Navigating An Evolutionary Fast Path to Exascale

R.F. Barrett, S.D. Hammond, C.T. Vaughan,
D.W. Doerfler, M.A. Heroux
Sandia National Laboratories
Albuquerque, NM, USA
Email: rfbarre,sdhammo,ctvaugh,
dwdoerf,maherou@sandia.gov

J.P. Luitjens
NVIDIA Corporation
Santa Clara, CA, USA
Email: jluitjens@nvidia.com

D.Roweth
Cray, Inc.
Reading, UK
Email: droweth@cray.com

***Abstract*—The computing community is in the midst of a disruptive architectural change. The advent of manycore and heterogeneous computing nodes forces us to reconsider every aspect of the system software and application stack. To address this challenge there is a broad spectrum of approaches, which we roughly classify as either revolutionary or evolutionary. With the former, the entire code base is re-written, perhaps using a new programming language or execution model. The latter, which is the focus of this work, seeks a piecewise path of effective incremental change. The end effect of our approach will be revolutionary in that the control structure of the application will be markedly different in order to utilize single-instruction multiple-data/thread (SIMD/SIMT), manycore and heterogeneous nodes, but the physics code fragments will be remarkably similar.**

Our approach is guided by a set of mission driven applications and their proxies, focused on balancing performance potential with the realities of existing application code bases. Although the specifics of this process have not yet converged, we find that there are several important steps that developers of scientific and engineering application programs can take to prepare for making effective use of these challenging platforms. Aiding an evolutionary approach is the recognition that the performance potential of the architectures is, in a meaningful sense, an extension of existing capabilities: vectorization, threading, and a re-visiting of node interconnect capabilities. Therefore, as architectures, programming models, and programming mechanisms continue to evolve, the preparations described herein will provide significant performance benefits on existing and emerging architectures.

***Index Terms*—scientific applications; high performance computing; parallel architectures.**

I. Introduction

High Performance Computing (HPC) architectures of the coming decade will be significantly different in structure and design than today. We have already seen clock rates and node counts stabilize, and core counts increase. Now emerging are increased vector lengths, greater levels of hardware-enabled concurrency and new memory architectures that are strongly non-uniform and may soon lose cache coherency. Adoption of new, potentially immature, technologies presents many challenges including hardware reliability, scalability and, in the case of many proposed technologies, programmability and performance portability. Since proposed designs represent a radical departure from existing petascale technologies, new research projects have started in order to identify and develop solutions for many of these problems. One of the greatest concerns facing programs such as the U.S. Department of Energy's Advanced Simulation and Computing (ASC) initiative in the United States is how best to port full applications that have been developed over nearly two decades. These applications consist of millions of lines of source code implemented in a variety of programming languages (some of which use non-standard features) and utilize tens of supporting libraries. These applications codify significant bodies of knowledge which have developed over multiple generations of scientists. Alongside the demands of porting such large codes are the continued requirements associated with on-going programmatic-level work. Put simply, science discovery cannot stop while applications and algorithms are rewritten for future systems and re-validated to ensure correct scientific output.

In this context, many in the HPC industry question how full science applications will be ported to new architectures. On one side of the debate is a view that significant application rewrites will be required in order to obtain the full potential performance of new hardware. On the other side is a view, described above, that the pedigree of existing code and the complex science which it embodies must be gradually evolved and augmented over time for new platforms so that scientific delivery can continue and the cost associated with porting is reduced or at least amortized. We regard these views as the *revolutionary* and *evolutionary* approaches to Exascale, respectively.

The work reported in this paper (and in expanded version in [5]) describes an initial series of experiments performed in the *evolutionary* context. We note that studies using revolutionary approaches are also underway and will be reported in future publications. Our work focuses on three levels of porting which we expect to be commonplace choices for the modification of codes on future platforms: (i) optimization within the processor core, typically investigating the improvement of vector-level parallelism and the modification of code to exploit near memory subsystems; (ii) optimization of code for the compute node as a whole using techniques such as thread-level parallelism, introduction of compiler directives to drive compute offloading, data motion reduction and adaptation for node-level topologies such as non-uniform memory architecture (NUMA) domains, and, (iii), optimization of inter-node communication to improve message pipelining or to utilize novel features in

advanced network interconnects.

The specific contributions of this work are:

- Documented porting and analysis of several key Sandia mini-applications (miniapps) running on advanced computing architectures. The use of miniapps from the Mantevo suite enables us to draw conclusions that are relevant to production codes used by the National Nuclear Security Agency (NNSA) and ASC programs. We employ novel technologies to aid in this porting including traditional OpenMP pragmas as well as introduction of OpenACC directives to enable execution on GPUs, and intrinsic functions to improve levels of compiler vectorization;
- Benchmarking the effects on runtime of changes in hardware and environment configuration, in particular changes in memory bandwidth, MPI-rank placement and the varying use of threads/MPI ranks to use available processor resources. Such studies enable us to investigate the opportunity for optimizations outside of traditional changes in source code and reflects our on-going view that optimization of runtime encompasses a wide range of options including software configuration as well as design. We note that such options can have significant impact at scale and demonstrate the effect of such options for runs of over 16,000 processor cores;
- Highlighting of several architectural parameters which serve as bottlenecks or limits to further improvements in performance. A natural effect of early generations of hardware is that a number of optimization opportunities are likely to still be present in the design. In our work we use miniapps to identify these and discuss how, when addressed, these may provide improvement in runtime performance.

Non-uniform levels of maturity of the systems used for this work prevents meaningful direct hardware-to-hardware comparisons, so we instead provide relative measures of improvement for each experiment. As the hardware and software stacks associated with the platforms, we will report direct performance comparisons.

A. Related work

The number and breadth of challenges associated with preparing for multi-petascale and exascale-class computing are significant and have helped to create a rich set of academic investigations touching on comparison of computer architectures [3], [20] optimization for specific classes of hardware [8], [12] and programming languages and mechanisms [2], [14], [22], [25].

II. Programming Models and Environments

Programming models (abstractions used to reason about program design and implementation) and programming environments (compilers and tools used to implement software, correlated in design to one or more programming models) are some of the most challenging and dynamic elements on the path to exascale computing. Single program multiple data (SPMD) built on top of MPI has by far been the dominant programming model and environment pair since the emergence of distributed memory computing two decades ago. There is overwhelming evidence that single-level SPMD (statically assigning a process per core) is insufficient to achieve optimal performance. Even more importantly, this approach will not scale with the performance potential of future systems. If we are going to continue tracking future performance trends, we need to augment or replace existing strategies.

