J Supercomput (2012) 62:1023–1044
DOI 10.1007/s11227-012-0789-3

Paradigmatic shifts for exascale supercomputing

Neal E. Davis · Robert W. Robey · Charles R. Ferenbaugh · David Nicholaeff · Dennis P. Trujillo

Published online: 9 June 2012
© Springer Science+Business Media, LLC (Outside the USA) 2012

Abstract As the next generation of supercomputers reaches the exascale, the dominant design parameter governing performance will shift from hardware to software. Intelligent usage of memory access, vectorization, and intranode threading will become critical to the performance of scientific applications and numerical calculations on exascale supercomputers. Although challenges remain in effectively programming the heterogeneous devices likely to be utilized in future supercomputers, new lan-

N.E. Davis · D. Nicholaeff · D.P. Trujillo
XCP-4 Methods & Algorithms, Los Alamos National Laboratory, Los Alamos, NM, USA

D. Nicholaeff
e-mail: dnic@lanl.gov

D.P. Trujillo
e-mail: dptru10@nmsu.edu

N.E. Davis (✉)
Department of Nuclear, Plasma, & Radiological Engineering, University of Illinois at Urbana–Champaign, Urbana, IL, USA
e-mail: davis68@illinois.edu

R.W. Robey
XCP-2 Eulerian Applications, Los Alamos National Laboratory, Los Alamos, NM, USA
e-mail: brobey@lanl.gov

C.R. Ferenbaugh
HPC-1 Scientific Software Engineering, Los Alamos National Laboratory, Los Alamos, NM, USA
e-mail: cferenba@lanl.gov

D. Nicholaeff
Department of Physics & Astronomy, University of California at Los Angeles, Los Angeles, CA, USA

D.P. Trujillo
Department of Physics, New Mexico State University, Las Cruces, NM, USA

guages and tools are providing a pathway for application developers to tackle this new frontier. These languages include open programming standards such as OpenCL and OpenACC, as well as widely-adopted languages such as CUDA; also of importance are high-quality libraries such as CUDPP and Thrust. This article surveys a purposely diverse set of proof-of-concept applications developed at Los Alamos National Laboratory. We find that the capability level of the accelerator computing hardware and languages has moved beyond the regular grid finite difference calculations and molecular dynamics codes. More advanced applications requiring dynamic memory allocation, such as cell-based adaptive mesh refinement, can now be addressed—and with more effort even unstructured mesh codes can be moved to the GPU.

Keywords Exascale computing · Heterogeneous architecture · GPGPU · Cell-based adaptive mesh refinement · Molecular dynamics

1 Introduction

Having passed the petaflop marker, the next major milestone in supercomputing is the design and construction of an exascale machine, following the 2008 Department of Energy (DOE) study [19] definition of an exascale machine as one which has "one or more key attributes [(functional performance, physical attributes, and application performance) with] 1,000 times the value of what an attribute of a 'petascale' system of 2010 [would have]." The United States DOE has set this as a target for American research and development to achieve by 2018 [28], and other authors have acknowledged the necessity and challenge of achieving this level of supercomputer performance (Snir [29]; Dongarra [10]).

The thorough 2008 DOE study [19] identifies four major challenges on the path to exascale: energy and power; memory and storage; concurrency and locality; and resiliency. From this and other studies, it is apparent that current trends in power consumption and memory hierarchy management, if extrapolated directly, will be inadequate for the architecture and operation of such a machine. The demand for increased processing power elicits a comparable demand for decreased power consumption per operation: for the exascale computing paradigm, at least a hundredfold reduction of power consumption will be required. As the primary power demand comes from the memory hierarchy and associated memory transfers, graphics processing unit (GPU)-based computing can hide intranode memory latency with increased throughput, and thus provides one of the most promising paths to exascale computing.

However, introduction of a much larger number of compute nodes meeting memory bandwidth and power consumption requirements, as well as the wide-scale scientific deployment of heterogeneous architectures, will not lead to exascale supercomputing in and of itself. That achievement will require advances in the software paradigm governing the production of scientific program codes and require that parallelism in control and data be exploited at all possible levels. In this article, we present the backdrop of the supercomputing development to give context to a discussion of exascale challenges. We then focus on the software changes required to implement a GPGPU-based paradigm as first steps toward an exascale heterogenenous programming paradigm across a wide range of applications in test problems developed at the DOE's Los Alamos National Laboratory (LANL).

2 Historical supercomputing paradigms

There has been a practical, if not theoretical, gulf between early vectorized or massively-multiprocessor (MMP) machines (supercomputers) and more modest thread-based approaches (microcomputers). Much of the early literature ignored this distinction (as true threading on microcomputers was virtually nonexistent until multiple central processing units (CPUs) became available), and so parallel computing as a discipline developed in many directions largely separate from contemporary microcomputer hardware. The basic approaches possible may be partitioned on their execution and memory models, which govern how independent portions of the process (if any) must interact.

2.1 Vector approach

One of the earliest methods for optimizing machines for large-scale scientific computing was vectorization of memory accesses and operations. Vectorization was introduced in the 1960s (Westinghouse's Solomon; University of Illinois's ILLIAC IV) and rose to prominence in 1976 with the Cray-1, which utilized vector operations to achieve a peak performance of 80 MFLOPS per chain (Mills and Wood [22]; Oyanagi [24]). Vector operations are a major contribution to microcomputer data-level parallelization and are present on most modern CPUs, an approach classified as single instruction, multiple data (SIMD).

With the introduction of the Pentium microprocessor architecture by Intel in 1995, microcomputer CPUs began to support MMX SIMD vector operations, which soon evolved into the short Streaming SIMD Extensions (SSE) instruction set extension and its successors and competitors. SSE provides operations for 128-bit wide data types, allowing for vector operations on 1–16 numbers or characters. These parallel operations allow a significant decrease in run-time when implemented correctly; in some cases the speed increase is analogous to an upgrade from a single-core to a quad-core CPU (Klimovitski [18]). The ongoing development and implementation of vector extensions reflect the continued relevance of vectorizable linear algebraic operations to both the scientific computing community and general-purpose application developers. Combined with the graphics processing unit (GPU) discussed below, these operations constitute the modern legacy of vectorization.

2.2 Massively-multiprocessor approach

Due to a combination of restrictions on bandwidth and growing demand for larger calculations, in the late 1980s supercomputer vendors began shifting to a distributed-processing, distributed-memory model known as multiple instruction, multiple data (MIMD), which achieved absolute dominance in supercomputing by the 1990s due to phenomenal performance increases (Boillat et al. [3]). Although MIMD had been around in some form since the 1960s (Casaglia and Olivetti [6]), there was still, by 1991, a demand for then-modern operating systems and programming paradigms suitable for MIMD supercomputing. As the paradigm developed, multiple communicating copies of a program are executed simultaneously across many compute nodes,

as in the Message Passing Interface (MPI) standard (Message Passing Interface Forum [21]; Gropp et al. [13]). The design goal of MPI is to provide a portable programming environment with internode communications between processes. (However, nothing is said about the structure of the node or the types of operations which may be performed on it. We will return to this point below.)

