3.2.2.2 Function Execution Domain Specifications

CUDA requires the user to annotate its functions with execution domain specifications. Functions that can only be executed on the GPU have to be annotated with `__device__`, functions that can be executed on the host and on the GPU have to be annotated with `__host__ __device__` and host only functions can optionally be annotated with `__host__`. The nvcc *CUDA* compiler uses these annotations to decide with which back-ends a function has to be compiled. Depending on the compiler in use, *alpaka* defines the macros `ALPAKA_FN_HOST`, `ALPAKA_FN_ACC` and `ALPAKA_FN_HOST_ACC` with the identical meaning which can be used in the same positions. When the *CUDA* compiler is used, they are defined to their *CUDA* equivalents, else they are empty.

3.2.2.3 Accelerator Executable Functions

Functions that should be executable on an accelerator do not only have to be annotated with the execution domain. They most probably also require access to the accelerator data and methods, such as indices and extents as well as functions to allocate shared memory and to synchronize all threads within a block. As it was explained previously, there are no implicit built-in variables and functions in *alpaka*. Therefore the accelerator has to be passed in as a template variable. Both, the execution domain specification combined with the accelerator reference parameter can be seen in figure 3.10.

```
template<
    typename TAcc>
ALPAKA_FN_ACC auto doSomethingOnAccelerator(
    TAcc const & acc,
    ...)
-> void
{
    //...
}
```

Figure 3.10: Definition of a function with execution domain specification and accelerator reference parameter.

3.2.2.4 Kernel Definition

The *alpaka* library requires the kernel entry point to be a function object. This means, the kernel is an object that has implemented the `operator()` member function and can be called like any other function. This allows, for example, to use C++11 lambdas, std::function objects or arbitrary other functions as kernels and not only global functions which are required by *CUDA*. Furthermore, unlike *CUDA*, the *alpaka* library does not differentiate between the kernel function that represents the entry point and other functions that can be executed on the accelerator. The entry point function that has to be annotated with `__global__` in *CUDA* is internal to the *alpaka CUDA* back-end and is not exposed to the user. It directly calls into the user supplied kernel function object whose invocation operator is declared with `ALPAKA_FN_ACC`, which equals `__device__` in *CUDA*. Thus there is no difference between the kernel entry point function and any other accelerator function in *alpaka*. Figure 3.11 shows a basic example kernel function object.

```
struct MyKernel
{
    template<
        typename TAcc>
    ALPAKA_FN_ACC static auto operator()(
        TAcc & acc/*,
        ...*/) const
    -> void
    {
        //...
    }
    // Members ...
};
```

Figure 3.11: Prototype definition of a user kernel function object.

The kernel function object is shared across all threads in all blocks. Due to the block execution order being undefined, there is no safe and consistent way of altering state that is stored inside of the function object. Therefore, the `operator()` of the kernel function object has to be `const` and is not allowed to modify any of the object members. This is especially important for the *CUDA* back-end, as it could possibly use the constant memory of the GPU to store the function object. The constant memory is a fast, cached, read-only memory that is beneficial when all threads uniformly read from the same address at the same time. In this case it is as fast as a read from a register.

3.2.2.5 Index and Work Division

CUDA requires the user to calculate the global index of the current thread within the grid by hand (already shown in fig. 3.9). On the contrary, *OpenCL* provides the methods `get_global_size`, `get_global_id`, `get_local_size` and `get_local_id`. Called with the required dimension, they return the corresponding local or global index or extent (size). In *alpaka* this idea is extended to all dimensions. The `alpaka::workdiv::getWorkDiv` and the `alpaka::idx::getIdx` functions both return a vector of the dimensionality the accelerator has be defined with. To unify the method interface and to avoid confusion between the differing terms and meanings of the functions in *OpenCL* and *CUDA*, in *alpaka* these methods are template functions. They are parametrized by the origin of the calculation as well as the unit in which the values are calculated. For example, `alpaka::workdiv::getWorkDiv<alpaka::Grid, alpaka::Threads>(acc)` returns a vector with the extents of the grid in units of threads.

3.2.2.6 Interface Properties

The *alpaka* library is different from other similar libraries (especially *CUDA*) in that it refrains from using implicit or hidden state. This and other interface design decisions will be explained int the following paragraphs.

No Current Device: The *CUDA* runtime API for example supplies a current device for each user code kernel-thread. Working with multiple devices requires to call `cudaSetDevice` to change the current device whenever an operation should be executed on a non-current device. Even the functions for creating a stream (`cudaStreamCreate`) or an event (`cudaEventCreate`) use the current device without any way to create them on a non current device. In the case of an event this dependency is not obvious, since at the same time streams can wait for events from multiple devices allowing cross-device synchronization without any additional work. So conceptually an event could also have been implemented device independently. This can lead to hard to track down bugs due to the non-explicit dependencies, especially in multi-threaded code using multiple devices.

No Default Device: In contrast to the *CUDA* runtime API *alpaka* does not provide a device by default per kernel-thread. Especially in combination with *OpenMP* parallelized host code this keeps users from surprises. The code snippet that is shown in figure 3.12 does not necessarily do what one would expect.

```
cudaSetDevice(1);

#pragma omp parallel for
for(int i = 0; i<10; ++i)
{
    kernel<<<blocks,threads>>>(i);
}
```

Figure 3.12: *OpenMP* annotated code using *CUDA* illustrating undesired default device effects.

Depending on what the *CUDA* runtime API selects as default device for each of the *OpenMP* threads (due to each of them having its own current device), not all of the kernels will necessarily run on device one.

In the *alpaka* library all such dependencies are made explicit. All functions depending on a device require it to be given as a parameter. The *alpaka CUDA* back-end checks before forwarding the calls to the *CUDA* runtime API whether the current device matches the given one and changes it if required. The *alpaka CUDA* back-end does not reset the current device to the one prior to the method invocation out of performance considerations. This has to be considered when native *CUDA* code is combined with *alpaka* code.

No Default Stream: *CUDA* allows to execute commands without specifying a stream. The default stream that is used synchronizes implicitly with all other streams on the device. If a command stream is issued to the default, all other asynchronous streams have to wait before executing any new commands, even when they have been enqueued much earlier. This can introduce hard to track down performance issues. As of *CUDA* 7.0 the default stream can be converted to a non synchronizing stream with a compiler option. Because concurrency is crucial for performance and users should think about the dependencies between their commands from begin on, *alpaka* does not provide such a default stream. All asynchronous operations (kernel launches, memory copies and memory sets) require a stream to be executed in.

No Implicit Built-in Variables and Functions: Within *CUDA* device functions (functions annotated with `__global__` or `__device__`) built-in functions (`__syncthreads`, `__threadfence`, `atomicAdd`,

...) and variables (`gridDim`, `blockIdx`, `blockDim`, `threadIdx`, `warpSize`, ...) are provided.
It would have been possible to emulate those implicit definitions by forcing the kernel function object to inherit from a class providing these functions and members. However functions outside the kernel function object would then pose a problem. They do not have access to those functions and members, the function object has inherited. To circumvent this, the functions and members would have to be public, the inheritance would have to be public and a reference to the currently executing function object would have to be passed as parameter to external functions. This would have been too cumbersome and inconsistent. Therefore access to the accelerator is given to the user kernel function object via one special input parameter representing the accelerator. After that this accelerator object can simply be passed to other functions. The built-in variables can be accessed by the user via query functions on this accelerator.

Abandoning all the implicit and default state makes it much easier for users of the library to reason about their code.

No Language Extensions: Unlike *CUDA*, the *alpaka* library does not extend the C++ language with any additional variable qualifiers (`__shared__`, `__constant__`, `__device__`) defining the memory space. Instead of those qualifiers *alpaka* provides accelerator functions to allocate memory in different the different memory spaces.

No Dimensionality Restriction: *CUDA* always uses three-dimensional indices and extents, even though the task may only be one or two dimensional. *OpenCL* on the other hand allows grid and block dimensions in the range $[1, 3]$ but does not provide corresponding n-dimensional indices, but rather provides functions like `get_global_id` or `get_local_id`, which require the dimension in which the one-dimensional ID is to be queried as a parameter. By itself this is no problem, but how can be assured that a two-dimensional kernel is called with grid and block extents of the correct dimensionality at compile time? How can it be assured that a kernel which only uses `threadIdx.x` or equivalently calls `get_global_id(0)` will not get called with two dimensional grid and block extents? Because the result in such a case is undefined, and most of the time not wanted by the kernel author, this should be easy to check and reject at compile-time. In *alpaka* all accelerators are templatized on the dimensionality. This allows a two-dimensional image filter to assert that it is only called with a two dimensional accelerator. Thereby the algorithms can check for supported dimensionality of the accelerator at compile time instead of runtime. Furthermore with the dimension being a template parameter, the CPU back-end implementations are able to use only the number of nested loops really necessary instead of the 6 loops (2×3 loops for grid blocks and block threads), which are mandatory to emulate the *CUDA* threaded blocking scheme.

By hiding all the accelerator functionality inside of the accelerator object that is passed to the user kernel, the user of the *alpaka* library is not faced with any non-standard C++ extensions. Nevertheless the *CUDA* back-end internally uses those language extensions.

Integral Sizes of Arbitrary Type: The type of sizes such as extents, indices and related variables are depending on a template parameter of the accelerator and connected classes. This allows the kernel to be executed with sizes of arbitrary ranges. Thereby it is possible to force the accelerator back-ends to perform all internal index, extent and other integral size depending computations with a given precision. This is especially useful on current *NVIDIA* GPUs. Even though they support 64-bit integral operations, they are emulated with multiple 32-bit operations. This can be a huge performance penalty when the sizes of buffers, offsets, indices and other integral variables holding sizes are known to be limited.

No synchronous and asynchronous function versions: *CUDA* provides two versions of many of the runtime functions, for example, `cudaMemcpyAsync` and `cudaMemcpy`. The asynchronous version requires a stream while the synchronous version does not need a stream parameter. The asynchronous version immediately returns control back to the caller while the task is enqueued into the given stream and executed later in parallel to the host code. The synchronous version waits for the task to finish before the function call returns control to the caller. Inconsistently, all kernels in a *CUDA* program can only be started either asynchronously by default or synchronously if `CUDA_LAUNCH_BLOCKING` is defined. There is no way to specify this on a per kernel basis. To switch a whole application from asynchronous to synchronous calls, for example for debugging reasons, it is necessary to change the names of all the runtime functions being called as well as their parameters. In *alpaka* this is solved by always enqueuing all tasks into a stream and not defining a default stream. Asynchronous streams as well as synchronous streams are provided for all devices. Changes to the synchronicity of multiple tasks can be made on a per stream basis by changing the stream type at the place of creation. There is no need to change any line of calling code.

3.2.2.7 Memory Management

Memory buffers can not only be identified by the pointer to their first byte. The C++ `new` and `malloc`, the *CUDA* `cudaMalloc` as well as the *OpenCL* `clCreateBuffer` functions all return a plain pointer. This is not enough when working with multiple accelerators and multiple devices. To know where a specific pointer was allocated, additional information has to be stored to uniquely identify a memory buffer on a specific device. Memory copies between multiple buffers additionally require the buffer extents and pitches to be known. Many APIs, for example *CUDA*, require the user to store this information externally. The memory allocation function of the *alpaka* library (`alpaka::mem::buf::alloc<TElem>(device, extents)`) is uniform for all devices, even for the host device. It does not return raw pointers but reference counted memory buffer objects that remove the necessity for manual freeing and the possibility of memory leaks. Additionally the memory buffer objects know their extents, their pitches as well as the device they reside on. This allows buffers that possibly reside on different devices with different pitches to be copied only by providing the buffer objects as well as the extents of the region to copy (`alpaka::mem::view::copy(bufDevA, bufDevB, copyExtents)`).

3.2.2.8 Kernel Execution

The following subsection will discuss the source code listing shown in figure 3.13.

As described in section 3.2.2.6 the dimensionality of the task as well as the type for index and extent sizes have to be defined explicitly. Following this, the type of accelerator to execute on, as well as the type of the stream have to be defined. For both of these types instances have to be created. For the accelerator this has to be done indirectly by enumerating the required device via the device manager, whereas the

```
// Define the dimensionality of the task.
using Dim = alpaka::dim::DimInt<1u>;
// Define the type of the sizes.
using Size = std::size_t;
// Define the accelerator to use.
using Acc = alpaka::acc::AccCpuSerial<Dim, Size>;
// Select the stream type.
using Stream = a::stream::StreamCpuAsync;

// Select a device to execute on.
auto devAcc(a::dev::DevManT<Acc>::getDevByIdx(0));
// Create a stream to enqueue the execution into.
Stream stream(devAcc);

// Create a 1-dimensional work division with 256 blocks a 16 threads.
auto const workDiv(alpaka::workdiv::WorkDivMembers<Dim, Size>(256u, 16u);
// Create an instance of the kernel function object.
MyKernel kernel;
// Create the execution task.
auto const exec(alpaka::exec::create<Acc>(workDiv, kernel/*, arguments ...*/);
// Enqueue the task into the stream.
alpaka::stream::enqueue(stream, exec);
```

Figure 3.13: Execution of a kernel by enqueuing the execution task into a stream.

stream can be created directly.

To execute the kernel shown in section 3.2.2.4, an instance of the kernel function object has to be constructed. Following this, an execution task combining the work division (grid and block sizes) with the kernel function object and the bound invocation arguments has to be created. After that this task can be enqueued into a stream for immediate or later execution (depending on the stream used). In appendix A.1 a minimal functional example of a kernel implementing a generalized vector addition, its invocation, as well as all surrounding *alpaka* operations such as memory allocations and copies, is shown and directly compared to the native *CUDA* implementation.

3.2.3 Implementation in C++

The full stack of concepts defined by the *alpaka* library and their inheritance hierarchy is shown in the third column of figure 3.14. Default implementations for those concepts can be seen in the blueish columns. The various accelerator implementations, shown in the lower half of the figure, only differ in some of their underlying concepts but can share most of the base implementations. The default implementations can, but do not have to be used at all. They can be replaced by user code in arbitrary granularity. By substituting, for instance, the atomic operation implementation of an accelerator, the execution can be fine-tuned, to better utilize the hardware instruction set of a specific processor. However, also complete accelerators, devices and all of the other concepts can be implemented by the user without the need to change any part of the *alpaka* library itself. The way this and other things are implemented is explained in the following paragraphs.

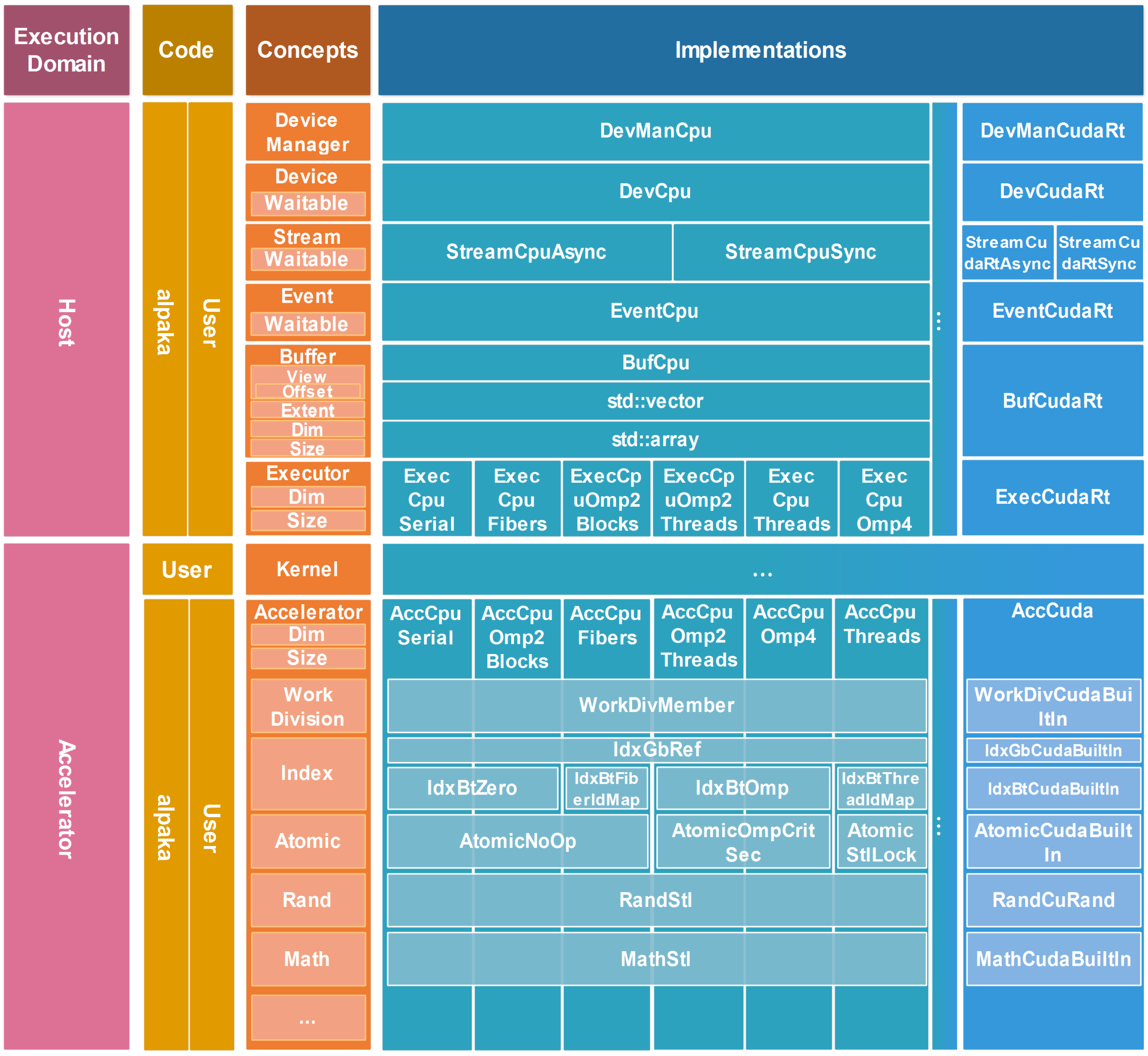

Figure 3.14: Overview of the structure of the *alpaka* library with concepts and implementations.

3.2.3.1 Concept Implementations

The *alpaka* library has been implemented with extensibility in mind. This means that there are no predefined classes, modeling the concepts, the *alpaka* functions require as input parameters. They allow arbitrary types as parameters, as long as they model the required concept.

C++ provides a language inherent object oriented abstraction allowing to check that parameters to a function comply with the concept they are required to model. By defining interface classes, which model the *alpaka* concepts, the user would be able to inherit his extension classes from the interfaces he wants to model and implement the abstract virtual methods the interfaces define. The *alpaka* functions in turn would use the corresponding interface types as their parameter types. For example, the `Buffer` concept requires methods for getting the pitch or changing the memory pinning state. With this intrusive object oriented design pattern the `BufCpu` or `BufCudaRt` classes would have to inherit from an `IBuffer` interface and implement the abstract methods it declares. An example of this basic pattern is shown in figure 3.15.

```cpp
struct IBuffer
{
    virtual std::size_t getPitch() const = 0;
    virtual void pin() = 0;
    virtual void unpin() = 0;
    ...
};

struct BufCpu : public IBuffer
{
    virtual std::size_t getPitch() const override { ... }
    virtual void pin() override { ... }
    virtual void unpin() override { ... }
    ...
};

ALPAKA_FN_HOST auto copy(
    IBuffer & dst,
    IBuffer const & src)
-> void
{
    ...
}
```

Figure 3.15: Enforcement of concepts with basic C++ object orientation.

The compiler can then check at compile time that the objects the user wants to use as function parameters can be implicitly cast to the interface type, which is the case for inherited base classes. The compiler returns an error message on a type mismatch. However, if the *alpaka* library were using those language inherent object oriented abstractions, the extensibility and optimizability it promises would not be possible. Classes and run-time polymorphism require the implementer of extensions to intrusively inherit from predefined interfaces and override special virtual functions.

This is feasible for user defined classes or types where the source code is available and where it can be changed. The `std::vector` class template on the other hand would not be able to model the `Buffer` concept because we can not change its definition to inherit from the `IBuffer` interface class since it is part of the standard library. The standard inheritance based object orientation of C++ only works well when all the code it is to interoperate with can be changed to implement the interfaces. It does not enable interaction with unalterable or existing code that is too complex to change, which is the reality in the majority of software projects.

Another option to implement an extensible library is to follow the way the C++ standard library uses. It allows to specialize function templates for user types to model concepts without altering the types themselves. For example, the `std::begin` and `std::end` free function templates can be specialized for user defined types. With those functions specialized, the C++11 range-based for loops (`for(auto & i : userContainer){...}`) [14, C++ Standard 6.5.4/1] can be used with user defined types. Equally specializations of `std::swap` and other standard library function templates can be defined to extend those with support for user types. One Problem with function specialization is, that only full specializations are allowed. A partial function template specialization is not allowed by the standard. Another problem can emerge due to users carelessly overloading the template functions instead of specializing them. Mixing function overloading and function template specialization on the same base template function can result in unexpected results. The reasons and effects of this are described more closely in an article from H. Sutter (currently convener of the ISO C++ committee) in the *C/C++ Users Journal* [34].

The solution given in the article is to provide *"a single function template that should never be specialized or overloaded"*. This function simply forwards its arguments *"to a class template containing a static function with the same signature"*. This template class can fully or partially be specialized without affecting overload resolution.

The way the *alpaka* library implements this is by not using the C++ inherent object orientation but lifting those abstractions to a higher level. Instead of using a non-extensible `class`/`struct` for defining the interface, a namespace is utilized. In place of abstract virtual member functions of the interface, *alpaka* defines free functions within those namespaces. All those functions are templates allowing the user to call them with arbitrary self defined types and not only those inheriting from a special interface type. Unlike member functions, they have no implicit `this` pointer, so the object instance has to be explicitly given as a parameter. Overriding the abstract virtual interface methods is replaced by the specialization of a template type that is defined for each such namespace function.

A concept is completely implemented by specializing the predefined template types. This allows to extend and fine-tune the implementation non-intrusively. For example, the corresponding pitch and memory pinning template types can be specialized for `std::vector`. After doing this, the `std::vector` can be used everywhere a buffer is accepted as argument throughout the whole *alpaka* library without ever touching its definition.

A simple function allowing arbitrary tasks to be enqueued into a stream can be implemented in the way shown in figure 3.16. The `TSfinae` template parameter will be explained in section 3.2.3.2.

