Journal of Molecular Graphics and Modelling 30 (2011) 82–89

Contents lists available at ScienceDirect

Journal of Molecular Graphics and Modelling

journal homepage: www.elsevier.com/locate/JMGM

Topical perspectives

Topical perspective on massive threading and parallelism

Robert M. Farber*

PNNL, P.O. Box 999, Richland, WA 99352, United States

ARTICLE INFO

Article history:
Received 9 May 2011
Received in revised form 15 June 2011
Accepted 17 June 2011
Available online 29 June 2011

Keywords:
CUDA
OpenCL
Parallel computing
GPU
Computer architecture

ABSTRACT

Unquestionably computer architectures have undergone a recent and noteworthy paradigm shift that now delivers multi- and many-core systems with tens to many thousands of concurrent hardware processing elements per workstation or supercomputer node. GPGPU (General Purpose Graphics Processor Unit) technology in particular has attracted significant attention as new software development capabilities, namely CUDA (Compute Unified Device Architecture) and OpenCL™, have made it possible for students as well as small and large research organizations to achieve excellent speedup for many applications over more conventional computing architectures. The current scientific literature reflects this shift with numerous examples of GPGPU applications that have achieved one, two, and in some special cases, three-orders of magnitude increased computational performance through the use of massive threading to exploit parallelism. Multi-core architectures are also evolving quickly to exploit both massive-threading and massive-parallelism such as the 1.3 million threads Blue Waters supercomputer. The challenge confronting scientists in planning future experimental and theoretical research efforts – be they individual efforts with one computer or collaborative efforts proposing to use the largest supercomputers in the world is how to capitalize on these new massively threaded computational architectures – especially as not all computational problems will scale to massive parallelism. In particular, the costs associated with restructuring software (and potentially redesigning algorithms) to exploit the parallelism of these multi- and many-threaded machines must be considered along with application scalability and lifespan. This perspective is an overview of the current state of threading and parallelize with some insight into the future.

Published by Elsevier Inc.

1. Introduction and application

The trend to massive parallelism appears inescapable as current high-end commodity processors, the workhorse for most scientific computing applications, now support eight or more simultaneous threads of execution per CPU socket. Most computational nodes and scientific workstations contain several multi-core sockets, allowing these systems to deliver sixteen or more concurrent threads of execution. Multi-threaded applications that fully utilize this hardware capability can deliver an order of magnitude increase in computational throughput and corresponding decrease in time-to-solution. A multi-threaded application breaks an application into multiple pieces, or threads of execution, that run concurrently. Highly parallel applications can utilize multi-threading to reduce application runtime by the number of threads, which can be significant when using a newer architectures that supports tens of thousands of parallel threads. Applications that use one or very few threads of execution will likely not benefit from newer architectures.[1]

A 10× performance increase is certainly significant, but it does not necessarily represent a fundamental change for computation-dependent science. Machines with this level of performance make the computational workflow more interactive because computational tasks that previously took hours can finish in minutes and extended computational tasks that previously took days can run overnight.

Applications that deliver 100× or faster performance, as has been demonstrated with General purpose graphics processors units (GPGPU) technology across a broad spectrum of scientific problems, are disruptive and have the potential to fundamentally affect scientific research by removing time-to-discovery barriers. Computational tasks that previously would have required a year to

[1] Some single-threaded distributed applications that use MPI or other distributed software frameworks can use each computational core as a separate process. As discussed later, this can be wasteful and begs the question if this is the most computationally efficient mapping for many applications. It also precludes using other architectures such as GPUs.

* Corresponding author.
E-mail address: rmfarber@usa.net

1093-3263/$ – see front matter. Published by Elsevier Inc.
doi:10.1016/j.jmgm.2011.06.007

complete can finish in days. Better scientific insight becomes possible because researchers can work with more data and have the ability to utilize more accurate, albeit more computationally expensive, approximations and numerical methods.

Dennard's scaling laws are at the heart of the industry-wide change to multi-core processors [1,2]. Effectively they say that power density (voltage×(current/area)) will remain constant even as the number of transistors and their switching speed increases. For that relationship to hold, voltages need to be reduced in proportion to the linear dimensions of the transistor. Fabrication techniques have reduced the size of transistors to the point that manufacturers are no longer able to lower operating voltages sufficiently to match the performance gains that can be achieved by simply adding more computational cores to the processor chip. In a competitive market, minor changes in processor performance do not translate into increased sales for CPU manufacturers – hence the proliferation of multi-core processors. In general, the industry appears to believe this trend will continue and the number of cores per processors will increase.

GPGPUs are an alternative commodity massively parallel architecture that is readily available and inexpensive. Since GPGPU architectures evolved in the highly competitive visualization and gaming market, they were designed from the beginning for massive parallelism. These architectures utilize replication of hardware computational units called shaders to perform real-time visualization tasks in a high-performance and scalable fashion. With the advent of programmable shaders, which have been refined into the current generation of general-purpose streaming multiprocessors, GPGPUs quickly evolved into very capable high performance computational devices that can deliver a number of teraflops (10^{12} floating-point operations per second) of computational capability for graphics and scientific problems. By varying the number of the streaming multiprocessors, low-end devices with tens to hundreds of hardware computational units can be offered at current retail price points under a hundred dollars while high-end devices containing many hundreds to a thousand hardware computational units can be sold at current retail price points of a few thousands of dollars. Since GPGPUs are small and can provide an excellent flop per watt ratio, very large supercomputer clusters are currently being built that utilize both multicore and GPGPU technology. As of the November 2010 TOP500 list, the first and third fastest supercomputers in the world (China's Nebulae and Tehane-1 hybrid CPU/GPU supercomputers) provide examples of this developing trend. For the right applications, GPGPU technology can provide orders of magnitude increased performance for applications running on personal computers to leadership class supercomputers.

