GPU Programming Basics: Getting Started

Louis JM Aslett

Department of Statistics, Trinity College, University of Dublin

7 February 2011

Trinity College Dublin

Goals of Talk

Extend the talk of May 2010 to include:

- High-level GPU overview
- Introduction to nVidia's Compute Unified Device Architecture (CUDA)
- Writing GPU kernels in C
- Compiling and using GPU code with R
- Using the Amazon Cloud

Trinity College Dublin

But first ...

It's possible to use the GPU without learning anything new, and the options are getting better all the time:

- gputools – an R package which does dense matrix multiplication, linear model fits etc using CUDA enabled GPUs
- cuBLAS – a standard C library written by nVidia to do basic linear algebra on the GPU (upto 10× faster than MKL)
- cuSPARSE – a standard C library written by nVidia to do sparse matrix linear algebra on the GPU (upto 32× faster than MKL)
- cuRAND – a standard C library written by nVidia to do large scale random number generation (upto 16 billion random values generated per sec)

Trinity College Dublin

But first ...

Since Amazon Cloud also a focus of the talk, special mention for **segue**.

Segue (http://code.google.com/p/segue/, not on CRAN yet) is only a month or so old as a project, but enables parallel processing on Amazon nodes in a few lines of R code, eg:

```
> library(segue)
> setCredentials('ACCESS_KEY_ID', 'SECRET_ACCESS_KEY')
> myCluster <- createCluster(numInstances=10)
> res <- emrlapply(myCluster, myList, myFunc, na.rm=T)
```

Very cool – highly recommend checking out!

Trinity College Dublin

As usual ...

The mantra recommended last May can be extended:

Statistical Programming Mantra

R when you can

- Exhaust the 5 tips (data frame write/read issues, pre-allocate memory, vectorize, BLAS)
- Identify bottlenecks (system.time, Rprof)

C when you must

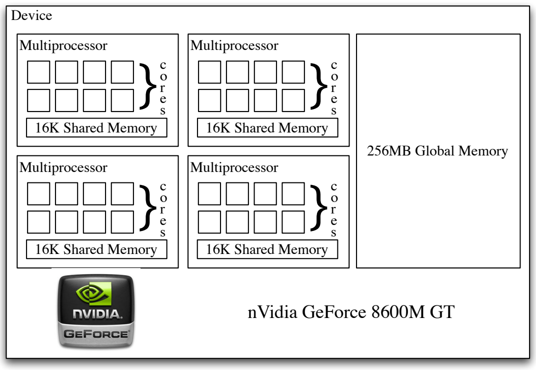
- Drop into C selectively (.C, .Call)

Low-level parallelize when you must (and can)!

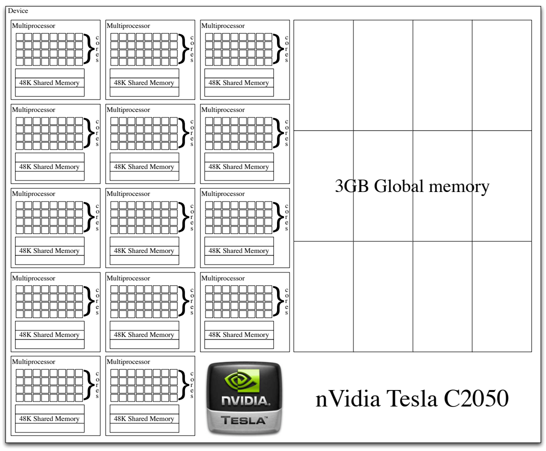
- Brute force massive parallelism (CUDA, ATI Stream, OpenCL, PThreads, OpenMP, MPI, Map Reduce, etc)

Trinity College Dublin

Highly Simplified GPU Architecture



Trinity College Dublin



CUDA Concepts and Terminology

- ***Kernel:*** a C function which is flagged to be run on a CUDA capable device
- A kernel is executed on the core of a multiprocessor inside a ***thread***. A thread can be thought of as just an index $j \in \mathbb{N}$. V Loosely: a index of cores in multiprocessors
- At any given time, a ***block*** of threads is executed on a multiprocessor. A block can be thought of as just an index $i \in \mathbb{N}$. V Loosely: an index of multiprocessors in devices
- Together, (i, j) corresponds to exactly one kernel running on a core of a single multiprocessor.

i.e. Very simplistically speaking, think of how to parallelize your problem by how to split it into identical chunks indexed by a pair $(i, j) \in \mathbb{N} \times \mathbb{N}$

Trinity College Dublin

What the heck does that mean!?

Imagine you want to parallelize a for loop:

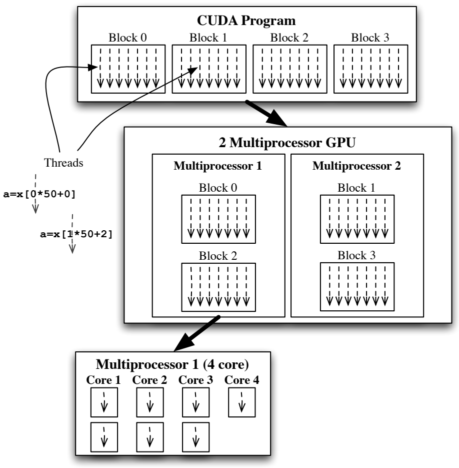
```
for(int i=0; i<1000; i++) { a=x[i]; }
```

In block/thread speak you could have a single block/1000 thread ($i = 0, j = 0 \dots 999$) kernel containing (pseudo-code):

```
a=x[thread_index];
```

