International Journal on Advances in Systems and Measurements, vol 9 no 1 & 2, year 2016, http://www.iariajournals.org/systems_and_measurements/

Performance Characterization of Multiprocessors and Accelerators Using Micro-Benchmarks

Javed Razzaq, Rudolf Berrendorf, Jan Philipp Ecker, Simon Eric Scholl
Computer Science Department
Bonn-Rhein-Sieg University of Applied Sciences
Sankt Augustin, Germany
e-mail: {javed.razzaq, rudolf.berrendorf, jan.ecker, simon.scholl}@h-brs.de

Florian Mannuss
EXPEC Advanced Research Center
Saudi Arabian Oil Company
Dhahran, Saudi Arabia
e-mail: florian.mannuss@aramco.com

Abstract—**In this paper, a set of micro-benchmarks is proposed to determine basic performance parameters of single-node mainstream hardware architectures for High Performance Computing. Performance parameters of recent processors, including those of accelerators, are determined. The investigated systems are Intel server processor architectures and the two accelerator lines Intel Xeon Phi and Nvidia graphic processors. Additionally, the performance impact of thread mapping on multiprocessors and Intel Xeon Phi is shown. The results show similarities for some parameters between all architectures, but significant differences for others.**

Keywords*–*Performance; micro-benchmarks; server processors; Intel Xeon Phi; Nvidia graphic processors; thread mapping

I. INTRODUCTION

For resource-intensive computations in High Performance Computing (HPC) on a single node level, a good performance can only be achieved if the performance characteristics of the processor, memory and core-core/core-memory interconnect architecture are understood. First investigations on that have been published in [1]. Finding and quantififying such characteristics for a certain system is motivated by the fact that, for most of their runtime, HPC applications stress only parts of the hardware in compute-intensive program kernels. Examples are compute-bound problems – such as direct linear solvers [2] that are bound by the floating point capability of a system, and memory-bandwidth-bound problems – such as the multiplication of a sparse matrix with a dense vector in iterative solvers [3], [4], [5]. Other application kernels may be bound differently.

This paper proposes a set of micro-benchmarks to characterize HPC hardware on a single-node level. The results of the micro-benchmarks are performance parameters related to performance bounds found in many computational kernels (see [6] for two such parameters). These parameters often allow conclusions to be drawn on the (at least relative) performance of real applications or performance critical application kernels of certain classes that are bound by one or few of those parameters. Additionally, if carefully chosen, architecture-bottlenecks can be revealed. The micro-benchmarks were chosen to allow conclusions on an application level rather than to evaluate deep structures in a processor architecture with sophisticated low-level programs, as for example in [7].

The proposed micro-benchmarks are applied to representatives of different classes of current hardware architectures. Results show similarities in performance between all architectures for some parameters (e.g., reaching near peak floating point performance for dense matrix multiply), but also significant differences between architectures (e.g., main memory latency and bandwidth). Consequently, only certain application classes are suitable for a specific architecture.

The paper is structured as follows. The following section discusses related work. Then, current mainstream HPC hardware architectures are briefly described, focusing on their differences. Section IV contains a description of the proposed micro-benchmarks. Section V describes our experimental setup and finally, in Section VI and Section VII, detailed evaluation results are presented and discussed, followed by a conclusion in Section VIII.

II. RELATED WORK

Benchmarks are widely used to evaluate certain performance properties of computer systems. A benchmark should thus be usable as an indicator that can support a decision, e.g., whether this system is feasible for a certain task or not. A multitude of different benchmarks exist, dependent on the question to be answered.

The Top500 list [8] uses the High Performance Linpack [2] to rank (very) large parallel systems. This benchmark produces only a single value, the Floating Point Operations per second (FLOP/s) for just one specific task, the direct solution of a very large dense linear system.

The widely used SPEC CPU benchmark [9] is a mix of several real world application programs for integer-dominant computations or floating point dominant applications. Running the benchmark on a system produces one number for each class, a performance factor. These two numbers then express a *relative* performance improvement compared to an older base system with respect to integer performance and floating point performance.

Williams et al. introduced the roofline model [6] to describe the expectable performance space in a resource-bound problem. The two resources in this model are computational density (operations per transfered byte) and peak floating point performance. This is an example where two limitating parameters on a system are used to show eligible perfomance values.

The NAS Parallel Benchmarks [10] are more application-oriented benchmarks. These benchmarks consist of larger compute-intensive kernels and were originally designed to test large parallel computers. Each of these applications in this benchmark represents a different computing aspect. The applications include, for example, Conjugate Gradient (irregular memory access), Multi-Grid (long- and short-distance commu-

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

nication), or fast Fourier Transform (all-to-all communication). These benchmarks have been, amongst others, implemented in OpenMP [11] and recently in OpenCL [12]. With the OpenCL extension they can be used to measure recent accelerators, such as Graphic Processor Units (GPUs).

While the Linpack and the NAS Parallel Benchmarks aim for rather scientific and computing intense applications, the Graph500 benchmark [13] aims for data intensive applications, which can arise in other fields such as social networks or cyber security. Here, graph operations such as parallel Breadth First Search or Single Source Shortest Path problems are considered. A sequential reference implementation and parallel implementations with OpenMP, XMT and MPI are given by the Graph500 committee [13]. Furthermore, various other implementations, e.g., with PGAS [14] or hybrid approaches [15], exist.

For a finer granularity, benchmarks that give individual results for several operational classes can be used. An example is the OpenMP micro-benchmark suite [16], [17] that gives a developer a measure of how well basic constructs of OpenMP [18] map to a given system. If a developer knows important parameters that mainly determine the overall performance of an application in this programming model, he is able to estimate how well his own application will perform on the system using these basic constructs.

In [19], Treibig et al. presented the likwid-bench micro-benchmark tool as part of the performance monitoring and benchmark suite likwid [20]. This benchmark tool works on an assembler level measuring streaming loop kernels. It consists of several default benchmarks and offers the possibility to add custom bemchmarks. The default benchmarks cover memory copy, load and memory bandwith. In [21], Hofmann et al. used the likwid-bench tool along with the performance counter tool of likwid, to give detailed insights on the current Intel Haswell CPUs. For this purpose, micro-benchmarks for the current instruction set of Haswell CPUs, e.g., fused multiply add (FMA), where added.

Lemeire et al. used micro-benchmarks and a modified roofline model to characterize current GPUs [22]. These benchmarks were tailored to address the specific architecture of GPUs. The benchmarks were implemented in OpenCL and results of a AMD Cayman GPU and a Nvidia Maxwell GPU where presented in the publication.

Other micro-benchmark suites, which aim for a finer granularity are proposed in [23], [24], [25]. These benchmark suites are based on OpenCL. Here, OpenCL is used to compare memory-related issues, low level floating point operations and real life applications on different hardware architectures, including accelerators.

Directly related to the memory performance are the well-known Stream benchmark for memory bandwidth [26] and papers that work on an even finer granularity taking coherence protocols in certain architectures into account [7], [27], [28].

III. Current Hardware Architectures

This section gives a very brief overview on current HPC processor architectures and memory technology. It is partitioned into sections on mainstream HPC processor architectures, HPC accelerator architectures and memory technologies.

A. *Processor Architectures*

We concentrate on the Intel Xeon EP line of current HPC relevant processors, as these processors are used in nearly all new systems in the HPC computer Top500 list [8]. Intel's recent micro-architectures are Sandy Bridge EP (SB), and its successor Ivy Bridge EP (IB). The lastest change in the architecture appeared late 2014 in the Haswell EP processors (HW). A detailed description of the architectures is given in the manufacturer's related literature [29].

Processors nowadays have several cores. In HPC clusters, multiprocessor nodes with 2 processors are often used. Keeping multiple core-private caches coherent is usually done in the hardware by cache coherence protocols. Keeping caches coherent costs latency, bandwidth and may also influence an architecture's scalability [7], [28].

B. *Accelerator Architectures*

Computations of certain application classes can be accelerated using special attached processors. Nvidia graphic processors (GPU) and Intel Many Integrated Core processors (MIC) of the Xeon Phi family are currently predominant in HPC [8].

A Nvidia GPU has a hierarchical design (CUDA architecture [30]) that differs from that of common CPUs. The execution units (SE, Streaming Processors) are organized in multiprocessors, called Streaming Multi-Processors (SM or SMX), and a GPU has several such multiprocessors. For example, the Kepler family of GPUs has up to 15 SMX and 192 SE per SMX, resulting in a total of 2880 SE in the largest device configuration. These execution units are always used by a group of 32 threads, called a warp. Such an architecture leads to several aspects that have to be respected in performance critical programs, e.g., coalesced memory access and thread divergence [4], [31].

An Intel Xeon Phi coprocessor [32] consists of multiple CPU-like cores. The current generation Xeon Phi Knights Corner (KNC) has between 57 and 61 such cores, which are connected via a bi-directional ring bus. To achieve good performance on a Xeon Phi, the application must use parallelism as well as vectorization. In [33], requirements for vectorization are specified for the usage of the Intel compiler, e.g., no jumps and branches in a loop.

Recent accelerators (i.e., GPU as well as Xeon Phi) are plugin cards connected to the host through a PCI Express (PCIe) adapter. This adapter is often a severe bottleneck, because the transfer rate through a PCIe connection is significantly lower (8 GB/s for PCIe 2.0 x16 and 16 GB/s for PCIe 3.0 x16) than, for example, memory transfer rates in a host system.