Programmable in high-level languages such as the C language with the NVIDIA CUDA (Compute Unified Development Architecture) tools, OpenCL™ and other frameworks, GPGPUs have brought massively parallel supercomputing to the worldwide masses. Freely available software development kits (SDKs) have made programming these devices accessible to anyone from teenagers to large research efforts.

Currently, the NVIDIA CUDA SDK is the most popular. First introduced in February 2007, CUDA is now taught at over 362 universities and academic institutions worldwide. NVIDIA estimates there is already an installed base of over 250 million CUDA-enabled GPGPUs [3]. OpenCL is a standard based alternative to CUDA that provides multi-platform support for products from most hardware vendors such as AMD, NVIDIA, IBM, Intel, Apple, and others. The OpenCL standard is administered by the Khornos group [4]. Source translators such as SWAN [5] can convert CUDA programs to OpenCL. Other software such as MCUDA (CUDA to Multi-core) [6] and Ocelot [7] allow CUDA applications to run on multi-core processors [8], while OpenCL can directly compile to run on multi-core processors. New data-parallel C++ extensions such as Thrust [9] can dramatically simplify GPGPU code development.

Performance, low cost, and power efficiency are three reasons to consider GPGPU technology as legacy application software must be modified or redesigned, which requires a software investment. The benefit is potentially orders of magnitude increased performance for both experimental and simulation based research. As the future is clearly parallel, legacy projects that do not consider investing in software and algorithm development to use one or more of these hardware architectures risk stagnation and loss of competitiveness [10].

The recent scientific and technical literature published during the previous three years demonstrates a proliferation of GPGPU-enabled applications and algorithms that deliver one to two orders of magnitude speedup (10–100×) in performance over conventional processors across a broad spectrum of algorithmic and scientific application areas and additional speedups through the use of multiple GPGPUs either within a workstation or joined together within a computational cluster. This represents a remarkable rate of adoption as CUDA in particular has only been around for 4 years.

Summarizing the breadth and proliferation of applications that map well to GPGPU technology is a challenge. Those applications that achieve high performance are massively threaded so they can fully utilize the many hundreds to thousands of simultaneous hardware threads of execution that become available when one or several GPGPU boards are plugged into a conventional processor motherboard or are joined within a computational cluster.

The NVIDIA Community Showcase [3] provides a central repository that can be used to examine the broad applicability of GPGPU technology to a wide-variety of computational problems as reported in the peer-reviewed and technical literature. While admittedly showcasing applications that perform well, the following graph shows the top 100 fastest reported speedups (maximum 2600×, median 1350×, and minimum 100×) as of May 9, 2011 on NVIDIA CUDA GPUs. Other peer-review surveys reporting both techniques utilized and speedups are appearing [11] (Fig. 1).

While successful GPU applications can deliver impressive performance, it is important to note that GPGPU technology is good for some but not all computational problems. Due to limitations in the SIMD (Single Instruction Multiple Data) execution model and per-thread resources, GPGPUs tend to have a "knife-edge" performance envelope, which means it can be challenging to get high-performance with these devices. Also, it is important to verify that the reported speedups represent realistic performance of a multicore system as opposed to single-core performance. As has been stressed by the Intel Throughput Computing Lab and Architectures Group, further scrutiny of reported results is needed to ensure that equal effort was made to optimize both CPU and GPU implementations [12].

Many authors report GPGPU performance based on single-precision floating-point performance. While double-precision (64-bit) performance is increasing, single-precision (32-bit) performance is still faster on the current generation of GPGPU products. Double-precision performance can be dramatically slower on older generations of GPGPUs. Using mixed-precision arithmetic judiciously, say for reduction operations [13,14] can preserve accuracy along with other techniques for long-running simulations [15].

Many statistical modeling applications map efficiently to GPGPUs. Since they are massively parallel, Monte Carlo methods map well as seen in the 30–100× speedups [16,17] for a single GPU as reported by Salazay and Rohonczy for simulations of dynamic NMR spectra [18]. Other authors report performance increases up to 120× for Monte Carlo simulations using multiple GPUs in an OpenMP framework [19]. In performance surveys of GPU-based

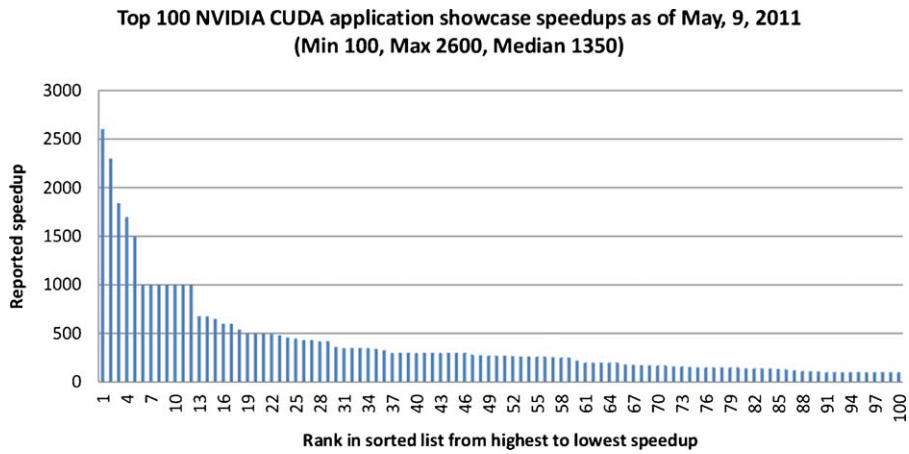

Fig. 1. NVIDIA's top 100 fastest reported speedups over conventional processors.

