SANDIA REPORT

SAND2015-8051
Unlimited Release
Printed Sep, 2015

Towards Accurate Application Characterization for Exascale (APEX)

S.D. Hammond

Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, 87185
sdhammo@sandia.gov

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2015-8051
Unlimited Release
Printed Sep, 2015

Towards Accurate Application Characterization for Exascale (APEX)

S.D. Hammond
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, 87185
sdhammo@sandia.gov

Sandia National Laboratories has been engaged in hardware and software codesign activities for a number of years, indeed, it might be argued that prototyping of clusters as far back as the CPLANT machines and many large capability resources including ASCI Red and RedStorm were examples of codesigned solutions. As the research supporting our codesign activities has moved closer to investigating on-node runtime behavior a nature hunger has grown for detailed analysis of both hardware and algorithm performance from the perspective of low-level operations.

The Application Characterization for Exascale (APEX) LDRD was a project concieved of addressing some of these concerns. Primarily the research was to intended to focus on generating accurate and reproducible low-level performance metrics using tools that could scale to production-class code bases. Along side this research was an advocacy and analysis role associated with evaluating tools for production use, working with leading industry vendors to develop and refine solutions required by our code teams and to directly engage with production code developers to form a context for the application analysis and a bridge to the research community within Sandia. On each of these accounts significant progress has been made, particularly, as this report will cover, in the low-level analysis of operations for important classes of algorithms. This report summarizes the development of a collection of tools under the APEX research program and leaves to other SAND and L2 milestone reports the description of codesign progress with Sandia's production users/developers.

Acknowledgment

Developing new tools and approaches to understanding performance is a particularly challenging endeavor, not least because many developers have unique perspectives on how they want to understand 'performance' in their codes. Without the many inputs from a whole range of developers and colleagues it would not have been possible to have made any progress on this task. I am particularly grateful to my architecture colleagues in 01422 including Arun Rodrigues, Scott Hemmert, Gwen Voskuilen, Dave Resnick, Jeanine Cook and Bob Benner for the many conversations. From the testbed and advanced architecture perspective inputs from Jim Laros, Sue Kelly, Jim Brandt, Ann Gentile, Victor Kuhns, Nathan Gauntt, Jason Repik, T.J. Lee, Mahesh Rajan and Dennis Dinge were immensely helpful.

The support from Jim Ang has been invaluable, particularly the encouragement to reach out to vendors and our own internal development teams for input.

Several other colleagues deserve much credit for their support - Christian Trott, Carter Edwards, Dan Sunderland, Mike Heroux, Doug Doerfler and Rob Hoekstra.

Our friends and colleagues in 1500 within Sandia have provided a great deal of direction and some great dialogue over vectorization, code structures and many issues present in a production environment. I am indebted to Mike Glass, Kendall Pierson, Nate Crane, Mike Tupek, Mark Mereweather, Travis Fisher and Micah Howard for all of their support. Working with these teams was, and continues to be, a great experience.

The error modeling discussion using the GLITCH tool came from collaboration with Rob Armstrong and Jackson Mayo of Sandia's Livermore campus. These discussions have been incredibly insightful into mechanisms that may be used to detect and correct errors in complex scientific codes.

Over the last two years I have benefited immensely from discussions with Scott Pakin at the Los Alamos Laboratory. His deep knowledge of architectures and tools to extract performance analysis helped in understanding where this project could fit in with the community and add value.

Finally, my thanks go to the LDRD program offie and support team at Sandia who make incredible research projects possible with financial and administration support. We recognize how much this contributes to making our lives easier.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For over four decades Sandia National Laboratories has been a leading laboratory for the design of large-scale engineering and physics simulation applications. These demanding calculations have and continue to place to great strain on the leading computing architectures of the day, not least because of the continued pressure for greater resolution of problems, enhanced models of physical phonomena and an ever broadening range of users.

Given the continued to pressure to deliver high performance solutions to analysts across the laboratories it will come as no surprise to many readers to know that a considerably broad range of research is undertaken to identify workflow bottlenecks and to analyze our existing use cases for opportunities to improve scientific delivery.

In recent years the term 'codesign' has been used to describe the bringing together of domain experts, application developers, users, machine architects and many others to collectively identify and analyze tradeoffs for novel solutions in optimizing the scientific delivery of the laboratories. It might be argued that even as far back as the early CPLANT clusters and many large capability resources including the NCUBE, ASCI Red [4] and most recently Red Storm [15, 16, 2], Sandia has been engaged in this activity.

A feature of many recent studies has however shifted from assessment of full-scale machine designs – which are often focused on interconnection networks – to assessment of in-node performance. This is partly driven by the broadening range of solutions available in contemporary supercomputer designs but also the growing difficulty being found by application developers in extracting performance from recent, more complex node architectures. We are finding that this continues to be particularly challenging in the area of instruction vectorization. A natural consequence of this refocusing is the need to efficiently identify and extract low-level performance information about the operations production-class applications and algorithms execute. The output has several uses including: (1) supporting algorithm and bottleneck analysis; (2) evaluation of libraries, compilers and system-software components which contribute to (or in some cases inhibit) performance and, finally, (3) as a design or feedback mechanism to hardware architects who rely on this kind of information to design the processors, accelerators and memory subsystems of the future.

The Application Characterization for Exascale (APEX) LDRD was a project conceived of addressing some of these concerns. Primarily the research was to intended to focus on generating accurate and reproducible low-level performance metrics using tools that could

scale to production-class code bases, as well as multi-threaded execution (preemptively supporting programming models that will be offered on the ASC Trinity Phase I and Phase II deployments). Along side this research was an advocacy and analysis role associated with evaluating tools for production use, working with leading industry vendors to develop and refine solutions required by our code teams and to directly engage with production code developers to form a context for the application analysis and a bridge to the research community within Sandia. On each of these counts significant progress has been made, particularly, as this report will cover, in the low-level analysis of operations for important classes of algorithms. This report summarizes the development of a collection of tools under the APEX research program and leaves to other SAND and L2 milestone reports the description of codesign progress with Sandia's production users/developers.

The reader is refered to some SAND reports, papers and invited talks [14, 11, 10, 8, 9] for further information on APEX and the results obtained using the tools described herein.

Chapter 2

APEX Toolkit

APEX is a suite of tools which allow users to perform low-level analysis of performance or other areas of investigation on applications ranging from kernels through to larger production-scale binaries. The input to APEX tools is a binary executable (either dynamically or statically linked) which allows developers to utilize standard compiler or build tool chains on our high performance computing platforms. Instrumentation for analysis is performed during runtime which allows for complete access to the executable's data spaces and all loaded libraries.

