Extending a Highly Parallel Data Mining Algorithm to the Intel® Many Integrated Core Architecture

Alexander Heinecke[1], Michael Klemm[3], Dirk Pflüger[1], Arndt Bode[2], and Hans-Joachim Bungartz[2]

[1] Technische Universität München, Boltzmannstr. 3, D-85748 Garching, Germany
[2] Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften, Boltzmannstr. 1, D-85748 Garching, Germany
[3] Intel GmbH, Dornacher Str. 1, D-85622 Feldkirchen, Germany

Abstract. Extracting knowledge from vast datasets is a major challenge in data-driven applications, such as classification and regression, which are mostly compute bound. In this paper, we extend our SG^{++} algorithm to the Intel® Many Integrated Core Architecture (Intel® MIC Architecture). The ease of porting an application to Intel MIC Architecture is shown: porting existing SSE code is very easy and straightforward. We evaluate the current prototype pre-release coprocessor board codenamed Intel® "Knights Ferry". We utilize the pragma-based offloading programming model offered by the Intel® Composer XE for Intel MIC Architecture, generating both the host and the coprocessor code. We compare the achieved performance with an NVIDIA C2050 accelerator and show that the pre-release Knights Ferry coprocessor delivers better performance than the C2050 and exceeds the C2050 when comparing the productivity aspect of implementing algorithms for the coprocessors.

Keywords: Intel® Many Integrated Core Architecture, Intel® MIC Architecture, Intel® Knights Ferry, NVIDIA Fermi*, GPGPU, accelerators, coprocessors, data mining, sparse grids.

1 Introduction

Experts expect that future exascale supercomputers will likely be based on heterogeneous architectures that consist of a moderate amount of "fat" cores and use a large number of accelerators or coprocessors to deliver a high ratio of GFLOPS/Watt [21]. Today, Graphic Processing Units (GPU) are very popular for accelerating highly parallel kernels like dense linear algebra or Monte Carlo simulations [20,8]. However, the performance increase is not for free and requires the ability to rewrite compute kernels in GPU-specific languages such as CUDA [13] or OpenCL [10]. This implies serious porting and tuning effort for legacy compute-intensive applications (CPU-optimized codes), which are executed in thousands of compute centers every day.

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part II, LNCS 7156, pp. 375–384, 2012.

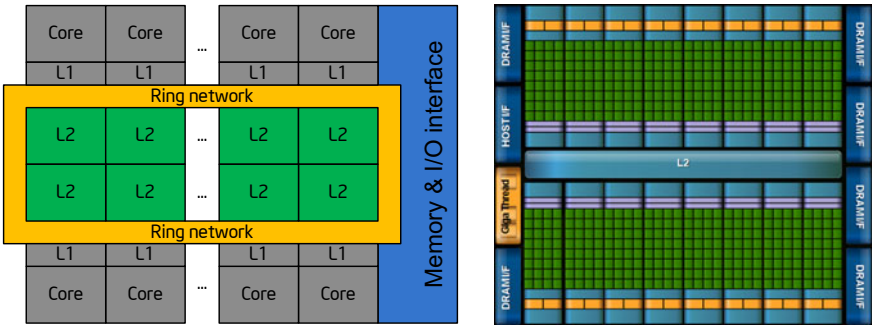

Fig. 1. High-level view on the Intel MIC Architecture (left) and NVIDIA Fermi (right) taken from [12]

The Intel® Many Integrated Core Architecture (Intel® MIC Architecture) is a massively parallel coprocessor based on Intel Architecture (IA). The existing tool chain for software development on IA can be used to implement applications for the Intel MIC Architecture. All traditional HPC programming models such as OpenMP* and MPI* on top of C/C++ and Fortran will be available. Developers do not need to accept the high learning curve and implementation effort to (partially) rewrite their source code to retrofit it for a GPU-based accelerator.

