CUDA: SCALABLE PARALLEL PROGRAMMING FOR HIGH-PERFORMANCE SCIENTIFIC COMPUTING

David Luebke

NVIDIA Corporation

ABSTRACT

Graphics processing units (GPUs) originally designed for computer video cards have emerged as the most powerful chip in a high-performance workstation. Unlike multicore CPU architectures, which currently ship with two or four cores, GPU architectures are "manycore" with hundreds of cores capable of running thousands of threads in parallel. NVIDIA's CUDA is a co-evolved hardware-software architecture that enables high-performance computing developers to harness the tremendous computational power and memory bandwidth of the GPU in a familiar programming environment – the C programming language. We describe the CUDA programming model and motivate its use in the biomedical imaging community.

1. MOTIVATION

GPU computing, or the use of graphics processors for general-purpose computing, began in earnest several years ago [1]. Work to date has included much promising research in the biomedical domain (see for example [2-5]). However, this research initially involved programming the GPU via a graphics language, which limited its flexibility and was arcane for non-graphics experts. NVIDIA's CUDA platform changed that, providing a massively multithreaded general-purpose architecture with up to 128 processor cores and thousands of threads in flight, programmable in C and capable of hundreds of billions of floating-point operations each second [6]. CUDA runs on all current NVIDIA GPUs, including the HPC-oriented Tesla product line. The ubiquitous nature of these GPUs (over 50 million CUDA-capable boards have been sold as of this writing) makes them a compelling platform for accelerating high-performance computing (HPC) applications.

How much can GPU computing speed up a real-world science or engineering code? Researchers and companies are achieving speedups ranging from 10× to 100× (and sometimes more!) by using CUDA, across domains ranging from computational chemistry [7], to astrophysics [8], to CT [5] and MRI [9], to gene sequencing [10]. Examples from biomedical imaging include work at Friedrich-Alexander-Universität Erlangen-Nürnberg to accelerate CT reconstruction on CUDA using the FDK algorithm [11] and work at the University of Illinois that accelerates advanced MRI reconstruction techniques using CUDA [12]. The latter focuses on reconstruction for non-Cartesian scan paths, which reduces image-space error from 45% for conventional reconstruction to 12% but has been considered computationally infeasible in practice. The approach runs 13x faster on an NVIDIA Quadro FX5600 GPU than on an Intel Core 2 Extreme quad-core CPU, achieving reconstruction times of under 2 minutes for a 128^3 volume.

Speedups of this magnitude can change science. In a medical setting, order-of-magnitude speedups can cause a phase change in clinical practice, for example by moving an analysis from the lab with a turnaround time of days, to the exam room with a turnaround of minutes. Another phase change occurs in biomedical imaging when a technique moves from evaluative (e.g. checking afterwards to see if the stint was emplaced properly) to interactive (using imaging throughout to guide the emplacement procedure). In short, the potential to greatly accelerate computational techniques opens exciting avenues for biomedical imaging research. This paper gives a high-level survey of CUDA concepts, an entry point for interested developers, and a "feel" for CUDA programming.

2. THE CUDA PROGRAMMING MODEL

The fundamental strength of the GPU is its extremely parallel nature. The CUDA programming model allows developers to exploit that parallelism by writing natural, straightforward C code that will then run in thousands or millions of parallel invocations, or *threads*. Matrix addition will serve as a simple example. To add two N×N matrices on the CPU in C, one would write a doubly-nested *for* loop:

```
// add 2 matrices on the CPU:
void addMatrix(float *a, float *b,
               float *c, int N)
{
  int i, j, index;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      index = i + j * N;
      c[index]=a[index] + b[index];
    }
  }
}
```

```
void main()
{
  .....
  addMatrix(a, b, c, N);
}
```

In CUDA, one writes a C function, called a *kernel*, to compute one element in the matrix, and invokes as many threads to run that function as the matrix has elements. In each thread the kernel runs with a predefined structure `threadIdx` indicating which of the many threads is running:

```
// add 2 matrices on the GPU (simplified)
__global__ void addMatrix(float *a,float *b,
                          float *c, int N)
{
  int i= threadIdx.x;
  int j= threadIdx.y;
  int index= i + j * N;
  c[index]= a[index] + b[index];
}

void main()
{
  // run addMatrix in 1 block of NxN threads:
  dim3 blocksize(N, N),
  addMatrix<<<1, blocksize>>>(a, b, c, N);
}
```

Here the `__global__` declaration specifier indicates a kernel function that will run on the GPU, and the `<<<N, N>>>` syntax indicates that the `addMatrix()` function will be invoked across a group of threads run in parallel, called a *thread block*. Thread blocks may be one-, two-, or three-dimensional, providing a natural way to invoke computation across the elements in a domain such as a vector, matrix, or field.

CUDA makes three key refinements to the core concept of running kernel functions across many parallel threads: hierarchical thread blocks, shared memory, and barrier synchronization.

