OpenMP Programming on Intel® Xeon Phi™ Coprocessors: An Early Performance Comparison

Tim Cramer*, Dirk Schmidl*, Michael Klemm†, and Dieter an Mey*
*JARA, RWTH Aachen University, Germany
Center for Computing and Communication
Email: {cramer, schmidl, anmey}@rz.rwth-aachen.de
†Intel Corporation
Email: michael.klemm@intel.com

***Abstract*—The demand for more and more compute power is growing rapidly in many fields of research. Accelerators, like GPUs, are one way to fulfill these requirements, but they often require a laborious rewrite of the application using special programming paradigms like CUDA or OpenCL. The Intel® Xeon Phi™ coprocessor is based on the Intel® Many Integrated Core Architecture and can be programmed with standard techniques like OpenMP, POSIX threads, or MPI. It will provide high performance and low power consumption without the immediate need to rewrite an application. In this work, we focus on OpenMP*-style programming and evaluate the overhead of a selected subset of the language extensions for Intel Xeon Phi coprocessors as well as the overhead of some selected standardized OpenMP constructs. With the help of simple benchmarks and a sparse CG kernel as it is used in many PDE solvers we assess if the architecture can run standard applications efficiently. We apply the Roofline model to investigate the utilization of the architecture. Furthermore, we compare the performance of a Intel Xeon Phi coprocessor system with the performance reached on a large SMP production system.**

I. Introduction

Since the demand for more and more compute power is growing ever since, new architectures have evolved to satisfy this need. Accelerators, such as GPUs are one way to fulfill the requirements. They often require a time-consuming rewrite of application kernels (or more) in specialized programming paradigms, e.g. CUDA [1] or OpenCL [2]. In contrast, Intel® Xeon Phi™ coprocessors offer all standard programming models that are available for Intel® Architecture, e.g. OpenMP* [3], POSIX threads [4], or MPI [5]. The Intel Xeon Phi coprocessor plugs into a standard PCIe slot and provides a well-known, standard shared memory architecture. For programmers of higher level programming languages like C/C++ or Fortran using well established parallelization paradigms like OpenMP, Intel® Threading Building Blocks or MPI, the coprocessor appears like a symmetric multiprocessor (SMP) on a single chip. Compared to accelerators this reduces the programming effort a lot, since no additional parallelization paradigm like CUDA or OpenCL needs to be applied (although Intel Xeon Phi coprocessors also supports OpenCL).

However, supporting shared memory applications with only minimal changes does not necessarily mean that these applications perform as expected on the Intel Xeon Phi coprocessor. To get a first impression of the performance behavior of the coprocessor when it is programmed with OpenMP, we did several tests with kernel-type benchmarks and a CG solver optimized for SMP systems. These tests were done on a pre-production system, so the results might improve with the final product. We compare the results to a 128-core SMP machine based on the Bull Coherence Switch (BCS) technology and elaborate on the advantages and disadvantages of both.

The structure of this paper is as follows. First, we shortly describe the systems used in our tests in Section II and present related work in Section III. We then describe our experiments, first with kernels to investigate special characteristics of both machines (Section IV) and second with a CG type solver as it is used in many PDE solvers (Section V). We break down the CG solver into several parts and detail on the performance behavior of each part, comparing the performance of the coprocessor to the BCS-based big SMP machine. We also compare the results on both systems with an estimation of the theoretical maximum performance provided by the Roofline model [6]. Section VI concludes the paper.

II. Environment

A. Intel Xeon Phi Coprocessors

Intel recently announced the Intel® Xeon Phi™ coprocessor platform [7] that is based on the concepts of the Intel Architecture and that provides a standard shared-memory architecure. The coprocessor prototype used for the evaluation has 61 cores clocked at 1090 MHz and offers full cache coherency across all cores. Every core offers four-way simultaneous multi-threading (SMT) and 512-bit wide SIMD vectors, which corresponds to eight double-precision (DP) or sixteen single-precision (SP) floating point numbers. Fig. 1 shows the high-level architecture of the Intel Xeon Phi coprocessor die. Due to these vectorization capabilities and the large number of cores, the coprocessor can deliver 1063.84 TFLOPS of DP performance. In the system we used, the coprocessor card contained 8 GB of GDDR5 memory and it was connected via PCI Express bus to a host system with two 8-core Intel® Xeon™ E5-2670 processors and 64 GB of host main memory.