In order to focus on the development and analysis aspects of this project the decision was made early on to limit the scope of the APEX toolkit to X86-64 and Xeon Phi Knights Corner microarchitectures as these are currently deployed in production or are used in preparation for the forthcoming Trinity deployment. Along a similar thought process the tools utilize the PIN binary analysis engine [13] which provides decode, analysis, instrumentation and recompile services although it is important to note this decision is simply to focus the scope of the project and a similar approach is possible across a variety of microarchitectures and platforms. PIN utilizes the XED2 [3] instruction assembly/disassembly engine to provide a lightweight ability to dynamically operate over instructions.

The analysis of applications using the APEX toolkit is usually performed in four phases, and as stated, these all occur during execution.

1. **Instruction Analysis** in which instruction information is decoded and provided to the analysis engine.
2. **Instruction/Function Instrumentation** based on analysis at instruction level specific instructions or groups of instructions may be instrumented. Typically these instructions have small counters added to their execution to keep track of behavior.
3. **Application Execution** - the main application (with any additional instrumented instructions) is executed and book-keeping added by APEX is maintained.
4. **Output Generation** - the results of booking keeping information is output into text files for later user-driven processing/analysis.

The dynamic decoding, instrumentation and then recompilation of instrumented analysis routines into the binary results in highly efficient analysis since only areas of interest need

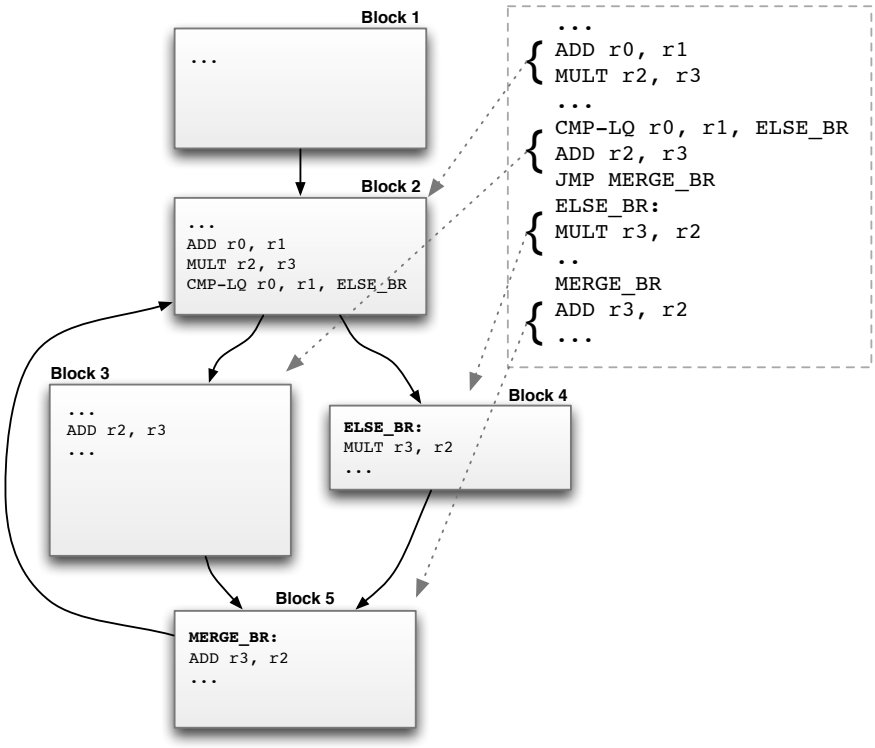

Figure 2.1. TESSERAE Instruction Blocking

to have instrumentation added. However, even small instrumentation routines added to the binary can have a significant impact on execution speed. The precise overhead is specific to the application, the amount of instrumentation added and the problem input/configuration being run by the application. In our experiences tools written using APEX are typically 3X to 40X slower than non-instrumented variants. In many cases the slowdown is near the lower end of this spectrum because APEX has been designed to be as lightweight as possible in adding instrumentation and to ensure the multi-threading support utilizes lightweight integer atomics to prevent interfearance with the running threads.

2.1 Low-Level Instruction Analysis using TESSERAE

The first developed under the APEX LDRD was the TESSERAE low-level instruction analysis engine. TESSERAE takes its name from its ability to break applications down into short instruction segments that are roughly the same as standard basic blocks found in many compilers. Basic blocks are essentially a linear flow of instructions which have no branches into them or out of them except as the end of sequence.

Figure 2.1 shows the process by which TESSERAE proceeds to break an instruction sequence into blocks. Each instruction being executed is presented by the PIN engine to the TESSERAE analysis routine. The instruction is checked for whether it is a branching operation. If the operation cannot cause a branch it is added to list which represents the "currently being decoded block." If the operation can cause a branching side effect, then the "currently being decoded block" is completed and saved. TESSERAE ensures blocks are unique by checking the address of the first instruction within each block, thus allowing only unique blocks to be saved. By recording the branch true and branch false addresses following a branch operation the control flow through blocks of the program can be recorded by TESSERAE and used for analysis.

When a block is marked as complete (*i.e.* a branching operation to terminate the sequence is found), TESSERAE re-analyzes the block operations to build a schedule of operations that are performed with focus on several metrics: (1) the number of floating point operations performed in both single and double precision; (2) the number of read/write memory operations; (3) the number of bytes read/written into the memory system (note that this is from the perspective of the processor, not the system RAM); (4) the level of vectorization achieved for double/single precision instructions and then several other metrics relating to logical operations, register clears etc. As a final addition to the block before the application is allowed to execute, TESSERAE inserts an instruction atomic increment on the first instruction in the block. This ensures a single map of the application block infrastructure can be generated and then used within a multi-threaded application without significant performance loss.

2.1.1 Floating Point Instruction Analysis

While performing floating point instruction analysis during block creation, TESSERAE is required to perform an assessment on the number of mathematical operations the instruction performs. We define *instructions* to be the number of decoded items that the processor will execute, e.g. a vector floating point addition; *operations* are defined to be the number of mathematical/logical sequences performed. In the case of a vectorized floating point add, the number of operations would be with operand width of the register. This is an important aspect of the tool because it allows users to gain feedback on the intensity of vectorization used by the processor. As a simple example of the number of operations divided by the number of floating point instructions equals, say, 4, then the block would have a high vector intensity if running on an AVX [7] or AVX-2 capable processor such as Haswell [12] (Table 2.1 shows the maximum vector width for single precision (SP) and double precision (DP) instructions). We perform this analysis on single and double precision instructions/operands independently because vector register widths are fixed bit-length meaning single-precision instructions should execute twice as many operands and therefore have a higher intensity.