A user who wants his stream type to be used with this `enqueue` function has to specialize the `Enqueue` template struct. This can be either done partially by only replacing the `TStream` template parameter and accepting arbitrary tasks or by fully specializing and replacing both `TStream` and `TTask`. This gives the user complete freedom of choice. The example given in figure 3.17 shows this by specializing the `Enqueue` type for a user stream type `UserStream` and arbitrary tasks. In addition figure 3.18 shows a full specialization for a given `UserStream` and a `UserTask`.

When the `enqueue` function template is called with an instance of `UserStream`, the most specialized version of the `Enqueue` template is selected depending on the type of the task `TTask` it is called with.

A type can model the stream concept completely by defining specializations for `alpaka::stream::`

```cpp
namespace stream
{
    template<
        typename TStream,
        typename TTask,
        typename TSfinae = void>
    struct Enqueue;

    template<
        typename TStream,
        typename TTask>
    ALPAKA_FN_HOST auto enqueue(
        TStream & stream,
        TTask & task)
    -> void
    {
        Enqueue<
            TStream,
            TTask>
        ::enqueue(
            stream,
            task);
    }
}
```

Figure 3.16: Definition of the function to enqueue tasks into a stream.

```cpp
struct UserStream{};

namespace stream
{
    // partial specialization
    template<
        typename TTask>
    struct Enqueue<
        UserStream
        TTask>
    {
        ALPAKA_FN_HOST static auto enqueue(
            UserStream & stream,
            TTask & task)
        -> void
        {
            //...
        }
    };
}
```

Figure 3.17: Partial specialization of the `Enqueue` type to extend the functionality for a user type.

```cpp
struct UserStream{};
struct UserTask{};

namespace stream
{
    // full specialization
    template<>
    struct Enqueue<
        UserStream
        UserTask>
    {
        ALPAKA_FN_HOST static auto enqueue(
            UserStream & stream,
            UserTask & task)
        -> void
        {
            //...
        }
    };
}
```

Figure 3.18: Full specialization of the `Enqueue` type to extend the functionality for a user type.

`Enqueue` and `alpaka::stream::Empty`. This functionality can be accessed by the corresponding `alpaka::stream::enqueue` and `alpaka::stream::empty` template functions.

Currently there is no native language support for describing and checking concepts in C++ at compile time. A study group (SG8) is working on the ISO specification [15] and compiler forks implementing them do exist. For usage in current C++ there are libraries like *Boost.ConceptCheck* ([33]) which try to emulate requirement checking of concept types. Those libraries often exploit the preprocessor and require non-trivial changes to the function declaration syntax. Therefore the *alpaka* library does not currently make use of *Boost.ConceptCheck*. Neither does it facilitate the proposed concept specification due to its dependency on non-standard compilers.

The usage of concepts as described in the working draft would often dramatically enhance the compiler error messages in case of violation of concept requirements. Currently the error messages are pointing deeply inside the stack of library template invocations where the missing method or the like is called. Instead of this, with concept checking it would directly fail at the point of invocation of the outermost template function with an expressive error message about the parameter and its violation of the concept requirements. This would simplify especially the work with extendable template libraries like *Boost* or *alpaka*. However, in the way concept checking would be used in the *alpaka* library, omitting it does not change the semantic of the program, only the compile time error diagnostics. In the future when the standard incorporates concept checking and the major compilers support it, it will be added to the *alpaka* library.

3.2.3.2 Template Specialization Selection on Arbitrary Conditions

Basic template specialization only allows for a selection of the most specialized version where all explicitly stated types have to be matched identically. It is not possible to enable or disable a specialization based on arbitrary compile time expressions depending on the parameter types. To allow such conditions, *alpaka* adds a defaulted and unused `TSfinae` template parameter to all declarations of the implementation template structs. This was shown in figure 3.16 using the example of the `Enqueue` template type. The C++ technique called SFINAE, an acronym for *Substitution failure is not an error* allows to disable arbitrary specializations depending on compile time conditions. Specializations where the substitution of the parameter types by the deduced types would result in invalid code will not result in a compile error, but will simply be omitted. An example in the context of the `Enqueue` template type is shown in figure 3.19.

```cpp
struct UserStream{};

namespace stream
{
    template<
        typename TStream,
        typename TTask>
    struct Enqueue<
        TStream
        TTask,
        typename std::enable_if<
            std::is_base_of<UserStream, TStream>::value
            && (TTask::TaskId == 1u)
        >::type>
    {
        ALPAKA_FN_HOST static auto enqueue(
            TStream & stream,
            TTask & task)
        -> void
        {
            //...
        }
    };
}
```

Figure 3.19: Specialization of the `Enqueue` type using SFINAE for arbitrary conditions.

The `Enqueue` specialization shown here does not require any direct type match for the `TStream` or the `TTask` template parameter. It will be used in all contexts where `TStream` has inherited from `UserStream` and where the `TTask` has a static const integral member value `TaskId` that equals one. If the `TTask` type does not have a `TaskId` member, this code would be invalid and the substitution would fail. However, due to SFINAE, this would not result in a compiler error but rather only in omitting this specialization. The `std::enable_if` template results in a valid expression, if the condition it contains evaluates to true, and an invalid expression if it is false. Therefore it can be used to disable specializations depending on arbitrary boolean conditions. It is utilized in the case where the `TaskId` member is unequal one or the `TStream` does not inherit from `UserStream`. In this cirumstances, the condition itself results in valid code but because it evaluates to false, the `std::enable_if` specialization results in invalid code and the whole `Enqueue` template specialization gets omitted.

4 Evaluation

To estimate the overhead introduced by the *alpaka* library and to show whether the abstraction introduces any noticeable problems regarding performance portability across multiple hardware architectures, the following chapter discusses measurements with artificial and real world codes. The exact methodology by which the tests have been carried out is described in section 4.1.
To compare the performance of the various back-ends, the generalized vector addition (AXPY) is used as synthetic benchmark in section 4.2 and the generalized matrix multiplication (GEMM) in section 4.3. To show that the abstraction does not only work in artificial cases, the *HASEonGPU*[39] simulation has been ported from *CUDA* to *alpaka*. The original *HASEonGPU* simulation can only be compared directly to the *alpaka* port of the simulation that uses the *CUDA* back-end. All the other *alpaka* back-ends are difficult to compare to because there is no version of *HASEonGPU* using *OpenMP* natively. Therefore *HASEonGPU* will be used to show that a port from *CUDA* to *alpaka* is possible and that the overhead which is introduced by the additional layer is small enough to be compensated by the benefit of hardware independence (sec. 4.4). The only comparison that can be done across all *alpaka* back-ends and all the different hardware devices utilized is to compare the achieved performance relative to the peak performance of the underlying hardware. This comparison is not always useful or meaningful but can give a rough estimate. For example, the properties of the memory hierarchy which could be very different across devices can have much more influence on the speed than the theoretical peak performance. Memory limited applications can, for instance, strongly benefit from the larger cache hierarchy and main memory sizes of current CPUs while other algorithms may gain from the very fast interaction within the small shared memory of current GPUs. All this is not expressed in the theoretical peak performance and will therefore be neglected when comparing across *alpaka* back-ends on different hardware devices.

4.1 Methodology

The measurements that will be presented in the following chapter have been performed on the *Hypnos* [6] cluster of the *Helmholtz-Zentrum Dresden Rossendorf* (HZDR). The measurements were taken on dual-socket nodes with *Intel Xeon* E5-2630 v3 CPUs (figure 4.1), 256 GB of RAM and 8 *NVIDIA* K80 GPUs (figure 4.2). Due to the architecture of the K80 GPU consisting of two independent *CUDA* devices only one half of one of the GPUs has been used. Ideas of how to overcome this limitation that is also present in native *CUDA* code are discussed in section 5.2.

architecture	Haswell-EP (22 nm)
number of cores	8 (16 Hyper-Threads)
clock frequency	2.4 GHz (3.2 GHz turbo)
instruction-set extension	AVX2
memory bandwidth	59 GB/s
L3 cache size	$20MB$
L2 cache size	$8 \times 256KB$
L1i cache size	$8 \times 32KB$
L1d cache size	$8 \times 32KB$

Table 4.1: Specification of the *Intel* Xeon E5-2630 v3 [10].

architecture	$2 \times KeplerGK210$ (28 nm)
compute capability	3.7
number of SMX	2×13
number of CUDA cores	2×2496
clock frequency	562 MHz (875 MHz turbo)
32-bit Tflop/s	2×2.8
64-bit Tflop/s	2×0.935
global memory	2×12 GB
memory bandwidth (ECC off)	2×240 GB/s
L2 cache size	1536 KB

Table 4.2: Specification of the *NVIDIA* K80 GPU [26].

The *HASEonGPU* measurements used *NVIDIA* K20 GPUs 4.3 and some highly parallel CPU measurements were taken on quad-socket nodes with AMD Opteron 6276 CPUs (figure 4.4) and 256 GB of RAM.

architecture	$2 \times KeplerGK110$ (28 nm)
compute capability	3.5
number of SMX	13
number of CUDA cores	2496
clock frequency	705 MHz (875 MHz turbo)
32-bit Tflop/s	3.52
64-bit Tflop/s	1.17
global memory	5 GB
memory bandwidth (ECC off)	208 GB/s
L2 cache size	1536 KB

Table 4.3: Specification of the *NVIDIA* K20 GPU [25].

During the measurements the nodes were always used exclusively, that is, all cores have been reserved, even though some of them may have been unused. This ensures that the measurements are not disturbed by other processes using cores on the respective nodes.

The executables have been built with the *NVIDIA* nvcc compiler from the *CUDA* 7.0 SDK and g++ 4.9.2. Binaries are built for the x86_64 instruction set which implies support for at least SSE2.

architecture	Interlagos (0.32 nm)
number of cores	16
clock frequency	2.3 GHz (3.2 GHz turbo)
instruction-set extension	AVX
max. memory bandwidth	51.2 GB/s
L3 cache size	$16MB$
L2 cache size	$16 \times 1000KB$
L1 cache size	$16 \times 48KB$

Table 4.4: Specification of the *AMD* Opteron 6276 [2].

The minimal runtime of multiple iterations is chosen to filter out disturbances from various sources, for instance, the possibly very long time of initial *CUDA* device context creation on first access.

Since the measurements were performed in a number of separate runs for different back-ends due to compiler incompatibilities, the node assigned by the batch system is not necessarily the same for all measurements. Even though the hardware is the same in all nodes, this results in slightly varying performance characteristics due to hardware inherent production variations and temperature differences. Both, GPUs and CPUs used can produce fluctuations in the results of the measurements due to their turbo boost, a temperature dependent over-clocking.

Multiple versions of the *BLAS* algorithms that are implemented in the following chapter exist for different underlying data types. For example, there is a generalized matrix-matrix-multiplication for single precision (SGEMM), double precision (DGEMM), half precision (HGEMM) and complex floating point numbers (ZGEMM).

Those versions can have very different performance characteristics depending on the hardware they are executed on. On a modern x86 CPU the single precision version will be approximately twice as fast as the double precision version. This is due to eight single precision values (32 bit each) or four double precision values (64 bit each) fitting into a 256 bit AVX vector register at once. The *Intel Xeon Phi* even has 512 bit wide vector registers allowing to operate on eight double precision values at once [11]. A similar but not compatible instruction set extension for 512 bit vector operations will be introduced in the next generation of *Intel* CPUs [12].

On GPUs this ratio can be much worser depending on the architecture. Because there are separate single precision (SP) and double precision (DP) processing units on current *NVIDIA* GPU architectures and the hardware development is mostly lead by graphics applications which are generally single precision only, the number of double precision units is considerably lower. On the current *NVIDIA Tesla* GPUs (K80, K40, K20X and K20) the ratio SP:DP is 3:1 while it is even worse on consumer cards. [25, 26] A GTX 580 has a ratio of 4:1, a GTX 680/780 of 24:1 and the current current high-end card, the GTX 980 has only one double precision unit per 32 single precision units.

The work division, the decomposition of the given grid-element extent into grid-block, block-thread and thread-element extents, is automatically calculated. The block extents are chosen to be the maximum the accelerator allows and can be constrained to be exactly quadratic, nearly quadratic or linear depending on the task. Furthermore, the block extents can be forced to divide the full grid extents without remainder. The native *CUDA* version will always be compared to a *alpaka CUDA* version using the same work division.

Graphs shown in the evaluation will either display absolute execution times of different implementations with varying problem sizes or the speed relative to a specified implementation with varying problem sizes. The relative speed ϵ is defined as $\epsilon = T_{orig}/T_{new}$, where T_{orig} is the time to solution of the original version and T_{new} is the time to solution of the version to compare to. Native implementations are often used as original version, while alpaka versions represent the new versions being compared to. If the *alpaka* versions reach a relative speed factor of one, they are as fast as the native version. If the factor is lower, they are slower.

4.2 Generalized Vector Addition

The generalized vector addition algorithm (AXPY) is part of the first of three levels of the *Basic Linear Algebra Subprograms* (BLAS) specification. It computes $\boldsymbol{Y} \leftarrow \alpha\boldsymbol{X} + \boldsymbol{Y}$ where $\boldsymbol{X}$ and $\boldsymbol{Y}$ are vectors while α is a scaling factor. The AXPY shorthand is derived from the formula pronounced *Alpha X Plus Y*. The standard sequential algorithm is shown in the source code listing 4.1.

```
template<
    typename TSize,
    typename TElem>
auto axpy(
    TSize const & n,
    TElem const & alpha,
    TElem const * const X,
    TElem * const Y) const
-> void
{
    for(TSize i = 0; i < n; ++i)
    {
        Y[i] = alpha * X[i] + Y[i];
    }
}
```

Figure 4.1: Source code of a sequential generalized vector addition.

The generalized vector addition is used as a trivial example to showcase the performance characteristics of basic algorithms ported with *alpaka*. The AXPY algorithm can be parallelized perfectly because all elements in the result vector depend on exactly one distinct element of both input vectors and there are no dependencies between the threads. Theoretically it is possible to execute it in constant time when there are as many processing elements as there are vector elements.

In the following sections the double precision version (DAXPY) will be used to compare a native implementation on the respective platform with the generic *alpaka* version using the corresponding back-end. The input vectors are densely filled with random values in the range $[0.0, 10.0]$. Only the time needed to compute the result is measured. The time for allocating the vectors on the host, filling them, a possible data transfer between the processor and a co-processor as well as device and stream initialization are not included.

Naturally all the *alpaka* tests on all platforms will use the same generic *alpaka* AXPY function object without any platform specific optimizations.

For benchmarking the different AXPY versions a library called *vecadd* [38] has been written. It is published as open-source under the *LGPLv3*.

4.2.1 CPU Sequential Execution

A basic implementation of the AXPY Kernel starts exactly one thread per vector element (fig. 4.2).

```
struct AxpyKernel
{
    template<
        typename TAcc,
        typename TSize,
        typename TElem>
    ALPAKA_FN_ACC auto operator()(
        TAcc const & acc,
        TSize const & n,
        TElem const & alpha,
        TElem const * const X,
        TElem * const Y) const
    -> void
    {
        auto const i(alpaka::idx::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0u]);
        if(i < n)
        {
            Y[i] = alpha * X[i] + Y[i];
        }
    }
};
```

Figure 4.2: Source code of a generalized vector addition *alpaka* kernel function object.

When directly comparing the basic sequential algorithm (fig. 4.1) with this *alpaka* kernel using the sequential back-end (fig. 4.2) the expected linearly raising computation time can be observed (fig. B.1 in the appendix). The *alpaka* sequential version reaches approximately 78% of the speed of the native sequential version as can be seen in figure 4.3.

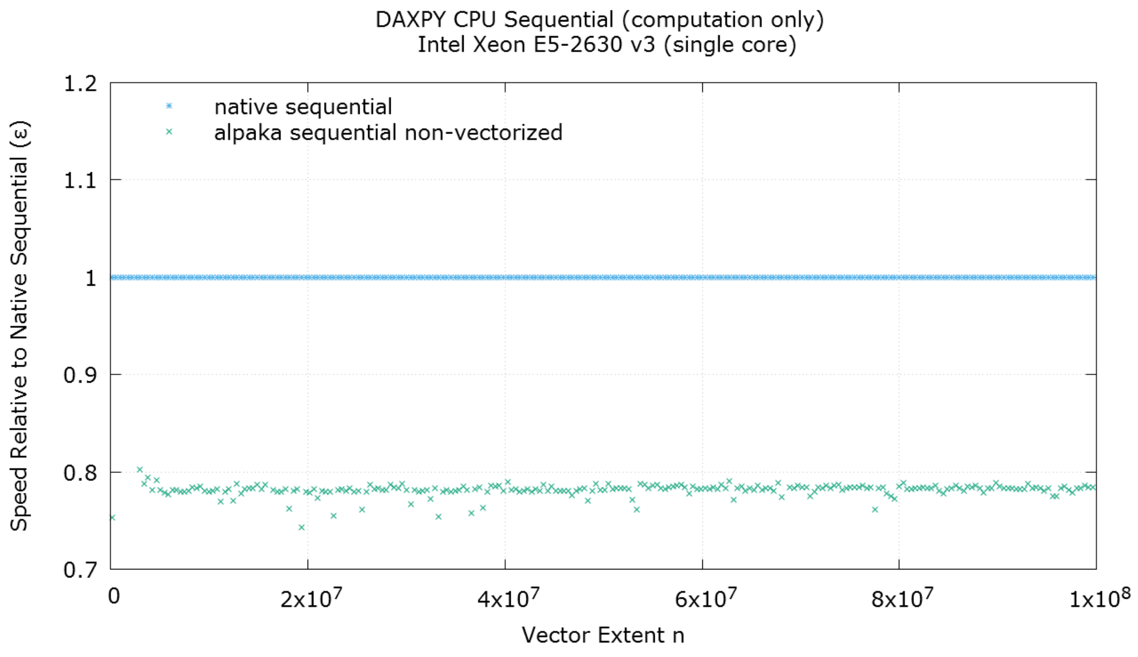

Figure 4.3: Comparison of the *alpaka* non-vectorized sequential relative to the native sequential generalized vector addition.

Looking at the assembler code being generated by the compiler (fig. A.2 in the appendix) it is possible to spot the difference. It can be seen that the compiler was not able to recognize the looping pattern due to the obfuscation by the conditional branch within the kernel. Even though the compiler was able to inline all the recursive function invocations within *alpaka* right up to the kernel function object invocation, the compiler was not able to optimize the code in the same way. Therefore only the native sequential version has been vectorized to use the packed double precision SSE2 instructions **`movupd`**, **`mulpd`** and **`addpd`** instead of the single value versions **`movsd`**, **`mulsd`** and **`addsd`**.

The assembler code of the *alpaka* version seems to be more streamlined because it has only half the number of instructions between the two timekeepings. But this impression is deceptive. The compiler optimizes the execution speed at the expense of the binary size (space-time trade-off). The additional lines of the native version are not in the inner loop of the algorithm but rather handle the special cases of an odd number of elements or an unaligned start address.

Changing the kernel to incorporate vectorization requires an additional loop within the kernel to compute multiple elements per thread. This loop should not contain any branches to ease the work of the loop vectorizer. The compiler will recognize the iteration independent looping pattern and optimizes this by using SIMD instructions to process multiple consecutive iterations together.

This allows *alpaka* to map the execution to exactly one grid of one block with one thread executing all the elements. The result is an only marginally slower execution time (99.5% of the native version see fig. 4.4) of the kernel shown in figure 4.5. It uses the grid, thread and element levels of the hierarchical redundant parallelism abstraction.

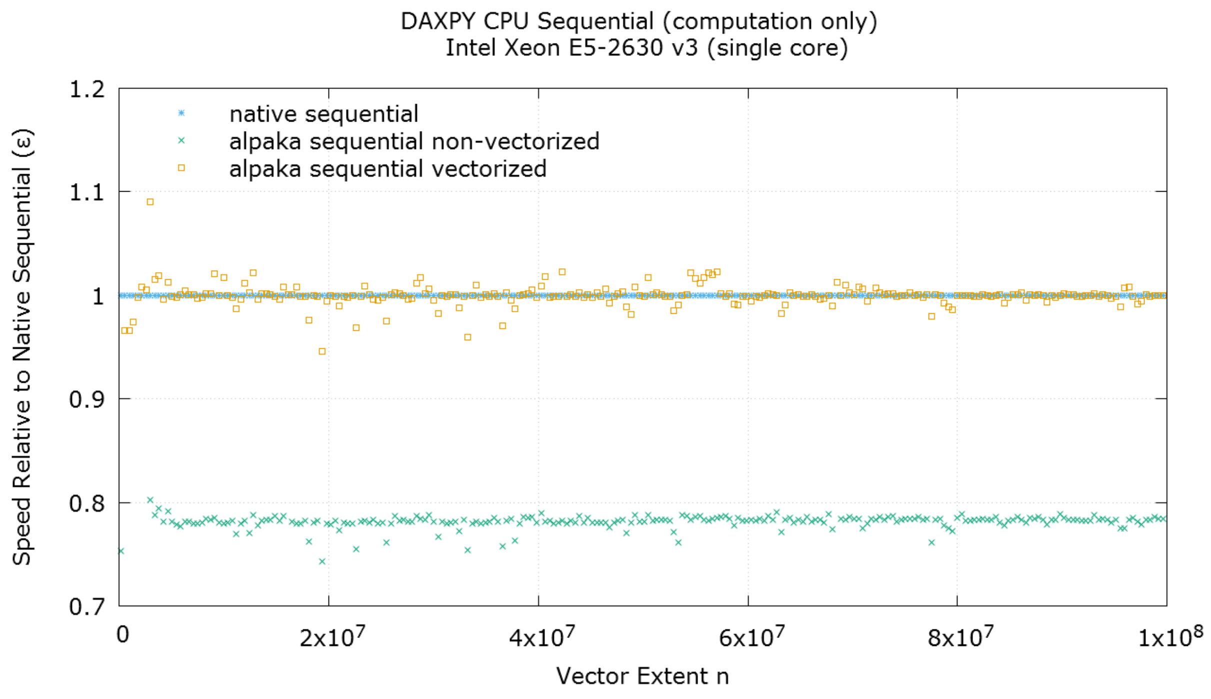

Figure 4.4: Comparison of the *alpaka* sequential vectorized relative to the native sequential generalized vector addition.

```
struct AxpyKernel
{
    template<
        typename TAcc,
        typename TSize,
        typename TElem>
    ALPAKA_FN_ACC auto operator()(
        TAcc const & acc,
        TSize const & n,
        TElem const & alpha,
        TElem const * const X,
        TElem * const Y) const
    -> void
    {
        // Get the index of the current thread in the grid.
        auto const gridThreadIdx(
            alpaka::idx::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0u]);
        // Get the number of elements per thread.
        auto const threadElemExtent(
            alpaka::workdiv::getWorkDiv<alpaka::Thread, alpaka::Elems>(acc)[0u]);
        // Calculate the index of the first element this thread calculates.
        auto const threadFirstElemIdx(
            gridThreadIdx * threadElemExtent);
        if(threadFirstElemIdx < n)
        {
            // All but the last thread calculate exactly threadElemExtent elements.
            // The last thread calculates the remaining elements from
                threadFirstElemIdx to n.
            auto const elems(threadElemExtent + alpaka::math::min(acc, 0, n-(
                threadFirstElemIdx+threadElemExtent)));
            // Loop over the elements this thread has to calculate.
            for(TSize i(threadFirstElemIdx); i<(threadFirstElemIdx+elems); ++i)
            {
                Y[i] = alpha * X[i] + Y[i];
            }
        }
    }
};
```

Figure 4.5: Source code of a generalized vector addition *alpaka* kernel function object using the grid, thread and element levels of the hierarchical redundant parallelism abstraction.