Monte Carlo implementations, OpenCL currently appears to be slower than CUDA [20]. Other statistical modeling applications such as Hidden Markov Models [21], Bayesian networks [22,23] and many others also run efficiently [11,24,25].

A number of researchers and organizations report excellent speedups on quantum chemistry simulations [11] including Gaussian and GAMESS [26–29]. Pseudospecteral methods such as BigDFT [30,31] perform well on both GPGPUs and multi-core processors.

Molecular modeling has also achieved significant performance benefits using GPGPU technology [32–35] including packages such as the NAMD/VMD molecular modeling system [36–39] and HOOMD-Blue [40] which was written from the ground-up for GPUs. Other GPU-enabled quantum chemistry packages such as TeraChem are being deployed at major supercomputing centers [27–29,41,42]. Papers such as "GPU-Accelerated Molecular Modeling Coming of Age" [39] have generated significant interest [43]. Multithreaded technology has also stimulated the development of new approximations for electrostatics that can provide up to 3 orders of magnitude speedup [44].

GPGPUs provide additional performance benefits for dedicated usage, visualization, and interactive workflows as they have been designed to be plugged into computers situated next to an instrument or individual's desk. The recent scientific literature demonstrates applications of this technology to a number of instruments and projects. A reported 350× speedup allowed Balanchi and Di Leonardo to incorporate real-time hologram generation into an optical micro manipulation [45] workflow. Commonly used visualization packages such as VMD help make molecular modeling more interactive [39] and incorporate haptic feedback [46]. "Think big" research projects to use GPGPU technology to perform automated image recognition and reconstruction tasks that were not tractable with older technology. For example the Connectome project is an innovative effort to map all the connections in the brain and peripheral nervous system of model animals, such as the lab mouse, utilizing massively parallel computational techniques to assemble and identify features based on 3 nm/pixel high-resolution microscopy images [47]. The Sscrett (Serial Section Reconstruction and Tracing Tool) and NeuroTraceapplications utilize GPGPU hardware programmed in CUDA to perform scalable large scale rendering and tubular surface reconstruction of 3D processes on $\sim 10^{14}$ of volumetric voxel data [48,49].

The previous examples represent but a few of many research efforts reporting significant benefits from the application of GPGPU technology. Bioinformatics [50], systems biology [51], multicellular biological modeling [52], chemical and protein search [53,54] name but a few additional broad areas. Use of a good Internet search engine can help identify specific research projects of interest. Many projects make their software freely available. Several general-purpose libraries to facilitate scientific computation such as NVIDIA's CUBLAS and CUFFT libraries along with the MAGMA (Matrix Algebra on GPU and Multicore Architectures) hybrid CPU/GPU library [55] are also included in the SDK or available for download for free.

Newer massively parallel machines that resemble conventional multi-core processors are on the horizon such as the IBM Blue Waters project. First shown in prototype form at Supercomputing 2009, the IBM system is based on a Power-7 processor that will support 128 threads per MCM (Multi-Chip Module). This water cooled hardware is the heart of the University of Illinois "Blue Waters" supercomputer that will support over a hundred thousand threads and deliver 1 petaflop (10^{15}) sustained floating-point operations per second. The fully loaded Blue Waters system will hold 2 PB (2×10^{15}or 2 million gigabytes) of main memory that is addressable by all the processors. One of the most salient features of the global memory architecture of this system is that it is not required to be cache coherent. This departure from conventional processor design is important to achieve scalability according to number hardware threads but violates data coherency assumptions generally assumed by parallel software designed for conventional hardware. As a result, some software investment is again likely to be required to efficiently utilize this system.

Other massively threaded architectures such as the Cray XMT target important yet specialized computational problems that are memory access latency bound rather than floating-point dominated. Large graph-based algorithms in particular perform poorly on commodity cache-based processors because most, if not all, memory accesses result in a CPU cache and potentially an MMU (Memory Management Unit) TLB (Translation Lookaside Buffer) miss as well. This can add thousands of cycles of delay per memory access causing application performance to degrade severely. For this reason, the Threadstorm processors and memory subsystem of the Cray XMT were designed to support very large numbers of threads (on the order of thousands to millions) to hide memory latency and deliver both high performance and near-linear scalability according to number of processors on these types of problems [56–59].

To utilize the current multi-threaded and massively parallel computational capability and the ever more massively parallel systems that will become available in the next few years requires a commitment to rewrite many of the computationally intensive por-

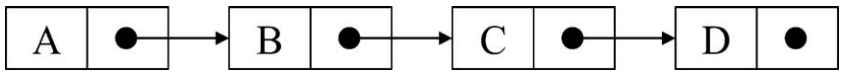

Fig. 2. Schematic representation of a linked-list.

tions of existing applications to effectively utilize a large number of concurrent threads of execution. Such a commitment appears to be necessary to keep computation-dependent research efforts competitive as readily available, ubiquitous and inexpensive parallel hardware makes orders of magnitude increases in computational performance available to everyone in the world-wide scientific community.