C. *Memory Technologies*

Memory Technologies are optimized for different aspects. DDR3 / DDR4 RAM, which is used in CPU-based systems, is optimized for a short latency time. However, GDDR5 memory, which is used in accelerators, is optimized for bandwidth. This difference is important, as the performance of accelerators mainly comes from Single Instruction Multiple Data (SIMD) parallelism [34], where the same instruction is applied concurrently to multiple data items. These data items have to be fed to the functional units in parallel, asking for high main memory bandwidth.

All processors discussed here, including recent GPUs, use caches to speed up memory accesses. While GPUs currently have at most a 2 level cache hierarchy, CPUs use 3 levels of caches with increasing sizes and latencies per level. Caches are only useful if data accesses initiated by the program instructions obey spatial or temporal locality [34].

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

International Journal on Advances in Systems and Measurements, vol 9 no 1 & 2, year 2016, http://www.iariajournals.org/systems_and_measurements/

TABLE I. OVERVIEW OF THE MICRO-BENCHMARKS.

Identifier	Benchmark	Category	Application
B1	Memory read latency	Memory access	Single-thread latency to main memory
B2	Memory bandwidth	Memory access	Bandwidth to main memory
B3	Atomic update	Synchronization	Multi-threaded atomic update of a shared scalar variable
B4	Barrier	Synchronization	Barrier operation of *n* threads
B5	Reduction	Synchronization	Parallel reduction of *n* values to a single value
B6	Communication	Communication	Data transfer bandwidth to/from an accelerator through PCI Express
B7	DGEMM	Computation	Parallel dense matrix multiply (compute-bound)
B8	SPMV	Computation	Sparse matrix multiplied with a dense vector (memory-bound)

IV. Proposed Micro-Benchmarks

We propose a set of 8 micro-benchmarks to determine performance critical parameters in single-node parallel HPC systems. Table I gives an overview of these benchmarks. Each single benchmark tests one specific aspect of a hardware architecture or parallel runtime system on that hardware. These aspects are performance critical for certain application classes. One or a combination of these parameters usually defines the performance bounds of the compute-intensive parts of an application. In real-life applications, it is possible that a combination of these parameters occurs with different factors/weights. It is up to the developer to use his knowledge of the application to weight these factors correctly. If the application is truly dominated by one of these parameters, the developers has an indication whether an architecture would be suitable for this application.

The presented set of micro-benchmarks were implemented in C with OpenMP for the use with Intel processors (including KNC). However, the OpenMP implementation could also be used for other shared memory architectures as well, such as Power 8, ARM, or AMD Processors. Moreover, widely used C compilers such as the Intel icc or the GNU gcc support this programming approach. Recently, the GNU gcc added support for OpenMP 4.0 constructs, which makes it possible to address future Intel Xeon Phi processors as well. For the usage with Nvidia accelerators, the commonly used CUDA programming approach was chosen, as this is the programming model delivering the best performance on these GPUs. Porting the CUDA implementation, for example, to OpenCL should be straightforward, because both programming platforms have similar concepts, although the syntax is quite different.

In the following, we describe the individual benchmarks and our reasons for using them.

A. Memory Performance

Memory accesses are often the main performance bottleneck in applications, for example in an iterative solver working on large sparse matrices [3] or graph processing [35]. The key performance parameters for memory performance are memory latency and memory bandwidth. An indicator of a latency-bound application are many accesses to different small data items (that are not cached). An indicator of a bandwidth-bound application kernel is a program kernel with low computational density, i.e., the ratio of the number of operations performed on data compared to the number of bytes that need to be transferred for that data is low.

1) Memory Read Latency (B1): Read latency can be determined by single threaded pointer chasing, i.e., a repeated read operation of type `ptr = *ptr` with a properly setup pointer table. If all accessed addresses are within an address space of size S (without associativity collisions in the cache) and S is smaller than the cache size, then all accesses can be stored in this cache.

2) Memory Bandwidth (B2): The Stream benchmark [26] is commonly used to measure main memory bandwidth. We adapted this freely available benchmark for the Xeon Phi using the OpenMP `target` construct [18] and for graphic processors using CUDA programming constructs [36], i.e., both are used in accelerator mode called from a host.

B. Synchronization Performance

Synchronization between execution units (threads, processes, etc.) is necessary at certain points during the program execution to ensure parallel program correctness. However, synchronization is often a very performance critical operation [37], because it requires serialization, e.g., atomic updates, or overall agreement, e.g., a barrier between the execution units. Moreover, reduction operations are another important and performance critical type of synchronization in real life parallel applications.

1) Atomic Updates (B3): In our atomic update benchmark, all participating threads perform an atomic increment operation on a single, scalar, shared, integer variable in parallel. As a side note, this operation also modifies the variable. Consequently, the coherence protocol initiates a cache line invalidation/update in a cache coherent multi-cache based system. The atomic increment operation is repeated by each thread many times during the benchmark. The benchmark then gives the time of one such operation performed by one thread. This operation is realized by the OpenMP atomic construct on the CPU/Xeon Phi and a Cuda atomic add operation on the GPU.

2) Barrier (B4): In the barrier benchmark, a barrier operation is carried out repeatedly. For multiprocessors, the benchmark uses an OpenMP barrier **`pragma`** inside a parallel region. For the Xeon Phi, this program kernel is surrounded by a `target` region. The CUDA execution model [36] does not support a barrier synchronization between all threads as such, because this would violate the basic concept of warp independence. In CUDA, a program with global steps is implemented using a sequence of multiple kernels. Therefore, the closest adequate comparison to a barrier is the kernel launch time (with an empty kernel), with the ensuing synchronization waiting for the kernel finalization.

3) Reduction (B5): In the reduction benchmark, a vector with n elements of type **`double`** is reduced to one **`double`** value summing up all vector elements. For a reduction, partial sums must be summed up in a synchronized way, which is additional work compared to a sequential implementation and needs some serialization between parallel entities. The program for the multiprocessors uses the OpenMP reduction clause in a parallel for-loop. On multiprocessor systems, the vector is initialized in parallel, such that parts of the vector are split over different Non-Uniform Memory Access [34] (NUMA) nodes

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE II. SELECTED HARDWARE PARAMETERS OF THE SYSTEMS USED.

Parameter	Processor Systems			Accelerator Systems			
Architecture	SB	IB	HW	KNC	M2050 (Fermi)	K20m (Kepler)	K80 ($2 \times$ Kepler)[4]
Clock [GHz] (with TurboBoost)	2.6 (3.3)	2.7 (3.5)	2.6 (3.6)	1.053	1.15	0.706	0.560 (0.875)
Peak double prec. perf.[1][GFlops]; 1 proc.	20.8	21.6	33.17	16.8	-	-	-
Peak double prec. perf.[1][GFlops]; all proc.	332.8	518.4	929	1010.8	515	1170	$2\times$ 935
Theor. memory bandwidth [GB/s][2]	102.4	119.4	136	320	148	208	2×240
Main memory size [GB]	128	256	128	8	3	5	2×12
Degree of parallelism [3]	32	48	56	240	448	2496	2×2496

[1] In relation to baseclock
[2] ECC off for accelerators
[3] Including hyperthreads
[4] We used only one of the two processors

in a NUMA system. Such a distribution is performed internally by the operating system following the parallel memory access pattern. As CUDA does not provide reduction operations itself, the open source (CUDA-based) Thrust library [38] of Nvidia is used for this benchmark on the GPU systems.

C. Communication Performance (B6)

In the communication benchmark, we measure the transfer rate of a certain amount of data between a host and an accelerator device over PCI Express. This measurement is carried out for both directions (input data from the host to the accelerator and result data from the accelerator to the host).

D. Programming Kernels

For many scientific application fields, linear algebra operations are building blocks and often belong to the most time-consuming parts of a program. Depending on the problem origin, dense or sparse matrices occur. Operations on dense or sparse matrices stress different parts of a system. The following two evaluation benchmarks cover both matrix types and stress, therefore, different parts of a system. These are both performance limiting for many applications, also outside linear algebra.

1) Compute-bound application kernel – DGEMM (B7): For dense matrix multiply with a high computational density, many techniques are known (and applied inside optimized library functions) that allow this operation to be run near the peak floating point performance. Consequently, if implemented adequatly, dense matrix multiply evaluates in essence the floating point capability of a core/processor/multiprocessor system. This operation has been well researched and is implemented efficiently in the BLAS library [39] and vendor optimized libraries such as the Intel MKL [40] and Nvidia cuBLAS [41].

2) Memory-bound application kernel – SPMV (B8): In contrast, a sparse matrix multiplied with a dense vector (SPMV) stresses almost only the memory system, as it has a low computational density. The operation is available for multiple storage formats [3] and is, at least for larger matrices, memory bandwidth limited and *not* compute bound. SPMV is also available in the vendor optimized libraries Intel MKL [40] and Nvidia cuSPARSE [42], both with a small selection of supported storage formats for the sparse matrix. The CSR format [3] is a general format with good/reasonable performance characteristics for many sparse matrices on CPU-based systems. For appropriate matrices (that have a small and ideally constant number of non-zero elements per row), the ELL format is a favorable storage format on GPUs [43]. This difference is related to the different memory systems of CPU-multiprocessors and GPU systems. Nevertheless, in this benchmark we are not interested in the best possible performance for a specific matrix. We are more interested in relating the performance of different systems for this type of operation in a more general way.

V. Experimental Setup

In this section, we specify the parallel system test environment where the benchmarks were applied. Additionally, we discuss the bencmark parameter settings, because performance can be a parameterized function, e.g., dependent on the number of used threads or data items.

