Notes on Programming Massively Parallel Processors

This is a documentation of my notes while studying from Kirk and Hwu's book on this topic.

1 Introduction to Data Parallelism and CUDA C

I have two hardware resources at my disposal - the *host*, which is the CPU, and the *device*, my GPU. A traditional C program is a CUDA program that contains only host code. Code meant for the device is marked with special CUDA keywords that are recognized by the `nvcc` compiler.

Thread Hierarchy

Data parallel functions in device code are called *kernels*. When a kernel function is called, it is executed by a large number of threads on the device. All the threads generated by a kernel launch constitute a *grid*. While the grid does its thing, the program can continue executing host code in parallel.

During kernel launch, the grid can be organized into blocks for the purpose of indexing threads. So, each block can be identified using up to three indices (depending on execution parameters). Thread block dimensionality can be accessed using `blockDim`.

Each thread in a grid has an associated `threadIdx`, which is a 3-component struct. So, threads can be identified using up to three-dimensional indices. So, for a block of size (N_x, N_y), the ID of a thread of index (i, j) is $(i + jN_x)$

A simple code for matrix addition using one block of threads is as follows.

```
__global__ void MatAdd(
        float A[n][n], float B[n][n], float C[n][n]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    // Kernel invocation with one block N^2 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(n, n);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

```
    ...
}
```

Kernel functions are declared using the `__global__` keyword before the regular C-style function declaration. Their calls are also 'decorated' using the *execution configuration syntax* $<<< \ldots >>>$. This is a pair of number of threads per block and number of blocks per grid, which can be either `int` or `dim3`.

The most obvious example of data-parallel execution is vector addition. How does one go about doing that in CUDA? consider the following 'pseudo code'

```
#include <cuda.h>
...
void vecAdd(float *A, float* B, float* C, int n) {
    int size = n * sizeof(float);

    // 1
    // allocate device memory for A, B and C
    // copy A and B to device/global memory

    // 2
    // kernel launch code - to have the device
    // perform the actual vector addition

    // 3
    // copy C from device memory
    // free device vectors
}
```

Note that the device and host memory are separate - I am assuming that they are also separate virtually. Upon some looking up online I found a nice answer on stackoverflow. Essentially, this is a fast moving field and what I write here today may not hold on tomorrow's heterogeneous computing systems.

Memory Allocation and Data Movement

Global/device memory is allocated using the function `cudaMalloc()`. From `man cudamalloc`

```
__cudart_builtin__ cudaError_t cudaMalloc (
        void** devPtr, size_t size)
```

The first argument is the the address of a pointer variable that will be set to point to the allocated object (casted to `void **`). The address of the allocated memory is written to this pointer variable. I do not know how exactly the global memory is virtualized, so, I will try not to do anything funny with it. The host code passes this pointer value to the kernels that need to access the global memory.

One of the reasons for going for this two argument malloc is so that the return value can be used for error handling (as can be seen in the function call). You can choose to discard the return values and treat it like a `void` return type at your own risk I guess.

Here's a demo usage

```
float *d_A;
int size = n * sizeof(float);
cudaMalloc((void**)&d_A, size);
...
cudaFree(d_A);
```

So, this is also where I learned that `sizeof` in C is a compile-time operator and it seems to be very unique in this regard. Notice how in the above listing `float` has been passed as an argument to sizeof. The keyword `float` does not compile down to some object that would have some binary representation that can be stored in the registers and caches of your CPU. The assembly code generated uses hardcoded values from what I am guessing are lookup tables used by the compiler.

Once device memory has been allocated, data objects can be moved to the device memory using `cudaMemcpy()`. So, keeping in mind that pointers to device memory locations are different for pointers to host memory locations, the following is the memory allocation and movement part of `vecAdd`.

```
void vecAdd(float *A, float *B, float* C, int n) {
    int size = n * sizeof(float);  // bytes
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudamemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // kernel invocation
    ...

    // retrieving result of computation from device to host
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Kernel Functions and Threading

CUDA programming is SPMD, so we are not limited to the same pipeline for all the data. So, we can go beyond vector addition.

Anyway. As mentioned before, a kernel launch spawns a grid of threads. There is another layer to this hierarchy - blocks. The grid is uniformly divided up into blocks of (up to 1024) threads. The number of threads spawned in each block is defined by the host code at the function call (keep it a multiple of 32). The same kernel can be launched with a variable number of threads at different locations in a program.