Revolutionary approaches in this area arguably include Chapel [13], ParalleX [14] and Swarm [2], since these languages or execution models, or both, would require a complete redesign of an existing application. Although future systems may demand such efforts, most application teams are using a more evolutionary approach, sometimes called MPI+X, by combining SPMD (via MPI) with one or more node-level programming environments such as OpenMP [11], Intel Threading Building Blocks (TBB) [24], CUDA [23] or C++11 threads.

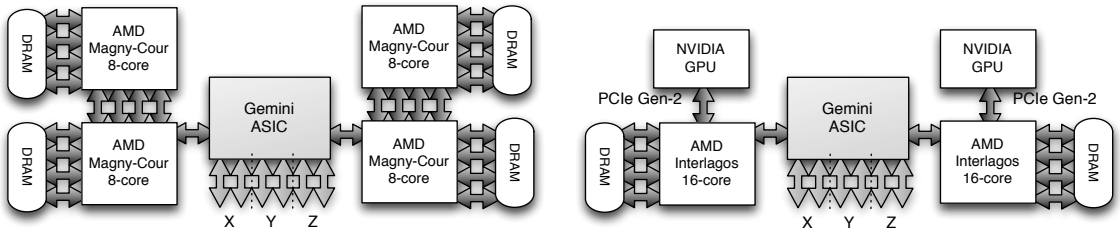
Although MPI+X may appear incremental, it is in fact very challenging to implement. In order to successfully scale on current and future heterogeneous and manycore nodes, the computational kernels of an existing (MPI) application must be redesigned to support dynamic partitioning and scheduling of work by the node-level runtime system. Typically this is most easily done by encapsulating kernels in stateless functions that can be parametrized to dynamically execute some runtime-selected portion of work such that, when scheduled to a processing element, the computation is performed efficiently. All node-level programming models and environments are compatible with this approach and some, such as TBB and CUDA, explicitly require it.

Interestingly, by going through the above refactoring processing, we not only make MPI+X work well, but we position ourselves well for any future programming and execution model. The effort of exposing and encapsulting parallelizable computations in this manner is intrinsically valuable. Furthermore, by optimizing node-level performance for runtime systems such as CUDA, we also move in the direction of hiding latency via task concurrency. Optimizing occupancy rates in CUDA is a harbinger of the kind of reasoning needed for efficient use of ParalleX and Swarm.

III. Experimental Platforms

We employ a breadth of architectures in pursuit of preparing application codes for exascale: *Teller*, a cluster of AMD Llano Fusion APU processors; *Cielo*, a capability-class Cray XE6 supercomputer, and *Curie*, a small installation of Cray XK6 nodes. Additional machines are used for experiments which require hardware or BIOS reconfiguration in order to reduce impact to other system users.

While these machines are not a complete set of prospective future architectures, they provide a distinct set of important configurations and characteristics which we expect future hardware systems to contain.



(a) Cielo/XE6 Node Architecture

(b) Curie/XK6 Node Architecture

Fig. 1. Cray X-Series Node Architectures

Each architecture meets our requirement of running existing MPI applications through execution on conventional commodity processor cores, requiring only recompilation. However, in order for the potential performance of each platform to be realized it is necessary to augment the code with additional programming languages or models (including OpenMP[11], OpenACC [1], and OpenCL [16]) that can target the specialized compute hardware within.

1) **AMD Fusion ("Teller"):** Teller is a 104-node cluster of single-socket AMD A8-3850 Llano Fusion APU nodes. Each APU comprises four K-10 AMD x86-64 cores running at 2.90GHz. Cores have private 64kB L1 instruction and data caches and a 1MB level-2 (giving a 4MB L2 in total). The GPU portion of the APU is a modified 400-core Radeon HD 6550D which runs at 600MHz. GPU cores are 5-way SIMD. In order to enable the study of memory performance, 100 cluster nodes contain 16GB DDR3 memory clocked at 1600MHz and 4 nodes contain 8GB DDR3 memory at 1866MHz. All nodes are equipped with a single 256GB Micro C400 SSD storage device. The machine interconnect employs QLogic QSFP QDR InfiniBand.

2) **Cray XE6 ("Cielo"):** Cielo, the latest Advanced Simulation and Computing (ASC) capability machine, consists of 9,216 nodes, of which 8,944 are compute nodes and 272 are service and I/O processing nodes. Although not technically a testbed system since the architecture is widely available, Cielo provides some new capabilities and hardware features which we expect will be extended and refined in future architectures. Each Cielo node consists of two oct-core AMD Opteron Magny-Cours processors connected via HyperTransport 3 links. Each Magny-Cours processor is divided into two memory regions (see Figure 1(a)), or "NUMA nodes", each consisting of four processor cores. Nodes are connected using Cray's Gemini 3D torus high-speed interconnect. A Gemini ASIC permits connectivity of two compute nodes each with a dedicated HyperTransport-3 connection.

3) **Cray XK6 ("Curie"):** The XK6 is the latest refinement to the X-family of machines developed by Cray. In this new series one processor is removed from each compute node and replaced with an NVIDIA Tesla-series GPU using a PCI-Express Gen-2 bus for connectivity (see Figure 1(b)).

At the time of writing the GPU used for node design is an NVIDIA "Fermi"-class GPU with 6 GBytes of GDDR5 memory and 16 streaming multiprocessors (SM), each containing 32 thread processing cores. are provided on the GPU card, enabling high speed transfers to/from the on-chip compute units. The Curie testbed system consists of 52 nodes, with AMD Opteron 2.1GHz 16-core Interlagos 6272 processors and 32GB of system memory. The Interlagos is also of interest in our studies since it utilizes the recently announced Bulldozer core in which a core-pair each have two integer processing units (one per core) but a unified floating-point pipeline which is intended to reduce cost and power consumption.

IV. Overview of the Mantevo Project

The Mantevo project [17] provides a set of proxies, or "miniapps," which enable rapid exploration of key performance issues that impact a broad set of scientific applications of interest to the ASC and broader HPC community. Mantevo miniapps are tools[1] with uses throughout the co-design space [15]. They are intended to be fluid and a mechanism to explore issues relating to hardware performance, programmability, porting, etc. As part of the on-going work in developing miniapps under Mantevo, a comprehensive initial validation exercise [4] has recently been conducted to ensure the first full release of codes is able to provide strong behavioral correlation to parent physics and engineering application currently in use.

