AMS 148 Chapter 3: Basics of CUDA C/C++ and Parallel Communication Patterns

Steven Reeves

In the last chapter we discussed the GPU programming model, the differences between CPUs and GPUs, and the basics of writing a simple program using CUDA. In this chapter we're going to build off of that. We will learn about important parallel communication patterns like scatter and gather and stencil. Additionally, we will dive a little deeper into the GPU hardware and learn about things like global memory and shared memory. Lastly, we'll put these together to learn how to write efficient GPU programs.

1 More Basics of CUDA C/C++

Recall that from the last chapter that a CUDA program is written in C or C++ with the CUDA extensions. To reiterate, the CPU is the called the host, and GPUs are referred to as devices. In CUDA, the host does the memory allocation, data copying and launches the CUDA kernel, essentially specifying the degree of parallelism. The device, does the computation thread by thread, remeber the device does not specify the degree of parallelism.

1.1 Kernel Configuration

Suppose we have the kernel from the last chapter, saxpy.

```
saxpy<<<1,64>>>(n, a, d_x, d_y)
```

In this kernel call we are launching one block of 64 threads. When a kernel is launched, the programmer specifies both the number of blocks and the number of threads per block. Suppose we want to run a larger problem than this. In terms of hardware, the GPU can run many blocks at once. However, depending on the age of the GPU used, there is limit to the number of threads one can launch. For newer GPUs, i.e. Pascal architectures used in Hummingbird, the thread limit is 1024. For older GPUs, like the Kepler cards in citrisdance, the limit is 512.

So, suppose we wanted to launch 1280 threads. The thread limits for both GPU types are below the number of threads we want to launch. A simple solution to this predicament is to launch 10 blocks, with 128 threads each:

```
saxpy<<<10,128>>>(n, a, d_x, d_y)
```

Or we can launch 5 blocks of 256 threads

```
saxpy<<<5,256>>>(n, a, d_x, d_y)
```

Given the multitude of options, you should pick the breakdown of threads and blocks that makes the most sense for your problem. Remember that there is a limit to the ammount of local and shared memory on a device, this can also influence the number of thread blocks to launch. For the purposes of saxpy, all three configurations are acceptable.

Each thread knows its index within its home block, and the index of the block. So within the kernel we can access the device array using the thread index and the block index. Notice that **blockDim**, appears in the following code. This variable specifies the width of the thread block in a specific direction.

```
tidx = threadIdx.x + blockDim.x*blockIdx.x
```

Now this is fine if our problem is one dimensional. The properties that we've discussed also generalize to 3D. In general a kernel launch will look like:

```
my_kernel<<<grid_dim,block_dim>>>(...)
```

Here grid_dim and block_dim are **dim** structures that can have up to 3 components. For example, we can call the saxpy kernel with 10 blocks and 128 threads by using the following code:

```
dim3 block_dim(128,1,1);
dim3 grid_dim(10,1,1);
saxpy<<<grid_dim,block_dim>>>(n, a, d_x, d_y)
```

We access the components of a `dim3` struct by using

```
int N, M, L;
dim3 foo(N,M,L);
std::cout<<foo.x<<std::endl; //<---- returns the value of N
std::cout<<foo.y<<std::endl; //<---- returns the value of M
std::cout<<foo.z<<std::endl; //<---- returns the value of L
```

So in the saxpy kernel when we see the code

```
tidx = threadIdx.x + blockDim.x*blockIdx.x
```

we know that `threadIdx.x` ranges from 0 to 127, `blockId.x` ranges from 0 to 9. Also, `blockDim.x` is equal to 128. In this case, `blockDim.y` and `blockDim.z` are irrelevant. So, `tidx` will range from 0 to 12799 in the kernel launch.

Suppose we want to do a matrix addition of two identically sized matrices $\mathbf{C} = \mathbf{A} + \mathbf{B}$, where $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{N \times M}$, in parallel with CUDA. For now we shall assume that NM is smaller than the thread limit on our GPU.

To begin we will start by designing our kernel:

```
//Kernel definition, Single Precision Matrix Addition
__global__ void MatAdd(float A[N][M], float B[N][M], float C[N][M])
{
        int idx = threadIdx.x + blockDim.x*blockIdx.x; //x index
        int idy = threadIdx.y + blockdim.y*blockIdx.y; //y index
        if(idx < N && idy < M)
                C[i][j] = A[i][j] + B[i][j];
}
```

Naturally matrix addition is 2D problem, so we have two indexes using the `threadIdx`, `blockDim`, and `blockIdx` structures.