The operation counts of individual instructions are performed by walking the instruction registers. Vector registers on Intel architectures are set to the full width supported by the micro-architecture but the number of operations actually performed during execution is set by bits in the instruction decoding. For instance, on a Haswell processor the physical width

Processor	SP Max/Ins	DP Max/Ins
Nehalem / Westmere	4	2
Sandy Bridge / Ivy Bridge	8	4
Haswell	8	4
Knights Corner	16	8
Knights Landing	16	8

Table 2.1. Maximum Operations Performed per Vector Instruction in Intel Processor Families

of the vector register is 256-bits but a 128-bit operation can be performed if the compiler generates a prefix informing the processor that this is intended. In order for TESSERAE to produce an accurate prediction of the operation counts the registers and prefix is assessed and the maximum found width used.

2.1.2 Masked Register Analysis

In the Haswell family of processors some instructions gain a capability of being able to mask out specific lanes during execution. This allows for instruction sequences to execute fewer than the maximum number of operations if the masks used are set to utilize only part of the register. Figures 2.2 and 2.3 show the standard and masked operations respectively. To correctly count masked operations TESSERAE creates a counter per instruction which utilize masks, initially set to zero. It then adds a small additional piece of instrumentation at these instructions which extracts the mask being used and atomically adds this to the counter associated with that instruction. This allows an intensity metric to be produced which genuinely reflects the executed sequence even when masked register are used.

We note that the use of masked registers is significantly expanded on the Xeon Phi Knights Corner card and is a significant feature of the AVX512 instruction ISA which is developed for the forthcoming Knights Landing processor used in Trinity. Therefore it is a requirement of good analysis tools that support for masking operations is included. During our prototyping analysis of TESSERAE on Knights Corner we were not able to find hardware support for independently counting the number of floating point operations executed when masks are applied. The inability of hardware to perform many of the operations developed in TESSERAE motivates the continued use of tools which can, independent of the hardware, permit such analysis to take place.

Figure 2.4 shows TESSERAE data relating to floating point intensity and vectorization levels achieved. Floating-point intensity evaluates the percentage of program instructions executed which exercise a floating-point unit to perform arithmetic (note TESSERAE excludes logical and register clears which utilize these units in X86). The most intense application

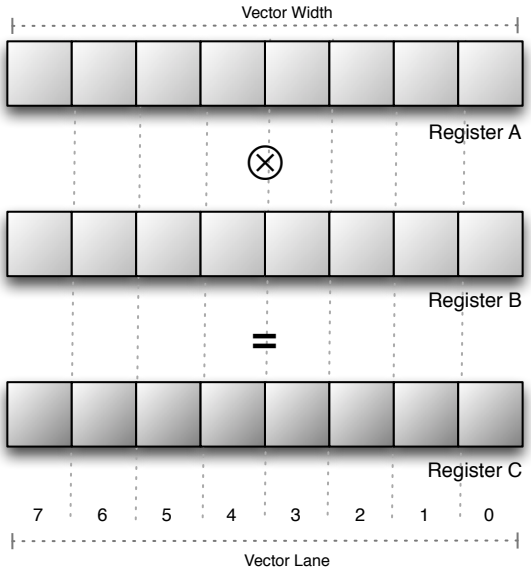

Figure 2.2. Standard (Unmasked) Vector Operations on an 8-wide Vector Unit

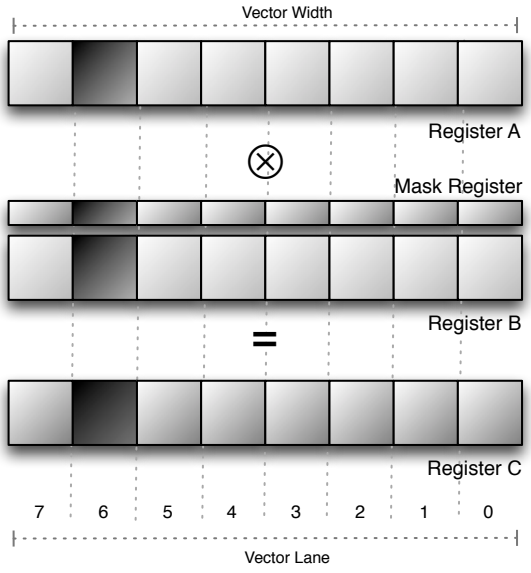

Figure 2.3. Masked Vector Operations on an 8-wide Vector Unit

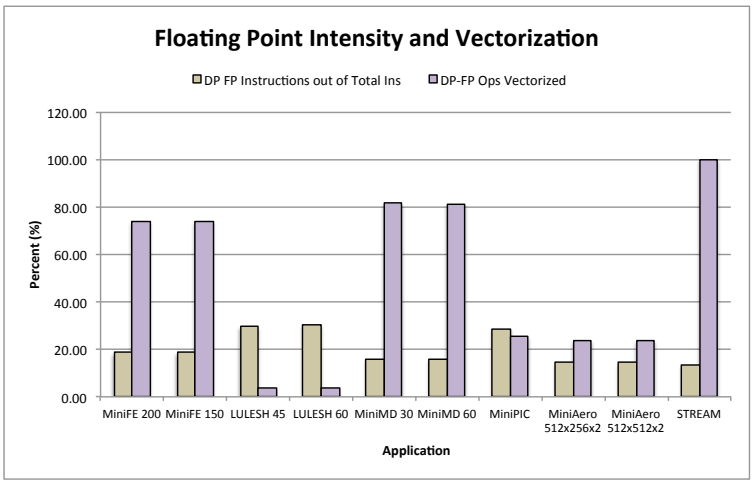

Figure 2.4. Floating Point Intensity and Vectorization of Total Instructions

executed in the LULESH hydrodynamics benchmark with approximately 30% of its instructions performing arithmetic. The typical intensity of the applications shown is between 15% and 30%. In terms of vectorization levels MiniFE and MiniMD have the highest level of operations performed by vector instructions – 74% and 82% respectively reflecting in part the aggressive level of optimization applied to the mini-applications over the past four years of Sandia's codesign program. LULESH has the lowest level of vectorization achieved (approx. 4%) which also helps to explain why a significant number of its program instructions are floating point arithmetic-based. The tradeoff here is executing more operations within each instruction (when code is well vectorized) versus executing many more scalar instructions.

