Understanding the Performance of Stencil Computations on Intel's Xeon Phi

Joshua Peraza* Ananta Tiwari† Michael Laurenzano†
Laura Carrington† William A. Ward, Jr.‡ Roy Campbell‡

*University of California, San Diego, CA, USA, *jperaza@cse.ucsd.edu*
†San Diego Supercomputer Center, La Jolla, CA, USA, {*tiwari, michaell, lcarring*}*@sdsc.edu*
‡High Performance Computing Modernization Program,
United States Department of Defense, {*william.ward, roy.campbell*}*@hpc.mil*

***Abstract*—**

Accelerators are becoming prevalent in high performance computing as a way of achieving increased computational capacity within a smaller power budget. Effectively utilizing the raw compute capacity made available by these systems, however, remains a challenge because it can require a substantial investment of programmer time to port and optimize code to effectively use novel accelerator hardware. In this paper we present a methodology for isolating and modeling the performance of common performance-critical patterns of code (so-called idioms) and other relevant behavioral characteristics from large scale HPC applications which are likely to perform favorably on Intel Xeon Phi. The benefits of the methodology are twofold: (1) it directs programmer efforts toward the regions of code most likely to benefit from porting to the Xeon Phi and (2) provides speedup estimates for porting those regions of code. We then apply the methodology to the stencil idiom, showing performance improvements of up to a factor of 4.7x on stencil-based benchmark codes.

I. Introduction

The push for increasingly larger supercomputers has spurred the development of specialized heterogeneous computing environments. Limits to power and cooling infrastructures prohibit adding more of the same commodity hardware. Instead, diversification and specialization of hardware can be used to increase power efficiency, thereby allowing for increased performance within the same power envelope. The potential for power efficiency of specialized hardware, however, can only be realized if it is matched to appropriate workloads, a feat that often demands great effort by an expert programmer. An inappropriate or inadequately-ported workload on an accelerator could result in a significant performance (and energy) penalty instead of a performance benefit. Therefore, as HPC systems grow in scale and complexity, the decisions to port applications must be made using methodologies that can accurately map software behavior and requirements to the capabilities of the hardware. Performance models can help understand and quantify performance sensitivity of applications to attributes of hardware and can provide a solid basis for developing methodologies that can guide computational scientists in porting and tuning their applications for a heterogeneous computing environment.

In this paper, we develop performance models using *idioms*, a set of frequently used compute and memory access patterns, as predictors of performance when porting code to an accelerator. We conduct a detailed study on the performance of these simple patterns on the target hardware. This study analyzes the performance of idioms under various software and hardware configurations – e.g., different working set sizes and numbers of threads. Our methodology then breaks complex HPC applications into their constituent idioms and provides hints about what sections are best candidates for porting based on what we have learned about how those idioms perform on target hardware. Our methodology is based on the premise that applications can be described by their constituent idioms and that given a sufficiently broad set of idioms, they can characterize the performance behavior of the application.

This paper uses Intel's Xeon Phi as the target accelerator and focuses on modeling the performance of stream and stencil idioms. We use two application kernels – 1. a 7-point stencil computation that solves a 3-D heat equation and 2. a 3-D 19-point Poisson solver. We also evaluate our methodology on a large scale earthquake simulation code – Anelastic Wave Propagation by Olsen, Day and Cui (AWP-ODC) – a 3-D finite difference based earthquake simulation code [3].

II. Characterizing the Xeon Phi

This section describes the experimental platform and idiom characterization studies designed to understand the performance behavior of stream and various flavors of stencil idioms.

A. Experimental Platform

We compare performance of idioms on a pair of Xeon X5680 CPUs (Sandy Bridge), which we refer to as the host, and a 52 core Intel MIC coprocessor (Xeon Phi). Each Xeon X5680 has 6 3.3 GHz cores with 2 hardware threads each. Each core has a 32 KB L1 cache, 256 KB L2 cache, and each chip has a shared 12 MB L3 cache. Xeon Phi has 52 1GHz cores with 4 hardware threads each. Each Xeon Phi core also has 32 KB of L1 cache and 512 KB of L2 cache. The L2 caches of each core are connected via a ring bus to create a 26 MB shared L3. Xeon Phi also has available 512 bit vector registers compared to 256 bits on the host.

