The Compute Unified Device Architecture (CUDA)

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

The CUDA Paradigm

CUDA is an NVIDIA-only product, but it is very popular, and got the whole GPU-as-CPU ball rolling, which has resulted in other products like OpenCL. CUDA also comes with several libraries that are highly optimized for applications such as linear algebra and deep learning.

CUDA View of the GPU’s Architecture

- The GPU has some number of Streaming Multiprocessors (SMs)
- Each SM operates on a Grid of Blocks

Organization: Blocks are Arranged in Grids

- Each SM operates on a Grid of Blocks
- Each Block in the Grid operates on a Grid of Threads
A Thread Block has:
- Size: allows some number of concurrent threads
- Shape: 1D, 2D, or 3D (really just a convenience)

Threads have Thread ID numbers within the Block.
The program uses these Thread IDs to select work and pull data from memory.
Threads share data and synchronize while doing their share of the work.
A "Warp" is a group of 32 threads that are simultaneously executing the same instruction on different pieces of data.
The threads in a Thread Block can cooperate with each other by:
- Synchronizing their execution
- Efficiently sharing data through a low latency shared memory
Threads from different blocks cannot cooperate.

A Block has a Grid of Threads

The hardware implements low-overhead Warp switching:
- Warps whose next instruction has its operands ready for consumption are eligible for execution.
- If not, go on to the first Warp that is ready
- All threads in a Warp execute the same instruction at any given time, but on different data.
This tells you that each SM needs a gang of Warps to work on so that something is always ready to run. 192 or 256 are good numbers of Threads per Block (multiples of 32).

Threads Can Access Various Types of Storage
- Each thread has access to:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read-only per-grid constant memory
  - Read-only per-grid texture memory
- The CPU can read and write global, constant, and texture memories

Thread Rules
- Threads can share memory with the other Threads in the same Block
- Threads can synchronize with other Threads in the same Block
- Global and Constant memory is accessible by all Threads in all Blocks
- Each Thread has registers and local memory
- Each Block can use at most some maximum number of registers, divided equally among all Threads
- A Block is run on only one SM (i.e., cannot switch to another SM)
- 192 or 256 are good numbers of Threads per Block (multiples of the Warp size 32)
Types of CUDA Functions

<table>
<thead>
<tr>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>device</em> float DeviceFunc()</td>
<td>GPU GPU</td>
</tr>
<tr>
<td><em>global</em> void KernelFunc()</td>
<td>GPU CPU</td>
</tr>
<tr>
<td><em>host</em> float HostFunc()</td>
<td>CPU CPU</td>
</tr>
</tbody>
</table>

___global___ defines a kernel function – it must return void.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>On-chip</td>
<td>N/A</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads + CPU</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + CPU</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + CPU</td>
</tr>
</tbody>
</table>

A CUDA Thread can Query where it Fits in its “Community” of Threads and Blocks

- dim3 gridDim;
  - Dimensions of the grid in blocks (gridDim.z is not used)
- dim3 blockIdx;
  - Block index within this grid
- dim3 blockDim;
  - Dimensions of this block in threads
- dim3 threadIdx;
  - Thread index within the block

A CUDA Thread needs to know where it Fits in its “Community” of Threads and Blocks

- dim3 gridDim;
  - Dimensions of the grid in blocks (gridDim.z is not used)
- dim3 blockIdx;
  - Block index within this grid
- dim3 blockDim;
  - Dimensions of this block in threads
- dim3 threadIdx;
  - Thread index within the block

For a 1D problem:
```
int blockThreads = blockIdx.x * blockDim.x;
int gid = blockThreads + threadIdx.x;
```

For a 2D problem:
```
int blockNum = blockIdx.y * gridDim.x + blockIdx.x;
int blockThreads = blockNum * blockDim.y;
int gid = blockThreads + threadIdx.y * blockDim.x + threadIdx.x;
```
1. CPU Serial Code

2. GPU Parallel Kernel
   KernelA<<< nBlocks, nThreadsPerBlock >>>(args);

3. CPU Serial Code

4. GPU Parallel Kernel
   KernelB<<< nBlocks, nThreadsPerBlock >>>(args);

The C/C++ Program Calls a CUDA Kernel using a Special <<<...>>> Syntax

KernelFunction<<< NumBlocks, NumThreadsPerBlock >>>(arg1, arg2, ...);

Creating your own CUDA Visual Studio Folder

1. Un-zip the ArrayMul2017.zip file into its own folder.
2. Rename that folder to what you want it to be.
3. Rename arrayMul.cu to whatever you want it to be (keeping the .cu extension). Without the .cu extension, we will call this the basename.
4. Rename the .sln and .vcxproj files to have the same basename as your .cu file has.
5. Edit the *.sln file. Replace all occurrences of “arrayMul” to what the basename.
6. Edit the *.vcxproj file. Replace all occurrences of “arrayMul” with the basename. Replace all occurrences of ArrayMul2017 with whatever you renamed the folder to.
7. In the .cu file, rename the CUDA function from ArrayMul to whatever you want it to be. Do this twice, once in the definition of the function and once in the calling of the function.
8. Now modify the CUDA code to perform the computation you require.

