CUDA, Supercomputing for the Masses: Part 2

A first kernel

April 29, 2008
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/207402986

In Part 1 of this article series, I presented a simple first CUDA (short for "Compute Unified Device Architecture") program called moveArrays.cu to familiarize you with the CUDA tools for building and executing programs. For C programmers, this program did nothing more than call the CUDA API to allocate memory and move data to and from the CUDA device. Nothing new was added that might cause confusion in learning how to use the tools to build and run a CUDA program.

This article builds on that first example by adding a few additional lines of code to perform a simple calculation on the CUDA device -- specifically incrementing each element in a floating-point array by 1. Amazingly, this example already provides the basic framework ("move data to CUDA-enabled device(s), perform a calculation and retrieve result") for solving many problems with CUDA!

Before tackling more advanced topics, you first need to understand:

- What is a kernel? A kernel is a function callable from the host and executed on the CUDA device -- simultaneously by many threads in parallel.
- How does the host call a kernel? This involves specifying the name of the kernel plus an execution configuration. For the purposes of this column, an execution configuration just means defining the number of parallel threads in a group and the number of groups to use when running the kernel for the CUDA device. This is actually an important topic that will be discussed in greater depth in future columns.
- How to synchronize kernels and host code.

At the top of the Listing One (incrementArrays.cu), we see an example host routine, `incrementArrayOnHost` and our first kernel, `incrementArraysOnDevice`.

The host function `incrementArrayOnHost` is just a simple loop over the number of elements in an array to increment each array element by one. This function is used for comparison purposes at the end of this code to verify the kernel performed the correct calculation on the CUDA device.

Next in the Listing One is our first CUDA kernel, `incrementArrayOnDevice`. CUDA provides several extensions to the C-language. The function type qualifier `__global__` declares a function as being an executable kernel on the CUDA device, which can only be called from the host. All kernels must be declared with a return type of `void`.

The kernel `incrementArrayOnDevice` performs the same calculation as `incrementArrayOnHost`. Looking within `incrementArrayOnDevice`, you see that there is no loop! This is because the function is simultaneously executed by an array of threads on the CUDA device. However, each thread is provided with a unique ID that can be used to compute different array indicies or make control decisions (such as not doing anything if the thread index exceeds the array size). This makes `incrementArrayOnDevice` as simple as calculating the unique ID in the register variable, `idx`, which is then used to uniquely reference each element in the array and increment it by one. Since the number of threads can be larger than the size of the array, `idx` is first checked against `N`, an argument passed to the kernel that specifies the number of elements in the array, to see if any work needs to be done.

So how is the kernel called and the execution configuration specified? Well, control flows sequentially through the source code starting at main until the line right after the comment containing the statement `Part 2 of 2` in Listing One.

```
// incrementArray.cu
#include <stdio.h>
#include <assert.h>
#include <cuda.h>
void incrementArrayOnHost(float *a, int N)
{
  int i;
  for (i=0; i < N; i++) a[i] = a[i]+1.f;
}
__global__ void incrementArrayOnDevice(float *a, int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx]+1.f;
}
int main(void)
{
  float *a_h, *b_h;           // pointers to host memory
  float *a_d;                 // pointer to device memory
  int i, N = 10;
  size_t size = N*sizeof(float);
  // allocate arrays on host
  a_h = (float *)malloc(size);
  b_h = (float *)malloc(size);
  // allocate array on device
  cudaMalloc((void **) &a_d, size);
  // initialization of host data
  for (i=0; i<N; i++) a_h[i] = (float)i;
  // copy data from host to device
  cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
  // do calculation on host
  incrementArrayOnHost(a_h, N);
  // do calculation on device:
  // Part 1 of 2. Compute execution configuration
  int blockSize = 4;
  int nBlocks = N/blockSize + (N%blockSize == 0?0:1);
  // Part 2 of 2. Call incrementArrayOnDevice kernel
  incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d, N);
  // Retrieve result from device and store in b_h
  cudaMemcpy(b_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // check results
  for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);
  // cleanup
  free(a_h); free(b_h); cudaFree(a_d);
```

`Listing One`: incrementArrays.cu.

