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.

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.

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);
}
```