Unlike a benchmark, the result of which is a metric to be ranked, the output of a miniapp is a richer set of information, which must be interpreted within some, often subjective, context. We distinguish this from a *compact-application* whose purpose is to replicate a complex domain-specific behavior being used in a parent application. Miniapps are designed specifically to capture some key performance issue in the full application but to present it in a simplified setting which is amenable to rapid modification and testing. Note that this is also distinct from a *skeleton application*, which is typically designed to focus on inter-process communication often producing a "fake" computation. Miniapps instead create a meaningful context in which to explore the key performance issue. Within many of the ASC programs, miniapps are developed and owned by application code teams; are limited to $O(1K)$ source lines of code (SLOC) and are intended to be modified with the only constraint being the continued relevance to parent application.

[1] http://mantevo.org

Mantevo miniapps provide base sequential reference implementations written in either C, C++, or Fortran, with optional parallelization configured using MPI and OpenMP. Additional hybrid implementations are provided for exploration using technologies such as CUDA, OpenCL, TBB, Cilk Plus, qthreads, and Chapel.

In this work we employ three principle Mantevo miniapps: (i) miniMD, a representation of a Lennard-Jones molecular-dynamics problem used in the LAMMPS parent application; (ii) miniFE, a finite-element miniapp which includes a finite-element assembly step as well as a solving phase using the CG method (representative of the Charon parent code), and (iii) miniGhost, a bulk-synchronous stencil code that replicates the behavior of finite difference and finite volume algorithms. For the purposes of this work, miniGhost is configured to represent the behavior of the CTH shock-hydro application used at Sandia.

V. Methodology

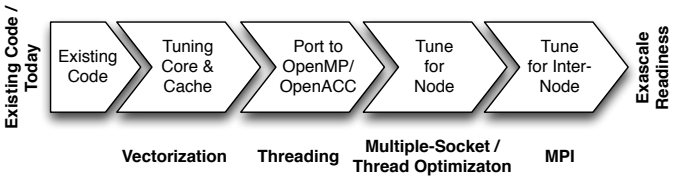


Fig. 2. Steps in an Evolutionary Path to Exascale Readiness

In this paper we document our experiences with attempting an evolutionary migration of code from existing petascale machines to exascale-class prototype hardware. The focus is therefore on maintaining as many of the existing investments in code as can be productively retained and, where possible, adapting the structure for new classes of hardware. Towards this end have empirically found our code modifications follow a largely similar path regardless of the hardware technology being employed. Figure 2 represents our migration strategy that starts with our existing code and proceeds through a series of steps to port and optimize this for future systems. In practice some of these steps may occur in parallel or out of the order described. The principle steps of our porting have so far been the following: 1) Vectorization 2) Threading 3) Tuning for the Compute Node and 4) MPI/Inter-Node Parallelism

The stages of this process capture our intent to introduce improved levels of parallelism into our code which will be essential for application performance in the coming decade. Whilst the specifics of how this parallelism is presented to the compiler or the machine are unique to each hardware platform, the locating of parallelism is primarily a property of the algorithm being used. Our experience is that the knowledge of parallelism obtained through this process can therefore be re-used between different compute architectures and programming models reducing the time and cost associated with porting.

VI. Case Study: Initial Porting of Mantevo Mini-Applications to Future Architectures

In this last section of the paper we describe a series of short case studies which describe the value of mini-applications in assessing the capabilities and characteristics of future computing architectures and programming models. In so doing we are able to survey state of the art prototype computing architectures and provide initial commentary on how applications may be mapped to them using evolutionary modifications to the application. In order to separate the various levels of tuning being conducted, our results are split into three sections: (1) optimization within the processor core; (2) optimization within a whole compute node and, (3), optimization between compute nodes.

In this section we use the term *core*, *processor*, and *node* in a flexible context since precise terminology is still being refined throughout the community. For purposes herein, we view a core as being as a unit which is capable of performing calculation including traditional processor cores or lightweight cores as found in new hardware types such as a GPU; we view a processor as essentially being a socket and a node is a network endpoint.

A. Processor Core Performance

Future computing hardware is expected to provide increased parallelism in the processor core through the availability of, and increased width of, vector registers which are able to perform a single instruction over multiple pieces of data simultaneously. With the introduction of MMX instructions by Intel in 1995 and subsequent additions in the form of SSE and latterly AVX, many of the available floating-point operations in commodity processors require codes to exploit high levels of data parallelism in order to achieve a high proportion of peak chip performance. A well tuned compiler can very often detect opportunities for vectorization providing the code is structured in a manner that the compiler can safely determine the introduction of vectored instructions will not violate the original statement ordering. Where code control is complex or inter-statement dependencies exist, compilers are often unable to generate vectorized code resulting in slower execution. One potential approach to addressing this issue is for the programmer to employ manual vectorization through the introduction of *intrinsic* operations – a library of routines which communicate how the developer would like vectorization to occur. Such code constructs allow developers to easily express low-level vector parameters and data operations that either expand directly to assembly instructions or provide programmer intent to the compiler allowing it to manipulate these statements into vectorized operations. Typically not for the faint of heart, we apply this approach herein in order to illustrate the potential performance advantages with a view that maturing compiler technology may provide additional opportunities for vectorization in the future and that the performance identified helps us to assess what the hardware may in fact be capable of.

TABLE I
SPEEDUP OF MANUAL VECTORIZATION FOR MINIMD FORCE CALCULATION AND FULL APPLICATION RUNTIME.

	Force (Speedup (x))	Total (Speedup (x))
AMD A8 3850 APU, 2.90GHz, SSE4a		
Single Precision	1.26	1.21
Double Precision	1.56	1.49
Intel Westmere 5690, 3.47GHz, SSE4.2		
Single Precision	1.57	1.43
Double Precision	1.42	1.33
Intel Sandy Bridge, 2.60GHz, AVX		
Single Precision	1.70	1.49
Double Precision	1.61	1.43
Intel Ivy Bridge (Core i7), 3.40GHz, AVX		
Single Precision	1.84	1.64
Double Precision	1.91	1.75

Our initial inspections into poorer than expected performance of miniMD on commodity processors has shown that the force compute loop (which is responsible for up to 90% of serial runtime) cannot be vectorized due to the complex pointer behavior being employed as well as sparse operand loads from memory. The use of double pointer indirection to map data structures into the compute kernel prevents the compiler from determining whether vector instructions can be utilized without violating ordering constraints despite a valid vectorization being possible from an algorithmic perspective.

The force function computes the interaction forces between each pair of atoms that exist in a specific neighbor list. Due to the sparse nature of the atom information and the condition operations associated with identifying whether the atom pair is within a cut-off zone, automatic vectorization of this code is particularly challenging. However, the high proportion of execution associated with the function makes this a candidate for optimization. In order to address the poor level of vectorization the main force compute loop was instrumented with vector intrinsic operations enabling the compiler to generate code with increased levels of vectorization. Table I shows the speedup obtained through the use of SSE4.2 and AVX intrinsic operations on several processors. Although the Ivy Bridge processor is not a server grade processor we are using high-end desktop processors to provide insight to future server versions of the processor.