2.1.3 Application Memory Access Traits

TESSERAE is able to record all memory access by walking the decoded address or index calculations within each instruction. The encoded pattern describes the base registers, any indexing required to be applied and the operand width which is to be either loaded or stored.

In Figure 2.5 we show the bytes read to bytes written ratio of the applications selected for study. MiniFE has the most intense read activity with 10 bytes read for each byte written. We attribute this to the complex generation of indices required when computing over CSR-formatted sparse matrices. MiniPIC has the lowest read intensity with only 2.4 bytes loaded for each byte written. Understanding these ratios has significant implications for the potential design of micro-architecture features such as the number of read/write ports on caches and the depth of load/store queues. In this particular study, few applications benefit from these being balanced and would benefit from a greater availability of resources for read

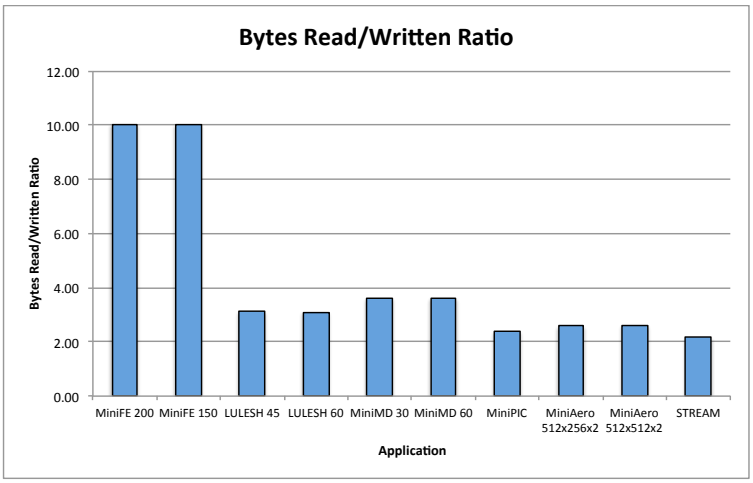

Figure 2.5. Ratio of Bytes Read to Bytes Written

operations.

The average memory operation request size per application is shown in Figure 2.6. These values are calculated by aggregating the number of bytes to be loaded and stored divided by a count of memory operations issued. MiniMD has the highest of these average request sizes with approximately 11 bytes per operation. The size of each memory operation is a useful metric in the optimization of processor bus widths which can consume large amounts of energy if over provisioned. A value of 8 is fairly common for high performance scientific codes because this reflects the width of double precision operands. Values above 8 mean that vectorized load/store operations are being used to move data into and out of the processor core registers.

2.1.4 Bytes to FLOP/s Ratios

A commonly cited metric in high performance computing over the past two decades has been that of bytes-to-FLOP/s ratio – essentially how many bytes of bandwidth or storage can the application utilize for each floating-point operation provided. In Figure 2.7 we evaluate the bandwidth-based ratio. Here we take each byte loaded/stored and evaluate it as a ratio of the number of double-precision floating point operations executed since it is typical for one of these metrics to gate application performance. MiniAero has highest ratio of bytes-to-FLOP/s at approximately 23 indicating that the code expects to require 23 bytes (greater than two double precision operands) to be loaded for each arithmetic operation. This may in part be explained by the unstructured nature of the mesh requiring additional indices to be loaded as well as copy operations to be performed in parts of the code. LULESH has a typical signature of a code with low register reuse, requiring roughly 15 bytes to be loaded

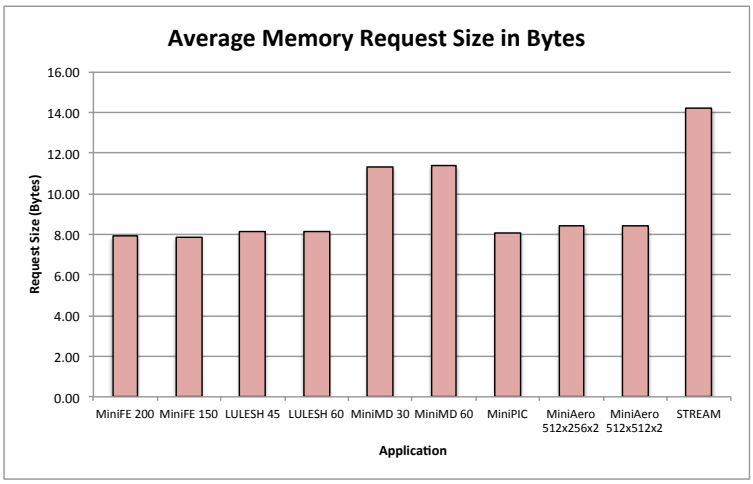

Figure 2.6. Average Memory Operation Request Size

(two double precision operands) per arithmetic operation. Finally, MiniMD and MiniFE have lower requirements which reflects the code's greater use of registers to store temporaries.

Given the significant growth of floating-point performance of processors, the results show a worrying trend. That our codes expect to be able to perform multiple bytes of loads/stores into the memory hierarchy and memory subsystem for each operation executed. Assuming a 1 TFLOP/s processor – which is a fairly low number by contemporary standards – assuming all data were serviced by the L1 caches at least 8 TB/s of cache-to-core performance would be required to deliver balanced performance for even the most efficient of these codes. At worst, nearly 23 TB/s of L1-cache-to-core bandwidth would be required assuming all operands were loaded from L1. The data from TESSERAE is a lower bound on memory-subsystem performance since it calculates the demanded bytes loaded/stored by the *processor*. Many of these requests are successfully returned from the various caches before memory. Nonetheless, these values help us to identify the absolute minimum level of bandwidth that is required by the codes evaluated. This is a key design tradeoff which has significant implications on the continued push to improve floating point performance versus genuinely balanced access to memory and compute.

