Association for Computing Machinery
Advancing Computing as a Science & Profession

Ubiquity, an ACM publication
August 2014

Ubiquity Symposium

The Multicore Transformation

GPUs: High Performance Accelerators for Parallel Applications

by Mark Silberstein

Editor's Introduction

Early graphical processing units (GPUs) were designed as high compute density, fixed-function processors ideally crafted to the needs of computer graphics workloads. Today, GPUs are becoming truly first-class computing elements on par with CPUs. Programming GPUs as self-sufficient general-purpose processors is not only hypothetically desirable, but feasible and efficient in practice, opening new opportunities for integration of GPUs in complex software systems.

Ubiquity, an ACM publication
August 2014

Ubiquity Symposium

The Multicore Transformation

GPUs: High Performance Accelerators for Parallel Applications

by Mark Silberstein

The last decade has seen tectonic shifts in processor architecture. Symmetric multiprocessor and multicore systems have outgrown their exclusive supercomputing niche and now dominate the processor landscape. But symmetric multi-core parallelism alone is only a short-term remedy for stagnating single-core performance. Diminishing performance gains from incremental enhancements of traditional general-purpose processors push toward heterogeneous system designs; a variety of special-purpose hardware accelerators are added alongside a CPU to accelerate specific compute-intensive functions, such as audio and video processing. Accelerators are dramatically more efficient than CPUs for their target applications. However, they are not as flexible and perform poorly on other workloads.

In this article we focus on graphical processing units (GPUs)[1], which are programmable computational accelerators that aim to accelerate a wide range of parallel applications. As their name suggests, early GPUs were designed as high compute density, fixed-function processors ideally crafted to the needs of computer graphics workloads. Over time, however, they gained increasingly general-purpose capabilities such as support for flexible control flow and random memory accesses. The GPU's throughput-optimized parallel architecture facilitated its steady performance growth; today, the raw computing capacity of a typical high-end GPU greatly exceeds that of a high-end general-purpose CPU.

Initial attempts to use GPUs for general purpose computing date back to early 2000s [1], but their broader adoption was ignited by the introduction of the NVIDIA CUDA software development environment [2]. CUDA made GPU programming much more accessible to non-graphics developers by exposing a clean general-purpose interface to the GPU's massively parallel hardware. Availability of a convenient programming environment coupled with the

[1] We use the term GPU to collectively refer to highly parallel coprocessors, including those that lack the ability to output images to a display, like Intel Xeon-Phi and NVIDIA TESLA.

impressive potential gains in compute and power efficiency served as a major trigger for tremendous development efforts to leverage GPUs in general-purpose compute-intensive applications.

Today, GPU software and hardware are approaching the "Plateau of Productivity" in a traditional technology hype cycle.[2] GPUs are found in a variety of computer systems, from supercomputers to mobile phones and tablets. Hundreds of production-grade engineering and scientific applications use GPU-accelerated components, with typical speedups of an order of magnitude over state-of-the-art parallel CPU versions,[3] but reaching three orders of magnitude acceleration in special cases [3]. OpenCL [4], a cross-platform parallel programming standard, enables code portability across multicore CPUs and GPUs from different vendors. GPU development environments have matured to provide standard debugging and profiling facilities matching the convenience of CPU application development. Finally, GPU computing is an active research area with hundreds of annually published research papers on new GPU-optimized algorithms and programming techniques expanding the boundaries of what can be done with GPU-accelerated systems today.

The goal of this article is to provide a high-level overview of the main general-purpose GPU computing concepts, assuming no expertise in computer architecture or systems programming. We discuss basic properties of the GPU architecture and programming principles, provide a few practical rules-of-thumb to estimate the benefits of using GPUs for a given algorithm, and conclude with a discussion on future trends in the evolution of GPU-accelerated systems.

GPU Computing Principles

This section provides a high-level overview of the GPU software/hardware model, highlighting the properties essential to understanding the general principles of running computations on GPUs. We focus on the concepts shared by most GPUs that support the cross-platform OpenCL standard [4]. An in-depth description of GPU programming can be found elsewhere (e.g. [5, 6]).