4.2.2 CPU Parallel Execution

The parallel CPU test compares the code that was shown in figure 3.8 with the non-vectorized (figure 4.2)) and the vectorized (figure 4.5)) *alpaka* kernel using the *OpenMP* block parallelizing back-end. *OpenMP* has been set to create 256 threads via the *OMP_NUM_THREADS* environment variable as this resulted in the fastest execution time of many tested thread counts. In figure 4.6 can be seen that the vectorized *alpaka* code is faster then the native *OpenMP* version for smaller vectors. This can be explained by the missing vectorization in the native *OpenMP* version that is as fast as the non-vectorized *alpaka* variant.

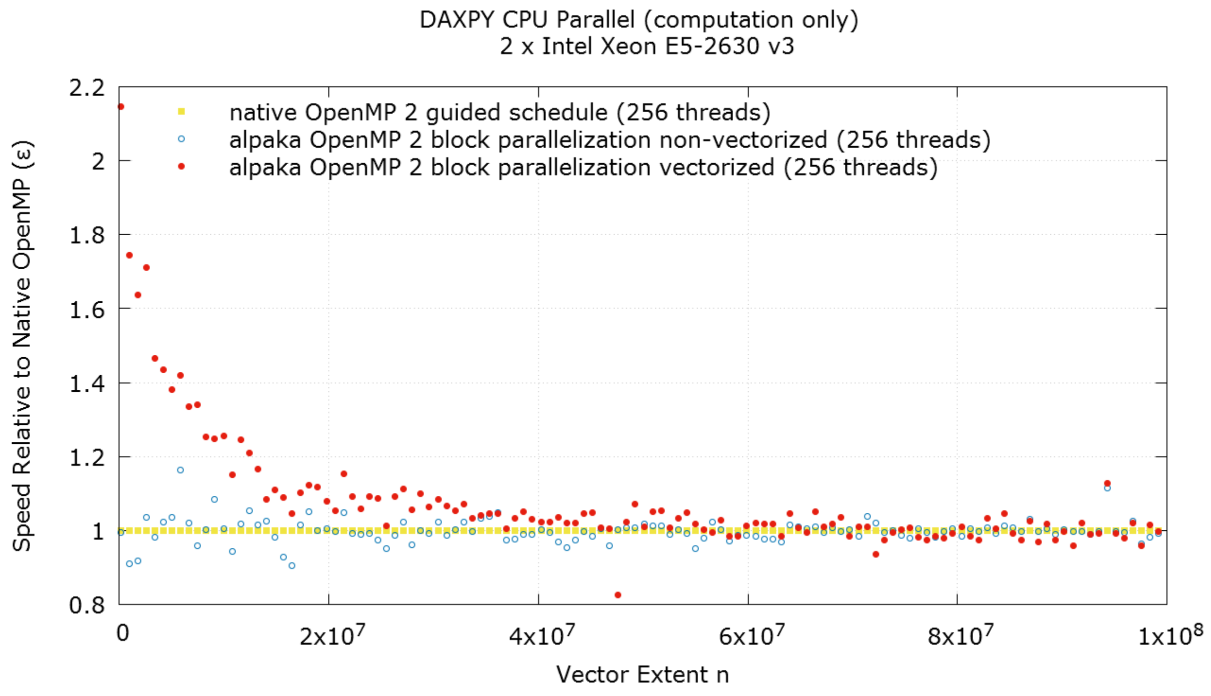

Figure 4.6: Comparison of the *alpaka* *OpenMP* relative to the native *OpenMP* generalized vector addition.

In figure B.2 in the appendix can be seen that the computation time grows linearly with the problem size. Comparing the timings with the previously shown sequential execution times it can be observed that both parallel versions are faster then the sequential versions but by no means as fast as one could have expected from two eight core CPUs. This is due to the nature of the task being fully memory bandwidth limited and not compute bound. For two corresponding elements from the $\boldsymbol{X}$ and $\boldsymbol{Y}$ vectors one addition and one multiplication have to be carried out. In fact, modern CPUs can even combine those two into a single fused-multiply-add instruction (FMA). The task being fully memory bandwidth limited does only give the vectorized version an advantage over the non-vectorized versions as long as the data fits into the caches. For larger vector extents all versions are equally fast due to the limited memory bandwidth.

4.2.3 CUDA Parallel Execution

The parallel *CUDA* test compares the native *CUDA* kernel that was shown in figure 3.9 with the *alpaka* kernel (figure 4.5)) using the *CUDA* back-end.

A comparison of the complete code to compute the DAXPY natively in *CUDA* and via *alpaka* can be found in appendix A.1. This code example also includes the setup of a device, a stream, the buffers, their initialization, copies, the kernel invocation and the following clean-up.
As might be expected due to the different peak performances the versions running on the K80 GPU are faster then those running on the Intel Xeon CPU (fig. B.3 in the appendix).
As can be seen in figure 4.7 there is only a minimal difference between the two *CUDA* versions. The *alpaka* version reaches 99.6% of the speed of the native one. This can further be backed up by comparing the generated PTX codes. The PTX code shown in figure A.3 in the appendix compares the native *CUDA* with the non-vectorized *alpaka* kernel 4.2. It is identical up to two additional but unused function parameters in the *alpaka* variant as well as different internal variable names. The vectorized *alpaka* kernel 4.5, introduces some additional instructions into the PTX code (fig. A.4) due to the additional loop which explains the marginal overhead. This shows that the only *alpaka* abstraction that introduces a minimal overhead within a simple kernel when using the *CUDA* back-end is the optional element level which is necessary to reach good CPU performance. Figure B.4 in the appendix shows that processing more then one element per thread on the K80 GPU does not increase the performance.

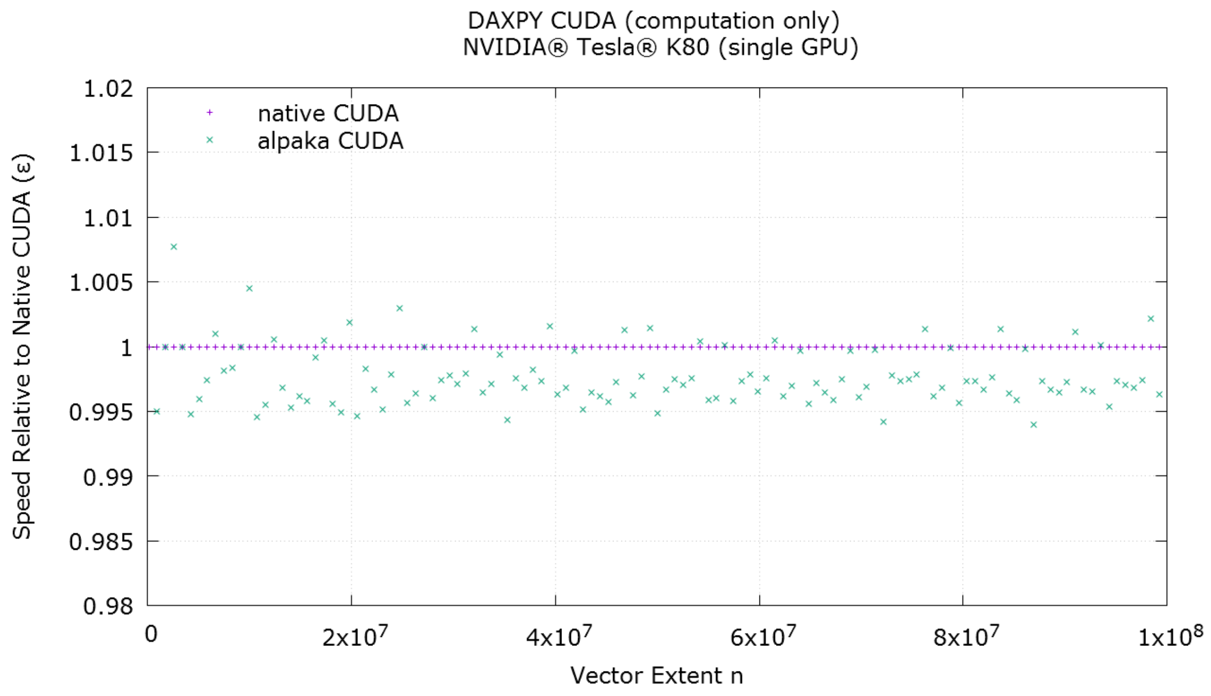

Figure 4.7: Comparison of the *alpaka CUDA* relative to the native *CUDA* generalized vector addition.

4.3 Generalized Matrix-Matrix-Multiplication

The generalized matrix-matrix-multiplication (GEMM) is part of the third of three levels of the *Basic Linear Algebra Subprograms* (BLAS) specification. It computes $\boldsymbol{C} \leftarrow \alpha\boldsymbol{AB} + \beta\boldsymbol{C}$ where $\boldsymbol{C}$, $\boldsymbol{A}$ and $\boldsymbol{B}$ are matrices $A = (a_{i,j})$, $B = (b_{i,j})$ while α and β are scaling factors. Additionally, all the matrices can be strided. The standard sequential cache optimized (ikj loop order) algorithm that computes the equation (4.1) is shown in figure 4.8 and has a complexity of $O(N^3)$.

$$c_{i,j} = \alpha \cdot \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j} + \beta \cdot c_{i,j} \quad 1 \leq i \leq l, 1 \leq j \leq n. \tag{4.1}$$

```
for(size_t i = 0; i < m; ++i){
    for(size_t j = 0; j < n; ++j){
        C[i*ldc + j] = beta * C[i*ldc + j];
    }
    for(size_t k2 = 0; k2 < k; ++k2){
        for(size_t j = 0; j < n; ++j){
            C[i*ldc + j] = alpha * A[i*lda + k] * B[k*ldb + j] + C[i*ldc + j];
        }
    }
}
```

Figure 4.8: Source code of a sequential generalized matrix multiplication (ikj).

The generalized matrix-matrix-multiplication is a thoroughly researched algorithm. It‘s performance characteristics on different hardware types are well known and libraries like the *Intel MKL* (math kernel library [13]) and *NVIDIA cuBLAS* [27] are optimized to get the best results possible on the corresponding hardware. The generalized matrix-matrix-multiplication can utilize all levels of parallelism on GPUs and CPUs ranging, for example, from blocking over shared memory to vectorization. Therefore it should be ideal to showcase the usage of these techniques within the *alpaka* library.

In the following sections the DGEMM will be used to compare a native implementation on the respective platform with the generic *alpaka* version using the corresponding back-end. The input matrices are dense and always have square extents to not have a bias towards implementations preferring column or row-major layout. Initially, the matrices are filled with random values in the range $[0.0, 10.0]$. Only the time needed to compute the result is measured. The time for allocating the vectors on the host, filling them, a possible data transfer between the processor and a co-processor as well as device and stream initialization are not included.

This time, in contrast to the generalized vector addition, the order will be in reverse and the native *CUDA* algorithm will be the starting point.

For benchmarking the different GEMM versions a library called *matmul* [36] has been written. It is published as open-source under the *LGPLv3*.

4.3.1 GEMM on CUDA capable GPUs

On *CUDA* capable GPUs a comparison between a GEMM written in native *CUDA*, an *alpaka* version using the *CUDA* back-end and the GEMM from the *NVIDIA cuBLAS* library will be carried out.

The native *CUDA* GEMM implementation used for measurements extents the algorithm given in the *CUDA* C Programming Guide [28, p. 27-29] that only computes $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$ with the addition of the $\boldsymbol{C}$ matrix and the two scaling factors α and β. The computation uses a blocked execution scheme by taking advantage of the *CUDA* inherent work division of the solution space (grid) into blocks of communicating threads. Each block of threads is responsible for computing a square sub-matrix of the result matrix $\boldsymbol{C}$ as sum over the corresponding sub-matrices of $\boldsymbol{A}$ and $\boldsymbol{B}$ as can be seen in figure 4.9.

The multiplication algorithm takes advantage of the block shared memory. Each thread is loading the one corresponding element from the sub-matrices of $\boldsymbol{A}$ and $\boldsymbol{B}$ from global memory into the shared memory.

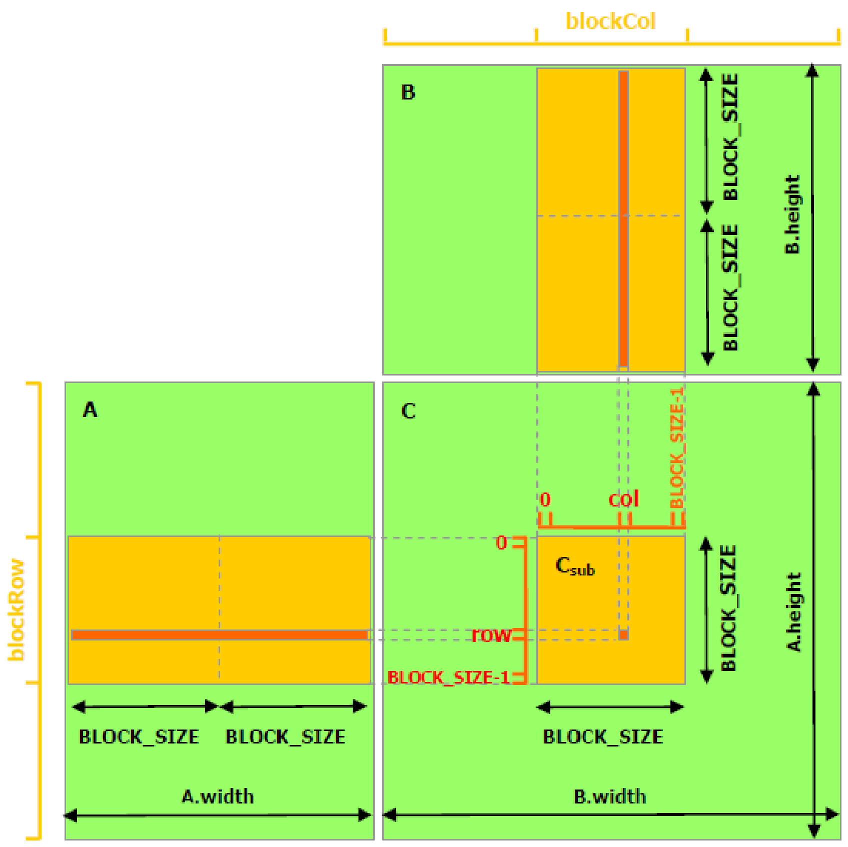

Figure 4.9: Generalized matrix-matrix-multiplication using shared memory [23].

After the synchronization of all threads within a block, the sub-matrices of $\boldsymbol{A}$ and $\boldsymbol{B}$ fully reside within the block shared memory. Now each thread can compute the sub-matrix dot product locally with the data being as close as possible to all of them. By repeating this loading and computation step, accumulating over all sub-matrix dot products of the corresponding pairs of sub-matrices of $\boldsymbol{A}$ and $\boldsymbol{B}$, the final elements of C are computed. The source code of the *CUDA* kernel implementation can be found in the appendix in figure A.5.

The *alpaka* version using the *CUDA* back-end is a nearly direct transcription of the equivalent native CUDA code. The complete kernel code is shown in figure A.6 in the appendix.

The DGEMM from the *NVIDIA cuBLAS* library is fine-tuned for different compute-capabilites and GPUs. In contrast to many other libraries, *cuBLAS* expects the matrices in column-major storage order. Because the storage order of multidimensional C arrays is row-major, we have to change the order of the arguments given to ‘cublasDgemm‘. By swapping the matrix $\boldsymbol{A}$ with the matrix $\boldsymbol{B}$ the result is still correct. Because *cuBLAS* sees the matrices in transposed order due to the inverse storage order it expects, the following computation is executed: $\boldsymbol{C}^T \leftarrow \alpha \boldsymbol{B}^T \boldsymbol{A}^T + \beta \boldsymbol{C}^T$. By reading the transposed result matrix $\boldsymbol{C}^T$ that has been written in column-major order as a row-major matrix we receive the expected untransposed result $\boldsymbol{C}$.

Figure 4.10 shows the expected cubic growth in computation time for increasing matrix sizes.

The variant using *cuBLAS* scales much better than the other two versions and reaches six to seven times the speed of the others (fig. B.5 in the appendix). This is due to *cuBLAS* not only optimizing the block

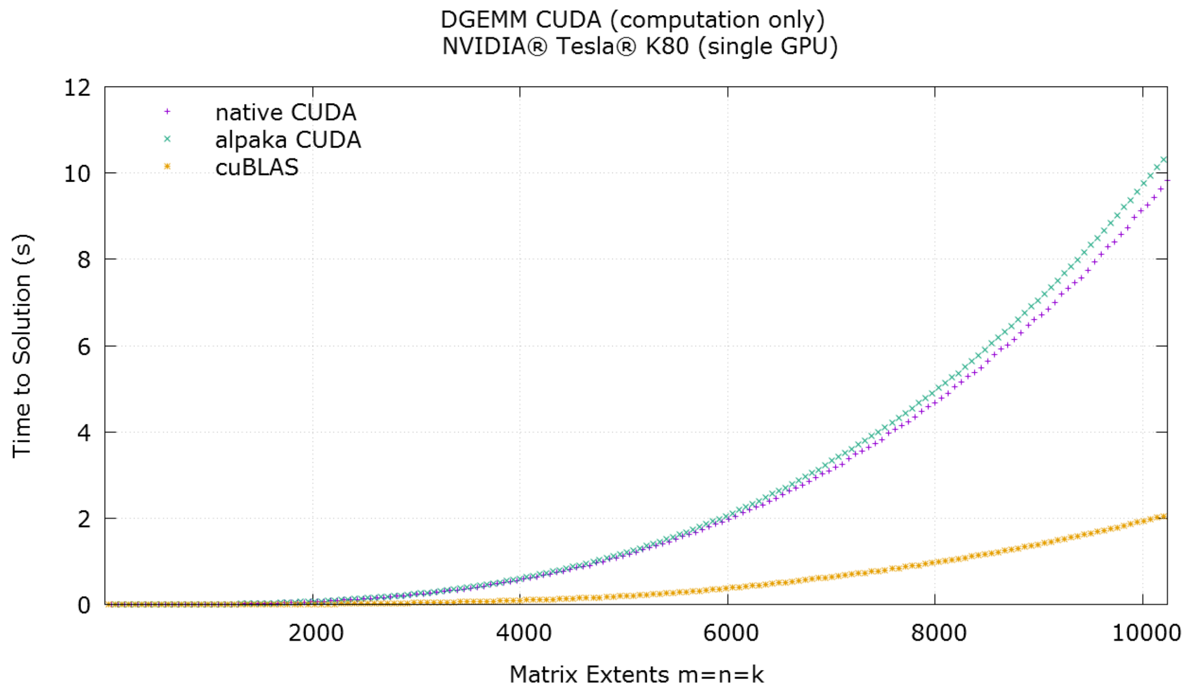

Figure 4.10: Comparison of the time to solution of the *NVIDIA cuBLAS* and the *alpaka CUDA* relative to the native *CUDA* generalized matrix-matrix-multiplication.

size for shared memory usage. Beyond what the basic implementation does it is also unrolling loops combined with prefetching into the thread local register file and reordering instructions to reduce the influence of latencies within the pipeline [35]. All these details lead to less memory loading operations and more calculations per memory load as well as a shift from operations within shared memory to operations within registers.

When directly comparing the execution time of the *alpaka CUDA* with the native *CUDA* version (figure 4.11) a small overhead can be observed. It can be explained by additional registers being used per thread to store the number of elements per thread as well as a minimal number of additional runtime calls the *alpaka CUDA* back-end has to do, to ensure correct behavior in all circumstances. For example, it has to check and set the current *CUDA* device on every function call that leads to the invocation of a device dependent *CUDA* function. This allows interoperability with user code that directly uses *CUDA* which could potentially have changed the device in between two *alpaka* calls.

When directly comparing the PTX device code shown in section A.3 in the appendix the two versions are identical up to a single additional but unused function parameter in the *alpaka* variant, the elements per thread extent variable as well as different internal variable names. This could further be optimized by removing the elements per thread extent variable completely at compile-time if there is exactly one element per thread.

