Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the Alpaka library*

Alexander Matthes[1,2], René Widera[1], Erik Zenker[3], Benjamin Worpitz[3], Axel Huebl[1,2], and Michael Bussmann[1]

[1] Helmholtz-Zentrum Dresden – Rossendorf
[2] Technische Universität Dresden
[3] LogMeIn, Inc.

Abstract. We present an analysis on optimizing performance of a single C++11 source code using the Alpaka hardware abstraction library. For this we use the general matrix multiplication (GEMM) algorithm in order to show that compilers can optimize Alpaka code effectively when tuning key parameters of the algorithm. We do not intend to rival existing, highly optimized DGEMM versions, but merely choose this example to prove that Alpaka allows for platform-specific tuning with a single source code. In addition we analyze the optimization potential available with vendor-specific compilers when confronted with the heavily templated abstractions of Alpaka. We specifically test the code for bleeding edge architectures such as Nvidia's Tesla P100, Intel's Knights Landing (KNL) and Haswell architecture as well as IBM's Power8 system. On some of these we are able to reach almost 50% of the peak floating point operation performance using the aforementioned means. When adding compiler-specific `#pragma`s we are able to reach 5 $^{\text{TFLOPs}}/_{\text{s}}$ on a P100 and over 1 $^{\text{TFLOPs}}/_{\text{s}}$ on a KNL system.

1 Introduction

1.1 Motivation

We have developed Alpaka [28] due to our own need in programming highly efficient algorithms for simulations [27] and data analysis on modern hardware in a

* This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 654220. This project received funding within the MEPHISTO project (BMBF-Förderkennzeichen 01IH16006C). Research leading to these results has in parts been carried out on the Human Brain Project PCP Pilot System JURON at the Juelich Supercomputing Centre, which received co-funding from the European Union (Grant Agreement no. 604102). We thank for the access to and support for the HPC cluster Taurus at the Centre for Information Services and High Performance Computing (ZIH), Technical University Dresden, as well as the cluster Hypnos at the Helmholtz-Zentrum Dresden – Rossendorf.

portable manner. The aim of our approach is to have a single C++ source code in which we can express all levels of parallelism available on modern compute hardware, using a parallel redundant hierarchy model similar to that found in CUDA or OpenCL. Taking a look at the recent top ten high performance computing (HPC) systems [16], it becomes clear that many-core architectures and heterogeneous systems are dominating the landscape and will continue to do so.

The main design goal of Alpaka is to describe all levels of parallelization available on modern heterogeneous hardware. It neither makes assumptions on the memory layout or access patterns, nor does it handle the underlaying resource and event management of the whole application, nor does it abstract the inter-node communication.

Our open-source projects PIConGPU [3,2] and HaseOnGPU [5] both use Alpaka for the kernel abstraction for various many-core hardware [27,28], but different libraries for the mentioned topics not handled by Alpaka, like Graybat [26] for the network communication, mallocMC for the memory management or libPMacc for containers and asynchronous event handling. Alpaka is not meant as full grown solution for developing or porting whole HPC applications, but as a single-purpose library that can easily be included into the individual software of an exiting HPC project. We have chosen to provide a lightweight, yet powerful C++ meta programming library for coherently expressing parallelism for a large variety of many-core platforms.

Modern C++11 template programming enables us to implement an abstraction layer between the application and the various, often vendor-specific programming models available for programming many-core hardware. With modern compilers the abstraction layer is completely resolved during compilation, leaving only efficient code in the binary.

While performance portability and close-to-zero overhead of Alpaka code could be shown in previous work [28] we will here concentrate on a subject important for high performance computing, namely optimization of code for various hardware platforms by means of tuning and vendor-specific compiler optimizations while maintaining a single-source, portable code. The presence of architecture independent parameters outside the algorithm implementation itself may also enable auto-tuning in a later step.

We will show that indeed parameter tuning and compiler optimization generate highly efficient code on various platforms. However, we will discuss some pitfalls of this approach that arise due to limiting tuning parameters to a small number and due to the lack of full support for C++11 in some vendor compilers.

1.2 Alpaka

Alpaka allows for a multidimensional, hierarchical abstraction of computation hardware as seen in Fig. 1. Kernels are written and run as threads executed in a task parallel manner. Threads are organized in *blocks*, which themselves are organized in *grids*. Every thread inside a block is assumed to run in parallel to the other threads in the same block, enabling intra-block synchronization. Blocks on the other hand may run concurrently or sequentially inside a grid. Every

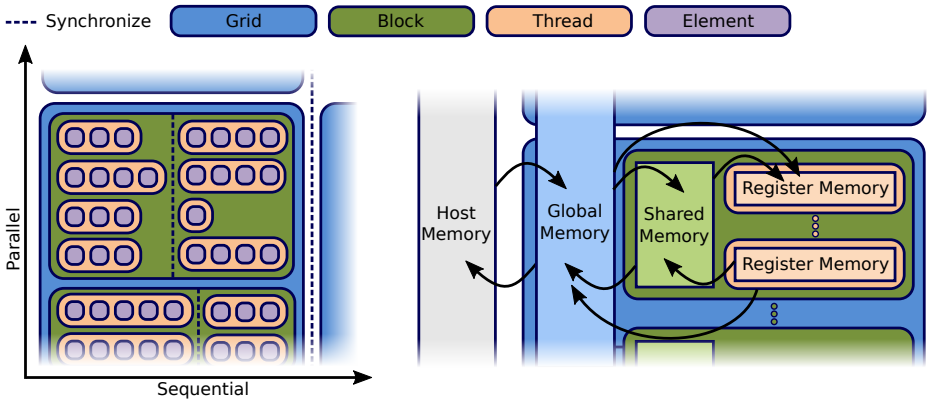

Fig. 1. Systematic view of the abstract redundant parallel hierarchy model of Alpaka taken from [28]. A compute device works on a grid, inside this grid are blocks. Every block has the same amount of threads, which are defined by the user as kernels. Explicit looping over elements inside the kernel enables autovectorization but also gives a better utilization for simple kernels. For each computation layer Alpaka introduces an appropriate memory layer. The copies between those are explicit (depicted as arrows).

execution layer has a corresponding memory hierarchy level. In addition to task-parallel execution Alpaka introduces an element layer inside the thread level for data-parallel execution, where the same instruction or program is executed for multiple data. This latter level is usually used for expressing vectorization.

For any given hardware, these layers are mapped onto the hardware using a suitable back end. As such, Alpaka does not implement any functionality beyond this mapping and the underlying optimizations come form the back end and mapping chosen for a specific hardware.