Now let's complete the program:

```
#define N 128 //If you know the size of your matrix
#define M 128

//Kernel definition, Single Precision Matrix Addition
__global__ void MatAdd(float A[N][M], float B[N][M], float C[N][M])
{
...
}

int main()
{
//Host arrays
float A[N][M], B[N][M], C[N][M];
// Some way of loading data into A and B
...

//device arrays
float (*d_A)[N], (*d_B)[N], (*d_C)[N]; //Pointer to float
cudaMalloc((void**)&d_A,(N*M)*sizeof(float));
cudaMalloc((void**)&d_B,(N*M)*sizeof(float));
cudaMalloc((void**)&d_C,(N*M)*sizeof(float));

//copy host data to device arrays

cudaMemcpy(d_A, A, (N*M)*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, (N*M)*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_C, C, (N*M)*sizeof(float), cudaMemcpyHostToDevice);

dim3 grid_dim(4,4);
//We cant use 1,1 here because 128*128 = 16384 > 1024 or 512 so by using 4,4
//we have 1024 threads per block
dim3 block_dim(N/grid_dim.x, M/grid_dim.y);
//Note even if we use dim3, we don't have to specify each dimension, default is 1
```

```
//Call the kernel!
MatAdd<<<grid_dim, block_dim>>>(A, B, C);

//copy the data back onto the hose

cudaMemcpy(d_C, C, (N*M)*sizeof(float), cudaMemcpyDeviceToHost);

//Do some IO to retrieve data
 ...

//Free the arrays
cudaFree(pA);
cudaFree(pB);
cudaFree(pC);

//Note that in this problem the host variables were static, so no deallocation needed.
}
```

This is a simple matrix multiplication to illustrate the use of multidimensional indexing. However, in practice we often don't know a-priori the size of the matrix or image we wish to operate on. So we generally will not be static memory for these types of problems. We will come to a more advanced matrix addition later.

2 Parallel Communication

Parallel computing is about many threads solving a problem by working together. Often threads need to communicate to attack an instruction. In CUDA this communication is done in memory. Sometimes threads need to read from the same input location, write to the same output location, and sometimes threads need to exchange partial results.

2.1 Map, Gather, and Scatter

Let's brief on the different communication patterns seen in parrallel computing. Usually this is about how to map tasks, and memory together. That is mapping threads in CUDA and the memory that they are communicating through.

2.1.1 Map

The pattern, **map**, the program has many data elements: such as the elements of an array, entries in a matrix, or pixels in an image. Map requires the application of the same function of computation on each piece of data. This means that each thread will read from and write to a specific place in memory. Here there is a one to one correspondence between input and output. Recall that this is an embarassingly parallel type of task, where the entire application is concurrent. So map will be very efficient on GPUs and it is easily expressed in an effective way in CUDA - by simply having 1 thread do each task. This is how we parallelized the SAXPY algorithm. This however, isn't a very flexible frame work - there are many things we cannot do with just a simple map pattern.

2.1.2 Gather

Suppose you want to each thread to compute and store the average accross a range of data elements. For example, average a set of 4 elements together. In this case, each thread is going to read the values from 3 lcoations in memory and write them into a single place. Or suppose that you want to apply a blur to an image, by setting each pixel to the mean of its neighbouring pixels. Here the new pixel in each location would be the average of 5 pixels. This will be the operation we will do in the next homework assignment. This operation is called **gather**, because each thread gathers input data elements together from different places to compute an output under some operation.

2.1.3 Scatter

Now suppose that you want to do the opposite operation. We can have each thread read an input and take a fraction of its value and add it to the neighbouring points as an output result. When each thread needs to write its output in a different or multiple places we call this the **scatter** operation.

There's an obvious problem in scatter that we as programmers have to worry about. Each thread is simultaneously writing its result onto multiple places, causing overlap of outputs. We will discuss more parallel pitfalls to avoid later in the course.

2.1.4 Exercise

Suppose that we have a list of MotoGP racers. In this list, there are the names of the racers, the brand of motorcycle they ride (i.e. Ducati, Yamaha, Suzuki), and their fastest lap time around the Jerez circuit in Spain. If our task was to write each racer's record into its location in a sorted list with respect to lap time using a CUDA program, what type of operation/pattern is this?

- Map?
- Gather?
- Scatter?

Explain your answer.

2.2 Transpose

Transpose is a pattern that can be very useful in array, matrix, image, and data structure manipulation. For example we might have a 2D array, such as an image in row-major order.

1	2	3	4
5	6	7	8

In CUDA, the memory is laid as:

1	2	3	4	5	6	7	8

It may be advantageous to operate on the columns of the image instead.