2. Why threading is important

Creating software that splits a computation into separate threads of execution is the keyto harnessing the computational power of multi-core processors and massively parallel hardware.

Essentially threads are individual pieces of the same program that can execute simultaneously. For example, the vector multiply shown below can be broken into **N** separate threads where each thread simultaneously calculates vector c for each element index **i**:

Example 1. *A simple vector example.*

```
for (i=0;  i<N;  i++
  c{i}=a{i}*b{i}
```

Good utilization of the multi-core or massively threaded hardware available on a computer can be achieved when the number of software threads equals or exceeds the number of hardware processing units. This makes sense because the software has broken up the computational task into enough pieces to give each hardware computational unit something to do. Obviously, there will be idle hardware units when there are fewer software threads than hardware units. This is reflected in Amdahl's law that notes the speedup of a program using multiple processors will be limited by the time spent in any sequential portions of the code.

A simple example of a purely sequential algorithm is linked list traversal. As can be seen below, the location of the next node in the list can only be determined after the contents of the current node are known (Fig. 2).

In many cases, it is possible to utilize alternative algorithms that perform the same function and parallelize effectively. For example, indexed data structures and associative arrays can be used to store and lookup values in a highly parallel fashion without the sequential limitations of a linked list [60,61]. Thus, rewriting an existing application might also entail re-examining the algorithms used to perform the work and/or calculation.

Utilizing massive multithreading (hundreds to millions of simultaneous threads) has important implications to achieving very high performance because it provides a common paradigm that both programmers and hardware designers can exploit to attain the highest possible performance.

Architectures that are designed to support massive multithreading, such as GPGPUs and the Cray XMT, have very efficient internal hardware schedulers that can very quickly assign ready to run software threads to any available hardware resources. From a software standpoint, this scheduling operation effectively happens for free. However, it can dramatically reduce the time to solution by ensuring that all available hardware resources are utilized as effectively as possible.

3. The benefits of massive threading for the Cray XMT

The Cray XMT is designed to utilize a very large number of threads (on order thousands to millions) to hide the latency of arbitrary memory accesses. Large graph based algorithms in particular tend to be latency dominated due to irregular memory accesses. These are algorithms that do not permit locality of reference or data reuse, so hardware caches are rendered useless because each memory access almost always causes a cache miss. Conventional cache-based architectures fail to perform well on this general class of problems that arise when solving graph, finite automata and a myriad of other algorithms associated with large-scale data analysis and data-mining.

In contrast to conventional processors, the memory subsystem of the XMT uses address scrambling to literally break locality by scattering addresses in a repeatable fashion throughout the address space of the machine. As a result, the underlying memory subsystem will see – on average – a uniform distribution of memory requests regardless of the memory access pattern an arbitrary program might exhibit. From a parallel performance point of view, this is important because multiple requests to a single memory bank, or bank conflict, will introduce a serial bottleneck into a parallel program as each separate memory request can only be serviced sequentially – one after another. As a result, the time to solution will suffer while individual processors stall waiting for data.

Applications that use a large number of threads provides the XMT thread scheduler with a plethora of outstanding memory requests from which it can pick and choose to eliminate bank conflicts. This in turn allows the memory subsystem to deliver data at the fastest possible rate at all times so the fewest number of processors will be stalled waiting for data and the best time to solution achieved. Of course, using large numbers of threads also creates the opportunity for other optimizations as well.

4. The benefits of massive threading for GPGPUs

In contrast to conventional processors with 4 or 8 cores per processor, GPGPUs support hundreds of concurrent floating-point threads of operations. Both NVIDIA and AMD offer GPGPUs that can provide over 2 teraflops (trillion floating-point operations per second) of peak single-precision floating-point performance and around 600 GF/s (billion floating-point operations per second) of peak double-precision floating point performance. AMD GPUs offer around 1500 stream processors while NVIDIA provides 512 hardware thread processors. GPGPUs from both manufacturers support tens of thousands of concurrent software threads of execution. Note that AMD products can only be programmed using OpenCL while NVIDIA products run both OpenCL and CUDA applications.

GPGPUs are external boards that plug into conventional computers through the PCI Express bus. Multiple boards can be plugged into a single conventional computer and located at a test instrument or next to a student or researcher's desk. Many of these same boards also provide superb visualization capabilities.

From a hardware standpoint, massively parallel multithreading on GPGPUs is achieved through the use of a common architectural building block called a multiprocessor that can be replicated as required by the manufacturer to provide the largest number of processing cores for a given product price-point. Each of these multiprocessors contains internal memory that can be utilized by the multiprocessor computational units to deliver actual or near-peak performance.

Unfortunately, each multiprocessor contains only a small amount of fast internal memory so most programs have to utilize data in the much slower global memory space shared between all the multiprocessors. Most GPGPUs provide several gigabytes of onboard global memory.