Anatomy of a CUDA Program:

```c
// #defines and #includes
#include <stdio.h>
#include <assert.h>
#include <malloc.h>
#include <math.h>
#include <stdlib.h>

// CUDA runtime
#include <cuda_runtime.h>

// Helper functions and utilities to work with CUDA
#include "helper_functions.h"
#include "helper_cuda.h"

#define BLOCKSIZE 128 // number of threads per block
#define SIZE 1*1024*1024 // array size
#define NUMTRIALS 100 // to make the timing more accurate
#define TOLERANCE 0.00001f // tolerance to relative error
```

```c
// array multiplication (CUDA Kernel) on the device: C = A * B
__global__ void ArrayMul( float *A, float *B, float *C )
{
    int gid = blockIdx.x*blockDim.x + threadIdx.x;
}
```
// allocate host memory:
float * hA = new float [SIZE];
float * hB = new float [SIZE];
float * hC = new float [SIZE];
for( int i = 0; i < SIZE; i++ )
{
    hA[i] = hB[i] = (float) sqrt( (float)i );
}

// allocate device memory:
float *dA, *dB, *dC;
dim3 dimsA(SIZE, 1, 1);
dim3 dimsB(SIZE, 1, 1);
dim3 dimsC(SIZE, 1, 1);
cudaError_t status;
status = cudaMalloc( reinterpret_cast<void **>(&dA), SIZE*sizeof(float) );
checkCudaErrors( status );
status = cudaMalloc( reinterpret_cast<void **>(&dB), SIZE*sizeof(float) );
checkCudaErrors( status );
status = cudaMalloc( reinterpret_cast<void **>(&dC), SIZE*sizeof(float) );
checkCudaErrors( status );

// copy host memory to the device:
status = cudaMemcpy( dA, hA, SIZE*sizeof(float), cudaMemcpyHostToDevice );
checkCudaErrors( status );
status = cudaMemcpy( dB, hB, SIZE*sizeof(float), cudaMemcpyHostToDevice );
checkCudaErrors( status );

// setup the execution parameters:
dim3 threads(BLOCKSIZE, 1, 1);
dim3 grid(SIZE / threads.x, 1, 1);

// Create and start timer
cudaDeviceSynchronize();

// allocate CUDA events that we'll use for timing:
cudaEvent_t start, stop;
status = cudaEventCreate( &start );
checkCudaErrors( status );
status = cudaEventCreate( &stop );
checkCudaErrors( status );
status = cudaEventRecord( start, NULL );
checkCudaErrors( status );

// record the start event:
status = cudaEventRecord( start, NULL );
checkCudaErrors( status );

// execute the kernel:
for( int t = 0; t < NUMTRIALS; t++ )
{
    ArrayMul<<<grid, threads>>>( dA, dB, dC );
}

Sometimes when a benchmark runs extremely fast, we run it a number of times for no other reason than to make the elapsed time a more reasonable number.
Anatomy of a CUDA Program:

Getting the Stop Time

```c
// record the stop event:
status = cudaEventRecord( stop, NULL );
checkCudaErrors( status );

// wait for the stop event to complete:
status = cudaEventSynchronize( stop );
checkCudaErrors( status );
```

Printing the Performance

```c
float msecTotal = 0.0f;
status = cudaEventElapsedTime( &msecTotal, start, stop );
checkCudaErrors( status );

// compute and print the performance
double secondsTotal = 0.001 * (double)msecTotal;
double doublePerSecond = (float)SIZE * (float)NUMTRIALS / secondsTotal;
double megaDoublePerSecond = doublePerSecond / 1000000.;
fprintf( stderr, "Size = %10d, MegaDouble/Second = %10.2lf\n", SIZE, megaDoublePerSecond );
```

Copying the Array from the Device to the Host

```c
// copy result from the device to the host:
status = cudaMemcpy( hC, dC, SIZE*sizeof(float), cudaMemcpyDeviceToHost );
checkCudaErrors( status );
```

Checking for Correctness

```c
// check for correctness:
for(int i = 1; i < SIZE; i++)
{
    double error = ( (double)hC[ i ] - (double)i ) / (double)i;
    if( fabs(error) > TOLERANCE )
    {
        fprintf( stderr, "C[%10d] = %10.2lf, correct = %10.2lf\n", i, (double)hC[ i ], (double)i );
    }
}
```
Anatomy of a CUDA Program:
Cleaning Up

```c
// clean up memory:
delete [] hA;
delete [] hB;
delete [] hC;
status = cudaFree( dA );
checkCudaErrors( status );
status = cudaFree( dB );
checkCudaErrors( status );
status = cudaFree( dC );
checkCudaErrors( status );
```

The Results

Running on an NVIDIA 1080 ti graphics card, this program achieved 14,448.99 MegaMultiplies / Second = 14.5 GigaMultiplies / Second