Alpaka currently supports Nvidia's CUDA [23], OpenMP [4] 2 and 4, Boost Fibers and C++Threads as back ends. Furthermore, we have started to add OpenACC [24] and Thread Building Blocks [11] (TBB) support, while support for AMD HIP [1] is foreseen for the near future. Alpaka has two accelerators using OpenMP 2: One is running blocks in a grid concurrently, the other one threads inside a block. For the first one only one thread per block is allowed. With the same constraint it is possible to run the code sequentially with Alpaka.

In the scope of this paper we will restrict ourselves to the OpenMP 2 Blocks and Nvidia CUDA back ends so that we are able to compare our new results to our previous work. Although OpenCL [7] is widely supported, it is not suitable as Alpaka back end, as it is not single source C++. SYCL [13,25] has the goal to close this gap and will probably be considered in the future. C++ AMP [17] looks similarly promising, but fails in support of current HPC architectures.

Alpaka leaves performance enhancements due to data layout to the user or another, independent library. Memory in Alpaka thus is always represented by a plain pointer. This strategy leaves room for optimization, but currently requires more development effort by the user.

Optimized memory access patterns are as important for achieving performance as expression of algorithmic parallelism and we have carefully chosen the example GEMM algorithm as it is simple enough to go without memory abstraction. However, optimizing memory access and memory copies is outside the scope of Alpaka, which distinguishes our approach from the design goals of libraries such as Kokkos [6] or RAJA [10] that aim for providing a full software environment for portable programming of many-core architectures. A separate memory access abstraction library is planned, but will be an independent, orthogonal part of the already mentioned software stack.

2 The Alpaka general matrix multiplication implementation

Similar to [28] we use a general matrix multiplication (GEMM) example

$$C = \alpha \cdot A \cdot B + \beta \cdot C \tag{1}$$

for performance tuning, as it allows for tiling without the need for changing the memory representation of the matrices.

For the sake of simplicity we choose A, B and C to be quadratic matrices with N rows and columns each. The total number of floating point operations then follows as

$$\mathrm{O}(N) = 3N^2 + 2N^3 \approx 2N^3 \ . \tag{2}$$

The number of elements per thread e and threads per block t result in the number of blocks in the grid

$$\mathrm{B}(e,t) = \frac{N}{t \cdot e} \ , \tag{3}$$

whereby $t = 1$ for the OpenMP 2 Blocks and the sequential accelerator.

We measure the time t in seconds for the run of the algorithm without copy operations to device memory, keeping the maximum over ten runs. With this we calculate the performance P in $^{\mathrm{GFLOPs}}/_{\mathrm{s}}$ as

$$\mathrm{P}(N,t) = \frac{\mathrm{O}(N)}{t} \cdot 10^{-9} = \frac{2N^3}{t} \cdot 10^{-9} \ . \tag{4}$$

2.1 Tiled GEMM algorithm

There exist many highly optimized GEMM implementations, reaching up to 90 % [14] of the theoretical peak performance. The solution depicted here is not intended to compete with these algorithms. Instead, it serves as a code example reasonably designed to exploit parallelism on many-core hardware. As such, it already achieves 20 % of the peak performance without tuning, which is a value regularly found in applications. In the following, we aim to show that Alpaka allows for platform specific tuning by parameter tuning without specializing the

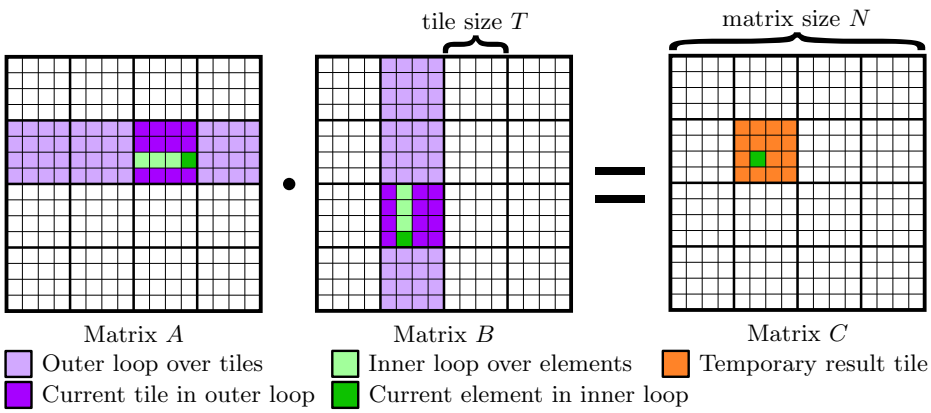

Fig. 2. Performance critical $A \cdot B$ part of the GEMM using a tiling strategy. A thread iterates over smaller sub matrices (tiles) in A and B (purple), performs the matrix multiplication per tile using the element layer (green) for vectorization, and adds it to a temporary thread local C tile (orange). The remaining part of the GEMM algorithm using the temporary C tile needs to load and write the C matrix only once (streaming), thus it doesn't need to be cached.

implementation. As long as the tiles of the two matrices A, B fit completely in the cache memory, increasing the tile size will usually result in better performance. Based on the size S in bytes of the data type used (single or double precision) the required cache size K is

$$\mathrm{K}(S,T) = 2T^2S \ . \tag{5}$$

The tiling matrix multiplication has $N_{\text{blocks}} = {}^{N}/_{T}$ tiles in each matrix dimension. For every tile of C N_{blocks} tiles of A and B need to be loaded (see Fig. 2). Furthermore the C tile itself needs to be loaded, leading a total number of

$$\mathrm{M}(N,T) = N_{\text{blocks}}^2(2T^2N_{\text{blocks}} + T^2) = 2\frac{N^3}{T} + N^2 = N^2\left(2\frac{N}{T} + 1\right) \tag{6}$$

memory operations, which gives us the ratio of compute to memory operations as

$$\mathrm{R}(N,T) = \frac{\mathrm{O}(N)}{\mathrm{M}(N,T)} = \frac{2N^3}{N^2(2\frac{N}{T}+1)} = \frac{2N}{(2\frac{N}{T}+1)} = \frac{2N}{\frac{2N+T}{T}} = \frac{2NT}{2N+T} \tag{7}$$

with $\lim_{N\to\infty} \mathrm{R}(N,T) = T$, showing again that larger tile sizes are preferable

