ARL-TR-8405 ● JULY 2018

US Army Research Laboratory

Performance-Portable Benchmarking Methods for Investigating Heterogeneous Computing Platforms

by Jamie K Infantolino, James A Ross, Song J Park,
Dale R Shires, Thomas M Kendall, and David A Richie

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8405 ● JULY 2018

US Army Research Laboratory

Performance-Portable Benchmarking Methods for Investigating Heterogeneous Computing Platforms

by Jamie K Infantolino, James A Ross, Song J Park, Dale R Shires, and Thomas M Kendall
Computational and Information Sciences Directorate, ARL

David A Richie
Brown Deer Technology, Forest Hill, MD

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
July 2018	Technical Report	November 2014–August 2015

4. TITLE AND SUBTITLE
Performance-Portable Benchmarking Methods for Investigating Heterogeneous Computing Platforms

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Jamie K Infantolino, James A Ross, Song J Park, Dale R Shires, Thomas M Kendall, and David A Richie

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-CIH-S
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8405

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Comparative benchmarking across heterogeneous computing platforms has become increasingly important for the evaluation of each platform's relative merit for high-performance-computing applications. The issues involved with performing meaningful benchmarks have further complicated the difficult task of constructing benchmarking codes that employ sound methodologies to accurately predict performance. One of the most popular benchmarks for heterogeneous platforms is the Scalable Heterogeneous Computing (SHOC) benchmark suite. We examined the benchmarking methods employed in the SHOC code and developed a generative programming benchmark code that is more predictive and representative of the true capabilities of a given platform. In this work, we developed innovative benchmarking methods capable of autotuning with parameterized kernels, dynamic sampling, and scaling analysis, while maintaining a single portable code base for all platforms. The end result is a benchmark code that employs autotuning to run optimal kernels for each platform—therefore making the performance results more realistic to optimal performance and making comparisons among platforms more accurate. Benchmark results are presented for NVIDIA Kepler K20 graphics processing units, Intel Xeon Phi accelerators, and Intel Xeon central processing units.

15. SUBJECT TERMS
performance portability, heterogeneous computing, autotuning benchmark, high-performance computing, software optimization

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Jamie K Infantolino
Unclassified	Unclassified	Unclassified	UU	28	19b. TELEPHONE NUMBER (Include area code) 410-278-0172

Contents

List of Figures

List of Tables

Acknowledgments

The authors wish to acknowledge the High-Performance Computing Modernization Program GS04T09DBC0017 for contributing support of this work. This work was supported in part by a grant of computer time from the Department of Defense (DOD) High-Performance Computing Modernization Program at the US Army Research Laboratory's DOD Supercomputing Resource Center.

1. Introduction

Heterogeneous platforms are pervasive in modern systems ranging from mobile devices to supercomputers. Since Los Alamos National Laboratory and IBM's Roadrunner reached the milestone of 1 petaflop,[1] hybrid designs with accelerators continue to make up the world's top supercomputers. The motivation behind this work is based on the observation that current and future trends reside in heterogeneous computing platforms.[2–4] With the increasing presence of heterogeneous systems, there is a need for a comparative benchmarking method that supports a diverse set of computational devices. A benchmarking method can be considered the technique followed to ensure the fairest performance is measured across a set of application programs and a set of architectures. Utilizing benchmarking methods provides a good indicator of performance and the performance results provide a common set of metrics to compare performance across different architectures.

Supporting comparative benchmarking of heterogeneous platforms requires generation of a performance-portable heterogeneous code for assessment. A portable code has the ability to target various architectures such as central processing units (CPUs), graphics processing units (GPUs), Many Integrated Core Architectures (MICs), field-programmable gate arrays (FPGAs), and Advanced Reduced instruction set computing Machines (ARMs). In addition, an effective heterogeneous code will run optimally on different computing devices without users involvement in architecture specific optimizations. Vendor-specific and proprietary application programming interfaces (APIs) like CUDA fail to meet the requirement of wide-range architecture portability. In contrast, OpenCL is an industry standard for heterogeneous parallel programming with platform portability. However, performance portability is not guaranteed in the OpenCL framework. Thus, an automated approach for performance portability is essential for a meaningful heterogeneous benchmarking methods.

Normally, a benchmark suite consists of a common set of application programs that are executed to compare performance across architectures or predict performance on a particular architecture. A benchmark suite consists not only of the application programs, usually called kernels, but it also includes the benchmarking methodology used. Benchmarking methodology refers to the techniques used to ensure a valid comparison across architectures and an accurate prediction on all architectures. Some examples of the techniques that could be used include autotuning and dynamic sampling. The complexity of modern-day platforms poses a challenge for developing a portable benchmarking methodology. Modern computing systems, being a mixture of multicore and many-core processors, introduce heterogeneous

computing with processor type diversity and unique parallelism for each architecture. Hence, an accurate approach for comparing benchmarking-application programs' performance across heterogeneous systems remains an open research question. When comparing benchmarking performance across the different architectures within a heterogeneous system, it is vital to ensure one is comparing optimal performance on one architecture with optimal performance on another architecture, which is a very complex problem to solve due to the drastic differences among architectures. One needs to ensure the same implementation of the same kernel is executed with the most optimal set of kernel parameters for the given architecture to make fair comparisons.

Moreover, a challenge and objective for a heterogeneous benchmark suite is to establish a correlation between performance results and real-world applications on modern computing platforms. Similarly, capabilities of benchmarking methods extend to developing a mechanism for predicting execution time of different classes of applications. Timely and accurate application assessments on emerging architectures provide highly valuable evaluations for the high-performance computing (HPC) community.

