Using CUDA on HokieSpeed

CMDA 3634: Computer Science Foundations for Computational Modeling and Data Analytics
Presented by John Burkardt, Advanced Research Computing, 28 March 2017

The ARC HokieSpeed cluster has 204 nodes, each with two NVIDIA Fermi GPU's. This discussion shows how a user CUDA program can be compiled and executed to take advantage of the GPU for extremely fast execution.

1 Exercise: Login to HokieSpeed, Copy Examples

Log into either HokieSpeed login node *hokiespeed1* or *hokiespeed2* with your PID and password:

```
ssh -X PID@hokiespeed1.arc.vt.edu
```

The **-X** switch is needed for any graphics interaction, including the use of **emacs**.

Copy the example files from my directory on HokieSpeed:

```
cp ~burkardt/demo_cuda/* .
```

2 Discussion: Module Command for CUDA

Before accessing CUDA, you must issue a `module` command. To load the default 5.0.35 version, just type:

```
module load cuda
```

This command defines some symbolic names, including `CUDA_DOC` and `CUDA_BIN`.

`CUDA_DOC` has CUDA documentation in *html*, *man* and *pdf* formats. Try a command like:

```
ls $CUDA_DOC/pdf
```

`CUDA_BIN` is the directory with CUDA binaries (executable programs). If we list this directory, we see the following:

```
bin2c*          cudafe++*                       cuda-memcheck*  nsight*        nvlink*   ptxas*
computeprof@    cuda-gdb*                       cuobjdump*      nvcc*          nvprof*   uninstall_cuda_6.
crt/            cuda-gdbserver*                 fatbinary*      nvcc.profile   nvprune*
cudafe*         cuda-install-samples-6.5.sh*    filehash*       nvdisasm*      nvvp*
```

All these files are now "in your path"; that is, you can run any of these programs simply by typing the name. The program that is of most interest to us is `nvcc`, the CUDA compiler.

3 Exercise: Hello, World!

One of the example files that you copied is called *hello.cu*. The `.cu` extension indicates that this is the text of a C program that includes some CUDA extensions. Take a look at the program text using `more` or `emacs`.

While the file has a C-like structure, several features that indicate that we are working with CUDA. The function doing the work, called the **kernel** function, is declared with the `__global__` statement; the main program calls the kernel using an unfamiliar triple angle brackets notation `<<< blocks, threads >>>`; inside the kernel, there is no loop, but we seem to refer to loop indices using structure variables `blockIdx.x` and `threadIdx.x`.

For now, you can think of this program an unusual way of writing:

```
for ( j = 0; j < blocks; j++ )       <-- j is blockIdx.x
{
  for ( i = 0; i < threads; i++ )    <-- i is threadIdx.x
```

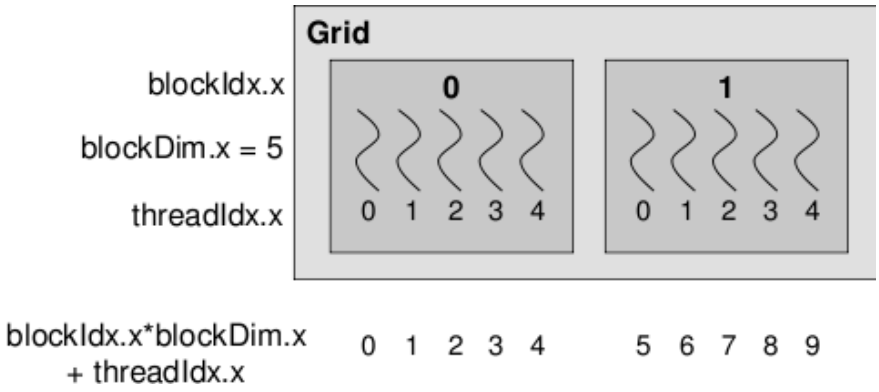

Figure 1: 2 blocks and 5 threads per block makes 10 threads.

```
  {
    k = threadIdx.x + blockDim.x * blockIdx.x;
    printf ( "Hello #%d from CUDA block %d, thread %d\n", k, j, i );
  }
}
```

We compile a CUDA code with the `nvcc` command. Because *hello.cu* wants to print from the GPU, we need an extra switch on the compile line to guarantee we have at least "compute capability 2.0", which is the point at which GPU's were given this ability;

```
nvcc -o hello -arch=compute_20 hello.cu
```

We run the program on the CPU, but it will automatically include the GPU in its work:

```
./hello
```

Now `blocks` was 2, and `threads` was 5, explaining why we get 10 messages, with 2 blocks between 0 and 1, and 5 threads between 0 and 4, as though we had executed a `for` loop.

