Editors: Volodymyr Kindratenko, kindr@ncsa.uiuc.edu
Pedro Trancoso, pedro@cs.ucy.ac.cy

SOLVING COMPUTATIONAL PROBLEMS WITH GPU COMPUTING

By Jonathan Cohen and Michael Garland

Modern GPUs are massively parallel microprocessors that can deliver very high performance for the parallel computations common in science and engineering.

As their name suggests, graphics processing units (GPUs) are microprocessors that have evolved to meet the needs of real-time computer graphics. Broadly speaking, real-time graphics exhibits tremendous inherent parallelism and places a premium on the total amount of work that can be completed in a single 1/30-second frame time. Consequently, developers have always designed GPUs with a high degree of hardware parallelism.

Fifteen years ago, GPUs were fixed-function hardware accelerators for the graphics pipeline rendering model defined by APIs such as OpenGL and Direct3D. Over time, GPUs evolved to incorporate progressively more flexible and programmable components, culminating in the vertex and pixel shaders on which modern graphics applications are based. As GPUs became increasingly programmable, many researchers explored ways to harness their parallel-processing potential by mapping relatively general-purpose computations onto the restricted paradigms offered by graphics APIs.[1]

While this line of investigation yielded some intriguing results, it was often difficult and severely constrained by graphics API restrictions. The CUDA architecture for GPU computing—which is supported by all modern Nvidia GPUs, starting with the GeForce 8800 GTX—provides a far more powerful platform than these earlier attempts at general-purpose programming via the graphics system. CUDA-capable GPUs are fully programmable parallel processors capable of executing programs written in full-featured languages such as C. Furthermore, for parallel computational problems, it's not uncommon to achieve running-time improvements with CUDA on the order of 10 to 100 times that of optimized multicore implementations.

CUDA Architecture for GPU Computing

Modern Nvidia GPUs are fully programmable parallel machines[2,3] built around an array of multithreaded multiprocessors. Each multiprocessor can support up to 1,024 coresident threads. The hardware handles all thread management and scheduling, with effectively no overhead. A high-end GPU such as the Tesla C1060 contains 30 multiprocessors, thus supporting a total of 30,720 coresident threads (see Figure 1). This degree of parallelism is orders of magnitude greater than the number of parallel threads that multicore CPUs are designed for and reflects the most essential architectural difference between GPU and CPU processors. Whereas CPUs are primarily designed to execute individual sequential threads as quickly as possible, GPUs are primarily designed to maximize the rate at which many thousands of parallel threads can complete their work. And, unlike more traditional microprocessors, GPUs rely on multithreading, as opposed to a cache, to hide the latency of transactions with external memory. It's therefore necessary to design algorithms that create enough parallel work to keep the machine fully utilized.

Multiprocessor Architecture

As Figure 1 shows, a single multiprocessor is physically composed of eight scalar processor cores (SPs) that execute the resident threads' instructions. The cores support 32-bit precision integer and IEEE floating-point operations. They're also augmented with a shared unit (DP) for performing 64-bit IEEE floating-point operations and two special function units (SFUs) for accelerating transcendentals.

Each multiprocessor is equipped with a 64-Kbyte register file, for a total of 16,384 32-bit registers. From this pool, each thread gets its own dedicated set of registers, which lets the hardware freely schedule runnable threads without needing to save/restore any thread state. There's an additional 16 Kbytes of low-latency shared on-chip memory, which is similar in speed to a typical L1 cache. It is, however, an explicitly managed RAM available to

 1521-9615/09/$26.00 © 2009 IEEE

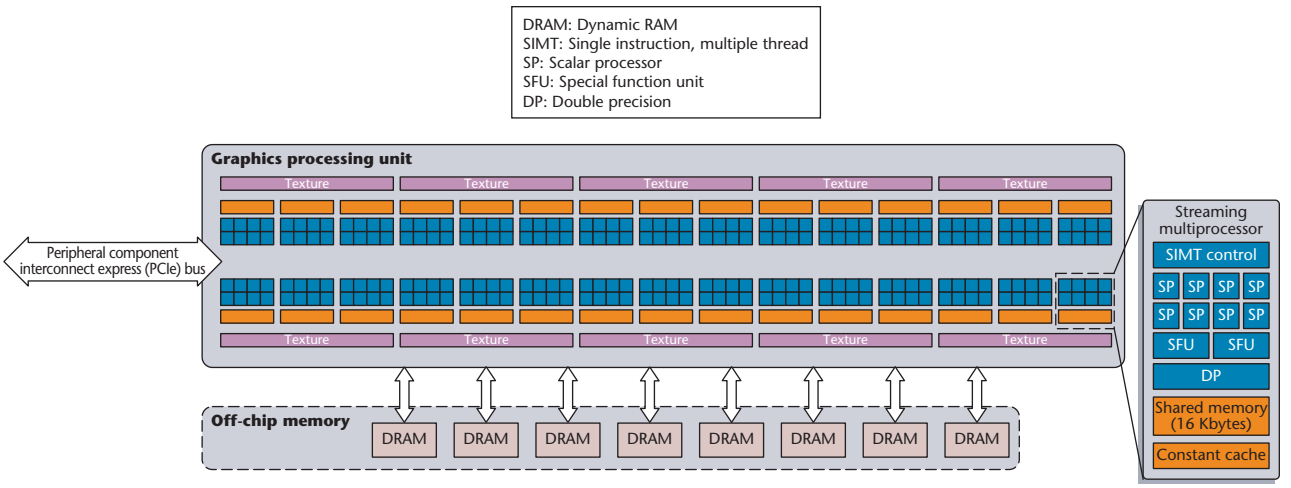

Figure 1. Block diagram of a high-end GPU. This Tesla C1060 contains 240 cores organized in 30 multiprocessors.

all threads. A constant cache offers additional memory and can accelerate accesses to read-only constant data. Finally, threads can access texture units, which can perform various image filtering operations; threads can also use the texture units' caches to aggregate repeated accesses to read-only arrays. The GPU connects to other system components via the PCI Express (PCIe) bus, which provides the channel for transferring data between the system memory and the GPU's on-board memory.

Parallel Execution

CUDA's parallel execution model lets a sequential host program running on the CPU launch parallel computations, or *kernels*, that execute on a parallel device—the GPU. Kernels are a blocked *single program*, *multiple data* (SPMD) parallel computation model; a kernel executes a single sequential program across many parallel threads, which are further organized into a grid of thread blocks.

A single block's threads are guaranteed to be resident on a single multiprocessor. This requires that we limit their size. While executing a kernel might launch millions of threads over its lifetime, thread blocks are currently limited to a maximum of 512 threads. CUDA's hierarchical organization into sequential threads, with multiple threads per block, and multiple blocks per kernel, defines the hierarchy of scopes at which threads can share memory and synchronize. Each running thread naturally has its own private variables, which are typically stored in registers. A single block's threads share a per-block memory space that is backed by their host multiprocessor's on-chip memory. This on-chip memory, which is nearly as fast as registers, provides for efficient interthread communication. Furthermore, the multiprocessor provides a barrier instruction that permits extremely lightweight synchronization between a block's threads. Finally, all threads executing on the GPU can access a global memory space backed by its on-board DRAM, and there's an implicit barrier synchronization between successive kernels.

To efficiently manage many thousands of active threads, Nvidia GPUs use a *single instruction*, *multiple thread* (SIMT) execution model. A thread block's threads are grouped into 32-thread warps in order of their thread index. In a given clock cycle, all of a warp's threads can fetch and issue a single instruction.

The SIMT architecture offers substantial efficiencies on typical data-parallel codes. Unlike *single instruction*, *multiple data* (SIMD) vector instruction extensions such as SSE, SIMT thread execution is largely invisible to the CUDA programmer because the processor transparently handles the case in which different threads of a warp want to follow divergent execution paths. However, much like cache line sizes on traditional CPUs, programmers can ignore it when designing for correctness but must consider it carefully when designing for peak performance.

Numerical Computing with CUDA

To help illustrate the structure of parallel programs that exploit the CUDA architecture, we'll use an example implementation of a simple numerical program in C. While our examples are written in C, and C/C++ is the language supported by the Nvidia CUDA Toolkit compiler, CUDA is not intrinsically linked to C. The CUDA architecture defines a fairly typical scalar assembly instruction set; a suitable compiler can compile essentially any standard sequential programming language to this instruction set.

Suppose that we want to solve the differential equation $u'' = 0$, where $u(0) = 1{,}000$; $u(n-1) = 0$ over a one-dimensional domain discretized at the n integer locations ranging from $i = 0$ to $i = n - 1$. We can solve this equation using a simple iterative solver that

```
void solve(float *solution, int n)
{
    int B = 128;              // threads per block
    int P = (n + B-1) / B;    // P = ceil(n/B)

    float *u, *du, t;

    cudaMalloc((void**)&u,  n*sizeof(float));
    cudaMalloc((void**)&du, n*sizeof(float));

    initialize<<<P,B>>>(u, n);

    do {
        compute_update<<<P,B>>>(u, du, n);
        apply_update<<<P,B>>>(u, 0.5, du, n);
        t = cublasSnrm2(n, du, 1);
    } while( t>1 );

    cudaMemcpy(solution, u, n*sizeof(float),
               cudaMemcpyDeviceToHost);

    cudaFree(u);
    cudaFree(du);
}
```

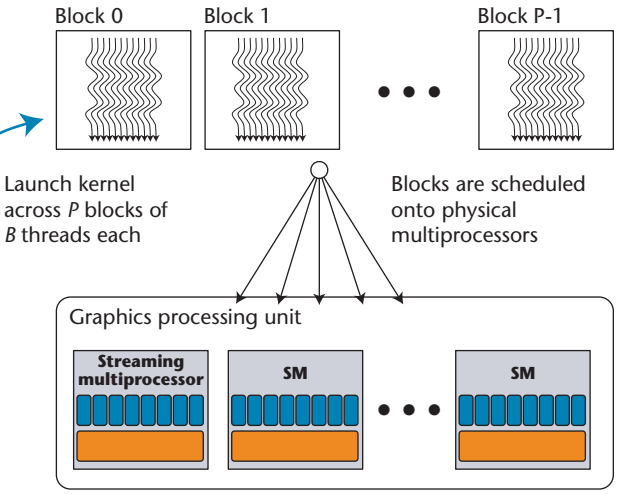

Figure 2. A host program launches parallel kernels on the device in the form of many parallel thread blocks. The program consists of three basic parallel components: initialize u, compute the Δu to update the solution, and apply the update $u \leftarrow u + \lambda \Delta u$.

```
__global__ void initialize(float *u, int n)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;

    if( i==0 )      u[i] = 1000.0;
    else if( i<n )  u[i] = 0.0;
}

__global__
void apply_update(float *u,  float lambda,
                  float *du, int n)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;

    if( i>0 && i<n-1 )
        u[i] = u[i] + lambda*du[i];
}
```

```
__global__
void compute_update(float *u, float *du, int n)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;

    if( i>0 && i<n-1 )
    {
        float u_left  = u[i-1],
              u_i     = u[i],
              u_right = u[i+1];

        du[i] = (u_left + u_right)/2 - u_i;
    }
}
```

Figure 3. The source for the solver's three parallel steps. The `__global__` specifier indicates that the function is a kernel entry point. Further, each kernel thread receives the same arguments when run, each block is given a unique integer identifier (`blockIdx`), and each thread within a block with dimensions `blockDim` is given a unique identifier `threadIdx`.

initializes an n-element vector representing the values of u to be

$$u_i = \begin{cases} 1{,}000 & \text{if} \quad i = 0 \\ 0 & \text{otherwise} \end{cases}$$

and iteratively applies the update:

$$u_i \leftarrow u_i + \lambda \Delta u_i$$
$$\text{where} \quad \Delta u_i = (u_{i+1} + u_{i-1})/2 - u_i.$$

This is obviously a toy problem. The differential equation has a known analytic solution, namely the line through (0,1000) and (n − 1,0). However, it's a useful example because it exhibits the same basic structure as stencil-based solvers for differential equations without analytic solutions. In particular, it's the low-dimensional analog of a solver for higher-dimension Poisson problems such as the one we explore below. Figure 2 shows the solver's source code. This is a standard sequential program and runs on the CPU. It consists of three basic components, each of which is clearly parallel:

- initialize u,
- compute the Δu with which to update the solution, and
- apply the update $u \leftarrow u + \lambda \Delta u$.

The solver iterates as long as $\|\Delta u\| > 1$. Figure 3 shows the source for the solver's

three parallel steps. To calculate $||\Delta u||$ we call the `cublasSnrm2()` procedure, which is a standard basic linear algebra subprogram (BLAS) routine provided by the CUDA Toolkit's CUBLAS library.

In a sequential implementation, each of the solver's three basic steps would be a loop over the elements of the vector u. For instance, the update step might look like this:

```
void sequential_update
  (float *u, float lambda,
    float *du, int n)
{
  for(int i=1; i<n-1; ++i)
      u[i] = u[i]
        + lambda*du[i];
}
```

Notice that every loop iteration is entirely independent of the other iterations. In other words, we could perform each iteration in parallel with all the others.

We can easily parallelize such loops by assigning one thread to process each element of u, thus assigning each independent iteration to a separate parallel thread. Figure 3 shows the resulting code. The `__global__` specifier indicates that this function is a kernel entry point. Also,

- each kernel thread receives the same arguments when run,
- each block is given a unique integer identifier (`blockIdx`), and
- each thread within a block with dimensions `blockDim` is given a unique identifier `threadIdx`.

For convenience when working with different spatial dimensions, CUDA lets these indices have up to three dimensions. In our example, we're using one-dimensional indexing, and hence use only the indices' .x component. From these unique indices, we can compute an index i for the element that each thread will process, which corresponds directly to the sequential implementation's iteration variable. The sequential version's loop bounds become a conditional in the kernel body, and the loop's body is identical.

The CUDA C compiler provides an extended function-call syntax for launching kernels. The expression `initialize<<<P,B>>>(u,n)` will launch the kernel `initialize` across P thread blocks of B threads each. Our sample implementation makes the somewhat arbitrary choice of using 128 threads per block and then computes the minimum number $P = \lceil n/B \rceil$ of 128-thread blocks necessary to guarantee that a total of at least n threads will be launched. It also uses CUDA runtime library calls to allocate memory on the GPU device (`cudaMalloc`), copy data between host memory and device memory (`cudaMemcpy`), and free GPU memory (`cudaFree`).

Table 1. Running time per simulation step of equivalent codes running a 3D Rayleigh-Bénard convection problem.

Resolution	CUDA (milliseconds)*	Fortran (milliseconds)**	Speedup
$64^2 \times 32$	24	47	2.0x
$128^2 \times 64$	79	327	5.3x
$256^2 \times 128$	498	4070	8.2x
$384^2 \times 192$	1,616	13,670	8.5x

** for a single Quadro FX5800 running on an Intel Core2 Duo E8500 at 3.17 GHz*

*** for an eight-core dual socket Intel Xeon E5420 at 2.5 GHz*

Case Study: Computational Fluid Dynamics

We've been developing a CUDA-based computational fluid dynamics (CFD) code to explore issues related to designing complex numerical software in CUDA.[4] This code solves the 3D incompressible Navier-Stokes equations using methods that are globally second-order accurate in time and space. We use a finite volume discretization over a staggered grid and use a projection method to enforce incompressibility. The pressure solver uses a multigrid scheme with a red-black Gauss-Seidel smoother, and all other discretizations use second-order centered schemes. For simplicity, our code uses structured Cartesian grids, but CUDA can also support the irregular, indirect access patterns needed for sparse matrix routines.[5]

Although it requires that we rethink data layout and algorithm choice, our CFD code demonstrates that CUDA can offer dramatic performance improvement over even a high-end CPU. We've validated and benchmarked our code and found that it executes approximately eight times faster than a comparable Fortran code running on an eight-core CPU. This translates into well over an order of magnitude speedup versus a single CPU code. Table 1 shows timing results for a typical problem.

Our CFD code contains several kernels that perform operations over 3D data arrays, reading from some number of arrays in GPU memory and writing the results into other arrays. Example routines include computing the divergence operator over a vector field to generate a Poisson equation's right-hand side or calculating an advection term using an upwind finite difference scheme.

In each of these kernels, we assign a single CUDA thread to each array element. This is a common pattern in most CUDA programs, and stands in stark contrast to the way applications are typically written for multicore CPUs. Whereas a multicore application might launch just four to eight threads—with each thread computing a term over a large 3D array region—our CUDA kernels launch literally millions of threads for the larger problem sizes.

```
struct Array2D {
  float *ptr;
  int nx, ny;
};

__device__ float &lookup(Array2D array, int i, int j)
{
    i = (i + array.nx) % array.nx;
    j = (j + array.ny) % array.ny;
    return array.ptr[i + j * array.nx];
}

__global__ void laplacian(Array2D dudt, Array2D u)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;

    if (i < dudt.nx && j < dudt.ny)
        lookup(dudt,i,j,k) = -4.0 * lookup(u,i,j) +
            lookup(u,i+1,j) + lookup(u,i-1,j) +
            lookup(u,i,j+1) + lookup(u,i,j-1);
}
```

Figure 4. Code example: a Laplacian operator with periodic boundary conditions. All reads from `u` and writes to `dudt` will be well coalesced.

This represents the biggest qualitative difference in writing CUDA programs over more traditional multithreaded applications.

In a second-order finite volume code such as ours, almost all routines are limited by memory bandwidth rather than by the chip's floating-point performance. The GPU memory system that CUDA exposes is designed for parallel access to large data sets. When a warp's threads load from or store to memory, the hardware can *coalesce* accesses to nearby locations into a single memory transaction. In our CFD code, arranging memory accesses to maximize coalescing can result in an order of magnitude speed improvement. This is similar in many ways to optimizing data layout for best cache performance on a multicore CPU. However, because the GPU provides no cache on memory operations, memory accesses need be contiguous only across "nearby" threads at a single point in time, rather than contiguous for a single thread at nearby points in time.

Coalescing considerations come into play when considering data layout. For example, on a sequential scalar processor, it might make sense to store a vector field in an interleaved format to improve cache performance. In interleaved storage, a single vector's x, y, and z components are stored in memory with x, y, and z for the first cell, then x, y, and z for the next cell, and so on. In CUDA, however, if all threads access the x component first, then y, then z, interleaved storage would result in all threads reading memory locations three spaces apart, which would degrade coalescing by a factor of three. A better approach would be to store all of the x values together in one array, all of the y values in another array, and all of the z values in a third. This type of "structure of arrays"—rather than "array of structures"—is typical of how to properly lay out memory for a parallel architecture such as CUDA, and mirrors the layouts necessary to take advantage of vectorized memory operations such as those offered by SSE.

As an example of what our code looks like, consider the case of a Laplacian operator with periodic boundary conditions. We'll describe this case in two dimensions for simplicity, but the 3D case is similar. The source and destination 2D arrays are laid out in `i`-major order, so in an array with dimensions `nx` × `ny`, element `i,j` is placed at offset `i + j * nx` from the beginning of the array. For good coalescing, we need only ensure that the mapping from `threadIdx.x` to `i` is sequential, which is easy to achieve. As the source code in Figure 4 shows, all reads from `u` and writes to `dudt` will be well coalesced, resulting in excellent performance.

The latest CUDA Toolkit—which includes the C compiler, debugger, and profiler—is available for free online at www.nvidia.com/cuda. The site also provides a directory of some of the many ongoing academic and commercial projects based on the CUDA architecture for parallel GPU computing, as well as a collection of documentation and example programs. A programming tools directory is at www.nvidia.com/object/tesla_software.html. Among the currently available tools are

- numerical libraries for linear algebra (BLAS) and Fourier transforms (FFT),
- image processing libraries,
- Matlab and Mathematica plugins, and
- additional language support including Fortran, Python, and Java.

We've provided here a very brief overview of parallel computing on GPUs, but John Nickolls and his colleagues provide more details on the CUDA programming model,[3] and Erik Lindholm and his colleagues describe the GPU architecture in depth elsewhere.[2] Other published surveys of GPU application experience offer

further insight into solving various computational problems with GPU computing.[6,7]

References

1. J.D. Owens et al., "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, 2007, pp. 80–113.
2. E. Lindholm et al., "Nvidia Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, 2008, pp. 39–55.
3. J. Nickolls et al., "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, 2008, pp. 40–53.
4. J.M. Cohen and M.J. Molemaker, *A Fast Double Precision CFD Code Using CUDA*, tech. report NVR-2009-001, Nvidia, May 2009.
5. N. Bell and M. Garland, *Efficient Sparse Matrix-Vector Multiplication on CUDA*, tech. report NVR-2008-004, Nvidia, Dec. 2008.
6. M. Garland et al., "Parallel Computing Experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, 2008, pp. 13–27.
7. J.D. Owens et al., "GPU Computing," *Proc. IEEE*, vol. 96, no. 5, 2008, pp. 879–899.

Jonathan Cohen is a research scientist at Nvidia, where he develops methods for using the company's massively parallel GPUs for scientific computing and real-time physical simulation. He previously worked in the feature film visual effects industry, and was awarded a Technical Academy Award in 2007 for his work in fluid simulation for visual effects. Cohen has a BS in mathematics and computer science from Brown University. Contact him at jocohen@nvidia.com.

Michael Garland is a research scientist at Nvidia and an adjunct professor in the Department of Computer Science at the University of Illinois, Urbana-Champaign. His research interests include computer graphics and visualization, geometric algorithms, and parallel algorithms and programming models. Garland has a PhD in computer science from Carnegie Mellon University. Contact him at mgarland@nvidia.com.