2.1. Hierarchical thread blocks

Thread blocks may contain up to 512 threads on an NVIDIA Tesla architecture GPU, but kernels are invoked on a *grid* consisting of many thread blocks that are scheduled independently. All the threads in all the blocks in a grid execute the kernel and then exit. Thus the CUDA code given above needs to be extended slightly:

```
// add 2 matrices on the GPU (scalable)
__global__ void addMatrix(float *a,float *b,
                          float *c, int N)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  int j=blockIdx.y*blockDim.y+threadIdx.y;
  int index = i + j * N;
  if ( i < N && j < N)
    c[index]= a[index] + b[index];
}
```

```
// not shown: allocate & copy matrices to GPU
void main()
{

  dim3 dimBlock (16, 16);
  dim3 dimGrid (N/dimBlk.x, N/dimBlk.y);

  // run addMatrix in blocks of 16x16 threads:
  addMatrix<<<dimGrid, dimBlock>>>(a, b, c,N);
}
```

Here a thread block size of 16x16 = 256 threads was chosen somewhat arbitrarily, and a grid is created with enough blocks to have one thread per matrix element as before. The threads in each thread block are run in parallel; multiple thread blocks may be run one after another or in parallel depending on the resources of the GPU. This hierarchical organization – with thread blocks restricted to a finite size and a grid of many blocks – enables programmers to write scalable code. A CUDA program will run on a low-cost, low-power GPU capable of processing only one block of threads in parallel, but scale efficiently in performance up to high-end GPUs capable of running dozens of blocks, and to future GPUs capable of running hundreds of blocks.

2.2. Shared memory

In our simple matrix addition example, threads can efficiently run independently: no thread needs to know the elements being accessed by other threads. Often however many threads can solve a problem more efficiently by cooperating, sharing the results of computations or memory fetches. CUDA enables this cooperation by providing *shared memory* where kernels can store data – e.g. variables or arrays – that are visible to all threads in a thread block. For example, the threads in a block could compute the sum of the elements in an array by each placing one element into an array in shared memory, then adding the element next to it, then the element located four array slots away, then eight, and so on. All the other threads in the block are doing the same thing in parallel – they are cooperating by computing partial sums through shared memory. Shared memory is on-chip and thus small (16K on NVIDIA's current GPUs) but extremely fast, so exploiting shared memory makes this summing operation dramatically faster.

2.3 Barrier synchronization

Once threads are operating in parallel on the same memory, it becomes important to provide a mechanism that guarantees (for example) that one thread will not attempt to read a result before another thread has finished writing it. CUDA provides the `__syncthreads()` intrinsic function for this purpose. `__syncthreads()` acts as a *barrier* at which all threads in the block must wait before any are allowed to proceed. CUDA's synchronization is intentionally simple and lightweight, so that programmers can use a barrier to synchronize before threads read or write to shared memory.

3. SUMMARY

NVIDIA GPUs provide massive computation resources, with up to hundreds of cores running thousands of threads. CUDA makes that raw computational power accessible and easy to program by allowing the user to write natural C code which is then run by thousands or millions of threads. Threads are organized in a two-level hierarchy of blocks and grids. Threads run in parallel with other threads in their thread block and can intercommunicate via shared memory, with simple barriers to synchronize. This minimal set of extensions to C nonetheless exposes the power of massive parallel programming.

Researchers around the world and across all scientific and engineering disciplines are successfully using CUDA and NVIDIA GPUs to speed their codes up by one to two orders of magnitude. We have attempted to motivate the use of GPU computing in biomedical imaging and provide a brief overview of the "feel" of CUDA programming. We encourage researchers to learn more, see what others are publishing, and try CUDA themselves [13].

11. REFERENCES

[1] See *http://gpgpu.org*.

[2] G. C. Sharp, N. Kandasamy, H. Singh, and M. Folkert, "GPU-based streaming architectures for fast cone-beam CT image recon struction and Demons deformable registration," *Physics in Medicine and Biology*, 2007.

[3] C. Vetter and C. Guetter and C. Xu and R. Westermann, "Non-rigid multi-modal registration on the GPU," *Medical Imaging 2007: Image Processing*, SPIE, vol. 6512, Mar 2007.

[4] J. Gu and L. Gu, "Fast DDR Generation Based on GPU," *Int'l Journal of Computer Assisted Radiology and Surgery*, Jun 2006.

[5] F. Xu and K. Mueller. "Real-Time 3D Computed Tomographic Reconstruction Using Commodity Graphics Hardware," Physics in Medicine and Biology, vol. 52, pp. 3405–3419, 2007.

[6] NVIDIA. 2007. CUDA Programming Guide 1.1; see *http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.*

[7] J. Stone, J. Phillips, D. Hardy, P. Freddolino, L. Trabuco, K. Schulten. "Accelerating molecular modeling applications with graphics processors," Journal of Computational Chemistry, Vol. 28, No. 16 , pp 2618 – 2640, 25 Sep 2007.

[8] R. G. Belleman, J. Bédorf and S. F. Portegies, "High Performance Direct Gravitational N-body Simulations on Graphics Processing Units II: An implementation in CUDA," arXiv:0707. 0438v2 [astro-ph], Jul 2007 (Accepted to New Astronomy).

[9] W. Jeong, P. T. Fletcher, R. Tao, and R.T. Whitaker, "Interactive Visualization of Volumetric White Matter Connectivity in DT-MRI Using a Parallel-Hardware Hamilton-Jacobi Solver," *Proc. IEEE Visualization 007*, Oct 2007.

[10] M. C Schatz, C. Trapnell, A. L Delcher, and A. Varshney, "MUMmerGPU: High-throughput sequence alignment using Graphics Processing Units," BMC Bioinformatics 2007, 8:474, Dec 2007.

[11] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger. "Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA)," *2007 Nuclear Science Symposium and Medical Imaging Conference*, Honolulu HW (Oct 2007).

[12] S. Stone, J. Haldar, S. Tsao, W. Hwu, Z. Liang, B. Sutton. "Accelerating Advanced MRI Reconstructions on GPUs", *ACM Frontiers in Computing*, Ischia, Italy, to appear (May 2008).

[13] See *http://nvidia.com/cuda*