Using Application Oriented Micro-Benchmarks to Characterize the Performance of Single-node Hardware Architectures

R. Berrendorf, J. P. Ecker, J. Razzaq and S. E. Scholl
Computer Science Department
Bonn-Rhein-Sieg University of Applied Sciences
Sankt Augustin, Germany
e-mail: {rudolf.berrendorf, jan.ecker,
javed.razzaq, simon.scholl}@h-brs.de

F. 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 as well as the two accelerator lines Intel Xeon Phi and Nvidia graphic processors. Results show similarities for some parameters between all architectures, but significant differences for others.**

Keywords–Performance benchmarks; Intel processors; Intel Xeon Phi; Nvidia graphic processors

I. INTRODUCTION

For resource-intensive computations in High Performance Computing (HPC) on a single node level the performance characteristics of the processor, memory and core-core / core-memory interconnect architecture are important to understand, to achieve good performance. Often HPC applications stress for most of their run time only parts of the hardware in compute intensive program kernels. Examples are compute bound problems, such as direct linear solvers [1] that are bound by the floating point capability of a system or memory bandwidth bound problems like the multiplication of a sparse matrix with a dense vector in iterative solvers [2]. 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 [3] for two of such parameters). These parameters often allow to draw conclusions for 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, bottlenecks of architectures can be revealed. The 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.

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 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, the evaluation results are discussed, followed by a conclusion.

II. RELATED WORK

Benchmarks are widely used to evaluate systems concerning certain performance properties. The result of a benchmark should 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 exists, dependent on the question to be answered.

The Top500 list [4] uses the High Performance Linpack [1] to rank (very) large parallel systems. This benchmark produces just a single value, the FLOP-rate (Floating Point Operations) per second for just one specific task, the direct solution of a very large linear system.

The widely used SPEC CPU benchmark [5] 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 factor for each class. These numbers express a *relative* performance improvement compared to an older system.

Williams et al. introduced the roofline model [3] to describe the expectable performance space in a resource bound problem. The two resources in this model, evaluated in a two-dimensional chart, are computational density (operations per transferred 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 [6] are more application oriented benchmarks. These benchmarks consist of larger compute intensive kernels and where originally designed to stress large parallel computers. Each of these applications represents a different computing aspect. The applicications include, e.g., Conjugate Gradient (irregular memory access), Multi-Grid (long- and short-distance communication), or fast Fourier Transform (all-to-all communication). These benchmarks have been, amongst others, implemented in OpenMP [7] and recently in OpenCL [8]. With the OpenCL extension they can be used to measure recent accelerators, such as GPUs.

For a finer granularity, benchmarks that give individual results for several operation classes can be used. An example is the OpenMP micro-benchmark suite [9] [10] that gives a

 ISBN: 978-1-61208-419-0

TABLE I. OVERVIEW OF THE MICRO-BENCHMARKS.

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

developer a measure of how well basic constructs of OpenMP [11] 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.

Other micro-benchmark suites, which aim for a finer granularity are proposed in [12], [13] and [14]. These benchmark-suites are based on OpenCL. Here, OpenCL is used to compare memory related issues, as well as low level floating point operations and real life applications on different hardware architectures, including accelerators.

III. Current Hardware Architectures

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

A. Processor Architectures

We concentrate on the Intel Xeon EP line of HPC relevant processors, as these processors are used in nearly all new systems in the Top500 list [4] of HPC computers. Intel's recent micro-architectures are Sandy Bridge (SB), and it's successor Ivy Bridge (IB). The last change in the architecture appeared late 2014 in the Haswell processors (HW). A detailed description of the architectures is given in the related literature of the manufacturer [15].

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.

B. Accelerator Architectures

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 predominant in HPC [4].

A Nvidia GPU has a hierarchical design (CUDA architecture [16]) that differs from common CPUs. The execution units (SE, Streaming Processors) are organized in multiprocessors, called Streaming Multi-Processors (SM or SMX), a GPU has several of such multiprocessors. For example, the Kepler GPU 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 [17].