The internode-communicating MIMD paradigm has governed supercomputing thought from the late 1980s through the early 2010s, and, despite the challenge of programming for a distributed environment, it has proven adequate for scientific computing and numerical applications on systems as varied as small heterogeneous research clusters and large-scale supercomputing facilities. Internode communication schemes will surely continue to be an integral part of future approaches to supercomputing, regardless of the changes in architecture.

2.3 Local threading approach

With the introduction in 1963 of J. Lyons and Co.'s LEO III, a multitasking operating system became a reality. Batch programs stored in memory were switched out as peripheral switches were reached, allowing the "concurrent" serial execution of several logical command queues. Later single-processor machines continued to use versions of this approach, including in-program interrupts and forced context-switching by the operating system. This approach has been termed "threading", although it does not allow for truly parallel execution of the threads. True parallel execution required the introduction of multiple processing units, the path followed by supercomputers. Thus, earlier approaches used threading as a paradigm for context switching or pipelining in multiplexing operating systems, while true threading allowed for completely independent execution strands.

True threading first became accessible to commercial microcomputer programmers with the 2001 introduction by IBM of the POWER4 microprocessor (Tendler et al. [32]). As multicore microcomputers became available, new or existing shared-memory models such as POSIX Threads (1995) and OpenMP (1997) were developed to exploit the natural parallelism in many common applications, such as matrix multiplication or image processing.

Thread-based microcomputing can be effective for data-parallel or command-parallel processes. For many applications, however, data-parallelism is easier to exploit, and thus more often encountered in practice. Hardware features such as the GPU extend the capacity of the contemporary multicore microcomputer to execute parallel programs. The GPU utilizes SIMD processing to achieve speedup on data-parallel linear algebraic operations (such as vector or matrix operations) relevant to graphics processing. Utilization of the GPU can decrease run-time for certain types of calculations due to the high number of arithmetic logic units (ALUs) and the impressive memory bandwidth (up to 40 Gb/s) available on-chip. Separate execution threads can be launched at each ALU with near zero context switching, limited in principle only by the amount of available memory.

Languages such as Brook were introduced to exploit this data-level parallelism, essentially casting common numerical methods as graphics routines which were then executed on the GPU. The manufacturers of GPUs, among others, soon realized the

potential for nongraphics programming, and general-purpose GPU (GPGPU) computing was born. Numerically heavy data-parallel applications can be executed on the GPU while concurrent execution of order-dependent code is performed on the CPU. Today NVIDIA's CUDA language and the industry standard OpenCL represent the state-of-the-art in GPGPU languages.

Of course, many other approaches have been explored academically; our intent in this section has been to examine those which were commercially implemented and have become influential paradigms for the next generation of supercomputers.

3 Motivations for a new paradigm

The Department of Energy has recently published target specifications [17] for the next generation of supercomputers, the first of which will hopefully become available around 2018. The stated targets include exaflop computational power (10^{18} floating-point operations per second, FLOPS), energy efficiency, and improved hardware performance and design, all of which are undoubtedly necessary to exascale supercomputing. However, the coordination of heterogeneous processing elements will present new challenges for typical programmers designing scientific and numerical applications for an exascale machine.

Indeed, some recent observers (Young [35]) have suggested that the primary benchmark of computational prowess is rapidly becoming software, not hardware—if a state-of-the-art supercomputer is not running appropriate and well-designed software, scaling computational power alone will no longer be sufficient to solve some of the complex problems now contemplated by researchers. The driving factor for new supercomputers will be primarily (but not exclusively) software-based, rather than hardware-based, meaning that the users rather than the vendors will dominate the performance characteristics of an exascale machine. Although clearly the statutory number of FLOPS available will continue to grow, exploitation of the available computational power will require better algorithm design and problem decomposition into data-parallel and command-parallel units. Furthermore, much of the control of the memory hierarchy must transfer to the application or library developer as automatic hardware techniques such as cache management are not efficient enough alone in the context of HPC GPGPU computing. Thus, software will be challenged to change more in response to this paradigm shift than at any previous time in the high-performance computing progression.

We are going to first look at the drivers for exascale hardware in the next sections. These include the issues of the so-called power wall and the memory wall. This provides the needed context for the direction of the software development. Then we will return to the subject of the parallelization approach for the next generation hardware. We expect this hardware to evolve to be some combination of the current GPU hardware and the soon-to-be released Intel Many Integrated Core (MIC) processor.

3.1 Power consumption

Computational efficiency for the most efficient supercomputers has increased by an order of magnitude in the past 4 years (see Fig. 1). Nevertheless, the historical hardware approach will be inadequate for future machines: the power consumption per

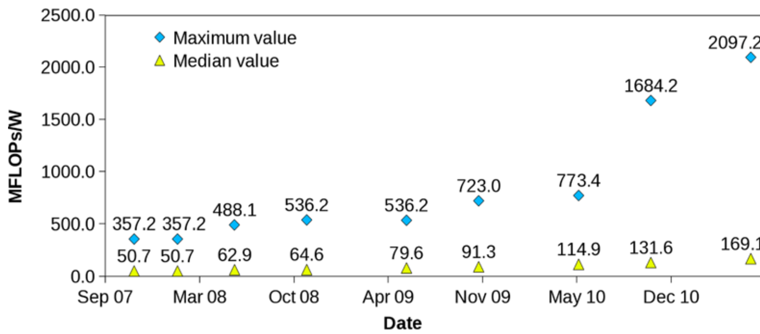

Fig. 1 Computations per power consumption for the 500 most efficient supercomputers, November 2007 through June 2011. From data available at http://www.green500.org/, Feng and Cameron [11]

FLOP of a supercomputer has risen steadily as error-correction, cache-fetching, and out-of-order execution methods have become standard on CPU-based compute nodes ([19], p. 202). Naïve scaling of the power requirements for a petaflop machine to an exaflop machine indicate that more than a gigawatt will be required for execution, along with the concomitant cooling demand. Contrast this with the DOE-specified practical power limit for an exascale data center of 20 MW as layed out in the DOE study led by Khaleel [17] and separately in Feng [11].

Power-intensive operations such as out-of-order execution methods and multiple-level cache hierarchies will also need to be jettisoned to some extent for an exascale machine, even as features such as error-correcting codes become more common. (Without error-correcting codes, it is possible that simulations would have to be run multiple times in order to deal with nondeterministic results, ultimately perhaps increasing the net power usage.) The hardware demand for devices with low power consumption will tend to favor heterogeneous supercomputing environments, such as using GPUs on compute nodes to increase data-parallel operations, and thus decrease the total demand on more power-intensive portions of the machine if possible.

3.2 Memory

As memory will not scale proportionally to processing power, more intelligent memory management will also be critical to successful software performance. Explicit cache management (by the application programmer or library developer), allowing in-cache accesses to be optimized by utilizing organized local data without pushing data out of the cache by a call to global RAM, will become critical as machines continue to grow in size and complexity, both in order to reduce the relative power consumption of the hierarchical cache system common on microcomputers today and to decrease the latency of the processor due to memory bandwidth. In addition, the amount of memory available on an exascale machine is not projected to be one thousand times that available on a petaflop machine, so memory structure economy will be necessary. (Additional challenges to memory power consumption, bandwidth, and scaling are discussed in the 2008 DOE study led by Kogge [19], pp. 113–122, 221–223).