2.1.5 Instruction Memory Access

A feature of many CISC instruction architectures (of which X86 is perhaps the best known) is that memory operands can often be loaded by instructions. In RISC architectures memory operands are typically loaded by specific load or store instructions and computation must occur only on registers. In most cases the number of memory operations performed is approximately equal since this is usually a property of the algorithm being compiled rather

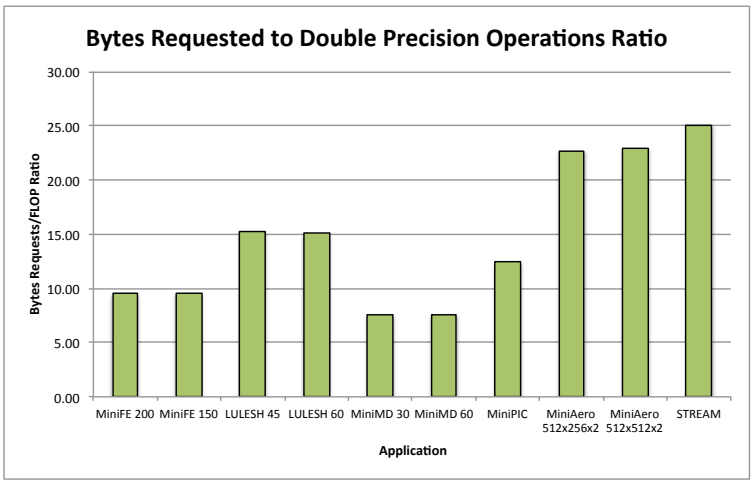

Figure 2.7. Ratio of Bytes Requested by Instructions to Floating Point Operations Performed

than the architecture being used for execution. In Figure 2.8 we show the approximate percentage of instructions in the application studied which perform memory accesses. The purpose of these results is to evaluate how many operations execute directly between registers only. The use of registers (and not memory accesses) has the potential to save energy as accesses can be localized and provides an indication of the level of optimization which should be spent in reducing access times to the processor register files and on-chip buses versus optimization for access to the memory hierarchy/memory sub-systems.

LULESH has the highest percentage of instructions which access memory directly at approximately 57%. As described earlier, the unstructured nature of the problem being solved and the low-level of vectorization make the generation of this type of code more likely. MiniMD has the lowest use of memory access in instructions pointing to a high level of register reuse within the code. This is reflected in the low level of bytes loaded/stored to floating point operations executed.

2.2 Floating Point Error Modeling using GLITCH

Late in the APEX LDRD a collaboration with Rob Armstrong and Jackson Mayo of Sandia's Livermore campus raised questions as to how likely Sandia's codes might be to errors in floating point units. For this specific case we assume operands may arrive from memory uncorrupted due to the presence of ECC checks within the memory system and then either ECC or parity checking within the caches and buses. However, when the operands are in the registers and are moved to the floating point arithmetic units a single bit flip may occur either during transit or during the calculation. This may seem a particularly specific case

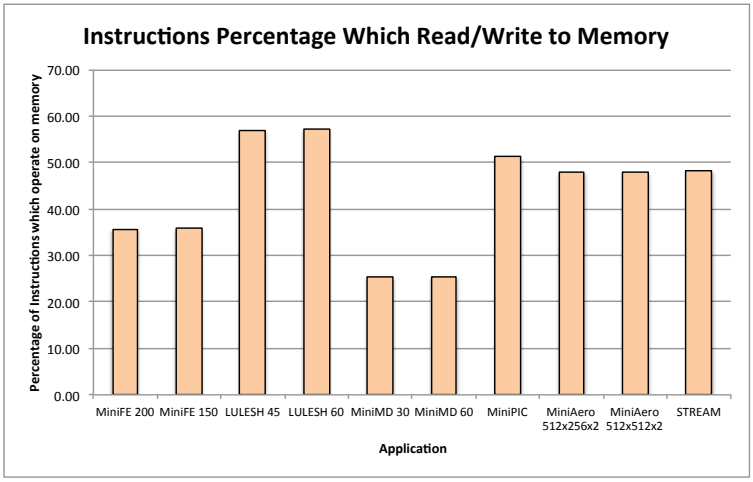

Figure 2.8. Instructions (%) which Request Memory Operations

but the floating point registers and floating point ALUs are particularly complex parts of the processor and due to area considerations may not have all of the units correctly error checked.

This works remains in progress but an initial prototype multi-threaded prototype tool called GLITCH has been written within the APEX framework. GLITCH uses the instruction instrumentation phase of APEX to detect floating-point operations (both scalar and vectorized) as well as the vector-width/operand-width logic detailed earlier. When the operands widths are known, GLITCH applies a Poisson model to selecting the floating point register and the precise bit which is to be inverted. The value (with an inverted bit) is then replaced within the floating point register being used and execution allowed to proceed. The exact parameters to the underlying Poisson process are configurable to investigate more or less error prone system behavior.

Initial testing shows many applications are able to continue execution but the errors injected propagate into the system eventually leading to numerical collapse of the systems being solved (which almost always causes the application to detect an error and halt) or to produce a significantly incorrect answer. Surprisingly few applications are able to gracefully handle an error of this kind and continue to yield a final numerical result close to an unmodified run. The results indicate high dependence on truly correct numerical results. This implies that current HPC codes are highly sensitive to errors in the handling of floating point values but in turn raises questions as to how likely silent data corruption is within existing runs. If applications are as sensitive to faults as they appear when being run in GLITCH, then even moderately frequent minor SDC would be very likely to result in application halts. Given that this is still thankfully a largely rare occurance on many machines today we might conclude that SDC is a fairly infrequently occuring event.

GLITCH is being further developed in collaboration with Rob Armstrong and Jackson Mayo for the purposes of prototyping algorithm based solutions to faults. The work started in this LDRD has given us a foundation in examining some of the issues outlined and future additions to the GLITCH tool will extend this to arrays of indices (currently an area of concern for error-prone memories) and other program structures such as the instruction pointer. We note that the use of the APEX framework and the design which includes multi-threading support is allowing for use on large applications that may include Kokkos or advanced parallelization schemes we expect to use on Trinity Phase-I and Phase-II.

2.3 Early Instruction Level Analysis of Knights Landing

The Intel Xeon Phi Knights Landing processor will be installed in the Trinity Phase-II deployment in 2017. While details of the processor remain confidential, Intel has announced that the processor will feature the AVX512 instruction set along with specific additions for HPC-oriented vector operations. The architecture specification for AVX512 is now public to enable compiler and tool writers to take advantage of early access. Along side the publication of the specifciation a tool named SDE (Software Development Environment) is also made available to permit emulation of the new instructions. SDE includes a mode to output several types of analysis including an instruction mix histogram which contains limited (when compared to APEX-TESSERAE) output on the running binary. For an initial analysis, which follows, we have used the SDE output with additional post-processing steps to generate information relating to several miniapplications running on an emulated KNL infrastructure. While these applications are run using 16 OpenMP threads to enable an analysis of multi-threading we realize more accurate emulation when the final core counts are made public will improve accuracy.