A. Test Environment

The used systems include the three latest generations of Intel server processors: Sandy Bridge-EP (SB), Ivy Bridge-EP (IB) and Haswell-EP (HW). All of the systems are 2-way NUMA multiprocessor systems with 2-way hyperthreading per processor. As representatives for accelerators the Intel Xeon Phi Knights Corner (KNC), with 4-way hyperthreading, as a many-core architecture and three most recent Nvidia GPU architechtures (M2050, K20m, K80) were examined. The tested accelerators use PCIe 2 x16 for KNC, M2050 and K20m (both Nvidia GPUs) and PCIe 3 x16 for the Nvidia K80 GPU. The new Nvidia K80 consists of two Kepler GPUs, which work as two single devices and have to be programmed seperately. Only one of the GPUs was used to perform the benchmarks. Table II summarizes key hardware parameters of the systems used.

B. Test Parameters

The benchmark tests were executed with the following parameter settings:

- *Memory latency (B1)*: Variable size of the pointer table with a single threaded run.
- *Memory bandwidth (B2)*: *a)* Fixed large vector size of `STREAM_ARRAY_SIZE=40000000` and a repeat factor of `NTIMES=1000` (all systems). *b)* Same, but different thread mapping (CPUs, KNC).
- *Atomic update (B3)*: *a)* Variable number of threads according to the systems used (all systems). *b)* Same, but different thread mapping (CPUs, KNC).
- *Barrier (B4)*: *a)* Variable number of threads according to the systems used (all systems). *b)* Same, but different thread mapping (CPUs, KNC).
- *Reduction (B5)*: *a)* Variable vector size with a full parallel run. *b)* Variable thread number with fixed vector size and different thread mappings (CPUs, KNC).
- *Communication (B6)*: *a)* Variable size of the transferred data (accelerators). *b)* Pinned and unpinned host memory (GPUs).

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

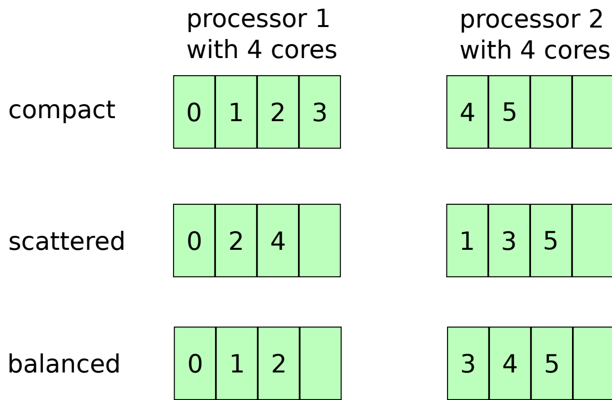

Figure 1. Strategies for thread mapping on CPUs. In this example 6 threads are mapped onto 2 processors with 4 cores each. Numbers in boxes are the thread numbers.

- *DGEMM (B7)*: *a)* Variable matrix size with a full parallel run (all systems). *b)* Fixed matrix size with a variable number of threads (CPUs, KNC).
- *SPMV (B8)*: Fixed test matrix according to the SPE10 problem [44]. *a)* SPMV implementation of MKL resp. cuSPARSE of CSR (all systems). *b)* Own implementation of ELL kernel with different thread mapping, ELL implementation of cuSPARSE (CPUs/KNC, GPUs).

C. Thread Mappings

Thread mapping/binding can be an important aspect achieving good performance. A thread mapping defines how application threads are mapped to hardware units, e.g., processor sockets, cores in a multi-core CPU, hardware threads in a hyperthreaded core. On a GPU, the definition of a grid size and block size defines a 1D-3D partitioning of the application data space (e.g., a 2D picture) to the hardware units. Thread mapping influences load balance, coherence issues, data locality and more.

Basic mapping strategies on a CPU-based system are (see Figure 1 for an example):

- *Compact*: keep consecutive threads as close as possible in the hardware, e.g., to exploit data locality between threads in a shared cache. Cores are filled up one by one with software threads.
- *Scattered*: spread threads to as many processors as possible in the hardware, e.g., to exploit as much memory bandwidth as possible from different CPU sockets in a NUMA system. If thread i was placed at processor p, then thread $i+1$ is placed at processor $p+1$ with a wrapping at the last processor. This means that all processors and the corresponding memory bandwidth is utilized if at least as many software threads are available as processors.
- *Balanced*: similar to scattered. Utilize as much processors as possible but fill nearby threads to the same core. This is a combination of locality utilization (nearby threads are mapped to the same processor with a unified last level cache for all cores on that processor) and memory bandwidth allocation (use as many processors as possible).

OpenMP defines appropriate environment variables to influence thread mapping strategies [18]. With the Intel icc compiler this can be accomplished by using the KMP_AFFINITY variable. This variable can be set to either compact or scattered (or balanced for KNC only) [45]. The names of the strategies are therefore different in OpenMP and the Intel specific KMP_AFFINITY variable, but the meaning is more or less the same.

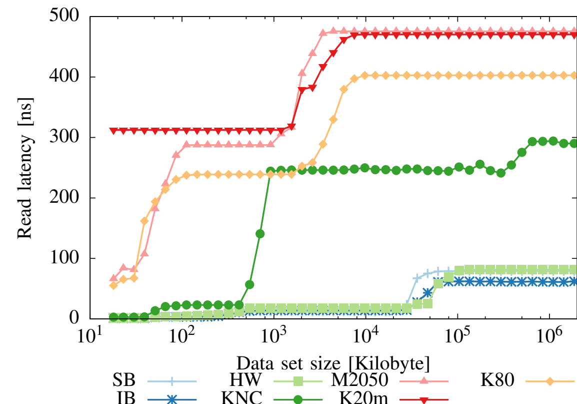

Figure 2. Memory latency results, absolute time.

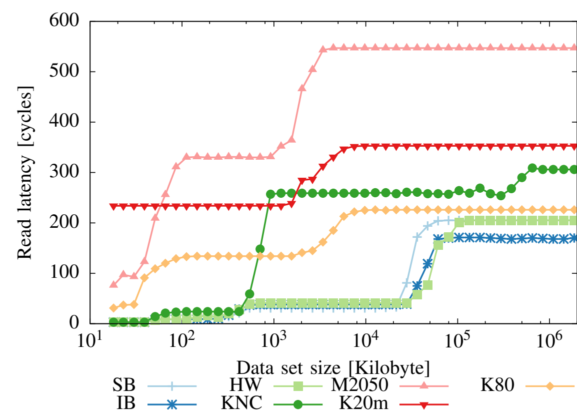

Figure 3. Memory latency results, relative cycles.

On GPUs the programming model is different to a CPU programming model. A thread mapping is done by specifying grid and block sizes. Different to a CPU-based system where usually a 1:1 mapping of software to hardware threads is established, on a GPU many more software threads are generated than hardware parallelism is available, with the aim to hide memory latency. If a hardware thread is blocked by a memory read operation, another runnable thread gets scheduled by the hardware scheduler to make the read latency tolerable. Specifying grid and block sizes partitions the space of software thread into up to 3 dimensions and these partition units get scheduled by the hardware scheduler on a GPU. While the thread mapping done by a programmer on CPU-based systems is optional, the thread partitioning on a GPU is an important part of GPU programming.

VI. Results

In this section, we discuss the main results of applying our proposed benchmarks to the different types of architectures described in Section V. We concentrate on the interesting

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

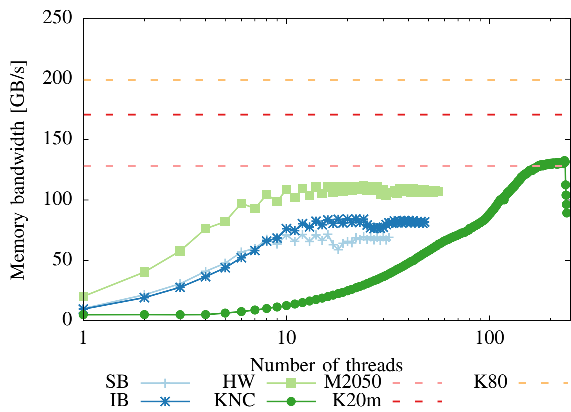

Figure 4. Memory bandwidth results.

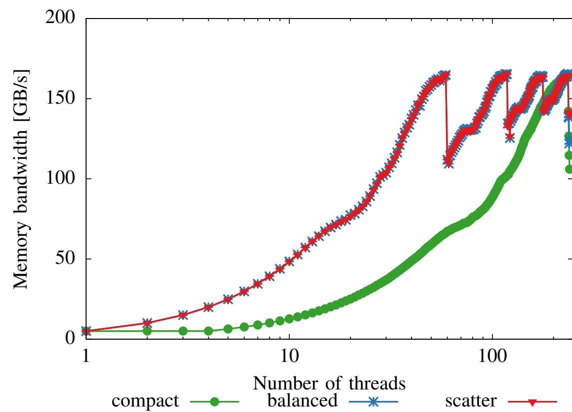

Figure 6. Memory bandwidth results on KNC, different thread mapping (balanced is nearly the same as scattered).

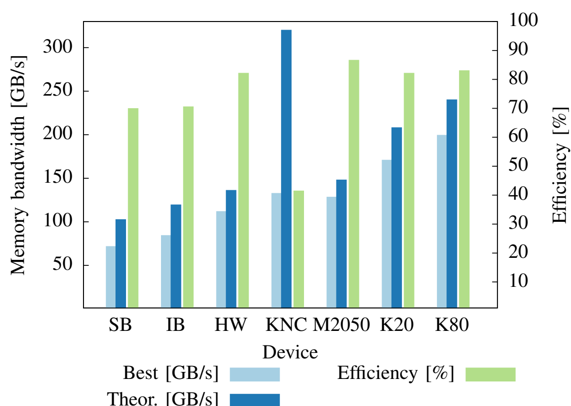

Figure 5. Memory bandwidth best results.

