# A CUDA Tutorial

Rippy

May 2022

**Abstract**

This report is a a tutorial on how to use CUDA and lay the groundwork for others to utilize CUDA in an easier manner with the guide in this project. I utilized CUDA in two scenarios, generating noisy signals, and then filtering those signals with a trapezoidal slide filter.

## 1 Introduction

CUDA stands for Compute Unified Device Architecture, and it is a parallel computing platform and API. It allows for very quick parallel vector computations. It is commonly utilized for videogames on GPUs to speed up the many vector calculations required for the games to run effectively. It is owned, distributed, and created by NVIDIA, who are well known for their creation of graphics cards. The language is designed to work well with C,C++. and Fortran. While there exists libraries to connect CUDA to languages such as python, it suffers noticeable slow down from conversion to the other languages it was not originally intended to work with. CUDA is the preferred platform to work with when compared to prior platforms like Direct3D or OpenGL due to the significant ease of access compared to the aforementioned options.

## 2 How Cuda Works

For the scope of this lab, C++ was the main language utilized to interface with CUDA 11.6. It is very important to delineate between device code and host code. Device code is run and stored on th GPU, while host code is run and stored on the CPU. Unified memory makes it easy to interchange between these two as it is a space of memory that is accessible to both the GPU and CPU on the computer. This unified memory is manipulated using the CUDA malloc commands. CUDA utilizes compute kernels, which are separate routines that can be run separately in parallel with all other processes on the machine. The way that CUDA performs operations on its kernels can be described in 4 simple steps.

- First, we copy data from the host memory space onto the device (GPU) memory space using the unified memory operations. The specifics of this will be described later.

- Next, the CPU sends kernel instruction code, written on the host, to the GPU. Here the parameters, threads, and other specifics are controlled.
- The GPU then performs the requested kernel operations and the output is put into a buffer vector.
- Again, using the unified memory operations, the buffer vector is copied from the GPU device memory through the unified memory back on to the host memory.

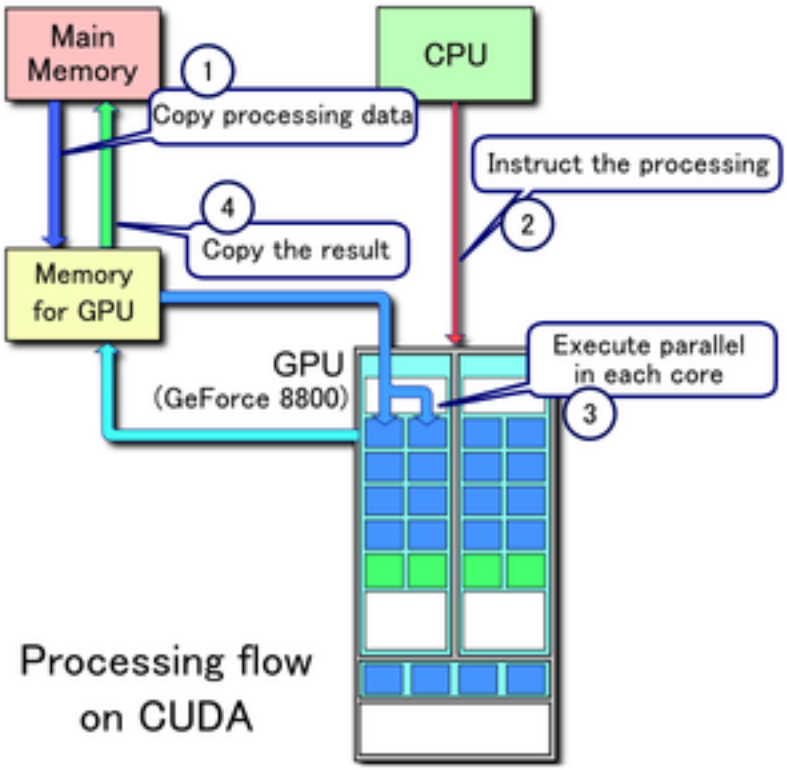


Figure 1: A visual aid of how the CPU interfaces with the GPU through CUDA

# 3 Breakdown of a CUDA kernel and wrapper code

This kernel code was utilized in the Tracer.cu code that was used to generate noisy signal data for use with the trapezoid filter.

```
void noiseAdd(double* c, double* b, double* a, int size) {

    double* dev_a = 0;
    double* dev_b = 0;
    double* dev_c = 0;
    cudaSetDevice(0); //Only different if on multi-GPU system

    //Allocate GPU buffers for three vectors
    cudaMalloc((void**)&dev_c, size * sizeof(double));
    cudaMalloc((void**)&dev_a, size * sizeof(double));
    cudaMalloc((void**)&dev_b, size * sizeof(double));


    //Copy the input vectors to the GPU
    cudaMemcpy(dev_a, a, size * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size * sizeof(double), cudaMemcpyHostToDevice);

    // Launch a kernel on the GPU with one thread for each element.
    int blockSize = 256;
    int numBlocks = (size + blockSize - 1) / blockSize;
    addDouble << <numBlocks, blockSize >> > (dev_c, dev_a, dev_b);

    // Copy output vector from GPU buffer to host memory.
    cudaMemcpy(c, dev_c, size * sizeof(double), cudaMemcpyDeviceToHost);

    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);
}
```

Figure 2: The Cuda Kernel Wrapper function

For use of the CUDA kernels, it is highly recommended to create wrapper functions to take care of the unified memory commands. For each wrapper function, there are several main parts. First, begin by defining device variables to be utilized by the GPU. For this example, we called them:

**dev_a, dev_b, dev_c**

Once we have defined the device variables, we choose which GPU this wrapper will utilize. For multi-GPU systems, we can offload different operations to other GPUs to further parallelize the operations. In the case of this example, we used a single GPU, so

**cudaSetDevice(0);**

was set. Next, we set aside the unified memory to interface between the CPU and GPU with

```
cudaMalloc((void**)&dev_c, size * sizeof(double));
cudaMalloc((void**)&dev_b, size * sizeof(double));
cudaMalloc((void**)&dev_a, size * sizeof(double));
```

Next, using the unified memory, we copy the input vectors from our wrapper function into the GPU memory space using:

```
cudaMemcpy(dev_a, a, size * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size * sizeof(double), cudaMemcpyHostToDevice);
```

Next comes the use of the kernel function. The kernel is defined in the host code as:

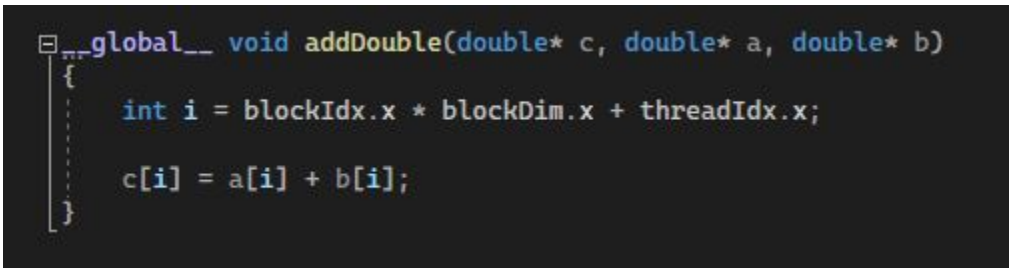


Figure 3: The Kernel Code to add two double vectors

We call the kernel in the wrapper using:

```
int blockSize = 256;
int numBlocks = (size + blockSize - 1) / blockSize;
addDouble <<< numBlocks, blockSize >>> (dev_c, dev_a, dev_b);
```

To properly utilize the kernel, we must understand the **block, the thread, and the block dimension.** A **block** contains threads for the kernel, the **block dimension** is how many threads are contained in the block, and the **thread** is the specific parallel instance being run. CUDA GPUs run kernels using blocks of threads that are a multiples of 32 in size, so any multiple of 32 will work for block size. In this case, 256 is the size we chose, but based on the size of the calculation done, another number may work better. For the case of this example, 256 seemed to work the best. A good way to visualize this is a block is a room, a thread is a person doing a task, and the block dimension is how many people we allow per room.

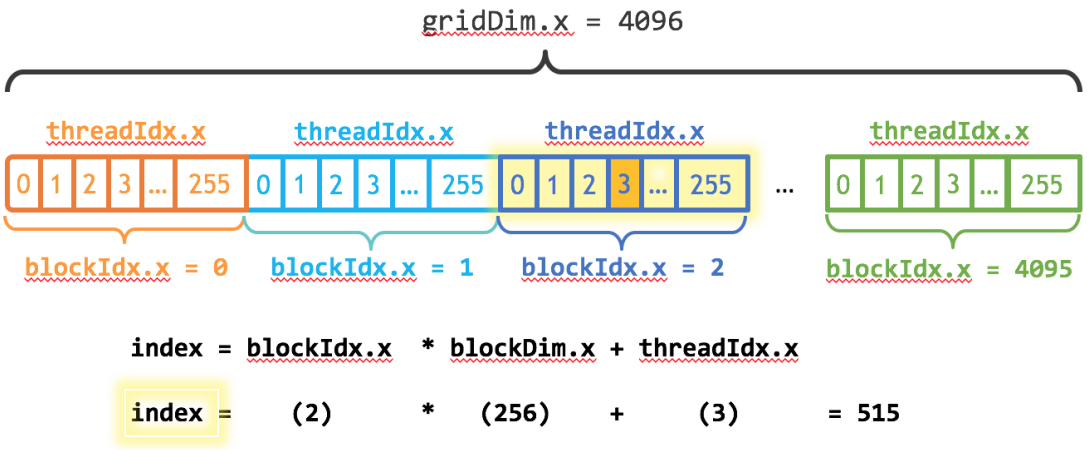


Figure 4: A visual guide to working with threads and blocks on the GPU

We calculate the number of blocks required based on our size of the vector $N$ and the block size. We always want to round up on the number of required blocks to guarantee we perform operations on all indices. Once we have completed the kernel task, we must once again utilize the unified memory to copy the output vector from the GPU device memory back to the host memory. This is accomplished by:

```
cudaMemcpy(c, dev_c, size * sizeof(double), cudaMemcpyDeviceToHost);
```

Finally, and most importantly, we need to free the memory on the GPU device to let it know we no longer need it. If we do not do this within the wrapper, the memory will stay allocated until the GPU is reset, as we cannot access it again.

```
cudaFree(dev_c);
cudaFree(dev_a);
cudaFree(dev_a);
```

# 4 Some Noteable Restrictions and Pitfalls

There are some important things to note about CUDA kernels. We have a significantly reduced set of functions we have access to. You cannot import any C++ libraries into the kernel. The following are allowed in the kernel:

**auto**
**lambdas**
**std::initializer_list**
**variadic templates**
**static_asserts**
**const expr**
**rvalve references**
**range based for loops**
**printf is supported for debugging**
**basic boolean logic**

For the basic boolean logic, it is important to note that if there is too much divergence with if-else statements, the kernels will suffer significant slowdown. Infinite loops inside the GPU are also a problem, and can only be killed by resetting the GPU, so before running the kernels, be sure to check to see if infinite loops exist.

# 5 An Application of CUDA to Signal Processing

As an example for using CUDA to speed up calculations by utilizing parallel threads, I took a look at signal processing. A noisy signal was generated using **Tracer.cu** and plotted using matplot on python. The following output signal was obtained:

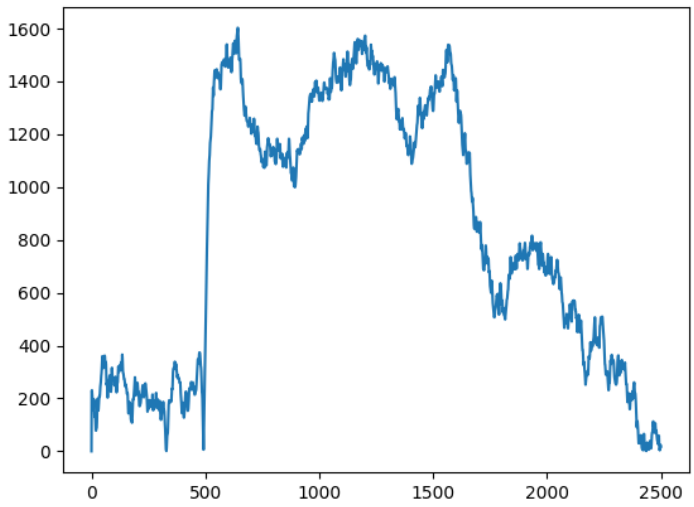


Figure 5: The generated noisy signal

Once the signal had been generated, it was run through a trapezoidal filter, **TrapFilter.cu**, that utilized CUDA to efficiently filter over the results. The following output was obtained for the filter:

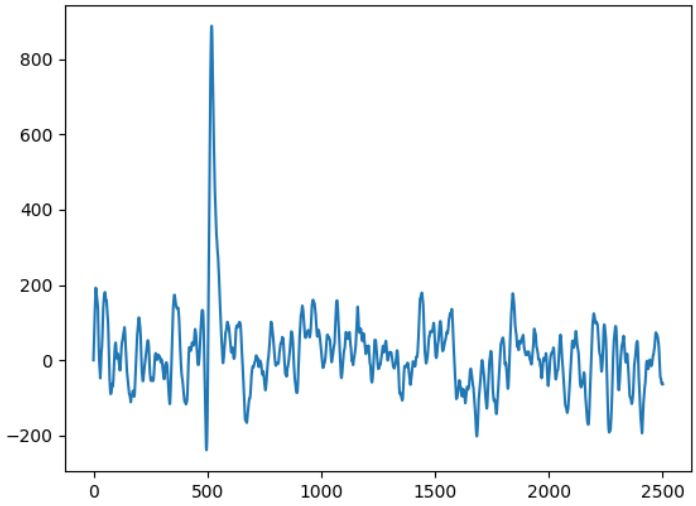


Figure 6: The Filtered Signal

The signal was created at 500, and successfully found using the trapezoidal filter. The filter is defined as:

$$y(n) = \frac{x_c(n-1)}{n_a} \tag{1}$$

Where $x_c$ is computed recursively using the following 3 equations:

$$x_c(n) = x_b(n) - x_b(n-n_b) + x_c(n-1) \tag{2}$$
$$x_b(n) = x_a(n-n_a) + x_b(n-1) \tag{3}$$
$$x_a(n) = x(n) - e^{\frac{T_s}{\tau_d}} x(n-1) \tag{4}$$

Where $x$ is the original signal data, $n_a$ is the rise/fall time of the trapezoid filter, and $n_b$ is the size of the flat top of the filter. The filter itself was recursively defined, so it was difficult to use CUDA to split the calculation up. The initial priming of the input signal $x_a$ was able to effectively use CUDA however, $x_b(n), x_c(n), y(n)$ are recursively defined and cannot use CUDA effectively.

# 6 Conclusion

The main takeaway from this report should be a solid basic understanding of how to utilize CUDA. It is my hope that this work can be used in future projects to avoid problems where large calculations slow down the ability to perform the research or analysis.