In general, **you should not run your GPU programs interactively.** There are 204 compute nodes available, but only two login nodes, which must be shared by many users. Use the batch system:

```
qsub hello.sh
```

4 Discussion: vecadd.cu moves data to and from the GPU

The file *vecadd.cu* adds two vectors, and demonstrates how data can be copied from the CPU to the GPU, and results copied back. The CPU defines and sets two vectors A and B, and then expects the GPU to compute the sum $C = A + B$. Here is the "interesting" portion of the main program:

```
float *a_cpu, *a_gpu, *b_cpu, *b_gpu, *c_cpu, *c_gpu;
int memsize;
int n = 10;

memsize = n * sizeof ( float );
a_cpu = ( float * ) malloc ( memsize );
b_cpu = ( float * ) malloc ( memsize );
loadArrays ( a_cpu, b_cpu, n );

cudaMalloc ( ( void** ) &a_gpu, memsize );
cudaMalloc ( ( void** ) &b_gpu, memsize );
cudaMalloc ( ( void** ) &c_gpu, memsize );
```

```
  cudaMemcpy ( a_gpu, a_cpu, memsize, cudaMemcpyHostToDevice );
  cudaMemcpy ( b_gpu, b_cpu, memsize, cudaMemcpyHostToDevice );

  add_vectors <<< 1, n >>> ( a_gpu, b_gpu, c_gpu );

  c_cpu = ( float * ) malloc ( memsize );
  cudaMemcpy ( c_cpu, c_gpu, memsize, cudaMemcpyDeviceToHost );
```

The kernel function is:

```
__global__ void add_vectors ( float *a_gpu, float *b_gpu, float *c_gpu )
{
  int k = threadIdx.x + blockDim.x * blockIdx.x;
  c_gpu[k] = a_gpu[k] + b_gpu[k];
}
```

Since `blocks is 1`, we could write instead `k = threadIdx.x`; however, the way we have written it will work just as well if we use `blocks=n` and `threads=1`, or any pair of values that multiply to `n=10`.

For a calculation in which both `blocks` and `threads` are scalars (in other words, not vectors!), the following values are all that the kernel needs to figure things out:

- `threadIdx.x`, the current thread index;
- `blockDim.x`, the number of threads per block;
- `blockIdx.x`, the current block index;
- `gridDim.x`, the number of blocks.

The main new feature in *vecadd* is how the user's program must manage the allocation and assignment of memory on the GPU, as well as transfers back and forth. Notice the following features of the program:

1. The arrays `a`, `b` and `c` will "live" on both the CPU and GPU. The CPU is responsible for making space for them on the GPU using the function `cudaMalloc()`.
2. The arrays `a` and `b` are set on the CPU, and copied to the GPU arrays using `cudaMemcpy()`.
3. The `add_vectors` kernel is run on the GPU, invoked, with a specific number of blocks and threads.
4. Once all the threads have completed the computation on the GPU, the function `cudaMemcpy()` copies the GPU results back to the CPU array `c`.

5 Exercise: Run vecadd.cu in batch

Now we want to compile and run the *vecadd.cu* program through the batch system. This requires a batch file which includes all the commands we might give interactively, preceded by some commands to the scheduler.

A new feature in these scheduler commands is that we ask for 1 node, 1 core, and 1 gpu:

```
#! /bin/bash
#PBS -l walltime=00:05:00
#PBS -l nodes=1:ppn=1:gpus=1
#PBS -W group_list=hokiespeed
#PBS -q normal_q
#PBS -j oe

cd $PBS_O_WORKDIR

module purge
module load cuda

nvcc -o vecadd vecadd.cu
./vecadd
```

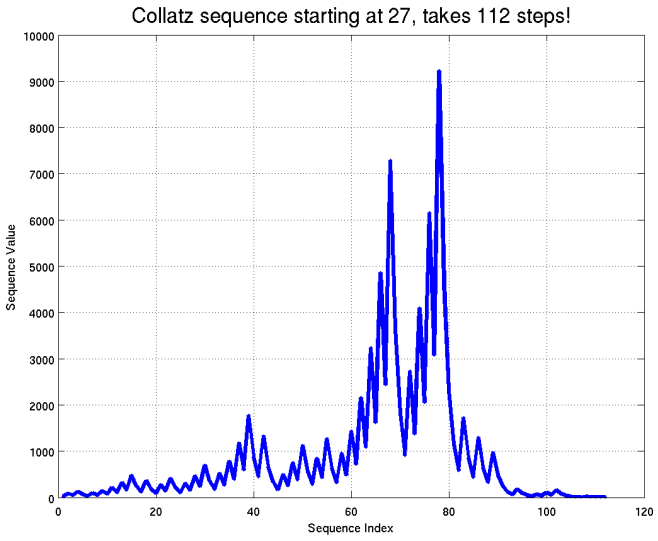

Figure 2: The Collatz sequence for 27 takes 112 steps.