With cache hierarchies present in most modern architectures, it is not trivially predictable for which cache T should be optimized. We thus chose to calculate one tile of the matrix C per Alpaka block. Every element in the block

```
// Class for optimal tile size depending on the Accelerator type
template< typename T_Acc >
struct OptimalVectorSize {
    using type = alpaka::dim::DimInt<1u>;
};
// Number of elements per tiles predefined, but changeable as compiler option
#ifndef GPU_ELEM_NUM
    #define GPU_ELEM_NUM 2u
#endif
#ifndef OMP_ELEM_NUM
    #define OMP_ELEM_NUM 256u
#endif
// Specialization of the tile size type for CUDA, steered by GPU_ELEM_NUM
#ifdef ALPAKA_ACC_GPU_CUDA_ENABLED
    template< typename... T_Args >
    struct OptimalVectorSize< alpaka::acc::AccGpuCudaRt< T_Args... > > {
        using type = alpaka::dim::DimInt<GPU_ELEM_NUM>;
    };
#endif
// Specialization for OpenMP Blocks, steered by OMP_ELEM_NUM
#ifdef ALPAKA_ACC_CPU_B_OMP2_T_SEQ_ENABLED
    template< typename... T_Args >
    struct OptimalVectorSize< alpaka::acc::AccCpuOmp2Blocks< T_Args... > > {
        using type = alpaka::dim::DimInt<OMP_ELEM_NUM>;
    };
#endif
// Easily extensible macro for every independent loop
#define VECTOR_PRAGMA \
    _Pragma ("ivdep") \
    _Pragma ("GCC ivdep")
```

Listing 1.1. Settings for the tiled matrix multiplication. `OptimalVectorSize::type::value` represents the tile size T. The parameters and the loop `#pragma`s can directly be used inside the kernel.

calculates one entry in the C tile. We use a two dimensional indexing for the parallelization levels. Every element stores the partial result of $\alpha \cdot A \cdot B$ in element local memory. Depending on the architecture, we can increase the number of elements per block by increasing the number of threads per block, which makes sense for GPUs, or the number of elements per thread, which should enable autovectorization for CPUs.

We implement the tile size T as an accelerator dependent class as seen in Listing 1.1, thus avoiding mixing tuning and kernel parameters. It is set via `#define`, thus making tuning easier. The matrix sizes N are passed as kernel parameters (not shown).

2.2 Architectures

We test Nvidia K80 and P100 GPUs. The K80 and the PCIe version of the P100 are hosted in the cluster Hypnos at the Helmholtz-Zentrum Dresden – Rossendorf whereas an nvlink using version of the P100 is part of the OpenPower pilot system JURON at the Jülich Supercomputing Center. All GPU architectures considered in this paper are listed in Tab. 1.

Vendor		Nvidia		
Architecture		K80	P100	
Interconnect to host		PCIe	nvlink	PCIe
Number of SMs		13 [21]	56 [22]	
Cores per SM	SP	192 [21]	64 [22]	
	DP	64 [21]	32 [22]	
Shared memory per SM		112 KB [21]	48 KB [22]	
Registers per SM (32 Bit)		131,072 [21] [22]		
Clock frequency		0.88 Ghz (Boost clock)	1.48 Ghz	1.39 Ghz
Theoretical peak performance	SP	4.37 TFLOPs/s [19]	10.6 TFLOPs/s [20]	9.3 TFLOPs/s [20]
	DP	1.46 TFLOPs/s [19]	5.3 TFLOPs/s [20]	4.7 TFLOPs/s [20]
Release date		Q4/2014	Q4/2016	

Table 1. Single (SP) and double (DP) precision peak performances and other characteristic variables of GPUs considered in this paper. Notice that the P100 connected via nvlink has a higher frequency and thus a higher theoretical peak performance. The K80 has two GPU chips on one board of which we use only one. The cores of GPUs are grouped in Streaming Multiprocessors (SMs) similar to CPU sockets.

As modern GPUs can directly access host CPU memory, we test both manual offloading and Nvidia unified memory. For the first case we do not measure the time for explicit memory transfer between CPU and GPU. Be aware that memory handling is not part of Alpaka and native vendor code is used when necessary. We thus focus on measuring algorithmic performance while disregarding analysis of e.g. efficient latency hiding when offloading code to an accelerator.

Intel Xeon E5-2680 v3 (Haswell) and Xeon Phi Knights Landing (KNL) architectures are hosted on the HPC cluster Taurus located at Technical University Dresden whereas the Power8 processor is also part of the HPC pilot system JURON. The CPU architectures considered in this paper are listed in Tab.2.

Clock frequency f, FLOP per cycle and core o, and number of cores n give the theoretical peak performance

$$\mathrm{P}(f, o, n) = f \cdot o \cdot n \ . \tag{8}$$

The Haswell CPU does not have hyperthreading activated and has two AVX units per core, which allows for instruction level parallelism and thus up to 64 single precision floating point operations (FLOPs) per cycle and clock. For measurements we use 2 sockets resulting in a total amount of 24 cores. The KNL architecture allows for up to 128 single precision floating point operations per cycle and core. With hyperthreading activated this architecture can be used similar to a multi-core CPU with 256 independent threads. The IBM Power8 processor has a uniquely high CPU frequency of 4 Ghz, but the lowest peak performance of all tested systems. However, with 8 hardware threads per core, 160 independent tasks can be executed without a context switch, allowing for high levels of parallelism.

We test different compilers for most architectures, see Tab. 3. The GNU compiler is used as a reference available for all architectures and for GPUs to compile the steering host code.

Vendor and architecture		Intel Xeon E5-2680 v3 (Haswell)	Intel Xeon Phi™ CPU 7210 (KNL)	IBM Power8
Used sockets		2	1	2
Total number of cores n		24	64	20
Hardware threads per core		1	4	8
Clock frequency f		2.1 Ghz (AVX base frequency [18])	1.3 Ghz	4.02 Ghz
FLOP per cycle and core o	SP	64 (2·AVX,FMA)	128 (2·AVX-512,FMA)	16 [9]
	DP	32 (2·AVX,FMA)	64 (2·AVX-512,FMA)	8 [9]
Theoretical peak performance (8)	SP	1.61 TFLOPs/s	5.33 TFLOPs/s	1.29 TFLOPs/s
	DP	0.81 TFLOPs/s	2.66 TFLOPs/s	0.64 TFLOPs/s
Cache sizes reducing the memory latency	L1	64 KB (core)		
	L2	256 KB (core)	1 MB (2 cores)	512 KB (core)
	L3	30 MB (socket)	–	80 MB (socket)
Release date		Q3/2014	Q2/2016	Q2/2014

Table 2. Single (SP) and double (DP) precision theoretical peak performances (see Eq. 8) and other characteristic variables of CPUs considered in this paper. Performance gains come mostly from vector operations and fused multiply adds, especially for Intel CPUs, and higher clock frequencies when running on Power8.