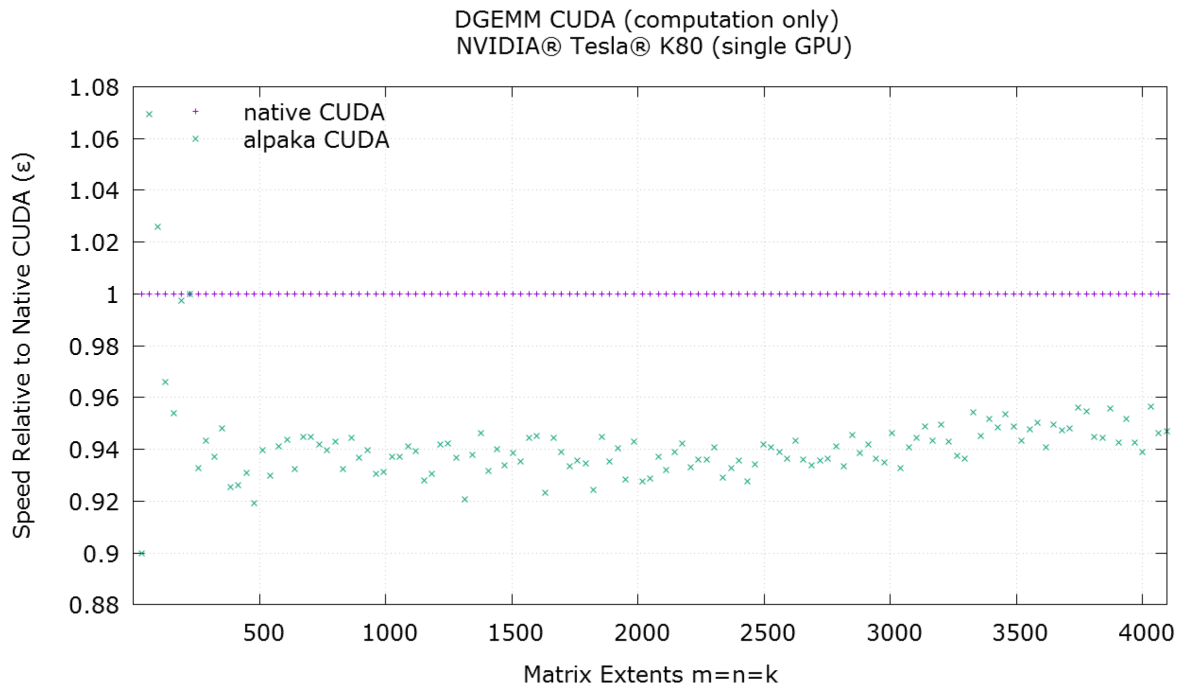

Figure 4.11: Comparison of the *alpaka CUDA* relative to the native *CUDA* generalized matrix-matrix-multiplication.

4.3.2 GEMM on x86 CPUs

On CPUs multiple series of DGEMM measurements will be taken. First of all a comparison between a sequential implementation and an *alpaka* version using the sequential back-end will be carried out. Those measurements are using only one single core. Furthermore a comparison between a native *OpenMP* implementation, and an *alpaka* version using the *OpenMP* back-end will be executed. Shortcomings of the kernel that is directly derived from the *CUDA* version will get revealed during these tests and ways how to overcome the speed problems will get pointed out.

CPU Sequential Execution

The comparison of the single core versions shows that a direct adaption of the *CUDA* matrix multiplication algorithm is not the most efficient version on the CPU. The sequential *alpaka* back-end only reaches up to 18% of the speed of the native sequential ikj blocked version and is even half as slow as the not optimized sequential ijk version (fig. 4.12). The blocked native version switches the j and the k loops to obtain better cache usage.
The *alpaka* version using the same kernel as the one used on the GPU does not optimally map to a CPU. It uses an additional level of blocking over the shared memory as was shown in figure 4.9. With the number of threads per block being one in the sequential case, the whole inner loop over the sub-matrices as well as the caching of the sub-matrix data in the shared memory is superfluous. This loop, the shared memory allocations as well as the two copies are useless and are not optimized away and contribute to over 50% of the instructions within the loop body. By removing this overhead, the relative speed of the CPU back-end versus the optimized native sequential version increases to up to nearly 40% of the optimized

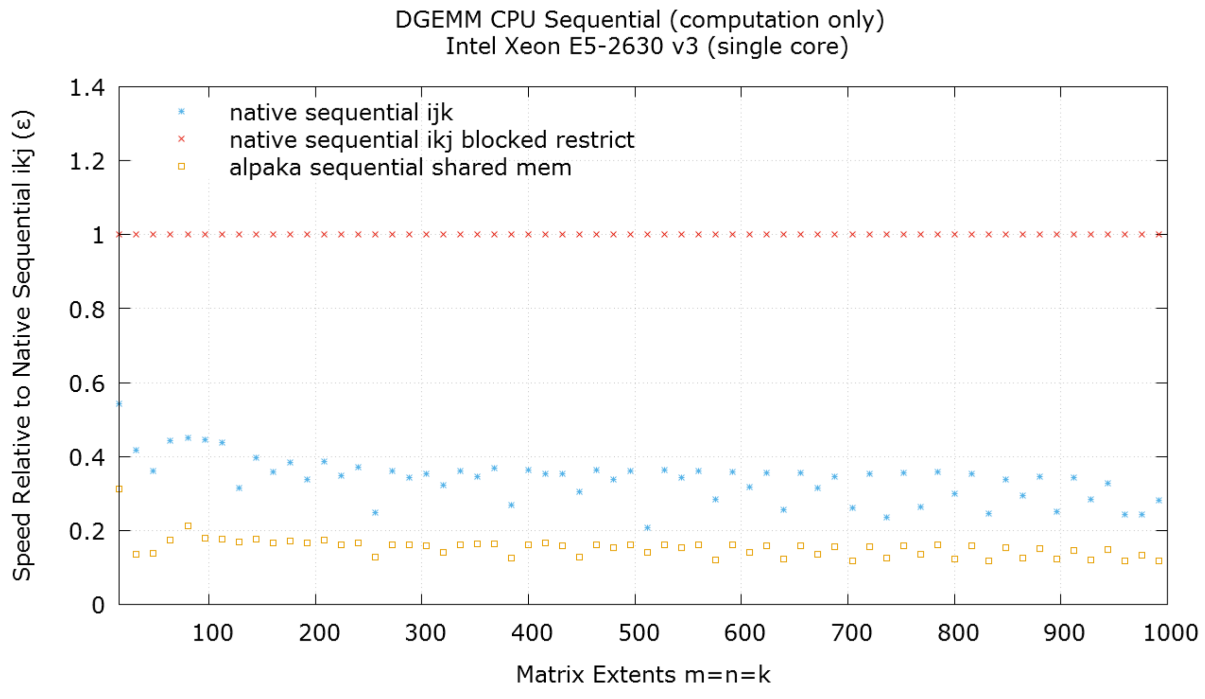

Figure 4.12: Comparison of the *alpaka* sequential relative to the native blocked C++ generalized matrix-matrix-multiplication.

sequential version (fig. 4.13). This shows that the *alpaka* abstraction does not automatically lead to performant ports of an application but also requires to think about the flow of data within the kernel. By additionally using adaptive data structures that are specialized for different accelerators, a single instance of the kernel can be defined that runs optimal on multiple architectures. In this case, a shared memory buffer is required that only caches the data on platforms where this is beneficial and directly forwards accesses to the underlying memory on all other accelerators. Because such data structures do not currently exist, an *alpaka* kernel is used, that dose not use any shared memory at all (fig. A.7).

The *alpaka* version of the algorithm without shared memory copies uses the same execution strategy as the original GPU kernel. The same basic ijk loop order as in the slow native version is used, where $i \times j$ kernels are executed, each calculating one element of the C matrix. This element is calculated by the dot product of a line of A and a column of B. This column-wise iteration over the B matrix stored in row-major order is very cache unfriendly leading to much more cache-misses and a decrease in performance especially when the matrices do not fit into the cache anymore. This is again a problem that can be solved by considering the properties of the underlying data structures. One solution could be to change the loop order, but it is equally correct to change the data structure and to convert the B matrix from a row-major into a column-major matrix. The resulting kernel is shown in figure A.8 in the appendix. Figure 4.14 shows the result of this change to a column-major matrix layout which is equivalent to storing the transposed matrix B in row-major order. The new version reaches nearly 50% of the performance of the native optimized sequential version even when the matrices exceed the kernel size. The rest of the performance difference can be explained most probably by the much higher number of integer index calculations that have to be done per data element. Each element is computed by a single

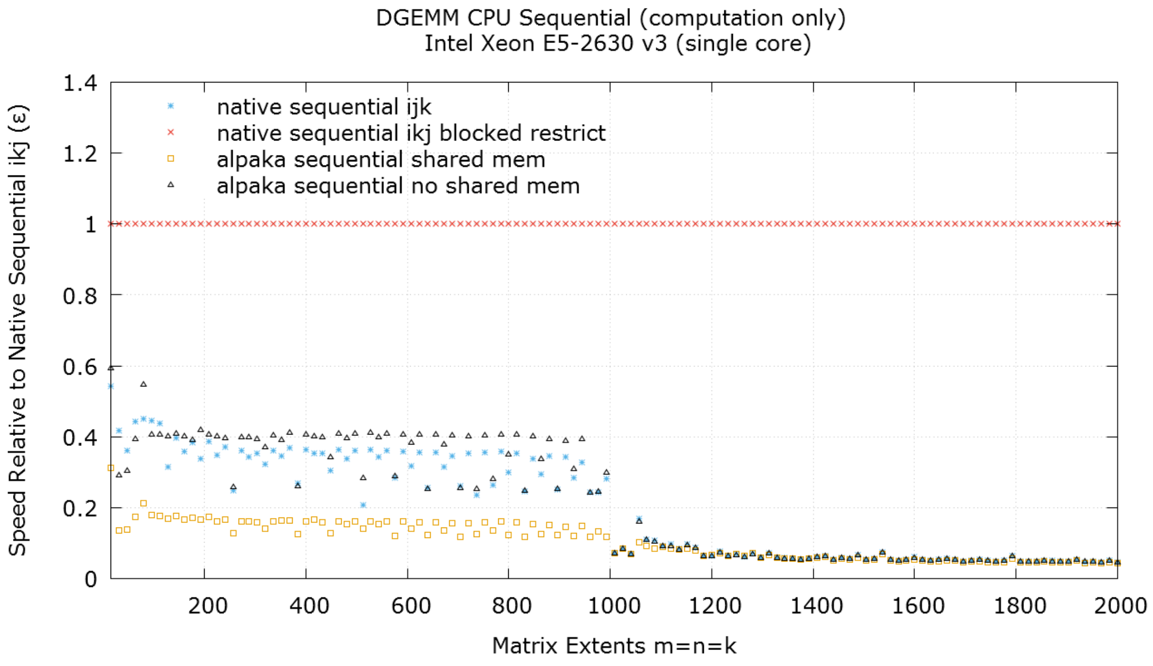

Figure 4.13: Comparison of the *alpaka* sequential kernel without usage of shared memory relative to the native blocked C++ generalized matrix-matrix-multiplication.

independent kernel invocation which has to calculate the entire indices into the A, B and C matrices. In contrast to this, the native version only has to do some basic increments to get to the next element. By utilizing the element level of the *alpaka* abstraction hierarchy and computing multiple elements per thread, this overhead could possibly be reduced.

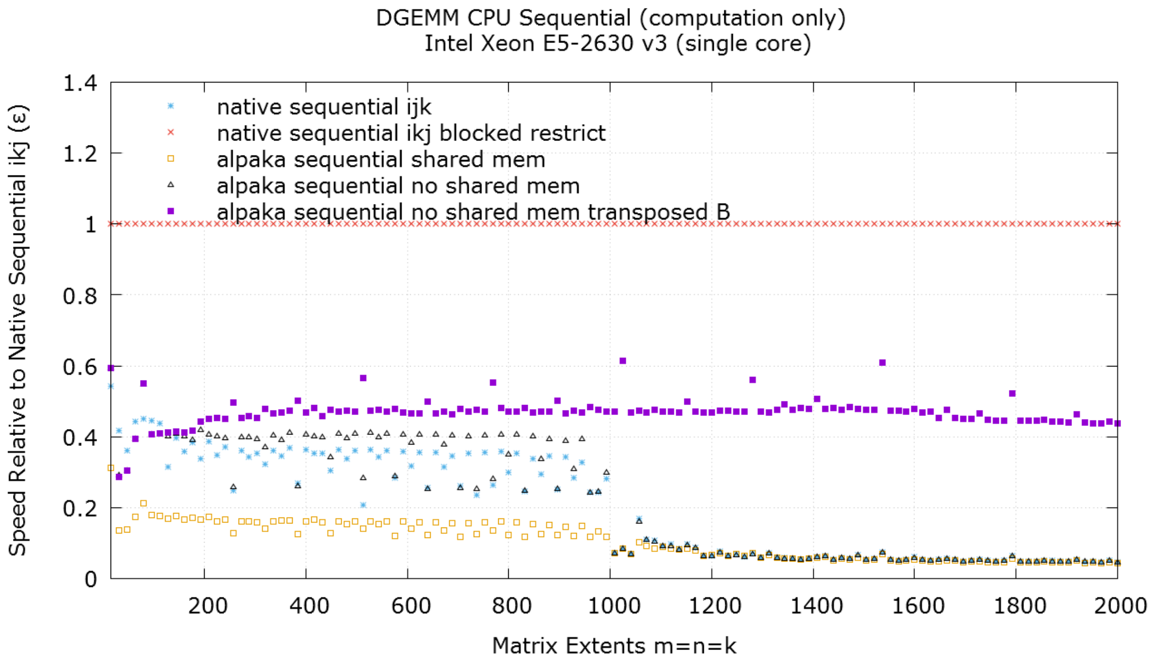

Figure 4.14: Comparison of the *alpaka* sequential kernel without usage of shared memory and a transposed B matrix relative to the native blocked C++ generalized matrix-matrix-multiplication.

CPU Parallel Execution

By parallelizing only the outer loop of the standard algorithm that had been shown in figure 4.8, multiple rows of the result matrix are calculated in parallel. The insertion of only one annotation directly in front of the loop is required to parallelize the algorithm using *OpenMP*. Everything else is automatically taken over by the compiler. By the way of example this is shown in figure 4.15.

```
#pragma omp parallel for
for(size_t i = 0; i < m; ++i){
    for(size_t j = 0; j < n; ++j){
        C[i*ldc + j] = beta * C[i*ldc + j];
    }
    for(size_t k2 = 0; k2 < k; ++k2){
        for(size_t j = 0; j < n; ++j){
            C[i*ldc + j] = alpha * A[i*lda + k] * B[k*ldb + j] + C[i*ldc + j];
        }
    }
}
```

Figure 4.15: C-code of a generalized matrix-matrix-multiplication with *OpenMP* annotations.

In this scheme one processing unit calculates one or multiple rows (m/p) of the result matrix (fig. 4.16) where m is the number of rows and p the number of *OpenMP* threads. The number of threads is limited to $p \leq n$. The corresponding rows of the matrices A and C are required, but always the complete B matrix

Theoretically it should be beneficial to initially distribute the work evenly with `schedule(static)`. Due to inherent differences between the threads, for example, the scheduled start time or the socket they are

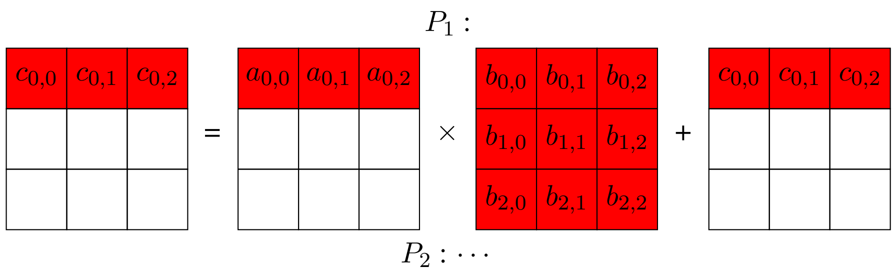
P_1 :
$c_{0,0}$ $c_{0,1}$ $c_{0,2}$
=
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$
×
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$
+
$c_{0,0}$ $c_{0,1}$ $c_{0,2}$
$P_2 : \cdots$

Figure 4.16: Scheme of the line-parallel matrix-matrix-multiplication.

pinned to, this is not correct in practice. If only one thread is slower then the others with a static work assignment, all other threads have to wait for the slower thread at the end. Because the probability of such variations increases with rising time to solution, it is useful for larger matrices to assign the work dynamically into parts of decreasing size with `schedule(guided)`.

Comparing the version of the *alpaka* kernel without shared memory and a transposed B matrix to the native *OpenMP* version (fig. 4.15) shows that the *alpaka* version reaches up to 90% of the native performance (fig. 4.17). This is due to the single core version being compute bound, but the multi-core version being memory bandwidth limited. The additional index calculations are nearly completely hidden by the memory latency.

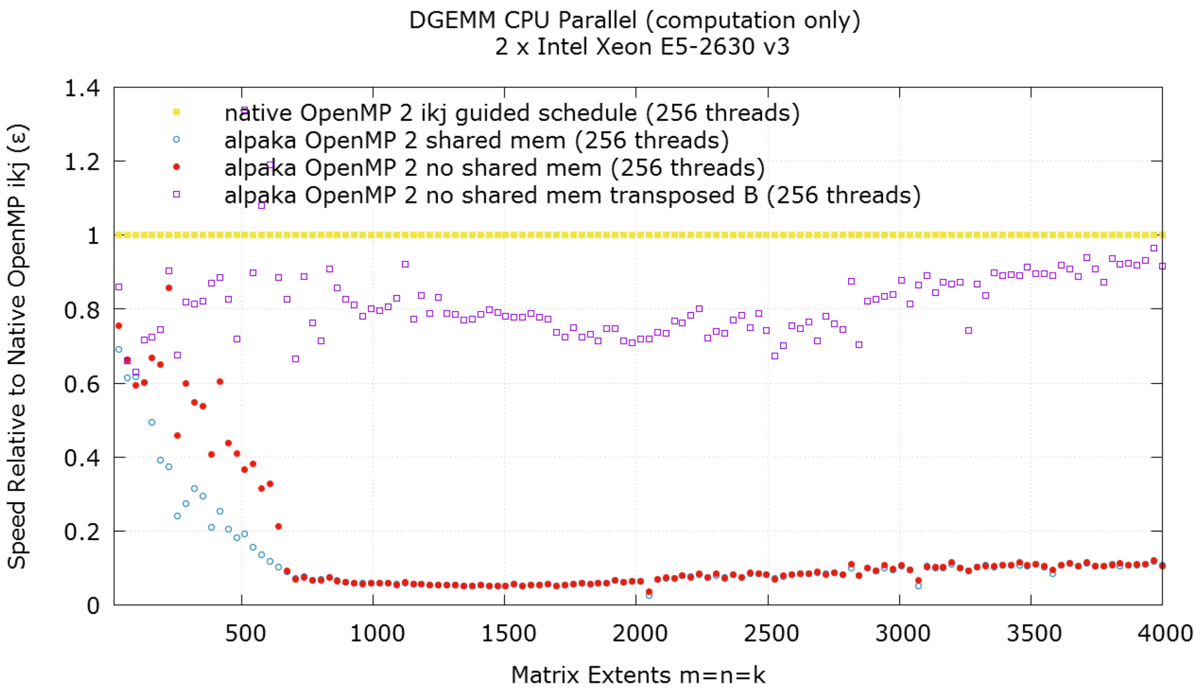

Figure 4.17: Comparison of the *alpaka OpenMP* kernel without usage of shared memory and a transposed B matrix relative to the native *OpenMP* generalized matrix-matrix-multiplication.

4.4 HASEonALPAKA

To complement the results of the artificial tests, the real world simulation code *HASEonGPU*[39] consisting of circa 10,000 lines of code has been ported to use the *alpaka* library instead of directly using *CUDA*. *HASEonGPU* implements an adaptive Monte Carlo algorithm for computing the amplified spontaneous emission (ASE) flux in laser gain media pumped by pulsed lasers. The project is published under the GPLv3 open-source license. It achieves excellent speedups by utilizing *MPI* to scale across whole clusters combined with *CUDA* for a highly data parallel computations. Porting *HASEonGPU* to *alpaka* has been carried out by an independent developer and required circa 3 weeks of time.

After the porting has been finished, *HASEonGPU* has successfully been executed on GPU and CPU clusters. Figure 4.18 compares the relative speed of an *HASEonGPU* computation executed with identical parameters on different systems. The original native *CUDA* version is used as the base case for comparison. The *alpaka* version using the *CUDA* back-end running on the same *NVIDIA* K20 cluster as the native version does not show any overhead at all leading to identical execution times. On the *Intel* and *AMD* CPU clusters the *OpenMP 2* accelerator back-end without support for the not required thread level parallelism is used. Each block contains exactly one thread computing multiple elements. This perfectly maps to the CPUs capabilities for independent vectorized parallelism and leads to very good results. The nearly doubled time to solution on both, the *Intel* and *AMD* CPU clusters, is on par with the halved double precision peak performance of those systems relative to the *NVIDIA* K20 used as the reference.

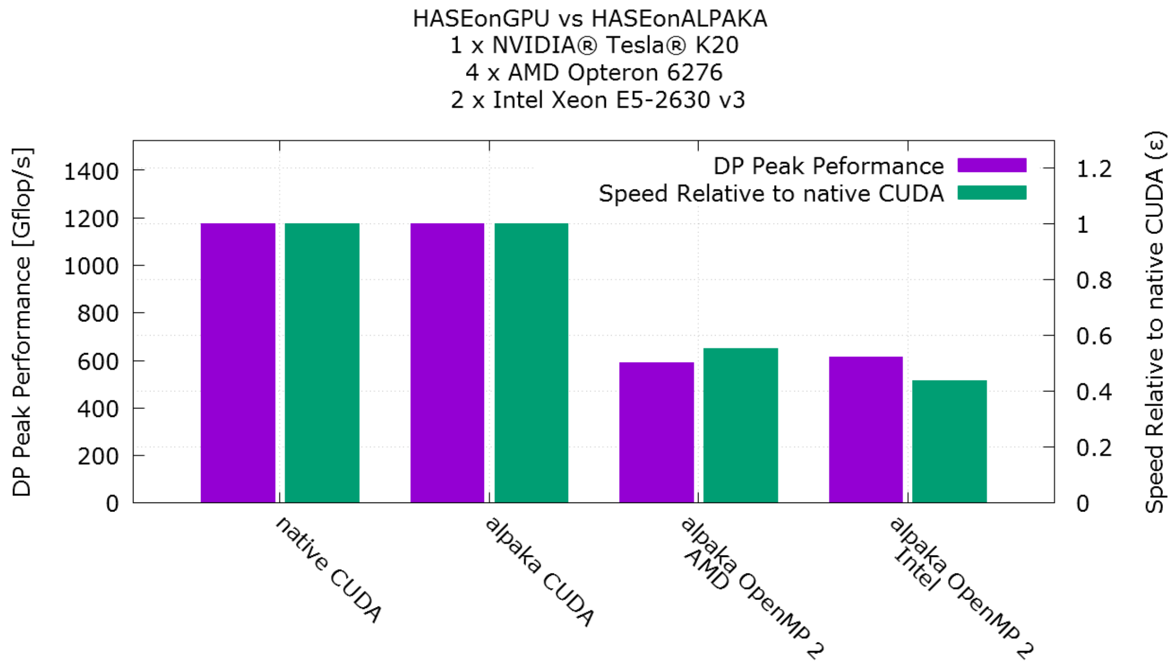

Figure 4.18: Comparison of the native *CUDA HASEonGPU* implementation relative to the *alpaka* port using the *CUDA* and the *OpenMP 2* accelerator back-ends. The measurement used 4209 sample points and 1 million rays per sample point.