Example 1, the multiplication of a vector, illustrates the impact of memory bandwidth on floating-point performance as 3 memory operations are required, two reads and a write, for every floating-point multiply. Assuming single-precision (32-bit or 4-byte) floating-point values are being used, the memory subsystem of a teraflop capable computer would need to provide 12 terabytes per second of memory bandwidth for this calculation to run at full speed. Roughly speaking such bandwidth is 50–100-times the capability of current GPU memory subsystems and approximately 375-times the bandwidth of the latest generation of high-end commodity processors. When the extra precision of 64-bit (8-byte) floating-point arithmetic is required, the reader can double these numbers (or halve the effective computational rate). Thus Example 1 illustrates the necessity of reusing as much data within each local multiprocessor as possible to achieve high performance. (Please note that data reuse is also important to attaining high performance on conventional processors as well.)

Scheduling a thread to run on a GPGPU multiprocessor requires knowledge of when all the memory dependencies are satisfied – be they internal to the streaming multiprocessor or external global memory. In addition, each multiprocessor contains a number of single and double precision floating point units, integer, and special function units that calculate transcendental functions very efficiently. Thus the hardware scheduler must take into account the thread requirements and availability of each of these different resources. For these reasons, the on-board GPGPU hardware thread scheduler is essential to achieving high floating-point performance.

Programming with a large number of threads allows the hardware thread scheduler to fully utilize the capabilities of the GPU because it can pick and choose amongst the active threads to:

- Fully utilize all the internal resources of the hardware (be they floating-point, integer, or special function units) by scheduling those threads that do not have to wait on the memory subsystem to use whatever internal resources happen to be available at that moment.
- Maximize the memory bandwidth of the memory subsystem by working together with internal coalescing units[2] to stream data to/from all the memory banks at the highest possible data rate.

Other caveats apply to GPU programming as well. For example, both AMD and NVIDIA execute multiple threads within a multiprocessor which utilizes a SIMD (Single Instruction Multiple Data) execution model. SIMD execution allows instructions to be broadcast inexpensively and quickly within the multiprocessor but requires that all parallel processing for each block of threads happen in lock-step. SIMD execution is very efficient, but conditionals (caused by if statements) can degrade performance because the multiprocessor must evaluate each branch of every conditional operation. However, GPGPUs are considered to be SPMD (Single Program Multiple Data) computers because different multiprocessors can be executing different instructions at the same time. Future GPGPU software and hardware versions will certainly help ease the SIMD conditional limitations through the ability to run multiple kernels and potentially provide other capabilities.

[2] A coalesced memory operation combines simultaneous memory accesses by multiple threads into a single memory transaction. This is in contrast to a bank conflict, which occurs when multiple memory requests fall in the same memory bank and causes the competing accesses to be serialized.

5. Hybrid systems and other architectures

Hybrid hardware systems that combine GPGPU and conventional processors appear to be a popular near-term solution for the next few hardware generations. Given the power efficiency and price performance, graphics processors are a natural platform for many – but not all problems. As previously discussed, a number of computational characteristics including resource limitations, global memory bandwidth, limitations in the SPMD model, and other concerns prevent many important problems from running with high-efficiency on graphics processors. However, a large number of those problems that do not run well on graphics processors do tend to run well on commodity hardware. For this reason, hybrid systems are of interest.

The IBM Power-7 represents the continued evolution of more conventional processor architectures to supporting more than a hundred processing cores per processor package. The elimination of cache coherency comes as no surprise because these operations are a known scaling bottleneck when SMP (Symmetric MultiProcessing) systems reach higher core counts. The ability to run without cache-coherent operations represents recognition with higher core counts that scalable software requires the programmer be in control of coherency decisions rather than mandating they be handled automatically. It will be interesting to see how this affects the programming model used on the Blue Waters supercomputer which will provide over a hundred thousand cores in a shared memory environment. It is likely that commodity multi-core processors will also experience cache coherency scaling issues as they evolve from 4- and 8-way processors to many-core processors with tens to hundreds of cores per processor.

Regardless of market: commodity processors, specialized machines, graphics processors or leadership class supercomputers, the move towards massive parallelism indicates that current applications must adapt to match the capabilities of highly parallel hardware.

6. Coding use

A number of approaches exist for programming with large numbers of threads. Unfortunately, there are no definitive guidelines about the cost and effort required to create a parallel version of an application – nor is there consensus on a single portable programming model.

Generally it is important to profile an application to find those routines (parts of the program) that consume most of the runtime. Attention can then be focused on parallelizing those routines to achieve the greatest speedup. Applications that spend most of their time in a few routines that can be parallelized will likely see the greatest speedup. Applications that have a more diffuse runtime behavior will either require greater effort to parallelize or may require reexamination of the work performed to determine if another algorithm better suited to parallelization can be utilized.

There are a number of different approaches to creating portable, multi-threaded applications. These include:

- Distributed frameworks such as MPI (Message Passing Interface)
- Threaded APIs (Application Programming Interface) such as pthreads
- Parallelizing compilers
- Language extensions
- New languages and frameworks

While technically a distributed rather than a threaded programming model, MPI is a well-established and commonly used

programming framework. Many applications that utilize MPI can run on conventional multi-core processors. Essentially, each processor core is utilized as a separate computational process distinct from all other MPI processes. Using MPI to run on multi-core processors can work but it might also waste significant memory and introduce needless communications overhead because all shared and modified data must be communicated and stored separately amongst the processes.