2.3 Single source code file vs. optimization

As pointer alignment and dependencies cannot be known at compile time, autovectorization needs some hints from developer side. As pointed out, applications or additional libraries can provide additional information on data types that fosters autovectorization when using Alpaka. We thus are forced to add compiler dependent `#pragma`s, namely `#pragma ivdep` and `#pragma GCC ivdep` for the Intel and GNU C++ compilers, respectively, in order to declare pointers inside loops as independent and executable in parallel. Furthermore, all memory is aligned to a multiplier of 64 with `__declspec(align(64))` (Intel) and `__attribute__ ((aligned (64)))` (GNU compiler), which makes it faster to load whole chunks of memory to vector registers on some architectures. As one cannot pass this information via function parameters, we also explicitly tell the compilers about this in the most time critical loop over the A and B tiles with `__assume_aligned` (Intel) and `__builtin_assume_aligned` (GNU).

XL C++ work around Alpaka is a very demanding C++ code and most compilers fully support C++11, with the exception of the IBM XL compiler. For this reason we move the most performance critical code, the matrix multiplication of tiles in A and B, to an extra C file for every XL test and compile all C code with the XL compiler, while the C++ code including all Alpaka abstractions is compiled with the GNU C++ compiler. This means that we are not testing XL's OpenMP implementation. With full C++11 support by the IBM compiler we expect similar to better performance than we see with this approach. This workaround currently breaks our single source goal and prevents code optimizations like code inlining, but still helps to improve performance compared to using just the GNU compiler.

	Intel Compiler	CUDA	XL Compiler	GNU Compiler
Haswell	`-Ofast -xHost` (Version: 17.0.0)	–	–	`-Ofast -mtune=native -march=native` (Version: 6.2)
KNL	`-Ofast -xHost` (Version: 17.0.0)	–	–	`-Ofast -mtune=native -march=native` (Version: 6.2)
Tesla P100	–	`--use_fast_math` (Version: 8.0.44)	–	`-mtune=native -march=native` (Version: 5.3, only host)
Tesla K80	–	`--use_fast_math` (Version: 8.0.44)	–	`-mtune=native -march=native` (Version: 5.3, only host)
Power8	–	–	`-O5` (Version: 14.01) **(Only for C!)**	`-Ofast -mtune=native -mcpu=native -mveclibabi=mass` (Version: 6.3)

Table 3. Compilers, compiler options, and compiler versions considered for every architecture in this paper. Every binary is compiled on the same system it is run on later, allowing for architecture- and system-aware compiler optimization.

KNL specific parameter settings The Intel KNL is programmable similarly to a CPU, but like an offloading acceleration device it brings its own dedicated memory called MCDRAM. Compared to the global RAM the latency is almost the same, but the bandwidth around five times higher with over 450 $^{\mathrm{GB}}/_{\mathrm{s}}$ ([12], p. 20). The Intel KNL supports three modes of accessing the MCDRAM: As a cache for RAM, directly accessed (flat memory) or a hybrid mode, where a part is used as cache and another part as flat memory. The first two modes are compared in performance, as they form opposite cases. The Intel KNL can furthermore be operated in different cluster modes, which may improve the cache latency. In this paper we restrict ourselves to using quadrant mode only.

Multidimensional parameter tuning We choose T and the number of hardware threads as tuning parameters before running scaling tests for different matrix sizes N. Tuning is performed for a fixed $N = 10240$ as a good compromise between runtime and problem size and further for an arbitrary $N = 7168$, thus avoiding effects only occurring at some certain combinations of parameters. After finding optimal parameter sets scaling tests with matrix sizes from $N = 1024$ up to $N = 20480$ with an increment of $\Delta N = 1024$ are performed. We repeat every measurement first 5 than 10 times, which in all cases yield the same maximum result. This shows that any effects visible are not due to statistics, and we thus refrain from averaging over more measurements.

3 Parameter Tuning

As hyperthreading is deactivated for the Haswell CPU and as we have found an efficient number of threads $e = 16^2$ for Nvidia GPUs in previous work, only the tile size T is used for tuning for these architectures, see Fig. 3. An obvious obser-

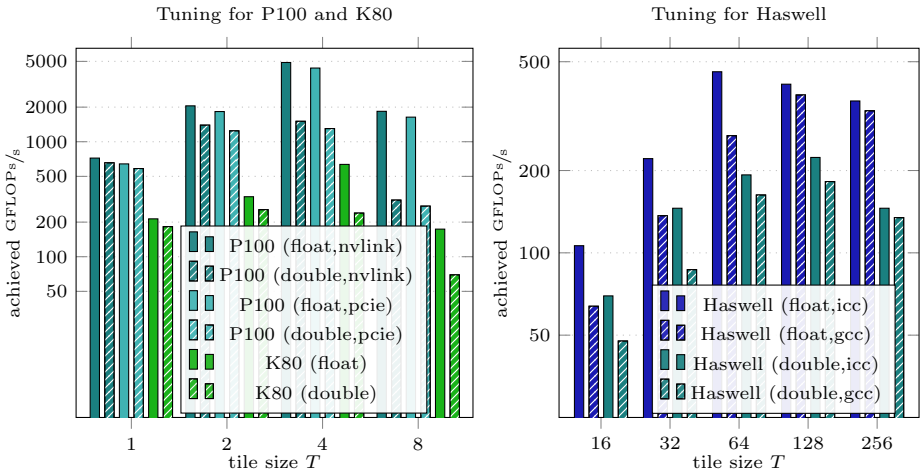

Fig. 3. Achievable GFLOPs/s for Nvidia K80 and P100, and for Intel Haswell depending on the compiler, the floating point precision and the chosen tile size of the GEMM algorithm. As there are not lesser cores than hardware threads, all of them are used.

vation for Haswell is that doubling the tile size often also doubles the achieved performance, while $T = 4$ seems to be optimal for current GPU generations.

Tuning for KNL and Power8 adds the number of hardware threads as a second parameter, see Fig. 4 for KNL. We see that optimal parameter combinations highly depend on the chosen precision and compiler. The double precision binary created by the Intel compiler using a single hardware thread results in best performance of 510 GFLOPs/s. We also do a measurement for the KNL in flat memory mode directly using the MCDRAM instead of the caching mechanism. Except for a slightly better performance ($\sim 2\%$), the results are the same.

For Power8 we test from $T = 16$ up to $T = 512$ and from one to eight hardware threads always using only powers of two as parameters similar to KNL (not shown). Contrary to KNL, optimization for the Power8 architecture deliver similar performance results for a variety of parameters even when using the IBM XL compiler. We don't see large deviations from our tuning results for the control case $N = 7168$ (not shown) on all architectures. Although bigger matrix sizes improve the GFLOPs/s slightly, optimum parameters remain the same.