You can submit this job on HokieSpeed in the usual way:

```
qsub vecadd.sh
```

6 Discussion: Collatz Sequences

The mathematician Lothar Collatz became interested in something called the hailstone game. Start with any positive number, divide it by 2 if even, or triple and add 1 if odd, and stop when you reach the value 1. It's not clear why, but the process always does seem to reach 1. For example, suppose we start with the number 17:

```
 0  17 is odd,  so 17 --> 3*17+1 = 52;
 1  52 is even, so 52 --> 52/2   = 26;
 2  26 is even, so 26 --> 26/2   = 13;
 3  13 is odd,  so 13 --> 3*13+1 = 40;
 4  40 is even, so 40 --> 40/2   = 20;
 5  20 is even, so 20 --> 20/2   = 10;
 6  10 is even, so 10 --> 10/2   =  5;
 7   5 is odd,  so  5 --> 3*5+1  = 16;
 8  16 is even, so 16 --> 16/2   =  8;
 9   8 is even, so  8 --> 8/2    =  4;
 0   4 is even, so  4 --> 4/2    =  2;
11   2 is even, so  2 --> 2/2    =  1;
12   1 is 1, so we stop after 12 steps.
```

We say that 17 has a Collatz sequence of length 12. An interesting question is, for any starting value **n**, how many iterations does it take before the sequence reaches 1 (or the hailstone hits the ground?) For 17, this value is 12; for 1, this value is 0. What about 27?

We can set up a CUDA program to compute the length of the Collatz sequence for every integer from 1 to N. Notice that, in this case, there is no need for the CPU to initialize a data array to be sent to the GPU; the GPU simply has to run through each possible starting value, compute the sequence, record its length, and return the array of sequence lengths to the CPU.

The interesting part of the program is:

```
steps_num = 100;
steps_size = steps_num * sizeof ( int );
cudaMalloc ( ( void** ) &steps_gpu, steps_size );

blocks = 25;
threads = 4;
collatz_steps <<< blocks, threads >>> ( steps_gpu );

steps_cpu = ( int * ) malloc ( steps_size );
cudaMemcpy ( steps_cpu, steps_gpu, steps_size, cudaMemcpyDeviceToHost );
```

after which we report on the starting value that resulted in the longest Collatz sequence that we observed. The kernel is set up to work for any combination of (scalar) blocks and threads. The interesting part is:

```
k = threadIdx.x + blockDim.x * blockIdx.x;
n = k + 1;
s = 0;
while ( 1 < n ){
  if ( ( n % 2 ) == 0 ){
    n = n / 2;
  }else{
    n = 3 * n + 1;
  }
  s = s + 1;
}
steps_gpu[k] = s;
```

7 Exercise: Batch collatz.cu

Look over the text of *collatz.cu* and convince yourself that you understand how:

- the CPU uses `cudaMalloc()` to set up memory on the GPU for the array of sequence lengths,
- the kernel function is called with 25 blocks and 4 threads per block, to do $n = 100$ calculations;
- the CPU calls `cudaMemcpy()` to copy the step length array from the GPU back to the CPU;
- the kernel function carries out task `K`, to check integer `N=K+1`, where `k=threadIdx.x+blockDim.x*blockIdx.x`.

Once you've thought about these ideas, go ahead and submit the batch file *collatz.sh*; check out its report on the starting value that resulted in the longest sequence.

8 Discussion: Julia Sets

Suppose we have a function $f(z)$ defined for any complex number z. Suppose that we repeatedly carry out the iteration $z \leftarrow f(z)$. It's natural to assume that the sequence of z values will blow up in magnitude; however it's possible that for some starting values z, the sequence will stay small. We call such points the **Julia set for** $f()$.

Computer scientists are interested in Julia sets because they can produce some surprisingly beautiful plots and mathematicians love them for reasons only mathematicians can understand.

A simple class of test functions has the form $f(z) = z^2 + c$, and some interesting values of the complex number c include 0.4+0.6i, 0.285+0.01i, and 0.7269+0.1889i. For our experiment, we will use $c = -0.8 + 0.156i$. We will only look at points in the rectangle $[-1.5, +1.5]x[-1.5, +1.5]$. We will pick a 1000x1000 grid in this rectangle, and for each $z_{i,j}$ we will iterate 200 times. If the norm of the iterates stays below 1,000, we will consider that it is a member of the Julia set. So our result will be a 1000x1000 array of 0's (not in the set) and 1's (in the set).