Due to the foundations in Intel architecture, the coprocessor can be programmed in several different ways. We used

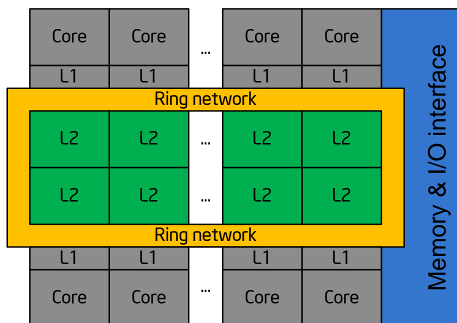

Fig. 1. High-level overview of the Intel Xeon Phi coprocessor [8].

two different ways for our experiments: 1) cross-compiled OpenMP programs natively on the coprocessor and 2) the Intel® Language Extensions for Offload (LEO) [9]. Several other ways are possible, like using MPI to send messages between the host and the coprocessor, but they have not been investigated in this work.

1) Native Execution on Intel Xeon Phi Coprocessors: All Intel Xeon Phi coprocessors execute a specialized Linux kernel providing all the well-known services and interfaces to applications, such as Ethernet, OFED, Secure Shell, FTP, and NFS. For native execution, we logged into the coprocessor and executed the benchmark from a standard shell. To prepare the application, the Intel® Composer XE 2013 on the host was instructed to cross-compile the application for the Intel Xeon Phi coprocessor (through the `-mmic` switch).

2) Language Extensions for Offload: The Intel Language Extensions for Offload offer a set of pragmas and keywords to tag code regions for execution on the coprocessor. Programmers have additional control over data transfers by clauses that can be added to the offload pragmas. One advantage of the LEO model compared to other offload programming models is that the code inside the offloaded region may contain arbitrary code and is not restricted to certain types of constructs. The code may contain any number of function calls and it can use any parallel programming model supported (e.g. OpenMP, POSIX Threads, Intel® Cilk™ Plus).

B. BCS System

For comparison we used a 16-socket 128-core system from Bull (refer to Fig. 2). The system consists of four bullx s6010 boards. Each board is equipped with four Intel Xeon X7550 (Nehalem-EX) processors and 64 GB of main memory. The Bull Coherence Switch (BCS) technology is used to combine those four boards into one SMP machine with 128 cores and 256 GB of main memory. Although this system and the Intel Xeon Phi coprocessor both contain a large number of cores accessing a single shared memory, there is a huge difference between them. The Bull system consumes 6 HU in a rack whereas the coprocessor is an extension card in the host system. Because of that, the Bull system contains much more peripheral equipment like SSDs, Infiniband HCAs and so on. Another important difference is the amount of main memory—the coprocessor has 8 GB of memory while the BCS System

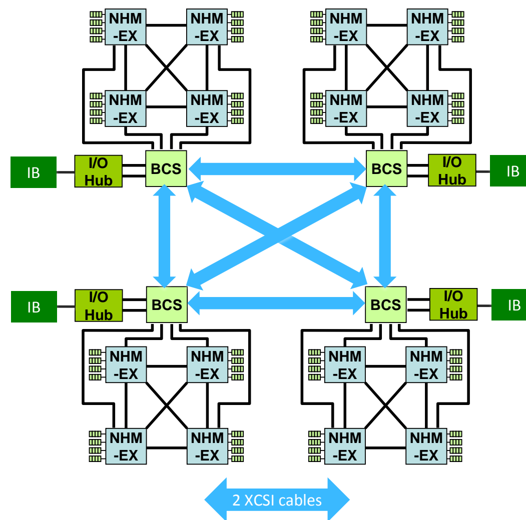

Fig. 2. High-level overview of the BCS system [10].

has 256 GB and it can easily be extended to up to 2 TB. However, many applications are tuned for these kind of SMPs and we want to investigate and compare if such applications can run efficiently on Intel Xeon Phi coprocessors. Although the BCS system contains two years old processors, both tested systems use a high number of cores and can deliver nearly the same floating point performance of about 1 TFLOPS, which makes the comparison valuable.

III. Related Work

The effort for porting scientific applications to CUDA or OpenCL can be much higher compared to directive-based programming models like OpenMP [11]. Early experiences on Intel Xeon Phi coprocessors revealed that porting scientific codes can be relatively straightforward [12], [13], which makes this architecture with its high compute capabilities very promising for many HPC applications. While [12] concentrates on the relative performance of the Intel® Knights Ferry prototype for several applications chosen from different scientific areas, we focus on absolute performance of a preproduction Intel Xeon Phi coprocessor, especially for memory-bound kernels. Heinecke et al show that the Knights Ferry prototype efficiently supports different levels of parallelism (threading and SIMD parallelism) for massive parallel applications. It has been shown that memory-bound kernels like sparse matrix vector multiplication can achieve high performance on throughput-oriented processors like GPGPUs [14] (depending on the matrix storage format), but only little knowledge is present of what the performance will be on Intel's upcoming many-core processor generation. Many applications use OpenMP already to utilize large shared memory systems. To make use of these NUMA machines, data and thread affinity has to be considered in order to obtain the best performance [15]. Taking these tuning advices into account, applications can scale to large core counts using OpenMP on these machines, like TrajSearch [16] and the Shemat-Suite [17] do.