Figure 2.9 breaks down the instructions executed by each benchmark on the KNL emulation platform. In an ideal high performing application we would like to see the majority of instructions executed to be of the AVX512 ISA with most of the data movement operations using contiguous access as opposed to gather/scatter operations since these usually provide the highest level of performance. We note that the majority of instructions of HPCG (which is Sandia's main solver benchmark) do not execute floating point arithmetic and a significant proportion of these relate to logical operations associated with managing the CSR matrix format and multi-grid handling. This presents a potential optimization target as the majority of performance in the KNL processor is directed towards vectorized codes.

As stated, the highest performing option on the Knights Landing processor is likely to come from wide, contiguous access. In Figure 2.10 we show a break down of memory accesses by the number of bytes requested and the operation being performed. XSBench provides the most concerning results with a very high proportion of small read operations likely attributable to the table look operations being performed in the benchmark. The effect of many small accesses is to underutilize the L1 to core bandwidth which expects wide vector

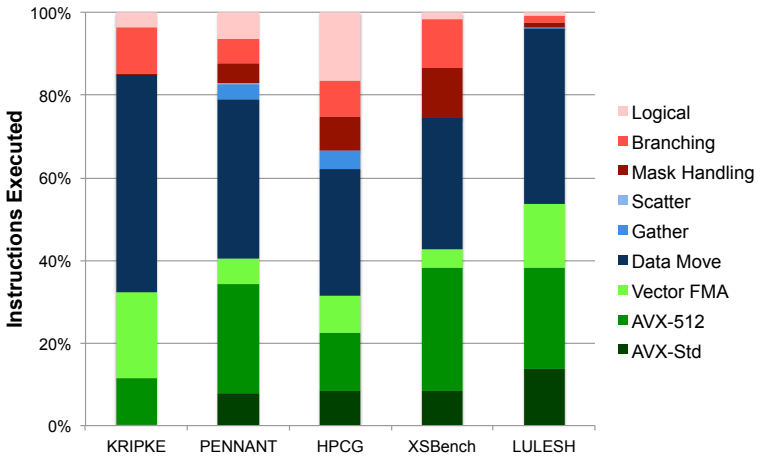

Figure 2.9. Breakdown of Instructions Executed on the Knights Landing Emulation Environment

loads/stores potentially losing performance.

A key feature of the APEX TESSERAE tool described earlier is the ability to inspect masked vector operations by arithemtic operation and type of instruction. The SDE environment reporting used for this study has much lower fidelity grouping all vector operations into single metrics without the breakdown supplied by TESSERAE. Therefore, the figures are likely be quite different to those produced by TESSERAE as all operations which utilize a vector unit are included. Nonetheless, it is currently the most accurate representation of KNL masked vector operations that we have available. Figure 2.11 shows the breakdown by benchmark of various masked vector operations. The majority of operations in all benchmarks are scalar (at minimum 60%). Very few codes except for HPCG are able to utilize the masking operations considerably. Over time we expect greater focus on vectorization and code optimization as the Trinity platform becomes available. This data serves as a starting baseline and shows we must continue to invest effort in addressing these concerns if we want to see strong performance on a heavily vector-based architecture.

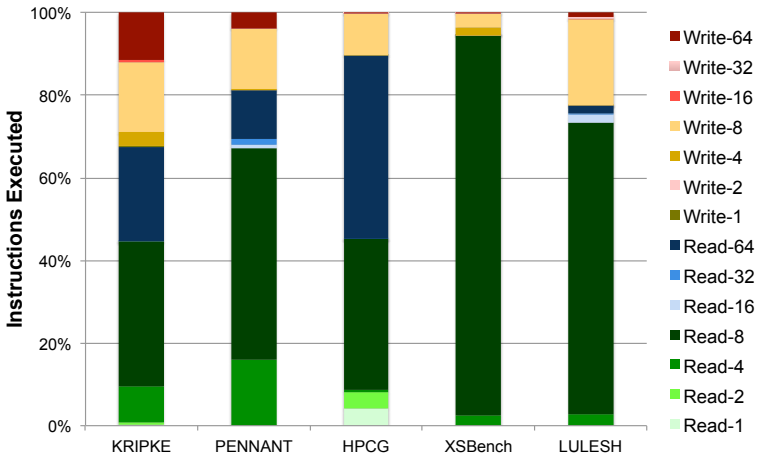

Figure 2.10. Breakdown of Memory Access Operations on the Knights Landing Emulation Environment

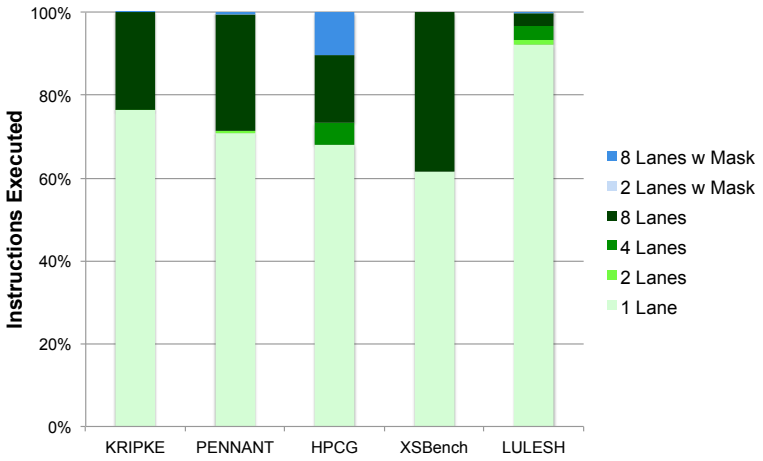

Figure 2.11. Breakdown of Vector Operations on the Knights Landing Emulation Environment

Chapter 3

Ariel Front End to the Structural Simulation Toolkit

The Structural Simulation Toolkit (SST) is one of the leading architectural simulation frameworks in the community today. The modular infrastructure allows users to design systems easily with plug-and-play swap out of components as needed. One of the processor front ends in the toolkit – Ariel – has been significantly augmented with research from the APEX LDRD with respect to the addition of instruction and operation counting mechanisms as well as enhanced memory operation processing. The basis for these additions originates in the TESSERAE block processing tool described earlier in this report except that Ariel forwards much of its block assessment into the Ariel-SST component to provide execution information. The Ariel front end (with the APEX) additions has had a significant impact on the SST user community and is now the most frequently used processor model at Sandia. Its use was pivotal to the recent multi-level memory analysis of sorting algorithms IPDPS submission [1] which was awarded best paper.