5 Summary and Outlook

5.1 Summary

In this work has been shown that it is possible to define and implement an abstract (chapter 2) interface (section 3.2) that allows to facilitate parallelism on all levels available in modern hardware (section 3.1). By implementing a proof-of-concept library it was possible to show that the portability problems of current codes can be solved sustainably. Heterogeneous accelerators can be used within a single application using one uniform way to describe the algorithms for all accelerators back-ends. Additionally it allows to implement heterogeneous work balancing queues that can run a kernel on any available device adaptively. The maintainability burden is reduced immensely by only implementing one version of each kernel for all platforms reducing the possibility of bugs and incompatibilities. The portability of the kernels is guaranteed by the library implementation and can be thoroughly tested in one place rather then for each algorithm in each project. By giving the kernel compile-time knowledge about the accelerator it is running on, users as well as libraries can optimize their implementations through abstraction and compile time template meta-programming. The *alpaka* library offers interoperability with all the underlying APIs (*CUDA*, *OpenMP*, Fibers, etc.) and direct access to native pointers and handles. Users can always extend the library with missing special functionality for their accelerator by utilizing the functional trait specialization design or they can even use the native underlying API where appropriate.
In chapter 4 has been shown that there is almost no overhead over the native APIs and that by utilizing the additional element level, kernels can be written in a performance portable way across GPUs, CPUs and other accelerators. It had been possible to write a single DAXPY and DGEMM kernel that runs nearly as fast on CPUs as it does on GPUs. The resulting *CUDA* PTX code is almost identical to that of the native *CUDA* equivalents. This shows the advantages of the compile nature of *alpaka* library. However, it also became evident, that the choice of the correct data structure as well as an access pattern that is supported by the executing hardware is very important. The *alpaka* abstraction library is only able to portably deliver performance when the platform differences with respect to the memory hierarchy are taken into account. However, this is also true for all other current acceleration libraries.
The extremely rapid and successful porting of the *HASEonGPU* simulation demonstrates the applicability and functionality in real world scenarios. A performance portable port was possible within 3 weeks without having to duplicate or rewrite major parts of the simulation. This resulted in simulation times scaling nearly ideally (relative to the native version) across CPUs and GPUs when considering the platform‘s theoretical peak performance.

5.2 Outlook

Porting PIConGPU The porting of the *PIConGPU* simulation that lead to the development of the *alpaka* abstraction hierarchy and library is currently in progress. On this basis, optimizations and implementations for other accelerator devices, especially the *Intel* Xeon Phi, are planned.

Kernel as Chain of Unsynchronized Functions An abstract kernel interface for heterogeneous devices is a huge step. Nevertheless, there is clearly more to do. The way a kernel currently is defined does not provide full freedom and all possibilities to optimally adapt to all existing and future hardware. Without full control over the kernel code and the ability to transform it in a way *OpenCL* [7] or the *PGI* compiler can do, it is the task of the user to enable vectorization and utilize CPUs as much as possible. However, we do not want to lose the ability to programatically specialize or replace every part of the acceleration hierarchy to a compiler. Therefore, a way to enhance the abstraction defined in this thesis is to further restrict the possibilities of kernel function objects and to expand the requirements of the kernel concept.

A way to automatically improve especially the performance of CPU back-ends is to allow a way of code transformation within the kernels via expression templates or other techniques. By representing a kernel as a chain of function objects (possibly a special directed acyclic graph for nested kernels/functions), where the function objects at the nodes can be executed in parallel and transitions between nodes represent synchronization points, the adaption to hardware capabilities could be enhanced. The current kernels would have to be partitioned into unsynchronized, parallel parts by splitting at the synchronization statements as can be seen in the upper row of figure 5.1.

When using *CUDA*, the synchronization mechanisms of the GPU could be used and the chain could be executed by a single kernel, interleaving the parallel parts (calls to the node‘s function objects) with synchronization calls. This would not introduce any overhead for the *CUDA* back-end.

On the other hand this could allow to remove the necessity of synchronization between threads within a block for CPU back-ends not providing fast synchronization. The chain‘s nodes would be executed sequentially as independent kernels, where the nodes themselves would be executed in parallel by the back-end. This would save the overhead of synchronization mechanisms because the synchronization is implicit in this model.

By removing the possibility of synchronization within a kernel function object itself and making it part of the interface, further optimizations could be implemented. For example, threads within a block would not necessarily have to be executed in parallel. A single kernel-thread could coalesce multiple threads and execute them sequentially. This would allow the compiler to optimize better and utilize vectorization across block threads without the user explicitly writing out a loop at all.

Description of Distributed Data Structures To achieve full performance portability across multiple accelerator architectures it is not only necessary to abstract the parallelism portably, but also to abstract performant and distributed data structures. The *alpaka* library by purpose is agnostic to the underlying data structures but as has been shown in multiple places in this work, different accelerators require different data structures for a performant execution. Whether column-major vs. row-major, caching in GPU shared memory vs. implicitly or explicitly within CPU caches as well as data layout in a structure of arrays vs. an array of structures and many more data structure details are depending on the underlying accelerator specifics. These differences have to be hidden behind an interface. The application developer should not have to know all these intricacies but simply use the wrapper library which automatically uses the best data structure depending on the accelerator back-end.

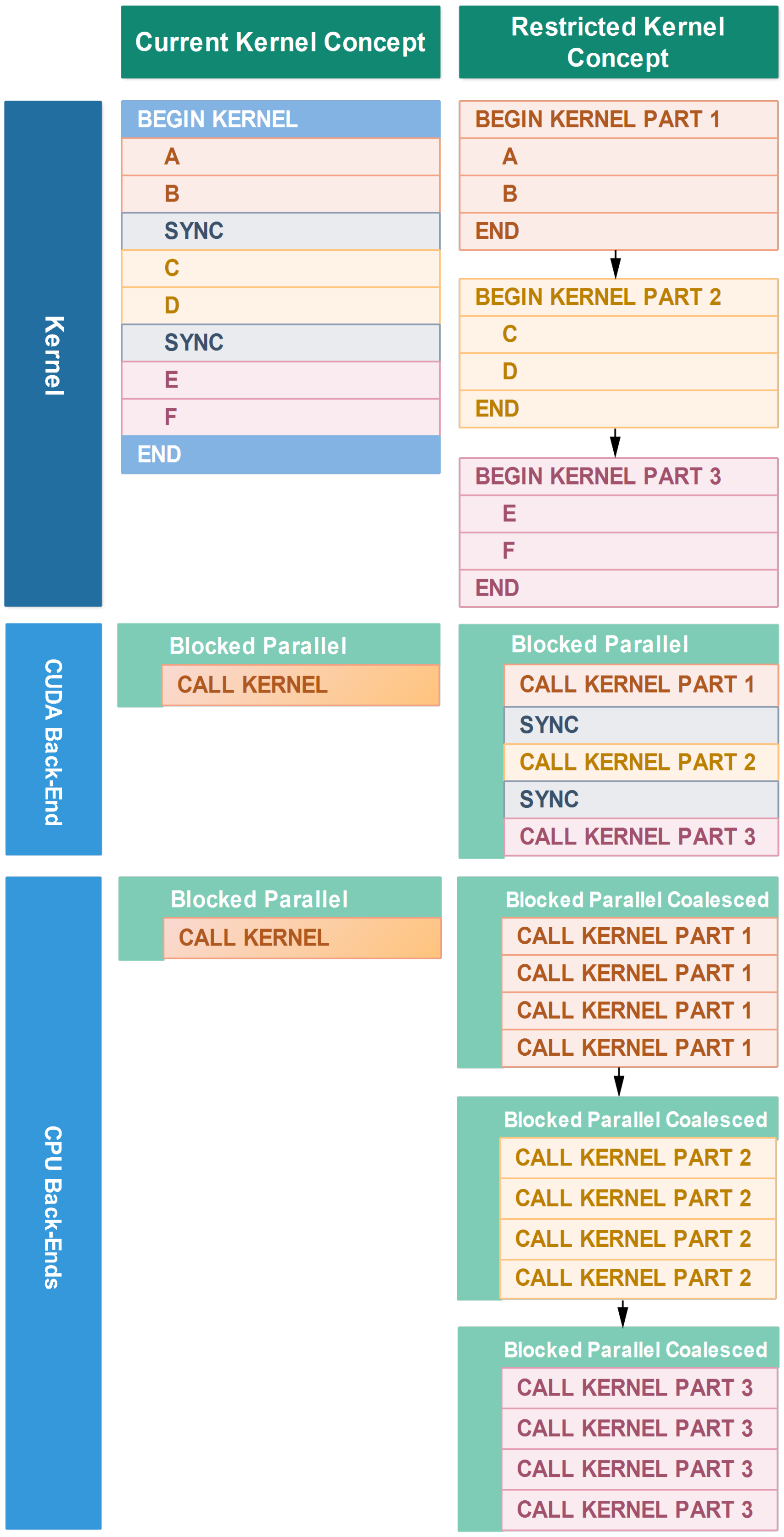

Figure 5.1: Comparison of the current and a possible extended kernel concept and their usages.

Concurrent Usage of Multiple Devices for a Single Kernel To extend the functionality of the library even more, the concurrent usage of multiple, possibly heterogeneous devices for acceleration of a single kernel within a node could be implemented. Multiple GPUs could be working together on an otherwise too memory limited task (shared, local or global memory). This could also fully utilize GPUs like the *NVIDIA* K80, which combines two K40 GPUs on one board sharing the PCI connection, because currently a K80 GPU can only ever be used as two independent K40 GPUs. This task is very hard because the library would have to know latencies and other timings to estimate the win or loss of utilizing a combination of possibly non-uniform devices. Another problem is the unrestricted relation between the thread index and the locations of global memory the results are written to. Duplicating the input data across all collaborating devices is one easy, but not memory sparing way to provide the kernel with input data. However, the procedure of merging the buffers the result data have been written to, to produce a single result buffer, is undefined. Maybe this could be achieved by further extending the kernel concept.

Partitioning of Devices Not only combining multiple heterogeneous devices is worth looking into, but also partitioning devices is a field that should be evaluated. Therefore, the current `Device` concept should be extended by a `DeviceView` concept. For example, a view of a device could select a subset of CPU cores and memory. The current `Device` concept would always model the `DeviceView` concept by permanently using all of its cores and memory. With a `DeviceView` it would be possible to partition for instance a eight core CPU into six cores for one `DeviceView` and two cores for another `DeviceView`. Then kernels executed in the corresponding streams will only be able to use the cores they are allowed to. This would prevent independent streams from mutually influencing each other. Furthermore, this would allow to guarantee selected compute resources for time critical tasks.

Support for More Accelerators and APIs Another thing that would greatly enhance the usefulness of the *alpaka* library would be the support of more accelerators and APIs. Especially support for offloading computations to the *Intel Xeon Phi* would be desirable. This would most probably be implemented with *OpenMP 4.1*[30] because it introduces the methods `omp_target_alloc` and `omp_target_free` as well as the target clauses `omp target data use_deviceptr(p)` and `omp target is_deviceptr(p)` which should make it easy to implement the abstraction defined by the *alpaka* library. Further on, APIs such as *Intel TBB*[9], *Intel Cilk Plus*[31], *C++AMP*[21] or *OpenACC*[29] should be evaluated for their usefulness and support for even more types of accelerators. All of this can be done independently by users without having to change any line of code of the existing *alpaka* library due to its modular, non-intrusive, extensible design. Those extensions can be added to the library later, resulting in benefit for all current users without having to change anything.

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series).* Addison-Wesley Professional, 2004. ISBN 0321227255.

[2] Advanced Micro Devices, Inc. Amd opteron processor solutions : Amd opteron^{TM} 6276. `http://products.amd.com/pages/OpteronCPUDetail.aspx?id=759`. [Online; accessed Jul 21, 2015].

[3] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18(1):1–33, January 2010. `http://dx.doi.org/10.1155/2010/540159`.

[4] M. Bussmann, H. Burau, T. E. Cowan, A. Debus, A. Huebl, G. Juckeland, T. Kluge, W. E. Nagel, R. Pausch, F. Schmitt, U. Schramm, J. Schuchart, and R. Widera. Radiative signatures of the relativistic kelvin-helmholtz instability. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 5:1–5:12, New York, NY, USA, 2013. ACM. `http://dx.doi.org/10.1145/2503210.2504564`.

[5] M. Bussmann and G. Juckeland. Pushing Plasma Simulations Towards Exascale Performance with PIConGPU, March 2013. GTC 2013, San Jose (CA), USA.

[6] Helmholtz-Zentrum Dresden Rossendorf. High Performance Computing at HZDR. `https://www.hzdr.de/db/Cms?pNid=1615`. [Online; accessed May 21, 2015].

[7] Dafei Huang, Mei Wen, Changqing Xun, Dong Chen, Xing Cai, Yuran Qiao, Nan Wu, and Chunyuan Zhang. Automated Transformation of GPU-Specific OpenCL Kernels Targeting Performance Portability on Multi-Core/Many-Core CPUs. In *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, pages 210–221, 2014. `http://dx.doi.org/10.1007/978-3-319-09873-9_18`.

[8] Axel Huebl. Injection Control for Electrons in Laser-Driven Plasma Wakes on the Femtosecond Time Scale. Technische Universität Dresden, August 2014. Diploma thesis for the german degree "Diplom-Physiker". `http://dx.doi.org/10.5281/zenodo.15924`.

[9] Intel Corporation. Intel Threading Building Blocks. `https://www.threadingbuildingblocks.org/`. [Online; accessed September 28, 2015].

[10] Intel Corporation. Intel xeon processor e5-2630 v3. `http://ark.intel.com/de/products/83356/Intel-Xeon-Processor-E5-2630-v3-20M-Cache-2_40-GHz`. [Online; accessed Jul 21, 2015].

[11] Intel Corporation. *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*, September 2012. `https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf` [Online; accessed Aug 15, 2015].

[12] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, October 2014. `https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf` [Online; accessed Aug 15, 2015].

[13] Intel Corporation. Intel Math Kernel Library Reference Manual. `https://software.intel.com/sites/default/files/managed/9d/c8/mklman.pdf`, 2015. [Online; accessed May 20, 2015].

[14] ISO/IEC JTC1 SC22 WG21. *Programming Languages — C++*. International Organization for Standardization (ISO), Genevan, Switzerland, 1st edition, February 2012. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`.

[15] ISO/IEC JTC1 SC22 WG21. Programming Languages — C++ Extensions for Concepts. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4377.pdf`, February 2015. [Online; accessed June 17, 2015].

[16] N.S. Jayasena, M.J. Schulte, G.H. Loh, and M. Ignatowski. Die-stacked memory device with reconfigurable logic, June 2015. US Patent App. 14/551,147.

[17] Ralf Karrenberg. *Automatic Packetization*. PhD thesis, Saarland University, July 2009.

[18] Ralf Karrenberg and Sebastian Hack. Whole Function Vectorization. In *International Symposium on Code Generation and Optimization*, CGO, 2011.

[19] Ralf Karrenberg and Sebastian Hack. Improving Performance of OpenCL on CPUs. In *Compiler Construction*, 2012. `http://dx.doi.org/10.1007/978-3-642-28652-0_1`.

[20] Martin Thompson. False Sharing. `http://1.bp.blogspot.com/-MYso5dR5ItE/TjOy1Mbr3NI/AAAAAAAAAAk/uLPjSeGhbmg/s1600/cache-line.png`. [Online; accessed September 27, 2015].

[21] Microsoft Corporation. C++AMP : Language and Programming Model. `http://download.microsoft.com/download/2/2/9/22972859-15C2-4D96-97AE-93344241D56C/CppAMPOpenSpecificationV12.pdf`, October 2013. Version 1.2 [Online; accessed July 24, 2015].

[22] NVIDIA Corporation. Grid of Thread Blocks. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/grid-of-thread-blocks.png`. [Online; accessed Aug 04, 2015].

[23] NVIDIA Corporation. Matrix Multiplication with Shared Memory. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/graphics/matrix-multiplication-with-shared-memory.png`. [Online; accessed May 20, 2015].

[24] NVIDIA Corporation. Parallel Thread Execution ISA. http://docs.nvidia.com/cuda/pdf/ptx_isa_3.2.pdf, July 2013. Version 3.2 [Online; accessed September 03, 2015].

[25] NVIDIA Corporation. Tesla Kepler Family Datasheet. http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf, October 2013. [Online; accessed May 19, 2015].

[26] NVIDIA Corporation. Tesla K80 Datasheet. http://international.download.nvidia.com/pdf/kepler/TeslaK80-datasheet.pdf, October 2014. [Online; accessed May 19, 2015].

[27] NVIDIA Corporation. cuBLAS Library. http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf, March 2015. [Online; accessed May 20, 2015].

[28] NVIDIA Corporation. NVIDIA CUDA C Programming Guide Version 7.0. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, March 2015. [Online; accessed May 20, 2015].

[29] OpenACC-standard.org. The OpenACC Application Programming Interface. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf, August 2013. Version 2.0a [Online; accessed July 24, 2015].

[30] OpenMP Architecture Review Board. OpenMP Application and Programming Interface. http://openmp.org/mp-documents/OpenMP4.1_Comment_Draft.pdf, July 2015. Version 4.1 rev4 [Online; accessed July 24, 2015].

[31] Arch D. Robison. Composable parallel patterns with intel cilk plus. *Computing in Science and Engg.*, 15(2):66–71, March 2013. http://dx.doi.org/10.1109/MCSE.2013.21.

[32] K. Rocki, M. Burtscher, and R. Suda. The Future of Accelerator Programming: Abstraction, Performance or Can We Have Both? In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 442–443, Dec 2013. http://dx.doi.org/10.1109/ICPADS.2013.76.

[33] Jeremy Siek, Andrew Lumsdaine, and David Abrahams. The Boost Concept Check Library (BCCL). http://www.boost.org/doc/libs/1_58_0/libs/concept_check/concept_check.htm. [Online; accessed July 20, 2015].

[34] Herb Sutter. Sutter's Mill: Why Not Specialize Function Templates? *C/C++ Users Journal*, 19(7):65ff, July 2001.

[35] V. Volkov and J.W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, Nov 2008. http://dx.doi.org/10.1109/SC.2008.5214359.

[36] Benjamin Worpitz. matmul - Generalized Matrix-Matrix-Multiplication Library with C Interface. https://github.com/BenjaminW3/matmul, 2013-2015. [Online; accessed July 24, 2015].

[37] Benjamin Worpitz. alpaka - Abstraction Library for Parallel Kernel Acceleration. `https://github.com/ComputationalRadiationPhysics/alpaka`, 2014-2015. [Online; accessed July 24, 2015].

[38] Benjamin Worpitz. vecadd - Generalized Vector Addition Library with C Interface. `https://github.com/BenjaminW3/vecadd`, 2015. [Online; accessed August 13, 2015].

[39] E. Zenker, C. Eckert, D. Albach, and M. Bussmann. HASEonGPU - High performance Amplified Spontaneous Emission on GPU. `http://dx.doi.org/10.5281/zenodo.13964`, 2015. [Online; accessed September 26, 2015].