In contrast, threaded programs generally share a common memory space, which can eliminate redundant storage and needless communications overhead. This article will focus on using threads, rather than distinct processes for the remainder of this article. It is worth noting that MPI can be utilized in a hybrid computational approach to distribute an application across many devices where each distinct MPI process internally utilizes a thread-based model to run efficiently within the system or on a GPGPU.

Another commonly utilized threaded approach for multi-core processors is to program via an API such as pthreads (Posix Threads) and variants. In general, these APIs provide a low level interface for parallel programming inside a shared-memory environment. At this time, pthread-based applications will not run on GPGPUs and the Cray XMT. It will be interesting to see how portable pthread applications can be in transitioning to these architectures and non-cache coherent environments such at the Blue Waters system.

Many compilers will generate threaded code either automatically or when directed by a pragma in the source code. The Cray XMT compiler, for example, will generate Threadstorm specific parallel code for the XMT. Popular compilers for multi-core systems such as Gnu, Portland Group, and Intel take this approach to generate parallel FORTRAN and C code using OpenMP. The following shows how to parallelize Example 1 using an OpenMP pragma with gcc:

Example 2. *Parallelizing with OpenMP.*

```
#pragma omp parallel for
for (i=0;  i<N;  i++
c[i]=a[i]*b[i]
```

Pragmas can be also used to affect the layout and scheduling of the threads in many of these compilers.

Some compilers will also accept additional command-line arguments to generate Intel instruction set SSE (Streaming SIMD Extensions) for 86× architecture machines that can accelerate floating-point performance by a factor of 2 or more.

Several companies are developing compilers to support GPGPU computing for legacy applications. The Portland Group, for example, has developed a FORTRAN GPGPU compiler. Another company, CAPS enterprise, is taking the innovative approach of generating hardware-specific codelets for C and FORTRAN code through the use of pragmas with their HMPP compiler.

NVIDIA has extended the C programming language through the addition of a few keywords and syntactic additions to simplify creation of efficient applications for CUDA-enabled GPGPUs. The myriad of scientific applications that now utilize GPGPU technology provide ample proof that scientists and engineers find the existing CUDA programming model, compiler, and development tools to be adequate. OpenCL is another technology to watch as are the data-parallel extensions such as Thrust [9].

CUDA allows massively threaded applications to be written in C or C++ because the GPGPU hardware manages the threads for the programmer. Instead of explicitly creating threads as one would on a conventional processor using a thread library like pthreads, a CUDA developer writes a kernel that will run on the GPU.

In reality, a kernel is nothing more that a C-language subroutine (or C++ functor) that runs and utilizes variables that reside on the GPU. Unlike a conventional subroutine call, kernels are launched asynchronously which means that the host processor merely queues the kernel in a pipeline to be launched when the hardware is ready. A CUDA kernel that performs the calculation shown in Example 1 is as follows. This assumes that the kernel is called with **N** threads and that the vectors **a**, **b**, and **c** are resident on the GPGPU. Note: the **for** loop in Example 1 is implicit. Thus, each thread must calculate its particular index**i**givena grid configuration defined when the kernel is call from the host. Also, the __global__ identifier specifies the kernel is visible and callable by the host processor.

Example 3. *Example 1 expressed as a CUDA kernel.*

```
__global__void cudaKernel (float *a, float *b, float *c
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  c[i]=a[i]*b[i];
}
```

In addition to the normal C language subroutine parameters, a call to a kernel includes the specification of an execution configuration that defines how the threads will be mapped to the GPGPU hardware as well as the number of threads that will be utilized. When not resource constrained, AMD and NVIDIA recommends using a large numbers of threads – on order of thousands per GPGPU- to future-proof an application to maintain high performance on future generations of GPU products.

Syntactically, a call to a CUDA kernel looks like a C-language subroutine call except the execution configuration is added between triple angle brackets "⋘" and "⋙":

Example 4. *An example call to a CUDA kernel from host code.*

```
cudaKernnel⋘nBlocks, nThreadsPerBlock⋙(a, b, c);
```

The first 2 parameters between the angle brackets, define the number of thread blocks, **nBlocks**, and the number of threads within a thread block, **nThreadsPerBlock**. The total number of simultaneously running threads for a given kernel is the product of these 2 parameters. Without going into further detail, other parameters in the execution configuration can specified to define 2D and 3D topologies and shared-memory allocations.

Experience has shown that writing or porting software to graphics processors consists of three steps:

1. Getting (and keeping) the data on the GPU. Recall that GPGPUs are separate devices that are plugged into the PCI Express bus of the host computer. The PCIe bus is very slow compared to even the slowest memory system on the GPGPU. For example, a Tesla 20-series GPU has roughly 200 GB/s global memory bandwidth while the latest PCIe 16× v2.0 bus can only transfer 8 GB/s in a single direction.
2. Maximizing the amount of work performed per call to the GPU to eliminate the latency of starting a kernel asynchronously on the GPGPU. In a nutshell, GPGPUs are fast enough that they complete small problems faster than the host processor can start kernels. To get a sense of the numbers, let's assume this overhead is 4 μs for a 1 teraflop GPU that takes 4 cycles to perform a floating-point operation. To keep the GPGPU busy each kernel must perform roughly 1 million floating-point operations. If the kernel only takes 2 μs to complete, then 50% of the GPU cycles will be wasted.
3. Exploiting internal resources on the GPU (such as registers, shared memory, etc.) to bypass internal memory bottlenecks and maximize performance.