Beyond memory bandwidth issues, internode communications are expensive, and any parallelization which can be performed on the compute node is therefore valuable. Bowers et al. [5], suggested that, in terms of processor cycles for LANL's Roadrunner supercomputer, an internode communication was two thousand times as expensive as moving memory between the CPU and the GPU. Clearly, intranode communications are preferable when possible for an algorithm.

In summary, the internode communications approach which has dominated supercomputing for about three decades will continue to be relevant, but alone it will be inadequate to meet the increasing level of sophistication which will be exacted of software. The power consumption per calculation should also greatly decrease to reach target specifications. Explicit memory management and reduction of reliance on certain power-intensive hardware operations will help with this process, and compute node optimizations which minimize internode data transfer will be required. Ultimately, however, a new approach is necessary which synthesizes these considerations with the broader lessons of parallel computing.

4 Parallelization for tomorrow

It is clear that the massively-multiprocessor (MMP) approach is a macrocosm of the multicore microcomputer approach—load division for many processors on a supercomputer is directly analogous to application threading on a multicore microcomputer. In MMP computing, data units are broken into several groups, one for each processor, which are simultaneously handled serially at each compute node by a conventional CPU-type processor. The lack of machine-global (not node-global) shared memory for an MMP supercomputer will continue to dominate internode interactions, although the hardware composition of a local node has been reexamined in recent years to encourage utilization of threading and vector operations on a single node. For example, consider the TianHe-1A supercomputer at the Chinese National University of Defense Technology, a machine with 7,168 NVIDIA Tesla M2050 GPGPUs in conjunction with 14,336 Intel Xeon X5670 CPUs and 2,048 FT-1000 CPUs which utilizes a unique parallel programming infrastructure to encourage users to develop applications which exploit local as well as distributed parallelism using OpenCL and CUDA (Yang et al. [34]).

The next generation of supercomputers targeting sustained exascale operation will clearly be based on hardware which has relatively low power consumption yet can handle sophisticated parallelism and execution: CPU compute nodes coupled with some sort of accelerators, such as GPUs, Intel MICs, or a similar not-yet-announced technology. (The GPU has certain attractive characteristics, but given its relatively low cache memory in comparison to the CPU and the ability to scale only data-parallel algorithms, the CPU continues to provide the best compute node platform for memory- or execution order-reliant sections of code.) This model cleanly unites the three major paradigms of parallel computing: vectorization, massively multiprocessor parallelization, and process threading. Extension of problem decomposition to its logical extreme suggests a scenario in which intranode communications and

cache management are managed explicitly by OpenCL or another GPU/heterogeneous device programming language and internode communications are handled by a message-passing interface such as MPI.

While we argue for a scenario in which GPUs are either the primary compute nodes or are tightly coupled with conventional CPUs at the compute nodes, the advantage of an open standard, heterogeneous device-based approach is that it allows any type of device at the compute node to be exploited without major changes to the compute kernel. For instance, if IBM's Blue Gene architecture, FPGAs, or an as-yet-unproposed exotic chip architecture becomes the dominant hardware for supercomputers instead of CPUs and GPUs, no obstacle arises to utilizing OpenCL as the primary language for interacting with the device hardware.

4.1 Software requirements

Perhaps the biggest change which users of supercomputers with heterogeneous or GPU-based architectures will perceive is that the effective utilization of such a machine will make new demands on authors and maintainers of scientific applications and numerical software. Fortunately, open industry standards such as OpenCL should decrease the amount of device-specific work required to within reasonable limits (e.g., querying the hardware to check for double-precision floating-point variable support). Vendor-specific languages such as CUDA C can gain portability to other hardware through the emergence of tools such as Swan, which can convert CUDA C code to OpenCL (Harvey [14]).

The treatment of numerical algorithms will finally have to unify the three logical strands of parallelization discussed above. An algorithm will be required to be able to be decomposed into logical units on at least two levels: one for the node level (utilizing internode communications schemes) and one for the intranode level (based on data-parallel threading and vector operations). Parallelization of an algorithm at multiple levels portends respectable increases in computing power at a relatively modest cost in electrical power. In connection with this, Dongarra [10] suggests that the driving model for high-performance software design could become data parallelism instead of control parallelism.

The logical combination of context-switching threading and parallel execution threading opens up the possibility that, within a node at least, the number of threads is not limited to the physical number of processors. Unshackling the number of execution threads on a node from the number of physical processors means that the problem can be decomposed intranode at the lowest possible data-parallel level. (It is not apparent that this argument extends to the nodes as in MPI there is no master coordination of execution and memory management without privileging a specific node.)

For large machines using MPI, recall that internode communication is an extremely cycle-intensive operation (Bowers et al. [5]). In short, software development and algorithm choice will be driven by considerations of multilevel data parallelism and minimization both of memory transfer to and from the GPU device and of internode communications, requiring creative domain decompositions and numerical methods to be discovered and implemented.

Some improvements in the MPI layers and GPU scheduling are also needed to improve performance. Interesting research in this area includes topology aware process placement for MPI (Bhatele [2]) which is being tested on our applications. Process placement leverages the work of processor affinity (Yuan [36]) that is already in OpenMPI and has been in use at LANL for a couple of years with typically a 5–10 % boost in performance and occasionally more. Research on GPU scheduling (Kato [15, 16]) demonstrates better control over the management of work in the GPU queue and will be crucial to enforcing priorities to GPU tasks and avoiding conflicts.

4.2 Code complexity

Many scientists and engineers who write parallel codes are not hardware programming specialists. For a GPU-parallel and ultimately a hybrid GPU/MPI-parallel approach to be palatable to them, the programmability and code complexity should not increase significantly just because an exascale computer is being used. A thousand-fold increase in processing power should not require a concomitant thousand-fold increase in the number of lines of code to exploit! A modest increase in the number of lines of code necessary to use a GPU or other device on a node will no doubt be necessary. We demonstrate below in the applications section that much of the cost incurred is due to the device and kernel setup and termination and thus fixed rather than proportional to the calculation size or number of threads or nodes necessary.

The OpenCL 1.1 specification is still an early standard and many of the hardware drivers are not fully developed. A runtime-level application programming interface (API), such as that available in CUDA, is also desirable and would further reduce the requisite code complexity. Array bounds checking or other memory protection would also ease implementation difficulties and help isolate errors.

On any GPGPU device, some additional setup will be required for algorithms; for instance, an intranode reduction must be performed prior to a global reduction operation. This is analogous to the lines-of-code overhead incurred by using MPI or a similar system for internode communications which is already familiar to users of HPC machines.