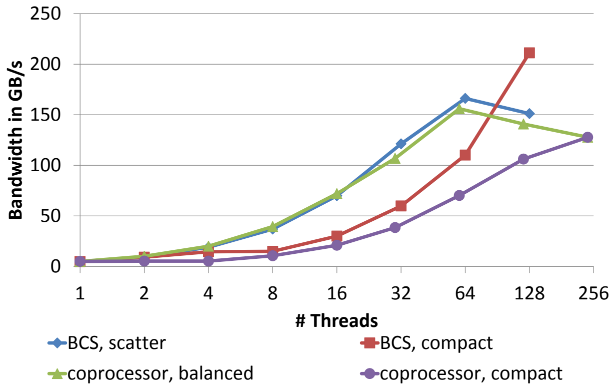

Fig. 3. Memory bandwidth of the coprocessor and the BCS system for different numbers of threads and thread-binding strategies.

IV. Basic Performance Characteristics

To get a first impression of the capabilities of the Intel Xeon Phi coprocessors, we evaluated basic performance characteristics** with kernel benchmarks. In the evaluation, we focus on the native coprocessor performance and exclude the offload model. First, we investigated the memory bandwidth of the coprocessor with the STREAM benchmark [18]. Since the memory bandwidth is the bottleneck in many sparse linear algebra kernels, this can give us a hint on the performance we can expect from the CG solver investigated in Section V. Second, we investigated the overhead of several OpenMP constructs with the help of the EPCC microbenchmarks [19]. Since the overhead of OpenMP constructs can be essential for the scaling of OpenMP applications and since applications have to scale up to hundreds of threads on the Intel Xeon Phi coprocessor, these benchmarks can give a first insight into the behavior of OpenMP applications. We compare the results with measurements on the BCS system.

A. STREAM

As described above, we use the STREAM benchmark to measure the memory bandwidth that can be achieved on each system. We use Intel Compiler XE (version 13.0.1.117) and to ensure a good thread placement we evaluate different strategies for the `KMP_AFFINITY` environment variable. To get meaningful results on the BCS system and its hierarchical NUMA design, we initialize the data in parallel in order to get a balanced data distribution across the NUMA nodes. We use a memory footprint of about 2 GB on both systems.

Fig. 3 shows the measured memory bandwidth for different numbers of threads and placement strategies on the coprocessor and on the BCS system. On the Intel Xeon Phi coprocessor we cross-compiled the benchmark and started it natively on the coprocessor. To get a good performance we needed to set compiler options in order to enable software prefetching.

On the BCS machine, we observe a difference in the binding schemes. The `compact` binding only yields small bandwidth improvements for small numbers of threads. This is because the binding first fills a whole socket before the next socket is used and so measurements with 1, 2, 4, and 8 threads only use the memory controller of one processor chip. For the 128-threads case all sockets are used and we see good performance of about 210 GB/s. With the `scatter` placement the sockets are used as soon as possible. With 16 threads all sockets and memory controllers of the system are used. We observe a fast increase of the bandwidth at the beginning, but for larger numbers of threads a plateau is reached and even slight drops are observed.

The Intel Xeon Phi coprocessor exhibits a similar behavior. The curve of the `compact` placement rises very slowly at the beginning and goes up at the end. The `compact` placement first fills the hardware threads of one physical core before going to the next. Hence, the Intel Xeon Phi achieves the best memory bandwidth when all cores are utilized. Although this seems to be quite natural, it is not the case for the Intel® Xeon™ X7750 of the BCS machine. Here using 4 of the available 8 cores is enough to saturate one chips total memory bandwidth.

The `balanced` placement [9] on the coprocessor does nearly the same as the `scatter` placement on the BCS system but the numbering of the threads is optimized, so that threads on the same core will have neighboring numbers whereas the `scatter` placement distributes the threads round-robin. The `balanced` placement achieves the best result for 60 threads, when a bandwidth of more than 156 GB/s is observed. With an increased number of threads the bandwidth goes down slightly to about 127 GB/s for 240 threads.

Overall the BCS system achieves in total an about 40 % higher memory bandwidth than the coprocessor, but of course it uses 16 processors and 16 memory controllers to do so. The coprocessor achieves a better bandwidth than 8 of the Xeon X7550 processors on a single chip which is quite an impressive result.

The memory available on the BCS system is much larger than that on the Intel Xeon Phi coprocessor, and for larger data sets the comparison would need to take into account data transfers through the PCI Express bus.