Tuning results are found in Tab. 4, while the corresponding mapping of Alpaka parallel hierarchies to hardware in the case of double precision and vendor compilers selected is presented in Fig. 5.

4 Results of the scaling

Fig. 6 and 7 show the achieved GEMM GFLOPs/s for all architectures considered, for both double and single precision and optimum parameter sets [15]. The Nvidia P100 as expected shows the best absolute performance in all cases, while

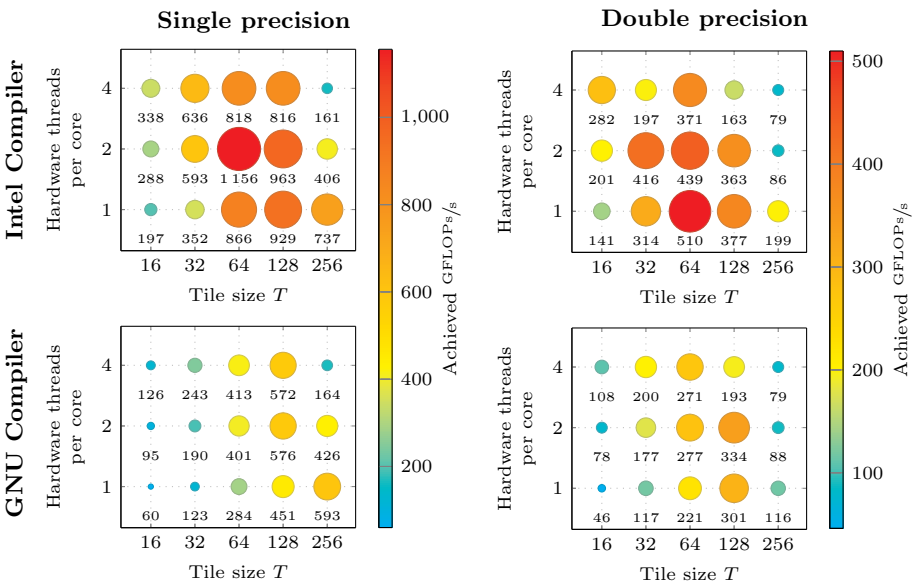

Fig. 4. Achievable GFLOPs/s for Intel Xeon Phi Knights Landing (KNL) depending on the compiler, the floating point precision, the chosen tile size of the tiled matrix multiplication algorithm and the used hardware threads per core. The bigger the mark size the higher the achieved GFLOPs/s. The mark radius is calculated with (achieved GFLOPs/s)$^{5/7}$ as this has been shown a good value for human perception [8]. GNU compiler 6.2 and Intel compiler 17 are used. For compiler options see Tab. 3.

the Power8 runtime is surprisingly faster than the K80 although the Nvidia GPU has a higher theoretical peak performance than the IBM CPU. The KNL shows a drastic drop in peak performance every second or fourth measurement beginning with $N = 8192$ for both precisions, regardless of using cached or flat memory when using the Intel compiler. To investigate this issue a test with $N = 8192$ is run in double precision but 91 hardware threads. With this we get 490 GFLOPs/s instead of 303 GFLOPs/s (64 threads), which is only 7% less than for $N = 7168$ and $N = 9216$ (both 527 GFLOPs/s).

Most architectures show an increase in the performance for higher N, with the exception of Intel Haswell which for single precision shows best peak performance (665 GFLOPs/s) for $N = 2048$ and afterwards decreases reaching a plateau at 400 GFLOPs/s. In contrast to our expectations, all GPUs show a better performance when using unified memory instead of device memory, especially for small N.

In order to compare results Fig. 8 shows the relative peak performance for the best parameter combinations for every architecture and single and double precision. For architectures investigated in 2016 [28], we find similar or only slightly better performance. But whereas the last paper has stated a general performance around 20% the most recent systems are now capable to reach almost 50% of the peak performance using Alpaka.