5 Numerical applications with GPGPU

Although some authors have examined the potential of heterogeneous devices such as IBM's Cell BE, IBM's Blue Gene, or the GPU for certain types of numerical calculations, even large-scale applications (Bowers et al. [4, 5]; Papadrakakis, Stavroulakis, and Karatarakis [26]; Wolfe [33]), there will be a demand for demonstration projects integrating multiple CPUs and GPUs with optimized numerical applications in an HPC environment. Accordingly, we highlight four applications developed at Los Alamos National Laboratory which utilize several standard features of scientific applications which are not currently well-supported or amenable to straightforward implementation on the GPU. These codes serve as proofs-of-concept that existing numerical algorithms can be successfully implemented on the GPU without extensive restructuring of the basic logic, and are conducted with an eye toward demonstrating

the types of adaptations of existing codes and algorithms which will be necessary for heterogeneous-device-based high-performance computing. We adopt as a metric of complexity the number of lines of code, which although unsatisfactory in some respects, approximately reflects the required effort spent in programming. All operating statistics have been gathered on the Darwin cluster operated by the Computer & Computational Sciences (CCS) division at Los Alamos National Laboratory using the AMD Opteron Processor 6168 for the CPU with the NVIDIA GF100 Tesla C2050 for the GPU.

5.1 Regular grid in OpenCL

The first application ported to OpenCL is a two-dimensional code implementing the shallow water equations on a regular discretized grid. The regular grid data structures are easier to adapt to the GPU data model (which is optimized for rectangular domains as often encountered in computer graphics applications) and in this application, the data is maintained on the GPU and only brought back for visualization. The finite difference stencil operations map to the GPU in a straightforward way and the maximum operation in the time-step calculation is done with a complex GPU reduction algorithm.

The algorithm has the three state variables: height H and momentum in the x and y directions U and V, representing the values of the state variables at the cell centers. A Lax–Wendroff predictor-corrector technique is used to interpolate the values a half-step in space and time. These predictor values are then used to estimate the new values at cell centers for the next time-step. A total variation diminishing (TVD) corrector term is used to damp out second-order oscillations in the solution. The method is similar to the centered TVD scheme described by Davis [8].

The OpenCL parallel code is very similar to the C code; it essentially is the main program loop with the two `for` loops over the mesh elements removed as shown in Listing 1. The state variables are copied to a local 16×8 memory tile which acts like a programmable cache with fast access times. Minor restructuring of the code was done to get some additional speedup. This restructuring consisted of changing the order of operations to reduce GPU memory bank conflicts and was done by tracking performance while code streamlining was done and keeping the fastest version. Recent GPU hardware has reduced the penalities for memory bank conflicts, so this type of work should become less important. Throughout this process, the code remains recognizable as the implementation of the original numerical equations.

The original C/MPI code consists of about two thousand lines including the graphics routine. After being ported to OpenCL, the code has a 75 % increase in the line count. But this is almost entirely accounted for in a support library, `ezcl`, written to handle the OpenCL calls. The `ezcl` library is designed to handle administration of the compute device, including initialization of the compute device and checking the error codes returned from the OpenCL calls. This demonstrates what we conjecture will happen with most application ports: the straightforward port will increase the line count by 50–75 %, but when a support library is developed with a higher level of abstraction, the increase in line count will become much more modest. After subtracting out the number of lines in the `ezcl` library, the GPU version line count is

Listing 1 Removing the `for` loops and using the thread identifiers in OpenCL

```
//  C Wave code
    for (j = 0; j < mysize; j++) {
      for (i = 0; i<=imax;i++) {
        //density calculation
        Hx[j][i]=0.5*(H[j+2][i+2]+H[j+2][i+1]) - Cxhalf*( HXFLUX(j,i)-
            HXFLUX(j,i-1) );
        ...
      }
    }

//  OpenCL Wave kernel
   const uint tiX  = get_local_id(0);
   const uint tiY  = get_local_id(1);
   const uint ntX  = get_local_size(0);
   const uint stridetY = ntX+4;
   const uint ic  = (tiX+2)+(tiY+2)*stridetY; // Current tile index
   const uint il  = ic-1;        // Index one cell  to the left
   const uint ir  = ic+1;        // Index one cell  to the right

   // Code for copy from H_array to H_val tile omitted

   HxPanel(0) = HALF*(Hval(ic) + Hval(il)) - Cxhalf * ( HXFLUX(ic) -
       HXFLUX(il) );
    ...
```

Table 1 Average time spent in seconds on each function for a regular grid code with a mesh of 1280 × 1280 cells for 10,000 time-steps

Function	1 CPU	16 CPU	48 CPU	144 CPU	CPU + GPU
Finite difference method	8105	539	175	56	127
Reduction	428	156	91	74	0.01
Write to device	0	0	0	0	0
Read to device	0	0	0	0	4.6
Total	8533	692	266	130	131

almost the same as the C/MPI version. We infer from this that the increase in code complexity for the GPU version is about the same order of magnitude as for the MPI parallelization (which was removed in the GPU implementation).

As a payoff for the increase in code complexity, there is a 45× performance improvement of the GPU as shown in Table 1. There are small differences in the GPU and CPU versions that may affect some of the detailed speedups. The relative level of optimization is fairly standard for both the CPU and GPU codes. More significant factors, such as both versions utilizing double precision, make the comparison, nevertheless, valid. Also shown in Table 1, the MPI version requires 144 processors to equal the speed of 1 GPU node in the GPU version.

Two-dimensional and three-dimensional computational fluid dynamics (CFD) kernels were also written and the results were similar but with a smaller speedup for the 3D kernel.

As a foray into a higher-productivity programming level, a Domain Specific Language (DSL), ForOpenCL, is being developed by Sottile [30]. DSLs are usually composed of part library and part compiler. ForOpenCL targets regular grid and adaptive mesh refinement (AMR) types of problems, similar to how the Liszt DSL targets Eulerian unstructured grid problems. (Liszt is discussed in the unstructured grid application section.) The ForOpenCL DSL utilizes the Rose open source compiler infrastructure and parts of the hand-written support code for a library. A Fortran kernel was developed for the shallow water equations and the ForOpenCL DSL generated the OpenCL code for the shallow water kernel with the goal of trying to match the hand-written kernel performance. The results are encouraging enough (Sottile et al. [30]; Sottile et al. [31]) that we are now attempting a similar effort to match the adaptive mesh refinement (AMR) discussed in the next section. The productivity savings for the regular grid are modest given that the OpenCL kernel is relatively easy to write. But a big advantage for using the DSL for regular grids is when we are generating multiple hardware specific codes such as an OpenCL kernel, a CUDA kernel, an MPI kernel and a hybrid implementation using a GPU language and MPI. For the AMR code, the greater complexity will mean much larger productivity gains for a DSL, but also more development challenges.

The results with the regular grid problems are encouraging, but most numerical applications now use some sort of dynamic-memory-based simulation such as adaptive mesh refinement (AMR) or an unstructured mesh. How do these types of applications fare on the current GPU architecture? The next category of applications examines this question.

5.2 Adaptive mesh refinement in OpenCL

CLAMR is a cell-based AMR code modeling the shallow water equations. A classic AMR program relies heavily on features of modern programming languages such as dynamic memory allocation and either a globally-available memory addressing space (on a microcomputer) or a distributed communicating message-passing interface (on a supercomputer). Neither dynamic memory allocation nor inter-thread communications are currently available on a GPU, and so the development of a numerical AMR code for the GPU requires methods for implementing or working around the lack of such common features as dynamic memory allocation.