In this paper, we compare a pre-release Intel coprocessor ("Knights Ferry") of the Intel MIC Architecture with a recent NVIDIA Tesla* C2050 GPU (Sect. 2). We focus on the performance of an existing highly parallel workload and assess the programming productivity during implementation. We use the SG^{++} data-mining algorithm (Sect. 3) as the workload for the evaluation. As with most HPC applications, SG^{++} is already available as highly optimized code for processors compatible to Intel® Xeon®. Hence, we use this as our starting point for the evaluation. The paper carries on with comparing the implementations and performance in Sect. 4. For the comparison, we restrict ourselves to genuine compilers and toolkits to ensure that the optimal software stack for the compute platforms is evaluated.

2 Intel MIC Architecture in Comparison to NVIDIA Fermi Architecture

In this section, we will investigate the differences and similarities between the Intel MIC Architecture [7] and the NVIDIA Tesla 2050 accelerator [12]. The Intel MIC Architecture has been announced at the International Supercomputing Conference [18] as a massively parallel coprocessor based on IA. It is currently available as pre-release hardware code-named *Knights Ferry* (based on Intel's previous Larrabee design [9]).

Fig. 1 gives an overview of the respective architectures. Knights Ferry offers 32 general-purpose cores with a fixed frequency of 1200 MHz. The cores are based

on a refreshed Intel® Pentium® (P54C) processor design [3] and have been extended with 64-bit instructions and a 512-bit wide Vector Processing Unit (VPU). Each of the cores offers four-way round robin scheduling of hardware threads, i. e., in each clock cycle a core switches to the next instruction stream.

The cores of the Knights Ferry coprocessor own a local L1 and L2 cache with 32 KB and 256 KB, respectively. With a total of 32 cores, this coprocessor offers a total of 8 MB shared L2 cache. The cores are connected through a high-speed ring-bus that interconnects the L2 caches for fast on-chip communication. An L3 cache does not exist in this design because of the high-bandwidth GDDR5 memory (1800 MHz). In total, the memory subsystem delivers a peak memory bandwidth of 115 GB/sec.

Since the Intel MIC Architecture is based on IA, it can support the programming models that are available for traditional IA-based processors. The compilers for the Intel MIC Architecture support Fortran (including Co-Array Fortran) and C/C++. OpenMP [15] and Intel® Threading Building Blocks [17] may be used for parallelization as well as emerging parallel languages such as Intel® Cilk™ Plus [6] or Intel® Array Building Blocks [5]. The VPU can be accessed through the auto-vectorization capabilities of the Intel compiler as well as low-level programming through intrinsic functions. The Intel MIC Architecture greatly simplifies programming, as well-known traditional programming models can be utilized to implement codes for it.

In contrast to the Intel MIC Architecture, the Tesla 2050 architecture [12] does not contain general purpose compute cores. Instead it consists of 14 multiprocessors with 32 processing elements each. The processing elements run at a clock speed of 1.15 GHz and a memory-bandwidth of 144 GB/sec. A 768 KB L2 cache is shared across the 14 multiprocessors.

Because of its special architecture the 2050, it only supports a limited set of programming models. The most important ones are CUDA and OpenCL, which are data-parallel programming languages that do not support arbitrary (task-)parallel programming patterns. Some production compilers, such as the PGI compiler suite [19] or HMPP [2] support offloading of Fortran code to the GPU, but restrict the language features in order to fit the GPU programming model.

3 Data Mining Using Sparse Grids

Regression and classification, considered as scattered data approximation problems, both start from m known observations, $S = \{(\boldsymbol{x}_i, y_i) \in \mathbb{R}^d \times \mathbb{R}\}_{i=1,\ldots,m}$, with the aim to recover (learn) the functional dependency $f(\boldsymbol{x}_i) \approx y_i$ as accurately as possible. Reconstructing a smooth function f then allows an estimate $f(\boldsymbol{x})$ for new properties $\boldsymbol{x}$.