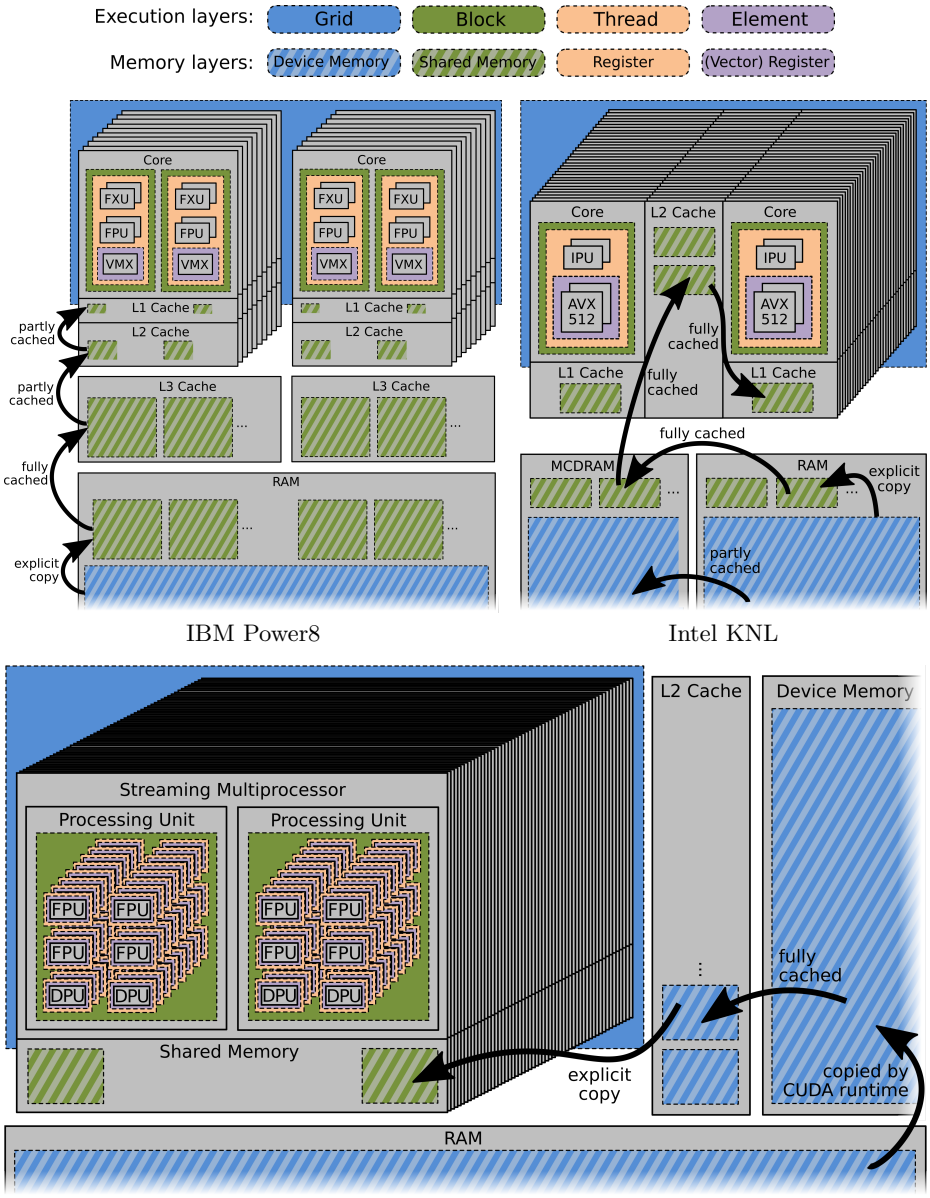

Fig. 5. Alpaka mappings for IBM's Power8, Intel's KNL, and Nvidia's Tesla P100. Every mapping uses the optimal parameters of the parameter tuning for double precision and the vendor compiler from Tab. 4. The CPU mappings use the OpenMP2 Block back end. The GPU mapping uses the CUDA back end and unified memory.

Architecture	Compiler	Precision	HW Threads	Optimized tile size T	K(S,T) (see (5))	Cache per HW thread L1	L2	L3
P100 (nvlink)	CUDA	single	–	4	128 B	–	–	–
		double		4	256 B			
P100 (pci)		single		4	128 B			
		double		4	256 B			
K80		single		4	128 B			
		double		2	64 B			
Haswell	Intel	single	1	64	32 KB	64 KB	256 KB	2.5 MB
		double		128	256 KB	64 KB	256 KB	2.5 MB
	GNU	single		128	128 KB	64 KB	256 KB	2.5 MB
		double		128	256 KB	64 KB	256 KB	2.5 MB
KNL	Intel	single	2	64	32 KB	32 KB	256 KB	–
		double	1	64	64 KB	64 KB	512 KB	
	GNU	single	1	256	512 KB	64 KB	512 KB	
		double	2	128	256 KB	32 KB	256 KB	
Power8	XL	single	2	512	2 MB	32 KB	256 KB	4 MB
		double	2	512	4 MB	32 KB	256 KB	4 MB
	GNU	single	8	256	512 KB	8 KB	64 KB	1 MB
		double	4	256	1 MB	16 KB	128 KB	1 MB

Table 4. Estimated optimal tile size T and number of hardware (HW) threads. Memory for A and B tiles K(S,T) (Eq. 5) and the available cache per HW thread and cache level are listed in addition. The first cache level that can hold a complete tile is marked.

5 Analysis

Autovectorization Listing 1.2 shows the dissembled KNL binary built by the Intel compiler for the most inner and performance critical loop of the tiling matrix multiplication kernel. C++ code is marked blue, assembler code red. With `vfmadd231pd` being the fused multiply add vector function working on AVX-512 vectors (`zmm*`) and loop unrolling we find that the Intel compiler is capable of optimizing the inner loop despite the heavy templated Alpaka code.

Parameter tuning We assume that tuning for KNL resulted in best FP performance using one hardware thread (see Fig. 4) because larger tiles then fit best into the L2 cache of 512 KB, which otherwise would have to be shared between threads. This is supported by the fact that using double precision often requires smaller tile sizes than single precision. Fig. 3 shows the element layer with $T = 4$ causing performance gain, especially for the P100, as it has more shared memory and registers available per thread than the K80.

Scaling Most architectures show poor performance for small matrix sizes $N \leq 2048$ which at first glance could be blamed on under-utilization, although at closer look is questionable e.g. in case of the KNL which performs 2×10^9 floating point operations that clearly dominate over memory operations following Eq. 7.

We found the KNL in flat memory mode to be only about $\sim 2\%$ faster than in cached memory mode, except for very small N, which can be explained by the fact that the same memory is needed very often, but needs to be copied from

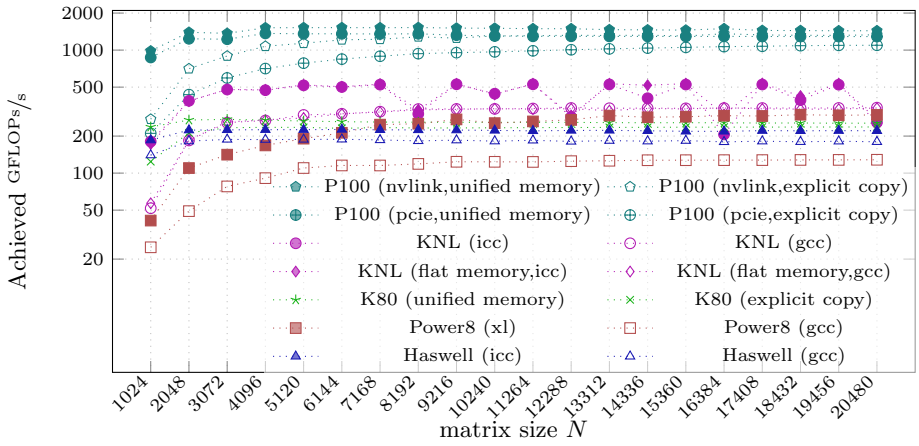

Fig. 6. Achievable GFLOPs/s for all considered architectures for double precision depending on the matrix size and the compiler.

RAM to MCDRAM only once. In all cases, using RAM only is much slower that using MCDRAM. We see performance degradation on KNL for (almost) every second N (DP) and for every fourth N (SP) starting with $N = 8192$, except for $N = 14336$ (flat memory, DP). When choosing an uneven number of 91 hardware threads, performance improves for $N = 8192$ (DP). As the issue always appears on very even numbers we assume that the KNL has performance issues if many hardware threads access the very same memory location at the same time. As this issue does not occur for the GNU compiler, we suspect Intel's optimized OpenMP implementation to cause this.

The K80's relative peak performance is only around 15% for single precision (SP) and around 18% for double precision (DP) whereas the P100 reaches 46% (SP) and 28% (DP). As loading to shared memory is not optimally realized, we attribute this difference to the P100 having more registers per thread and more shared memory than the K80, thus more blocks can run concurrently which better hides memory latencies. Although SP values need half the space of DP the K80 has three times more SP units than DP, thus the SP version needs to load more memory for all scheduled blocks, which leads to performance degradation, which is not the case for the P100 with only two times more SP than DP units. Another problem of the algorithmic implementation (but not of Alpaka) is that the index arithmetics lead to a unfavorable ratio of integer to floating point operations, thus degrading FPU utilization. We emphasize that platform-dependent memory access optimizations are within the responsibility of the user code when using Alpaka.