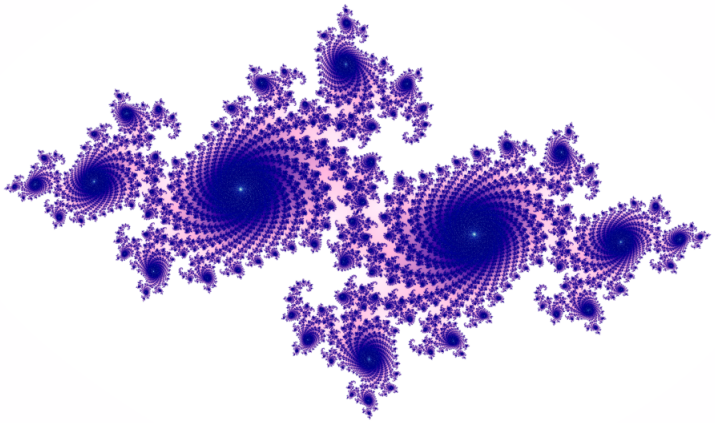

Figure 3: An example of a Julia set plot.

9 Exercise: julia_gpu.cu

The program *julia_gpu.cu* carries out a computation of the Julia set over the region R=[-1.5,+1.5]x[-1.5,+1.5]. It uses a 1,000x1,000 array of points within this region, checking each one to see if it is in the Julia set.

The results, an array of 0's and 1's, will then be reformatted on the CPU for display as a graphics image.

How does the GPU participate in this calculation? Unlike the vector addition example, there is no large data array that the CPU has to send to the GPU; the only information is some scalars that define the shape of the region and the shape of the grid to be filled in.

The CPU calls the kernel function `julia_set`, but this time, the angle bracket parameters are `<<<blocks,threads>>>`, where threads is simply 1, but `blocks` is a vector of type `dim3`, with value $(w, h, 1)$ which tells the GPU that the data is physically arranged in a $w \times h \times 1$ array, so that each invocation of the kernel will get indices (i, j, k) (with k always 0), telling it which pixel to work on. The individual results are packed into an array and sent back to the CPU.

Note that when either `blocks` or `threads` is to specified as a 2D or 3D array, the special CUDA datatype `dim3` must be used, and the value is set like this:

```
dim3 blocks ( 5, 2 );
dim3 blocks ( 5, 2, 1 );
dim3 threads ( 2, 4 );
dim3 threads ( 8, 4, 2 );
```

The second thing to notice is that there are actually **two** functions that run on the GPU. The kernel is declared as `__global__ julia_set (...)` so that the CPU can "see" it. But a second function is declared as `__device__ julia_value(...)` because it runs on the GPU and the CPU doesn't call it directly.

The file *julia_cpu.c* is a version of the program that does not use the GPU at all. Compare the `julia_set` functions in the CPU and GPU versions of the program to get a better understanding of how a kernel function works.

Submit the batch file *julia_gpu.sh* to the HokieSpeed queue. When the results are returned, you should have a file called *julia_gpu.tga*. If you bring that file back to your laptop, you can view it with the command:

```
eog julia_gpu.tga
```

10 Discussion: When Blocks * Threads and N Differ

So far, in our examples, the product of `blocks` and `threads` was exactly equal to the number of tasks. Can we only work in this way? The GPU can do all the tasks, but each thread will do several tasks, and monitor when they should stop. We simply have to have each kernel execute an appropriate `while` loop til all the tasks are assigned. Assuming that both `blocks` and `threads` are scalars, a kernel that simply prints the id `T` of each task between 0 and `N-1` would be:

```
i = threadIdx.x;
j = blockIdx.x
k = threadIdx.x + blockDim.x * blockIdx.x;
printf ( "K=%d: thread %d, block %d did task", k, i, j );
t = k;
while ( t < n )
{
  printf ( " %d", t );
  t = t + blockDim.x * gridDim.x;    <-- here, gridDim.x is the number of blocks
}
printf ( "\n" );
```

where we have made sure to pass into the kernel the value of `N`. Let's suppose we want to do 37 tasks, but for whatever reason, we want to use `blocks=2` and `threads=5`. The output from the kernel would be:

```
K=0: thread 0, block 0 did task  0 10 20 30
K=1: thread 1, block 0 did task  1 11 21 31
K=2: thread 2, block 0 did task  2 12 22 32
K=3: thread 3, block 0 did task  3 13 23 33
K=4: thread 4, block 0 did task  4 14 24 34
K=5: thread 0, block 1 did task  5 15 25 35
K=6: thread 1, block 1 did task  6 16 26 36
K=7: thread 2, block 1 did task  7 17 27
K=8: thread 3, block 1 did task  8 18 28
K=9: thread 4, block 1 did task  9 19 29
```

which shows that every task gets done, in a roughly even and regular way.