978-1-4799-0898-1/13/$31.00 ©2013 IEEE

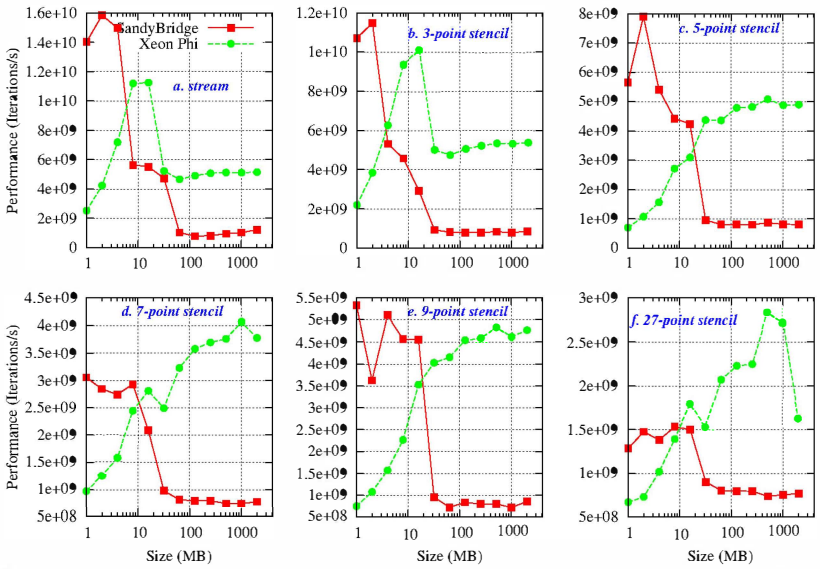

Fig. 1: Idiom performance as a function of the working set size for the host & Xeon Phi

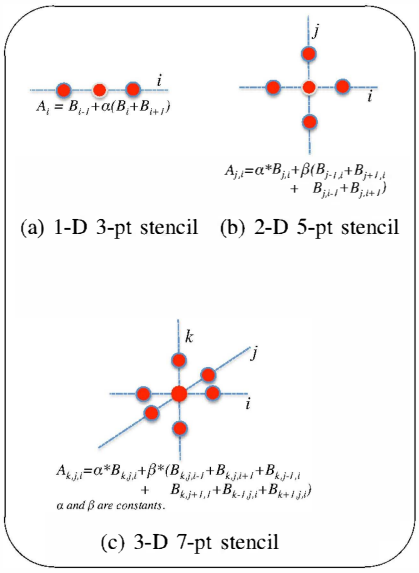

Fig. 2: Different Types of Stencils

B. *Characterizing the Idioms*

We study of two classes of idioms – streams and stencils. Streams consist of a combination of reads and writes linearly through memory. Streams may have different strides, typically corresponding to the size of a unit of data. In these experiments, we use 8 byte doubles. There are many possible configurations of stencils and we characterize most common forms here – 1-D 3-point stencils, 2-D 5-point stencils, 2-D 9-point stencils, 3-D 7-point stencils, and 3-D 27-point stencils. The experimental setup involves offloading a loop to Xeon Phi which performs the operation described by the idiom over a dataset many times. Figures 1a, 1b, 1c, 1d, 1e, and 1f show the peak performance for the best-performing thread count for each idiom at various data sizes for both Xeon Phi and the Sandy Bridge. The performance number shown in each graph indicates the number of innermost loop iterations per second. For example, consider the code:

```
for n=1 to NREPS
  for x=1 to xx, y=1 to yy, z=1 to zz
    A[x][y][z] = B[x][y][z]
```

The total number of inner-loop iterations executed would be $NREPS * xx * yy * zz$. The performance reported would be this number divided by the time to execute the benchmark.

The results for each of these idioms shows that Xeon Phi can achieve significant performance benefits at sufficiently large dataset sizes. The exact point where Xeon Phi's performance overtakes the host's performance depends on the particular idiom, but the greatest benefit is seen when the dataset falls out of the host's cache and Xeon Phi can take advantage of its large memory bandwidth.

The stream idiom's performance, shown in Figure 1a, on the host is a reflection of the memory hierarchy bandwidths. Dataset sizes above 32 MB measure main memory bandwidth. Datasets 16 MB or smaller fit in L3 (24 MB) and datasets 2 MB or smaller fit in L2 (3 MB). Sandy Bridge results for the stream benchmark (red line in Figure 1a) show the results we expect as datasets move to each level of cache. We do not provide an absolute number for Xeon Phi's peak bandwidth because achievable bandwidth depends on what cache level is being used and what idiom is accessing data. An estimate can be determined by examining the performance at the rightmost of each idiom's performance graph and multiplying by the amount of data accessed by the idiom. The stream main memory bandwidth, for example, was measured to be 82 GB/s for Xeon Phi compared to 20 GB/s for the host.