The Haswell architecture shows a different behavior than all other systems for SP where the peak performance has its peak at $N = 2048$ and then slowly decreases. For $N = 2048$ matrices A and B use only 32 MB which fits into the L3 cache of one Haswell CPU (see Tab. 3 2), thus accelerating memory access.

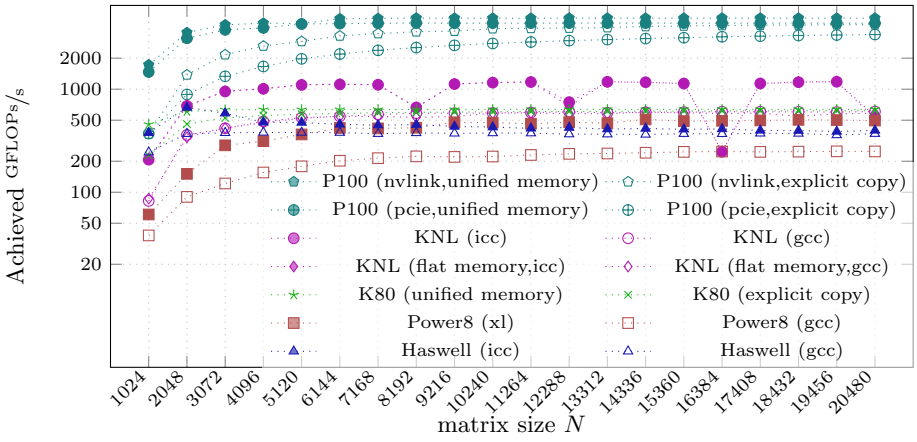

Fig. 7. Achievable GFLOPs/s for all considered architectures for single precision depending on the matrix size and the compiler.

6 Conclusion

Within the scope of this work we have shown that portable single-source C++11 code using Alpaka can run on current many-core architectures without changing any line inside the algorithmic relevant part of the source code, seeing good floating point performance for the most recent systems when reasonably designing the code for exploiting many-core parallelism. We find that optimizing the number of hardware threads and the tile size for a simple GEMM algorithm leads to considerable increase in performance that can be well explained by the architectural characteristics and is independent of the Alpaka abstractions.

This becomes evident when analyzing the effects of vendor-specific compiler optimization. These do not only show that expected optimizations such as autovectorization, loop unrolling and the use of fused multiply adds are performed using Alpaka but that for bleeding edge hardware like Intel KNL, Nvidia P100 and IBM Power8 using vendor compilers gives a significant boost in performance.

When using vendor-specific compilers with appropriate optimization flags and `#pragma` statements we are able to come close to 50% of the expected peak floating point performance on the Nvidia P100 and IBM Power8 architectures, and in addition could increase the performance on well known architectures like Haswell by about five percentage points. We can thus conclude that the abstract parallel redundant hierarchy interface introduced by Alpaka does not prevent compiler optimization and tuning. However, we also find that the performance gains observed heavily depend on the target architecture and software environment available. We express our hope that the implementation of modern C++ support in compilers relevant for high performance computing will foster the approach we take to performance portability with Alpaka.

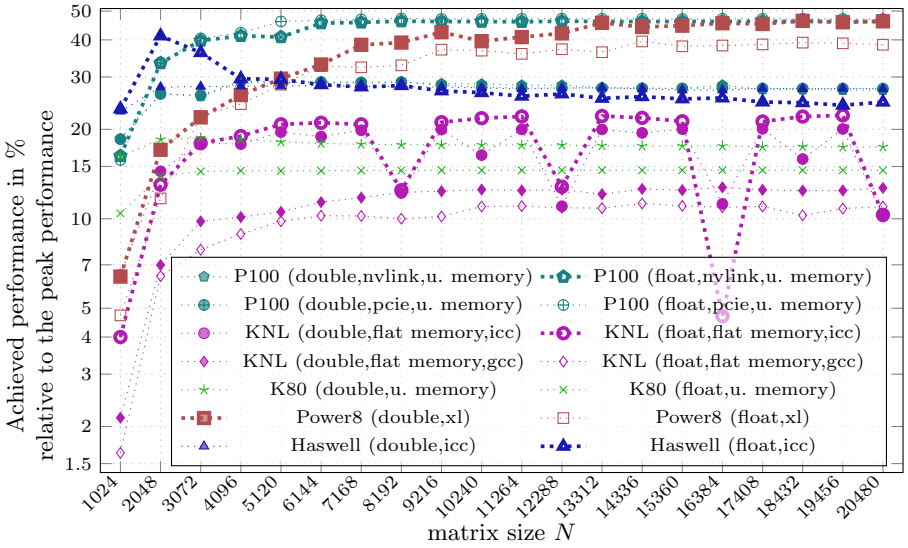

Fig. 8. Achieved performances relative to the peak performance for the fastest parameter combinations of every architecture for single and double precision. Some scalings of particular interest are highlighted.

Our analysis shows that for some architectures such as Intel's KNL more tuning parameters have to be included in order to achieve optimum results for certain problem sizes when optimizing with vendor-specific compilers. For future applications this potentially increases the time it takes for tuning a code, making tuning itself a compute- and memory-intensive task.

We clearly find that most modern vendor-specific compilers, with the prominent exception of IBM's XL compiler, are able to create highly optimized code for their target architecture from the Alpaka GEMM implementation. This shows that with Alpaka writing abstract, single-source C++ code with close-to-zero overhead is possible on todays high performance many-core architectures, demonstrating that code abstraction for sake of portability and architecture-specific tuning do not contradict each other.