The basic data-parallel unit for CLAMR is a cell, consisting of state variables and position information. Thus each cell's calculations can be launched as a separate thread in the GPU. Additionally, initialization of the mesh, calculation of the time-step, evolution of the state variables, and mesh refinement dictated by gradient calculations, are all performed natively on the GPU. Control flow is returned back to the CPU only to reallocate GPU global memory as all state information is transferred to the updated refined mesh. Neighbor calculations are resident on the GPU as well. The full architecture of CLAMR is discussed in more detail in Nicholaeff et al. [23].

For the regular grid code, the addition of a global reduction call and a basic iterative loop did not significantly impact the number of lines of code. In contrast, there is a nontrivial amount of code necessary for the dynamic memory management found in CLAMR as shown in Table 2. For the serial AMR code from which the parallel version was adapted, around 1500 lines of C/C++ code in the main computational loop

Table 2 Lines of code in each function in CLAMR. (For GPU, lines are shown for host and device as "host + device" lines. Adjustment made for comment lines

Function	CPU	CPU + GPU	MPI	MPI + GPU
Calculate time-step	29	66 + 94	31	68 + 94
Main calculation	545	65 + 758	551	133 + 795
Refine potential	218	106 + 530	226	126 + 530
Mesh refinement	446	117 + 384	508	161 + 384
Calculate neighbors	206	68 + 123	508	628 + 420
Mass sum	34	70 + 117	42	72 + 117
Subtotals	1478	492 + 2006	1866	1188 + 2340
Total	1478	2498	1866	3528

were necessary for the implementation. The MPI version of CLAMR has about 33 % more lines of code, not including our `L7` sparse communication package. The GPU version of CLAMR requires about 66 % more lines of code in the main computational loop. The lines in our GPU support library `ezcl` that perform the device initialization and error checking are not included in this count. Based on this metric, the level of effort for the GPU implementation is roughly twice the effort for the MPI implementation. Per kernel, roughly thirty lines of code are required to set up the on-chip memory arrays, execute the kernel, and read the data off after the kernel completes. The 'subroutine call' part of the host code for the GPU where every argument in the call list must be set with an OpenCL call is responsible for a large part of the increase and is the most error prone part of writing OpenCL code as the arguments must be manually matched between the host code and the kernel code in separate files. Improvements in the OpenCL standard in this area could reduce the effort of writing an OpenCL implementation. It should be noted that a better subroutine call interface is available in CUDA with the 'triple chevron' syntax. Implementing both MPI and GPGPU in the code requires a 140 % increase in lines of code and more than the increase due to MPI and GPGPU alone. This is because when doing the complex MPI sparse communication, the data must first be retrieved from the GPU, the communication scheduled, and then the data sent back to the GPU. The difficult section of the code is primarily in the calculate neighbors section of the code where the ghost cells for the mesh are established. Each step in the hybrid parallelization implementation is of reasonable effort, but in total, the effort is substantial and adequate manpower and expertise must be planned to be successful with such an effort. Some reduction in the difficulty should occur with the maturing of the GPU programming languages, availability of libraries, and familiarity among programmers.

A key breakthrough in parallel algorithms was made to enable the full GPU port of the CLAMR code. Neighbor calculation was originally done using a k-D tree algorithm. Tree-based algorithms are challenging to port to a parallel thread model and would take several months. Instead, we developed an new algorithm based on a hash. Hash algorithms are $O(n)$ instead of $O(n \log n)$ for the k-D tree. The difficulty is deciding what key to use. A key that would result in a perfect hash is desirable so that a collision handler is not needed. It was discovered that using the finest level of

Table 3 Average time spent on each function in CLAMR in seconds for an initial 450 × 450 coarse mesh cells with one level of refinement run for 5000 cycles. Not all time is listed so the columns do not add up to the total time. The scalability of this problem is considered in Nicholaeff et al. [23].

Function	CPU	CPU + GPU	Speed-up
Calculate time-step	57.8	0.86	67.2×
Main calculation	1694.9	31.6	53.6×
Refine potential	102.3	4.00	25.6×
Mesh refinement	64.6	1.36	47.5×
Calculate neighbors	145.5	1.68	86.6×
Mass sum	13.0	0.64	20.3×
Write to device	0.0	0.01	
Read to device	0.0	0.07	
Total	2087.7	40.27	51.8×

the mesh would result in a minimum hash size for a perfect hash. The algorithm has each cell write its index to all the cells that it contains at the finest level. Each cell than queries its neighboring cell locations and reads the index off of the hash. This algorithm turns out to be 350× than the k-D tree on the CPU. Because it is inherently parallel, the algorithm was ported to the GPU in a single day for a total speedup of 22,000× (Nicholaeff, et al. [23]). Since then, we have demonstrated that hash-based algorithms can be used for a wide range of spatial operations, yielding speedups from hundreds to thousands due to the multiplicative effect of the algorithm and the GPU parallelism (Robey et al. [27]).

Algorithmic breakthroughs—such as that just described—will be necessary to exploit the full power of the hardware and the parallelism of the algorithm. For the irregular memory structure in CLAMR, a fast GPU scan is critical so that every thread knows where to store their results independently, thus making the operations parallel. As more numerical libraries (and possibly a runtime-level OpenCL API) become available, the amount of this code that must be written by a scientist or engineer utilizing an application code for research purposes should decrease dramatically.

The hash algorithms also demonstrate portable performance for both ATI and NVIDIA GPU hardware with OpenCL on cutting-edge algorithms using general coding that yields good perfomance on both architectures (Robey et al. [27]). Similar performance portablity has also been demonstrated in a Supercomputing 2010 demonstration code running simultaneously on the IBM Cell processor, IBM Power, nVIDIA Tesla GPUs, AMD Opteron CPUs and Intel Core i7 CPUs (Bergen et al. [1]).

The speedup of the parallel code versus the serial version is outlined in Table 3. All of the main computational loop is done on the GPU. The CPU and GPU code were written in normal physics style with a standard level of compiler optimizations. The dynamic memory requirements are handled by a double buffering technique similar to that used for graphics. This technique uses a new and old state buffer with the new state being reallocated before the mesh refinement and the old afterwards so that the old data is refined into the larger new mesh array. There are also pointer swaps along the way to move data arrays from new to old. Thus intelligent management by the CPU of the GPU memory combined with cache management and thread-based parallelization yielded over a 50× speedup for CLAMR over the serial mesh code.

Timing results for the GPU device were sensitive to the tile size chosen, experimentally varied from 32 to 256 threads, presumably due to excessive memory demands. This necessitates careful consideration whether declaration of additional variables in a compute kernel could be replaced by updating previously existing variables (at the risk of polluting the variable namespace). But this consideration goes a step further, as selection of an optimal tile size lessens the number of required reads to global device memory. Additionally, however, increasing tile size increases memory pressure in the sense that more threads are pushed onto local memory potentially causing a memory overflow. Hence, careful testing is required to find the optimal range for the current kernel. Developing standard libraries which explicitly test kernels with variable tile sizes would therefore be of high value.