List of Figures

List of Tables

A Code Listings

A.1 Generalized Vector Addition

A.1.1 Direct Comparison of SAXPY Host and Kernel Codes

```
#include <cuda_runtime.h>



__global__ void axpyKernel(

    const float alpha, const float *X,
        float *Y,
    int numElements)
{
    int i = blockDim.x * blockIdx.x +
        threadIdx.x;










    if (i < numElements)
    {











        Y[i] = alpha * X[i] + Y[i];
    }
}


int main()
{
    // CUDA is always 3 dimensional.

    // CUDA always uses size_t.

    // CUDA is the only possible
        accelerator back-end.


    // The CUDA default stream is
        synchronous, it synchronizes with
        all other streams and the host
        thread implicitly.
```

```
#include <alpaka/alpaka.hpp>
struct AxpyKernel
{
    template<typename TAcc>
    ALPAKA_FN_ACC void operator()(
        TAcc const & acc,
        const double alpha, const double
            * X, double * Y,
        int numElements) const
    {
        auto const gridThreadIdx(alpaka::
            idx::getIdx<alpaka::Grid,
            alpaka::Threads>(acc)[0u]);
        auto const threadElemExtent(
            alpaka::workdiv::getWorkDiv<
            alpaka::Thread, alpaka::Elems
            >(acc)[0u]);
        auto const threadFirstElemIdx(
            gridThreadIdx *
            threadElemExtent);
        if(threadFirstElemIdx < n)
        {
            auto const elems(
                threadElemExtent + alpaka
                ::math::min(acc, 0, n-(
                threadFirstElemIdx+
                threadElemExtent)));
            for(TSize i(
                threadFirstElemIdx); i<(
                threadFirstElemIdx+elems);
                 ++i)
            {
                Y[i] = alpha * X[i]+Y[i];
            }
        }
    }
};
int main()
{
    // The dimensionality of the task.
    using Dim = alpaka::dim::DimInt<1u>;
    // Define the type used for sizes.
    using Size = std::size_t;
    // Select the accelerator type to
        execute on.
    using Acc = alpaka::acc::AccGpuCuda<
        Dim, Size>;
    // Select the stream type.
    using Stream = alpaka::stream::
        StreamCpuAsync;

    // Get the host device this thread is
         running on.
```

```
// CUDA implicitly selects a default
    device.


// Use the cuda default stream.



int numElements = 50000;
size_t size = numElements * sizeof(
    double);

// Allocate the host vectors
double * h_X = (double *)malloc(size)
    ;

double * h_Y = (double *)malloc(size)
    ;


// Initialize the input
double const alpha(rand()/(double)
    RAND_MAX);
for (int i = 0; i < numElements; ++i)
{
    h_X[i] = rand()/(double)RAND_MAX;

    h_Y[i] = rand()/(double)RAND_MAX;
}




// Allocate the device vectors
double * d_X, * d_Y;
cudaMalloc((void **)&d_X, size);

cudaMalloc((void **)&d_Y, size);


// Copy X and Y from host to device
    memory
cudaMemcpy(d_X, h_X, size,
    cudaMemcpyHostToDevice);
cudaMemcpy(d_Y, h_Y, size,
    cudaMemcpyHostToDevice);

// Set the block and grid sizes.
int threadsPerBlock = 256;
int blocksPerGrid =(numElements +
    threadsPerBlock - 1) /
    threadsPerBlock;





// Launch the Vector Add CUDA Kernel
axpyKernel<<<
    blocksPerGrid, threadsPerBlock
        >>>(
    alpha,
    d_X,
    d_Y,
    numElements);
```

```
auto devHost(alpaka::dev::cpu::getDev
    ());

// Select a device to execute on.
auto devAcc(alpaka::dev::DevManT<Acc
    >::getDevByIdx(0));

// Create a stream on the accelerator
     device.
Stream stream(devAcc);

int numElements = 50000;



// Allocate the host vectors
auto h_X(alpaka::mem::buf::alloc<
    double,Size>(devHost,numElements))
    ;
auto h_Y(alpaka::mem::buf::alloc<
    double,Size>(devHost,numElements))
    ;

// Initialize the input
double const alpha(rand()/(double)
    RAND_MAX);
for (int i = 0; i < numElements; ++i)
{
    alpaka::mem::view::getPtrNative(
        h_X)[i] = rand()/(double)
        RAND_MAX;
    alpaka::mem::view::getPtrNative(
        h_Y)[i] = rand()/(double)
        RAND_MAX;
}

// Allocate the device vectors

auto d_X(alpaka::mem::buf::alloc<
    double,Size>(devAcc,numElements));
auto d_Y(alpaka::mem::buf::alloc<
    double,Size>(devAcc,numElements));


// Copy X and Y from host to device
    memory
alpaka::mem::view::copy(d_X, h_X,
    numElements);
alpaka::mem::view::copy(d_Y, h_Y,
    numElements);

// Set the block and grid sizes.
auto const workDiv(alpaka::workdiv::
    getValidWorkDiv<Acc>(devAcc,
    numElements, 1, false));


// Crete an instance of the kernel
    functor
AxpyKernel axpyKernel;


// Create the kernel executor.
auto const exec(alpaka::exec::create<
    Acc>(
    workDiv,
    axpyKernel,
    alpha,
    mem::getNativePtr(d_X),
    mem::getNativePtr(d_Y),
    numElements));

// Enqueue the execution task into
```

```
    // Copy Y from device to host memory.
    cudaMemcpy(h_Y, d_Y, size,
        cudaMemcpyDeviceToHost);

    // Free the memory
    cudaFree(d_X); cudaFree(d_Y);
    free(h_X); free(h_Y);

    return EXIT_SUCCESS;
}
```

```
        the stream.
    alpaka::stream::enqueue(stream, exec)
        ;

    // Copy Y from device to host memory.
    alpaka::mem::view::copy(h_Y, d_Y,
        numElements);

    // Memory is automatically freed when
         going out of scope.

    return EXIT_SUCCESS;
}
```

Figure A.1: Direct comparison of SAXPY host and kernel codes. Left: native *CUDA*, Right: *alpaka*

A.1.2 Direct comparison of assembler code snippets generated for sequential DAXPY implementations

```
    call    getTimeSec
    testl   %ebx, %ebx
    movsd   %xmm0, (%rsp)
    movsd   8(%rsp), %xmm2
    jle .L10
    movq    %r12, %rax
    salq    $60, %rax
    shrq    $63, %rax
    cmpl    %ebx, %eax
    cmova   %ebx, %eax
    cmpl    $3, %ebx
    jg  .L23
    movl    %ebx, %eax
.L11:
    movsd   0(%rbp), %xmm0
    cmpl    $1, %eax
    movl    $1, %edx
    mulsd   %xmm2, %xmm0
    addsd   (%r12), %xmm0
    movsd   %xmm0, (%r12)
    je  .L5
    movsd   8(%rbp), %xmm0
    cmpl    $3, %eax
    movb    $2, %dl
    mulsd   %xmm2, %xmm0
    addsd   8(%r12), %xmm0
    movsd   %xmm0, 8(%r12)
    jne .L5
    movsd   16(%rbp), %xmm0
    movb    $3, %dl
    mulsd   %xmm2, %xmm0
    addsd   16(%r12), %xmm0
    movsd   %xmm0, 16(%r12)
.L5:
    cmpl    %eax, %ebx
    je  .L10
.L4:
    subl    %eax, %ebx
    movl    %eax, %ecx
    leal    -2(%rbx), %r8d
    shrl    %r8d
    addl    $1, %r8d
    cmpl    $1, %ebx
    leal    (%r8,%r8), %r9d
    je  .L7
```

```
    call    getTimeSec
    movq    112(%rsp), %rax
    movq    152(%rsp), %rdx
    movsd   %xmm0, 40(%rsp)
    movq    128(%rsp), %rsi
    movq    $0, 264(%rsp)
    movq    $0, 272(%rsp)
    movq    %rax, %rdi
    movq    %rax, 208(%rsp)
    movq    120(%rsp), %rax
    movq    %rdx, 248(%rsp)
    xorl    %edx, %edx
    testl   %edi, %edi
    movq    %rsi, 224(%rsp)
    movq    $0, 280(%rsp)
    movq    %rax, 216(%rsp)
    movq    136(%rsp), %rax
    movl    $0, 288(%rsp)
    movq    $0, 304(%rsp)
    movq    %rax, 232(%rsp)
    movq    144(%rsp), %rax
    movq    %rax, 240(%rsp)
    leaq    288(%rsp), %rax
    movq    %rax, 256(%rsp)
    jle .L1021
    .p2align 4,,10
    .p2align 3
.L1020:
    movl    %esi, %eax
    movl    %edx, 288(%rsp)
    imull   %edx, %eax
    cmpl    8(%rsp), %eax
    jge .L1019
    movq    24(%rsp), %r8
    movsd   16(%rsp), %xmm0
    cltq
    movq    32(%rsp), %rcx
    mulsd   (%r8,%rax,8), %
        xmm0
    leaq    (%rcx,%rax,8),
        %rcx
    addsd   (%rcx), %xmm0
    movsd   %xmm0, (%rcx)
.L1019:
    addl    $1, %edx
```

```
    call    getTimeSec
    movq    112(%rsp), %rax
    movq    128(%rsp), %rbp
    movsd   %xmm0, 24(%rsp)
    movq    144(%rsp), %r10
    movq    $0, 264(%rsp)
    movq    $0, 272(%rsp)
    movq    %rax, %r12
    movq    %rax, 208(%rsp)
    movq    120(%rsp), %rax
    testl   %r12d, %r12d
    movq    %rbp, 224(%rsp)
    movq    %r10, 240(%rsp)
    movq    $0, 280(%rsp)
    movl    $0, 288(%rsp)
    movq    %rax, 216(%rsp)
    movq    136(%rsp), %rax
    movq    $0, 304(%rsp)
    movq    %rax, 232(%rsp)
    movq    152(%rsp), %rax
    movq    %rax, 248(%rsp)
    leaq    288(%rsp), %rax
    movq    %rax, 256(%rsp)
    jle .L1130
    movsd   (%rsp), %xmm1
    movq    %r15, 32(%rsp)
    xorl    %r8d, %r8d
    movl    20(%rsp), %r11d
    movq    40(%rsp), %r15
    xorl    %r13d, %r13d
    unpcklpd    %xmm1, %xmm
        1
    movq    48(%rsp), %rbx
    movq    %r14, 56(%rsp)
    .p2align 4,,10
    .p2align 3
.L1129:
    movl    %ebp, %edx
    movl    %r8d, 288(%rsp)
    imull   %r8d, %edx
    imull   %r10d, %edx
    cmpl    %r11d, %edx
    jge .L1131
    leal    (%rdx,%r10), %
        eax
    movapd  %xmm2, %xmm0
    salq    $3, %rcx
    xorl    %eax, %eax
    leaq    0(%rbp,%rcx), %
        rsi
    xorl    %edi, %edi
    addq    %r12, %rcx
    unpcklpd    %xmm0, %xmm
        0
.L8:
    movupd  (%rsi,%rax), %
        xmm1
    addl    $1, %edi
    mulpd   %xmm0, %xmm1
    addpd   (%rcx,%rax), %
        xmm1
    movaps  %xmm1, (%rcx,%
        rax)
    addq    $16, %rax
    cmpl    %edi, %r8d
    ja  .L8
    addl    %r9d, %edx
    cmpl    %r9d, %ebx
    je  .L10
.L7:
    movslq  %edx, %rdx
    mulsd   0(%rbp,%rdx,8),
         %xmm2
    leaq    (%r12,%rdx,8),
        %rax
    addsd   (%rax), %xmm2
    movsd   %xmm2, (%rax)
.L10:
    xorl    %eax, %eax
    call    getTimeSec
```

```
    movq    $0, 272(%rsp)
    cmpl    %edi, %edx
    jne .L1020
.L1021:
    call    getTimeSec
```

```
    movl    %r11d, %esi
    subl    %eax, %esi
    testl   %esi, %esi
    movl    %esi, %eax
    cmovg   %r13d, %eax
    addl    %r10d, %eax
    addl    %edx, %eax
    cmpl    %edx, %eax
    jle .L1131
    movslq  %edx, %rsi
    subl    %edx, %eax
    leaq    (%rbx,%rsi,8),
        %rcx
    movq    %rcx, %rdi
    salq    $60, %rdi
    shrq    $63, %rdi
    cmpl    %eax, %edi
    cmova   %eax, %edi
    cmpl    $3, %eax
    cmovbe  %eax, %edi
    testl   %edi, %edi
    je  .L1133
    movsd   (%rsp), %xmm2
    cmpl    $1, %edi
    movsd   (%r15,%rsi,8),
        %xmm0
    mulsd   %xmm2, %xmm0
    addsd   (%rcx), %xmm0
    movsd   %xmm0, (%rcx)
    leal    1(%rdx), %ecx
    je  .L1156
    movslq  %ecx, %rcx
    cmpl    $3, %edi
    movsd   (%r15,%rcx,8),
        %xmm0
    leaq    (%rbx,%rcx,8),
        %r9
    leal    2(%rdx), %ecx
    mulsd   %xmm2, %xmm0
    addsd   (%r9), %xmm0
    movsd   %xmm0, (%r9)
    jne .L1156
    movslq  %ecx, %rcx
    addl    $3, %edx
    movsd   (%r15,%rcx,8),
        %xmm0
    leaq    (%rbx,%rcx,8),
        %r9
    mulsd   %xmm2, %xmm0
    addsd   (%r9), %xmm0
    movsd   %xmm0, (%r9)
.L1134:
    cmpl    %edi, %eax
    je  .L1131
.L1133:
    subl    %edi, %eax
    movl    %edi, %ecx
    leal    -2(%rax), %r9d
    shrl    %r9d
    addl    $1, %r9d
    cmpl    $1, %eax
    leal    (%r9,%r9), %edi
    movl    %edi, 8(%rsp)
    je  .L1136
    addq    %rcx, %rsi
    xorl    %edi, %edi
    xorl    %ecx, %ecx
    salq    $3, %rsi
    leaq    (%r15,%rsi), %r
        14
    addq    %rbx, %rsi
```

```
.L1137:
    movupd   (%r14,%rcx), %
        xmm0
    addl     $1, %edi
    mulpd    %xmm1, %xmm0
    addpd    (%rsi,%rcx), %
        xmm0
    movaps   %xmm0, (%rsi,%
        rcx)
    addq     $16, %rcx
    cmpl     %r9d, %edi
    jb   .L1137
    movl     8(%rsp), %esi
    addl     %esi, %edx
    cmpl     %eax, %esi
    je   .L1131
.L1136:
    movsd    (%rsp), %xmm0
    movslq   %edx, %rdx
    leaq     (%rbx,%rdx,8),
        %rax
    mulsd    (%r15,%rdx,8),
        %xmm0
    addsd    (%rax), %xmm0
    movsd    %xmm0, (%rax)
.L1131:
    addl     $1, %r8d
    movq     $0, 272(%rsp)
    cmpl     %r12d, %r8d
    jne .L1129
    movq     32(%rsp), %r15
    movq     56(%rsp), %r14
.L1130:
    call     getTimeSec
```

Figure A.2: Direct comparison of assembler code snippets generated for sequential DAXPY implementations. Left: native, Mid: *alpaka* sequential, Right: *alpaka* sequential vectorized

A.1.3 Direct comparison of the PTX code generated for *CUDA* DAXPY implementations

```
//
// Generated by NVIDIA NVVM Compiler
//
// Compiler Build ID: CL-19324607
// Cuda compilation tools, release 7.0, V
    7.0.27
// Based on LLVM 3.4svn
//

.version 4.2
.target sm_35
.address_size 64

    // .globl   _Z27vecadd_axpy_par_cuda_
        kernelidPKdPd


.visible .entry _Z27vecadd_axpy_par_cuda_
    kernelidPKdPd(
```

```
//
// Generated by NVIDIA NVVM Compiler
//
// Compiler Build ID: CL-19324607
// Cuda compilation tools, release 7.0, V
    7.0.27
// Based on LLVM 3.4svn
//

.version 4.2
.target sm_35
.address_size 64

    // .globl   _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm1EEi16
        AxpyAlpakaKernelJidPKdPdEEEvNS_3
        VecIT_T0_EET1_DpT2_
.visible .entry _ZN6alpaka4exec4cuda6
    detail10cudaKernelISt17integral_
    constantImLm1EEi16
    AxpyAlpakaKernelJidPKdPdEEEvNS_3VecIT_
    T0_EET1_DpT2_(
     .param .align 16 .b8 _ZN6alpaka4exec4
    .param .u32 _Z27vecadd_axpy_par_cuda_
        kernelidPKdPd_param_0,


    .param .f64 _Z27vecadd_axpy_par_cuda_
        kernelidPKdPd_param_1,


    .param .u64 _Z27vecadd_axpy_par_cuda_
        kernelidPKdPd_param_2,


    .param .u64 _Z27vecadd_axpy_par_cuda_
        kernelidPKdPd_param_3


)
{
    .reg .pred  %p<2>;
    .reg .s32   %r<6>;
    .reg .f64   %fd<5>;
    .reg .s64   %rd<8>;


    ld.param.u32    %r2, [_Z27vecadd_axpy
        _par_cuda_kernelidPKdPd_param_0];


    ld.param.f64    %fd1, [_Z27vecadd_
        axpy_par_cuda_kernelidPKdPd_param
        _1];


    ld.param.u64    %rd1, [_Z27vecadd_
        axpy_par_cuda_kernelidPKdPd_param
        _2];


    ld.param.u64    %rd2, [_Z27vecadd_
        axpy_par_cuda_kernelidPKdPd_param
        _3];


    mov.u32     %r3, %ctaid.x;
    mov.u32     %r4, %ntid.x;
    mov.u32     %r5, %tid.x;
    mad.lo.s32  %r1, %r4, %r3, %r5;
    setp.ge.s32 %p1, %r1, %r2;
    @%p1 bra    BB6_2;

    cvta.to.global.u64  %rd3, %rd2;
    cvta.to.global.u64  %rd4, %rd1;
    mul.wide.s32    %rd5, %r1, 8;
    add.s64     %rd6, %rd4, %rd5;
    ld.global.nc.f64    %fd2, [%rd6];
    add.s64     %rd7, %rd3, %rd5;
    ld.global.f64   %fd3, [%rd7];
```

```
        cuda6detail10cudaKernelISt17
        integral_constantImLm1EEi16
        AxpyAlpakaKernelJidPKdPdEEEvNS_3
        VecIT_T0_EET1_DpT2__param_0[16],
    .param .align 1 .b8 _ZN6alpaka4exec4
        cuda6detail10cudaKernelISt17
        integral_constantImLm1EEi16
        AxpyAlpakaKernelJidPKdPdEEEvNS_3
        VecIT_T0_EET1_DpT2__param_1[1],
    .param .u32 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm1EEi16
        AxpyAlpakaKernelJidPKdPdEEEvNS_3
        VecIT_T0_EET1_DpT2__param_2,
    .param .f64 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm1EEi16
        AxpyAlpakaKernelJidPKdPdEEEvNS_3
        VecIT_T0_EET1_DpT2__param_3,
    .param .u64 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm1EEi16
        AxpyAlpakaKernelJidPKdPdEEEvNS_3
        VecIT_T0_EET1_DpT2__param_4,
    .param .u64 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm1EEi16
        AxpyAlpakaKernelJidPKdPdEEEvNS_3
        VecIT_T0_EET1_DpT2__param_5
)
{
    .reg .pred  %p<2>;
    .reg .s32   %r<6>;
    .reg .f64   %fd<5>;
    .reg .s64   %rd<8>;


    ld.param.u32    %r2, [_ZN6alpaka4exec
        4cuda6detail10cudaKernelISt17
        integral_constantImLm1EEi16
        AxpyAlpakaKernelJidPKdPdEEEvNS_3
        VecIT_T0_EET1_DpT2__param_2];
    ld.param.f64    %fd1, [_ZN6alpaka4
        exec4cuda6detail10cudaKernelISt17
        integral_constantImLm1EEi16
        AxpyAlpakaKernelJidPKdPdEEEvNS_3
        VecIT_T0_EET1_DpT2__param_3];
    ld.param.u64    %rd1, [_ZN6alpaka4
        exec4cuda6detail10cudaKernelISt17
        integral_constantImLm1EEi16
        AxpyAlpakaKernelJidPKdPdEEEvNS_3
        VecIT_T0_EET1_DpT2__param_4];
    ld.param.u64    %rd2, [_ZN6alpaka4
        exec4cuda6detail10cudaKernelISt17
        integral_constantImLm1EEi16
        AxpyAlpakaKernelJidPKdPdEEEvNS_3
        VecIT_T0_EET1_DpT2__param_5];
    mov.u32     %r3, %ctaid.x;
    mov.u32     %r4, %ntid.x;
    mov.u32     %r5, %tid.x;
    mad.lo.s32  %r1, %r4, %r3, %r5;
    setp.ge.s32 %p1, %r1, %r2;
    @%p1 bra    BB6_2;

    cvta.to.global.u64  %rd3, %rd2;
    cvta.to.global.u64  %rd4, %rd1;
    mul.wide.s32    %rd5, %r1, 8;
    add.s64     %rd6, %rd4, %rd5;
    ld.global.f64   %fd2, [%rd6];
    add.s64     %rd7, %rd3, %rd5;
    ld.global.f64   %fd3, [%rd7];
    fma.rn.f64  %fd4, %fd2, %fd1, %fd3;
    st.global.f64   [%rd7], %fd4;

BB6_2:
    ret;
}
```

```
    fma.rn.f64  %fd4, %fd2, %fd1, %fd3;
    st.global.f64   [%rd7], %fd4;

BB6_2:
    ret;
}
```

Figure A.3: Direct comparison of the PTX code generated for *CUDA* DAXPY implementations. Left: native *CUDA*, Right: *alpaka CUDA* non-vectorized

A.1.4 Vectorized *alpaka CUDA* DAXPY PTX Code

```
//
// Generated by NVIDIA NVVM Compiler
//
// Compiler Build ID: CL-19324607
// Cuda compilation tools, release 7.0, V7.0.27
// Based on LLVM 3.4svn
//

.version 4.2
.target sm_35
.address_size 64

// .globl   _ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_constantImLm1EEi26
    AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_DpT2_
.visible .entry _ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_constantImLm1
    EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_DpT2_(
    .param .align 16 .b8 _ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_
        constantImLm1EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_
        DpT2__param_0[16],
    .param .align 1 .b8 _ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_
        constantImLm1EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_
        DpT2__param_1[1],
    .param .u32 _ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_constantImLm1
        EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_DpT2__param_2,
    .param .f64 _ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_constantImLm1
        EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_DpT2__param_3,
    .param .u64 _ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_constantImLm1
        EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_DpT2__param_4,
    .param .u64 _ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_constantImLm1
        EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_DpT2__param_5
)
{
    .reg .pred  %p<4>;
    .reg .s32   %r<19>;
    .reg .f64   %fd<5>;
    .reg .s64   %rd<14>;


    ld.param.u32    %r4, [_ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_
        constantImLm1EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_
        DpT2__param_0];
    ld.param.u32    %r9, [_ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_
        constantImLm1EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_
        DpT2__param_2];
    ld.param.f64    %fd1, [_ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_
        constantImLm1EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_
        DpT2__param_3];
    ld.param.u64    %rd8, [_ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_
        constantImLm1EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_
        DpT2__param_4];
    ld.param.u64    %rd9, [_ZN6alpaka4exec4cuda6detail10cudaKernelISt17integral_
        constantImLm1EEi26AxpyVectorizedAlpakaKernelJidPKdPdEEEvNS_3VecIT_T0_EET1_
        DpT2__param_5];
    mov.u32     %r1, %ntid.x;
    mov.u32     %r2, %ctaid.x;
    mov.u32     %r3, %tid.x;
    mad.lo.s32  %r10, %r1, %r2, %r3;
    mul.lo.s32  %r18, %r10, %r4;
    setp.ge.s32 %p1, %r18, %r9;
    @%p1 bra    BB7_4;

    sub.s32     %r11, %r9, %r4;
    sub.s32     %r12, %r11, %r18;
    mov.u32     %r13, 0;
    min.s32     %r14, %r13, %r12;
    add.s32     %r15, %r18, %r4;
    add.s32     %r6, %r15, %r14;
    setp.ge.s32 %p2, %r18, %r6;
    @%p2 bra    BB7_4;

    cvta.to.global.u64  %rd13, %rd9;
    cvta.to.global.u64  %rd12, %rd8;
    mul.lo.s32  %r17, %r4, %r10;
    mul.wide.s32    %rd3, %r17, 8;

BB7_3:
    add.s64     %rd10, %rd12, %rd3;
    ld.global.f64   %fd2, [%rd10];
    add.s64     %rd11, %rd13, %rd3;
    ld.global.f64   %fd3, [%rd11];
    fma.rn.f64  %fd4, %fd2, %fd1, %fd3;
    st.global.f64   [%rd11], %fd4;
    add.s64     %rd13, %rd13, 8;
    add.s64     %rd12, %rd12, 8;
    add.s32     %r18, %r18, 1;
    setp.lt.s32 %p3, %r18, %r6;
    @%p3 bra    BB7_3;

BB7_4:
    ret;
}
```