An Intel Xeon Phi coprocessor [18] is compromised of multiple CPU-like cores. The current generation Xeon Phi Knights Corner (KNC) has between 57 and 61 of 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 [19] 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 plug-in 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

On one side DDR3 / DDR4 RAM, which is used in CPU-based systems is mostly optimized for a short latency time. On the other side, GDDR5 memory used in accelerators is optimized for bandwidth. This is important, as the performance of accelerators mainly comes from Single Instruction Multiple Data (SIMD) parallelism, where the same instruction is applied concurrently to multiple data items. These data items have to be fed to the functional units in parallel, needing high main memory bandwidth.

All processors of discussion, including recent GPUs, use caches to speed up memory accesses. While GPUs currently have a 2 level cache hierarchy, CPUs use 3 levels of caches with increasing sizes and latencies. Caches are only useful if data accesses initiated by the program instructions obey spatial or temporal locality [20].

IV. Proposed Micro-Benchmarks

We propose a set of 8 micro-benchmarks to determine performance critical parameters in single-node parallel HPC systems. 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. Table I gives an overview of the proposed set. One or a combination of these parameters are usually the performance bounds of an application. In real-life application it is possible, that a combination of these parameters occur with different factors / weights. It is up to the developer to use his knowledge of the application to weight these factors correctly. Nevertheless, if the application is truly dominated by one of these parameters the developer has an indication whether an architecture would be suitable for this application.

The presented set of micro-benchmarks was implemented in C with OpenMP for the use with Intel Procesors (including KNC). However, the OpenMP implementation could also be used for further architectures, like Power 8, ARM, or AMD Processors. Moreover, widely used C compilers like the Intel

 ISBN: 978-1-61208-419-0

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. 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 as both programming platforms have similar concepts.

A. Memory Performance

Memory accesses are often the main performance bottleneck in applications. An example for that is an iterative solver working on large sparse matrices [2] or graph processing [21]. Memory performance itself is mainly influenced by memory latency and memory bandwidth as the key performance parameters. An indicator for a latency bound application are many accesses to different small data items (that are not cached). An indicator for a bandwidth bound application 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 ,which need to be transferred for that data, is low.

1) Memory Read Latency: 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 a cache size then all accesses can be stored in this cache.

2) Memory Bandwidth: To measure main memory bandwidth the Stream benchmark [22] is commonly used. We adapted this freely available benchmark for the Xeon Phi using the OpenMP `target` construct [11] and for graphic processors using CUDA programming constructs [23].

B. Synchronization Performance

Synchronization between execution units (threads, processes, etc.) at certain points during the program execution is necessary to ensure parallel program correctness. However, synchronization is often a very performance critical operation [24], because serialization, e.g., atomic updates, or overall agreement, e.g., barrier between the execution units, is necessary. Moreover, reduction operations are another important and performance critical type of synchronization in real life applications.

1) Atomic Updates: 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 several times during the benchmark by each thread.

2) Barrier: 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 is surrounded by a `target` region. The CUDA execution model [23] does not support a barrier synchronization 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 kernel launch time (with an empty kernel) with the following synchronization to wait for the kernel finalization is the closest adequate comparison to a barrier.

3) Reduction: 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 multiprocessors systems the vector is initialized in parallel, so that parts of the vector are split over different Non-Uniform memory Access [20] (NUMA) nodes in a NUMA system. It should be pointed out that such a distribution is done internally by the operating system. As CUDA does not provide reduction operations itself, the open source (CUDA-based) Thrust library [25] of Nvidia is used for this benchmark on the GPU systems.

C. Communication Performance

In the communication benchmark, we measured 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 (to and from the accelerator).

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. Dependend on the problem origin, dense or sparse matrices are used. The following two evaluation benchmarks cover both matrix types and also stress different parts of a system (these are both performance limiting for many applications also outside linear algebra).

1) Compute bound application kernel: For dense matrix multiply, with a high computational density, many techniques are known (and applied inside optimized library functions), which allow to run this operation near the peak floating point performance. Consequently, if done the right way, dense matrix multiply evaluates in essence the floating point performance of a core / processor / multiprocessor system. This operation is well examined and implemented efficiently in the BLAS library [26] and vendor optimized libraries, like the Intel MKL [27] and Nvidia cuBLAS [28].

2) Memory bound application kernel: On the other side, 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 [2] 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 [27] and Nvidia cuSPARSE [29], both with a small selection of supported storage formats. The CSR format [2] is a general format with good/reasonable performance characteristics for many sparse matrices on CPU based systems. The ELL format is, for appropriate matrices (a small and ideally constant number of non-zero elements per row), a favorable storage format on GPUs [30]. It should be pointed out, that in this benchmark we are not interested in the best possible performance for a specific matrix. We rather want to relate the performance of different systems for this type of operation in a more general way.