SSE provides a 128-bit wide vector unit (four single-precision operands or two double-precision operands) and AVX provides 256-bits enabling a maximum speedup of floating point calculations equivalent to the vector width (up to 8x in the case of single precision AVX values). Although each operation within the force kernel is vectorized using an intrinsic operation, the speedup obtained is lower than the theoretical maximum as some instructions over vector registers are serialized within the processor core and other architectural bottlenecks such as memory operations are not executed in parallel. The output of the vector intrinsic instrumented codes is not *identical* to non-vectored source as the operations may lead to a change in rounding effects, but has been thoroughly tested to ensure the results are acceptable to domain scientists. Our experience of rounding has shown that the introduction of intrinsics can yield subtle changes in output and even execution behavior and therefore careful design and post-implementation testing are required.

As we look forward the coming arrival of the Intel MIC architecture with an increased vector width, we predict that the addition of intrinsics to key application kernels may become more commonplace. We further expect the structure of the refactored intrinsic force kernel to be reusable on MIC platforms moving forward with only minor changes to adapt to the wide vector registers. Within the evolutionary context, adaption of key kernels using intrinsics therefore allows us to migrate applications to new platforms and provide significant improvement in execution speed with changes isolated to only short sections of code. Although intrinsics are not available for the GPU in the same sense, the knowledge of algorithm parallelism is fundamental to informing us of future ports to such architectures. Furthermore, many of the improvements being prepared for future exascale-class platforms can be reused on a number of our existing systems.

B. Intra-node performance

Effectively exploiting the performance potential provided by increasingly complex node architectures is seen as one of the major challenges for on-going code development. In this section we provide three examples illustrating several issues that the application developer may need to consider on an increasingly complex compute node. Specifically we focus on : (1) the porting of miniGhost to NVIDIA GPUs using the recently announced OpenACC compiler directive toolkit, demonstrating initial identification of parallelism in the algorithm and the performance obtained from OpenACC; (2) the porting of the miniFE Finite Element assembly phase to NVIDIA GPUs using CUDA which identifies the high level of register spilling present in the code and discusses how this will be addressed in future GPU systems and, finally, (3) a study of miniFE assembly and solve phase sensitivity to changes in memory bandwidth. The issues raised in these small studies demonstrate the value of mini-applications in identifying potential performance bottlenecks or sensitivities. The information being obtained through this work is able to drive analysis of larger applications as well as hardware and can have real impact in informing our hardware selection choices and subsequent optimization activities.

1) Finite difference stencils: Computation in miniGhost is based on a triply nested loop, whereby each point in the domain is updated as a function of the average over adjacent points. The simplicity of this computation and the ability to easily configure varying levels of compute complexity let us expose and explore issues expected to significantly impact the performance of full applications that employ difference stencils and the halo exchange.

The hybrid MPI + OpenMP version began with a straightforward wrapping of the outer-most loop the `!$OMP PARALLEL DO` directive. It was also then a straightforward port to OpenACC, replacing this directive with the `!$acc parallel`

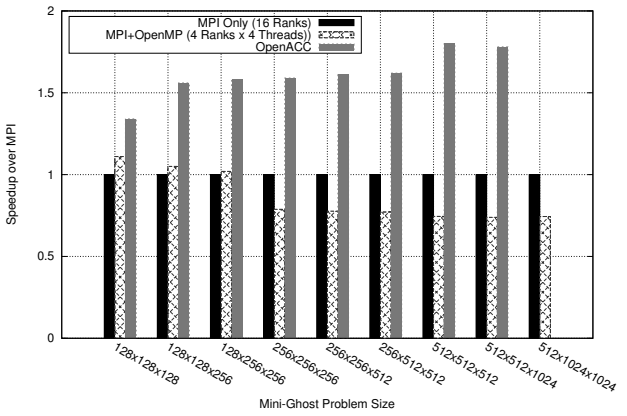


Fig. 3. Performance of varying miniGhost problems on an XK6 node

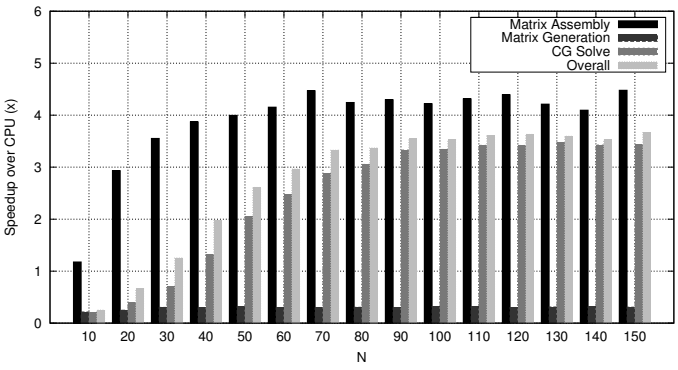


Fig. 4. Speedup of miniFE CUDA Implementation (NVIDIA Fermi M2090 vs. Hex-Core 2.7GHz E5-2680

`loop` directive. However, on a node with a memory hierarchy, such as that on a dual-socket node, the memory affinity is different: with OpenMP, the arrays should be physically distributed across the memory hierarchy of the processing cores (via first-touch) while with a hybrid off-load system, the entire array should be resident on a single processor's memory so that the movement to the device is straightforward. The MPI-everywhere version automatically maintains processor core and memory affinity.

The stencil loop required no additional use of OpenMP directives. The OpenACC version enabled the means for effectively mapping the data onto the GPU, required to achieve acceptable performance. The Fermi processor is organized into 16 groups of 32 cores, so a directive is used to map the computation as $x-y$ slices of `GRID` to 16 blocks (*num_gangs*), mapping them each as vectors (*vector_length*) of length 32 onto the 32 core warps.

Single node performance on Curie of the MPI-everywhere, MPI+OpenMP, and MPI+OpenACC implementations, illustrated in Figure 3, shows that the GPU gives a speedup over the MPI-everywhere ranging from around 25% to 80% as the amount of work increases. However, if the data to be operated on must be moved to and from the GPU, the advantage reverses. The best MPI+OpenMP configuration of 4 MPI ranks per node each with 4 OpenMP threads outperforms the MPI version by about 10%, but this quickly reverses as the problem size increases, with performance decreasing to 80% of the MPI version. It is beyond the scope of this paper examine this more closely, though its likely that this is an artifact of the first generation Interlagos node architecture and thus its reasonable to expect that it will be addressed in the next generation Trinity node.