B. EPCC Microbenchmarks

The EPCC Microbenchmarks [19] are used to investigate overheads of key OpenMP constructs. The micro-benchmarks assess the performance of these constructs and provide a data point for potential parallelization overheads and the scaling behavior in real applications. Here we focus on the `syncbench` that measures the overhead of OpenMP constructs that require synchronization. Of course we expect the overhead to increase with growing numbers of threads, since more threads need to be synchronized. The overhead of the OpenMP constructs can be critical for the scaling of OpenMP applications and thus it is worthwhile to take a look at the performance on the coprocessor and to compare it to the BCS system while running with a similar number of threads. Table I shows the overhead of the OpenMP `parallel for`, `barrier` and `reduction` constructs. The experiments were done on the BCS system and on the coprocessor with the original EPCC benchmark code. We cross-compiled the code for the Intel

BCS System			
#Threads	PARALLEL FOR	BARRIER	REDUCTION
1	0.27	0.005	0.28
2	8.10	2.50	7.34
4	9.55	4.69	9.75
8	18.63	8.52	27.18
16	22.78	8.83	37.46
32	25.16	12.34	42.47
64	43.56	15.57	60.63
128	59.04	20.61	80.79
Intel Xeon Phi coprocessor (native / offload)			
#Threads	PARALLEL FOR	BARRIER	REDUCTION
1	2.01 / 2.41	0.08 / 0.10	2.31 / 2.59
2	4.32 / 7.17	1.28 / 1.70	4.28 / 7.77
4	7.63 / 8.86	2.49 / 3.47	7.39 / 10.08
8	12.24 / 11.60	4.56 / 4.56	12.39 / 12.68
16	13.81 / 12.59	5.83 / 6.46	21.60 / 22.42
30	15.85 / 16.86	8.20 / 8.34	24.79 / 27.88
60	17.71 / 21.19	9.96 / 9.96	29.56 / 35.33
120	20.47 / 24.65	11.79 / 12.28	34.61 / 41.70
240	27.55 / 30.39	13.36 / 16.66	48.86 / 52.17

TABLE I
OVERHEAD IN MICROSECONDS FOR OPENMP CONSTRUCTS MEASURED WITH THE EPCC MICROBENCHMARK `SYNCBENCH` ON THE BCS SYSTEM AND AN THE INTEL XEON PHI COPROCESSOR. HERE, THE BENCHMARKS WERE RUN NATIVELY ON THE COPROCESSOR AND STARTED WITH AN OFFLOAD DIRECTIVE FROM THE HOST SYSTEM.

```
start   = getclock();
#pragma offload target(mic)
for (j=0; j<innerreps; j++){
  delay(delaylength);
}
times[k] = (getclock() - start);
```

Fig. 4. Kernel to compute the reference time of `innerreps` executions of `delay` on the coprocessor

```
start   = getclock();
for (j=0; j<innerreps; j++){
#pragma offload target(mic)
  {
    delay(delaylength);
  }
}
times[k] = (getclock() - start);
```

Fig. 5. Kernel to compute the time to offload `innerreps` times a kernel that executes the `delay` function

Xeon Phi coprocessor and started it natively on the device. We also measured a slightly modified version of the code using LEO to offload all parallel regions.

The first thing to note is that there is no big performance difference between the native coprocessor version and the hybrid version using LEO. On both investigated systems the overhead is in the same range, although the scaling is slightly better on the Intel Xeon Phi coprocessor. Comparing for example the results of 128 threads on the BCS system with 120 threads on the coprocessor, we observe that the coprocessor achieves faster synchronization for all constructs investigated. It is obvious that the physical distance on the BCS system is much higher than the distance on the coprocessor chip. Overall, this is a sign that applications scaling on the big SMP system might also scale well on a coprocessor since synchronization is cheaper there.

Finally, we extended the original EPCC benchmark set by a benchmark that measures the overhead of the `offload` pragma itself. We applied the same procedure as it is done for the other constructs. We did a reference run that measured the overhead of a `delay` function `innerreps` times (see Fig. 4) and then we measured the time to offload and execute the `delay` function `innerreps` times (see Fig. 5).

This allows to calculate the overhead as:

```
(OffloadTime - ReferenceTime)/innerreps
```

The overhead we observed for the offload directive on our test system was 91.1 μs. Thus, the overhead of one offload region is about 3 times larger than that of a `parallel for` construct with 240 threads on the coprocessor. The ability of the coprocessor to handle function calls and other high-level programming constructs allows to offload rather large kernels and helps hide the overhead in the computation.

V. Conjugate Gradient Method

To evaluate the performance of a real-world compute kernel we use a CG solver [20] that runs natively on the Intel Xeon Phi coprocessor. The runtime of the algorithm is dominated by the Sparse-Matrix-Vector-Multiplication (SMXV). For the performance evaluation we use our own implementation that uses OpenMP `for` constructs to parallelize all operations and we compare it to a version that uses the Intel® Math Kernel Library (MKL) sparse routines. We use the Compressed Row Storage (CRS) format to store only the non-zero values and the sparsity pattern of the matrix and to have a cache-friendly memory access.