 ISBN: 978-1-61208-419-0

TABLE II. SELECTED HARDWARE PARAMETERS OF THE SYSTEMS USED.

Parameter	**Processor Systems**			**Accelerator Systems**			
Architecture	SB	IB	HW	KNC	M2050 (Fermi)	K20 (Kepler)	K80 ($2\times$ Kepler)
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

V. Experimental Setup

In this section, we specify our parallel system test environment where the benchmarks were applied. Additionally, we discuss parameter settings of the benchmarks, 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 last generations of Intel server processors and for accelerators the Intel Xeon Phi KNC as a many-core architecture, as well as three recent Nvidia GPU architectures. These include the most actual systems in each class. The tested accelerators use PCIe 2 (x16) for KNC, M2050 and K20and PCIe 3 (x16) for K80. 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. Otherwise this would have to be viewed as a multi-GPU setup and would not be comparable to the other acceelerators. Table II summarizes key hardware parameters of the systems used. The CPU based systems are all 2-way multiprocessor systems.

B. Test Parameters

The benchmark tests were executed with the following parameter settings:

- *Memory latency*: Variable size of the pointer table with a single threaded run.
- *Memory bandwidth*: Fixed large vector size of `STREAM_ARRAY_SIZE=40000000` and a repeat factor of `NTIMES=1000`.
- *Atomic update*: Variable number of threads according to the systems used.
- *Barrier*: Variable number of threads according to the systems used.
- *Reduction*: Variable vector size with a full parallel run.
- *Communication*: Variable size of the transferred data.
- *DGEMM*: Variable matrix size with a full parallel run.
- *SPMV*: Fixed test matrix according to the SPE10 problem [31], SPMV implementation of MKL and cuSPARSE, CSR and/or ELL format (dependent on the library).

VI. Results

In this section, we discuss the main results and concentrate on the interesting aspects. 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. On GPUs usually all stream processors of such a processor are used (with even much more concurrency in the application to hide latencies) instead of specifying the exact number of threads.

Figure 1 shows the results for the memory latency, with an access stride of 256 byte, in absolute times. Figures 2 shows these results in cycles relative to the respective base CPU/GPU clock. For all systems, 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) are clearly visible. If only absolute times are considered, as expected, one can see that all accelarators have higher latency than the processor architectures and that 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, one can see that the GPUs improve over the generations quite significantly, as the base clock is much lower. Related to relative cycles, the newest K80 outperforms the KNC and gets even close to the CPUs in access to global/main memory. So, read latency seems to be limited by the lower base clock on the K80. Looking at the different cache levels, the measurements on the M2050 and K80 GPUs show three different levels in access time which can be explained by the L1/L2 caches and accesses to the main memory. On the K20 only two levels of similar access times are visible. This is induced by different versions of the the Kepler architecture. The K20 does not cache global memory accesses in the L1 cache, but this is done in the newer generation K80. On the CPU based systems one can see the smaller L1 and L2 caches, then the larger L3 cache and at last seen 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. Therefore applications that are already latency bound have a severe problem on accelerator systems if they cannot hide this latency, e.g. by allowing multiple read requests to be open at the same time.

The memory bandwidth performance is shown in Figure 3. For the processor systems the default thread scheduling was used here, with variable numbers of threads. For graphic processors the usage model is different to a multiprocessor system, because usually all stream processors of such a processor are used instead of specifying the exact number of threads. The performance number(s) for GPUs are therefore given as a

 ISBN: 978-1-61208-419-0

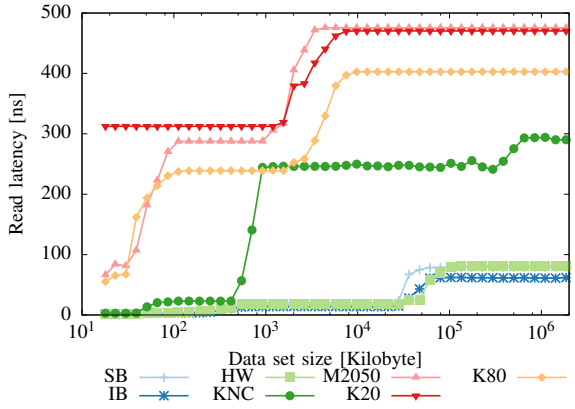

Figure 1. Memory latency results (absolute time).