Direct comparison of performance between the host and device is problematic since memory sizes differ significantly, a situation common to their sorts of architectures. On Curie, the GPU device has significantly less memory that that available to the host (6 GBytes vs. 32 GBytes), and therefore the node can execute the larger problem shown on the graph while the GPU cannot. From an application perspective, the ultimate comparison then is multi-node strong scaling, examined in Section VI-C1.

2) miniFE on a GPU: MiniFE consists of three principle phases: generation of the matrix structure, assembly of the finite-element matrix, and the solution of the sparse linear system using the Conjugate Gradient method. Here we focus on key performance limiters in porting the matrix assembly phase of the algorithm to an NVIDIA GPU using the CUDA programming model.

The assembly phase involves computing the element operators for each element and then summing the operators into a final matrix. We parallelize this phase by having threads operate on separate elements with the computation of the element operator and the summation into the linear system performed within a single kernel. Although the computation of the element operators is embarrassingly parallel, the summing into a linear system requires synchronization to avoid data race conditions. The use of a single kernel is preferred in this instance because it avoids having to store the state for the element operator and then having to later re-read that state during summing into the linear system.

By using one thread per element we were able to leverage the original code for the construction of the element operator subject to additions for compilation to a CUDA kernel and a modification of the code to use the ELL sparse-matrix representation [7] of the original compressed-row (CSR) form. Atomic addition operations are employed in the kernel to prevent race conditions in updating the global matrix.

The computation of the element operator involves a number of floating-point heavy operations including computing the matrix determinant and the Jacobian. The large number of floating point operations suggest that the performance should be FLOP limited but analysis using NVIDIA's compute profiler has shown that the performance is in fact bandwidth bound due to register spilling.

The cause of this register spilling can be identified as the element operator which requires a large thread state, including 32 bytes for node-IDs, 96 bytes for node coordinates, 512 bytes for the diffusion matrix, 64 bytes for the source vector as well as data to store the Jacobian and matrix determinant. The

Fermi GPU architecture supports up to 63 32-byte registers per thread limiting the total register storage to 252 bytes. As a result of this limit, any additional state must be spilled to at least L1 cache and potentially further to L2 or memory. Since the L1 cache can be up to 48kB in size but is shared by 512 threads this can result in as little as 96 bytes of L1 cache storage per thread. In addition, the L2 cache is 768kB shared by 8192 threads, again leaving only 96 bytes of storage per thread. Since L1 and L2 are insufficiently sized to store the required operator state, registers are spilled to global memory causing the computation to become bandwidth bound.

One method to improve the performance of bandwidth bound kernels is to increase the occupancy. However, in this case, the kernel's occupancy is limited by register usage. Since the register usage is higher than is available in hardware it is not possible to increase this occupancy without further increasing register spilling.

We tuned the kernel to reduce register usage, including algorithmic changes that exploiting symmetry in the diffusion operator and reordering computations so that data is loaded immediately prior to being used. We have also applied several traditional optimization techniques including pointer restriction, inlining of functions, and unrolling of loops. Finally, we also position a portion of the state in shared memory and experimented with L1 cache sizes. The best performance is achieved by placing the source vector into shared memory and enabling a larger L1 cache. Whilst these optimization greatly reduce register spilling, 512 bytes of state is still spilled per thread. To ensure fair comparison, all optimizations that were applicable to the original CPU code were back ported also improving the CPU performance.

The performance of the CUDA version of miniFE was compared to the MPI-parallel version of miniFE running on a Tesla M2090 and a hex-core Intel Xeon 2.7GHz E5-2680. We tested for various problem sizes of N^3 hexahedral elements. The speedup for each of the three phases of the algorithm is reported in Figure 4.

The assembly realizes a 4x speedup and the solve phase is 3x faster. The generation of the matrix structure exhibits a slowdown because it is computed on the host in CSR format, transferred to the device, and then converted to ELL format. Whilst possible to move this computation to the device, the low proportion of time consumed by these operations does not dominate performance.

Future generations of NVIDIA systems are expected to address some of the findings from this study, including an increased number of registers per thread and increases in the size of L1 and L2 memories. Improvements in the CUDA compiler may also lead to a reduction in the number of register spills or the impact that register spills will have on execution time.

3) The Impact of Memory Speeds: It is widely accepted that the performance of the memory sub-systems used in future exascale computers will improve substantially. This is driven by a realization that application performance, even on contemporary systems, is frequently limited by the "memory wall" [21]. If compute devices experience rapid improvements in calculation throughput, memory will *need* to be improved significantly if applications are not to become entirely throttled on memory access. However, the question of whether the improvement in memory performance will exceed or even match that of compute hardware is still unanswered. Therefore, there is a very real risk that applications will need to execute using lower memory bandwidth in the future.

In this section we describe a study in which we alter the clock rate of memory components in a system through BIOS control, effectively slowing the rate at which memory can process requests. This enables an analysis of code performance where there is a growing divergence between compute throughput and that of memory (which may be the effect of significant improvement in compute hardware that is unmatched by equivalent improvement in memory subsystems). Table II presents the relative effect on sections of miniFE and Charon (the parent application to miniFE) runtime as the clock rate of memory is lowered from 1333MHz to 1066MHz and 800MHz. The processor used for this study is a dual-socket oct-core AMD Interlagos running at 2.9GHz. The assembly time in miniFE and the Jacobian generation of Charon is unaffected by the change indicating that these sections of code are not predominantly memory bound, whilst the runtime of the conjugate-gradient (CG) solver increases by approximately 16% in the 800MHz case. Of interest is that miniFE is able to closely track equivalent changes in its parent code for the solve phase (which in both codes is the dominant contributor to runtime) giving us confidence in the relevance of our studies using mini-applications.

From this short study we can begin to assess the likely impact that a reduced per-core memory bandwidth may induce. A runtime increase of 16% is approximately half of the drop in memory bandwidth, indicating that miniFE and Charon are clearly highly sensitive to memory bandwidth in their solve phases but that some of the bandwidth loss can be covered either through efficient use of the memory hierarchy or latency hiding by the processor through the use of prefetching and deep instruction pipelines.

TABLE II
RELATIVE RUNTIME SLOWDOWN OF MINIFE AND CHARON FROM REDUCED MEMORY FREQUENCY (RELATIVE TO MEMORY FREQUENCY OF 1333MHZ, LOWER IS WORSE)

	800MHz	**1066MHz**	**1333MHz**
miniFE Mini-Application			
Finite Element Setup	0.996	1.000	1.000
CG Solve	0.841	0.957	1.000
Charon Device Simulation Application			
Prec/Newt	0.960	0.980	1.000
Jac/Newt	1.000	1.000	1.000
Adv/Newt	0.920	0.970	1.000
Solve/Newt	0.840	0.940	1.000

C. Inter-node performance

Our goal of an evolutionary path includes the assumption that inter-node parallelism will continue, in the foreseeable future,

to be implemented using functionality provided by MPI. (This does not rule out the use of MPI in a revolutionary approach.)