Notice that the exact same code will work correctly for **any value of N**, including 3 (small), 10 (just right), and 37 (too big). Moreover, it will work even if we change `blocks` and `threads`.

This means that we are free to choose the configuration of `blocks` and `threads` to take advantage of the capabilities of our GPU, and, perhaps, of the peculiarities of our task size `N`.

11 Exercise: cuda_tasks.c

The C program in the file *cuda_tasks.c* shows how CUDA will deal with `N` tasks, for a given configuration of blocks and threads. Now we must realize that CUDA allows both blocks and threads to be 3D arrays.

Compile the program, then run it for 2 blocks, 5 threads, and 37 tasks:

```
./cuda_tasks 2 1 1 5 1 1 37
```

In general, for a block configuation `bx,by,bz` and thread configuration `tx,ty,tz` and tasks `N`, we would write

```
./cuda_tasks bx by bz tx ty tz n
```

to see how CUDA would organize the operation.

What happens for these sets of blocks, threads and tasks? (Just input numbers, not commas or parentheses!)

- (1, 1, 1), (1, 1, 1), 23;
- (2, 3, 1), (2, 1, 4), 40;
- (1, 1, 1), (2, 2, 2), 23;

12 Discussion: Choosing Blocks or Threads based on Problem Size N

Most of our examples have been written knowing in advance the value of `N`. If we wish to generate exactly `N` threads, then it was easy to choose values of `blocks` and `threads` whose product was `N`. But to take advantage of thread or SIMD parallelism on the GPU's, it's good to use a small but reasonable value for `threads` such as 16 or 32. We want `blocks` to be large enough to guarantee that we generate at least `N` threads in total. The value `N/threads` will actually be slightly too small, unless `N` happens to be exactly divisible by `threads`. The way to set `blocks` is to use a formula like this:

$$\text{blocks} = \frac{N - 1 + \text{threads}}{\text{threads}} = 1 + \frac{N-1}{\text{threads}}$$

See that, if we have fixed `threads`=5, the formula chooses `blocks` to match any given value of `N`:

N	1	2	3	4	5	6	7	8	9	10	11	12	13	...
blocks	1	1	1	1	1	2	2	2	2	2	3	3	3	...
total threads	5	5	5	5	5	10	10	10	10	10	15	15	15	...

13 Discussion: Multidimensional Blocks and Threads

Sometimes it can be convenient to have multidimensional block and thread arrays.

- the array (which can be 1D, 2D, or 3D) must be declared using the CUDA datatype `dim3`, as in

```
dim3 blocks ( 3, 5 );     <-- Notice these are PARENTHESES, not CURLY BRACKETS!
dim3 blocks ( 3, 5, 7 );
dim3 threads ( 2, 4 );
dim3 threads ( 2, 4, 8 );
```

- the block indices are stored in `blockIdx.x, blockIdx.y, blockIdx.z` and the values of `blocks` are in `gridDim.x, gridDim.y, gridDim.z`;
- the thread indices are stored in `threadIdx.x, threadIdx.y, threadIdx.z` and the values of `threads` are in `blockDim.x, blockDim.y, blockDim.z`;
- the kernel index `K` has the formula:

$$\begin{aligned} k = &\text{threadIdx.x} \\ &+\text{blockDim.x} * \text{threadIdx.y} \\ &+\text{blockDim.x} * \text{blockDim.y} * \text{threadIdx.z}] \\ &+\text{blockDim.x} * \text{blockDim.y} * \text{blockDim.z} * \text{blockIdx.x} \\ &+\text{blockDim.x} * \text{blockDim.y} * \text{blockDim.z} * \text{gridDim.x} * \text{blockIdx.y} \\ &+\text{blockDim.x} * \text{blockDim.y} * \text{blockDim.z} * \text{gridDim.x} * \text{gridDim.y} * \text{blockIdx.z} \end{aligned}$$

If a dimension has extent 1, then the only id value is zero, and so that term drops out of the formula. This is why, when `blocks` = 2 and `threads`=5, only the first and fourth terms survive, and the formula simplifies to:

$$k = \text{threadIdx.x} + \text{blockDim.x} * \text{blockIdx.x} = \text{threadIdx.x} + 5 * \text{blockIdx.x}$$