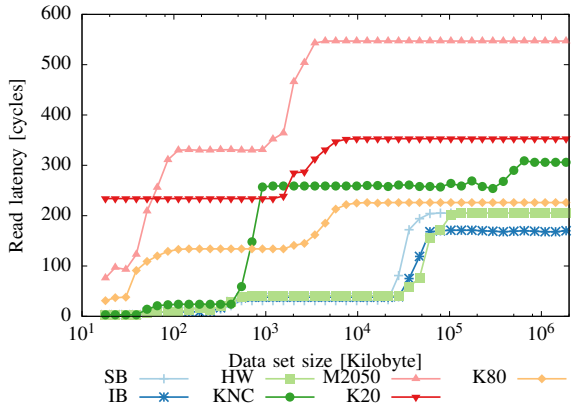

Figure 2. Memory latency results (relative cycles).

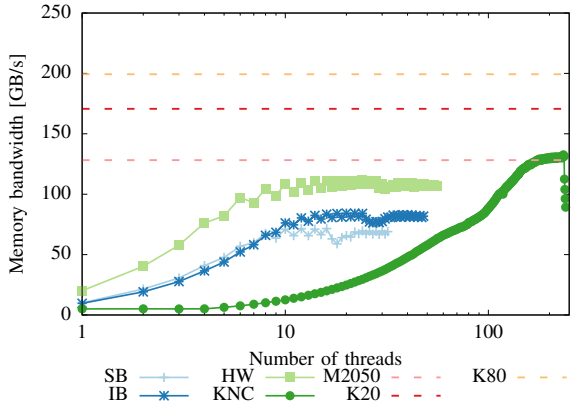

Figure 3. Memory bandwidth results.

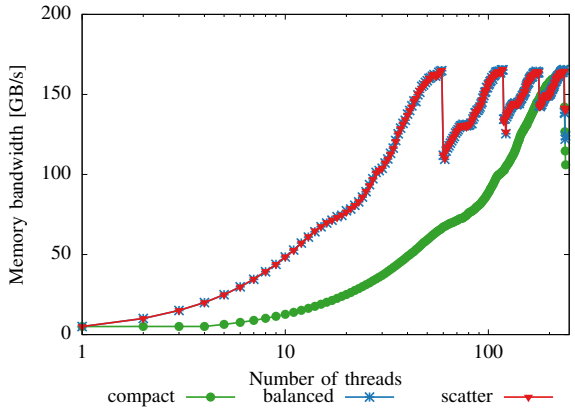

Figure 4. Memory bandwidth results on KNC, different thread mapping.

dashed line (for all stream processors used). In contrast to the results on latency, the accelerators perform better than the CPU systems. It is noteable that the KNC shows relativly bad performance here. Its 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 the highest theoretical bandwidth of all tested systems. All processor systems almost reach their theoretical bandwith.

Thread mapping / binding can be an important aspect reaching good performance. A thread mapping defines how application threads are mapped to hardware threads, e.g., processors sockets, cores in a multi-core CPU, or hardware threads in a hyperthreaded core. Basic mapping strategies are to keep threads as close as possible in the hardware (compact; e.g., to exploit data locality between threads) or to spread as wide as possible (scattered; e.g., to exploit as much memory bandwidth as possible). Figure 4 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 4) bandwith increases steadily with increasing number of threads. The performance drop with the last four threads occurs, because at this point the last core with its four hardware threads is used in the application, but that performs a busy waiting for operating system tasks (communication with the host system). When scattered thread mapping is used, the impact of the four hardware threads 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. Moreover, again the impact of the operating system core can be seen when all available threads of the KNC are used. Different to the KNC, changing the thread mapping for processor systems showed no difference at all for the benchmark.

Figure 5 shows the performance results for the atomic operation on the different systems. On the multiprocessors systems and KNC time increases linearly, proportional to the number of (competing) threads in use. Because the performance numbers show the normalized time for *one* operation, there is an increase in time *per operation* with the number of threads. This can be explained with the coherence and synchronization protocol, which is run by the processors / cores to ensure coherence and atomicity of such an operation. With more threads involved, the overhead increases [21]. For all three GPU systems the time is constant, which can be explained by the use of the unified L2 cache and the weak memory model without memory coherence. Moreover the performance improvement for atomic operations from Fermi (M2050) to Kepler (K20, 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).