The primary benefit of our approach is performance portability of the presented benchmark across heterogeneous computing platforms where we can accurately represent the computational capabilities of emerging hybrid systems. We present a novel approach that provides a meaningful and sound benchmarking method for heterogeneous platforms possessing performance portability, dependable timing schemes, and execution time convergence. A kernel parameterization and automatic-optimization technique was developed for high performance to support a diverse range of processors. In addition, this report introduces dynamic sampling and scaling analysis to measure benchmarks more consistently and help direct the sampling of the platform.

Our contributions are the following:

- Development of a generalized performance-portable benchmark code to better predict and represent the capabilities of heterogeneous computing platforms.
- A method for autotuning with parameterized kernels, automatic selection, and optimization that is critical for heterogeneous architecture benchmarking to accurately compare performance results among architectures.
- An approach for dynamic sampling and scaling analysis to ensure that the benchmark is sufficiently sampled for convergent results through the

performance regimes observed for heterogeneous architectures with accelerators.

- Analysis of timing methods, comparing accuracy and consistency for benchmarking accelerators.

2. Heterogeneous Benchmark Design

The goal of our work was to develop benchmarking methods that can execute at maximum performance with all available accelerators on any given platform to accurately and fairly predict and compare performance. Characteristics pertaining to open source, popularity, and community base of multiple benchmark suites were examined to determine the best candidate for portable heterogeneous benchmarking. A sampling of the suites evaluated includes OpenDwarfs,[5] Scalable Heterogeneous Computing (SHOC),[6] Rodinia,[7] Parboil,[8] LINPACK,[9] and Phoronix. OpenDwarfs is a suite consisting of 13 dwarf programs developed by Virginia Polytechnic Institute and State University. The dwarfs or application programs cover a wide variety of domains within in the science and engineering field. The authors made an effort to not optimize the programs for a specific platform. Another benchmarking suite examined was SHOC developed by the Oak Ridge National Laboratory and the University of Tennessee. This suite consists of three levels of application programs to not only benchmark the performance of the machine but the low-level hardware characteristics of the machine as well. This suite has OpenCL, CUDA, and Message Passing Interface (MPI) implementations available. Rodinia is another benchmark suite available for benchmarking heterogeneous platforms. It consists of 19 application programs developed using CUDA or OpenCL for GPU and MIC architectures. Parboil is a suite consisting of 11 application programs to study the throughput of various architectures with the intent on helping programmers get the most out of any architecture available to them. It has optimized version of the programs available in CUDA, OpenCL, and C. LINPACK is the suite used to generate the TOP500 supercomputer list twice a year. It was developed by Dongarra and Luszczek,[9] to accurately determine peak performance of a computer. It uses BLAS libraries to perform basic operations. Phoronix is another benchmark suite consisting of more than 60 test programs to be run on GPU cards. It consists of a large variety of different applications that can be used to match a specific application.

In this work, we chose kernels from SHOC as a starting point because of its portability support across a wide range of platforms. To be able to benchmark heterogeneous platforms, the code itself needs to be able to run across all available platforms, which OpenCL offers. The ability to run different types of tests (the

levels within the SHOC suite) was also a benefit to using SHOC over others. To profile the low-level hardware characteristics of a particular machine in addition to the performance of the machine leads to a more complete understanding of the different architectures.

Previously, we have benchmarked individual compute-intensive algorithms on heterogeneous architectures with accelerators for offloading calculations.[10,11] The work presented in this report attempts to develop a general benchmarking method for heterogeneous systems as opposed to a specific application study on a particular platform. Hence, research into methodologies for benchmarking the technical characteristics of accelerators has been explored including issues of cross-architecture comparisons, sensitivity to kernel implementation, latency and throughput issues, asynchronous scheduling cost mitigation, and bandwidth and workload saturation regimes. For heterogeneous architectures, relevant benchmarks must provide more than a simple number in flops.

Accordingly, in this work, we apply benchmarking methods and techniques to create a more accurate and robust benchmark code for evaluating current and future systems. The key methods and techniques include the use of kernel parameterization for automatic kernel selection and configuration and automatic scaling analysis for identifying regimes of operation to ensure sufficient and correctly targeted sampling is performed. We also examine timing methods and quantify the overestimation of performance with the common use of hardware event timers for benchmarking heterogeneous platforms. We demonstrate the use of the developed benchmark code by applying it to kernels that are equivalent to the kernels used in the SHOC Level-1 reduction and 2-D stencil benchmarks. This enables a direct comparison using the SHOC benchmarks as a reference. The reduction benchmark takes a 1-D vector of data and computes the sum over the elements of this vector. This is a trivial algorithm; however, achieving high performance across a range of architectures is nontrivial. Additionally, since each value in the vector is only read once, the algorithm is fundamentally limited by interconnect speed and bandwidth on most architectures. Reduction operations on GPUs are most beneficial when the data are already resident on the device and do not require copying from the host.

Complexities in the order of operations can result in an accumulation of error in the result. For example, with single-precision floating-point operations, adding very small numbers to very large numbers exposes the limits of the hardware.

The 2-D stencil benchmark uses a symmetric 9-point stencil operation to compute a filtered 2-D buffer result. Unlike the reduction code, each value in the 2-D input array may be used more than once and up to nine times. As a result of symmetry,

there are 11 floating-point operations per stencil-point update. The data reuse and additional mathematical operations have a tendency to improve the relative performance compared to the reduction kernel. The 2-D stencil kernel has its own set of kernel optimizations that target these algorithmic features.