The exact same kernel is called blocks × threads times with the block and thread indices changing.
Or, to make use of more than one multiprocessor, say $i = 0 \dots 19, j = 0 \dots 49$ and kernel:

```
a=x[block_index*50+thread_index];
```



Rules

- There is a cap on the maximum number of blocks and threads (though both can – and should – exceed the physical number of multiprocessors and cores)
- Can't assume threads will complete in the order you index them.
- Can't assume blocks will complete in the order you index them.
- To deal with execution order dependency either:
  - run dependent items in the same block (`__syncthreads()`, beyond talk scope)
  - split into kernels which you call consecutively from C
- Don't write to the same memory location from different threads (proviso: shared memory, beyond talk scope)

First example: Add two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^{1000}$

Kernel Code

```
__global__ void VecAdd(float* a, float* b, float* c) {
   int j = threadIdx.x;
   c[j] = a[j] + b[j];
}
```

Called from your standard C code by:

```
      VecAdd<<<1, 1000>>>(a, b, c);
```

Number of blocks Number of threads

So this executes on a single multiprocessor with 1000 threads automatically scheduled over the 8 (laptop) or 32 (Tesla) cores.

Clearly a massive waste, so easily adapt to run over many multiprocessors:

Kernel Code

```
__global__ void VecAdd(float* a, float* b, float* c) {
   int k = blockIdx.x * 50 + threadIdx.x;
   c[k] = a[k] + b[k];
}
```

Called from your standard C code by:

```
VecAdd<<<20,50>>>(a, b, c);
```

So this queues 20 blocks over all multiprocessors with each block having 50 threads automatically scheduled over the cores.

But, the GPU cannot access system memory, so there is some setup to do before the `VecAdd` call. The full code to call from R:

Host Code

```
void VecAddR(float* a, float* b, float* c) {
   float *a_GPU, *b_GPU, *c_GPU;

   cudaMalloc(&a_GPU, 1000*sizeof(float));
   cudaMalloc(&b_GPU, 1000*sizeof(float));
   cudaMalloc(&c_GPU, 1000*sizeof(float));

   cudaMemcpy(a_GPU, a, 1000*sizeof(float),
                                 cudaMemcpyHostToDevice);
   cudaMemcpy(b_GPU, b, 1000*sizeof(float),
                                 cudaMemcpyHostToDevice);

   VecAdd<<<20,50>>>(a_GPU, b_GPU, c_GPU);
```

Host Code (ctd.)

```
   cudaThreadSynchronize();

   cudaMemcpy(c, c_GPU, 1000*sizeof(float),
                                 cudaMemcpyDeviceToHost);

   cudaFree(a_GPU);
   cudaFree(b_GPU);
   cudaFree(c_GPU);
}
```

Note that if you are compiling kernel and host code in one file, the host code must be wrapped in `extern "C" { ... }` and the file should have a `.cu` extension rather than `.c`

See handout for full details and how to compile for use in R.

LOCAL DEMO & AMAZON DEMO

Performance Tips

- Can get surprisingly good results even ignoring performance considerations and just making sure:
  # blocks > # multiprocessors
  and
  # threads > # cores per multiprocessor.
- This makes GPU programming very attractive ... if you can translate problem to independent double for loop, then you can almost blindly GPU-ize and likely win.
- However, even more massive gains on non-trivial problems with some understanding of performance tuning. Three simple to understand things can make a massive difference ...

Simple Performance Tip #1

Recall:

- Number of blocks can exceed number of multiprocessors
- Number of threads can exceed number of cores per multiprocessor

Worst case, at least both should equal the physical device sizes or else cores sit idle.

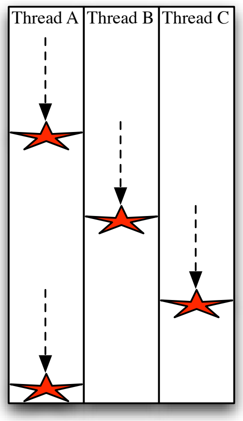
But in reality, **ensure the thread figure exceeds the number of cores per multiprocessor** for performance reasons.

nVidia provide an ‘occupancy calculator’ in the form of an Excel spreadsheet which allows you to tune how many threads to choose for any given problem.

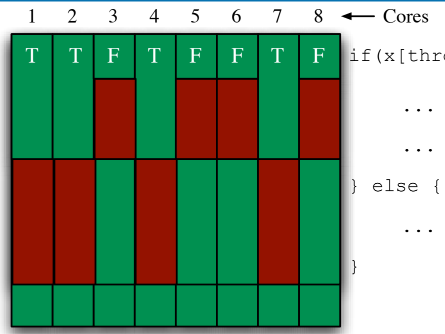
★ = Global memory access
=> Execution stall!

Global memory accesses are slow, so a core will stall when a request is made.

But, if # threads > # cores then another thread will be interleaved and run until the memory request is fulfilled and the first thread can run again.



Simple Performance Tip #2



```
if(x[thrd_id]>0) {
    ...
    ...
} else {
    ...
}
```

Threads execute in lock-step on the cores of a multiprocessor, so beware of very divergent code ... best to use block indices to separate highly divergent paths.

Simple Performance Tip #3

The multiprocessors are able to pull in ranges of memory in large blocks rather than element by element as each core requires it (note also, due to the lock-step all cores will be ready for memory access at the same time).

Hence, the multiprocessor will try to ‘coalesce’ the memory requests. **It can only do this if each thread accesses consecutive bytes in memory which are aligned with the memory stride.**

Simplistically speaking: `a[blk*n+thr]` is fast; `a[thr*n+blk]` is slow!

Full answer: it is worth looking up `cudaMallocPitch` and `cudaMemcpy2D`.

Happy coding!