We aim for representations $f = \sum_{j=1}^{N} \alpha_j \varphi_j(\boldsymbol{x})$ as a linear combination of N basis functions $\varphi_j(\boldsymbol{x})$ with coefficients α_j. To obtain an algorithm that scales only linearly in m, we associate the basis functions to grid points on some grid, rather than fitting their centers to the data. Here, we are considering piecewise d-linear functions. Unfortunately, regular grid structures suffer from the *curse of*

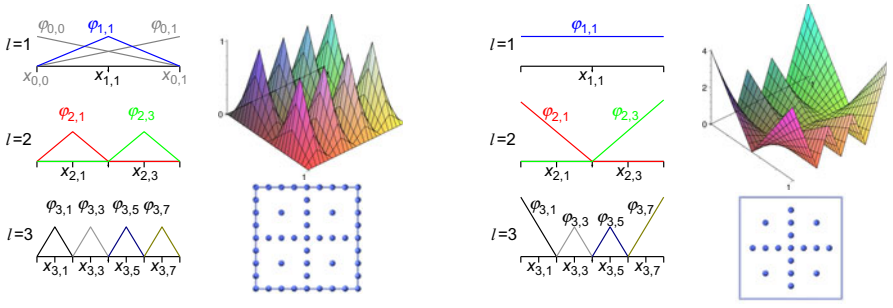

Fig. 2. Classical basis functions for the first three levels of discretization in 1d (left) and modified ones with different types of basis functions (right). For both, a selection of $2d$ basis functions and a $2d$ sparse grid of level 3 is shown.

dimensionality: regular grid with equidistant meshes and k grid points in each dimension contain k^d grid points in d dimensions. The exponential growth typically prevents considering more than 4 dimensions for reasonable discretizations.

We rely on *adaptive sparse grids* (see [1,16] for details) to mitigate the curse of dimensionality. They are based on a hierarchical grid structure with basis functions defined on several levels of discretization in $1d$, a hierarchical basis, and d-dimensional basis functions as products of one-dimensional ones. We employ two kinds of basis functions: *uniform* and *modified non-uniform*. Uniform basis functions lead to grids with a large number of grid points on the domain's boundary, whereas modified non-uniform ones extrapolate towards the domain's boundary, which lead to a smaller grid structure; see Fig. 2.

The hierarchical tensor product approach allows to represent a function on several scales. Trying to find out which scales contribute most to the overall solution, it can been seen that plenty of grid points can be omitted in the hierarchical representation as they have only little contribution—at least for sufficiently smooth functions. The costs are reduced from $\mathcal{O}(k^d)$ to $\mathcal{O}(k \log(k)^{d-1})$, maintaining a similar accuracy as for full grids.

The function f should be as close to the data S as possible, minimizing the mean squared error. At the same time, close data points should very likely have similar function values to generalize from the data. We minimize the trade-off between both regularization parameter λ (Eq. 1, left) and the hierarchical basis allowing for a simple generalization functional. This leads to a system of linear equations (Eq. 1, right), with matrix B, $B_{i,j} = \varphi_i(\boldsymbol{x}_j)$, and identity matrix I.

$$\arg\min_f \frac{1}{m} \sum_{i=1}^{m} \left(f(\boldsymbol{x}) - y_i\right)^2 + \lambda \sum_{j=1}^{N} \alpha_j^2 \quad \Rightarrow \quad \left(\frac{1}{m} B B^T + \lambda I\right) \boldsymbol{\alpha} = \frac{1}{m} B \boldsymbol{y}. \quad (1)$$

Because of the storage required for the large matrices, the linear system is solved iteratively, with repeated recomputation of B and B^T. Both correspond to function evaluations, as $(B^T \alpha)_i = f(\boldsymbol{x}_i)$. Unfortunately, from a parallelization point