Each block has an identifier `blockIdx`. There are `blockDim` threads per block. Each thread in a block has an identifier `threadIdx`. So, data entry i of an array can be identified as

```
i = blockId.x*blockDim.x + threadIdx.x
```

The kernel function definition looks like this.

```
// each thread performs one scalar addition
__global__
void vecAddKernel(float *A, float *B, float *C, int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n) C[i] = A[i] + B[i];
}
```

The `.x` implies that there would be `.y` and `.z` too. Note that the block and thread identifiers are private to each thread, which means that the variables `i` is also private to each thread. `i` is called an *automatic variable.*

An `i < n` conditonal is required because the vector may not occupy the entirety of the last block.

The keyword list (executed on the <- only callable from the):

- `__device__ float DeviceFunc()`: device <- device, can only be called from a kernel function or another device function

- `__global__ void KernelFunc()`: device <- host

- `__host__ float HostFunc()`: host <- host

Now that the kernel function has been defined, it can be called.

```
int vecAdd(float* A, float *B, float *C, int n) {
    ...
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

The final code, `vecadd.cu` looks like so

```
1   #include <cuda.h>
2   #include "vecadd.h"
3
4   __global__
5   void vecAddKernel(float* A, float *B, float* C, int n) {
6       int i = threadIdx.x + blockDim.x*blockIdx.x;
7       if (i < n)
8           C[i] = A[i] + B[i];
9   }
10
11  void vecAdd(float *A, float *B, float *C, int n) {
12      int size = n * sizeof(float);
13      float *d_A, *d_B, *d_C;
14
15      cudaMalloc((void **) &d_A, size);
16      cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
17      cudaMalloc((void **) &d_B, size);
18      cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
```

```
19
20      cudaMalloc((void **) &d_C, size);
21
22      vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
23
24      cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
25
26      cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
27  }
```

Listing 1: Vector addition kernel function and kernel launch function

Epic. Note that `nvcc` requires a `main()` function and you cannot compile to relocatables individually. Or so it would seem, I have not yet looked into all the flags etc that can be passed to `nvcc`.

The following is a matrix addition code that does one scalar addition per thread.

```
1   #include <cuda.h>
2   #include "matrixAdd.h"
3
4   __global__
5   void matrixAddKernel_elementwise(
6           float *A, float *B, float *C, int n) {
7       // A, B, and C are device memory pointers
8       int i = blockDim.x*blockIdx.x + threadIdx.x;
9       int j = blockDim.y*blockIdx.y + threadIdx.y;
10      if (i < n && j < n) {
11          int k = n*i + j;
12          C[k] = A[k] + B[k];
13      }
14  }
15
16
17  void matrixAdd(float *A, float *B,
18          float *C, int n) {
19      int size = n * n * sizeof(float);
20      float *d_A, *d_B, *d_C;  // device memory pointers
21      const dim3 threadsPerBlock(16, 16);   // 256 threads per block
22      const dim3 numBlocks(ceil(n/threadsPerBlock.x),
23          ↪  ceil(n/threadsPerBlock.y));
24      cudaMalloc((void**) &d_A, size);
25      cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
26      cudaMalloc((void **) &d_B, size);
27      cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
28
29      cudaMalloc((void **) &d_C, size);
30
31      matrixAddKernel_elementwise<<<numBlocks, threadsPerBlock>>>(
32              d_A, d_B, d_C, n);
```

```
33
34      cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
35      cudaFree(d_A); cudaFree(d_B), cudaFree(d_C);
36  }
```

Listing 2: Kernel and host functions for performing matrix addition

Data-Parallel Execution Model

The execution is SPMD, with each thread using unique coordinates to access some region of the data-structure. A grid is a 3D-array of blocks and a block is a 3D array of threads.

Each block has a 3-vector `blockIdx`, and each thread has a 3-vector `threadIdx`, the components having identifiers `x`, `y`, `z`. dimensions of each block and each grid can be specified using variables of type `dim3`. If scalar dimension specifiers are used (e.g. `int`), then the `x` dimension is set to the scalar value while the `y` and `z` are set to `1`. A grid can have higher dimensionality then its blocks.

It might be a good practice to assign indices such that the largest of the dimensions comes first, e.g.