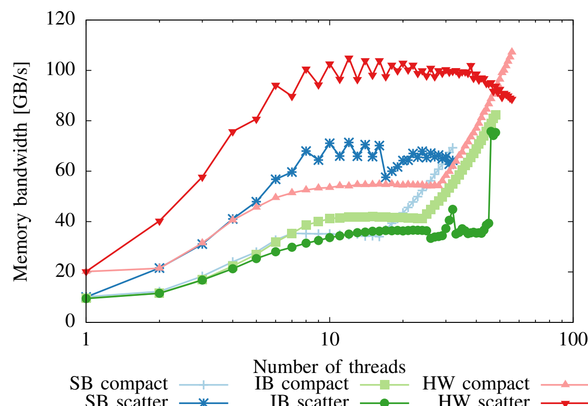

Figure 7. Memory bandwidth results on CPUs, different thread mapping.

aspects of the results. When performance data is plotted as a function of the number of threads, it is meant as number of thread blocks for GPUs, because the usage model for graphic processors differs from a multiprocessor system, as explained before. On GPUs, usually all stream processors of such a processor are used (with even more concurrency in the application to hide latencies) instead of specifying the exact number of threads, as it is usually done on a CPU.

A. Memory Read Latency (B1)

Figure 2 shows the results for the memory latency with an access stride of 256 byte in absolute times. Figure 3 shows these results in cycles relative to the respective base CPU/GPU clock. Clearly visible for all systems are the levels of the same latency induced by cache sizes of the different cache levels and the huge difference to a main memory access (the last step to the right). If only absolute times are considered, all accelerators have higher latencies than the processor architectures and the GPU-based Nvidia accelerators are slower than a CPU-based KNC. Moreover, there seems to be hardly any improvement between GPU generations. But, if relative latencies are considered, the GPUs improve over the generations quite significantly, as the base clock is much lower while the parallelism is higher. Related to relative cycles, the newest K80 outperforms the KNC and even gets close to the CPUs in access to the global/main memory.

The different cache levels show that the measurements on the M2050 and K80 GPUs have three different levels in access time, which can be explained by the L1/L2 caches and accesses to the main memory. On the K20m, only two levels of similar access times are visible. This is induced by different versions of the Kepler architecture in the K20m and the K80. The K20m does *not* cache global memory accesses in the L1 cache, but the newer generation K80 does.

On the CPU-based systems, the curves show first the smaller L1 and L2 caches, then the larger L3 cache and finally, in a fourth step, the access to the main memory. Access to the L1, L2, L3 caches is very fast, for L1 and L2 even on KNC. Altogether the processor systems still outperform the accelerators in latency time, although newer accelerator generations have improved (relatively). Therefore, applications that are already latency bound have a severe problem on accelerator systems if they cannot hide this latency, e.g., by allowing many read requests to be open at the same time.

B. Memory Bandwidth (B2)

The memory bandwidth performance is shown in Figure 4 as a function of used threads. For the processor systems, the default thread scheduling was used here. For graphic processors, the usage model is different to that of a multiprocessor system, because usually all stream processors of such a processor are used instead of specifying the exact number of threads. The

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

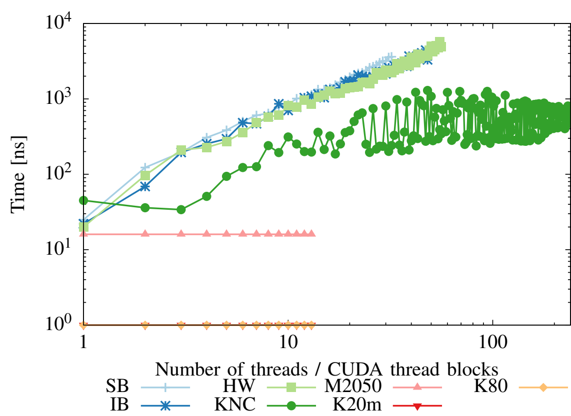

Figure 8. Atomic update results.

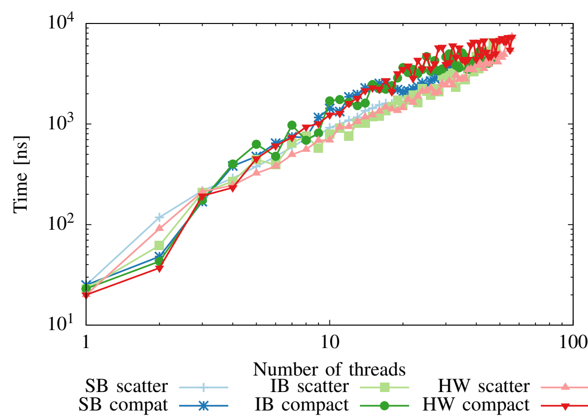

Figure 10. Atomic results CPU, different thread mapping.

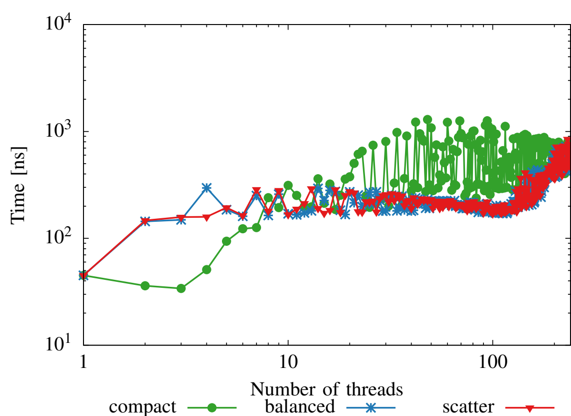

Figure 9. Atomic results KNC, different thread mapping.

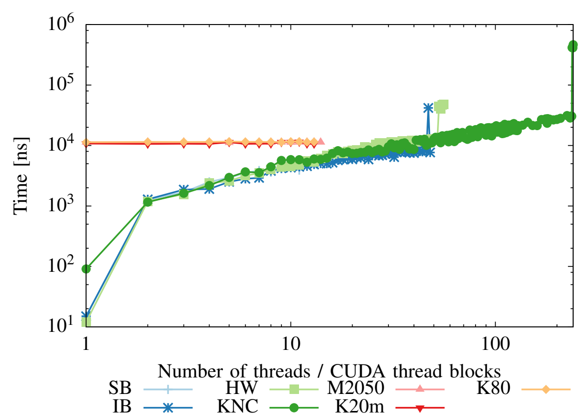

Figure 11. Barrier results.

performance number(s) for GPUs are therefore given as a dashed line with all stream processors used. In contrast to the results on latency, the accelerators perform better than the CPU systems. A summary and comparison to the theoretical bandwidth is given in Figure 5. It is noteable that the KNC performs relativly poorly here. Its measured bandwidth is comparable to the Haswell CPUs and the older Nvidia Fermi GPUs. Moreover, the KNC is not able to reach its theoretical bandwidth at all, though it has by far the highest theoretical bandwidth of all tested systems. For the CPUs, the efficiency within one similar microarchitecture (Sandy Bridge and Ivy Bridge) stays the same. A gain in performance is achieved with the new Haswell microarchitecture.

Figure 6 shows the bandwidth test for the KNC with different thread mapping in OpenMP. A significant difference can be observed when different thread mappings are used. If the compact thread mapping is used (same as in Figure 6), bandwith increases steadily with an increasing number of threads. The performance drops with the last four threads, because, at this point, the last core with its four hardware threads is used in the application, but that core is busy waiting for operating system tasks (communication with the host system).

When a scattered or balanced thread mapping is used, the impact of the four hardware threads per core can be seen. The performance increases until all cores are evenly utilized (one thread per core). Then, as soon as one core gets a second thread, the performance drops and increases again steadily. Again, the impact of the operating system core can be seen when all available threads of the KNC are used.

In Figure 7, similar to the KNC, changing the thread mapping for processor systems shows differences between compact and scattered thread mapping. When using a compact thread mapping on all three CPU architectures, the effect when the second CPU socket gets populated with threads is clearly visible. When only one socket is used, bandwith increases slowly to a point of saturation. Then, when the second socket is used, bandwith increases dramatically. This behavior is different to the KNC compact thread mapping, where a steady increase can be observed. This can be explained by the different layout of the memory connection in KNC (ring-bus) and the CPUs (cc-NUMA).

For the scattered thread mapping, the Ivy Bridge system behaves differently to the other CPU systems, because this node could not be used exclusively in our tests (some system services were active). For the Sandy Bridge and Haswell, results show similar behavior. First the bandwith increases steadily but oscilates. With an odd number of threads, the bandwith drops, and with an even number of threads, the bandwith rises again, because here the memory channels of both CPUs can be used evenly. Moreover, the overall bandwith drops when hyperthreads get used. This effect can be clearly

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

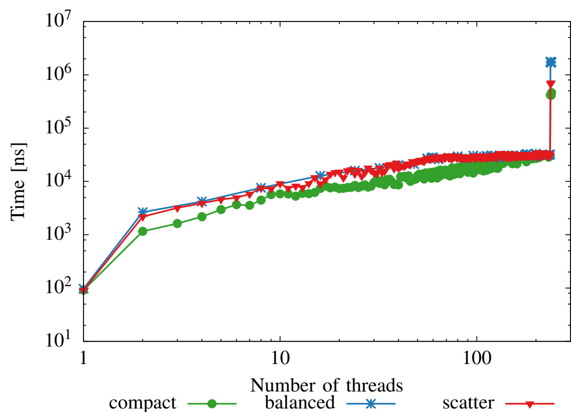

Figure 12. Barrier results on KNC, different thread mapping.