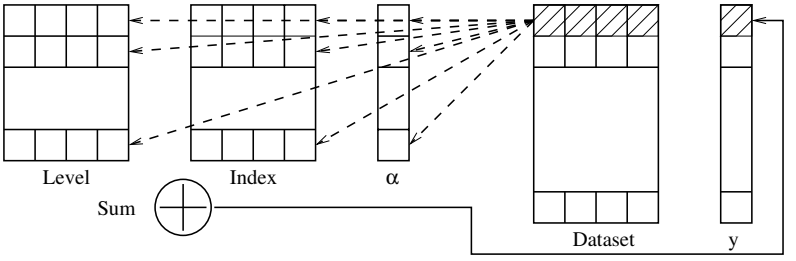

Fig. 3. Data containers to manage adaptive sparse grids and datasets for streaming access

of view, efficient algorithms for function evaluations on sparse grids are inherently multi-recursive in both level and dimensionality. This imposes severe restrictions on parallelization and vectorization, especially on accelerators.

A straightforward alternative approach evaluates all basis functions (even those resulting to zero) for all data points and sums up the results as shown in Fig. 3. This is less computationally efficient, but streaming access of the data and the avoidance of recursive structures and branching easily pays back the additional computation: it is arbitrarily parallelizable and can be vectorized.

In the following, we use two test scenarios, both with a moderate dimensionality of $d = 5$ and distinct challenges. The first dataset with 2^{18} data points classifies a regular $3 \times \cdots \times 3$ checkerboard pattern. The second one is a real-world dataset from astrophysics, predicting spectroscopic redshifts of galaxies based on more than 430,000 photometric measurements. For both, excellent numerical results are obtained using our method, see [11] for details.

4 Implementation and Performance Measurements

We start from an optimized implementation of SG^{++} for the Intel Xeon Processor as described in [11]. Two steps are needed to run this code on the Intel MIC Architecture: (1) offload the data from Fig. 3 to the coprocessor, and (2) transcribe the compute kernels from SSE to code for the MIC VPU. In contrast to NVIDIA CUDA and OpenCL, the Intel MIC Architecture uses a simplified offloading model without buffers, device contexts, and kernels that are executed via command queues.

Intel MIC Architecture. Code to be offloaded to the coprocessor is qualified by a single pragma (`#pragma offload target(mic)`) to mark a region of code (listing 1.1). Data transfers are automatically handled through the offload statement. Transfer options can be selected by adding `in` (host to coprocessor) and `out` (coprocessor to host) clauses. Listing 1.1 shows the offloading of the grid data (*Level,Index,*α), the training dataset (*Data*), and the result (Y). For arrays, `in` and `out` receive a pointer as the array's address and a length specifier. Please note that compilers without support for the Intel MIC Architecture (like GCC) safely ignore the offload pragma.

Listing 1.1. Offloading computation and data for execution on the Intel MIC Architecture coprocessor by preprocessor directives and calling the compute kernel.

```
#pragma offload target(mic) in(ptrLevel:length(dims*no_Grid)) \
    in(ptrIndex:length(dims*no_Grid)) in(ptrAlpha:length(no_Grid)) \
    in(ptrData:length(dims*no_Inst)) in(no_Grid,no_Inst,dims) \
    out(ptrY:length(no_Inst))
{
   multBT(ptrLevel,ptrIndex,ptrAlpha,ptrData,ptrY,no_Grid,no_Inst,dims);
}
```

In case of SG^{++}, the Xeon Processor implementation of `multBT` uses SSE intrinsics and relies on OpenMP for parallelization. We can semi-automatically (by searching and replacing) transcribe the kernel from SSE to MIC VPU intrinsics (e. g., substitute `_mm_mul_ps` with `_mm512_mul_ps`) and adjust loop counters to the new vector length. With these small changes, we reach the performance in Table 1 (column *simple*). By applying minor tweaks such as inserting prefetch instructions, the performance can be increased further (column *optimized*).

NVIDIA Fermi C2050. We focus on the NVIDIA Fermi Architecture as the most general-purpose GPU available due to its architecture featuring multiple cache levels. Since every instance can be evaluated independently of each other, the operators' iterative formulation constitutes a highly parallel workload. We choose OpenCL as the vehicle to implement SG^{++}, as it is an open standard for both GPUs and CPUs and it optimally fits our algorithm. Note that OpenCL is closely related to NVIDIA CUDA: kernels execute on the accelerator, and the programming paradigm resembles the notion of shader-style, data-parallel languages. Buffers and messages are used to communicate with the GPU, and the memory model distinguishes global and local sections. The handling of offloaded code and data requires some additional programming effort compared to the offload model based on pragmas.