GPU as parallel co-processor. GPUs are fully programmable processors, but unlike CPUs they are not used to run stand-alone programs. Instead, a CPU application manages the GPU and uses it to offload specific computations. GPU code is encapsulated in parallel routines called kernels. As Figure 1 illustrates, the CPU executes the main program, which prepares the input

[2] http://en.wikipedia.org/wiki/Hype_cycle

[3] http://www.nvidia.com/object/gput applications.html

data for GPU processing, invokes the kernel on the GPU, and then obtains the results after the kernel terminates. A GPU kernel maintains its own application state, which, for example, may include GPU-optimized data structures not shared with the CPU.

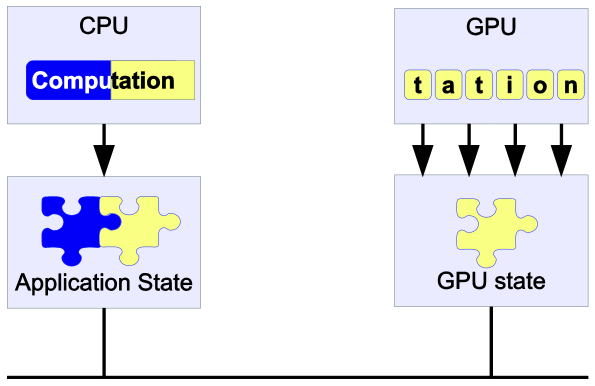

Figure 1. Offloading computations to a GPU.

A GPU kernel looks like an ordinary sequential function, but it is executed in parallel by thousands of GPU threads. The hardware supplies each thread with a unique identifier, allowing different threads to select different data and control paths. Developers may write GPU programs in plain C++/C or Fortran with only few restrictions and minor language extensions.

```
CPU:
void vector_sum(float* A, float* B,
                float* C, int n)
{
     float* gA=GPU_get_reference(A);
     float* gB=GPU_get_reference(B);
     float* gC=GPU_allocate_mem(n);

     GPU_set_num_threads(n);
     // GPU will invoke n threads
     GPU_run(vector_sum_kernel(gA,gB,gC));

     GPU_retrieve(C,gC);
}

GPU:
void vector_sum_kernel(float* gA,
                       float* gB, float*gC)
{
     int my=HardwareThreadID;
     gC[my]=gA[my]+gC[my];
}
```

Figure 2. A code sketch for a GPU-accelerated vector sum.

To gain some intuition, consider a task of computing a sum of two n-element vectors A and B. A sketch of a GPU-accelerated program for this task is shown in Figure 2. First, the CPU sets up the input in GPU memory, then invokes the GPU kernel in n parallel threads, and finally retrieves the results. The GPU kernel computes one element of the output array C in each thread.

This example illustrates two basic GPU programming principles. First, GPU threads are lightweight, and enable fine-grained parallelizm (e.g., one arithmetic operation per thread). Second, the number of threads is not limited to the number of available processors, and is often determined by the program logic, such as the total number of vector elements in this example. A GPU generally performs best when the number of threads is large, as we will explain shortly.

Note that vector sum is a purely data-parallel application with identical, completely independent subtasks. A majority of real applications are not as easily parallelizable, however, and have a more complex structure. Consider, for example, computing a dot product of two vectors. This task includes parallel reduction, which requires coordination among threads.

To understand how GPUs can support this kind of workloads we delve one level deeper into the GPU hardware and software model, and then revisit the dot product example. We note the hardware model described here deliberately simplifies various technical details for clarity, while highlighting the most important concepts.

Hierarchical hardware parallelism. GPUs are parallel processors, which expose programmers to hierarchically structured hardware parallelism, as depicted in Figure 3. At the highest level, GPUs are similar to CPU shared-memory multicores. GPUs support coarse-grain task-level parallelism via concurrent execution of different tasks on different powerful cores.