 ISBN: 978-1-61208-419-0

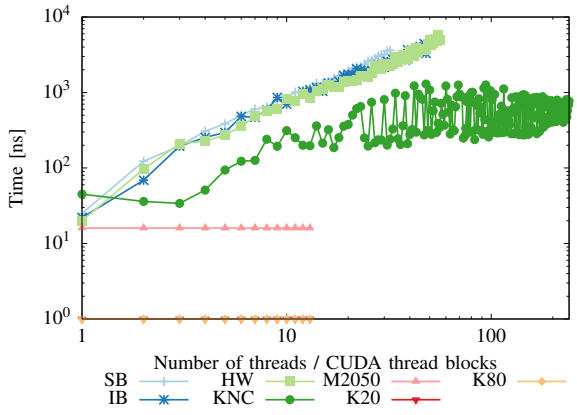

Figure 5. Atomic update results.

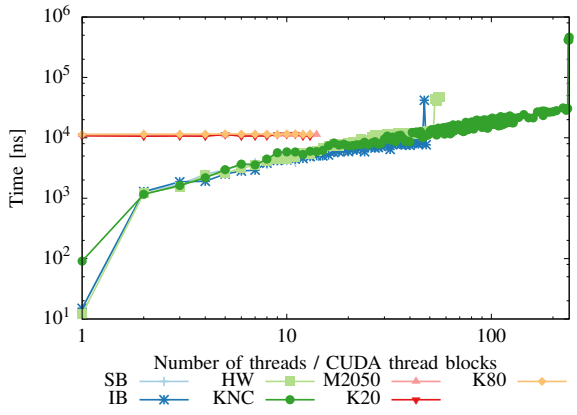

Figure 7. Barrier results.

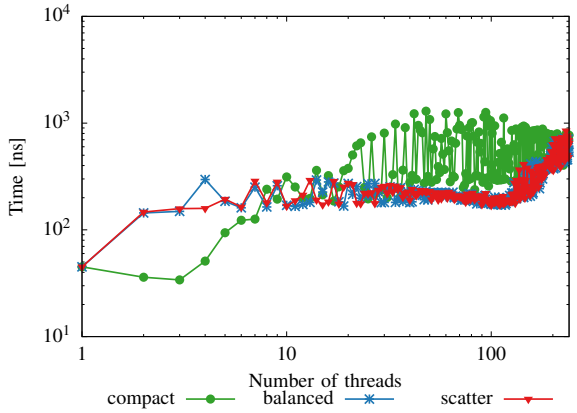

Figure 6. Atomic update results on KNC, different thread mapping.

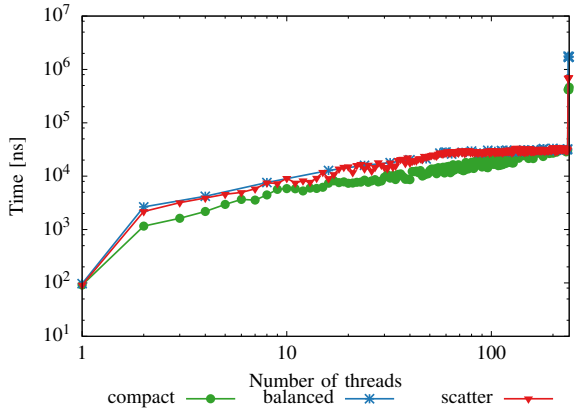

Figure 8. Barrier results on KNC, different thread mapping.

Figure 6 shows the atomic benchmark for KNC with different thread mappings. When scattered and balanced thread mapping is used the fluctuations become smaller, but still the changes of the performance with increasing number of thread is quite 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 popuated with all four hardware threads, time for an atomic update increases significantly. Again, changing the thread mapping for processor systems showed no difference at all for the benchmark.