New programming frameworks are being proposed to facilitate the efficient development of massively parallel software in a portable manner. OpenCL is a framework that has fairly broad support from manufacturers that promises to allow the portable development of efficient massively parallel programs for CPUs, GPUs (from multiple vendors), and other processors. Based on the

C language, OpenCL requires the application programmer write kernels similar to CUDA.

7. Summary

Capitalizing on multicore and multithreaded hardware architectures is a significant issue facing scientists in planning future experimental and theoretical research efforts. At a minimum, thought should be given to decide if a software investment is necessary to utilize the performance of newer multi-core hardware. Legacy applications and research efforts that do not invest in multithreaded software will likely not benefit from modern multi-core processors because single-threaded and poorly scaling software will not be able to utilize extra processor cores. As a result, computational performance will plateau at or near current levels placing the projects that depend on these legacy applications at risk of both stagnation and loss of competitiveness. The payoffs can be tremendous for those research efforts can make use of the available orders of magnitude increased performance that can be achieved with the current massively parallel software SDKs and hardware.

Acknowledgements

I would like to thank EMSL, a national scientific user facility sponsored by the Department of Energy's Office of Biological and Environmental Research, located at Pacific Northwest National Laboratory, for providing the funds to write this article.

References

[1] M. Bohr, A 30 year retrospective on Dennard's MOSFET scaling paper, IEEE SSCS 12 (2007).
[2] R.H. Dennard et al., Design of Ion-implanted MOSFETs with Very Small Physical Dimensions, IEEE J. Solid-State Circuits 9, 0018–9200.
[3] NVIDIA, CUDA Zone, Online: http://www.nvidia.com/cuda.
[4] Group, Khronos, Online: http://www.khronos.org/.
[5] Multiscalelab, Online: http://www.multiscalelab.org/swan.
[6] Stratton, John, Stone, Sam, Hwu, Wen-mei, MCUDA: an efficient implementation of CUDA Kernels for multi-core CPUs, in: A. JosÃ (Ed.), Languages and Compilers for Parallel Computing, vol. 5335, Springer, Berlin/Heidelberg, 2008, pp. 16–30 (doi:10.1007/978-3-540-89740-8_2).
[7] Gpuocelot, Online: http://www.code.google.com/p/gpuocelot/.
[8] J.A. Stratton, et al., Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs, ACM (2010) 111–119.
[9] Google code, Thurst – code at the speed of light, 2011 Online: http://www.code.google.com/p/thrust/ (accessed 10.01.11).
[10] R.M. Farber, Redefining what is possible, Sci. Comput. (2011), http://www.scientificcomputing.com/articles-HPC-GPGPU-Redefining-What-is-Possible-010711.aspx.
[11] Hwu, W. Wen-mei, GPU Computing Gems, Morgan Kaufmann, 2011, ISBN 978-0-12-384988-5.
[12] V.W. Lee, et al., Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, SIGARCH Comput. Archit. News 38 (2010) 451–460.
[13] K. Yasuda, Accelerating density functional calculations with graphics processing unit, J. Chem. Theor. Comput. 4 (2008) 1230–1236.
[14] R. Farber, Niumerical precision: how much is enough? Sci. Comput. (2009) 14.
[15] M. Taufer et al., Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs, 2010.
[16] Alerstam Erik, Svensson Tomas, Andersson-Engels Stefan, Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration, J. Biomed. Opt. 13 (2008), doi:10.1117/1.3041496.
[17] Q. Fang, D.A. Boas, Monte Carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units, Opt. Express 17 (2009) 20178–20190.
[18] Z. Szalay, J. Rohonczy, Fast calculation of DNMR spectra on CUDA-enabled graphics card, J. Comput. Chem. (2010).
[19] L. Xu et al., Parallelization of tau-leap coarse-grained Monte Carlo simulations on GPUs, 2010.
[20] Rick Weber, et al., Comparing hardware accelerators in scientific applications: a case study, IEEE Trans. Parallel Distrib. Syst. 22 (2011) 58–68.
[21] N. Ganesan, et al., Accelerating HMMER on GPUs by implementing hybrid data and task parallelism, in: Proceedings of the International Conference On Bioinformatics and Computational Biology (ACM-BCB), Niagara Falls, NY, USA, 2010.
[22] M. Wahib et al., A Bayesian optimization algorithm for De Novo ligand design based docking running over GPU (2010) 1–8.
[23] M.A. Suchard, et al., Understanding GPU programming for statistical computation: studies in massively parallel massive mixtures, J. Comput. Graph. Stat. 19 (2010) 419–438.
[24] H. Zhou, L. Kenneth, A. Suchard, Graphics processing units and high-dimensional optimization, Stat. Sci. 25 (2010) 311–324.
[25] M. Januszewski, M. Kostur, Accelerating numerical solution of Stochastic Differential Equations with CUDA, 2009.
[26] NVIDIA, Computational Chemistry, 2011 Online: http://www.nvidia.com/object/computational_chemistry.html.