The first stencil we characterize is the 3-point stencil, which is described graphically in Figure 2a. The center of the stencil is the same as right hand side of a stream idiom and the other two points are the adjacent items in memory; so, as can be expected, it performs very similarly to the stream idiom (see Figure 1b). Like the stream idiom, Xeon Phi can outperform the host when the dataset grows into the L3 cache.

The 5- and 9-point stencils are both 2-D stencils and perform very similarly to each other. A 5-point stencil is shown in Figure 2b. A 9-point stencil is similar but with the extra accesses in each of the two dimensions. They are distinguished from 3-point stencils in that the host outperforms Xeon Phi even when the data moves into the L3 cache (see Figures 1c and 1e). Xeon Phi does not outperform the host until the working set grows into main memory.

As with the 2-D stencils, we measure two types of 3-D stencils, 7- and 27-point (7-point is shown graphically in Figure 2c). The 7-point stencil only contains points directly adjacent to the center in each direction, while the 27-point stencil contains all the points in the cube surrounding the center point. The 7- and 27-point stencils have lower performance than the 2-dimensional stencils (see Figures 1d and 1d). At the largest data point we measured, a 2 GB working set, the 27-point stencil saw a significant performance drop off on Xeon Phi. We speculate that this has to do with reuse of cached data between subsequent iterations for the 2nd and 3rd dimensions lessening when the working set grows.

III. Identifying and Porting Code to Xeon Phi

A full-size application can be daunting to port to an accelerator. The source code is usually large and may not be well understood. It is important to have tools available to guide this process. This section describes our suite of tools and elaborates how the information that these tools provide can be used along with the idiom characterizations in the previous section to locate application hot spots and estimate the performance of those hot spots when ported to Xeon Phi.

A. Analysis Toolchain

Our tool suite consists of PEBIL[6], a binary instrumenter, PIR[2], [8] a source code analyzer for identifying idioms, and a database tool for manipulating and querying trace data. PEBIL takes as input an application binary; it can disassemble the binary, analyze it, and insert instrumentation. PEBIL is used to gather static information such as the types of instructions in basic blocks and block membership to loops and functions. PEBIL can also insert code into an application to gather information at runtime. We utilize various tools built on top of PEBIL that can be used to study various relevant aspects of a given application, including a basic block counter and a cache simulator. PEBIL's basic block counter tool keeps track of how many times every block in the application is executed. This can be combined with the static information to determine application properties such as the total number and types of instructions executed within different control units. The cache simulator, which is capable of concurrently simulating many different cache structures, also keeps track of the range of addresses accessed by each instruction. This allows us to determine working set sizes and to distinguish which instructions are operating on which data structures.

PIR is a source code analyzer implemented as a plugin to gcc. It searches through source code for idioms and reports their locations. PIR is configurable with an idiom description language which allows searching for various definitions of idioms beyond those already provided. PIR builds abstract representations of idioms described by the users and compares them (via feature matching) to the intermediate representations of the source code produced by gcc.

Our database tool has at its center a sqlite database. It consumes trace data from several possible sources, including PEBIL and has interfaces for querying that data in a simple way. It allows the user to quickly generate custom reports to summarize traces far too large for direct examination.

B. Methodology

To aid a programmer in porting a code to an accelerator, our toolchain must first identify the sections of code most important to the application's performance, and then determine whether those sections might see a performance benefit after being ported to a specific accelerator. We can locate performance critical sections of an application using PEBIL's basic block counter. PEBIL produces a summary of the most frequently visited blocks and we use this list to focus the remainder of our analysis. The cache simulator instruments these most important blocks for analysis and the resulting traces are passed to our database tool. The database tool can then produce various types of reports summarizing the application's activity. A loops report will list each loop of the application, sorted by the number of dynamic instructions and reports entry counts, block counts, the loops' share of total instructions executed by the application, types of operations (e.g. floating point vs memory), and cache hit rates.