CUDA and OpenCL do not support dynamic memory allocation. We need such capabilities to implement optimal prefetching of data. However, we solve this issue by exploiting the JIT compiler of OpenCL. When the OpenCL runtime invokes a kernel, all runtime parameters are known and the JIT compiler can tailor the kernel to optimally fit the input. Our experiments with CUDA and OpenCL have shown that the JIT-compiled OpenCL code outperforms the CUDA version by about 2x. This dramatic difference is due to too small dimensional loops (five dimensions in our case). Because of the relatively high loop overhead, these loops are very detrimental to performance. Because of the OpenCL JIT compiler, the SG^{++} implementation can generate a fully unrolled loop at runtime. To confirm this, we have tested a specialized CUDA kernel on a specific data set and manually unrolled the loops over dimensions. In this case, the CUDA performance was on par with OpenCL. In summary, OpenCL is the better choice for SG^{++}, due to its competitive performance and higher flexibility.

Uniform basis functions. Due to the shader-style code (see [14]) the implementation of $B^T\boldsymbol{\alpha}$ evaluates an instance of the dataset for each work item.

Table 1. Performance of both simple and optimized software port to Intel Knights Ferry and to the NVIDIA C2050. Performance is measured in GFLOPS using single precision floating point numbers.

Dataset	Intel Knights Ferry**		NVIDIA C2050**	
GFLOPS	simple	optimized	simple	optimized
DR5 (std. grid)	423	441	276	345
5d checkerboard (std. grid)	435	442	325	418
DR5 (mod. grid)	–	276	–	65
5d checkerboard (mod. grid)	–	297	–	70
DR5 (std. grid), dual	–	854	–	582
5d chk.brd. (std. grid), dual	–	842	–	741

NVIDIA suggests the local size (number of work items in a work group) to be a multiple of 32. Our tests have shown that a local size of 64 gives the best performance on the Fermi GPU. Although the Fermi architecture offers standard L1 and L2 caches, the performance of a simple straightforward port is clearly behind the Intel MIC Architecture performance as shown in Table 1.

Several kernel optimization techniques can be applied to improve performance. First, the local storage of a workgroup can be used to prefetch data into the caches. Second, the runtime compilation of OpenCL can be instructed to perform runtime code generation. At runtime, the compiler knows about the loop length and thus the loop over the dimensions can be completely unrolled to reduce the amount of control flow in the kernel.

Because the multiplication of B is parallelized along grid points, an implementation difficulty arises. The grid may contain an arbitrary number of grid points, but we have to map the grid to workgroups with a discrete distribution of points. There are two options to mitigate this issue: First, we could use a workgroup size of one (i. e., a workgroup is mapped to a work item). Second, we may split the operator into a GPU and a CPU part. The GPU then handles all multiples of 64 that are smaller than the number of grid points and the CPU takes on the remainder. We make use of the second approach as it exhibits better performance. However, besides an optimized GPU implementation, an optimized Xeon Processor-based implementation is also needed. The Intel MIC Architecture does not require such padding. Its cores can handle odd numbers of iterations efficiently because of their standard IA-based instruction set.

Modified non-uniform basis functions. Modified non-uniform basis functions (Fig. 2) need a four-way `if` statement for each data point to check if an extrapolation towards the boundary is needed. The `if`s must be kept in the inner-most loop since the kernel computes a non-linear function, which prohibits hoisting the `if` statements out of the loop nest. Such code structures significantly impact the performance of the algorithm in terms of GFLOPS (see Table 1).

However, modified non-linear basis functions reduce grid sizes and memory consumption. On Knights Ferry, this halves runtime, whereas the C2050 suffers from a 63% higher runtime.