Figure 7 illustrates performance results of the barrier test. The barrier synchronization on the Xeon Phi shows a similar behavior as on the multiprocessor systems. For the Nvidia accelerators the number of threads in the figure represents the number of used thread blocks (with 1024 threads per block). The figure shows that the kernel launch time is nearly constant and equal for M2050, K20 and K80. Further it does not depend on the number of blocks. Moreover, changing the thread mapping showed no significant differences on the multiprocessor systems. The impact of different thread mapping for the KNC can be seen in Figure 8. For balanced and scatter thread mapping, time increases steadily until all cores are populated with one thread. Afterwards, when more than one thread runs on a core, the releveance on the barrier execution time becomes less important. For compact thread mapping, the number of used cores increases steadily with the number of threads, so the time increases steadily, too.

For the reduction test, Figure 9 shows the parallel run time using all available parallelism on the system with increasing vector size. The M2050 card was limited by the available memory size and the largest vector size 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 follows the usage model of GPUs.

Figure 10 shows data transfer rates in GB/s from the host to the attached accelerator and vice versa. The results show that, for reasonable large data sizes the bus is used efficiently on all systems (the theoretical data transfer rate of 8 GB/s for PCIe 2.0 and 16 GB/s for PCIe 3.0 minus protocol overhead). For the transfer back from the accelerator to the host there is a performance drop on the M2050 reaching only approx. 5 GB/s instead of nearly 7 GB/s on the other accelerators. The low bandwidth seems to be a problem with our combination of host system and accelerator card. The data transfer rates of the KNC are low, starting at smaller data sizes (e.g. below 2 GB/s for 80,000 bytes). The difference between host to device and device to host bandwidth could be due to the Direct Memory Access (DMA) initiator. When the DMA is initiated from the CPU it has better performance. Moreover the K80 does not reach the theoretical limit for the PCIe 3. This could perhaps

 ISBN: 978-1-61208-419-0

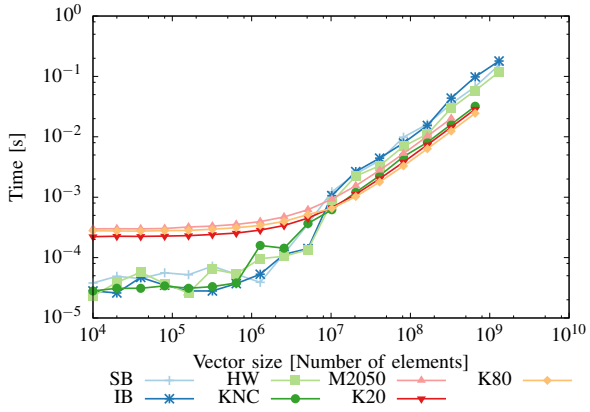

Figure 9. Reduction results.

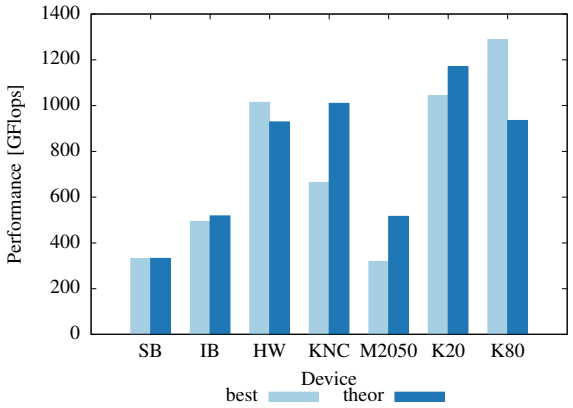

Figure 11. Dense Matrix Multiply results (DGEMM).

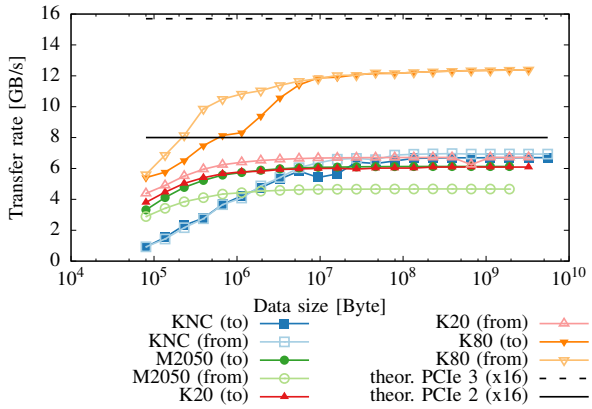

Figure 10. Communication performance results.