3. Related Works

Many benchmarking suites are available, including OpenDwarfs, SHOC, Rodinia, Graph500, LINPACK, and so on. Benchmark suites are constantly evolving and changing. Case in point: The High-Performance Conjugate Gradient was just introduced within the last few years to better represent modern-day applications faced by computers and, as such, the results from this suite will be presented along with LINPACK results for the computers listed on the TOP500 list. However, a generic, high-quality, performance-portable heterogeneous benchmark designed to target current and emerging platforms is lacking.

For this work, we wanted to use a benchmark suite to create a heterogeneous suite that was optimized for any platform available. Previous work on this topic was done by Du et al.,[12] where the authors autotuned OpenCL code to improve performance on many architectures. Another example of work can be seen in Fang et al.,[13] where performance of CUDA kernels is compared to OpenCL kernel performance. The work done in these papers was focused mainly on GPUs and autotuning of kernels across GPUs using CUDA. The authors did use autotuning to improve performance, which shows autotuning helps performance on a GPU. However, due to the limitations with CUDA, they could not compare across architectures as this work attempts to do. Our work is to take the autotuning methodology and apply it across all platforms, including Xeon Phi and CPUs, using the same code base. This reduces the maintenance of the programs and allows for the optimal kernel selection. Other work has compared results across multiple architectures, like the ones presented here, using OpenCL.[14] The difference in that work is an autotuning methodology was not used. The same kernel with the same parameters was executed on each architecture and the results were compared.

Research has also been conducted on automatic performance tuning. Autotuning techniques have been employed to optimize a specific class of application.[15,16] The work presented by Abu-Sufah and Karim[16] uses a different approach to autotuning by using training data to classify what type of matrix the inputs are, then setting run-time parameters and determining how to process the data based on the classification of matrix. The authors used the training data to create a sequence of steps to provide the fastest results based on what type of matrix the input matrix is determined to be. The kernels we used are quite different; therefore, we could not

use the same training data. We used an empirical approach to search the parameter space instead. Li et al.[17] tuned a general matrix–matrix multiplication (GEMM) kernel utilizing CUDA on a GPU. In our work, as opposed to the work presented by Lin et al.,[17] we employ autotuning methodologies across multiple platforms utilizing OpenCL, which has a different set of concerns. We also examine a different type of kernel. Additionally, autotuning has been utilized for performance portability.[12,18,19] However, the work presented here incorporates OpenCL abstraction and evaluation of a Xeon Phi accelerator. The work presented by Phothilimthana et al.[18] concentrated on one algorithm type and does not use autotuning. The authors presented a case for improvements in performance by examining the impact of workgroup size on the results. They did not create a way for the software to automatically determine the best workgroup size and only examined workgroup size as a parameter, whereas this work creates a methodology to automatically determine the best workgroup size, in addition to other important parameters, via the kernel parameterization used. Pennycook et al.[19] showed poor performance can be seen when bad values for various parameters are chosen, which supports the case for autotuning. However, they implemented a different set of kernels than the work presented here, and they implemented a hybrid MPI/OpenCL approach whereas we concentrate just on one node using OpenCL to get the most accurate benchmarking results.

4. Method

4.1 Kernel Parameterization and Automatic Optimization

The role of code optimizations in benchmarking is widely understood to be significant and may often lead to incorrect conclusions regarding the relative performance of different architectures. The significance of optimization is more pronounced for heterogeneous computing platforms with accelerators since the technology is less mature and exhibits greater sensitivity to source-level optimizations and precise run-time parameters. The sensitivity to source-level optimization can also lead to incorrect perceptions of the relative performance of various APIs used to target accelerators. A nonportable API such as CUDA or the Intel-specific offload pragmas will naturally be used to target more precisely the associated vendor-specific architecture due to a positive impact in performance one may see using the vendor specific APIs. However, this improvement comes at the cost of portability. In contrast, OpenCL is a portable API that often leads developers to target more generic implementations that necessarily avoid target-specific optimizations. OpenCL is portable—performance is not.

As an example, the SHOC benchmark provides multiple implementations of benchmarks implemented with CUDA, OpenCL, and the Intel offload-pragma APIs. Unsurprisingly, the portable OpenCL versions are outperformed by the versions that utilize vendor-specific APIs, which implicitly target a specific architecture. This could be attributed to the superiority of vendor-specific APIs like CUDA and the Intel offload pragma, the superiority in their implementation, or a comparison between a generic and an optimized kernel implementation when comparing performance on different architectures. This situation raises a larger issue for developing portable benchmarks that provide meaningful results suitable for fair comparison among diverse architectures. For a benchmark code to utilize a portable API such as OpenCL, optimization must be integrated into the benchmarking methodology in a manner not requiring manually tuning for specific platforms, since this is prone to large discrepancies in effort and uncertain evaluations. In this work, we use source-level parameterized kernels and integrate an autotuning sweep into the benchmark code itself. The result is the identification of optimized kernels, more representative benchmark results suitable for relative comparisons, as well as a rich sampling of the platform response to parameter sweeps that reveal characteristics of the platform.