Since the Intel MIC Architecture relies on a mixture of traditional threading and vectorization, a suitable vectorization for the modified linear basis functions is as follows. As the `if` branches are independent of the evaluation point, several instances can be loaded into a vector register and one grid point is broadcast into vector registers. Depending on a grid point's property in a certain dimension, the `if` condition can be computed for all data points that are currently stored in vector registers, since there is no need to evaluate the `if` statement for each data point. Hence, the GFLOPS rate only drops by about 40%.

The root-cause analysis for the NVIDIA C2050 exhibits two reasons for the increase in runtime. First, noticeably more time is spent executing on the accelerator due to the frequent evaluation of the `if` statements. The `if` statements slow down the code, as the GPU's streaming processor executes them through predicates and parts of the processor may execute no-op instructions. Second, the grid sizes are significantly smaller for the non-uniform basis functions. Since the operator B is parallelized over the number of grid points, a smaller grid leads to a smaller degree of parallelism that can no longer satisfy the high number of processing elements of the NVIDIA C2050.

Multi-device configurations. The offload model of compiler for the Intel MIC Architecture directly supports multiple coprocessors. All Intel MIC Architecture devices in the system are uniquely identified by an integer number and can be selected by their ID in the offload pragma. For streaming applications, only the length of the offloaded arrays has to be adjusted according to the number of available devices. This boils down to simple mathematical expressions involving the array length, number of devices, and device ID. OpenCL multi-device support is based on a replication of API objects such as buffers, kernels, and command queues. Instead of simple handles for arrays, a second level of handles must be introduced to keep track of arrays on different devices. This complicates the implementation as it requires additional boilerplate code.

As the grids with modified linear basis functions need more tuning and rewriting of the algorithm to fully exploit the GPU, we restrict ourselves to grids with standard basis functions when evaluating the performance of the dual-device configuration. Table 1 lists the measured results on both platforms in the last two rows. It is obvious that the additional padding needed for the GPU has negative effects on the dual-GPU version especially for the small grids in early stages of the learning process. Since the Intel MIC Architecture implementation does not need host-CPU padding, both coprocessors can unfold their full power when dealing with small grids. For all input data, the Knights Ferry coprocessor achieves a speed-up of at least 1.9x when adding a second device, whereas a second NVIDIA C2050 yields a speed-up of about 1.7x.

Performance summary. The workload covered in this paper is neither compute bound nor memory bound (it behaves similar to a band matrix multiplication

but with a more compute-intensive kernel). The code can fully exploit the 16 times bigger general-purpose L2 cache of Knights Ferry, which explains the better baseline performance of Knights Ferry over the Tesla C2050. Table 1 shows that prefetching for MIC only slightly speeds up the compute kernel, whereas adding manual local storage loads boosts performance of the C2050 kernel. For the smaller DR5 input data, the Fermi GPU is not able to utilize its full power, while the Intel MIC Architecture is less sensitive to the size of the input data.

Productivity. In total, only two workdays were spent to enable SG^{++} for the Intel MIC Architecture, since the tool chain of Intel® Composer XE, Intel® Debugger and Intel® VTune™ Amplifier XE helped root-cause and fix performance issues in a well-known workflow. Additional implementation complexity arose from the workgroup padding needed for the C2050. We used the Visual Compute Profiler to optimize the C2050 kernel. In total, five workdays were required to implement the C2050 kernel. To keep the development time for the devices comparable, all code variants have been developed by the same person who is also one of the main developers of SG^{++} and has deep insight into the mathematical structure of SG^{++}. Hence, we exclude the time needed to analyze SG^{++} and acquaint the developer with the existing host implementation. The developer had access to the documentation for Knights Ferry and the development tools. For NVIDIA, the developer had access to both the official OpenCL documents as well as best-practice guides that can be found on the Internet. On both platforms standard dense linear algebra benchmarks are clearly above 0.5 TFLOPS, which highlights the excellent performance of the implementations presented.