This queues the launch of `incrementArrayOnDevice` on the CUDA-enabled device and illustrates another CUDA addition to the C-language, an asynchronous call to a CUDA kernel. The call specifies the name of the kernel and the execution configuration enclosed between triple angle brackets "<<<" and ">>>". Notice the two parameters that specify the execution configuration: `nBlocks` and `blockSize`, which will be discussed next. Any arguments to the kernel call are provided via a standard C-language argument list for a function delimited in the standard C-language fashion with "(" and ")". In this example, the pointer to the device global memory `a_d` (which contains the array elements) and `N` (the number of array elements) are passed to the kernel.

Since the CUDA device is idle, the kernel immediately starts running based on the execution configuration and according to the function arguments. Meanwhile, the host continues to the next line of code after the kernel launch. At this point, both the CUDA device and host are simultaneously running their separate programs. In the case of incrementArrays.cu, the host immediately calls `cudaMemcpy`, which waits until all threads have finished on the device (e.g., returned from `incrementArrayOnDevice`) after which it pulls the modified array back to the host. The program completes after the host system performs a sequential comparison to verify we got the same result on the parallel CUDA device with `incrementArrayOnDevice` as on the host with the sequential version `incrementArrayOnHost`.

There are several variables determined at kernel startup through the execution configuration (in this example via the variables `nBlocks` and `blockSize` contained between the triple angle brackets "<<<" and ">>>") that are available to any kernel. The thinking behind `nBlocks` and `blockSize` is actually quite elegant because it allows the developer to account for hardware limitations without requiring the recompilation of the application -- which is an essential feature for developing commercial software with CUDA.

As I'll examine in future columns, threads within a block have the ability to communicate and synchronize with each other. This is a marvelous software feature that unfortunately costs money from a hardware standpoint. Expect more expensive (and future) devices to support a greater number of threads per block than less expensive (and older) devices. The grid abstraction was created to let developers take into account -- without recompilation -- differing hardware capabilities regardless of price point and age. A grid, in effect, batches together calls to the same kernel for blocks with the same dimensionality and size, and effectively multiplies by a factor of `nBlocks` the number of threads that can be launched in a single kernel invocation. Less capable devices may only be able to run one or a couple of thread blocks simultaneously, while more capable (e.g., expensive and future) devices may be able to run many at once. Designing software with the grid abstraction requires balancing the trade-offs between simultaneously running many independent threads, and requiring a greater number of threads within a block that can cooperate with each other. Please be cognizant of the costs associated with the two types of threads. Of course, different algorithms will impose different requirements, but when possible try to use larger numbers of thread blocks.

In the kernel on the CUDA-enabled device, several built-in variables are available that were set by the execution configuration of the kernel invocation. They are:

- `blockIdx` which contains the block index within the grid.
- `threadIdx` contains the thread index within the block.
- `blockDim` contains the number of threads in a block.

These variables are structures that contain integer components of the variables. Blocks, for example, have `x`-, `y`-, and `z`- integer components because they are three-dimensional. Grids, on the other hand, only have `x`- and y-components because they are two-dimensional. This example only uses the `x`-component of these variables as the array we moved onto the CUDA device is one-dimensional. (Future columns will explore the power of this two-dimensional and three-dimensional configuration capability and how it can be exploited.)

Our example kernel used these built-in variables them to calculate the thread index, `idx` with the statement:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

The variables `nBlocks` and `blockSize` are the number of blocks in the grid and the number of threads in each block, respectively. In this example, they are initialized just before the kernel call in the host code:

```
int blockSize = 4;
int nBlocks = N/blockSize + (N%blockSize == 0?0:1);
```

In cases where `N` is not evenly divisible by `blockSize`, the last term in the `nBlocks` calculation adds an extra block, which for some cases implies some threads in the block will not perform any useful work.

