Worked example 1: A first CUDA program and execution through the scheduler

Let us start with a straightforward C++ program to add the elements of two arrays with a million elements each:

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements. 1 should be left shifted 20 times

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

// Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

Save this in a file called add.cpp and then compile with a C++ compiler. For the GNU compiler:

g++ add.cpp -o add.x

Running it:
./add.x

Gives the result:
Max error: 0.000000

As expected, it prints that there was no error in the summation and then exits. To get this computation running (in parallel) on the many cores of a GPU, we first have to turn our add function into a function that the GPU can run, called a kernel in CUDA.

To do this, we have to do is add the specifier __**global**__ to the function, which tells the CUDA C++ compiler that this is a function that runs on the GPU and can be called from CPU code.

```
// CUDA Kernel function to add the elements of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

These __global__ functions are known as kernels, and code that runs on the GPU is often called **device** code, while code that runs on the CPU is **host** code.

To compute on the GPU, we also need to allocate memory accessible by the GPU.

Unified Memory in CUDA makes this easy by providing a single memory space accessible by all GPUs and CPUs in your system.

To allocate data in unified memory, call **cudaMallocManaged()**, which returns a pointer that you can access from **host** (CPU) code or **device** (GPU) code. To free the data, just pass the pointer to **cudaFree()**.

```
// Allocate Unified Memory -- accessible from CPU or GPU
  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  ...

  // Free memory
  cudaFree(x);
  cudaFree(y);
```

Finally, we need to launch the add() kernel, which invokes it on the GPU.

CUDA kernel launches are specified using the triple angle bracket syntax **<<< >>>**. We add it to the call to **add** before the parameter list.

add<<<1, 1>>>(N, x, y);

This line launches one GPU thread to run add().

Just one more thing: I need the CPU to wait until the kernel is done before it accesses the

results (because CUDA kernel launches don't block the calling CPU thread). To do this I just call cudaDeviceSynchronize() before doing the final error checking on the CPU.

Here's the complete code:

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y;

  // Allocate Unified Memory – accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

CUDA files should have the file extension **.cu** so save this as the file **add.cu**
As it is a CUDA file, it needs the be compiled with **nvcc** , the NVIDIA C++ compiler.

On the ARC3 system, the nvcc compiler is part of the cuda module, so this needs to be loaded first:

module load cuda

and needs to be loaded when CUDA code is compiled or run.

Note: You can compile CUDA code on any node, so it's OK to compile on a login node that doesn't have any GPUs. CUDA code must be executed on a GPU node though.

So to compile it:

nvcc add.cu -o add_cuda.x

To run it though, we'll need to submit it as a job to one of the GPU nodes. A typical submission script will look like:

```
#$ -cwd -V
#$ -l h_rt=00:10:00
#$ -l coproc_p100=1

module load cuda
./add_cuda.x
```

Save this with an appropriate name (say submit.sh) and then submit to the queue:

qsub submit.sh

Inspecting the output file that the job generates will give the same answer as before.

This is only a first step, because as written, this kernel is only correct for a single thread, since every thread that runs it will perform the add on the whole array. Moreover, there is a race condition since multiple parallel threads would both read and write the same locations.

Also, you've only got my word for it that the code actually ran on the GPU at all, so we need to explore in more detail what the code is doing.

The simplest way to find out how long the kernel takes to run is to run it with nvprof, the command line GPU profiler that comes with the CUDA Toolkit. To profile, use:

nvprof ./add_cuda.x

In the submission script. The output from nvprof will appear in the e file this time, something like:

```
==54067== NVPROF is profiling process 54067, command: ./add_cuda.x
==54067== Profiling application: ./add_cuda.x
==54067== Profiling result:
            Type  Time(%)      Time  Calls       Avg    Min       Max     Name
 GPU activities:  100.00%  217.52ms      1  217.52ms  217.52ms  217.52ms  add(int,
float*, float*)
```

So it takes about ⅕ of a second on one of the P100's on ARC3 (interestingly, the same code takes a bit longer on one of the K80 nodes).

To make it faster though:

The key is in CUDA's <<<1, 1>>>syntax. This is called the execution configuration, and it tells the CUDA runtime how many parallel threads to use for the launch on the GPU.

There are two parameters here, but let's start by changing the second one: the number of threads in a thread block. CUDA GPUs run kernels using blocks of threads that are a multiple of 32 in size, so 256 threads is a reasonable size to choose.

add<<<1, 256>>>(N, x, y);

Running the code with only this change will do the computation once per thread, rather than spreading the computation across the parallel threads.

To do it properly, we need to modify the kernel. CUDA C++ provides keywords that let kernels get the indices of the running threads. Specifically, **threadIdx.x** contains the index of the current thread within its block, and **blockDim.x** contains the number of threads in the block. Modify the loop to stride through the array with parallel threads.

So the add function becomes:

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

In fact, setting index to 0 and stride to 1 makes it semantically identical to the first version.

This time, save the complete code as **add_block.cu** then compile, run and profile through the scheduler as before.

This time round:

==54696== Profiling result:
Type Time(%) Time Calls Avg Min Max Name
GPU activities: 100.00% 7.4534ms 1 7.4534ms 7.4534ms 7.4534ms add(int, float*, float*)

Blocks and Streaming Multiprocessors

CUDA GPUs have many parallel processors grouped into Streaming Multiprocessors, or SMs. Each SM can run multiple concurrent thread blocks. As an example, a Tesla P100 GPU based on the Pascal GPU Architecture has 56 SMs, each capable of supporting up to 2048 active threads. To take full advantage of all these threads, we could launch the kernel with multiple thread blocks.

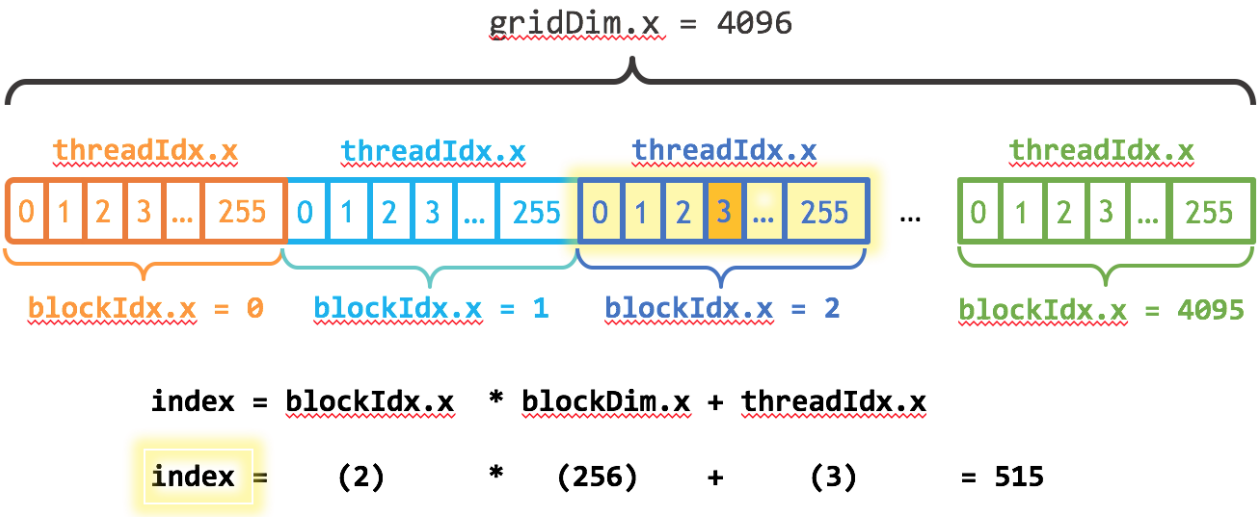
The first parameter of the execution configuration specifies the number of thread blocks.

Together, the blocks of parallel threads make up what is known as the grid. Since we have N elements to process, and 256 threads per block, we need to calculate the number of blocks to get at least N threads. To do this, divide N by the block size (being careful to round up in case N is not a multiple of blockSize).

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

Finally, we also need to update the kernel code to take into account the entire grid of thread blocks. CUDA provides **gridDim.x**, which contains the number of blocks in the grid, and **blockIdx.x**, which contains the index of the current thread block in the grid.

The diagram illustrates the the approach to indexing into an array (one-dimensional) in CUDA using blockDim.x, gridDim.x, and threadIdx.x.



The idea is that each thread gets its index by computing the offset to the beginning of its block (the block index times the block size: **blockIdx.x * blockDim.x**) and adding the thread's index within the block (**threadIdx.x**). The code `blockIdx.x * blockDim.x + threadIdx.x` is classic CUDA.

The final exercise is to consider how to specify the size of the total grid to distribute these threads more efficiently across the CPU.

Based on: https://devblogs.nvidia.com/even-easier-introduction-cuda/ accessed 12/12/2017