In this section we explore some issues associated with the ubiquitous nearest neighbor communication pattern. Our work is informed by CTH, an explicit three dimensional multi-material shock hydrodynamics code [18]. CTH models high-speed hydrodynamic flow and the dynamic deformation of solid materials, and solves the equations of mass, momentum, and energy in an Eulerian finite volume formulation. MiniGhost, shown to effectively represent the inter-process communication requirements of CTH, provides a tractable means for exploring strategies for improving the performance characteristics of the full application.

We begin by examining miniGhost on Curie. Next, we address a performance issue associated with process-to-processor mapping, noticeable only at large scales. Then we investigate an alternative to the very large message strategy implemented in CTH and many other applications.

1) On the XK6: As shown in Section VI-B1, Curie's GPU provides a significant performance capability for the computation of difference stencils, but the cost of moving data between the host and device on a node overwhelms the performance of the computation. To minimize this expense, all arrays are maintained on the device, and only the halo is transferred to and from the host for MPI handling. The MPI+OpenACC implementation uses one MPI rank per node, a (current) limitation of the OpenACC implementation used here.

Problem sets were configured to mimic the profiles of CTH. Weak scaling experiments demonstrate the power of the GPU in comparison with all of the processor cores for the MPI and MPI+OpenMP implementations. Strong scaling experiments were configured to demonstrate the manner in which a domain scientist would use Curie's capabilities, highlighting the effects of maintaining the state on the GPU. Representative results are shown in Figure 5.

The weak scaling problem involves 16 variables on a $256 \times 256 \times 512$ grid per node. This ensures that the MPI and OpenMP processes on a host node have a reasonable amount of work ($128 \times 128 \times 128$ per rank or thread, resp.) while still allowing that work to fit onto the GPU device. Although the communication cost dominates the MPI+OpenACC runtime, the speed of the computation allows it to maintain its advantage over MPI and MPI+OpenMP. However, the gap closes at higher node counts.

The strong scaling problem involves 20 variables operating on a $1024 \times 1024 \times 1024$ grid. Most notable is that due to the GPUs' memory constraint relative to the host node, the OpenACC implementation requires a minimum of 32 nodes while the MPI implementations can run on eight nodes. However, at that scale, the MPI+OpenACC implementation out-performs the MPI implementation by about 40%. The MPI+OpenMP implementation becomes competitive with the MPI implementation at higher node counts, a trend we see for the strong scaling results as well as in the largest core counts on Cielo, discussed in the following sections.

The MPI and MPI+OpenMP implementations spent 10-20% and 5-10% of runtime, respectively, in communication, highlighting the computational issue for OpenMP. As seen in the following sections MPI+OpenMP outperforms MPI-everywhere on Cielo at very high scales. Interprocess communication is the sum of the work required to move data between the parallel processes, which includes the time spent packing and unpacking the message buffers as well as the time spent in MPI. For the OpenACC implementation, then, this includes the time spent moving the message buffers and partial sums between the host and device, resulting in mid-90% proportion of runtime spent in communication. We optionally aggregated the variable boundaries into a single array for host-device movement, but this increased the cost since the individual transfers could be partially hidden by the packing of the other buffers. This demonstrates that the data movement is impacted by the injection rate more than by either latency or bandwidth: the PCI-e host-device connection is able to inject the transfer and quickly return to packing the next buffer.

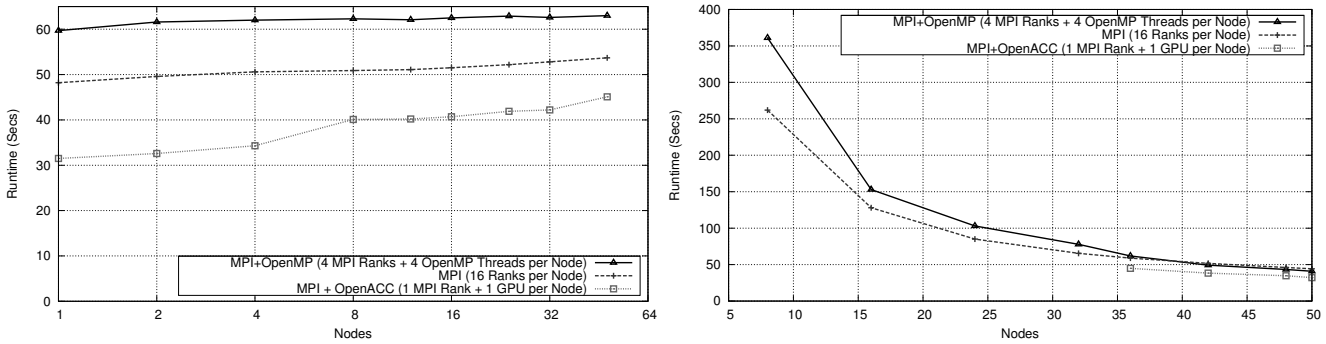
Its important to note that OpenACC is a new specification, and supporting compilers have only recently appeared. Our hope is that compilers will improve over time since we found this programming methodology to be rather easy to use.

2) Mapping processes to processors: CTH provides a typical example of a code team adapting to computing architectures: in order to avoid message latencies and exploit global bandwidth, computation is performed across as many variables as possible before an boundary exchange across those variables can be consolidated in to a single message per neighbor. But in a recently completed broad-based study of Cielo capabilities [19], the nearest neighbor boundary exchange encountered significant scaling degradation beyond 8,000 processor cores.

The problem was traced to the mapping of the parallel processes to the three dimensional torus topology. Neighbors in the x direction required a maximum of one hop and in the y direction a maximum of two hops. But the number of hops across the network (referred to as the *Manhattan distance*) was shown to increase significantly in the z direction. This combined with the very large messages of a typical CTH problem set (e.g. for the "shaped charge" problem, 40 three dimensional state variable arrays generated message lengths of almost 5 MBytes) resulted in poor scaling beginning at 8k processes, a trend that accelerated after 16k processes.

In response, we implemented a means by which the parallel processes could be logically re-mapped to take advantage of the physical locality induced by the communication requirements. In the normal mode, CTH (and miniGhost) assigns blocks of the mesh to cores in a manner which ignores the connectivity of the cores in a node. On Cielo, as with other Cray X-series architectures, cores are numbered consecutively on a node, and this numbering continues on the next node.