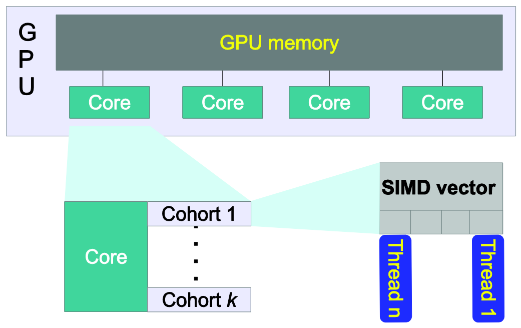

Figure 3. Hierarchical hardware parallelism in a GPU.

A GPU kernel comprises multiple individual threads. A GPU thread forms a basic sequential unit of execution. At the lowest level, the hardware scheduler manages threads in small cohorts.[4] All threads in one cohort are invoked at the same time and executed in lockstep, in a SIMD (Single Instruction Multiple Data) fashion, enabling fine-grained data-parallel execution similar to the execution of vector instructions on a CPU.

The main architectural property in which CPU and GPU architectures differ is the way GPUs execute parallel code on each core. Unlike CPU threads, which usually exclusively occupy one CPU core, multiple cohorts are concurrently scheduled to run on each GPU core. The hardware scheduler interleaves instructions from alternate cohorts on each core to maximize hardware utilization when threads in a cohort get stalled, for example while waiting on a slow memory access. The execution state of all concurrently scheduled threads is kept in the core memory to allow zero-time switching between them. This type of parallelism, sometimes called thread-level parallelism, or fine-grained multithreading, is essential to achieving high hardware utilization and performance in GPUs.

Most GPUs share the same structure of the hierarchical hardware parallelism, but differ with respect to the number of threads per cohort, cohorts per core, or cores per processor. For example, the latest Kepler processor in NVIDIA GTX Titan GPU features 32 threads per cohort (called warp), 14 cores, and up to 64 cohorts per core, so it can concurrently execute up to 28,672 threads in total.

One key observation, however, is GPU and CPU threads have very different properties, and thus are used in different ways. In particular, a single GPU thread is slow and performs poorly in branching code. For example, multiplying a vector by a scalar in a single thread is about two orders of magnitude slower on NVIDIA C2070 TESLA GPU than on Xeon L5630 CPU core. Hence, GPUs must expose their massively parallel hardware to the programmer and enable invocation of thousands of threads to achieve high parallel throughput.

GPU programming model. The programming model closely follows the hierarchy of parallelism in the hardware. All threads in a GPU kernel are subdivided into equal-sized static workgroups of up to a few hundred threads. Threads within each workgroup may share state and synchronize efficiently, enabling coordinated data processing. A workgroup is a course-grain unit of execution that matches the task-level parallelism support in the hardware: All threads in a single workgroup are scheduled to execute at once on a single core.

[4] Cohort is a vendor independent name for NVIDIA's warp and AMD's wavefront.

It is helpful to think of a GPU kernel as a stream of independent workgroups executed by hardware in an arbitrary, non-deterministic order. An application enqueues all workgroups comprising a kernel into a global queue on a GPU. The number of workgroups in each kernel ranges from tens to hundreds, and typically exceeds the number of cores, leaving some workgroups waiting idle in the hardware queue until some core becomes available. Oversubscribing the cores facilitates load balancing and portability across GPU systems with different numbers of cores.

Example: Dot product. A parallel algorithm for computing dot product of two vectors uses a binary reduction tree to compute the sum of n elements in $O(log(n))$ phases. Unfortunately the GPU programming model does not generally support barriers across all the GPU threads in a kernel, and thus requires each phase to be invoked in a separate kernel. To reduce the number of costly GPU invocations we can leverage efficient synchronization primitives for threads within a workgroup. We partition the input into chunks and compute each chunk independently by performing multiple parallel reductions in parallel in each workgroup. The intermediate results are then processed in a separate GPU kernel.

Figure 4 shows a code sketch of the per-chunk reduction.[5] This implementation illustrates a typical workgroup-centric GPU programming pattern: The code expresses a parallel algorithm for the threads of a single workgroup, rather than a sequential algorithm for a single GPU thread. The GPU programming model provides a few hardware-optimized primitives to facilitate such a workgroup-centric code structure. Here we use hierarchical thread indexing (`wgid` and `tid` variables), a fast on-core memory shared only among the threads of the same workgroup (`local_res` array), and an efficient barrier (`BARRIER` function) for intra-workgroup synchronization. The number of threads per workgroup and the number of workgroups per kernel is specified dynamically in CPU code when the GPU kernel is invoked (which is not shown here).

[5] We refer an interested reader to an excellent tutorial by Mark Harris on optimizing parallel reduction for NVIDIA GPUs.

```
void vector_dotproduct_kernel(float* gA,
                              float* gB, float* gOut)
{
     // allocate local core memory for WG_SIZE elements,
     // one per workgroup thread
     _local_ float local_res[WG_SIZE];

     int tid=LocalThreadID;
     int wgid=HardwareWorkgroupID;

     int offset=wgid*WG_SIZE+tid;

     local_res[tid]=gA[offset]*gB[offset];

     BARRIER(); // wait for all products

     for(int i=WG_SIZE/2; i>0; i/=2)
     {
          if (tid<i) local_res[tid]+=local_res[i+tid];
          BARRIER(); // wait for all partial sums
     }
     if (tid==0) gOut[wgid]=local_res[0];
}
```

Figure 4. A code sketch for a GPU-accelerated vector inner product.

GPU-CPU communication. The cost of CPU-GPU communications and data transfers differs for discrete and hybrid GPUs. Discrete GPUs are peripheral devices connected to the main CPU via a PCIe bus. Discrete GPUs provide the highest computing capacity, and thus are broadly used in high-performance computing systems. Discrete GPUs have their own physical memory, which cannot be directly referenced by CPU programs. As a result, an input to a GPU kernel is staged into GPU memory before the kernel is invoked, and the output is transferred back to allow access from a CPU program. Unfortunately, the PCIe bandwidth to the main host memory is an order of magnitude lower—more than 20× in NVIDIA K20X GPU, for example—than the GPU's bandwidth to its local memory. Therefore, CPU-GPU data transfers might significantly slow down the application, making minimization of communication overheads among the most important optimization goals.

A new generation of hybrid processors such as NVIDIA Tegra K1, Intel Broadwell, Qualcomm Snapdragon, and AMD Kaveri, integrate a CPU and a GPU on the same die, and provide both processors with access to a shared memory. It then becomes possible to share data between CPU and GPU code, not only eliminating costly data copies but also simplifying GPU memory management code. For example, calls GPU_get_reference and GPU_retrieve in Figure 2 become redundant.[6]

[6] Recently released CUDA 6.0 provides similar support for automatic data management for discrete GPUs, but related transfer overheads still remain.

Hybrid GPUs significantly extend the computing capabilities of embedded and low-end computer systems, enabling applications that would be too slow to run on their peer CPUs alone. From the absolute performance perspective, however, discrete GPUs provide substantially higher computational capacity than any currently available hybrid GPU. Therefore they achieve much higher performance for their target applications despite CPU-GPU communication costs, and thus remain the processor of choice for high-performance computing systems.

Using GPUs without GPU programming. Recent developments make it significantly easier to accelerate computations on GPUs without GPU code development. Popular numerical computing platforms such as Matlab and Mathematica offload some of their functions to GPUs, and provide the convenience of calling user-supplied GPU functions from within the framework. There are comprehensive libraries of GPU-accelerated algorithms and data structures like Thrust, and domain-specific libraries such as cuBLAS and CUDA NPP. New compiler frameworks, e.g. OpenACC, support special source annotations that allow them to automatically parallelize and offload user-annotated code to GPUs. Finally, there is growing support for higher-level languages such as Python and Java, complementing traditional C and Fortran GPU kernel development.

Designing GPU-Accelerated Applications

GPUs hold tremendous performance potential and have already shown impressive speedups in many applications. Yet, realizing that potential might be challenging. In this section we offer a few simple guidelines to help designing GPU-accelerated applications.

Accelerate performance hotspots. Usually GPUs are used to run only a subset of application code. It is thus instructive to estimate the contribution π of the code intended for acceleration in the application execution time, to guarantee that making it faster will have a tangible effect on the overall performance. Following Amdahl's law, the maximum application speedup is limited to $\frac{1}{1-\pi}$, so porting to a GPU a routine consuming 99 percent of the total execution time ($\pi = 0.99$), for example, has 10× higher acceleration potential than porting a routine consuming 90 percent alone.

Move more computations to GPUs. GPUs are not as efficient as CPUs when running control flow-intensive workloads. However, longer kernels with some control flow ported to a GPU are often preferable over shorter kernels controlled from CPU code. In iterative algorithms, for example, offloading to the GPU the computations performed in every iteration may be less efficient than moving the whole loop into the kernel. The latter design hides GPU kernel invocation overheads, avoids thread synchronization effects when the kernel terminates, and allows for keeping intermediate data in hardware caches.

Stay inside GPU memory. Memory capacity of discrete GPUs reaches 12GB in high-end systems. However if the working set of a GPU kernel is too large, as is often the case for stream processing applications, for example, the performance becomes limited by the throughput of CPU-GPU data transfers. Achieving performance improvement on GPUs in such applications is possible for compute-intensive workloads with high compute-to-memory access ratio.

Conform to hardware parallelism hierarchy. Ample parallelism in the computing algorithm is an obvious requirement for successful acceleration on GPUs. However having thousands of independent tasks alone might not be sufficient to achieve high performance. It is crucial to design a parallel algorithm, which maps well onto the GPU hierarchical hardware parallelism. For example, it is best to employ data-parallel computations in the threads of the same workgroup. Using these threads to run unrelated tasks is likely to result in much lower performance.

Consider power efficiency. A typical high-end discrete GPU consumes about 200W of power, which is about twice the consumption of a high-end CPU. Therefore, achieving power efficient execution in GPU-accelerated systems implies that the acceleration speedups must exceed 2× if only a GPU is used. In practice, employing both a CPU and a GPU may achieve better power efficiency than running on each processor alone [7].

Understand memory performance. To feed their voracious appetites for data, high-end GPUs employ a massively parallel memory interface, which offers high bandwidth for local access by GPU code. The GPU memory subsystem often becomes the main bottleneck, however, and it is useful to estimate application performance limits imposed by the memory demands of the parallel algorithm itself.

Consider an algorithm, which performs c operations for every m memory accesses, i.e., its compute-to-memory access ratio $A = \frac{c}{m}$. Assuming that memory accesses dominate the execution time and are completely overlapped with arithmetic operations in GPUs, the algorithm performance cannot exceed $A \times B$, where B is the GPU local memory bandwidth. For example, the vector sum kernel in Figure 2 performs 1 addition per 3 4-byte memory operations (2 reads and 1 write) for each output, hence $A = \frac{1}{12}$. If we use NVIDIA GeForce Titan GPU with B =228 GB/s, the maximum performance is at most 19 GFLOP/s,[7] which is 0.5% of the GPU's raw capacity.

Estimating memory performance of an algorithm may help identify its inherent performance limits and guides the design toward cache-friendly parallel algorithms which optimize memory reuse and leverage fast on-core memory, as we showed in our work [8].

Future of GPUs

Modern GPUs are fully-programmable parallel processors that no longer deserve their reputation as an esoteric technology for computing enthusiasts. With the growing deployment of hybrid CPU-GPU processors in particular in mobile computing platforms, programmable general-purpose GPUs are becoming an integral part of modern computer systems, crucially important for achieving high application performance and power efficiency.

Both discrete and hybrid GPUs are rapidly evolving, becoming faster, more programmable and more versatile with each generation. In particular, newer architectures enable discrete GPUs to execute functions that previously required CPU-side code. For example, NVIDIA Kepler GPUs support "nested parallelism" in hardware, allowing invocation of new GPU kernels from GPU code without stopping the running kernel first. Similarly, discrete GPUs now provide direct access to peripheral devices, such as storage and network adapters, eliminating the CPU from the hardware data path. Hybrid GPUs, in contrast, facilitate tighter coupling with same-dye peer CPUs via shared virtual memory, and also offer new features like hardware support for CPU function calls from GPUs.

We believe discrete and integrated GPUs will continue to co-exist for years to come. They embody very different tradeoffs between power consumption, production costs and system performance, and thus serve different application domains. In particular, discrete GPUs have consistently shown performance and power efficiency growth over the past few hardware generations. This progress is due to the hardware design flexibility enabled by a modular system organization in which a GPU resides on a stand-alone add-on peripheral device. Specifically, the aggressive throughput-optimized hardware designs of discrete GPUs used as

[7] FLOP – floating point operation

computational accelerators in data centers and supercomputers, heavily rely on a fully-dedicated multi-billion transistor budget, tight integration with specialized high-throughput memory, and increased thermal design power (TDP). In contrast, hybrid GPUs are allocated only a small fraction of the silicon and power resources available to discrete processors, and thus offer an order of magnitude lower computing capacity and memory bandwidth.

Future high throughput processors [9] are expected to enable more efficient sequential processing, and some indications of this trend can already be observed. For example, the AMD Graphics Core Next 1.1 used in all modern AMD GPUs, contains a scalar processing unit. In addition, NVIDIA-IBM recently announced a partnership that aims to integrate NVIDIA GPUs and IBM Power CPUs targeting data center environments.

These trends indicate GPUs are departing from their traditional role as slave co-processors and are becoming truly first-class computing elements on par with CPUs. We envision a system architecture where GPUs will run complete self-contained programs, and will have full access to standard operating system services of their host system just like CPUs. For example, GPUs will be able to directly read files in the host file system, or send messages over network, without CPU involvement. In our current work [10] we show the first steps toward realizing this vision in today's systems, by enabling GPUs access host files via standard API directly from GPU code. Our findings suggest programming GPUs as self-sufficient general-purpose processors is not only hypothetically desirable, but feasible and efficient in practice, opening new opportunities for integration of GPUs in complex software systems.

References

[1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE* 96, 5 (2008), 879–899.

[2] NVIDIA. NVIDIA CUDA Programming Guide. Retrieved 2013.

[3] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep Learning with COTS HPC Systems. In *Proceedings of the 30th International Conference on Machine Learning* (ICML-13). 1337–1345, 2013.

[4] Khronos Group. The OpenCL Specification. Retreived 2013.

[5] D. B. Kirk and W. H. Wen-mei. *Programming Massively Parallel Processors: A hands-on approach*. Morgan Kaufmann, New York, 2010.

[6] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg. *OpenCL Programming Guide*. Pearson Education, 2011.

[7] M. Silberstein and N. Maruyama. An Exact Algorithm for Energy-Efficient Acceleration of Task Trees on CPU/GPU Architectures. In *Proc. of the International Conference on Systems and Storage* (SYSTOR'04). ACM, 2011.

[8] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient Computation of Sum-Products on GPUs Through Software-Managed Cache. In *Proc. of the International Conference on Supercomputing* (ICS'08). 2008.

[9] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *Micro, IEEE* 31, 5 (2011), 7–17.

[10] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: integrating file systems with GPUs. In *Proc. of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS'18). ACM, 2013.

About the Author

Mark Silberstein is an Assistant Professor in the department of Electrical Engineering at the Technion--Israel Institute of Technology. Mark holds a Ph.D. in computer science from the Technion. Prior to joining the Technion, Mark was a post-doctoral researcher at the University of Texas at Austin in the operating systems and architectures group led by Prof. Emmett Witchel. Mark's research aims to alleviate the complexity of integrating computational accelerators in large systems via novel accelerator-centric operating system design, where accelerators may natively access I/O services like files or network without using CPUs.

DOI: 10.1145/2618401