1	5
2	6
3	7
4	8

So this means we have to reorder the elements in memory as such

1	5	2	6	3	7	4	8

Note to do this, each thread is reading from an adjacent element in the array, but writing to a scattered location in memory according to the stride of the rows in the transposed array.

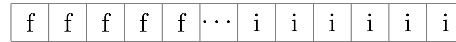
Suppose we have a structure:

```
struct foo {
        float f;
        int i;
};
foo array[1000];
```

Now, we have an array of thousand foos. In main memory it the array will look like this:

f	i	f	i	f	i	f	i	⋯	⋯	⋯	i

We see that we have floats and integers interspersed throughout memory. As will be discussed later, if we want to do operations on the floats, it is more efficient to access all the floats contiguously. So we want an operation that takes the above array and transmutes it into something like this:



This takes what is called an array of structures respresentation and turns it into a structure of arrays. These terms are used frequently and are abbreviated as AoS and SoA respectively. This is indeed a transpose as well.

2.2.1 Exercise

Label code snippets by pattern:

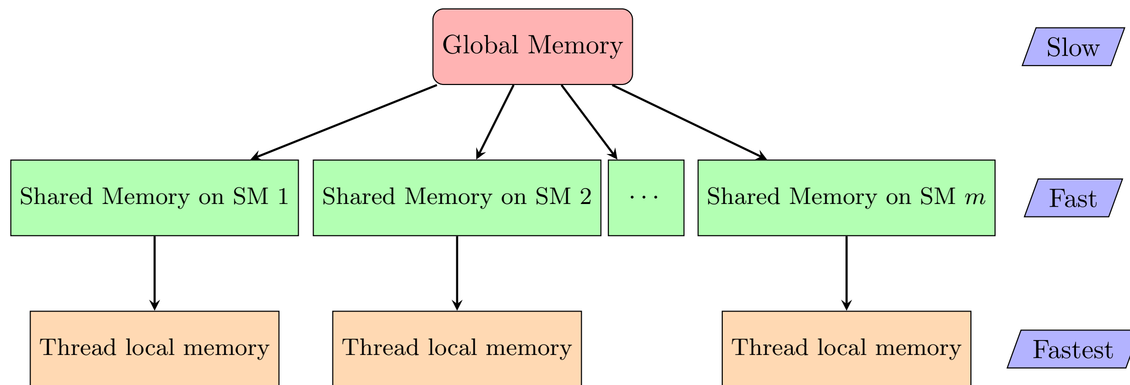
- A. Map
- B. Gather
- C. Scatter
- D. Transpose

```
float out[], in[];
int i = threadIdx.x;
int j = threadIdx.y;

const float pi = 3.1415;

out[i] = pi* in[i];

out[i + j*128] = in[j + i*128];
```

3 GPU Memory Model

Like cache on CPUs, Nvidia GPUs have a memory hierarchy that should be taken advantage of for the best GPU programs. Figure 1 shows a simplified schematic of the memory hierarchy on an Nvidia GPU. The figure shows the memory type from slowest to fastest, also biggest to smallest. The largest ammount of memory is the global memory. This memory is analogous to main memory of the CPU, and each kernel can access all of this memory. Because of the non-locality it is also the slowest to access.

Figure 1: Memory Heirarchy on an Nvidia GPU



Shared memory is memory that is local to the streaming multiprocessor. That is, each SM has a cache that is accessible by the CUDA cores on that SM. In practice, shared memory is local to a thread block, and

is accessed accordingly. Threads on another block or another SM will not be able to see this memory. Shared memory is an incredibly useful tool for localizing memory transactions that are necessary for computation in a thread block. For example on a Kepler series GPU (Tesla K20), shared memory bandwidth is $\sim$ 1.7TB/s where global memory is around 150GB/s. However, there is not an endless supply of shared memory, depending on the series of GPU, shared memory ranges from 16KB to 48KB.

The fastest memory type is thread local memory (registers). That is memory that is allocated within the kernel. Generally, small statically allocated data within the kernel is put on the thread local register via the compiler. This will be the fastest type of memory, but the smallest capacity. For example the Kepler card has up to 255 registers per thread. Each register is 4 bytes, so one can hold 255 single precision floats in each thread register. Now if you allocate more than this, you will not incur a segmentation fault, but the register will spill into global memory, which is slower than shared memory. If you know that you will be needing more than 255 floats per thread, I strongly encourage you to use shared memory, as the delay in register spilling will slow down your application.