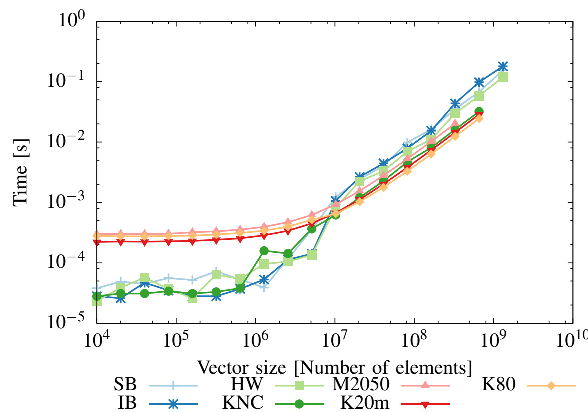

Figure 14. Reduction results, vector size variable.

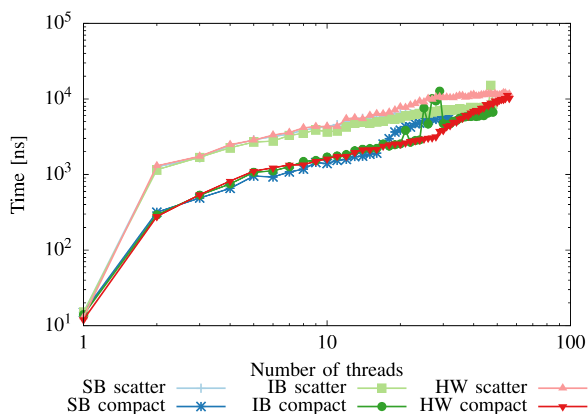

Figure 13. Barrier results on CPUs, different thread mapping.

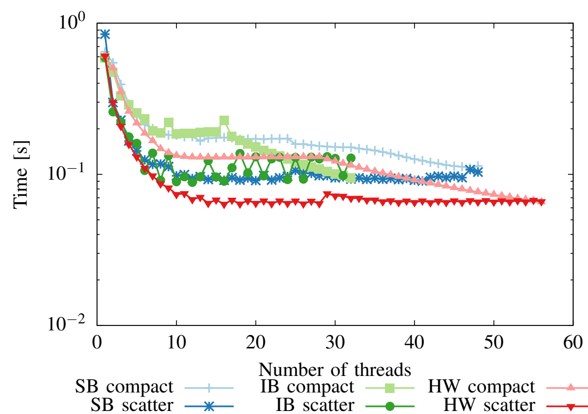

Figure 15. Reduction results on CPUs, thread number variable.

seen for Sandy Bridge with the strong drop in the curve. For the Haswell system, that drop is not as strong, but still the bandwidth decreases steadily from the point hyperthreads where are used. Furthermore, the bandwith drops even below the level of compact thread mapping at a certain point.

C. Atomic Updates (B3)

Figure 8 shows the performance results of the atomic operation on the different systems. On the multiprocessor systems, time increases linearly, proportional to the number of competing threads in use. Because the performance numbers show the normalized time for *one* operation of *one* thread, there is an increase in time *per operation* with the number of threads. This increase can be explained by the coherence and synchronization protocol, which is run by the processors/cores to ensure coherence and atomicity of such an operation. With more competing threads involved, the overhead increases [35]. For all three GPU systems, the time is constant, which can be explained by the use of the single unified L2 cache and the weak memory model without memory coherence. Moreover, the performance improvement for atomic operations from Fermi (M2050) to Kepler (K20m, K80) is clearly visible in this figure. For the KNC with compact thread mapping, quite large fluctuations can be observed (note the logscale of the plot).

Figure 9 shows the atomic benchmark for KNC with different thread mappings. When scattered and balanced thread mapping is used, the fluctuations become smaller, but the performance change with an increasing number of threads is still fairly unsteady. An explanation for this could be the ring bus of the KNC. The cache of *all* cores in the KNC have to be kept coherent via this ringbus. Moreover, from the time when one core is populated with all four hardware threads, the time for an atomic update increases significantly.

Figure 10 shows the results for changing the thread mapping for processor systems. The curves show the same linear behavior for compact and for scattered thread mapping.

D. Barrier (B4)

Figure 11 illustrates performance results of the barrier test with the default thread mapping. The barrier synchronization on the KNC shows a similar behavior to that on the multiprocessor systems with a linear increase with the number of used threads. Using the last core on KNC and on multiprocessors shows a large performance degradation. Again this can be explained by operating system tasks that perturb the (global) barrier operation, if the last available hardware thread is used.

For the Nvidia accelerators, the number of threads in the figure represents the number of used thread blocks (with 1024 threads per block used). The figure shows that the kernel launch time is nearly constant and equal for M2050, K20m and K80. Further, it does not depend on the number of blocks.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

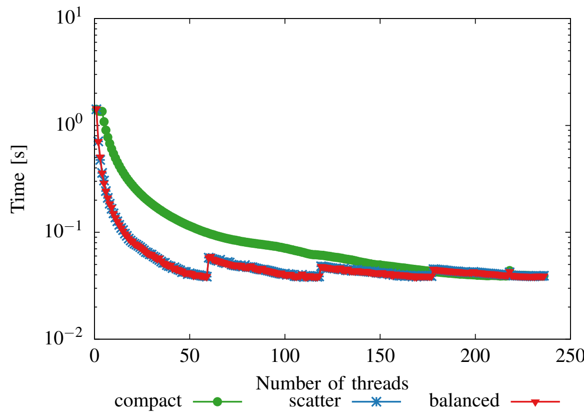

Figure 16. Reduction resultson KNC, thread number variable.

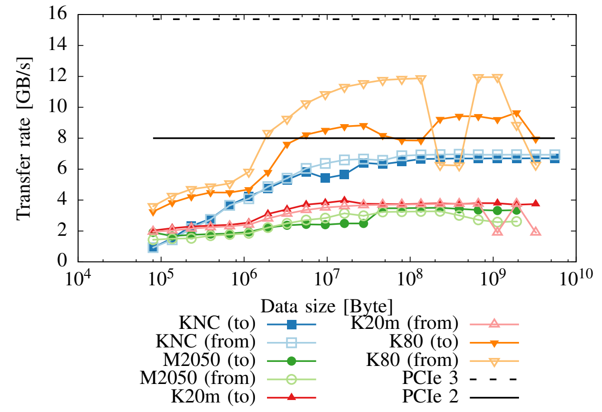

Figure 18. Communication performance results, unpinned memory.

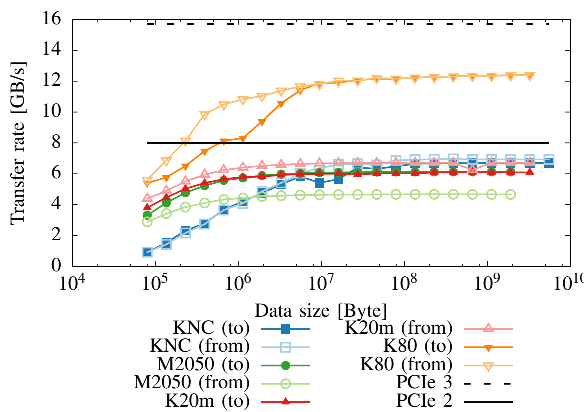

Figure 17. Communication performance results, pinned memory.

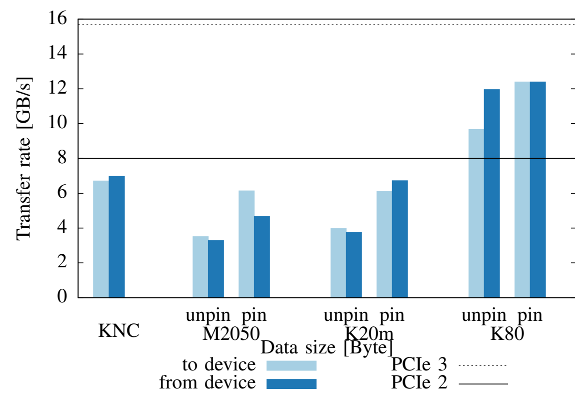

Figure 19. Communication performance best results.

The impact of different thread mappings for the KNC can be seen in Figure 12. The compact strategy is faster compared to scatter, because threads on the same core are synchronized faster than between cores. Therefore, utilizing as few cores as possible for a fixed thread count is the fastest strategy.

Figure 13 shows that, for the multiprocessor systems, the barrier operation is faster with few threads if the compact thread mapping is used (using less cores utilizing the hyperthreads on these cores) compared to the scatter strategy. When all threads are used, the performance is invariant of the thread mapping strategy.

E. Reduction (B5)

For the reduction test, Figure 14 shows the parallel run time using all available parallelism on each system with an increasing vector size. The M2050 card was limited by the available memory size, thus the largest vector size used on other systems could not be used on this system. The GPUs are slower than the multiprocessors for a smaller number of elements, and they are faster than the multiprocessors for large vectors, which corresponds to the usage model of GPUs.

In Figure 15, the results for the multiprocessors with varying thread numbers and different thread mappings are shown. Here, a sufficiently large vector with 10^9 elements was chosen. The figure shows that scattered thread mapping has a better overall performance than compact thread mapping. For a compact thread mapping, the time for the reduction decreases faster when the second CPU socket is used and both memory channels get used. However, times for compact and scattered thread mapping are equal when all threads are used. Furthermore, for scattered thread mapping, the effect of hyperthreads can be seen in the jumps of the execution time.

Figure 16 shows the results for a variable number of threads with a fixed vector size of 8×10^8 on KNC. Again the compact thread mapping shows an overall weaker performance than the scattered thread mapping due to memory bandwidth requirements. For a scattered thread mapping, the effect of the 4-way hyperthreading can be seen in the jumps of the execution time, too. Similar to the multiprocessors, compact and scattered thread mapping have equal results when all threads are used.