Depending on the sparsity pattern of the matrix an adequate load balancing is also needed. For that reason we do not use a static schedule for the distribution of the matrix rows but rather pre-calculate the number of rows for each thread depending on the number of nonzero values. On big ccNUMA machines correct data and thread placement is essential [15], so we initialize the data in parallel to distribute the pages over the sockets and bind the threads to the cores to avoid thread migration. Since the two test systems differ in amount and usability of hardware threads we use different binding strategies. For the 16-sockets machine we set `KMP_AFFINITY=scatter` to fill up the sockets round-robin and `KMP_AFFINITY=compact` to place each thread as close as possible to the previous thread. For the thread placement on the Intel Xeon Phi coprocessor, we set `KMP_AFFINITY=balanced,granularity=fine` to achieve a more balanced thread placement given the four available hardware threads on each core and to obtain the best performance for this kernel. The matrix represents a computational fluid dynamics problem (Fluorem/HV15R) and is taken

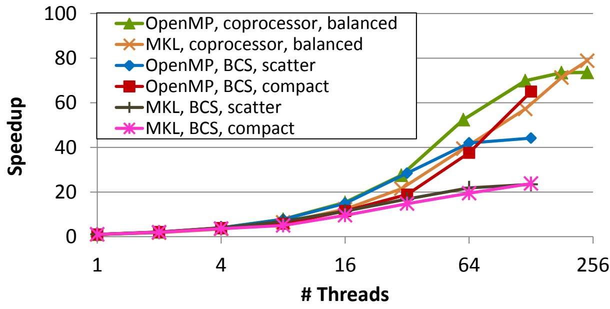

Fig. 6. Scalability of the CG kernel on 128-core SMP machine and on the coprocessor.

Test case	#Threads	Serial Time [s]	Minimal Time [s]
OpenMP, coprocessor, balanced	244	2387.40	32.24
MKL, coprocessor, balanced	244	3014.48	38.46
OpenMP, BCS, scatter	128	1175.89	26.63
OpenMP, BCS, compact	128	1176.81	18.10
MKL, BCS, scatter	128	1024.52	43.72
MKL, BCS, compact	128	1025.99	43.13

TABLE II
SERIAL AND MINIMAL PARALLEL RUN TIME OF 1000 ITERATIONS OF THE CG METHOD FOR BOTH IMPLEMENTATIONS ON BOTH SYSTEMS.

from the University of Florida Sparse Matrix Collection [21]. The matrix dimension is $N = 2,017,169$ and the number of nonzero elements is $nnz = 283,073,458$, which results in a memory footprint of approximately 3.2 GB. Hence, the data set is big enough not to fit into the caches, even on the 16-sockets machine.

Fig. 6 shows the scalability on the 16-socket machine and on the coprocessor for 1,000 CG iterations. If all 61 cores are utilized (without SMT) on the coprocessor, it reaches a speedup of over 53 with our CG implementation (OpenMP). With all hardware threads the speedup increases up to over 74. Although the speedup curve increases more slowly for the Intel MKL version, a better speedup is reached if all hardware threads are used. However, the total elapsed time to execute 1,000 iterations is 20 % higher than for our cross-compiled version (also see Table II). The figure shows that the Intel Xeon Phi coprocessor can reach a very high scalability, even for this memory-bound problem. In contrast to that, the speedup of our (OpenMP) implementation on the big SMP system with the `scatter` placement strategy only increases slightly when utilizing all 128 cores without SMT compared to using only 64 threads. The `compact` thread distribution reaches a speedup of 65. It increases from 64 to 128 threads, because the former case uses only two out of the four 4-sockets boards and the full memory bandwidth is not available.

The consideration of the speedup does not take the absolute performance into account. Table II shows the serial and the minimal parallel runtime for 1,000 iterations of the CG method for both implementations and the number of threads which obtain the minimal parallel run time. As the CG method is dominated by the SMXV and it is also important for many

system	#Threads	read bandwidth	write bandwidth
Coprocessor	244	122.1 GB/s	62.9 GB/s
BCS	128	236.5 GB/s	142.2 GB/s

TABLE III
MEMORY BANDWIDTH FOR ONLY READ AND ONLY WRITE OPERATIONS ON THE BCS SYSTEM AND THE COPROCESSOR CARD.

other sparse algorithms, we focus on the performance of this operation in the following section.