Chapter 4

KokkosP Profiling Interface

Kokkos is C++-based programming model [5, 6] which abstracts out underlying differences in hardware and data accessing to deliver strong, portable performance across diverse architectures. At its heart the programming model provides two areas of focus: (1) a set of parallel patterns that provide efficient parallel dispatch for CPUs, multi/many-core processors and GPUs; (2) a set of 'views' which provide a logical mapping of data access than can be changed at compile time to favor the device being used for execution. Results from the initial Kokkos porting activities show performance than is approximately 90% of well tuned native implementations.

4.1 Profiling Interface for Kokkos Kernels

During the development and optimization of Kokkos applications it became obvious that many existing performance tools are well optimized for specific programming models (for instance, Intel's VTune product provides strong support for OpenMP directives). When using Kokkos, which seeks to abstract out the dispatch of kernels, it was noticed that kernel execution times were attributed to components of the Kokkos runtime and not to the kernels being called. Similar problems have been shown on a variety of tools. This motivated the need for a solution independent of any specific performance analysis tool.

Figure 4.1 shows how KokkosP calls are made. The process works by having a series of function pointers defined in the Kokkos runtime – a pair for each of the parallel dispatch types presently supported (parallel-for, parallel-reduce and parallel-scan). During initialization these function pointers are resolved to profiling libraries specified by the user in their environment variables. When a parallel-for call is made during execution, the function pointer is queried. If it has been defined then the profiling tool is called otherwise execution continues. By providing runtime hooks that have low overhead checks during the parallel dispatch regions we can enable the KokkosP interface to be compiled into every Kokkos program with low overhead. This provides an "always-available" set of profiling hooks if the application developer needs to query the kernels to analyze performance.

So far we have used the KokkosP interface to produce simple profiling tools such as kernel timers, memory heap checkers as well as tools that utilize platform specific information such

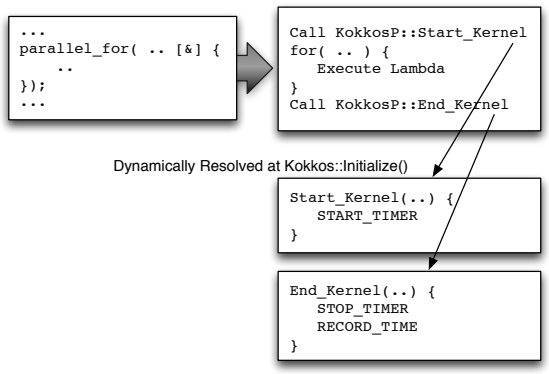

Figure 4.1. KokkosP Infrastructure for a Parallel-For Dispatch

as memory bandwidth profiling on a per-kernel basis for Haswell processors and floating-point intensity profiling facilities on Sandy Bridge processors. We note that the initialization parameters of Kokkos are passed to the profiling tools as well as device information for which the parallel dispatch will be launched. This enables us to utilize counters on the GPU when kernels are executed on this class of device as well.

Figures 4.2 and 4.3 show kernel timing analysis from the Intel Haswell and IBM POWER8 test beds respectively. We note the similarity between the profiles showing that kernel time for MiniAero is similar across the two machines (also reflected in application runtime). In this instance, we were able to run the same profiling tool across two diverse architectures which is rarely possible with many performance tools. The profiling output presented is for the kernels written using Kokkos allowing us to provide context awareness (such as dispatch type) to the output if needed.

KokkosP profiling tools remain an area of further research, particularly during FY16. We expect to be able to utilize tools for several activities in the Trilab codesign milestone and in our engagement with several code groups in 1500. The ownership for development of KokkosP will move from the prototypes and APEX connectors developed in this LDRD to the Kokkos development group for these activities.

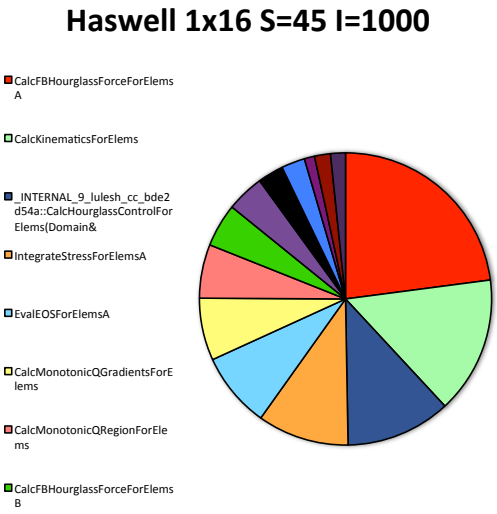

Figure 4.2. LULESH Kernel Timer Analysis using KokkosP on Intel Haswell

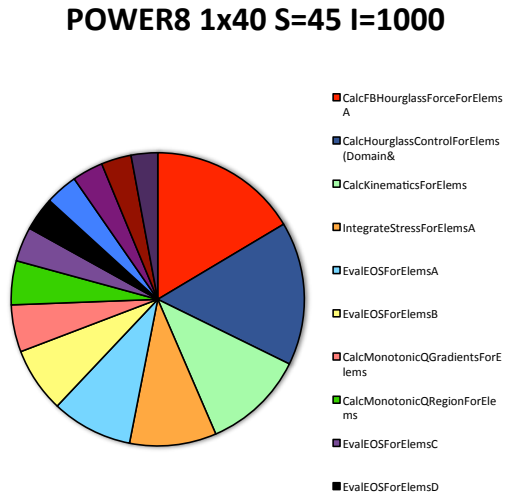

Figure 4.3. LULESH Kernel Timer Analysis using KokkosP on IBM POWER8

Chapter 5

Conclusions

This SAND report is a short description of the Application Characterization for Exascale LDRD. It describes the creation of several tools for the purposes of exploring low-level application metrics, impact of unreliable floating-point calculations and several other characterization mechanisms including emulation analysis for the future ATS-1 platform. The tools are slowly being applied to a greater range of codes to inform our codesign activities and provide initial informative metrics in our discussions with vendors. Moreover, these base metrics, which were often previously described by rules of thumb, are helping to quantitatively guide some of our thinking as to the directions and potential improvements that we may be able to find in platform procurements or through the National Supercomputing Initiative (NSCI) which is due to start operation in the coming financial year.