Figure A.4: Vectorized *alpaka CUDA* DAXPY PTX code

A.2 Generalized Matrix-Matrix-Multiplication

A.2.1 Direct Comparison of the native *CUDA* and *alpaka CUDA* DGEMM Kernels

```
//-----------------------------------------------------------------------------
// CUDA generalized matrix-matrix-multiplication kernel.
// It uses extern shared memory to support dynamic block sizes.
//-----------------------------------------------------------------------------
__global__ void matmul_gemm_par_cuda_dyn_block_size_1d_extern_shared_kernel(
    TSize const m, TSize const n, TSize const k,
    TElem const alpha,
    TElem const * const MATMUL_RESTRICT A, TSize const lda,
    TElem const * const MATMUL_RESTRICT B, TSize const ldb,
    TElem const beta,
    TElem * const MATMUL_RESTRICT C, TSize const ldc)
{
    // Column and row of C to calculate.
    TSize const gridThreadIdxX = blockIdx.x*blockDim.x + threadIdx.x;
    TSize const gridThreadIdxY = blockIdx.y*blockDim.y + threadIdx.y;

    // Column and row inside the block of C to calculate.
    TSize const blockThreadIdxX = threadIdx.x;
    TSize const blockThreadIdxY = threadIdx.y;

    // The block threads extents.
    TSize const blockThreadsExtentX = blockDim.x;
    TSize const blockThreadsExtentY = blockDim.y;
    TSize const & blockThreadsExtent = blockThreadsExtentX;

    // Shared memory used to store the current blocks of A and B.
    extern __shared__ TElem pBlockSharedA[];
    TElem * const pBlockSharedB(pBlockSharedA + blockThreadsExtentX*
        blockThreadsExtentY);

    TSize const sharedBlockIdx1d(blockThreadIdxY*blockThreadsExtentX +
        blockThreadIdxX);

    // If the element corresponding to the current thread is outside of the
        respective matrix.
    bool const insideA = (gridThreadIdxY < m);
    bool const insideB = (gridThreadIdxX < n);
    bool const insideC = (insideA && insideB);

    TElem dotProduct(0);

    // Loop over all blocks of A and B that are required to compute the C block.
    TSize const blockMulCount(
        static_cast<TSize>(
            ceil(
                static_cast<float>(k) / static_cast<float>(blockThreadsExtent))));
    for (TSize k2(0); k2<blockMulCount; ++k2)
    {
        // Copy the current blocks of A and B into shared memory in parallel.
        // If the element of the current thread is outside of the matrix, zero is
            written into the shared memory.
        // This is possible because zero is a result neutral extension of the
            matrices regarding the dot product.
        TSize const AIdxX(k2*blockThreadsExtentX + blockThreadIdxX);
        TSize const AIdx1d(gridThreadIdxY*lda + AIdxX);
        pBlockSharedA[sharedBlockIdx1d] =
            ((!insideA) || (AIdxX >= k))
            ? static_cast<TElem>(0)
            : A[AIdx1d];

        TSize const BIdxY(k2*blockThreadsExtentY + blockThreadIdxY);
        TSize const BIdx1d(BIdxY*ldb + gridThreadIdxX);
        pBlockSharedB[sharedBlockIdx1d] =
            ((!insideB) || (BIdxY >= k))
            ? static_cast<TElem>(0)
            : B[BIdx1d];

        // Synchronize to make sure the complete blocks are loaded before starting
            the computation.
        __syncthreads();

        // Compute the dot products within shared memory.
        for (TSize k3(0); k3<blockThreadsExtent; ++k3)
        {
            dotProduct += pBlockSharedA[blockThreadIdxY*blockThreadsExtentX + k3]
                * pBlockSharedB[k3*blockThreadsExtentY + blockThreadIdxX];
        }

        // Synchronize to make sure that the preceding computation is done before
            loading the next blocks of A and B.
        __syncthreads();
    }

    if (insideC)
    {
        TSize const CIdx1d(gridThreadIdxY*ldc + gridThreadIdxX);
        C[CIdx1d] = alpha * dotProduct + beta * C[CIdx1d];
    }
}
```

Figure A.5: Generalized matrix-matrix-mulitplication kernel using *CUDA*

```
//#############################################################################
// alpaka generalized matrix-matrix-multiplication kernel.
//#############################################################################
class GemmAlpakaKernel
{
public:
```

```cpp
    template<
        typename TAcc,
        typename TElem>
    ALPAKA_FN_ACC auto operator()(
        TAcc const & acc,
        TSize const & m, TSize const & n, TSize const & k,
        TElem const & alpha,
        TElem const * const MATMUL_RESTRICT A, TSize const & lda,
        TElem const * const MATMUL_RESTRICT B, TSize const & ldb,
        TElem const & beta,
        TElem * const MATMUL_RESTRICT C, TSize const & ldc) const
    -> void
    {
        // Assure that the kernel is executed with a 2-dimensional accelerator.
        static_assert(alpaka::dim::Dim<TAcc>::value == 2u,
            "The accelerator used for the GemmAlpakaKernel has to be 2 dimensional!"
                );

        // Column and row of C to calculate.
        auto const gridThreadIdx(alpaka::idx::getIdx<alpaka::Grid, alpaka::Threads>(
            acc));
        TSize const & gridThreadIdxX(gridThreadIdx[1u]);
        TSize const & gridThreadIdxY(gridThreadIdx[0u]);

        // Column and row inside the block of C to calculate.
        auto const blockThreadIdx(alpaka::idx::getIdx<alpaka::Block, alpaka::Threads
            >(acc));
        TSize const & blockThreadIdxX(blockThreadIdx[1u]);
        TSize const & blockThreadIdxY(blockThreadIdx[0u]);

        // The block threads extents.
        auto const blockThreadsExtents(alpaka::workdiv::getWorkDiv<alpaka::Block,
            alpaka::Threads>(acc));
        TSize const & blockThreadsExtentX(blockThreadsExtents[1u]);
        TSize const & blockThreadsExtentY(blockThreadsExtents[0u]);
        TSize const & blockThreadsExtent(blockThreadsExtentX);

        // Shared memory used to store the current blocks of A and B.
        TElem * const pBlockSharedA(acc.template getBlockSharedExternMem<TElem>());
        TElem * const pBlockSharedB(pBlockSharedA + blockThreadsExtentX*
            blockThreadsExtentY);

        TSize const sharedBlockIdx1d(blockThreadIdxY*blockThreadsExtentX +
            blockThreadIdxX);

        // If the element corresponding to the current thread is outside of the
            respective matrix.
        bool const insideA = (gridThreadIdxY < m);
        bool const insideB = (gridThreadIdxX < n);
        bool const insideC = (insideA && insideB);

        TElem dotProduct(0);

        // Loop over all blocks of A and B that are required to compute the C block.
        TSize const blockMulCount(
            static_cast<TSize>(
                alpaka::math::ceil(
                    acc,
                    static_cast<float>(k)/static_cast<float>(blockThreadsExtent))));
        for(TSize k2(0); k2<blockMulCount; ++k2)
        {
            // Copy the current blocks of A and B into shared memory in parallel.
            // If the element of the current thread is outside of the matrix, zero is written into the shared memory.
            // This is possible because zero is a result neutral extension of the matrices regarding the dot product.
            TSize const AIdxX(k2*blockThreadsExtentX + blockThreadIdxX);
            TSize const AIdx1d(gridThreadIdxY*lda + AIdxX);
            pBlockSharedA[sharedBlockIdx1d] =
                ((!insideA) || (AIdxX>=k))
                ? static_cast<TElem>(0)
                : A[AIdx1d];

            TSize const BIdxY(k2*blockThreadsExtentY + blockThreadIdxY);
            TSize const BIdx1d(BIdxY*ldb + gridThreadIdxX);
            pBlockSharedB[sharedBlockIdx1d] =
                ((!insideB) || (BIdxY>=k))
                ? static_cast<TElem>(0)
                : B[BIdx1d];

            // Synchronize to make sure the complete blocks are loaded before starting the computation.
            acc.syncBlockThreads();

            // Compute the dot products within shared memory.
            for(TSize k3(0); k3<blockThreadsExtent; ++k3)
            {
                dotProduct += pBlockSharedA[blockThreadIdxY*blockThreadsExtentX + k3]
                    * pBlockSharedB[k3*blockThreadsExtentY + blockThreadIdxX];
            }

            // Synchronize to make sure that the preceding computation is done before loading the next blocks of A and B.
            acc.syncBlockThreads();
        }

        if(insideC)
        {
            TSize const CIdx1d(gridThreadIdxY*ldc + gridThreadIdxX);
            C[CIdx1d] = alpha * dotProduct + beta * C[CIdx1d];
        }
    }
};
```

Figure A.6: Generalized matrix-matrix-mulitplication kernel using *alpaka*

```
//#############################################################################
// alpaka generalized matrix-matrix-multiplication kernel.
//#############################################################################
class GemmAlpakaNoSharedKernel
{
public:
    template<
        typename TAcc,
        typename TElem>
    ALPAKA_FN_ACC auto operator()(
        TAcc const & acc,
        TSize const & m, TSize const & n, TSize const & k,
        TElem const & alpha,
        TElem const * const MATMUL_RESTRICT A, TSize const & lda,
        TElem const * const MATMUL_RESTRICT B, TSize const & ldb,
        TElem const & beta,
        TElem * const MATMUL_RESTRICT C, TSize const & ldc) const
    -> void
    {
        static_assert(alpaka::dim::Dim<TAcc>::value == 2u,
            "The accelerator used for the GemmAlpakaKernel has to be 2 dimensional!"
                );

        // Column and row of C to calculate.
        auto const gridThreadIdx(alpaka::idx::getIdx<alpaka::Grid, alpaka::Threads>(
            acc));
        TSize const & gridThreadIdxX(gridThreadIdx[1u]);
        TSize const & gridThreadIdxY(gridThreadIdx[0u]);

        // If the element corresponding to the current thread is outside of the
            respective matrix.
        bool const insideA(gridThreadIdxY < m);
        bool const insideB(gridThreadIdxX < n);
        bool const insideC(insideA && insideB);

        // If the element is outside of the matrix it was only a helper thread that
            did not calculate any meaningful results.
        if(insideC)
        {
            TElem dotProduct(0);

            // Compute the dot products.
            for(TSize k3(0); k3<k; ++k3)
            {
                TSize const AIdx1d(gridThreadIdxY*lda + k3);
                TSize const BIdx1d(gridThreadIdxX + k3*ldb);
                dotProduct += A[AIdx1d] * B[BIdx1d];
            }

            TSize const CIdx1d(gridThreadIdxY*ldc + gridThreadIdxX);
            C[CIdx1d] = alpha * dotProduct + beta * C[CIdx1d];
        }
    }
};
```

Figure A.7: Generalized matrix-matrix-mulitplication kernel using *alpaka* without shared memory copy

```
//#############################################################################
// alpaka generalized matrix-matrix-multiplication kernel.
//#############################################################################
class GemmAlpakaNoSharedTransposedBKernel
{
public:
    template<
        typename TAcc,
        typename TElem>
    ALPAKA_FN_ACC auto operator()(
        TAcc const & acc,
        TSize const & m, TSize const & n, TSize const & k,
        TElem const & alpha,
        TElem const * const MATMUL_RESTRICT A, TSize const & lda,
        TElem const * const MATMUL_RESTRICT B, TSize const & ldb,
        TElem const & beta,
        TElem * const MATMUL_RESTRICT C, TSize const & ldc) const
    -> void
    {
        // Assure that the kernel is executed with a 2-dimensional accelerator.
        static_assert(alpaka::dim::Dim<TAcc>::value == 2u,
            "The accelerator used for the GemmAlpakaNoSharedTransposedBKernel has to
                 be 2 dimensional!");

        // Column and row of C to calculate.
        auto const gridThreadIdx(alpaka::idx::getIdx<alpaka::Grid, alpaka::Threads>(
            acc));
        TSize const & gridThreadIdxX(gridThreadIdx[1u]);
        TSize const & gridThreadIdxY(gridThreadIdx[0u]);

        // If the element corresponding to the current thread is outside of the
            respective matrix.
        bool const insideA = (gridThreadIdxY < m);
        bool const insideB = (gridThreadIdxX < n);
        bool const insideC = (insideA && insideB);

        // If the element is outside of the matrix it was only a helper thread that
            did not calculate any meaningful results.
        if(insideC)
        {
            TElem dotProduct(0);

            // Compute the dot products.
            for(TSize k3(0); k3<k; ++k3)
            {
                TSize const AIdx1d(gridThreadIdxY*lda + k3);
                TSize const BIdx1d(gridThreadIdxX*ldb + k3);
                dotProduct += A[AIdx1d] * B[BIdx1d];
            }

            TSize const CIdx1d(gridThreadIdxY*ldc + gridThreadIdxX);
            C[CIdx1d] = alpha * dotProduct + beta * C[CIdx1d];
        }
    }
};
```

Figure A.8: Generalized matrix-matrix-mulitplication kernel using *alpaka* without shared memory copy and with a transposed B matrix

A.2.2 Direct Comparison of the PTX code generated from native *CUDA* and *alpaka CUDA* DGEMM

```
//
// Generated by NVIDIA NVVM Compiler
//
// Compiler Build ID: CL-19324607
// Cuda compilation tools, release 7.0, V
   7.0.27
// Based on LLVM 3.4svn
//

.version 4.2
.target sm_35
.address_size 64

.extern .shared .align 8 .b8
    pBlockSharedA[];

    // .globl   _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi


.visible .entry _Z59matmul_gemm_par_cuda_
    dyn_block_size_1d_extern_shared_
    kerneliiidPKdiS0_idPdi(


    .param .u32 _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi_param_0,


    .param .u32 _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi_param_1,
```

```
//
// Generated by NVIDIA NVVM Compiler
//
// Compiler Build ID: CL-19324607
// Cuda compilation tools, release 7.0, V
    7.0.27
// Based on LLVM 3.4svn
//

.version 4.2
.target sm_35
.address_size 64

.extern .shared .align 8 .b8 _ZN6alpaka3
    acc5shMemE[];

    // .globl   _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2_
.visible .entry _ZN6alpaka4exec4cuda6
    detail10cudaKernelISt17integral_
    constantImLm2EEi22
    GemmAlpakaSharedKernelJiiidPKdiS8_
    idPdiEEEvNS_3VecIT_T0_EET1_DpT2_(
    .param .align 16 .b8 _ZN6alpaka4exec4
        cuda6detail10cudaKernelISt17
        integral_constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_0[16],
    .param .align 1 .b8 _ZN6alpaka4exec4
        cuda6detail10cudaKernelISt17
        integral_constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_1[1],
    .param .u32 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_2,
    .param .u32 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
    .param .u32 _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi_param_2,


    .param .f64 _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi_param_3,


    .param .u64 _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi_param_4,


    .param .u32 _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi_param_5,


    .param .u64 _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi_param_6,


    .param .u32 _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi_param_7,


    .param .f64 _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi_param_8,


    .param .u64 _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi_param_9,


    .param .u32 _Z59matmul_gemm_par_cuda_
        dyn_block_size_1d_extern_shared_
        kerneliiidPKdiS0_idPdi_param_10


)
{
    .reg .pred  %p<14>;
    .reg .f32   %f<5>;
    .reg .s32   %r<35>;
    .reg .f64   %fd<34>;
    .reg .s64   %rd<28>;


    ld.param.u32    %r16, [_Z59matmul_
        gemm_par_cuda_dyn_block_size_1d_
        extern_shared_kerneliiidPKdiS0_
        idPdi_param_0];
```

```
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_3,
    .param .u32 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_4,
    .param .f64 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_5,
    .param .u64 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_6,
    .param .u32 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_7,
    .param .u64 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_8,
    .param .u32 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_9,
    .param .f64 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_10,
    .param .u64 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_11,
    .param .u32 _ZN6alpaka4exec4cuda6
        detail10cudaKernelISt17integral_
        constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
        param_12
)
{
    .reg .pred  %p<14>;
    .reg .f32   %f<5>;
    .reg .s32   %r<35>;
    .reg .f64   %fd<34>;
    .reg .s64   %rd<28>;


    ld.param.u32    %r16, [_ZN6alpaka4
        exec4cuda6detail10cudaKernelISt17
        integral_constantImLm2EEi22
        GemmAlpakaSharedKernelJiiidPKdiS8_
        idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
```

```
ld.param.u32    %r17, [_Z59matmul_
    gemm_par_cuda_dyn_block_size_1d_
    extern_shared_kerneliiidPKdiS0_
    idPdi_param_1];

ld.param.u32    %r18, [_Z59matmul_
    gemm_par_cuda_dyn_block_size_1d_
    extern_shared_kerneliiidPKdiS0_
    idPdi_param_2];

ld.param.f64    %fd10, [_Z59matmul_
    gemm_par_cuda_dyn_block_size_1d_
    extern_shared_kerneliiidPKdiS0_
    idPdi_param_3];

ld.param.u64    %rd6, [_Z59matmul_
    gemm_par_cuda_dyn_block_size_1d_
    extern_shared_kerneliiidPKdiS0_
    idPdi_param_4];

ld.param.u32    %r19, [_Z59matmul_
    gemm_par_cuda_dyn_block_size_1d_
    extern_shared_kerneliiidPKdiS0_
    idPdi_param_5];

ld.param.u64    %rd7, [_Z59matmul_
    gemm_par_cuda_dyn_block_size_1d_
    extern_shared_kerneliiidPKdiS0_
    idPdi_param_6];

ld.param.u32    %r20, [_Z59matmul_
    gemm_par_cuda_dyn_block_size_1d_
    extern_shared_kerneliiidPKdiS0_
    idPdi_param_7];

ld.param.f64    %fd11, [_Z59matmul_
    gemm_par_cuda_dyn_block_size_1d_
    extern_shared_kerneliiidPKdiS0_
    idPdi_param_8];

ld.param.u64    %rd8, [_Z59matmul_
    gemm_par_cuda_dyn_block_size_1d_
    extern_shared_kerneliiidPKdiS0_
    idPdi_param_9];

ld.param.u32    %r21, [_Z59matmul_
    gemm_par_cuda_dyn_block_size_1d_
    extern_shared_kerneliiidPKdiS0_
    idPdi_param_10];

mov.u32     %r1, %ntid.x;
mov.u32     %r22, %ctaid.x;
mov.u32     %r2, %tid.x;
mad.lo.s32  %r3, %r1, %r22, %r2;
mov.u32     %r4, %ntid.y;
mov.u32     %r23, %ctaid.y;
mov.u32     %r5, %tid.y;
mad.lo.s32  %r6, %r4, %r23, %r5;
mul.lo.s32  %r24, %r4, %r1;
cvt.s64.s32 %rd1, %r24;
```

```
    param_2];
ld.param.u32    %r17, [_ZN6alpaka4
    exec4cuda6detail10cudaKernelISt17
    integral_constantImLm2EEi22
    GemmAlpakaSharedKernelJiiidPKdiS8_
    idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
    param_3];
ld.param.u32    %r18, [_ZN6alpaka4
    exec4cuda6detail10cudaKernelISt17
    integral_constantImLm2EEi22
    GemmAlpakaSharedKernelJiiidPKdiS8_
    idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
    param_4];
ld.param.f64    %fd10, [_ZN6alpaka4
    exec4cuda6detail10cudaKernelISt17
    integral_constantImLm2EEi22
    GemmAlpakaSharedKernelJiiidPKdiS8_
    idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
    param_5];
ld.param.u64    %rd6, [_ZN6alpaka4
    exec4cuda6detail10cudaKernelISt17
    integral_constantImLm2EEi22
    GemmAlpakaSharedKernelJiiidPKdiS8_
    idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
    param_6];
ld.param.u32    %r19, [_ZN6alpaka4
    exec4cuda6detail10cudaKernelISt17
    integral_constantImLm2EEi22
    GemmAlpakaSharedKernelJiiidPKdiS8_
    idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
    param_7];
ld.param.u64    %rd7, [_ZN6alpaka4
    exec4cuda6detail10cudaKernelISt17
    integral_constantImLm2EEi22
    GemmAlpakaSharedKernelJiiidPKdiS8_
    idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
    param_8];
ld.param.u32    %r20, [_ZN6alpaka4
    exec4cuda6detail10cudaKernelISt17
    integral_constantImLm2EEi22
    GemmAlpakaSharedKernelJiiidPKdiS8_
    idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
    param_9];
ld.param.f64    %fd11, [_ZN6alpaka4
    exec4cuda6detail10cudaKernelISt17
    integral_constantImLm2EEi22
    GemmAlpakaSharedKernelJiiidPKdiS8_
    idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
    param_10];
ld.param.u64    %rd8, [_ZN6alpaka4
    exec4cuda6detail10cudaKernelISt17
    integral_constantImLm2EEi22
    GemmAlpakaSharedKernelJiiidPKdiS8_
    idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
    param_11];
ld.param.u32    %r21, [_ZN6alpaka4
    exec4cuda6detail10cudaKernelISt17
    integral_constantImLm2EEi22
    GemmAlpakaSharedKernelJiiidPKdiS8_
    idPdiEEEvNS_3VecIT_T0_EET1_DpT2__
    param_12];
mov.u32     %r1, %ntid.y;
mov.u32     %r22, %ctaid.y;
mov.u32     %r2, %ntid.x;
mov.u32     %r23, %ctaid.x;
mov.u32     %r3, %tid.y;
mad.lo.s32  %r4, %r1, %r22, %r3;
mov.u32     %r5, %tid.x;
mad.lo.s32  %r6, %r2, %r23, %r5;
mul.lo.s32  %r24, %r1, %r2;
cvt.s64.s32 %rd1, %r24;
```

```
    mul.lo.s32  %r7, %r5, %r1;
    cvt.rn.f32.s32  %f1, %r1;
    cvt.rn.f32.s32  %f2, %r18;
    div.approx.f32  %f3, %f2, %f1;
    cvt.rpi.f32.f32 %f4, %f3;
    cvt.rzi.s32.f32 %r8, %f4;
    mov.f64     %fd28, 0d
        0000000000000000;
    setp.lt.s32 %p3, %r8, 1;
    @%p3 bra    BB6_9;

    cvta.to.global.u64  %rd2, %rd7;
    cvta.to.global.u64  %rd3, %rd6;
    add.s32     %r26, %r7, %r2;
    mul.lo.s32  %r9, %r6, %r19;
    setp.ge.s32 %p1, %r6, %r16;
    cvt.s64.s32 %rd9, %r26;
    mul.wide.s32    %rd10, %r26, 8;
    mov.u64     %rd11, pBlockSharedA;

    add.s64     %rd4, %rd11, %rd10;
    setp.ge.s32 %p2, %r3, %r17;
    add.s64     %rd12, %rd9, %rd1;
    shl.b64     %rd13, %rd12, 3;
    add.s64     %rd5, %rd11, %rd13;
    mov.f64     %fd13, 0d
        0000000000000000;
    mov.u32     %r33, 0;
    mov.f64     %fd33, %fd13;

BB6_2:
    mov.f64     %fd29, %fd33;
    mad.lo.s32  %r11, %r33, %r1, %r2;
    setp.ge.s32 %p4, %r11, %r18;
    or.pred     %p5, %p4, %p1;
    mov.f64     %fd32, %fd13;
    @%p5 bra    BB6_4;

    add.s32     %r27, %r11, %r9;
    mul.wide.s32    %rd14, %r27, 8;
    add.s64     %rd15, %rd3, %rd14;
    ld.global.nc.f64    %fd2, [%rd15];
    mov.f64     %fd32, %fd2;

BB6_4:
    mov.f64     %fd3, %fd32;
    st.shared.f64   [%rd4], %fd3;
    mad.lo.s32  %r12, %r33, %r4, %r5;
    setp.ge.s32 %p6, %r12, %r18;
    or.pred     %p7, %p6, %p2;
    mov.f64     %fd31, %fd13;
    @%p7 bra    BB6_6;

    mad.lo.s32  %r28, %r12, %r20, %r3;
    mul.wide.s32    %rd16, %r28, 8;
    add.s64     %rd17, %rd2, %rd16;
    ld.global.nc.f64    %fd31, [%rd17];

BB6_6:
    st.shared.f64   [%rd5], %fd31;
    bar.sync    0;
    mov.u32     %r34, 0;
    setp.lt.s32 %p8, %r1, 1;
    mov.f64     %fd30, %fd29;
    @%p8 bra    BB6_8;

BB6_7:
    add.s32     %r30, %r34, %r7;
    mul.wide.s32    %rd18, %r30, 8;
    add.s64     %rd20, %rd11, %rd18;
    mad.lo.s32  %r31, %r34, %r4, %r2;
```

```
    mul.lo.s32  %r7, %r3, %r2;
    cvt.rn.f32.s32  %f1, %r2;
    cvt.rn.f32.s32  %f2, %r18;
    div.approx.f32  %f3, %f2, %f1;
    cvt.rpi.f32.f32 %f4, %f3;
    cvt.rzi.s32.f32 %r8, %f4;
    mov.f64     %fd28, 0d
        0000000000000000;
    setp.lt.s32 %p3, %r8, 1;
    @%p3 bra    BB6_9;

    cvta.to.global.u64  %rd2, %rd7;
    cvta.to.global.u64  %rd3, %rd6;
    add.s32     %r26, %r7, %r5;
    mul.lo.s32  %r9, %r4, %r19;
    setp.ge.s32 %p1, %r4, %r16;
    cvt.s64.s32 %rd9, %r26;
    mul.wide.s32    %rd10, %r26, 8;
    mov.u64     %rd11, _ZN6alpaka3acc5
        shMemE;
    add.s64     %rd4, %rd11, %rd10;
    setp.ge.s32 %p2, %r6, %r17;
    add.s64     %rd12, %rd9, %rd1;
    shl.b64     %rd13, %rd12, 3;
    add.s64     %rd5, %rd11, %rd13;
    mov.f64     %fd13, 0d
        0000000000000000;
    mov.u32     %r33, 0;
    mov.f64     %fd33, %fd13;

BB6_2:
    mov.f64     %fd29, %fd33;
    mad.lo.s32  %r11, %r33, %r2, %r5;
    setp.ge.s32 %p4, %r11, %r18;
    or.pred     %p5, %p4, %p1;
    mov.f64     %fd32, %fd13;
    @%p5 bra    BB6_4;

    add.s32     %r27, %r11, %r9;
    mul.wide.s32    %rd14, %r27, 8;
    add.s64     %rd15, %rd3, %rd14;
    ld.global.f64   %fd2, [%rd15];
    mov.f64     %fd32, %fd2;

BB6_4:
    mov.f64     %fd3, %fd32;
    st.shared.f64   [%rd4], %fd3;
    mad.lo.s32  %r12, %r33, %r1, %r3;
    setp.ge.s32 %p6, %r12, %r18;
    or.pred     %p7, %p6, %p2;
    mov.f64     %fd31, %fd13;
    @%p7 bra    BB6_6;

    mad.lo.s32  %r28, %r12, %r20, %r6;
    mul.wide.s32    %rd16, %r28, 8;
    add.s64     %rd17, %rd2, %rd16;
    ld.global.f64   %fd31, [%rd17];

BB6_6:
    st.shared.f64   [%rd5], %fd31;
    bar.sync    0;
    mov.u32     %r34, 0;
    setp.lt.s32 %p8, %r2, 1;
    mov.f64     %fd30, %fd29;
    @%p8 bra    BB6_8;

BB6_7:
    add.s32     %r30, %r34, %r7;
    mul.wide.s32    %rd18, %r30, 8;
    add.s64     %rd20, %rd11, %rd18;
    mad.lo.s32  %r31, %r34, %r1, %r5;
    cvt.s64.s32 %rd21, %r31;
    add.s64     %rd22, %rd21, %rd1;
    shl.b64     %rd23, %rd22, 3;
    add.s64     %rd24, %rd11, %rd23;
    ld.shared.f64   %fd16, [%rd24];
    ld.shared.f64   %fd17, [%rd20];
    fma.rn.f64  %fd29, %fd17, %fd16, %fd
        29;
    add.s32     %r34, %r34, 1;
    setp.lt.s32 %p9, %r34, %r1;
    mov.f64     %fd30, %fd29;
    @%p9 bra    BB6_7;

BB6_8:
    mov.f64     %fd33, %fd30;
    bar.sync    0;
    add.s32     %r33, %r33, 1;
    setp.lt.s32 %p10, %r33, %r8;
    mov.f64     %fd28, %fd33;
    @%p10 bra   BB6_2;

BB6_9:
    setp.lt.s32 %p11, %r6, %r16;
    setp.lt.s32 %p12, %r3, %r17;
    and.pred    %p13, %p12, %p11;
    @!%p13 bra  BB6_11;
    bra.uni     BB6_10;

BB6_10:
    cvta.to.global.u64  %rd25, %rd8;
    mad.lo.s32  %r32, %r6, %r21, %r3;
    mul.wide.s32    %rd26, %r32, 8;
    add.s64     %rd27, %rd25, %rd26;
    ld.global.f64   %fd18, [%rd27];
    mul.f64     %fd19, %fd18, %fd11;
    fma.rn.f64  %fd20, %fd28, %fd10, %fd
        19;
    st.global.f64   [%rd27], %fd20;

BB6_11:
    ret;
}
```