We parameterize kernels at the source level by generating a multiplicity of specific kernels using an extended C preprocessing syntax that generates individual kernels based on specific parameter values prior to passing these kernels to the platform-specific device compilers. Parameters introduced are both logical, such as whether to use local memory for caching data, and numerical, such as the number of values to update per kernel invocation or the number of iterations for manual loop unrolling. Figure 1 depicts a high-level diagram of the code-generation process. It consists of the C preprocessing syntax (Block a) that takes the individual run-time parameters (Block d) and produces kernel code (Blocks b and c). The kernel-generator code is a parameterized code that produces different output based on the runtime parameter inputs. In Fig. 1, the parameter “NUNROLL”, which corresponds to the number of times a loop is unrolled, is the runtime parameter being modified with possible values of 4, 8, and 16. Each potential value is individually supplied to the kernel-generator code. Blocks b and c show how the input impacts the output. For an input value of 4, the loop is unrolled four times (Block b), whereas for 8 it is unrolled eight times (Block c). A value of 16 (not pictured) would result in 16 lines of code within the loop to be created. This figure only depicts the case where there is only one parameter as an input. In the work presented here, the individual kernels have numerous kernel parameters that are being varied.

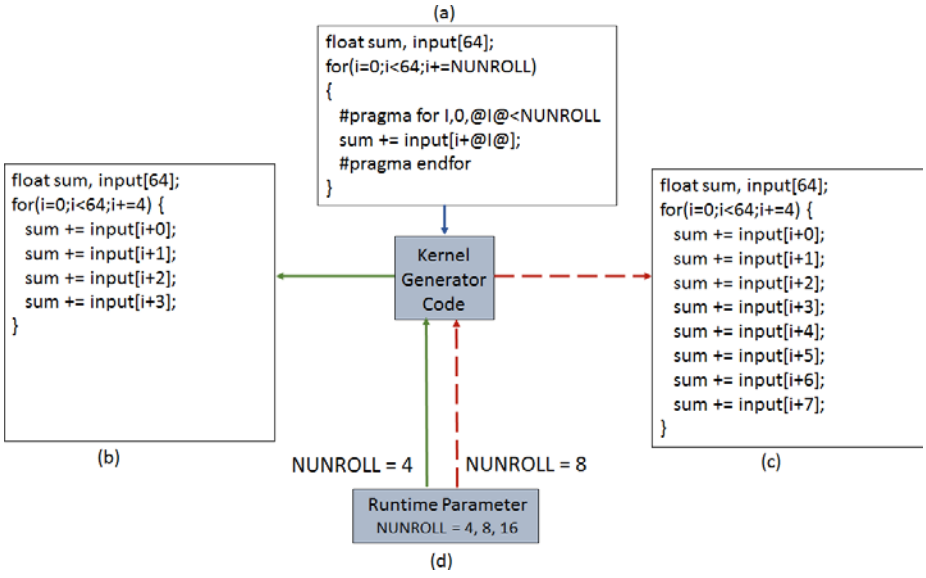

Fig. 1 Diagram of the code generator

We selected the reduction and 2-D stencil kernels that are found in the SHOC Level-1 benchmarks. Each kernel represents an algorithmic method found within the original set of dwarfs outlined by researchers at the University of California Berkely,[20] which laid the foundation for all benchmark suites. The set of dwarfs presented there represent a range of computations commonly used by the scientific community. An examination of both kernels used in the OpenCL version of SHOC very strongly indicates optimizations were intended to target a GPU.

We modify the canonical reduction kernel and 2-D stencil kernel to incorporate logical and numerical parameters that control various source-level optimizations known to impact performance. Parameters common to both kernels include the vector length of data types, the size of workgroups, the use of local memory, and the degree of explicit loop unrolling employed. For the reduction kernel, a parameter is used to select one of six index permutations used for accessing the elements in different orders, and there is also a logical parameter selecting whether to employ an incrementing pointer as opposed to array indexing. For the 2-D stencil kernel, a parameter is used to specify the number of rows to update per thread, and a logical parameter is used to select whether or not the triple loop should be fused into a single loop. Using typical values appropriate for these parameters can lead to several hundred unique kernels sampled by the benchmark code.

4.2 Dynamic Sampling and Scaling Analysis

Kernels operating on small workloads can be greatly influenced by the execution overhead, introducing greater variability in timing measurements. In contrast,

kernels operating on larger workloads can be expected to yield more stable timing since this overhead normally entails a fixed cost that becomes negligible compared with the overall compute time. This often leads benchmarks to be driven into the latter regime to assess peak performance of the platform. However, the underlying premise of benchmarking is to assess the relative merit of a given platform for use in real applications. The actual size of the workloads encountered with real applications is defined by the problem being solved and not peak capability of a hardware platform. Therefore, it is important to benchmark a platform across a full range of workloads spanning all regimes of operation, including where the hardware may not be operating at full capability. As an example, a modern high-end GPU will require many thousands of concurrent threads to saturate the compute capability of the device, yet a given problem may not admit such a high degree of parallelism.

When benchmarking a platform across the full range of such scenarios, a different number of samples may be needed to compensate for differing levels of variability. For relatively large workloads or computationally intensive kernels, timing 10 iterations may reveal significant stability and a simple average will suffice. In regimes of greater variability, significantly more samples may be needed to converge to a meaningful average. In order to compensate for this when evaluating platforms without bias, a dynamic timing scheme was implemented in which timing results are gathered while running averages and standard deviations are calculated; the number of results accumulated depend on achieving a prescribed level of convergence.