Unlike performance counter based solutions, the tools described in this report provide cross platform capabilities and greater reproducibility between generations of architectures. The inability to utilize counters for this kind of work remains a frustrating aspect to our codesign activities but is in part driven by the many processor/hardware designs being offered (which then lead to varying implementations that cannot count the same hardware activity the same way during execution). By being active at the executable level, APEX tools are able to obtain metrics that provide deeper understanding than is currently possible through hardware counters alone. We argue that analysis from both types of tool is desirable, perhaps even required, for this kind of study.

Throughout this LDRD there has been a keen focus of engagement - both with industry leading vendors and with production code teams at Sandia. The collaborations with industry have involved providing some and other metrics, not reported here, to Intel, AMD, Hewlett Packard and IBM helping to shape deeper discussions as to the properties of our algorithms. With production code teams we have been able to help drive a focus on vectorization and assess slow code (often the result of high levels of memory access or gather/scatter instructions). Much more remains to be done, the porting of these tools to support analysis on Intel's Xeon Phi Knights Landing architecture remains ahead as well as deepening the capabilities to support even finer levels of analysis.

As a final addition late in the LDRD a collaboration with the Kokkos developers (Carter Edwards and Christian Trott) lead to the development of the KokkosP interface – initially prototyped as a mechanism for dynamic binary attachment within APEX before the PIN tool development showed promise. The interface utilizes dynamic loading of profiling libraries

to provide performance information within the context of Kokkos parallel patterns. Use of the interface for the ASC L2 Trilab milestone in FY15 showed that the tools connected to KokkosP were able to provide insight even in environments (such as the OpenMP backend) where profiling was not currently possible. The ability to now connect this interface to APEX-based tools is a further extension which allows us to provide vectorization and memory access ratios to performance critical kernels written in Kokkos to a wide community of developers.

In summary, APEX has been a broad project encompassing architecture research, the development of tools and interfaces as well as the application of these tools to provide the first insight for Sandia to the low-level mechanics of many of the laboratories' most important algorithms. The tools developed are robust and support multi-threading, priming them for further use during the porting of applications to Trinity. Further, the research activities have created an environment in which it was possible to reach out to code teams and begin collaborations which will be essential as we continue to move towards our Exascale ambitions. The capabilities described in this report are just the start of greater levels of sophistication that Sandia will bring to the assessment of production algorithms in the coming years.

References

[1] Michael A Bender, Jonathan Berry, Simon D Hammond, K Scott Hemmert, Samuel McCauley, Branden Moore, Benjamin Moseley, Cynthia A Phillips, David Resnick, and Arun Rodrigues. Two-Level Main Memory Co-Design: Multi-Threaded Algorithmic Primitives, Analysis, and Simulation. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium 2015 (IPDPS15)*, May 2015.

[2] Ron Brightwell. A Comparison of Three MPI Implementations for Red Storm. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 425–432. Springer, 2005.

[3] Mark Charney. XED2 User Guide, 2010.

[4] M Christon, D Crawford, E Hertel, J Peery, and A Robinson. ASCI Red - Experiences and Lessons Learned with a Massively Parallel Teraflop Supercomputer. *Proceedings of the IEEE/ACM Supercomputer 1997*, 1997.

[5] H Carter Edwards and Daniel Sunderland. Kokkos Array Performance-Portable Manycore Programming Model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10. ACM, 2012.

[6] H Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. Manycore Performance-Portability: Kokkos Multidimensional Array Library. *Scientific Programming*, 20(2):89–114, 2012.

[7] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency. *Intel White Paper*, 2008.

[8] S.D. Hammond. Challenges of Codesign. In *ASC Principal Investigators' Meeting*, February 2015.

[9] S.D. Hammond. Designing the Performance Tools of Your Dreams. What do We Have and What Do We Want? In *Multicore Assembly Working Group, Sandia National Laboratories, NM*, September 2015.

[10] S.D. Hammond. Preparations for Trinity KNL. In *J34X Extreme Scale Joint Working Group (NNSA Trilabs and AWE)*, February 2015.

[11] S.D. Hammond. Why LINPACK (and STREAM) Aren't the Answer. In *Vendor Discussions, 2015*, 2015.

[12] Tarush Jain and Tanmay Agrawal. The Haswell Microarchitecture – 4th Generation Processor. *International Journal of Computer Science and Information Technologies*, 4(3):477–480, 2013.

[13] C.-K. Luk, Cohn R., Muth R., Patil H., A. Klauser, G. Lowney, S. Wallace, Reddi V.-J., and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation (PLDI) 2005*, 2005.

[14] R. Mahesh, D. Doerfler, and S.D. Hammond. Trinity Benchmarks on Intel Xeon Phi (Knights Corner), January 2015.

[15] James L Tomkins, Ron Brightwell, William J Camp, Sudip Dosanjh, Suzanne M Kelly, Paul T Lin, Courtenay T Vaughan, John Levesque, and Vinod Tipparaju. The Red Storm Architecture and Early Experiences with Multi-core Processors. *Technology Integration Advancements in Distributed Systems and Computing*, page 196, 2012.

[16] Keith D Underwood, Michael Levenhagen, and Arun Rodrigues. Simulating Red Storm: Challenges and Successes in Building a System Simulation. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

DISTRIBUTION:

1	MS 0110	D.E. Womble, 01220
1	MS 1322	K.F. Alvin, 01420
1	MS 1319	J.A. Ang, 01422
1	MS 1319	R. Benner, 01422
1	MS 1319	J. Cook, 01422
1	MS 1319	J. Laros, 01422
1	MS 1319	S.D. Hammond, 01422
1	MS 1319	K.S. Hemmert, 01422
1	MS 1319	D.R. Resnick, 01422
1	MS 1319	A.F. Rodrigues, 01422
1	MS 1319	G.R. Voskuilen, 01422
1	MS 1320	R. Hoekstra, 01426
1	MS 1320	H.C. Edwards, 01426
1	MS 1320	M.A. Heroux, 01426
1	MS 1320	C.R. Trott, 01426
1	MS 0845	S.A. Hutchinson, 01540
1	MS 0845	M. Glass, 01545
1	MS 0823	D.J. Pavlakos, 09326
1	MS 0807	M. Rajan, 09326
1	MS 0807	D. Dinge, 09326
1	MS 0801	J.P. Noe, 09328
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories