An Even Easier Introduction to CUDA

May 02, 2025
By Mark Harris

This post is a super simple introduction to CUDA, the popular parallel computing platform and programming model from NVIDIA. I wrote a previous post, An Easy Introduction to CUDA in 2013 that has been popular over the years. But CUDA programming has gotten easier, and GPUs have gotten much faster, so it's time for an updated (and even easier) introduction.

CUDA C++ is just one of the ways you can create massively parallel applications with CUDA. It lets you use the powerful C++ programming language to develop high performance algorithms accelerated by thousands of parallel threads running on GPUs. Many developers have accelerated their computation- and bandwidth-hungry applications this way, including the libraries and frameworks that underpin the ongoing revolution in artificial intelligence known as Deep Learning.

So, you've heard about CUDA and you are interested in learning how to use it in your own applications. If you are a C++ programmer, this blog post should give you a good start. To follow along, you'll need a computer with a CUDA-capable GPU (Windows, WSL, or 64-bit Linux, and any NVIDIA GPU should do), or a cloud instance with GPUs (AWS, Azure, Google Colab, and other cloud service providers have them). You'll also need the free CUDA Toolkit installed.

Let's get started!

Starting Simple

We'll start with a simple C++ program that adds the elements of two arrays with a million elements each.

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
 for (int i = 0; i < n; i++)
     y[i] = x[i] + y[i];
}

int main(void)
{
 int N = 1<<20; // 1M elements

 float *x = new float[N];
 float *y = new float[N];

 // initialize x and y arrays on the host
 for (int i = 0; i < N; i++) {
   x[i] = 1.0f;
   y[i] = 2.0f;
 }
```

```
  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

First, compile and run this C++ program. Put the code above in a file and save it as add.cpp, and then compile it with your C++ compiler. I'm on Linux so I'm using g++, but you can use MSVC on Windows (or g++ on WSL).

```
> g++ add.cpp -o add
```

Then run it:

```
> ./add
Max error: 0.000000
```

As expected, it prints that there was no error in the summation and then exits. Now I want to get this computation running (in parallel) on the many cores of a GPU. It's actually pretty easy to take the first steps.

First, I just have to turn our add function into a function that the GPU can run, called a *kernel* in CUDA. To do this, all I have to do is add the specifier __global__to the function, which tells the CUDA C++ compiler that this is a function that runs on the GPU and can be called from CPU code.

```
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *sum, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    sum[i] = x[i] + y[i];
}
```

This __global__ function is known as a CUDA *kernel*, and runs on the GPU. Code that runs on the GPU is often called *device code*, while code that runs on the CPU is *host code*.

Memory Allocation in CUDA

To compute on the GPU, I need to allocate memory accessible by the GPU. Unified Memory in CUDA makes this easy by providing a single memory space accessible by all GPUs and CPUs in your system. To allocate data in unified memory, call cudaMallocManaged(), which returns a pointer that you can access from host (CPU) code or device (GPU) code. To free the data, just pass the pointer to cudaFree().

I just need to replace the calls to new in the code above with calls to cudaMallocManaged(), and replace calls to delete [] with calls to cudaFree.

```
// Allocate Unified Memory -- accessible from CPU or GPU
float *x, *y, *sum;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

...

// Free memory
cudaFree(x);
cudaFree(y);
```

Finally, I need to *launch* the add() kernel, which invokes it on the GPU. CUDA kernel launches are specified using the triple angle bracket syntax <<< >>>. I just have to add it to the call to add before the parameter list.

```
add<<<1, 1>>>(N, sum, x, y);
```

Easy! I'll get into the details of what goes inside the angle brackets soon; for now all you need to know is that this line launches one GPU thread to run add().

Just one more thing: I need the CPU to wait until the kernel is done before it accesses the results (because CUDA kernel launches don't block the calling CPU thread). To do this I just call cudaDeviceSynchronize() before doing the final error checking on the CPU.

Here's the complete code:

```
#include <iostream>
#include <math.h>

// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
 for (int i = 0; i < n; i++)
   y[i] = x[i] + y[i];
}

int main(void)
{
 int N = 1<<20;
 float *x, *y;

 // Allocate Unified Memory – accessible from CPU or GPU
 cudaMallocManaged(&x, N*sizeof(float));
 cudaMallocManaged(&y, N*sizeof(float));
 // initialize x and y arrays on the host
 for (int i = 0; i < N; i++) {
   x[i] = 1.0f;
   y[i] = 2.0f;
 }

 // Run kernel on 1M elements on the GPU
 add<<<1, 1>>>(N, x, y);

 // Wait for GPU to finish before accessing on host
 cudaDeviceSynchronize();

 // Check for errors (all values should be 3.0f)
 float maxError = 0.0f;
 for (int i = 0; i < N; i++) {
   maxError = fmax(maxError, fabs(y[i]-3.0f));
 }
 std::cout << "Max error: " << maxError << std::endl;

 // Free memory
 cudaFree(x);
 cudaFree(y);
  return 0;
}
```

CUDA files have the file extension .cu. So save this code in a file called add.cu and compile it with nvcc, the CUDA C++ compiler.

```
> nvcc add.cu -o add_cuda

> ./add_cuda

Max error: 0.000000
```

This is only a first step, because as written, this kernel is only correct for a single thread, since every thread that runs it will perform the add on the whole array. Moreover, there is a race condition since multiple parallel threads would both read and write the same locations.

Note: on Windows, you need to make sure you set Platform to x64 in the Configuration Properties for your project in Microsoft Visual Studio.

Profile it!

A good way to find out how long the kernel takes to run is to run it with the NSight Systems CLI, `nsys`. We could just type nsys profile -t cuda --stats=true ./add_cuda on the command line. However, this produces verbose statistics, and in this post we really only want to know how long the kernel takes to run. So I wrote a wrapper script called nsys_easy that only produces the output we need and stops nsys from producing intermediate files that will clutter the source directory. The script is available on GitHub. Just download nsys_easy and put it somewhere in your PATH (or even in the current directory).

```
> nsys_easy ./add_cuda
Max error: 0
Generating '/tmp/nsys-report-bb25.qdstrm'
[1/1] [========================100%] nsys_easy.nsys-rep
Generated:
   /home/nfs/mharris/src/even_easier/nsys_easy.nsys-rep
Generating SQLite file nsys_easy.sqlite from nsys_easy.nsys-rep
Processing 1259 events: [=====================================100%]
Processing [nsys_easy.sqlite] with [cuda_gpu_sum.py]...

** CUDA GPU Summary (Kernels/MemOps) (cuda_gpu_sum):

Time (%)  Total Time (ns)  Instances  Category   Operation
--------  ---------------  --------- ----------- --------------------------
    98.5       75,403,544      1      CUDA_KERNEL add(int, float *, float *)
     1.0          768,480     48      MEMORY_OPER [memcpy Unified H2D]
     0.5          352,787     24      MEMORY_OPER [memcpy Unified D2D]
```

The CUDA GPU Summary table shows a single call to add. It takes about 75ms on an NVIDIA T4 GPU. Let's make it faster with parallelism.

Picking up the Threads

Now that you've run a kernel with one thread that does some computation, how do you make it parallel? The key is in CUDA's <<<1, 1>>> syntax. This is called the execution configuration, and it tells the CUDA runtime how many parallel threads to use for the launch on the GPU. There are two parameters here, but let's start by changing the second one: the number of threads in a thread block. CUDA GPUs run kernels using blocks of threads that are a multiple of 32 in size; 256 threads is a reasonable size to choose.

```
add<<<1, 256>>>(N, x, y);
```

If I run the code with only this change, it will do the computation once per thread, rather than spreading the computation across the parallel threads. To do it properly, I need to modify the kernel. CUDA C++ provides keywords that let kernels get the indices of the running threads. Specifically, threadIdx.x contains the index of the current thread within its block, and blockDim.x contains the number of threads in the block. I'll just modify the loop to stride through the array with parallel threads.

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

The add function hasn't changed that much. In fact, setting index to 0 and stride to 1 makes it semantically identical to the first version.

Save the file as add_block.cu and compile and run it in nsys_easy again. For the remainder of the post I'll just show the relevant line from the output.

```
Time (%) Time (ns) Instances  Category   Operation
-------- --------- --------- ----------- ----------------------
79.0     4,221,011     1     CUDA_KERNEL add(int, float *, float *)
```

That's a big speedup (75ms down to 4ms), but not surprising since execution went from one thread to 256 threads. Let's keep going to get even more performance.

Out of the Blocks

CUDA GPUs have many parallel processors grouped into Streaming Multiprocessors, or SMs. Each SM can run multiple concurrent thread blocks, but each thread block runs on a single SM. As an example, an NVIDIA T4 GPU based on the Turing GPU Architecture has 40 SMs and 2560 CUDA cores, and each SM can support up to 1024 active threads. To take full advantage of all these threads, I should launch the kernel with multiple thread blocks.

By now you may have guessed that the first parameter of the execution configuration specifies the number of thread blocks. Together, the blocks of parallel threads make up what is known as the *grid*. Since I have N elements to process, and 256 threads per block, I just need to calculate the number of blocks to get at least N threads. I simply divide N by the block size (being careful to round up in case N is not a multiple of blockSize).

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

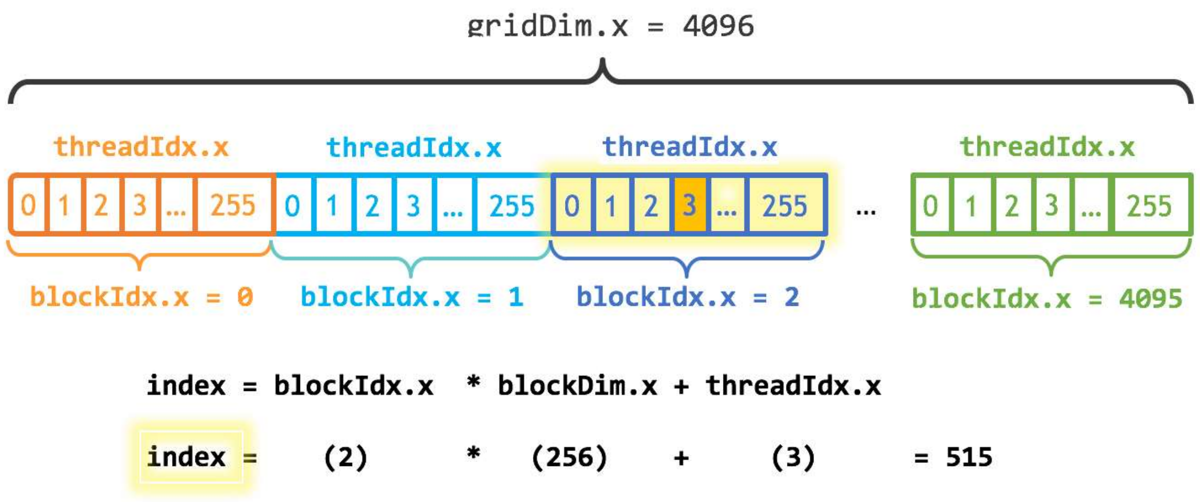

Figure 1. Grid, Block and Thread indexing in CUDA kernels (one-dimensional).

I also need to update the kernel code to take into account the entire grid of thread blocks. CUDA provides gridDim.x, which contains the number of blocks in the grid, and blockIdx.x, which contains the index of the current thread block in the grid. Figure 1 illustrates the the approach to indexing into an array (one-dimensional) in CUDA using blockDim.x, gridDim.x, and threadIdx.x. The idea is that each thread gets its index by computing the offset to the beginning of its block (the block index times the block size: blockIdx.x * blockDim.x) and adding the thread's index within the block (threadIdx.x). The code blockIdx.x * blockDim.x + threadIdx.x is idiomatic CUDA.

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

The updated kernel also sets stride to the total number of threads in the grid (blockDim.x * gridDim.x). This type of loop in a CUDA kernel is often called a *grid-stride loop*.

Save the file as add_grid.cu and compile and run it in nsys_easy again.

```
Time (%) Time (ns) Instances Category    Operation
-------- --------- --------- ----------- ------------------------
79.6     4,514,384     1     CUDA_KERNEL add(int, float *, float *)
```

Well, that's interesting. There's no speedup from this change, and possibly a slight slowdown. Why is that? If increasing compute by a factor of 40 (the number of SMs) doesn't bring down total time, then computation was not the bottleneck.

Unified Memory Prefetching

There's a clue to the bottleneck if we look at the full summary table from the profiler:

```
Time (%) Time (ns) Instances  Category    Operation
-------- --------- --------- ----------- ----------------------
   79.6 4,514,384      1      CUDA_KERNEL add(int, float *, float *)
   14.2   807,245     64      MEMORY_OPER [CUDA memcpy Unified H2D]
    6.2   353,201     24      MEMORY_OPER [CUDA memcpy Unified D2H]
```

Here we can see that there are 64 host-to-device (H2D) and 24 device-to-host (D2H) "unified" memcpy operations. But there are no explicit memcpy calls in the code. Unified Memory in CUDA is virtual memory. Individual virtual memory pages may be resident in the memory of any device (GPU or CPU) in the system, and those pages are migrated on demand. This program first initializes the arrays on the CPU in a for loop, and then launches the kernel where the arrays are read and written by the GPU. Since the memory pages are all CPU-resident when the kernel runs, there are multiple page faults and the hardware migrates the pages to the GPU memory when the faults occur. This results in a memory bottleneck, which is why we don't see a speedup.

The migration is expensive because page faults occur individually, and GPU threads stall while they wait for the page migration. Because I know what memory is needed by the kernel (x and y arrays), I can use *prefetching* to make sure that the data is on the GPU before the kernel needs it. I do this by using the cudaMemPrefetchAsync() function before launching the kernel:

```
// Prefetch the x and y arrays to the GPU
cudaMemPrefetchAsync(x, N*sizeof(float), 0, 0);
cudaMemPrefetchAsync(y, N*sizeof(float), 0, 0);
```

Running this with the profiler produces the following output. The kernel now takes under 50 microseconds!

```
Time (%) Time (ns) Instances  Category    Operation
-------- --------- --------- -----------  ----------------------
   63.2   690,043     4      MEMORY_OPER  [CUDA memcpy Unified H2D]
   32.4   353,647    24      MEMORY_OPER  [CUDA memcpy Unified D2H]
    4.4    47,520     1      CUDA_KERNEL  add(int, float *, float *)
```

Summing Up

Prefetching all the pages of the arrays at once is much faster than individual page faults. Note that this change can benefit all versions of the add program, so let's add it to all three and run them in the profiler again. Here's a summary table.

Version	Time	Speedup vs. Single Thread	Bandwidth
Single Thread	91,811,206 ns	1x	137 MB/s
Single Block (256 threads)	2,049,034 ns	45x	6 GB/s
Multiple Blocks	47,520 ns	1932x	265 GB/s

Once the data is in memory, the speedup going from a single block to multiple blocks is proportional to the number of SMs (40) on the GPU.

As you can see, we can achieve very high bandwidth on GPUs. The add kernel is very bandwidth-bound (265 GB/s is over 80% of the T4's peak bandwidth of 320GB/s) , but GPUs also excel at heavily compute-bound computations such as dense matrix linear algebra, deep learning, image and signal processing, physical simulations, and more.

Exercises

To keep you going, here are a few things to try on your own. Please post about your experience in the comments section below.

1. Browse the CUDA Toolkit documentation. If you haven't installed CUDA yet, check out the Quick Start Guide and the installation guides. Then browse the Programming Guide and the Best Practices Guide. There are also tuning guides for various architectures.

2. Experiment with printf() inside the kernel. Try printing out the values of threadIdx.x and blockIdx.x for some or all of the threads. Do they print in sequential order? Why or why not?

3. Print the value of threadIdx.y or threadIdx.z (or blockIdx.y) in the kernel. (Likewise for blockDim and gridDim). Why do these exist? How do you get them to take on values other than 0 (1 for the dims)?

Where To From Here?

I hope that this post has whet your appetite for CUDA and that you are interested in learning more and applying CUDA C++ in your own computations. If you have questions or comments, don't hesitate to reach out using the comments section below.

There is a whole series of older introductory posts that you can continue with:

- How to Implement Performance Metrics in CUDA C++
- How to Query Device Properties and Handle Errors in CUDA C++
- How to Optimize Data Transfers in CUDA C++
- How to Overlap Data Transfers in CUDA C++
- How to Access Global Memory Efficiently in CUDA C++
- Using Shared Memory in CUDA C++
- An Efficient Matrix Transpose in CUDA C++
- Finite Difference Methods in CUDA C++, Part 1
- Finite Difference Methods in CUDA C++, Part 2
- Accelerated Ray Tracing in One Weekend with CUDA

There is also a series of CUDA Fortran posts mirroring the above, starting with An Easy Introduction to CUDA Fortran.

There is a wealth of other content on CUDA C++ and other GPU computing topics here on the NVIDIA Developer Blog, so look around!

If you enjoyed this post and want to learn more, the NVIDIA DLI offers several in-depth CUDA programming courses.

- For those of you just starting out, see Getting Started with Accelerated Computing in Modern CUDA C++, which provides dedicated GPU resources, a more sophisticated programming environment, use of the NVIDIA Nsight Systems visual profiler, dozens of interactive exercises, detailed presentations, over 8 hours of material, and the ability to earn a DLI Certificate of Competency.

- For Python programmers, see Fundamentals of Accelerated Computing with CUDA Python.

- For more intermediate and advanced CUDA programming materials, see the *Accelerated Computing* section of the NVIDIA DLI self-paced catalog.