In theory, we expect that a proportionate increase in the amount of work, be it data transfer or computation, should exhibit a proportionate increase in execution time. In practice, the complexity of the overall hardware and software platform will exhibit many regimes and crossover points where such scaling is not observed. As an example, when a GPU is starved for threads, increasing the workload can exhibit a superlinear behavior in performance, showing an increase in execution time that is less than proportionate. When linear scaling regimes can be identified, they may be used to extract valuable information including effective measures for quantities that are otherwise unmeasurable.

We employ such scaling analyses dynamically within the benchmark code itself to allow for a more dynamic operation and make more efficient use of the sampling that is carried out. As an example, kernels are timed within a short inner loop to test the effects of running one kernel in quick succession after another. The number of iterations within this short loop is increased until linear scaling behavior is observed; in general, such behavior is only observed after at least three iterations and sometimes requires more.

4.3 Timing Methods

The proper methodology for benchmarking parallel architectures is a long-standing topic of discussion. In the context of heterogeneous platforms with accelerators the topic takes on an even greater complexity. The method of timing becomes critical for obtaining meaningful results to a greater extent on heterogeneous platforms than on monolithic platforms.

Many accelerators provide hardware event timers that may be accessed through a common API such as OpenCL. Hardware event timers can provide valuable information when comparing two algorithms on identical hardware platforms with identical software configurations. However, the use of event timers for comparative benchmarks between diverse architectures should be avoided since it provides inconsistent results for comparison and will generally overestimate performance. Wall-clock time provides the only true and consistent metric for benchmarks across heterogeneous architectures.

Nevertheless, we employ both event timers and wall-clock timing in order to compare results directly with the SHOC benchmark suite, which employs event timers for some benchmarks. The method of timing will by noted in the results presented. We also provide an analysis showing that the use of event timers overestimates performance.

5. Implementation

We have implemented a benchmark code based on the previously described methodology as a C++ framework with the following design objectives. The benchmark code completely abstracts the actual kernels, run-time parameters, and workloads using object-oriented design patterns. The software is designed with this abstraction to allow for consistency in the benchmarking methods employed, which should not depend on these specific elements. The benchmark is designed to gather an arbitrary number of samples, where each sample is defined by the specification of a specific timing method, kernel, workload, and run-time parameter set. For each sample, any number of timing measurements will be made, with the exact number determined dynamically and automatically to achieve a converged measurement.

The benchmark is run in two stages to avoid expending significant time obtaining precise performance estimates for low-performing samples. If a given kernel-parameter set yields a kernel that performs very badly on a given architecture, it makes little practical sense to expend a significant effort measuring that poor performance with precision. The first stage performs a sweep over all samples without employing the dynamic sampling to obtain converged timing estimates.

Samples exhibiting a performance significantly below the best-performing sample are then filtered out and a second stage is used to measure the performance of the remaining samples with converged precision. As the samples are collected in the second stage, the benchmark code performs a scaling analysis across sweeps over workloads, method-iteration parameters, and run-time parameters, to allow for analysis-based decisions about the range of samples to be tested.

All data collected down to the resolution of individual timing measurements are recorded in an Extensible Markup Language (XML) file that may be used for both postprocessing analysis and a checkpoint and restart capability. The restart capability is important and nontrivial since the capabilities of the code allow for a very broad exploration of the platform's performance, including dynamic and automatic decisions regarding the samples that are tested and the number of timing measurements gathered per sample.

6. Results

6.1 Hardware and Software Platform

Benchmarks were run on a 64-node heterogeneous cluster consisting of 48 IBM dx360M4 nodes, each with one Intel Phi 5110P and 16 IBM dx360M4 nodes each with 1 NVIDIA Kepler K20M GPU. Each node contained dual Intel Xeon E5-2670 (Sandy Bridge) CPUs, 64 GB of memory, and a Mellanox FDR-10 InfiniBand host channel adaptor. All benchmarks reported here used a single accelerator device. The goal of this work is to benchmark the different processor architectures, which is best done using a single processor since multinode benchmarks have no significant impact on the analysis of a processor architecture. Instead, they measure the performance of the larger platform (which is not the target of this investigation). The benchmarks were run with exclusive access to the host node and accelerator card using an interactive scheduler session.

All benchmarks were executed with error-correcting code (ECC) memory enabled. This provides consistency in detection and correction of bit errors for comparison between the Phi and K20 results. Enabling ECC on the K20s reduces the available memory capacity by approximately 12.5% and has a corresponding decrease in bandwidth.

The software stack on the system included the Intel Compiler XE, NVIDIA CUDA-5 toolkit, the COPRTHR software development kit (SDK) version 1.6, and the benchmark code developed in this work. The SHOC benchmark code was used for baseline comparisons and included the OpenCL and CUDA versions and the MIC optimized version using the Intel-proprietary API with offload pragmas.

6.2 Improved Benchmark Performance

Benchmark results were gathered for the reduction and 2-D-stencil kernels for the K20 GPU, Phi accelerator, and Intel CPU using our benchmark code that employs an autotuning sweep over parameterized kernels to identify the optimized kernel for the target architecture. Equivalent benchmarks were gathered using all available SHOC benchmark variants as a baseline reference—results of which are not presented here to avoid confusion.