A. *Sparse-Matrix-Vector-Multiplication (SMXV)*

To assess the absolute performance, we apply a simplified version of the Roofline model [6] to the BCS system (Fig. 7) and to the Intel Xeon Phi coprocessor (Fig. 8). The Roofline model gives an estimate of the maximum performance in GFLOPS an algorithm can achieve on a given system depending on its operational intensity. The operational intensity is the number of FLOPS per byte loaded into the system. Algorithms that compute many FLOPS per byte are bound by the peak floating point performance of the system modeled as a horizontal line in the right part of the diagram. The rising line in the left part of the diagram shows the maximum performance an algorithm with low operational intensity can reach because of the limited memory bandwidth of the system. The point of intersection of both lines marks the operational intensity where the algorithm fully utilizes the bandwidth and the floating point capacity of the system.

We obtained the peak floating point performance of the systems from the hardware specifications of the Intel Xeon X7550 processors and the Intel Xeon Phi coprocessor. The peak memory bandwidth was measured with a STREAM-like test program. In contrast to the original stream code used in Sec. IV we calculate the bandwidth for only read operations and only write operations. The results are shown in Table III. As you can see on both systems the bandwidth for read operations is higher compared to the bandwidth for write operations. This is because the processor needs to load a cache line first before it can be written, which results in two transfers through the memory controller for a write operation whereas only one transfer is needed for read (assuming no streaming stores). To simplify matters, we assume that the vectors of the SMXV kernel can be kept in the cache, so only the matrix needs to be loaded whose elements are stored consecutively. So, for the Roofline model we used the read memory bandwidth as basis for the rising left line.

To load one entry of the matrix in CRS format we have to load the value (`double`) and index (`int`) variables, resulting in 12 bytes to load per entry. One add and one multiply operation is needed per matrix element and each matrix element is needed only once, so we cannot keep results in the cache to be reused later on. This leads to an operational intensity (O) of $O = \frac{2\ FLOPS}{12\ bytes} = \frac{1}{6}\frac{FLOPS}{byte}$. For this operational intensity the Roofline model shows that the SMXV kernel is bound by the memory bandwidth on both systems and that it can reach a maximum performance of 39.42 GFLOPS on the BCS system and 20.35 GFLOPS on the coprocessor.

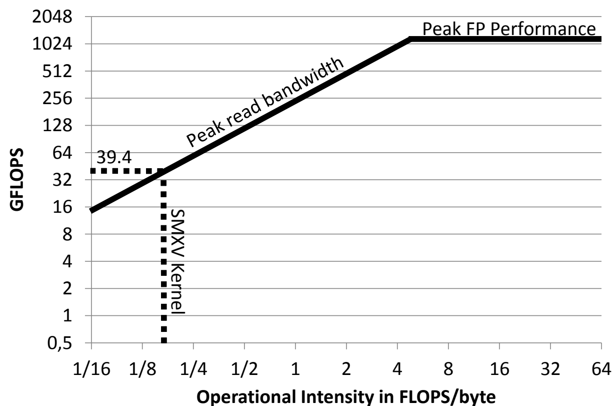

Fig. 7. Roofline model applied to the BCS system. The model shows that the SMXV kernel which has an operational intensity of 1/6 FLOPS/byte can reach a maximum performance of 39.4 GFLOPS on the BCS system.

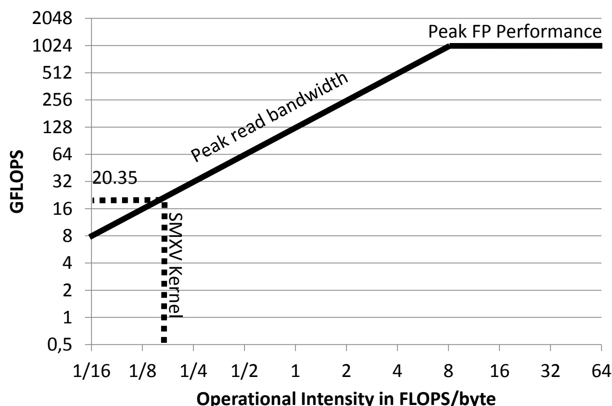

Fig. 8. Roofline model applied to the Intel Xeon Phi coprocessor. The model shows that the SMXV kernel which has an operational intensity of 1/6 FLOPS/byte can reach a maximum performance of 20.35 GFLOPS on the Intel Xeon Phi coprocessor.

Fig. 9 shows the performance measured for SMXV within the CG method. As expected, the best performance is achieved on the BCS system (with `compact` distribution strategy) which delivers 37.24 GFLOPS for our implementation, which is close to the theoretical maximum of 39.42 GFLOPS as predicted by the Roofline model. For the Intel MKL version only 31.64 GFLOPS are reached. Here, the reason for the lower performance on this big BCS system is that our own implementation is less generic and that we have the full control of the correct data and thread placement. The slower performance increase for the `scatter` strategy when utilizing more than 64 threads is directly connected to the `STREAM` measurement of Fig. 3. Due to the bandwidth drop with large thread counts, the best result is obtained with 64 threads. As predicted by the Roofline model with 18.67 GFLOPS the performance on the coprocessor is also close to the maximum of 20.35 GFLOPS. While the difference between the Intel MKL version and our own implementation is about 5.5 GFLOPS without SMT (61 threads), the maximum performance of both versions (244 threads) is almost the same.