```
dim3 dimBlock(2, 2, 1);
dim3 dimGrid(4, 2, 2);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

This works better when mapping thread coordinates into data indices in accessing multidimensional arrays.

NVIDIA GPUs in 2012 could support up to 1024 threads per block with flexibility in how one can index them.

Mapping Threads To Multidimensional Data

The programmer needs to think about the dimensionality of the grid and blocks based on the nature of the data in consideration. E.g. an image is a 2D grid of values. If the image is greyscale, then we can do with a representation that takes a scalar value per pixel.

So, if the resolution of our image is 76×62, we could create a grid using 16×16 blocks, and let the grid be 5×4 blocks. 16×16 is a typical choice of block dimensions for 2D data structures. In this example, we are allocating 80×64 threads to process 76×62 pixels.

To avoid wasteful computations, there should be a conditional guard so that spawned threads in the grid that do not have pixel values to work with do not do any computations, like was done in the vector addition example, wherein four 256-thread blocks were spawned to deal with a vector of length 1000.

If you have been provided with parameters for a 2D array, e.g. the dimensions $m \times n$ for the image, then you can use something like this two spawn a grid for processing the image.

```
dim3 dimBlock(ceil(n/16.0), ceil(m/16.0));
dim3 dimGrid(16, 16, 1);
pictureKernel<<<dimGrid, dimBlock>>>(d_Pin, d_Pout, n, m);
```

So, here's the important bit. We have a global memory pointer `d_Pin` for our input data. As of 2012, CUDA C is based on ANSI C, which requires the number of rows and columns in 2D arrays to be known at compile time, which spells doom for most of our dynamically allocated datastructures. So, the programmer has to explicitly flatten the tensor to an equivalent 1D array.

The most up to date documents on NVIDIA are on C++. The C documents are 2010-ish old.The CUDA 11.0 guides on programming and best practices are exclusive to C++.

Checking C99 compatibility would require some testing. For now, all I need to know is that C arrays are stored in row-major order.

The following listing shows an example of a CUDA code for matrix multiplication for square matrices

```
1  #include <cuda.h>
2  #include "matrixMul.h"
3
4  __global__ void matrixMulSquareKernel(float *d_M, float *d_N,
5          float *d_P, int width) {
6      // P_{ij} = M_ikN_kj
7      int row = blockIdx.y * blockDim.y + threadIdx.y;
8      int col = blockIdx.x * blockDim.x + threadIdx.x;
9      if ((row < width && col < width)) {
10         int k;
11         int P_ind = row * width + col;
12         d_P[P_ind] = 0.0;
13         for (k = 0; k < width; k++) {
14             d_P[P_ind] += d_M[row*width + k]*d_N[k*width + col];
15         }
16     }
17 }
18
19 void matrixMulSquare(float *M, float *N, float *P, int width) {
20     const dim3 dimBlock(16, 16);
21     const int numBlocks = ceil(width/16.0);
22     const dim3 dimGrid(numBlocks, numBlocks);
23     float *d_M, *d_N, *d_P;
24
25     int sz = width*width * sizeof(float);
26
27     cudaMalloc((void **) &d_M, sz);
28     cudaMalloc((void **) &d_N, sz);
29     cudaMalloc((void **) &d_P, sz);
30
31     cudaMemcpy(d_M, M, sz, cudaMemcpyHostToDevice);
32     cudaMemcpy(d_N, M, sz, cudaMemcpyHostToDevice);
33
34     matrixMulSquareKernel<<<dimGrid, dimBlock>>>(d_M,
35             d_N, d_P, width);
36
37     cudaMemcpy(P, d_P, sz, cudaMemcpyDeviceToHost);
```

```
38    cudaFree(d_M); cudaFree(d_N); cudaFree(d_P);
39  }
```

Listing 3: Multiplication of square matrices

Synchronization and Transparent Scalability

In general one has to coordinate the execution of threads. CUDA allows threads
in the same block to coordinate using a barrier synchronization function `__syncthreads()`.
All threads in the calling block will be held until all threads reach the same lo-
cation in the program. I see potential deadlocks.

At the very least using this particular barrier synchronization would allow
different blocks to execute independent of each other.