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

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.

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 Block has 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

Threads Can Access Various Types of Storage
- Each thread has access to:
  - RW per-thread registers
  - RW per-thread local memory
  - RW per-block shared memory
  - RW 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

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).
Different Types of CUDA Memory

<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>

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)

Thread Rules

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

Types of CUDA Functions

- `__device__ float DeviceFunc();`
  - Executed on the: GPU
  - Only callable from the: GPU
- `__global__ void KernelFunc();`
  - Executed on the: GPU
  - Only callable from the: CPU
- `__host__ float HostFunc();`
  - Executed on the: CPU
  - Only callable from the: CPU

A CUDA Thread can query where it fits in its “Community” of Threads and Blocks

For a 1D problem:

```c
int blockThreads = blockIdx.x*blockDim.x;
int gid = blockThreads + threadIdx.x;
C[gid] = A[gid]*B[gid];
```

For a 2D problem:

```c
int blockNum = blockIdx.y*gridDim.x + blockIdx.x;
int blockThreads = blockNum*blockDim.x*blockDim.y;
int gid = blockThreads + threadIdx.y*blockDim.x + threadIdx.x;
C[gid] = A[gid]*B[gid];
```

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

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

Types of CUDA Functions

- `__device__ float DeviceFunc();` only callable from the GPU
- `__global__ void KernelFunc();` executable on the GPU
- `__host__ float HostFunc();` executable on the CPU

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

1. CPU Serial Code
2. GPU Parallel Kernel `Kernel<<<nBlocks, nThreadsPerBlock >>>(args)`
3. CPU Serial Code
4. GPU Parallel Kernel `Kernel<<<nBlocks, nThreadsPerBlock >>>(args)`
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.

Creating your own CUDA Visual Studio Folder

Anatomy of a CUDA Program: #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                    10*1024*1024 // array size
#define NUMTRIALS              100 // to make the timing more accurate
#define TOLERANCE             0.00001f // tolerance to relative error
```

Anatomy of a CUDA Program: The Kernel Function

```
__global__  void ArrayMul( float *A, float *B, float *C )
{
    int gid = blockIdx.x*blockDim.x + threadIdx.x;
}
```

Anatomy of a CUDA Program: Setting Up the Memory for the Arrays

```
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) sqrtf(  (float)i );
}
```

Anatomy of a CUDA Program: Getting Ready to Execute

```
// setup the execution parameters:
dim3 threads(BLOCKSIZE, 1, 1 );
dim3 grid( SIZE / threads.x, 1, 1 );
// Create and start timer
cudaDeviceSynchronize( );
```

A defined constant in one of the .h files
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.

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

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

```
float msecTotal = 0.0f;
status = cudaEventElapsedTime( &msecTotal, start, stop );
checkCudaErrors( status );
// compute and print the performance
double secondsTotal = 0.001 * (double)msecTotal;
double multsPerSecond = (float)SIZE * (float)NUMTRIALS / secondsTotal;
double megaMultsPerSecond = multsPerSecond / 1000000.;
fprintf( stderr, "Size = %10d, MegaMults/Second = %10.2lf\n", SIZE, megaMultsPerSecond );
```

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

```
// 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 );
  }
}
```

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

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