F. Communication Host-Device (B6)

For the communication benchmark experiments, the data transfer from the host to the device and back from the device to the host was considered. Moreover, we differentiated on the GPUs for the communication to/from a GPU between pinned and unpinned host memory. For GPUs, it is explicitly possible to allocate page-locked memory on the host using CUDA functions [36].

Figures 17 and 18 show data transfer rates in GB/s from the host to the attached accelerator and vice versa for pinned

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

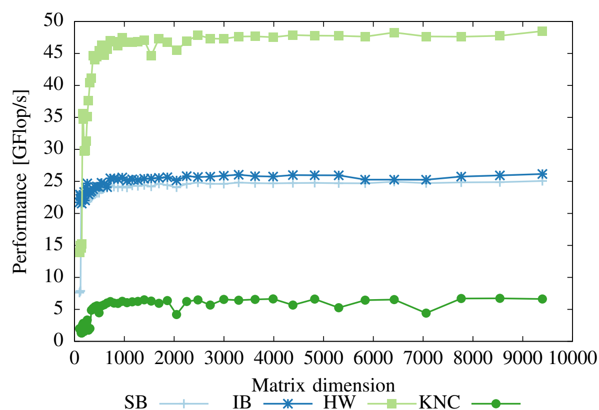

Figure 20. DGEMM results, single threaded.

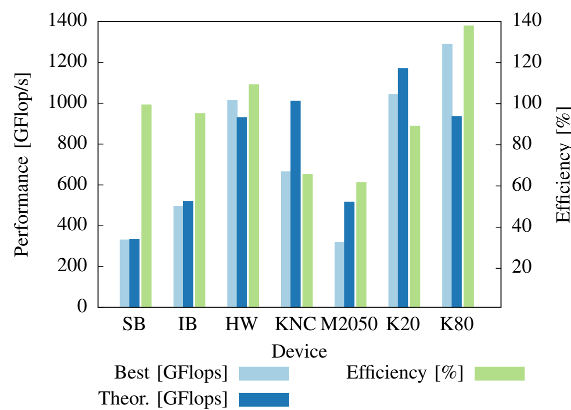

Figure 22. DGEMM best results, full parallel.

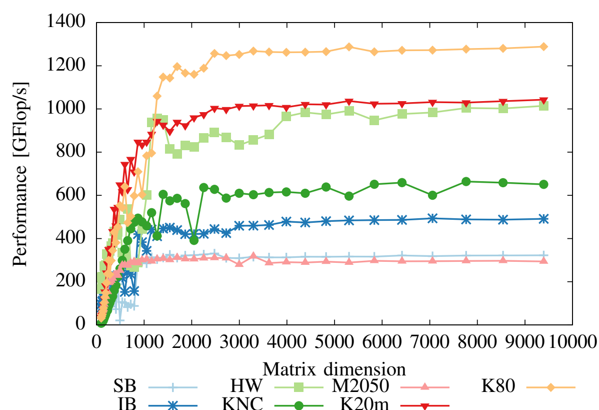

Figure 21. DGEMM results, full parallel.

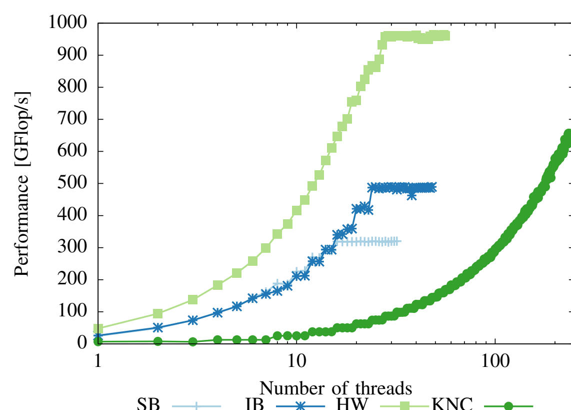

Figure 23. DGEMM results, thread number variable, n=5000.

and unppined memory. For KNC there is no such distinction. The figures show that, for a reasonably large data size, the communication links are used efficiently on the KNC, the theoretical data transfer rate of 8 GB/s for PCIe 2.0 minus protocol overhead can nearly be reached. Figure 17 shows that this is also true for the K20m and M2020 GPUs, when pinned memory is used. Here, the K20m and the M2050 perform similarly in data transfer to the device. For the transfer back from the accelerator to the host, there is a performance drop on the M2050, which reaches only approx. 5 GB/s instead of nearly 7 GB/s as on the other accelerators. The lower bandwidth seems to be a problem with our combination of host system and accelerator card.

On the K20m, the transfer from the device performs slightly better than to the device. For the K80, the transfer rate to and from the device is the same for larger data sizes, but this value does not reach the theoretical limit for the PCIe 3. Perhaps this limit could be reached if both GPUs of the K80 are utilized (as explained, we used only one of them). Moreover, for smaller data sizes, again transfer rates from the K80 are better than to the K80. The difference between host to device and device to host bandwidth for the GPUs could be due to the Direct Memory Access (DMA) initiator. When the DMA is initiated from the CPU, it has a better performance.

Figure 18 shows that not using pinned memory for GPUs deteriorates the transfer rate. The K20m and M2050 have almost completely lower transfer rates than the KNC and do not reach the theoretical limit at all. Moreover, in contrast to pinned memory, transfer rates from and to these GPUs show nearly the same behavior. The K80 is able to reach the pinned memory transfer rates only for larger data sizes.

Figure 19 summarizes the best communication results of the acclerators. It clarifies that there is only a minor difference between communication to or from the device. But there is a quite large difference in transfer rates between pinned and unpinned memory on GPUs.

G. DGEMM (B7)

In Figure 20, the results for a single threaded run with variable matrix dimension of the DGEMM benchmark are displayed. Here the GPUs are omitted, because a single threaded run does not correspond to the GPU usage model. It can be seen that, for an increasing matrix size, the perfomance quickly saturates.

The KNC has by far the weakest single thread performance of all CPU-like systems, because the single CPUs in the KNC have a rather limited performance compared to the other multiprocessors used. Moreover, Sandy Bridge and Ivy Bridge show nearly the same performance because they have more or less the same microarchitecture. The Ivy Bridge performs only a little bit better because of its higher clock. Finally, it can be seen that the new Haswell shows the best performance, twice

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

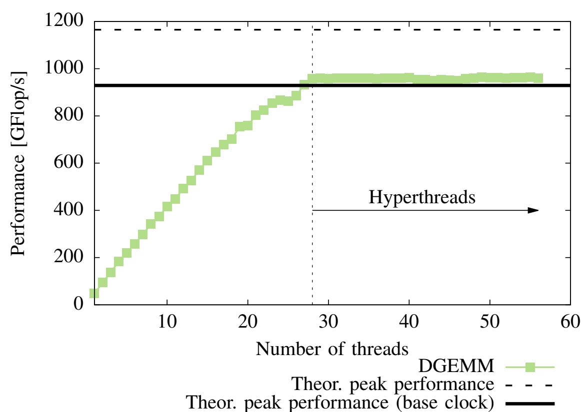

Figure 24. DGEMM on HW, thread number variable.

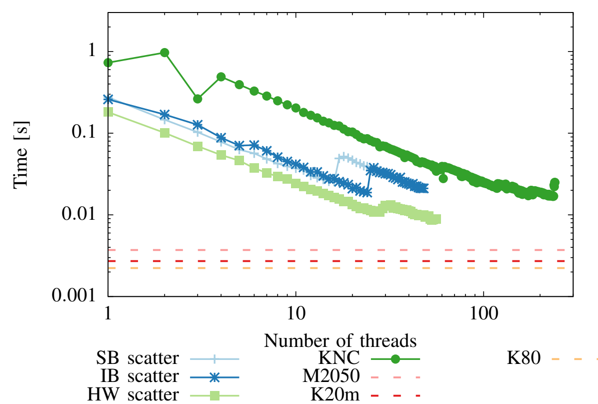

Figure 26. SpMV results, ELL format.

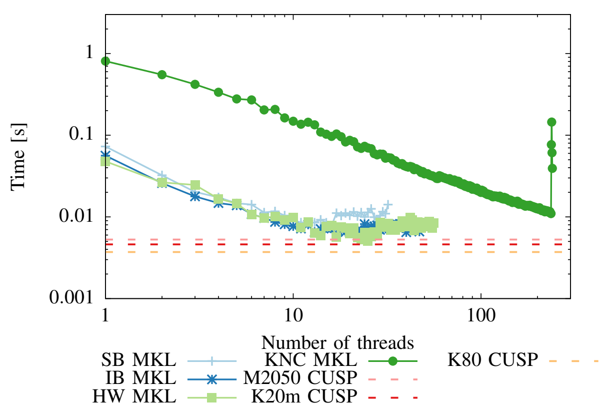

Figure 25. SpMV results, CSR format.

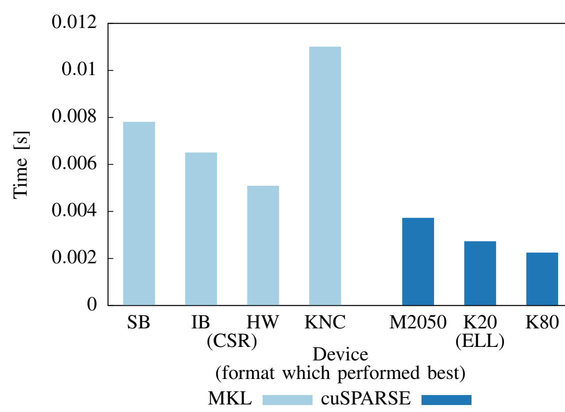

Figure 27. SpMV best results.

the performance of the older systems.