TABLE I: Heat3D and Poisson Characterizations

Benchmark	Idiom Component	Misses Per Instruction
Heat3D	Streaming write	0.0127
	7pt stencil: i-1,j,k	0.0127
	7pt stencil: i,j-1,k	0.0003
	7pt stencil: i,j,k+1	0.0003
	7pt stencil: i,j+1,k	0.0127
	7pt stencil: i+1,j,k	0.0127
Poission	Streaming read	0.00455
	Streaming write	0.00455
	3-D stencil read	0.01478

The instruction-level report lists each instruction in the application (eliminating rarely executed instructions), what loop it belongs to, its source code location, its L3 cache miss rate per dynamic instruction in its parent loop, the range of addresses accessed, and the L3 cache miss rate per instruction for all instructions in the loop. We use this report to determine what share of memory activity is attributable to each instruction in each loop and map the instructions back to source code locations and data structures.

Once instructions and their memory activity are mapped to source code locations, PIR can be used to help determine which idioms exist at those source code locations. At this stage, we have a report describing how much memory activity the application requires for each type of idiom. We then use this summary to estimate possible performance benefits using our characterizations in the previous section as a guide.

IV. Results

In this section, we utilize the tool suite we described in section III in conjunction with the idiom characterizations presented in section II to predict the performance of benchmark applications when ported to Xeon Phi. The first benchmark we consider is Heat3D – heat equation application benchmark. The performance of this benchmark is dominated by a single loop containing a 7-point stencil. Table I summarizes the memory activity for Heat3D. The rows of the table are broken down by the components of the stencil. In each row, the misses per instruction metric indicates the number of L3 cache misses for memory references in this component per dynamic instruction executed in the loop. Summing the rows indicates the total fraction of instructions that caused L3 cache misses. The streaming write covers writing to the destination array. The remaining rows of the table indicate the most significant points of the stencil based on the L3 cache miss rate. There is a row for each point in the stencil except i,j,k-1 and i,j,k. Those points very rarely miss because they are pulled into cache in advance by the i,j,k+1 point. To clarify, examine the source code for Heat3D:

```
1: A[i][j][k] =
     c0*B[i][j][k] + c1*( B[i][j][k-1] +
2:         B[i-1][j][k] +
3:         B[i][j-1][k] +
4:         B[i][j][k+1] +
5:         B[i][j+1][k] +
6:         B[i+1][j][k]);
```

We have labeled the lines with the rows of Table I they correspond to. Since this benchmark is a simple 7-point stencil,

we can use the address range information reported for Heat3D to make a performance prediction. The reports for Heat3D indicated that a 2 GB dataset was used. Using Figure 1d, which shows the performance on Xeon Phi as a function of the data set size or range of data accessed, we predict a speedup of 4.9x. The measured speedup is 4.7x (see Table III).

The next benchmark we evaluate is Poisson. Poisson is also dominated by a single loop but it contains two idioms instead of just one as in Heat3D. Poisson contains an 18-point 3-D stencil from one array and a streaming read from another. Rather than providing a full breakdown by idiom components as we did with Heat3D, we report all the points for the 3-D stencil read in a single line that sums all relevant cache misses. The characterization is shown in Table I.

This particular run of Poisson uses three 700 MB arrays: one being written to in a stream, one being read from in a stream, and one being read from in an 18-point stencil. We estimate that the 18-point stencil is approximately the same as a 27-point stencil—the only points missing from the 18-point stencil are single offsets in the first dimension from a point that does exist in the 18-point stencil, so, they have little impact on performance. Our idiom characterizations showed that 27-point stencils (see Figure 1f) with 500-1000 MB arrays had speedups of 2.1x to 3.9x. Streams with 500-1000 MB arrays had speedups of 4.2x to 5.0x (see Figure 1a). It is unclear yet how to combine the performance effects of two simultaneous idioms, but we expect that it will be some compromise between the two, 2.1x to 5.0x. The actual speedup as shown in Table III is 4.0x.