Blocks of the mesh are distributed to the processors row by row. Starting in one corner of the mesh, first in the x direction, then moving in the y direction for the next row, and then in the z direction when all of the blocks in a plane are assigned. This process continues until all of the blocks are assigned to



(a) Weak scaling: 256x256x256 grid per node,16 variables

(b) Strong scaling, 1024x1024x1024 grid, 20 variables

Fig. 5. Strong and Weak Scaling of miniGhost on XK6

Number of MPI ranks	Regular Order			Reordered		
	X	Y	Z	X	Y	Z
16	0.0	0.0	0.0	0.0	0.0	0.0
32	0.0	0.0	0.0	0.0	0.0	0.0
64	0.0	0.0	0.3	0.0	0.3	0.0
128	0.0	0.0	1.0	0.0	0.5	0.0
256	0.0	0.0	1.0	0.0	0.5	0.3
512	0.0	0.1	2.0	0.0	0.6	0.4
1024	0.0	0.3	2.1	0.2	1.0	0.7
2048	0.0	0.3	2.7	0.3	1.2	1.2
4096	0.0	0.3	3.7	0.3	1.2	1.2
8192	0.0	0.5	5.1	0.2	1.1	2.0
16384	0.0	0.5	4.9	0.2	1.1	2.2
32768	0.0	0.5	5.6	0.2	1.1	2.5
65536	0.0	1.1	10.2	0.2	1.6	2.8
131072	0.0	1.1	10.1	0.2	1.6	3.1

TABLE III
MINIGHOST AVERAGE HOP COUNTS ON CIELO

processors.

Our remapping algorithm assigns blocks of the mesh to the cores of the machine by groups. On Cielo, a group of blocks consists of a $2 \times 2 \times 4$ group of blocks. These blocks are then assigned to nodes as above. The result is a slight increase in the average hop counts in the x and y directions, but a significant decrease in the average hop count in the z direction. A comparison of the number of hops between the two approaches is shown in Table III.

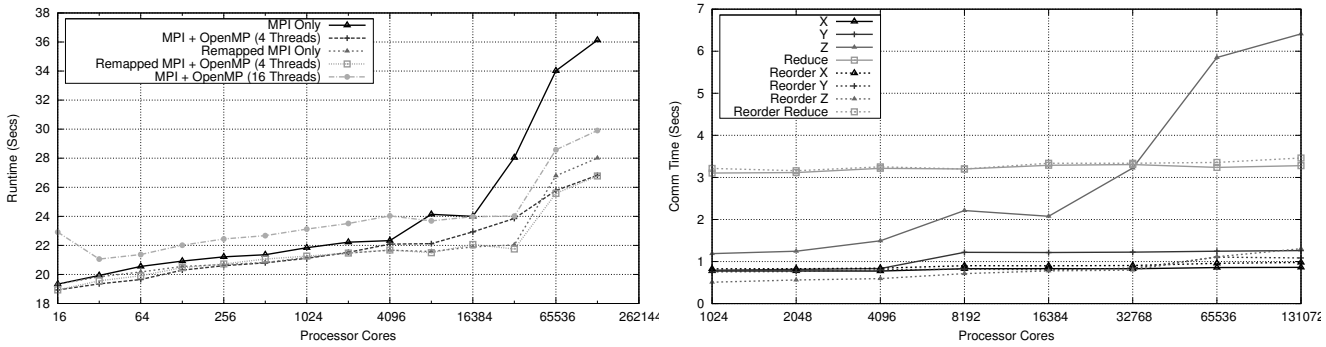
This remapping strategy results in a significant improvement in scaling performance, illustrated in Figure 6(a). Figure 6(b) shows that this is attributable to controlling the time spent sending data in the z direction. We include the time spent in the reduction sum across each grid variable (inserted after computation on each variable to add application realism as well as a synchronization point), illustrating that this functionality is not the source of the issue, scaling well regardless of the processor mapping. We do see indications of the issue at the highest processor counts, though it is less pronounced.

As discussed in the related work section above, we are exploring ways for incorporating these ideas into a more general interface. We are also exploring the use of `MPI_Datatype` in handling the non-contiguous (but patterned) face data.

3) Alternative communication strategies: Node interconnects are also evolving, driven by new node architectures as well as cost and energy conservation goals, encouraging exploration of new approaches within the context of application requirements. Interconnects are designed as a balance of global bandwidth (the ability of the interconnect to move data), inject bandwidth (the ability of the NIC to put data onto the interconnect), and injection rate (the ability of a node to place messages onto the NIC). Global bandwidth typically incurs the highest costs, both in terms of money and power consumption, and therefore we are preparing for a proportional decrease in that capability.

MiniGhost includes an application-relevant infrastructure for exploring alternative boundary exchange configurations [6]. The configuration used above mimics that of CTH, which we call Bulk Synchronous Parallel with Message Aggregation (BSPMA). The second, called Single Variable Aggregated Faces (SVAF) transmits data as soon as computation on a variable is completed, and thus six messages are transmitted for each variable (up to 40), one to each neighbor, each time step. The two $x-y$ faces are contiguous in memory, so each may be directly sent using a call to a single MPI function. The other four faces are aggregated into buffers, resulting in four messages to their neighbors. A third mode, called single variable, contiguous pieces, computational overlapping mode (SVCP), is designed for use on architectures that are strongly biased toward significantly increased message injection rates and injection bandwidth, a trend we see developing but not yet to the extent of supporting this configuration using MPI [25].

BSPMA and SVAF have been configured for MPI-everywhere as well as MPI+OpenMP. For the latter on Cielo and Curie, its best configuration is four MPI ranks on each node, each spawning four OpenMP threads. Note that this increases the size of each message in comparison with the MPI-everywhere version. Performance of these implementations on Cielo are shown in Figure 7. Effective mapping of processes to processors is again critical to achieving good scaling, and as the number of processors increases, SVAF becomes the best strategy. This is of significant interest since it reduces demand



(a) Time to solution

(b) Communication time per direction

Fig. 6. Performance of miniGhost with MPI-rank remapping on Cielo

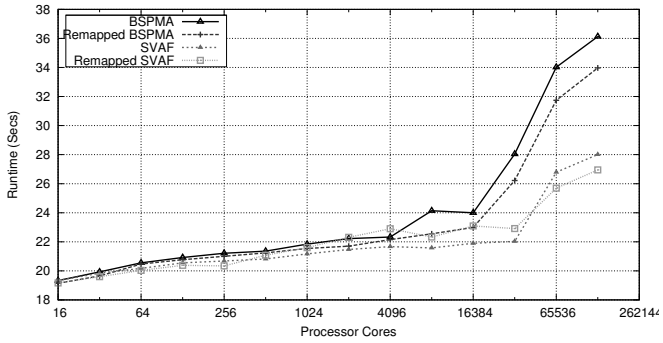


Fig. 7. Performance of miniGhost Communication Strategies on Cielo

on costly global bandwidth by a factor of N, where N is the number of variables aggregated (40 for the shaped charge problem).