As mentioned, reducing the number and size of memory write operations is imperative. Specifically, in the development of the main compute kernel of CLAMR, the write operation of the arrays containing the information of the width and height of the cells was removed from the code. Instead, these lengths were computed on the fly from the level of refinement array (which is passed in anyway for neighbor calculations to be discussed below). Removing those two arrays reduced the need for global GPU memory accesses, resulting in a 25 % speedup of the main compute kernel. The speedup which can be gained by masking memory latency with actual computation is simply too valuable to overlook, as demonstrated here. The time and effort required to devise real-time calculations in algorithms to replace storing values in memory is well worth the work as exascale computing moves to extensive use of GPUs.

Thus far, the techniques mentioned address the shift in thinking required to move away from multiple-level cache hierarchies. Another major consideration involves explicit control flow and avoiding out-of-order execution. In particular, perhaps the greatest source of difficulty encountered by the authors in implementing AMR on a GPU device stems from neighbor-dependent calculations. Updating state variables requires knowledge of the state variables of a neighboring cell (and, in some cases, the neighbor of a neighbor, depending on the relative refinement of a neighboring cell). While significant speedup arises from the implementation of a local tile, boundary cells around the tile continue to require access to the device global memory. Aside from pointing out the slowdown arising from the global memory read operation, there is a significant penalty stemming from control-flow branching. Conditional branching between a global or local read operation should be used sparingly, as threads in a workgroup may execute in lockstep, potentially negating the effective speedup gained from working on the local tile.

The code for CLAMR and Wave, the regular grid code, are available online under the open-source New BSD License at http://www.github.com/losalamos. The `ezcl` and `L7` libraries referred to herein will be available in conjunction with CLAMR.

5.3 Unstructured mesh code in CUDA

To study possible implementations of unstructured meshes on GPUs, a small compute kernel (an artificial viscosity, or AV, computation) from the FLAG multiphysics code has been isolated and implemented in CUDA C on the GPU ([20]). The CPU AV kernel is a strictly serial implementation. (The FLAG code as a whole is typically run

Table 4 Average time spent in seconds on each function for an unstructured mesh test problem with 32,400 cells and about 2,000 time-steps

Function	CPU	CPU + GPU	Speed-up
Compute artificial viscosity	134.52	6.10	22.1×
Other calculations + support code	316.52	317.46	1.0×
Total	451.04	323.56	1.4×

in parallel under MPI using domain decomposition; however, the AV computation on each domain is completely independent of the others, so for purposes of this study it was treated as a serial code.)

The GPU is intended for graphics computations, which generally can be constructed on a regular rectangular grid in two or three dimensions. In contrast, many numerical applications, such as finite element methods, utilize unstructured meshes to better optimize the solution for a particular geometry. Unstructured meshes are particularly challenging to implement on accelerated architectures such as GPUs. A regular grid can easily be divided into chunks such that the cells and points of each chunk are contiguous in memory, or at least fall in a regular pattern. This facilitates the movement of different chunks between GPU and CPU memory and the processing of those chunks in parallel. In an unstructured mesh, however, there is generally no straightforward way to divide the different mesh entities (cells, points, edges, etc.) into independent chunks at the same time. A GPU-based implementation must do some form of extra bookkeeping or data reordering in order to process chunks in parallel and recombine them when the computation is complete.

The kernel used in this study consists of several loops over "sides" (triangular subregions of a cell), and it uses variables defined over points and cells as well as sides. Therefore, the problem is conceptually easy to parallelize, using one thread per side. The complication comes in identifying the cells and the points corresponding to each side, and in moving the needed cell and point data from the CPU to fast memory on the GPU. Three different data movement strategies have been identified and implemented. Timing results suggest that an efficient strategy is to have the GPU scatter point data arrays into side-based arrays at the beginning of each kernel invocation, and to gather the side-based arrays back to their point-based counterparts as each kernel finishes. Implementation details are included in Ferenbaugh [12].

The preferred GPU implementation of the kernel contained about 1,100 lines of CUDA C code, compared to 700 lines of Fortran in the original CPU kernel, an increase of about 55 %. The CUDA compute kernel code on the GPU was very similar to the corresponding CPU code, while much of the additional CUDA host code performed bookkeeping tasks such as initialization, shutdown, memory management, and overlapping of data movement with computation. The rest of the FLAG code (about 550,000 lines) remained unchanged. It is conjectured that if additional CPU kernels had been moved to the GPU, much of the bookkeeping code could have been shared between them and the percentage increase would have been smaller.

Results comparing CPU and GPU versions of the code are shown in Table 4. For this study, only the artificial viscosity kernel was moved to the GPU, so results for the remainder of the code are not given in detail. Also, reads and writes are overlapped with computation, so there are no separate timings for communication. The optimized

GPU kernel ran about 22× faster than the CPU version, yielding roughly a 28 % speedup for the full code. Presumably, if more of the application code could be moved to the GPU, a speedup closer to 22× for the full code would be possible.

Another programming model that has proven useful for some unstructured mesh algorithms is the Liszt DSL [9]. This model allows a code developer to work with general unstructured meshes at a fairly high level of abstraction, while the optimization for specific hardware such as GPUs is done by the Liszt compiler. One of the authors has worked with Liszt in another context and found the abstraction to be helpful and easy to use. Unfortunately, this version of Liszt had some limitations (limited support for moving mesh, no support for subcell entities) which would make it impossible to implement the particular physics algorithms described in this section.

A freestanding mini-app PENNANT, which implements similar physics to the kernel used here, is under development; it will be used to further explore the ideas from this study. We anticipate that the code for PENNANT will be made available under an open-source license at http://www.github.com/losalamos in the near future.

5.4 Molecular dynamics in CUDA

REF-MD is a simple molecular dynamics mini-app developed at LANL for use in testing different GPU implementation strategies. REF-MD was created as an exercise in an exascale programming workshop, so its developers have not had the opportunity to bring it to the maturity level of the other applications described previously. Still, it is illustrative of another type of application that can be effectively implemented on a GPU.

Unlike the previous applications, this problem is defined in terms of particles rather than finite-element meshes. We compute the time evolution of an ensemble of interacting particles, where the movement of each particle is determined by the forces exerted on it by its neighbors within a given cutoff distance. To make the neighbor search more efficient, REF-MD superimposes a regular grid on the problem domain, with cell size chosen such that a neighbor within the cutoff distance of a particle must be in the same grid cell or an adjacent one. Using these neighbor lists, a force is computed between a particle and each of its neighbor particles; these are then summed to provide a total force and acceleration for the particle. A serial CPU version of the REF-MD code and a CUDA GPU version have been completed. An MPI version was attempted but not finished.

In the CUDA GPU version of the code, the entire computation was successfully moved onto the GPU; only initial and final input and output steps remained on the CPU. This allowed for the elimination of (nearly all) data movement between CPU and GPU, in addition to making each computational step more efficient. The most expensive step of the CPU calculation is the force and acceleration update; in this step, each particle's update is independent of all others (full data-level parallelism), so it was particularly efficient to assign one thread per particle. The particle sort step, though much less expensive on the CPU, was more difficult to move to the GPU, since sorting contains many data dependencies and is harder to implement on a highly threaded architecture. Our GPU version used the optimized sorting algorithm from the CUDA Thrust library, which is probably more general than necessary but proved to be sufficiently efficient in this context.