Subsequently, Figure 21 displays the results of the DGEMM benchmark for a full parallel run (now GPUs included) with variable matrix size. Again, on all systems, the perfomance rises quickly and reaches saturation for sufficently large matices. As one would expect, the K80 shows the highest performance, because it has the highest theoretical performance.

Figure 22 summarizes the performance results for the dense matrix multiply operation. Here, only the best performance over all matrix sizes is given. Moreover, the theoretical (base) performance and the efficiency (performance divided by theoretical (base) performance) for each system are shown. As expected, the operation has better performance on the accelerators due to their better raw floating point performance, which can be utilized in a DGEMM operation. It can be seen that, on the majority of the processor systems, almost peak performance is reached. The Haswell processor even shows better performance than the given theoretical peak performance in Table II (related to the base clock). This can be explained by the intelligent turbo boost and temporal overclocking of these processors. Moreover, the Haswell processors are the first CPUs that reach (nearly) one Teraflop performance. That makes them comparable to even recent accelerators. Haswell outperforms the older Fermi architechture and the KNC, which does not reach its theoretical performance at all. The Haswell results for matrix multiply are on nearly the same level as the recent Kepler K20m GPU and are only clearly beaten by the new Kepler K80 (with one GPU used). The K80 also shows better performance than its given theoretical peak performance based on the base clock. Again, this can be explained by the use of turboboost.

Furthermore, the number of threads on the CPU systems and KNC were varied for a dense matrix with a fixed size ($n = 5,000$) that is large enough to reach the compute performance limit. The results for that configuration are shown in Figure 23. For the KNC, the performance increases linearly (logscale used) until all threads are used. However, for all CPU systems, the performance increases linearly until a certain point (hyperthreads come into use) and then remains at that level. In Figure 24, this is shown for the Haswell architecture. Here, without the logscale, one can clearly see the linear increase in performance. Moreover, the figure clearly shows that the stagnation begins when hyperthreads are used. For the KNC this is different. Here, the hyperthreads have to be used to achieve performance [32].

H. SPMV (B8)

Figures 25 and 26 show the results for the SPMV benchmark using the CSR format and the ELL format, respectively. For the multiprocessors and the KNC, the number of threads was varied. For the GPUs the cuSPARSE library was used,

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

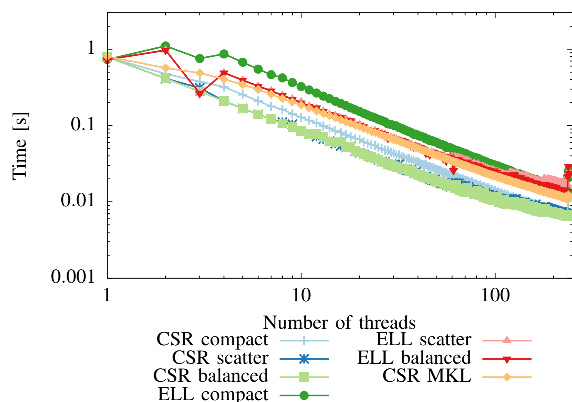

Figure 28. SpMV on KNC, CSR and ELL, different thread mapping.

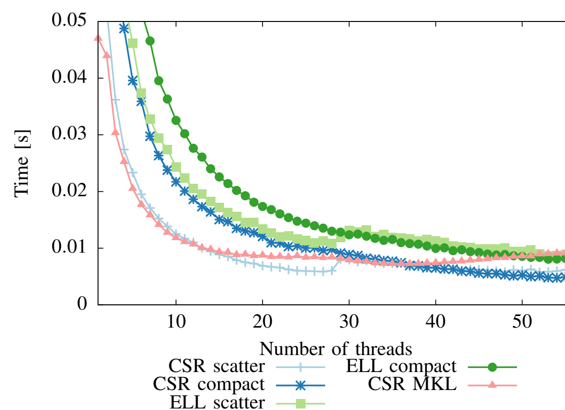

Figure 29. SpMV results on HW, CSR and ELL, different thread mapping.

consequently, the parallelism could not be explicitly controlled. Hence, in all SPMV figures, the GPU results are presented as a dashed line. In Figure 25, the results for the CSR format using the MKL and the cuSPARSE library are shown, because it is assumed that these highly optimized vendor libraries can achieve the best performance for the particular architectures. Overall, it can be seen that, for the test matrix in CSR format, the KNC shows the weakest performance, the Haswell CPU can compete with the older M2050 and K20m and the K80 shows the best performance. For the multiprocessors, it can be seen that the execution time first declines, but then rises again after a certain point (details see below). For the KNC, the execution time decreases steadily until all cores are used, again when the last core gets utilized there is a jump in the execution time because of the operating system administration.

Figure 26 shows the results for the ELL matrix format that is supposed to be a more suitable format for GPUs compared to the CSR format [43]. For the test matrix, this can indeed be verified, because here all GPU systems show relatively better performance than the multiprocessor systems and the KNC, which again has the weakest performance. Moreover, even the absolute execution times show an improvement for the GPUs and a degradation for the multiprocessors and the KNC. For the multiprocessors and KNC, the results of the scattered thread mapping are shown because this performed better on these system (again, for details see below).

As a summary, in Figure 27 only the best results for a system and a format are given. All GPU systems perform well, compared to the multiprocessors and KNC. The K20m performs around 26% faster than the M2050 using the ELL format. The low performance improvement of the K20m can be explained by the fact that the SPMV operation is memory bound and the memory bandwidth of the K20m is only around 25% higher compared to the M2050. Similar relations apply for K20m and K80. Surprisingly, the KNC shows the weakest performance in this test although this card has the highest nominal memory bandwidth of all used systems. The low performance is related to the bandwidth results, where the KNC reached only half of its peak memory bandwidth.

We investigated further the weak performance of the KNC. Figure 28 shows results on the KNC for the CSR and ELL format. Here a rather simple own OpenMP implementation with different thread mappings was used and performance compared to the Intel MKL version. The figure shows that our own CSR version with *all* thread mappings performs better than the Intel MKL version.

Figure 29 shows detailed results for the Haswell multiprocessor. Here the SPMV using CSR and ELL with different thread mapping and the implementation of the MKL for the CSR format are compared. The CSR format has an overall better performance for this matrix on the Haswell multiprocessors than the ELL format. Again, the curve for the execution time of the scattered kernels has jumps due to the use of hyperthreads.

Once more it should be pointed out that, as a common representative of memory bandwidth-bound application kernels, the SPMV benchmark should give a rather general view on these architectures. For detailed insights on computing techniques and further formats for sparse matrices see, for example, [3], [4], [43].

VII. Discussion

Characteristics of the examined architectures could be revealed using the proposed benchmarks. Additionally, the effect of different thread mappings for CPU based architectures was shown.

For memory latency, GPUs are still behind CPU-based systems in absolute times, but newer GPUs gain performance in terms of *relative* clock cycles, alleviating the effects of latency. However, on GPUs, latency time (and clock rate) is traded against parallelism, which follows the usage model of GPUs.

For the memory bandwidth benchmark, GPUs outperform multiprocessor systems and also the KNC. Moreover, the KNC shows a fairly weak memory bandwidth in practice, although it has the highest theoretical memory bandwidth. If on CPUs and KNC not all available hardware threads are used, scattered thread mapping should be used, to utilize as much bandwidth as possible.

For the atomic operations, it was shown that the strong cache coherence model in CPU based systems is disadvantegeous for the performance of that operation. This is especially true for the current MIC architechture of the KNC, where a lot of caches have to be kept coherent. The weak cache coherence of CUDA has clear performance advantages here.

Barrier operations are usually used to separate different program stages. The kernel launch time on a GPU (comparable to such a use) is independent of the number of used threads

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

blocks. This is different on CPUs where the number threads and thread mapping have an impact on the performance of a barrier. The more threads, the longer the time for the barrier operation. But here a compact thread mapping is better for barrier operations if not all available hardware threads are used.

For the reduction operation, for a small amount of data, CPU-based systems with fast L1/L2 caches and a low latency have an advantage compared to GPUs. For a large amount of data, GPUs with the higher bandwidth outperform the CPUs.

For the communication between a host and an accelerator, it was shown that, for GPUs, pinned memory should be used to achieve good transfer rates. The accelerators generally reached their respective theoretical transfer rates via the PCI Bus for sufficiently large data packages. However, these PCI transfer rates of at most 8/16 GB/s (2nd/3rd generation PCIe) are still far behind memory transfer times of approx. 100 GB/s on two socket CPU systems. Therefore, the PCIe is still a severe bottleneck for accelerators. Particularly, the transfer of a small amount of data shows only low transfer rates. Consequently, the number of transfer packages should be reduced. Instead of many small transfers, a few large transfers should be preferred. Additionally, an asynchronous transfer should be used to hide the latency of such an operation. These aspects mean that accelerators are not appropiate for kernels that depend a lot on such a communication.

The DGEMM benchmark reached (near) peak floating performance on all systems, when it was executed with enough parallelism and sufficiently large matrices. For same generations, GPUs show a performance improvement compared to CPUs based on their better raw performance. On CPUs, DGEMM computations do not benefit from the use of hyperthreads due to way hyperthreads work (see, for example, [46]).

Generally, memory bandwidth-bound kernels such as the SPMV are far from reaching peak floating point performance on a system. However, these operations can benefit from the high memory bandwith of recent accelerators, especially on GPUs. The KNC shows severe problems here. This result reflects the result of the memory bandwith benchmark, where the KNC showed a weak performance, too.