This example is obviously contrived for simplicity as it assumes that the array size is smaller than the number of threads that can be contained within four (4) thread blocks. This is an obvious oversimplification but it let us, with simple code, explore the kernel call to `incrementArrayOnDevice`.

It is important to emphasize that each thread is capable of accessing the entire array `a_d` on the device. There is no inherent data partitioning when a kernel is launched. It is up to the programmer to identify and exploit the data parallel aspects of the computation when writing kernels.

Figure 1 illustrates how `idx` is calculated and the array, `a_d` is referenced. (If any of the preceding text is unclear, I recommend adding a `printf` statement to `incrementArrayOnDevice` to print out `idx` and the associated variables used to calculate it. Compile the program for the emulator, "make emu=1", and run it to see what is going on. Be certain to specify the correct path to the emulator executable to see the `printf` output.)

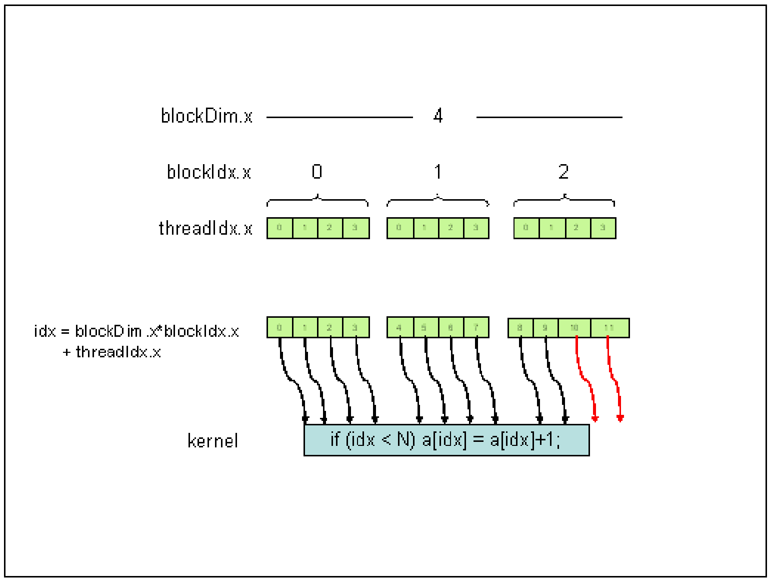
blockDim.x
4
blockIdx.x
0
1
2
threadIdx.x
idx = blockDim.x*blockIdx.x + threadIdx.x
kernel
if (idx < N) a[idx] = a[idx]+1;

Figure 1

Again, kernel calls are asynchronous -- after a kernel launch, control immediately returns to the host CPU. The kernel will run on the CUDA device once all previous CUDA calls have finished. The asynchronous kernel call is a wonderful way to overlap computation on the host and device. In this example, the call to `incrementArrayOnHost` could be placed after the call to `incrementArrayOnDevice` to overlap computation on the host and device to get better performance. Depending on the amount of time the kernel takes to complete, it is possible for both host and device to compute simultaneously.

Until the next column, I recommend:

- Try changing the value of `N` and `nBlocks`. See what happens when they exceed the device capabilities.
- Think about how to introduce a loop to handle arbitrary sized arrays.
- Distinguish between the types of CUDA-enabled device memory (e.g., global memory, registers, shared memory, and constant memory). Take a look at the CUDA occupancy calculator and either the `nvcc` options `-cubin` or `--ptxas-options=-v` to determine the number of registers used in a kernel.

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

For More Information

- CUDA, Supercomputing for the Masses: Part 14
- CUDA, Supercomputing for the Masses: Part 13
- CUDA, Supercomputing for the Masses: Part 12
- CUDA, Supercomputing for the Masses: Part 11
- CUDA, Supercomputing for the Masses: Part 10
- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1