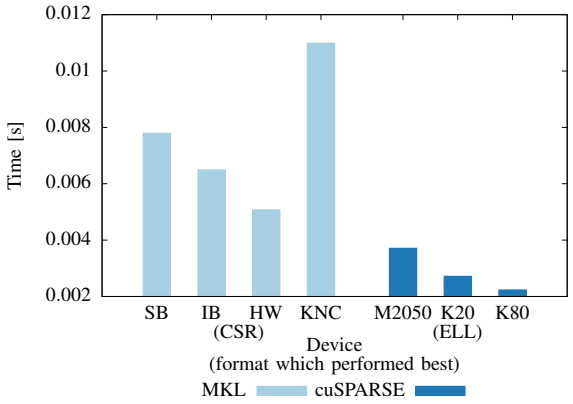

Figure 12. Sparse Matrix Vector Multiply results (SPMV).

be different when both GPUs are utilized.

In Figure 11, performance results for the dense matrix multiply operation are shown. Here, only the best performance, over all matrix sizes, is given. As expected the operation has better performance on the accelerators due to their better raw floating point performance. 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 processors which reach (nearly) one Teraflop performance. That makes it 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 architectures and get only beaten by the new K80. The K80 as well shows better performance than its given theoretical peak performance. Again, this can be explained by the use of turboboost.

For the SPMV operation shown in Figure 12 only the best results for a system and a format are given. All GPU systems perform well, compared to the multiprocessor and Xeon Phi systems. The K20 performs around 26% faster than the M2050 using ELL format. The low performance improvement of the K20 can be explained by the fact that the SPMV operation is memory bound and the memory bandwidth of the K20 is only around 25% higher compared to the M2050. Similar relations apply for K20 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 reason for that was shown in the bandwidth results where the KNC did only reach half of its peak memory bandwidth.

VII. 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 limitating in parallel applications.

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 a rather similar behavior for compute-dense problems reaching near-peak floating point performance that is quite comparable between accelerators and latest generation processors. Due to design decisions in the processor architecture graphic proces-

 ISBN: 978-1-61208-419-0

sors show a remarkable performance on some synchronization operations, operations that often limit the parallel performance.

In this paper we discussed only single-node parameters. An extension of this work would be to include cluster architectures, i.e., multiple-node architectures. Another extension could be to include multi-accelerator architechtures,e.g., using both GPUs of the K80. Further investigation would include 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] 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: Jun, 2015].

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

[3] 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.

[4] Top 500 List, http://www.top500.org/, [retrieved: Jun, 2015].

[5] SPEC CPU 2006, Standard Performance Evaluation Corporation, https://www.spec.org/cpu2006/, [retrieved: Jun, 2015].

[6] 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: Jun, 2015].

[7] 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: Jun, 2015].

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

[9] 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.

[10] 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.

[11] OpenMP Application Program Interface, 4th ed., OpenMP Architecture Review Board, http://www.openmp.org/, Jul. 2013, [retrieved: Jun, 2015].

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

[13] 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.

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

[15] 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: Jun, 2015].

[16] Nvidia CUDA, https://developer.nvidia.com/cuda-zone, [retrieved: Jun, 2015].

[17] 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: Jun, 2015].

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

[19] M. Corden, Requirements for Vectorizable Loops, Intel, https://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/, 2012, [retrieved: Jun, 2015].

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

[21] 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.

[22] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep. TM-88, 1991-2007, [retrieved: Jun, 2015]. [Online]. Available: http://www.cs.virginia.edu/stream/

[23] 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: Jun, 2015].

[24] 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.

[25] Nvidia, Thrust, https://developer.nvidia.com/thrust, [retrieved: Jun, 2015].

[26] BLAS (Basic Linear Algebra Subprograms), http://www.netlib.org/blas/, [retrieved: Jun, 2015].

[27] Intel® Math Kernel Library, https://software.intel.com/en-us/intel-mkl, [retrieved: Jun, 2015].

[28] Nvidia cuBLAS, https://developer.nvidia.com/cublas, [retrieved: Jun, 2015].

[29] Nvidia cuSPARSE, https://developer.nvidia.com/cusparse, [retrieved: Jun, 2015].

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

[31] SPE Comparative Solution Project, Society of Petroleum Engineers, http://www.spe.org/web/csp/, [retrieved: Jun, 2015].

 ISBN: 978-1-61208-419-0