VIII. Conclusions

This paper introduced a set of benchmarks to determine important performance parameters of single-node parallel systems. One or a combination of these parameters are often performance limiting in parallel applications. The benchmarks can easily be ported to other architectures.

The benchmarks were applied to systems of the same basic architecture but different processor generations (Intel Haswell, Ivy Brige, Sandy Bridge) as well as to different architectures (CPU, two different accelerator architectures).

It was shown that some parameters (e.g., the memory-related ones) show fairly different performance characteristics between the systems, qualifying or disqualifying a system for certain application classes. In contrast, all systems showed similar behavior for compute-dense problems reaching near-peak floating point performance, which is reasonably comparable between accelerators and latest generation multiprocessors. Due to design decisions in the processor architecture, graphic processors show a remarkable performance on some synchronization operations, operations that often limit the parallel performance.

For certain application classes, additional performance parameters might be important where appropriate benchmarks could be developed as well. This paper discussed only single-node parameters. An extension of this work would be to include cluster architectures, i.e., multiple-node architectures. Further investigation could include also the impact of different programming models such as OpenACC or OpenCL instead of CUDA on a GPU.

Acknowledgements

We would like to thank the CMT team at Saudi Aramco EXPEC ARC for their support and input. Especially we want to thank Ali H. Dogru for making this research project possible.

References

[1] R. Berrendorf, J. Ecker, J. Razzaq, S. Scholl, and F. Mannuss, “Using application oriented micro-benchmarks to characterize the performance of single-node architectures,” in Proc. Ninth International Conference on Advanced Engeneering Computing and Applications in Sciences (ADVCOMP 2015), C.-P. Rueckemann, Ed. IARIA, 2015, pp. 31–38.

[2] A. Petitet, R. Whaley, J. Dongarra, and A. Cleary, “HPL - a portable implementation of the high-performance Linpack benchmark for distributed-memory computers,” http://www.netlib.org/benchmark/hpl/, Tech. Rep., 2008, version 2.0, [retrieved: May 2016].

[3] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd ed. SIAM, 2003.

[4] J. Ecker, R. Berrendorf, J. Razzaq, S. E. Scholl, and F. Mannuss, “Comparing different programming approaches for SpMV-operations on GPUs,” in Proc. 11th International Conference on Parallel Processing and Applied Mathematics (PPAM 2015), 2015, pp. 537–547.

[5] R. Berrendorf, M. Weierstall, and F. Mannuss, “Program optimization strategies to improve the performance of SpMV-operations,” in Proc. 8th Intl. Conference on Future Computational Technologies and Applications (FUTURE COMPUTING 2016), 2016, to appear.

[6] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” Comm. ACM, vol. 52, no. 4, Apr. 2009, pp. 65–76.

[7] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel, “Cache coherence protocol and memory performance of the Intel Haswell-EP architecture,” in Proc. 44th Intl. Conference on Parallel Processing (ICPP 2015). IEEE, Sep. 2015, pp. 739–748.

[8] Top 500 List, http://www.top500.org/, [retrieved: May 2016].

[9] SPEC CPU 2006, Standard Performance Evaluation Corporation, https://www.spec.org/cpu2006/, [retrieved: May 2016].

[10] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, “The NAS parallel benchmarks,” NASA Ames Research Center, http://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf, Tech. Rep., 1994, [retrieved: May 2016].

[11] H. Jin, M. Frumkin, and J. Yan, “The OpenMP implementation of NAS parallel benchmarks and its performance,” NASA Ames Research Center, http://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf, Tech. Rep., 1999, [retrieved: May 2016].

[12] S. Seo, G. Jo, and J. Lee, “Performance characterization of the NAS parallel benchmarks in OpenCL,” in Proc. International Symposium on Workload Characterization (IISWC). IEEE, 2011, pp. 137–148.

[13] R. Murphy, K. Wheeler, B. Barrett, and J. Ang, “Introducing the graph 500,” Cray User’s Group (CUG), http://www.graph500.org/, Tech. Rep., 2010, [retrieved: May 2016].

[14] N. Brown, “A task-oriented graph500 benchmark,” in Proc. Intl.Supercomputing Conference (ISC 2014), ser. LNCS, no. 8488. Springer-Verlag, 2014, pp. 460–469.

[15] J. Jose, S. Potluri, K. Tomko, and D. K. Panda, “Designing scalable graph500 benchmark with hybrid mpi+openshmem programming models,” in Proc. Intl.Supercomputing Conference (ISC 2013), ser. LNCS, no. 7905. Springer-Verlag, 2013, pp. 109–124.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[16] J. Bull and D. O'Neill, "A microbenchmark suite for OpenMP 2.0," SIGARCH Comput. Archit. News, vol. 29, no. 5, 2001, pp. 41–48.

[17] J. Bull, F. Reid, and N. McDonnell, "A microbenchmark suite for OpenMP tasks," in Proc. 8th Intl. Conference on OpenMP in a Heterogeneous World (IWOMP'12), 2012, pp. 271–274.

[18] OpenMP Application Program Interface, 4th ed., OpenMP Architecture Review Board, http://www.openmp.org/, Jul. 2013, [retrieved: May 2016].

[19] J. Treibig, G. Hager, and G. Wellein, "likwid-bench: An extensible microbenchmarking platform for x86 multicore compute nodes," in Tools for High Performance Computing 2011. Springer-Verlag, 2012, pp. 27–36.

[20] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, 2010.

[21] J. Hofmann, D. Fey, J. Eitzinger, G. Hager, and G. Wellein, "Analysis of intels haswell microarchitecture using the ecm model and microbenchmarks," in Architecture of Computing Systems – ARCS 2016, ser. LNCS, no. 3697. Springer-Verlag, 2016, pp. 210–222.

[22] J. Lemeire, J. G. Cornelis, and L. Segers, "Microbenchmarks for gpu characteristics: the occupancy roofline and the pipeline model," in 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2016, pp. 456–463.

[23] P. Thoman, K. Kofler, H. Studtand, J. Thomson, and T. Fahringer, "Automatic OpenCL device characterization: Guiding optimized kernel design," in Proc. Euro-Par 2011. Springer-Verlag, 2011, pp. 438–452.

[24] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. ACM, 2010, pp. 63–74.

[25] X. Yan, X. Shi, and Q. Sun, "An OpenCL micro-benchmark suite for GPUs and CPUs," in Proc. 13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). IEEE, 2012, pp. 53–58.

[26] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, http://www.cs.virginia.edu/stream/, Tech. Rep. TM-88, 1991-2007, [retrieved: May 2016].

[27] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Memory performance and cache coherence effects on an Intel Nehalem multiprocessor system," in Proc. 18th Intl. Conference on Parallel Architectures and Compilation Techniques (PACT 2009). IEEE, Sep. 2009, pp. 261–270.

[28] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems," in Proc. 42th Annual IEEE/ACM Intl. Symposium on Microarchitecture (MICRO 42). IEEE/ACM, 2009, pp. 413–422.

[29] Intel® 64 and IA-32 Architectures Optimization Reference Manual, Intel, http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html, Sep. 2014, [retrieved: May 2016].

[30] Nvidia CUDA, https://developer.nvidia.com/cuda-zone, [retrieved: May 2016].

[31] M. Wolfe, Understanding the CUDA Data Parallel Threading Model. A Primer, pgiinsider ed., PGI, https://www.pgroup.com/lit/articles/insider/v2n1a5.htm, Feb. 2010, (Updated December 2012), [retrieved: May 2016].

[32] J. Jeffers and J. Reinders, Intel® Xeon Phi™Coprocessor High-Performance Programming. Morgan Kaufmann, 2013.

[33] M. Corden, Requirements for Vectorizable Loops, Intel, https://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/, 2012, [retrieved: May 2016].

[34] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th ed. Morgan Kaufmann Publishers, Inc., 2012.

[35] R. Berrendorf and M. Makulla, "Level-synchronous parallel breadth-first search algorithms for multicore- and multiprocessors systems," in Proc. Sixth Intl. Conference on Future Computational Technologies and Applications (FUTURE COMPUTING 2014), 2014, pp. 26–31.

[36] Nvidia, CUDA C Programming Guide, pg-02829-001_v6.5 ed., http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, Aug. 2014, [retrieved: May 2016].

[37] R. Berrendorf, "A technique to avoid atomic operations on large shared memory parallel systems," Intl. Journal on Advances in Software, vol. 7, no. 7&8, 2014, pp. 197–210.

[38] Nvidia, Thrust, https://developer.nvidia.com/thrust, [retrieved: May 2016].

[39] BLAS (Basic Linear Algebra Subprograms), http://www.netlib.org/blas/, [retrieved: May 2016].

[40] Intel® Math Kernel Library, https://software.intel.com/en-us/intel-mkl, [retrieved: May 2016].

[41] Nvidia cuBLAS, https://developer.nvidia.com/cublas, [retrieved: May 2016].

[42] Nvidia cuSPARSE, https://developer.nvidia.com/cusparse, [retrieved: May 2016].

[43] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," Nvidia Corp., Tech. Rep. NVR-2008-004, Dec. 2008.

[44] SPE Comparative Solution Project, Society of Petroleum Engineers, http://www.spe.org/web/csp/, [retrieved: May 2016].

[45] User and Reference Guide for the Intel C++ Compiler 15.0, https://software.intel.com/en-us/ ed., Intel Corporation, 2014, [retrieved: May 2016].

[46] G. Hager and G. Wellein, Introduction to High Performance Computing for Scientists and Engineers. CRC Press Taylor and Francis Group, 2011.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org