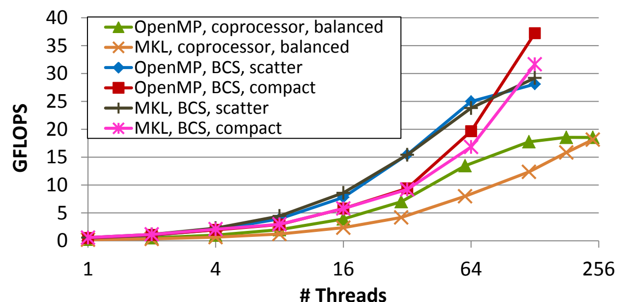

Fig. 9. Performance of the SMXV within the CG method on 128-core SMP machine and on the coprocessor.

System	#Threads	Time [s]	dxpay/ daxpy	dot product	SMXV
Coprocessor	244	32.24	3.71 %	1.89 %	94.03 %
BCS	128	18.10	11.41 %	4.45 %	84.01 %

TABLE IV
RUN TIME SHARES AND SOLVING TIMES FOR THE LINEAR ALGEBRA KERNELS ON BOTH SYSTEMS.

B. Vector Operations

Accelerators like GPGPUs often require to not only optimize the SMXV kernel, but also other vector operations within the CG solver (e.g., reductions for the dot product). Table IV shows that the share of vector operations in SMXV is more dominant on the Intel Xeon Phi coprocessor than on the BCS machine (94.03 % vs. 84.01 %). The vector operations daxpy ($\vec{y} = a * \vec{x} + \vec{y}$) and dxpay ($\vec{y} = \vec{x} + a * \vec{y}$) take only 3.71 % of the total runtime while they consume up to 11.41 % on the BCS system. After the matrix vector multiplication most of the vectors are discarded from the cache and need to be reloaded. In contrast to the coprocessor, the BCS system cannot reach full memory bandwidth for these small arrays (about 15 MB each). The share of the dot product is lower on the coprocessor which shows that the OpenMP reduction works fine on this new architecture. Since the SMXV kernel has the most significant run time impact we concentrated our analysis on it.

C. SIMD Vectorization

As mentioned in Section II, the Intel Xeon Phi coprocessor has new powerful SIMD capabilities. To analyze these, we built two versions of the CG method with and without vectorization (through the compiler flag `-no-vec`). Fig. 10 shows that the gain from vectorization within the CG method is quite small. Although one observes a 2x gain for small numbers of threads, there is no difference when the coprocessor is fully utilized. In combination with the presented performance model, one can see that it is possible to utilize the full available bandwidth although the code was not vectorized. Since the data loaded through the gather operations is the same as with the scalar version of the CG code, the gather operations cannot provide any additional performance gain. This means that memory-bound applications in general can benefit from the

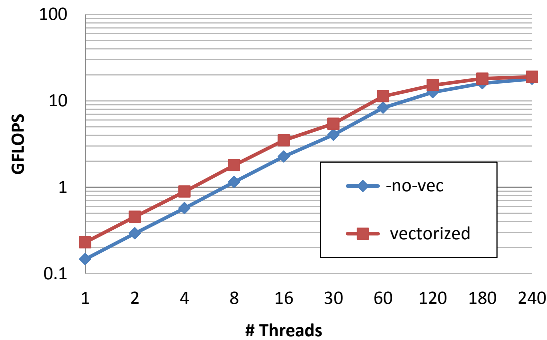

Fig. 10. Performance of the SMXV within a CG method with and without vectorization on the Intel Xeon Phi coprocessor.

high memory bandwidth of the coprocessor, even if they do not use SIMD vectors.

VI. Conclusion and Future Work

The growing demand for both compute power and low energy consumption has led to a growing adaption of accelerators and coprocessors for HPC to fulfill this requirement. While for GPUs a rewrite of the application might be necessary, we have shown that with the upcoming Intel Xeon Phi coprocessor it is possible to efficiently port compute kernels with no or just minor code modifications.

The overhead of the standard OpenMP constructs which use synchronization is smaller than on big SMP machines, which makes the approach very promising for many HPC applications using OpenMP. The overhead of the offload pragma used in the language extension (LEO) is also quite low, so that it will not limit the scalability.

The bandwidth of one coprocessor is up to 156 GB/s and exceeds eight Intel Xeon X7550 processors. This is reflected in real-world kernels like the SMP optimized sparse CG solver. With the Roofline model we have predicted a maximum performance of about 20 GFLOPS for the SMXV kernel (limited by the read memory bandwidth) and achieved almost 19 GFLOPS with the unmodified kernel. This shows that scientific OpenMP applications can run efficiently on the upcoming Intel Xeon Phi coprocessor without requiring a rewrite.