Figure A.9: Native *CUDA* DGEMM PTX code

```
    cvt.s64.s32 %rd21, %r31;
    add.s64     %rd22, %rd21, %rd1;
    shl.b64     %rd23, %rd22, 3;
    add.s64     %rd24, %rd11, %rd23;
    ld.shared.f64   %fd16, [%rd24];
    ld.shared.f64   %fd17, [%rd20];
    fma.rn.f64  %fd29, %fd17, %fd16, %fd
        29;
    add.s32     %r34, %r34, 1;
    setp.lt.s32 %p9, %r34, %r2;
    mov.f64     %fd30, %fd29;
    @%p9 bra    BB6_7;

BB6_8:
    mov.f64     %fd33, %fd30;
    bar.sync    0;
    add.s32     %r33, %r33, 1;
    setp.lt.s32 %p10, %r33, %r8;
    mov.f64     %fd28, %fd33;
    @%p10 bra   BB6_2;

BB6_9:
    setp.lt.s32 %p11, %r4, %r16;
    setp.lt.s32 %p12, %r6, %r17;
    and.pred    %p13, %p12, %p11;
    @!%p13 bra  BB6_11;
    bra.uni     BB6_10;

BB6_10:
    cvta.to.global.u64  %rd25, %rd8;
    mad.lo.s32  %r32, %r4, %r21, %r6;
    mul.wide.s32    %rd26, %r32, 8;
    add.s64     %rd27, %rd25, %rd26;
    ld.global.f64   %fd18, [%rd27];
    mul.f64     %fd19, %fd18, %fd11;
    fma.rn.f64  %fd20, %fd28, %fd10, %fd
        19;
    st.global.f64   [%rd27], %fd20;

BB6_11:
    ret;
}
```

Figure A.10: *alpaka CUDA* DGEMM PTX code

B Additional Measurements

B.1 Generalized Vector Addition

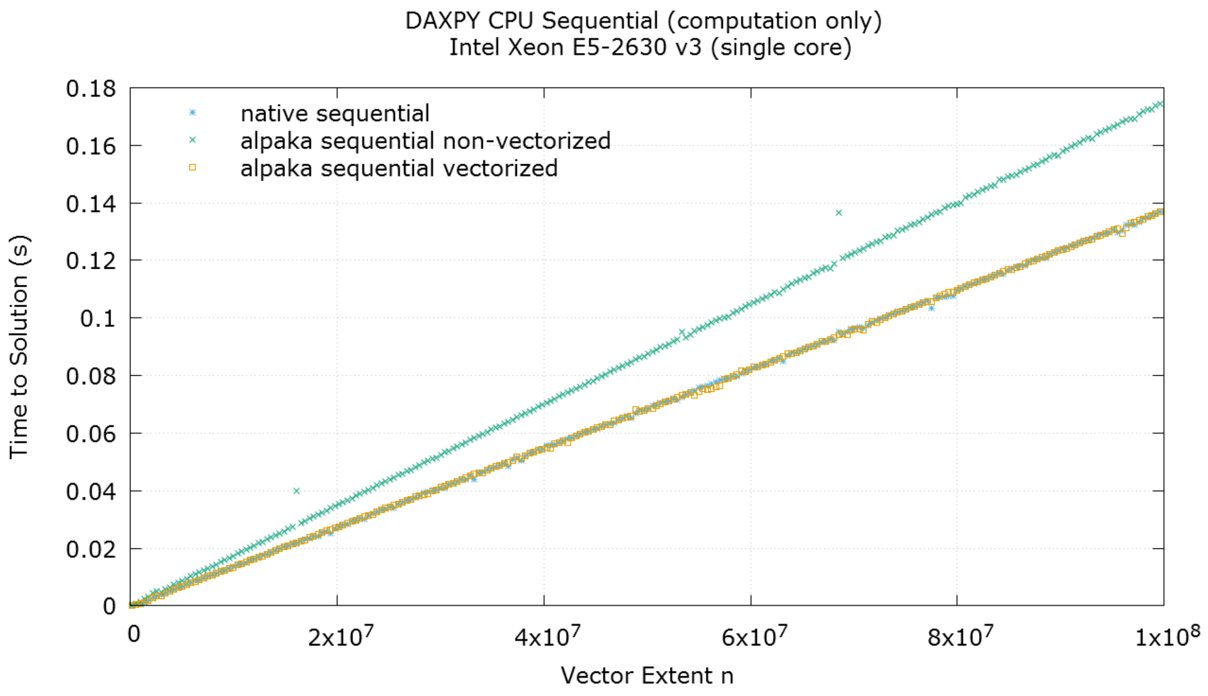

Figure B.1: Comparison of the time to solution of the vectorized and non-vectorized *alpaka* sequential and the native sequential generalized vector addition.

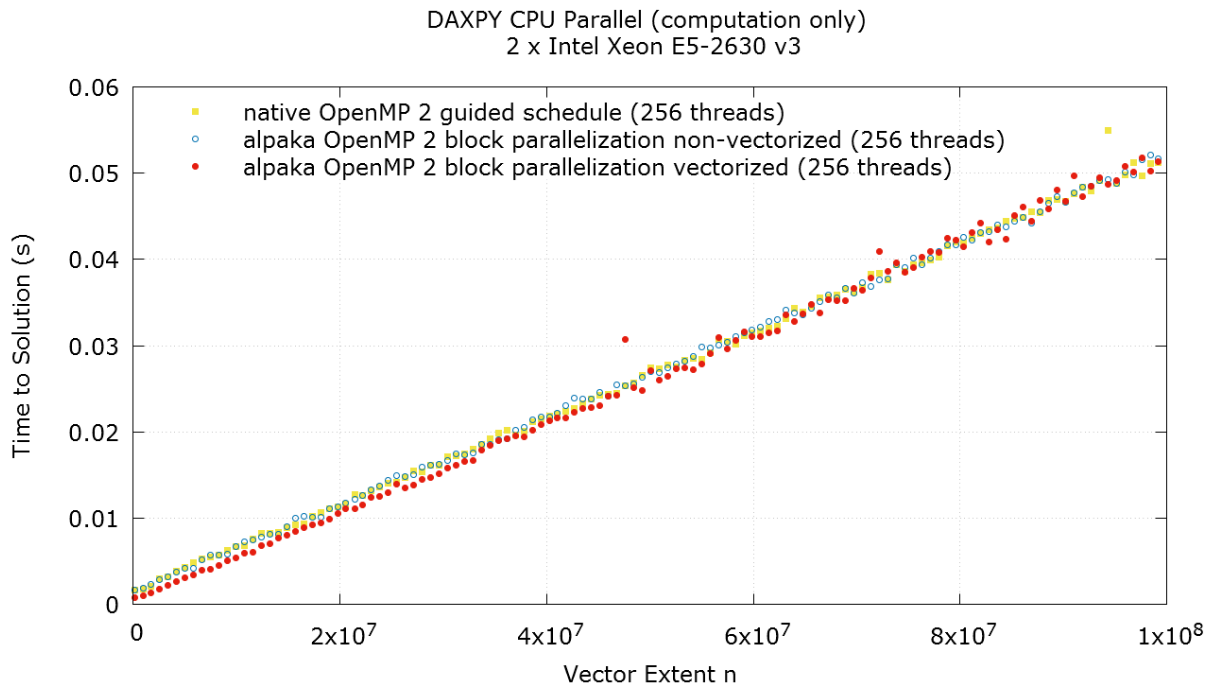

Figure B.2: Comparison of the execution time of the *alpaka OpenMP* and the native *OpenMP* generalized vector addition.

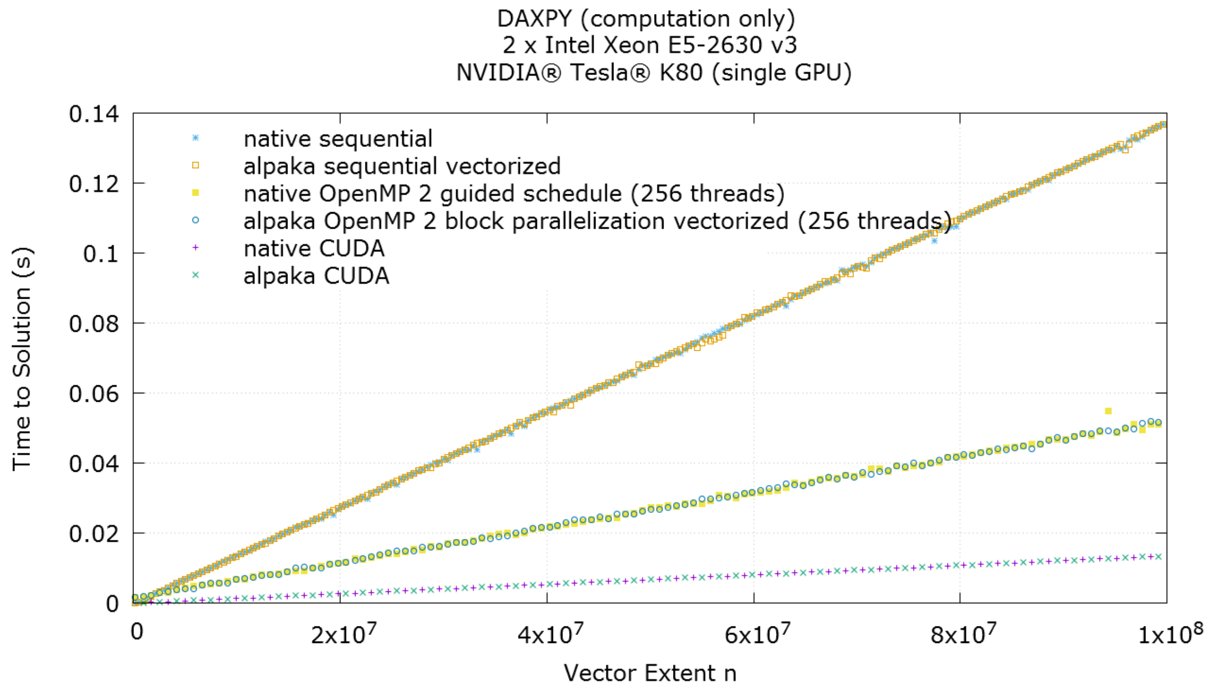

Figure B.3: Comparison of the time to solution of generalized vector addition variants.

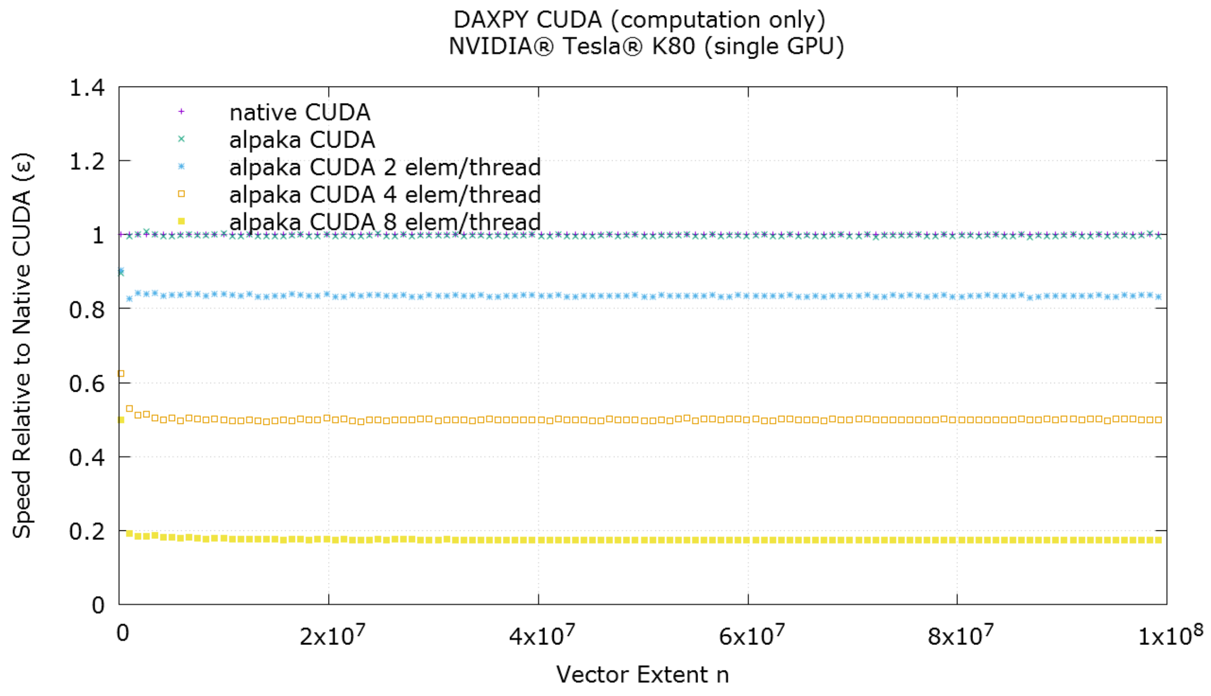

Figure B.4: Comparison of *alpaka CUDA* versions with varying element per thread counts relative to the native *CUDA* generalized vector addition.

B.2 Generalized Matrix-Matrix-Multiplication

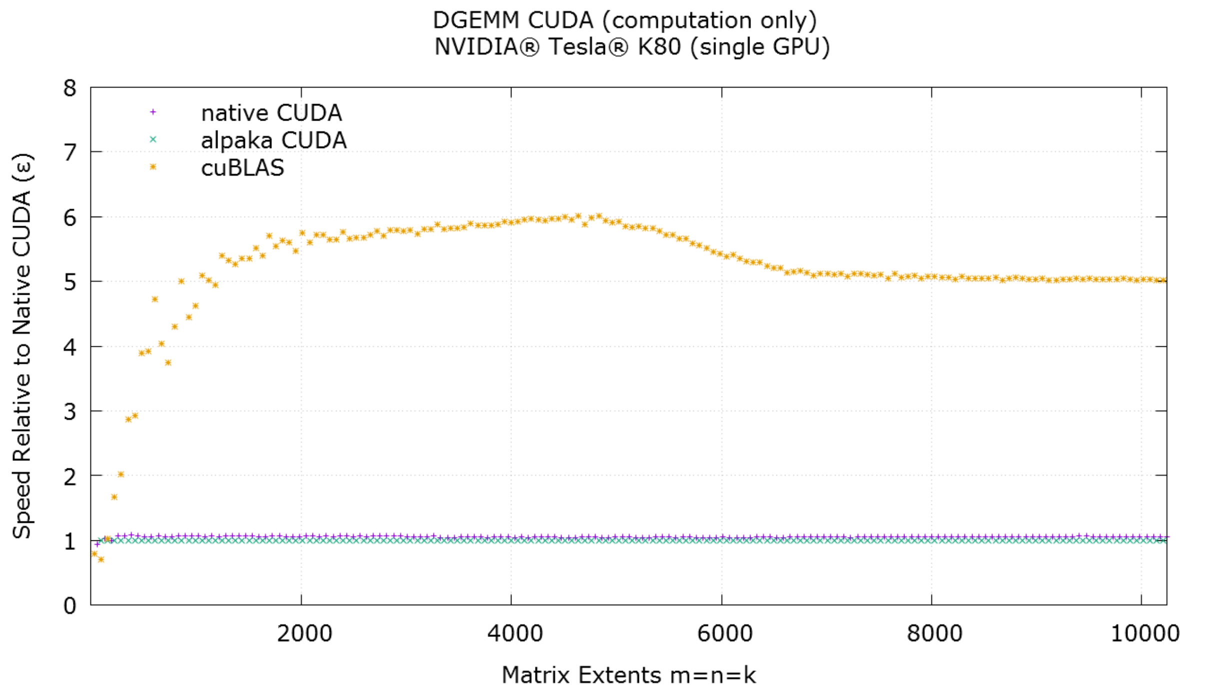

Figure B.5: Comparison of the *NVIDIA cuBLAS* and the *alpaka CUDA* relative to the native *CUDA* generalized matrix-matrix-multiplication.

Copyright Information