The final application we examine is AWP-ODC, a 3D Finite Difference based earthquake simulation code [3]. AWP-ODC contains two main loops where most of the computation occurs. We characterize the first loop in Table II. The characterization and performance of the second loop is similar to the first loop so we omit it. To help us understand how to better characterize the many ways stencils may be configured, we provide a detailed breakdown of which idiom components are causing memory activity. We divide the table into sections by which arrays are being accessed. Then for each instruction that contributes significantly ($> 10^{-5}$ L3 misses per instruction) to memory activity we indicate which idiom component it corresponds to as we did with Heat3D. The loop is comprised of streams and stencils, however, the stencils do not exactly match the stencils we characterized previously in Section II. For example, in array 1, there are two 1-D stencils. In the source code, they appear similar to:

```
1:  A[i][j][k-2] + A[i][j][k-1] +
     A[i][j][k]   + A[i][j][k+1]
    ...
2:  A[i-1][j][k] + A[i][j][k]   +
     A[i+1][j][k] + A[i+2][j][k]
```

The points on line 1 form a 1-D stencil in the first dimension which is approximately the same as the 3-point stencils or streams we have characterized. However, the points on line 2 form a 1-D stencil in the third dimension. Since these points are not adjacent in memory, their performance impact is different from a stencil in the 1st dimension. We hypothesize that if array A is sufficiently large in size, the points of this stencil will be dispersed and behave as 4 concurrent streams. Applying this hypothesis to the rest of the loop, each row in Table II indicates a single stream, for a total of 31 simultaneous

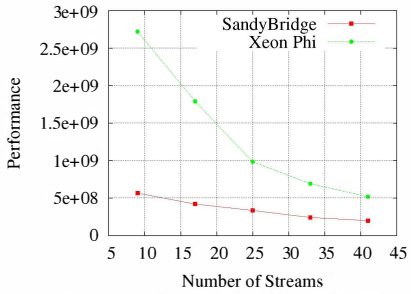

Fig. 3: Performance of multiple simultaneous streams

streams in ten 85 MB arrays. Using this approximation and our characterizations for streams (see Figure 1a), we would estimate that the loop gets a speedup of 4-6x, however, we measured the actual speedup to be only 1.13x.

For Heat3D and Poisson, it was sufficient to compare the benchmarks directly to relevant idiom characterizations because they only used one or two idiom instances in the loop and the characterizations in Section II look at idioms in isolation. Predicting the speedup for AWP-ODC is a more difficult problem because we must first understand how to convolve many idioms together.

Re-examining our characterizations, we see that the 27-point stencil is a convolution of 9 3-point stencils. The fact that the 27-point stencil has a very different characterization from a 3-point stencil further demonstrates the need to better understand how smaller idioms interact when they occur together in a loop. To characterize the effect of many concurrent streams, we conduct an experiment in which there are many streaming reads in a large array. We simulate this using a stencil with many offsets in the 3rd dimension. We use two 500 MB arrays, one for writing in a stream and one for reading in several streams and measure the performance on both the Xeon Phi and the host for various numbers of streams. Figure 3 verifies the performance degrading effect of having multiple concurrent streams. The effect is more pronounced on Xeon Phi than the host. If we estimate the speedup for AWP directly from this experiment at 31 streams, we would predict a speedup of 2.8x. This brings our estimate closer to the measured 1.13x speedup.

The discrepancy between this new prediction and actual speedup may be explained by the simplicity of this characterization. The experiment is based only on the number of streams and does not control the amount of reuse that might take place between the streams. For AWP-ODC, we saw that there is certainly some reuse of data between points of streams and stencils operating in the same arrays. For example, for array 1 in Table II, the i+2,j,k point has the highest L3 miss rate because it is reading ahead of the other points. The other points can take advantage of the warm cache if the data is not evicted between accesses and therefore, they all have a lower L3 miss rate. This effect is also present in each of the other arrays. In the future, we plan to extend the characterization to support controlling the L3 miss rate for each of the streams to simulate the amount of reuse between points.

In this paper, we have focused primarily on how memory behavior of applications relates to their performance on Xeon Phi. To improve the accuracy of our methodology, other features of Xeon Phi that can have significant impacts on