References

1. AMD: HIP DATA SHEET - It's HIP to be Open. `https://gpuopen.com/wp-content/uploads/2016/01/7637_HIP_Datasheet_V1_7_PrintReady_US_WE.pdf` (Nov 2015), [Online; accessed April 11, 2017]
2. Burau, H., Widera, R., Honig, W., Juckeland, G., Debus, A., Kluge, T., Schramm, U., Cowan, T.E., Sauerbrey, R., Bussmann, M.: Picongpu: A fully relativistic particle-in-cell code for a gpu cluster. IEEE Transactions on Plasma Science 38(10), 2831–2839 (2010)

```
1  for( TSize j(0); j < numElements; ++j )
2  {
3          lineC[j] += a * lineB[j];
4     422a5e: 62 a2 ed 40 b8 0c 18  vfmadd231pd (%rax,%r11,1),%zmm18,%zmm17
5     422a65: 62 a2 ed 40 b8 44 18  vfmadd231pd 0x40(%rax,%r11,1),%zmm18,%zmm16
6     422a6c: 01
7     422a6d: 62 32 ed 40 b8 7c 18  vfmadd231pd 0x80(%rax,%r11,1),%zmm18,%zmm15
8     422a74: 02
9     422a75: 62 32 ed 40 b8 74 18  vfmadd231pd 0xc0(%rax,%r11,1),%zmm18,%zmm14
10    422a7c: 03
11    422a7d: 62 32 ed 40 b8 6c 18  vfmadd231pd 0x100(%rax,%r11,1),%zmm18,%zmm13
12    422a84: 04
13    422a85: 62 32 ed 40 b8 64 18  vfmadd231pd 0x140(%rax,%r11,1),%zmm18,%zmm12
14    422a8c: 05
15    422a8d: 62 b2 ed 40 b8 64 18  vfmadd231pd 0x180(%rax,%r11,1),%zmm18,%zmm4
16    422a94: 06
17    422a95: 62 b2 ed 40 b8 5c 18  vfmadd231pd 0x1c0(%rax,%r11,1),%zmm18,%zmm3
18    422a9c: 07
```

C++ code

AVX-512 register

Fused multiply add

Unrolled assembler code

Listing 1.2. Dissambled output of `objdump -DSC` for the most inner and performance critical loop of the tile matrix multiplication kernel. It shows that loop unrolling, vectorization and fused multiply add are realized by the compiler.

3. Bussmann, M., Burau, H., Cowan, T.E., Debus, A., Huebl, A., Juckeland, G., Kluge, T., Nagel, W.E., Pausch, R., Schmitt, F., Schramm, U., Schuchart, J., Widera, R.: Radiative signatures of the relativistic kelvin-helmholtz instability. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 5:1–5:12. SC '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2503210.2504564
4. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering 5(1), 46–55 (1998)
5. Eckert, C., Zenker, E., Bussmann, M., Albach, D.: Haseongpu – an adaptive, load-balanced mpi/gpu-code for calculating the amplified spontaneous emission in high power laser media. Computer Physics Communications 207, 362–374 (2016)
6. Edwards, H.C., Trott, C.R.: Kokkos: Enabling performance portability across manycore architectures. In: 2013 Extreme Scaling Workshop (xsw 2013). pp. 18–24. IEEE (2013)
7. Group, K.: The opencl specification - Version 2.1. https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf (11 Nov 2015), [Online; accessed March 23, 2017]
8. Gumhold, S.: Lecture "Scientific Visualization" (2011)
9. Hernandez, O.: Overview of the Power8 Architecture. https://indico-jsc.fz-juelich.de/event/24/session/24/contribution/0/material/slides/ (2016), [Online; accessed March 24, 2017]
10. Hornung, R., Keasler, J., et al.: The raja portability layer: overview and status. Lawrence Livermore National Laboratory, Livermore, USA (2014)
11. Intel Corporation: Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/, [Online; accessed April 12, 2017]
12. Jeffers, J., Reinders, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming Knights Landing Edition. Morgan Kaufmann (1 Jul 2016)

13. Khronos OpenCL Working Group SYCL subgroup: Sycl specification - Version 1.2. https://www.khronos.org/registry/sycl/specs/sycl-1.2.pdf (8 May 2015), [Online; accessed March 23, 2017]
14. Li, J., Li, X., Tan, G., Chen, M., Sun, N.: An optimized large-scale hybrid dgemm design for cpus and ati gpus. In: Proceedings of the 26th ACM international conference on Supercomputing. pp. 377–386. ACM (2012)
15. Matthes, A., Widera, R., Zenker, E., Worpitz, B., Hübl, A., Bussmann, M.: Matrix multiplication software and results bundle for paper "Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the Alpaka library" for P^3MA submission (Apr 2017), https://doi.org/10.5281/zenodo.439528
16. Meuer, H.W., Strohmaier, E., Dongarra, J., Simon, H., Meuer, M.: November 2016 — TOP500 Supercomputer Sites (November 2016)
17. Microsoft Corporation: C++ amp : Language and programming model - Version 1.2. http://download.microsoft.com/download/2/2/9/22972859-15c2-4d96-97ae-93344241d56c/cppampopenspecificationv12.pdf (Dec 2013), [Online; accessed March 23, 2017]
18. Newman, B.: Intel Xeon E5-2600 v3 "Haswell" Processor Review — Microway. https://www.microway.com/hpc-tech-tips/intel-xeon-e5-2600-v3-haswell-processor-review/ (8 Sep 2014), [Online; accessed March 24, 2017]
19. Nvidia: Tesla K80 HPC and Machine Learning Accelerator. https://www.nvidia.com/object/tesla-k80.html (2014), [Online; accessed March 23, 2017]
20. Nvidia: Tesla P100 Most Advanced Data Center Accelerator. https://www.nvidia.com/object/tesla-p100.html (2016), [Online; accessed March 23, 2017]
21. Nvidia Corporation: NVIDIAs Next Generation - CUDA Compute Architecture: Kepler GK110/210. Whitepaper (2014)
22. Nvidia Corporation: NVIDIA Tesla P100 - The Most Advanced Datacenter Accelerator Ever Built. WP-08019-001_v01.1 (May 2016)
23. NVIDIA Corporation: NVIDIA CUDA C Programming Guide Version 8.0. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (January 2017), [Online; accessed March 23, 2017]
24. OpenACC-Standard.org: The OpenACC Application Programming Interface - Version 2.5. http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf (Oct 2015), [Online; accessed March 23, 2017]
25. Wong. Michael, Andrew, R., Rovatsou, M., Reyes, R.: Khronos's OpenCL SYCL to support Heterogeneous Devices for C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0236r0.pdf (12 Feb 2016), [Online; accessed March 23, 2017]
26. Zenker, E.: Graybat - Graph Approach for Highly Generic Communication Schemes Based on Adaptive Topologies (5 Mar 2016), https://github.com/ComputationalRadiationPhysics/graybat
27. Zenker, E., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W.E., Bussmann, M.: Performance-portable many-core plasma simulations: Porting picongpu to openpower and beyond. In: International Conference on High Performance Computing. pp. 293–301. Springer (2016)
28. Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W.E., Bussmann, M.: Alpaka–an abstraction library for parallel kernel acceleration. In: Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International. pp. 631–640. IEEE (2016)