Workloads were selected to reflect a reasonable range of problem sizes. For the reduction kernel, workloads of 131,072 (small), 8,388,608 (medium), and 268,435,456 (large) elements—corresponding to 256 kB, 32 MB, and 1024 MB of memory, respectively—were used, reflecting a range of computational work spanning over 3 orders of magnitude. For the 2-D stencil kernel, workloads with a side length of 256 (small), 2048 (medium), and 16384 (large)—corresponding to 256 kB, 16 MB, and 1024 MB of memory, respectively—were used, again spanning more than three orders of magnitude in computational work performed. (Note: Workloads are not adjusted for the target architecture since the size of a computational problem is dictated by the problem to be solved and not the hardware employed.)

The results for the reduction and 2-D-stencil kernels are shown in Fig. 2. The GPU outperforms the Phi and the CPU in all scenarios even with autotuning. These results do not indicate autotuning did not work on the Phi and CPU. It is more important to compare the unoptimized results with the autotuning results for each workload and for each kernel on each architecture. This shows the real value in autotuning. The results now allow us to make a fair comparison for optimally performing kernels across each architecture. These results mean these particular kernels performed better on the GPU, which is expected for these particular kernels. This conclusion can be drawn fairly because we know the optimal kernels were run on each architecture.

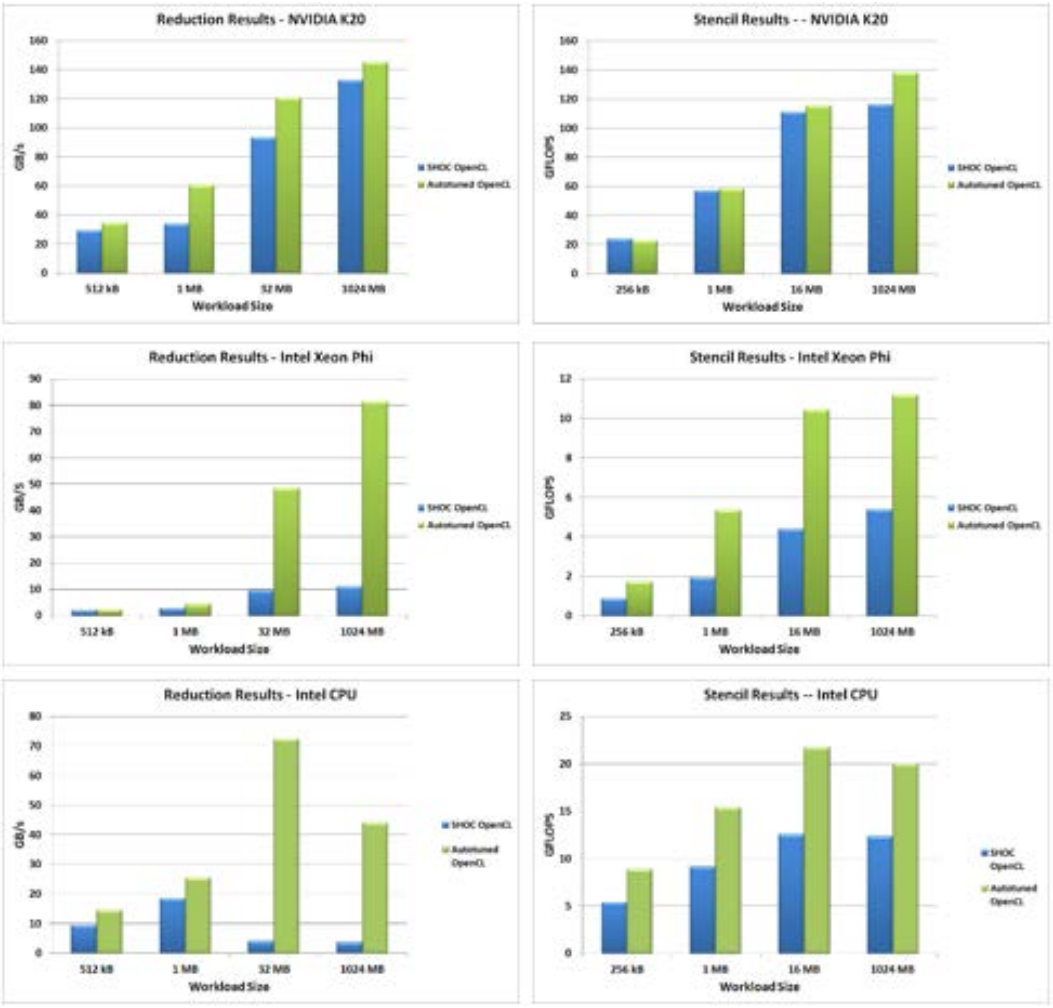

Fig. 2 Results for reduction kernels (left column) and the stencil kernels (right column) using different benchmark implementations in a range of workloads on Intel CPU, Intel MIC, and NVIDIA K20 platforms; y-axis scales differ between reduction results and stencil results

These results make clear the importance of kernel optimization in benchmarks designed for portability and used in comparative benchmarking studies.

Using fixed kernels that may have been optimized to some extent for a given architecture, such as those in the original SHOC benchmark, does not effectively measure the performance across heterogeneous architectures.

The GPU results for both kernels for the original SHOC kernels are much closer to the optimal kernel found from the autotuning methodology. This supports the claim the kernels were originally written for GPUs. The performance on the Xeon Phi is very poor with these kernels especially when viewed beside the results from the GPUs—an unfair comparison when trying to determine the difference in performance on the different architectures. Examining the autotuning results is a much fairer comparison toward understanding how the individual kernels will

perform on each architecture when using the optimal set of parameters. The kernels run and reported on both architectures were the optimal kernels for each individual architecture.