TABLE II: AWP-ODC Characterization: Loop 1

Array	Idiom Component	Misses Per 1,000,000 Instructions
1	1-D stencil in 1st dim	360
1	3rd dim stencil: i-1,j,k	380
1	3rd dim stencil: i+1,j,k	380
1	3rd dim stencil: i+2,j,k	570
2	1-D stencil in 1st dim	180
2	2nd dim stencil: i,j-2,k	200
2	2nd dim stencil: i,j-1,k	200
2	2nd dim stencil: i,j+1,k	570
3	stream	170
3	2nd dim stencil: i,j-2,k	170
3	2nd dim stencil: i,j-1,k	150
3	2nd dim stencil: i,j+1,k	450
3	3rd dim stencil: i-1,j,k	360
3	3rd dim stencil: i+1,j,k	270
3	3rd dim stencil: i+2,j,k	570
4	1-D stencil in 1st dim	530
5	2nd dim stencil: i,j-1,k	200
5	2nd dim stencil: i,j,k	200
5	2nd dim stencil: i,j+1,k	200
5	2nd dim stencil: i,j+2,k	570
6	3rd dim stencil: i-2,j,k	380
6	3rd dim stencil: i-1,j,k	380
6	3rd dim stencil: i,j,k	380
6	3rd dim stencil: i+1,j,k	570
7	stream	570
8	stream	570
9	stream	570
10	3-D stencil: i+1,j,k	570
10	3-D stencil: i+1,j-1,k	150
10	3-D stencil: i,j,k	360
10	3-D stencil: i,j-1,k	170

TABLE III: Summary of Results

Benchmark	Predicted Speedup	Measured Speedup
Heat3D	4.9	4.7
Poisson	2.1-5.0	4.0
AWP-ODC Loop	2.8	1.13

performance (e.g., 512 bit vector registers and 4-way SMT) have to be considered as well. In particular, the 512 bit vector registers have the potential to double the performance of vectorizable code; therefore, being able to detect vectorizability of code sections will be important in estimating performance on Xeon Phi. We plan to investigate how these features can be accommodated within our methodology.

V. Related Work

This work extends our earlier research on using idiom characterizations and performance models to make assertions on how well sections of code port to specialized hardware. Carrington et al. [2] used idiom characterizations to estimate performance benefits of offloading gather/scatter operations to FPGAs. Meswani et al. [7] used stream and gather/scatter idioms to predict performance on GPUs and FPGAs.

Performance models have also been extensively used to study how codes perform on accelerators. Govindaraju et al. [4] present a memory model that incorporates GPU characteristics such as smaller cache sizes and apply that model to analyze and improve the performance of memory intensive kernels. Alam et al. [1] model the multi-streaming and vector processing capabilities of the Cray X1E on the NAS Parallel Benchmark's SP kernel. Hong et al. [5] presented an analytical model for GPU performance based on parallel memory requests, number of threads and memory bandwidth.

VI. Conclusion

We presented a methodology that uses idiom characterizations and the models that represent them to estimate the performance of computational phases on the MIC architecture. Our methodology improves the prospects for successfully porting an HPC application to the Xeon Phi in two ways. First, programmer effort can be targeted toward sections (functions or loops) of the application that have the highest contribution to overall performance and that are mostly likely to benefit from porting. Second, it allows a programmer to estimate the benefit of porting a section of code using speedup up estimates derived from our models.

We focused on the stream and stencil idioms and studied and modeled the performance of these idioms on Xeon Phi. We demonstrated how the models can guide decisions on whether or not it will be valuable to port sections of code to Xeon Phi.

Acknowledgements

We acknowledge the support of this project by the DoD HPCMP's User Productivity Enhancement, Technology Transfer, and Training (PETTT) Program (Contract No:GS04T09DBC0017 through High Performance Technologies, Inc.). This material is also based upon work supported by the Air Force Office of Scientific Research under AFOSR Award No. FA9550-12-1-0476.

References

[1] S. R. Alam, N. Bhatia, and J. S. Vetter. An exploration of performance attributes for symbolic modeling of emerging processing devices. In R. Perrott, B. Chapman, J. Subhlok, R. Mello, and L. Yang, editors, *High Performance Computing and Communications*, Lecture Notes in Computer Science, pages 683–694. Springer Berlin Heidelberg, 2007.

[2] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An idiom-finding tool for increasing productivity of accelerators. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 202–212, New York, NY, USA, 2011.

[3] Y. Cui, K. Olsen, T. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, J. Levesque, S. M. Day, and P. Maechling. Scalable earthquake simulation on petascale supercomputers. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–20, 2010.

[4] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[5] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[6] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 175 –183, march 2010.

[7] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole. Modeling and predicting performance of high performance computing applications on hardware accelerators. *International Journal of High Performance Computing Applications*, 2012.

[8] C. Olschanowsky, A. Snavely, M. R. Meswani, and L. Carrington. Pir: Pmac's idiom recognizer. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 189–196, Washington, DC, USA, 2010. IEEE Computer Society.