[27] I.S. Ufimtsev, T.J. Martinez, Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation, J. Chem. Theory Comput. 4 (2008) 222–231.
[28] I.S. Ufimtsev, T.J. Martinez, Quantum chemistry on graphical processing units. 2. Direct self-consistent-field implementation, J. Chem. Theory Comput. 5 (2009) 1004–1015.
[29] I.S. Ufimtsev, T.J. Martinez, Quantum chemistry on graphical processing units. 3. Analytical energy gradients, geometry optimization, and first principles molecular dynamics, J. Chem. Theory Comput. 5 (2009) 2619–2628.
[30] BigDFT, Institut Nanosciences et Cryogénie. Online: http://www.inac.cea.fr/L_Sim/BigDFT/.
[31] L. Genovese et al., Density functional theory calculation on many-cores hybrid CPU–GPU architectures, 2009.
[32] S. Hampton et al., Towards microsecond biological molecular dynamics simulations on hybrid processors (2010) 98–107.
[33] C.I. Rodrigues et al., GPU acceleration of cutoff pair potentials for molecular modeling applications (2008) 273–282.
[34] E. Roberts et al., Long time-scale simulations of in vivo diffusion using GPU hardware (2009) 1–8.
[35] P. Eastman, V.S. Pande, Efficient nonbonded interactions for molecular dynamics on a graphics processing unit, J. Comput. Chem. 31 (2010) 1268–1272.
[36] W. Humphrey, A. Dalke, K. Schulten, VMD – visual molecular dynamics, J. Mol. Graph. 14 (1996) 33–38.
[37] K. Laxmikant, et al., NAMD2: greater scalability for parallel molecular dynamics, J. Comput. Phys. 151 (1999) 283–312.
[38] NAMD, Theoretical and Computational Biophysics Grpup, University of Illinois at Urbana, Champaign, http://www.ks.uiuc.edu/Research/namd/.
[39] J.E. Stone, et al., GPU-accelerated molecular modeling coming of age, J. Mol. Graph. Modell. 29 (2010) 116–125.
[40] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, J. Comput. Phys. 227 (2008) 5342–5359, doi:10.1016/j.jcp.2008.01.047 (10, San Diego, CA).
[41] NCSA, NCSA deploys GPU-enabled TeraChem software on Lincoln cluster, 2010 Online: http://www.illinois.edu (accessed 22.04.11) (http://www.illinois.edu/lb/article/2101/46446/page=1/list=list).
[42] I.S. Ufimtsev, T.J. Martinez, Graphical processing units for quantum chemistry, Comput. Sci. Eng. 10 (2008), doi:10.1109/MCSE. 2008.148.
[43] Paper on GPU-accelerated molecular modeling is "#1 Most Downloaded" for journal. Online: http://www.GPUcomputing.net, 2011 (accessed 10.01.11) (http://www.gpucomputing.net/?q=node/2504).
[44] R. Anandakrishnan, et al., Accelerating electrostatic surface potential calculation with multi-scale approximation on graphics processing units, J. Mol. Graph. Modell. 28 (2010) 904–910.
[45] S. Bianchi, R. Di Leonardo, Real-time optical micro-manipulation using optimized holograms generated on the GPU, Comput. Phys. Commun. 181 (2010) 1444–1448.
[46] J. Stone, et al., Immersive molecular visualization and interactive modeling with commodity hardware, in: B. George, et al. (Eds.), Advances in Visual Computing, vol. 6454, Springer, Berlin/Heidelberg, 2010, pp. 382–393 (doi: 10.1007/978-3-642-17274-8_38).
[47] B.B. Biswal, et al., Toward discovery science of human brain function, Proc. Natl. Acad. Sci. U S A 107 (2010) 4734–4739.
[48] W.-K. Jeong, et al., Ssecrett and neurotrace: interactive visualization and analysis tools for large-scale neuroscience data sets, IEEE_M_CGA 30 (2010) 58–70.
[49] Ju Lu, Semi-automated reconstruction of neural processes from large numbers of fluorescence images, PLoS One 4 (2009) e5655.
[50] B. Schmidt (Ed.), Bioinformatics: High Performance Parallel Computer Architectures, Francis and Taylor, 2010, ISBN 9781439814888, p. 370.
[51] L. Dematte, D. Prandi, GPU computing for systems biology, Brief. Bioinform. 11 (2010) 323–333.
[52] S. Christley, et al., Integrative multicellular biological modeling: a case study of 3D epidermal development using GPU algorithms, BMC Syst. Biol. 4 (2010) 107.
[53] I.S. Haque, V.S. Pande, W.P. Walters, SIML: a fast SIMD algorithm for calculating LINGO chemical similarities on GPUs and CPUs, J. Chem. Inf. Model. 50 (2010) 560–564.
[54] A. Stivala, P. Stuckey, A. Wirth, Fast and accurate protein substructure searching with simulated annealing and GPUs, BMC Bioinformatics 11 (2010) 446.
[55] MAGMA. Innovative Computing Laboratory, The University of Tennessee. Online: http://icl.cs.utk.edu/magma/.
[56] D.A. Bader, K. Madduri, Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2 (2006) 523–530.
[57] D.A. Bader, K. Madduri, Parallel algorithms for evaluating centrality indices in real-world networks (2006) 539–550.

[58] D.A. Bader, K. Madduri, SNAP, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks (2008) 1–12.
[59] D. Ediger, et al., Massive social network analysis: mining twitter for social good (2010) 583–593.
[60] Practical lock-free data structures, Computer Laboratory. University of Cambridge. Online: http://www.cl.cam.ac.uk/research/srg/netos/lock-free/, 2011 (accessed 10.01.11).
[61] K. Fraser, T. Harris, Concurrent programming without locks, ACM Trans. Comput. Syst. 25 (2007).