6.3 Comparison of Timing Methods

As noted previously, two timing methods are employed for benchmarking heterogeneous platforms. As an example, in the case of the SHOC benchmark, the use of event timers and wall-clock timing are used depending upon the specific kernel within the Level-1 benchmarks. Benchmark results using both event timers and wall-clock timing of kernel execution were used to estimate the long run time of a benchmark calculation allowed to run 1 h. The error in the estimation is shown in Table 1. Not only do the event timers overestimate performance, they also are inconsistent. Wall-clock timing provides a more accurate and realistic estimate of performance benchmarking accelerators.

Table 1 Long run-time prediction error using performance benchmarks based on event timers vs. wall-clock timing

Architecture/kernel	Event timer	Wall-clock timer
K20/Reduction	4.6%	<0.1%
K20/Stencil	3.7%	<0.1%
Phi/Reduction	11.8%	1.0%
Phi/Stencil	7.0%	1.8%

6.4 System-Wide Variability in Performance

The variance in the wall-clock timing among multiple nodes was measured as it is important for cluster-level code scalability with respect to application-performance jitter. It is not clear whether smaller workloads with shorter execution times would have greater variance than larger workloads. Although the trends across multiple nodes are apparent in Fig. 3, the performance results with respect to the workload size are unintuitive for the reduction benchmark on both the K20 GPU and Phi accelerator. No general conclusions can be drawn from these figures related to problem size and expected variance in timings for arbitrary algorithms.

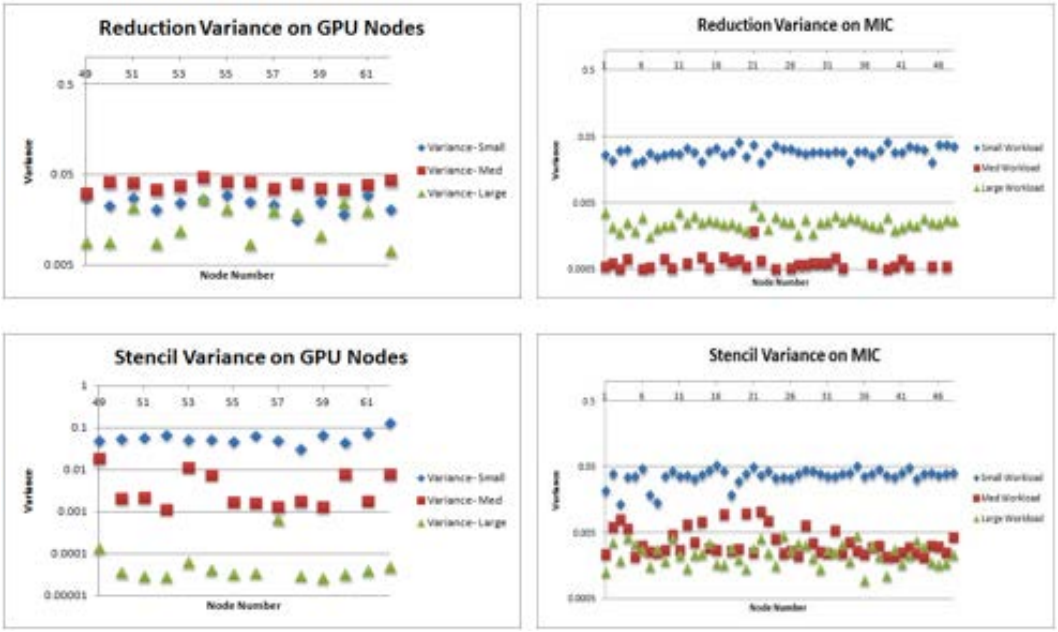

Fig. 3 Variance of performance results for reduction and stencil codes across multiple workload sizes and for both the K20 and Phi

7. Conclusions and Future Work

A performance-focused, portable heterogeneous benchmarking methodology was developed to assess emerging accelerator architectures. For performance portability, a technique utilizing parameterized kernel sweeps was developed and applied to NVIDIA GPU and the Intel Xeon Phi accelerators. Results show the performances for accelerators are highly sensitive to input and runtime parameters, requiring the use of autotuning methods for performance portability. In addition, these results show that to accurately benchmark different architectures for comparison of performance results, autotuning is a necessity. It provides a mechanism for direct performance comparisons because it ensures the most optimal performance is seen for each architecture; therefore, optimal performance is being compared against optimal performance. Moreover, this work evaluates timing methods and shows that the use of true wall-clock timing is more accurate and consistent than the use of event timers for measuring benchmark performance. This will be beneficial to HPC software and algorithm designers seeking overall faster solutions to problems that can benefit from heterogeneous processing technology. This benchmarking approach can be leveraged to predict computational throughput and execution time of applications with stencil or reduction operations at their core.

The next step in this work is to add autotuning capabilities to more of the SHOC kernels and incorporate them into our suite to discover the set of optimal parameters across architectures. These will be added to the suite so they will benefit from the dynamic sampling methods also utilized in this work. Additional work will explore other autotuning methodologies to improve overall run time of the suite. The exhaustive search guarantees the best result is found, but the run times, especially in the larger workloads, can make it infeasible in some situations. Future work will explore the use of more advanced parameter-space exploration algorithms that improve the trade-off between run time and optimal parameter configuration.

8. References

1. Barker KJ, Davis K, Hoisie A, Kerbyson DJ, Lang M, Pakin S, Sancho JC. Entering the petaflop era: the architecture and performance of roadrunner. In Proceedings of the 2008 ACM/IEEE Conference on Super-Computing; IEEE Press; 2008; Austin, TX. p. 1.