3.1 Declaring Global Memory

Global memory arrays and data are declared using host pointers. Global memory is allocated using the host function `cudaMalloc`. For example:

```
float *d_A;
cudaMalloc((void**)&d_A,N*sizeof(float));
```

so we first declare a pointer named `d_A`, and then use `cudaMalloc` to allocate memory on the device. In this example, the `(void**)` term is not necessary in this example, but makes the memory allocation more general, especially when using pointers to pointers.

3.2 Declaring Shared Memory

There are two ways of declaring shared memory, externally when calling the kernel, or inside the CUDA kernel.

3.2.1 Static Shared Memory

Typically when shared memory is allocated within the CUDA kernel the programmer knows how much shared memory they wish to allocate. This is called **static** shared memeory. Lets consider a program in which we wish to reverse the order of an array. Quiz: what type of parallel programming communication pattern is this?

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

Notice in this kernel that declaration `__shared__` is used to signify that the memory being declared and allocated should be shared memeory and not local. Also note that in this kernel we use a funtion not used before, `__syncthreads()`. This function is a stopping point for each thread in a thread block. This way, there is no competition for loading into the array, disallowing a **race condition**. We will talk more about race conditions later in the course.

3.2.2 Dynamic Shared Memory

Dynamically allocated shared memory is typically used when the amount of shared memory is not konwn at compile time. In this case the shared memory allocation size per thread block must be specified (in bytes) using an optional third kernel execution configuration parameter:

```
dynamicReverse<<<1, n, n*sizeof(int)>>>(d_d,n);
```

And inside this kernel the `extern` declaration is used. So we declare a single extern that is unsized.

```
__global__ void dynamicReverse(int *d, int n)
{
  extern __shared__ int s[];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

Suppose we want to have multiple dynamically allocated shared memory arrays. In this case we declare one shared unsized array as before, and then use pointers to that array. That is:

```
extern __shared__ int s[];
int *integerData = s;                        // nI ints
float *floatData = (float*)&integerData[nI]; // nF floats
char *charData = (char*)&floatData[nF];      // nC chars
```

Note that in the kernel launch the programmer needs to be careful of the shared memory limit. If one over allocates the amount of shared memory, a compilation or runtime error will occur.

Here is a complete code that you can test on your own:

Listing 1: Array Reversal

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}

__global__ void dynamicReverse(int *d, int n)
{
  extern __shared__ int s[];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}

int main(void)
{
  const int n = 64;
  int a[n], r[n], d[n];

  for (int i = 0; i < n; i++) {
    a[i] = i;
    r[i] = n-i-1;
    d[i] = 0;
  }

  int *d_d;
  cudaMalloc(&d_d, n * sizeof(int));

  // run version with static shared memory
  cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
  staticReverse<<<1,n>>>(d_d, n);
  cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);

  // run dynamic shared memory version
  cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
  dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);
  cudaMemcpy(d, d_d, n * sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}
```

3.2.3 Shared Memory Bank Conflicts

To achieve high memory bandwidth for concurrent accesses, shared memory is divided into equally sized memory modules (banks) that can be accessed simultaneously. So, any memory store of m addresses that spans x distinct memory banks can be serviced simultaneously, yielding an effective memory bandwidth that is m times as high as the bandwidth of a single bank.

A word of warning: If multiple threads' requested addresses map to the same memory bank, the accesses are serialized. The GPU will split the conflicting memory accesses into as many conflict-free sequential requests as necessary. This can dramatically slow down a GPU code.

We will examine how to minimize shared memory bank conflicts later in the course when we consider optimization and debugging.

In summary, shared memory is a powerful feature for writing well optimized CUDA code. Access to shared memory is much faster than glocal memory, because it is located near the SMs. Because shared memeory is shared by threads in a block, it provides a mechanism for threads to cooperate. One way to use shared memory that leverages such thread cooperation is to enable global memory **coalescing**, as demonstrated by the array reversal example. By reversing the array using shared memory we are able to have all global memory reads and writes performed with unit stride, achieving full coalescing on any CUDA GPU.

4 Exercise List

These exercises are computational, please turn in your CUDA code, with comments.

4.1 CUDA Matrix-Vector Product

Use your algorithm in homework 1 to write a parallel matrix-vector product in CUDA. For extra credit, integrate the use of shared memory. Compare the execution times against a CPU implementation. For simplicity use a square matrix, and vector with a comparable size. Use: $N = 16$, 128, 1024, 2048, 65536.

4.2 CUDA Matrix-Transpose

Write two matrix transpose CUDA codes. The first without using shared memory and the second using shared memory. Compare the time to completion with various sizes of square matrices: $N = 16$, 128, 1024, 2048, 65536.