5 Conclusion

We demonstrated that Intel MIC Architecture devices can easily be used to bring highly parallel applications into, or even beyond, GPU performance regions. Using well-known programming models like OpenMP and vectorization, the Intel MIC Architecture minimizes the porting effort for existing high-efficiency processor implementations. Moreover, programming for the Intel MIC Architecture does not require any special tools since its support is integrated into the complete Intel tool chain ranging from compilers over math libraries to performance analysis tools. As future HPC systems will most likely be hybrid machines with fat cores and coprocessors, programming for the Intel MIC Architecture eases the burden for developers; codes developed for the CPU portion of the system can be re-used on the coprocessor without too much of a porting effort, while achieving a better level of performance than with GPU-based accelerators.

References

1. Bungartz, H.-J., Griebel, M.: Sparse Grids. Acta Numerica 13, 147–269 (2004)
2. CAPS Enterprise. Rapidly Develop GPU Accelerated Applications (2011)

3. Intel Corporation. Pentium® Processor 75/90/100/120/133/150/166/200, Order Number 241997-010 (1997)
4. Intel Corporation. Intel® Xeon® Processor X5680 (2010), http://ark.intel.com (last accessed August 18, 2011)
5. Intel Corporation. Intel® Array Building Blocks (2011), http://software.intel.com/en-us/articles/intel-array-building-blocks/ (accessed June 15, 2011)
6. Intel Corporation. Intel® Cilk™ Plus Language Specification, Document Number 324396-001US (2011)
7. Intel Corporation. Introducing Intel® Many Integrated Core Architecture (2011), http://www.intel.com/technology/architecture-silicon/mic/index.htm (accessed June 15, 2011)
8. Lee, A., et al.: On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods. Journal of Computational and Graphical Statistics 19(4), 769–789 (2010)
9. Seiler, L., et al.: Larrabee: a Many-core x86 Architecture for Visual Computing. ACM Trans. Graph. 27(3), 18:1–18:15 (2008)
10. Khronos OpenCL Working Group. The OpenCL Specification, Version 1.1 (2010)
11. Heinecke, A., Pflüger, D.: Multi- and many-core data mining with adaptive sparse grids. In: Proc. of the 2011 ACM Intl. Conf. on Computing Frontiers (2011)
12. NVIDIA. Next Generation CUDA™ Compute Architecture: Fermi™ (2010)
13. NVIDIA. NVIDIA® CUDA™ C Programming Guide (2011)
14. NVIDIA. OpenCL™ Best Practices Guide (2011)
15. OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 3.0 (2008)
16. Pflüger, D.: Spatially Adaptive Sparse Grids for High-Dimensional Problems. Dissertation, Institut für Informatik, TUM, München (2010)
17. Reinders, J.: Intel Threading Building Blocks. O'Reilly, Sebastopol (2007)
18. Skaugen, K.: Petascale to Exascale. Keynote speech at the Intl. Supercomputing Conf. 2010 (2010)
19. The Portland Group. PGI Accelerator Compilers (2011), http://www.pgroup.com/resources/accel.htm (accessed June 15, 2011)
20. Volkov, V., Demmel, J.W.: Benchmarking GPUs to Tune Dense Linear Algebra. In: Proc. of the 2008 ACM/IEEE Conf. on Supercomputing, pp. 31:1–31:11 (2008)
21. Yelick, K.: Exascale Computing: More and Moore? 2011. Keynote speech at the 2011 ACM Intl. Conf. on Computing Frontiers (2011)

** 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 Shady Cove Platform with 2S Intel Xeon processor X5680 [4] (24GB DDR3 with 1333 MHz, SLES11.1) and single Intel 5520 IOH, Intel Knights Ferry with D0 ED silicon (GDDR5 with 3.6 GT/sec, driver v1.6.501, flash image/micro OS v1.0.0.1140/1.0.0.1140-EXT-HPC, Intel Composer XE for MIC v048), and NVIDIA C2050 (GDDR5 with 3.0 GT/sec, driver v270.41.19, CUDA 4.0).