Table 5 Average time spent in seconds on each function for a molecular dynamics code on 256,000 particles for 10 time-steps

Function	CPU	CPU + GPU	Speed-up
Sort particles into cells	1.25	0.0994	12.6×
Generate ghost particles	0.45	0.0313	14.5×
Update position, velocity	0.20	0.0196	10.4×
Update force, acceleration	63.73	0.5561	114.6×
Write to device		0.0075	
Read from device		0.0109	
Total	65.64	0.7249	90.6×

The CPU code contained roughly 2,000 lines of C++ source and header files; of these, about half were for the main computational kernel, while the rest were various kinds of support code (data types, I/O, timing, etc.) The GPU code shared the same support code, but replaced the kernel code with about 700 lines of CUDA C code and C++ interface code. In this case, the comparison of lines of code is somewhat misleading: the CPU code was in fully object-oriented C++, while the GPU code was almost entirely in CUDA C with very few C++ features.

For the parallelized portion of the application, the GPU code showed a performance improvement of 90× over the CPU code (Table 5). This figure is probably somewhat inflated, since the workshop participants did not have the opportunity to fully optimize the CPU code. We estimate based on timing data for some of the alternate versions described below that the CPU code could have been rewritten and made up to 3× faster. This would be done by removing some inefficient uses of the C++ Standard Template Library (e.g., "map" searches inside of deeply-nested loops) and replacing them with better-performing alternatives. Even so, the performance improvement on the GPU would have been a respectable 30×. This is not surprising since the highly parallel nature of the force calculations, mentioned earlier, makes the REF-MD code a particularly good candidate for GPU optimization.

The REF-MD code was also used for exploration of other GPU programming models. These versions were partial implementations which moved only some of the functions to the GPU, unlike the CUDA version described above which performed the entire computation on the GPU. So, it is inappropriate to give a detailed comparison between the alternate versions and the CUDA version. We note, however, that an OpenCL version showed similar performance to an earlier CUDA version with a comparable set of functions running on the GPU. In addition, two directive-based programming models were tested: HMPP, and a directive set from PGI. (The PGI directive set is a predecessor to the new OpenACC standard, which was not yet defined when this work was done.) Run times for these were within a factor of about 1.5–2.5 of a comparable CUDA version. From a code development standpoint, the directive-based models were conceptually much easier to work with than OpenCL or CUDA. At the time, both directive-based models had some limitations and bugs to work around; hopefully, most of these have been resolved in newer releases since this work was done.

5.5 Implementation observations, techniques, and lessons learned

Heterogeneous compute architectures composed of CPUs and some form of compute accelerators, such as GPUs, are one of the most likely hardware architectures to be present in exascale-class systems. The foregoing applications demonstrate that many applications common to scientific computing can be successfully implemented on the GPU or other compute accelerator devices, indicating that exploiting intranode/internode multilevel parallelism in algorithms will be a key strategy for high-performance computing, although some algorithm adaptation will need to take place. Many software applications which are executed on current heterogeneous supercomputers, such as Roadrunner or TianHe-1A, exploit only the CPU compute node-level parallelism of the machine, neglecting the data-level parallelism available with the compute accelerator part of the architecture. For an cost comparable to the effort required to implement MPI, OpenCL, or CUDA can be introduced, offering at least a 10× speedup and possibly much more.

Returning to the regular grid problem, we saw that it took 144 processors with MPI to match the compute power of 1 GPU. When we added the additional processors for speeding up the calculation, we also had to add the memory for each processor even though we did not need it. As a rough figure, the memory consumes about half the power for a node. This means that our energy efficiency dropped by a factor of 70 for unneeded memory just to get the processing speedup. For heterogeneous architectures, the compute accelerator level of the system can provide powerful low-latency parallelism, and the MPI layer is needed to give memory scalability.

Some further design lessons learned from developing these research codes are the necessity of conceptually separating processor and memory scaling and dealing with each programmatically; and the need for a commodity programming stack to avoid application and hardware dead ends and enable rapid hardware development iterations.

One of the surprising outcomes for all of the participants in this study was how much of each application could be implemented on the GPU using OpenCL or CUDA. Coming into the effort, the goal was to get the main computational kernel on the GPU for each of the applications. Quickly it became apparent that most of the compute cycle could be moved to the GPU given enough resources, innovative algorithm development, and rapid hardware/GPU language improvements. There are still challenges and research to be done in porting some application parts to the GPU. These include many aspects of the unstructured computations and code blocks with large amounts of divergent logic (conditional statements) where the lockstep nature of the current GPU implementations results in poor parallelization. Our optimism for improvement in these areas is high given the breakthroughs in porting complex algorithms to the GPU that we thought to be unsuited for such architectures just a short time ago.

At the outset of the project, there was concern over contention for the GPU while running these kinds of applications. No contention problems were encountered on a wide range of hardware, including Mac OS X-based and Unix-based laptops and desktops with GPUs from both ATI and NVIDIA. On the research cluster with 42 processors and 2 GPUs, we ran 48 MPI ranks driving one or both NVIDIA GPUs without seeing contention for the devices.

It is important to note that not all of the participants were GPU programming experts or even familiar with MPI when they started the study. Yet within a few months' time, they were comfortable with the programming languages and implementing complex numerical algorithms with them. The newcomers to both OpenCL/CUDA and MPI were observed to be more comfortable with the GPU language after a few months, though this is likely due to the relatively greater time spent on GPU programming during the project.

6 Conclusion

The unification of the separate strands of parallel computing, namely vectorization, MMP, and threading, will allow numerical calculations to be performed on larger scales than ever before, but will make new demands on developers of scientific software to exploit the hardware potential. As exascale supercomputers become available over the next decade, well-designed software libraries and applications that exploit the hardware in a portable yet efficient way will provide the best speedup for high-performance scientific computing. Development of standard libraries for testing and optimizing kernels with various tile sizes would be a priority for workers in the eventually adopted industry-wide standard for GPGPU computing. We note that there have been some substantial steps taken in this direction, such as Intel's Hierarchy-Savvy parallel algorithm design (Hi-Spade) approach (Chen, Gibbons, and Nath [7]) and several DOE-based initiatives in exascale codesign work (listed in Pao [25], among others), although much of the challenge stills lies ahead.

In this article, we have sought to illustrate some of the considerations and advances for initial test problems in this area using OpenCL and MPI, demonstrating that the two-level paradigm of intranode/internode communications can be exploited and that applications requiring dynamic memory allocation and nonregular grids or meshes are practicable. Although challenges certainly remain, portable and open industry standards such as MPI and OpenCL will allow scientists and engineers to program heterogeneous devices without wasting excessive time on porting the code from one architecture to another. As hardware is pushed to more extreme limits, scientific and numerical applications will need to exploit the available computing power more effectively. And as parallel programming becomes more ubiquitous, the demand for skills to effectively manage data and numerical calculations on high-performance supercomputers will only increase.

Acknowledgements The authors would like to thank Ben Bergen and Marcus Daniels for the use of Darwin, the LANL CCS GPU cluster.

The authors are also grateful to the LANL CCS/X-Division Exascale working group led by Tim Kelley and to Scott Runnels for organizing the LANL X-Division Summer Workshop exascale group at which much of the foundational work for this article was performed. These groups encouraged the work on the applications in different computational domains.

This work was supported by Los Alamos National Laboratory. Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the US Department of Energy under contract DE-AC52-06NA25396.

References

1. Bergen BK, Daniels MG, Weber PM (2010) A hybrid programming model for compressible gas dynamics using OpenCL. In: 2010 39th international conference on parallel processing workshops. doi:10.1109/ICPPW.2010.60
2. Bhatele A (2010) Automating topology aware mapping for supercomputers. Dissertation, University of Illinois at Urbana–Champaign
3. Boillat J, Burkhart H, Decker K, Kropf P (1991) Parallel computing in the 1990's: attacking the software problem. Phys Rep 207(3–5):141–165
4. Bowers KJ, Albright J, Bergen B, Yin L, Barker J, Kerbyson DJ (2008) 0.374 Pflop/s trillion-particle kinetic modeling of laser plasma interaction on Roadrunner. In: Proceedings of the 2008 ACM/IEEE conference on supercomputing, SC '08. IEEE, Piscataway, pp 63:1–63:11
5. Bowers KJ, Albright BJ, Yin L, Bergen B, Twan T (2008) Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. Phys Plasmas 15(5):055703. doi:10.1063/1.2840133
6. Casaglia G (1976) Distributed computing systems: a biased review. Euromicro Newsl 2(4):5–18
7. Chen S, Gibbons B, Nath S (2011) Rethinking database algorithms for phase change memory. In: Proceedings of the 5th biennial conference on innovative data systems research (CIDR'11)
8. Davis SF (1987) A simplified TVD finite difference scheme via artificial viscosity. SIAM J Sci Comput 8(1):1–18. doi:10.1137/0908002
9. DeVito Z, Joubert N, Palacios F, Oakley S, Medina M, Barrientos M, Elsen E, Ham F, Aiken A, Duraisamy K, Darve E, Alonso J, Hanrahan P (2011) Liszt: a domain specific language for building portable mesh-based PDE solvers. In: Proceedings of the 2011 ACM/IEEE conference on supercomputing
10. Dongarra J (2009) An overview of HPC and challenges for the future. In: HPC Asia 2009. http://www.nchc.org.tw/en/news/index.php?NEWS_ID=49. Accessed 29 July 2011
11. Feng W, Cameron K (2007) The Green500 list: encouraging sustainable supercomputing. Computer 40(12):50–55
12. Ferenbaugh C (in review) A comparison of GPU strategies for unstructured mesh physics. Concurr Comput Pract Exp
13. Gropp W, Lusk E, Doss N, Skjellum A (1996) A high-performance, portable implementation of the MPI message passing interface standard. Parallel Comput 22(6):789–828
14. Harvey M, Fabritiis GD (2011) Swan: a tool for porting CUDA programs to OpenCL. Comput Phys Commun 182(4):1093–1099
15. Kato S, Lakshmanan K, Kumar A, Kelkar M, Ishikawa Y, Rajkumar R (2011) RGEM: a responsive GPGPU execution model for runtime engines. In: 2011 IEEE 32nd real-time systems symposium (RTSS), pp 57–66
16. Kato S, McThrow M, Maltzahn C, Brandt S (in press) Gdev: first-class GPU resource management in the operating system. In: 2012 USENIX annual technical conference (USENIX ATC'12)
17. Khaleel MA (2010) 2010 exascale workshop panel report meeting. Technical report PNNL-19515, Pacific Northwest National Laboratory, Department of Energy, Washington, DC
18. Klimovitski A (2001) Using SSE and SSE2: misconceptions and reality. In: Intel developer update magazine, March 2001, pp 1–8
19. Kogge P, Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, Franzon P, Harrod W, Hill K, Hiller J, Karp S, Keckler S, Klein D, Lucas R, Richards M, Scarpelli A, Scott S, Snavely A, Sterling T, Williams RS, Yelick K (2008) Exascale computing study: Technology challenges in achieving exascale systems. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.6676. Accessed 23 March 2012
20. Los Alamos National Laboratory (2011) Flag 3.2 alpha 5 radiation–hydrodynamics code (LA-CC 11-065)
21. Message Passing Interface Forum (1994) MPI: a message-passing interface standard. Int J Supercomput Appl High Perform Comput 8(3–4):159–416
22. Mills A, Wood L (1981) Cray-1: a powerful delivery system for engineering software. Adv Eng Softw 3(2):62–66
23. Nicholaeff D, Davis N, Trujillo D Robey, R (in review) A cell-based adaptive mesh refinement implemented with general-purpose graphics processing units. SIAM J Sci Comput
24. Oyanagi Y (2002) Future of supercomputing. J Comput Appl Math 149(1):147–153
25. Pao K (2011) Co-design and you: why should mathematicians care about exascale computing. In: 2011 DOE applied mathematics program meeting

26. Papadrakakis M, Stavroulakis G, Karatarakis A (2011) A new era in scientific computing: domain decomposition methods in hybrid CPU–GPU architectures. Comput Methods Appl Mech Eng 200(13–16):1490–1508
27. Robey RN, Nicholaeff D Robey, RW (in review) Hash-based algorithms for discretized data. SIAM J Sci Comput
28. Simon H, Zacharia T, Stevens R (2007) Modeling and simulation at the exascale for energy and the environment. Technical report, Department of Energy, Washington, DC
29. Snir M, Gropp W, Kogge P (2011) Exascale research: preparing for the post-Moore era. http://hdl.handle.net/2142/25469. Accessed 25 July 2011
30. Sottile M, Rasmussen C, Weseloh W, Robey R, Quinlan D, Overbey, J (in press) ForOpenCL: transformations exploiting array syntax in Fortran for accelerator programming. Int J Comp Sci Eng
31. Sottile MJ, Rasmussen CE, Weseloh WN, Robey RW, Quinlan J, Overbey J (2011) ForOpenCL: transformations exploiting array syntax in fortran for accelerator programming. CoRR abs/1107.2157
32. Tendler J, Dodson JS, Fields S, Le H, Sinharoy B (2002) POWER4 system microarchitecture. IBM J Res Dev 46(1):5–25
33. Wolfe M (2008) How we should program GPGPUs. Linux Journal, November 2008. http://www.linuxjournal.com/magazine/how-we-should-program-gpgpus. Accessed 29 July 2011
34. Yang XJ, Liao XK, Lu K, Hu QF, Song JQ, Su JS (2011) The TianHe-1A supercomputer: its hardware and software. J Comput Sci Technol 26(3):344–351
35. Young J (2011) Supercomputers let up on speed. The Chronicle of Higher Education, April 2011. http://chronicle.com/article/In-University-Supercomputing/126979/. Accessed 20 July 2011
36. Zhang C, Yuan X, Srinivasan A (2010) Processor affinity and MPI performance on SMP–CMP clusters. In: 2010 IEEE international symposium on parallel distributed processing, workshops and PhD forum (IPDPSW), pp 1–8. doi:10.1109/IPDPSW.2010.5470774