In addition to our work which focused on shared memory programming, it would be interesting to investigate the performance of the Intel Xeon Phi coprocessor for distributed memory paradigms like MPI on one or multiple coprocessors.

Acknowledgments

Parts of this work were funded by the German Federal Ministry of Research and Education (BMBF) as part of the LMAC project (Grant No. 01IH11006).

Intel, Xeon, and Xeon Phi are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other brands and names are the property of their respective owners.

** Performance tests are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. System configuration: Intel® W2600CR baseboard with 2S Intel® Xeon™ processor E5-2670 (64 GB DDR3, with 1600 MHz, Scientific Linux 6.2) and single Intel® C600 IOH, Intel® Xeon Phi™ coprocessor with B0 ES2 silicon (GDDR5 with 5.5 GT/sec, driver v2.1.4346-16, flash v2.1.01.0375, OS v2.6.32-220, Intel® Composer XE 2013 v13.0.1.117).

References

[1] NVIDIA, "CUDA C Programming Guide, Version 4.2," April 2012.

[2] Khronos OpenCL Working Group, "The OpenCL Specification, Version 1.1, Revision 44," June 2011.

[3] OpenMP Architecture Review Board, "OpenMP Application Program Interface, Version 3.1," July 2011.

[4] U. Drepper and I. Molnar, "The Native POSIX Thread Library for Linux," Redhat, Tech. Rep., February 2003.

[5] MPI Forum, "MPI: A Message-Passing Interface Standard, Version 3.0," July 1997.

[6] S. Williams, A. Waterman, and D. Patterson, "Roofline: an Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, April 2009.

[7] Intel Corporation, "Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual," September 2012, reference number 327364-001.

[8] A. Heinecke, M. Klemm, and H.-J. Bungartz, "From GPGPUs to Many-Core: NVIDIA Fermi* and Intel® Many Integrated Core Architecture," *Computing in Science and Engineering*, vol. 14, no. 2, pp. 78–83, March–April 2012.

[9] Intel Corporation, "Intel® C++ Compiler XE 13.0 User and Reference Guides," September 2012, document number 323273-130US.

[10] D. Gutfreund, "Mesca BCS Systems," Bull SAS, rue Jean Jaurs, 78340 Les Clayes sous Bois, France, October 2012.

[11] S. Wienke, D. Plotnikov, D. an Mey, C. Bischof, A. Hardjosuwito, C. Gorgels, and C. Brecher, "Simulation of bevel gear cutting with GPGPUs - performance and productivity," *Computer Science - Research and Development*, vol. 26, pp. 165–174, 2011.

[12] K. W. Schulz, R. Ulerich, N. Malaya, P. T. Bauman, R. Stogner, and C. Simmons, "Early Experiences Porting Scientific Applications to the Many Integrated Core (MIC) Platform," TACC-Intel Highly Parallel Computing Symposium, Tech. Rep., April 2012.

[13] A. Heinecke, M. Klemm, D. Pflüger, A. Bode, and H.-J. Bungartz, "Extending a Highly Parallel Data Mining Algorithm to the Intel® Many Integrated Core Architecture," in *Euro-Par 2011: Parallel Processing Workshops*, Bordeaux, France, August 2011, pp. 375–384, LNCS 7156.

[14] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," NVIDIA Corporation, Tech. Rep. NVR-2008-004, December 2008.

[15] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and Thread Affinity in OpenMP Programs," in *Proc. of the 2008 Workshop on Memory Access on Future Processors: a Solved Problem?*, Ischia, Italy, May 2008, pp. 377–384.

[16] N. Berr, D. Schmidl, J. H. Göbbert, S. Lankes, D. an Mey, T. Bemmerl, and C. Bischof, "Trajectory-Search on ScaleMP's vSMP Architecture," *Advances in Parallel Computing: Applications, Tools and Techniques on the Road to Exascale Computing*, vol. 22, pp. 227–234, 2012.

[17] D. Schmidl, C. Terboven, A. Wolf, D. an Mey, and C. H. Bischof, "How to Scale Nested OpenMP Applications on the ScaleMP vSMP Architecture," in *Proc. of the IEEE Intl. Conf. on Cluster Computing*, Heraklion, Greece, September 2010, pp. 29–37.

[18] J. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," http://www.cs.virginia.edu/stream, 1999, [Online, accessed 29-March-2012].

[19] J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP," in *Proc. of the 1st European Workshop on OpenMP*, Lund, Sweden, October 1999, pp. 99–105.

[20] M. R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, December 1952.

[21] T. A. Davis, "University of Florida Sparse Matrix Collection," *NA Digest*, vol. 92, 1994.