VII. Conclusions and Future Work

With the goal of enabling an effective piecewise evolutionary path to making effective use of Exascale computing architectures, we described our explorations of a breadth of issues throughout the codesign space. While faced with uncertain choices of programming models, mechanisms, and perhaps languages designed to run in an uncertain computing environment, we have demonstrated a variety of ways the application developer can begin to concretely and effectively prepare for an unknown specific machine but a widely accepted architectural approach. Although this work is presented in terms of processor, node, and inter-node strategies, we also see that a system-wide design is required to achieve overall reductions in runtime. The common theme is, not unexpectedly, the organization of our data and the way it is moved around and presented to the various components of the architecture. Most reassuring in terms of modifications to large code bases, we have demonstrated an evolutionary path that can also significantly improve performance on current and emerging architectures. Additional information in support of this paper is presented in [5], and deeper studies on some of the topics herein will be presented in papers in preparation.

Our use of miniapps significantly improves our ability to rapidly explore ideas, and these miniapps have been demonstrated to be predictive of full application codes with regard to some key performance issues [4], guiding our focus here. For example, the remapping strategy has proven beneficial to CTH. That said, we reiterate that the output of a miniapp is information that must be interpreted within the context of the full application, and therefore the application developer must apply and probably extend the experiences described in this paper.

We are also studying revolutionary options, including less commonly used and new languages (e.g. [9], [10], [22], [26]). It is also possible that a completely new architecture could emerge from the exascale initiatives. Regardless, it appears that the fundamental concepts for exploiting these architectures will remain: presenting data to the compute engine in a manner that allows it to operate on the data in a vectorized multi-threaded fashion, sharing that data with the parallel processes in efficient ways and exposing sufficient parallelism to effectively hide ever-increasing relative latencies. The lesson learned from our incremental evolutionary approach will not only help applications in the near to medium term, but also set the stage for a smoother transition to revolutionary environments.

Acknowledgements

The breadth of our work has required special efforts from a variety of entities and staff within the Department of Energy and with our industrial collaborators. We acknowledge support from AMD Inc, Cray Inc and NVIDIA for providing detailed information on hardware platforms and information relating to optimization opportunities. The test beds used for this research are funded by the Department of Energy's NNSA ASC program and the Office of Science Advanced Scientific Computing Research (ASCR) program.

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.

REFERENCES

[1] TheOpenACC Application Programming Interface, Version 1.0, November 2011.

[2] D.A. Bader, V. Kanade, and K. Madduri. SWARM: A Parallel Programming Framework for Multi-Core Processors. In *First Workshop on Multithreaded Architectures and Applications (MTAPP)*, March 2007.

[3] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho. Entering the Petaflop Era: the Architecture and Performance of Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 1:1–1:11, Piscataway, NJ, USA, 2008. IEEE Press.

[4] R.F. Barrett, P.S. Crozier, S.D. Hammond, M.A. Heroux, P.T. Lin, T.G. Trucano, and C.T. Vaughan. Assessing the Validity of the Role of Mini-Applications in Predicting Key Performance Characteristics of Scientific and Engineering Applications. Technical Report SAND2013-TBD, Sandia National Laboratories, 2013. In preparation.

[5] R.F. Barrett, S.D. Hammond, C.T. Vaughan, D.W. Doerfler, M.A. Heroux, J.P. Luitjens, and D. Roweth. Navigating An Evolutionary Fast Path to Exascale. Technical Report SAND 2012-4667, Sandia National Laboratories, 2012. http://www.sandia.gov/~rfbarre/pubs_list.html.

[6] R.F. Barrett, C.T. Vaughan, and M.A. Heroux. MiniGhost: A Miniapp for Exploring Boundary Exchange Strategies Using Stencil Computations in Scientific Parallel Computing. Technical Report SAND2011-5294832, Sandia National Laboratories, May 2011.

[7] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.

[8] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. Graph.*, 22(3):917–924, July 2003.

[9] B.L. Chamberlain, D.Callahan, and H.P. Zima. Parallel programming and the Chapel language. *International Journal on High Performance Computer Applications*, 21(3):291–312, 2007.

[10] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA)*, October 2005.

[11] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46 –55, 1998.

[12] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Auto-Tuning on State-of-the-Art Multicore Architectures. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*, pages 1 –12, November 2008.

[13] R.E. Diaconescu and H.P. Zima. An Approach to Data Distribution in Chapel. *International Journal on High Performance Computer Applications*, 21(3), 2007.

[14] G.R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. ParalleX: A Study of A New Parallel Computation Model. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2007*, March 2007.

[15] A. Geist and S. Dosanjh. IESP Exascale Challenge: Co-Design of Architectures and Algorithms. *Int. J. High Perform. Comput. Appl.*, 23:401–402, November 2009.

[16] Khronos OpenCL Working Group. OpenCL specification, v1.0.29. http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

[17] M.A. Heroux, D.W. Doerfler, P.S. Crozier, J.M. Willenbring, H.C. Edwards, A. Williams, M. Rajan, E.R. Keiter, H.K. Thornquist, and R.W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, September 2009.

[18] E.S. Hertel, Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. Mcglaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *Proceedings, 19th International Symposium on Shock Waves*, pages 377–382, 1993.

[19] S.M. Kelly et al. Report of Experiments and Evidence for ASC L2 Milestone 4467 - Demonstration of a Legacy Application's Path to Exascale. Technical Report SAND2012-1750, Sandia National Laboratories, 2012.

[20] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: an Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[21] S.A. McKee. Reflections on the Memory Wall. In *First Conference on Computing Frontiers*, 2004.

[22] R.W. Numrich and J.K. Reid. Co-Array Fortran for Parallel Programming. *ACM Fortran Forum*, 17(2):1–31, 1998.

[23] NVIDIA Corporation. CUDA programming guide. http://www.nvidia.com/object/cuda_home.html.

[24] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[25] H. Shan, N.J. Wright, J. Shalf, K. Yelick, M. Wagner, and N. Wichmann. A Preliminary Evaluation of the Hardware Acceleration of the Cray Gemini Interconnect for PGAS languages and comparison with MPI. In *Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems*, PMBS '11, pages 13–14, New York, NY, USA, 2011. ACM.

[26] UPC. Consortium, UPC Language Specification. May 31 2005.