2. Advanced Micro Devices, Inc. The industry-changing impact of accelerated computing. Austin (TX): Advanced Micro Devices, Inc.; 2018 [accessed 2014]. http://sites.amd.com/jp/Documents/AMD_ fusion_Whitepaper.

3. Chung ES, Milder PA, Hoe JC, Mai K. Single-chip heterogeneous computing: does the future include custom logic, FPGAs, and GPGPUs? In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture; IEEE Computer Society; 2010; Atlanta, GA. p. 225–236.

4. Hameed R, Qadeer W, Wachs M, Azizi O, Solomatnikov A, Lee BC, Richardson S, Kozyrakis C, Horowitz M. Understanding sources of inefficiency in general-purpose chips. In: ACM SIGARCH Computer Architecture News. Vol. 38, ACM; 2010. p. 37–47.

5. Feng W-c, Lin H, Scogland T, Zhang J. OpenCL and the 13 dwarfs: a work in progress. In: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering; ACM; 2012; Boston, MA. p. 291–294.

6. Danalis A, Marin G, McCurdy C, Meredith JS, Roth PC, Spafford K, Tipparaju V, Vetter JS. The scalable heterogeneous computing (SHOC) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units; ACM; 2010; Pittsburgh, PA. p. 63–74.

7. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee S-H, Skadron K. Rodinia: A benchmark suite for heterogeneous computing. In: Workload Characterization, IISWC 2009. IEEE International Symposium on IEEE; 2009; Austin, TX. p. 44–54.

8. Stratton JA, Rodrigues C, Sung IJ, Obeid N, Chang LW, Anssari N, Liu GD, Hwu WW. Parboil: a revised benchmark suite for scientific and commercial throughput computing. Urbana–Champaign (IL): University of Illinois Center for Reliable and High-Performance Computing; 2012. Report No.: IMPACT-12-01.

9. Dongarra J, Luszczek P. Linpack benchmark. In: Padua D, editor. Encyclopedia of parallel computing. New York (NY): Springer; 2011. p. 1033–1036.

10. Richie DA, Ross JA, Park SJ, Shires DR. Ray-tracing-based geospatial optimization for heterogeneous architectures enhancing situational awareness. In: Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on IEEE; 2013; Sydney, NSW, Australia. p. 81–86.

11. Richie D, Ross J, Ruloff J, Park S, Pollock L, Shires D. Investigation of parallel programmability and performance of a Calxeda arm server using OpenCL. In: Euro-Par 2013: Parallel Processing Workshops; 2014; Berlin–Heidelberg (Germany): Springer. p. 865–874.

12. Du P, Weber R, Luszczek P, Tomov S, Peterson G, Dongarra J. From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming. Para Comp. 2012;38(8):391–407.

13. Fang J, Varbanescu AL, Sips H. A comprehensive performance comparison of CUDA and OpenCL. In: Parallel Processing (ICPP), 2011 International Conference on IEEE; 2011; Taipei City, Taiwan. p. 216–225.

14. McIntosh-Smith S, Boulton M, Curran D, Price J. On the performance portability of structured grid codes on many-core computer architectures. In: Kunkel JM, Ludwig T, Meuer HW, editors. Supercomputing. Proceedings of ISC 2014, 29th International Conference; 2014 June 22–26; Leipzig, Germany. Switzerland: Springer International Publishing; 2014. p. 53–75.

15. Whaley RC, Petitet A, Dongarra JJ. Automated empirical optimizations of software and the atlas project. Para Comp. 2001;27(1):3–35.

16. Abu-Sufah W, Karim AA. Auto-tuning of sparse matrix-vector multiplication on graphics processors. In: Kunkel JM, Ludwig T, Meuer HW. Supercomputing. Proceedings of ISC 2013, 28th International Supercomputing Conference; 2013 June 16–20; Leipzig, Germany. Switzerland: Springer International Publishing; 2013. p. 151–164.

17. Li Y, Dongarra J, Tomov S. A note on auto-tuning GEMM for GPUS. In: Allen G, editor. Computational science. Proceedings of ICCS 2009. Berlin, Germany: Springer-Verlag; 2009. p. 884–892.

18. Phothilimthana PM, Ansel J, Ragan-Kelley J, Amarasinghe S. Portable performance on heterogeneous architectures. ACM SIGPLAN Notices. 2013; 48(4):431–444.

19. Pennycook SJ, Hammond SD, Wright SA, Herdman J, Miller I, Jarvis SA. An investigation of the performance portability of OpenCL. J Para Distr Comp. 2013;73(11):1439–1450.

20. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, et al. The landscape of parallel computing research: a view from Berkeley. Berkeley (CA): University of California EECS Department; 2006. Report No.: UCB/EECS-2006-183.

List of Symbols, Abbreviations, and Acronyms

1-D	1-dimensional
2-D	2-dimensional
API	application programming interface
CPU	central processing unit
ECC	error-correcting code
GEMM	general matrix–matrix multiplication
GPU	graphics processing unit
HPC	high-performance computing
MIC	Many Integrated Core Architectures
MPI	Message Passing Interface
SDK	software development kit
SHOC	Scalable Heterogeneous Computing
XML	Extensible Markup Language

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIR ARL IMAL HRA RECORDS MGMT RDRL DCL TECH LIB
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
4 (PDF)	ARL RDRL CIH S J INFANTOLINO D SHIRES S PARK J ROSS