CUDA: first impressions

Maxim Potekhin

Wirecell meeting at BNL May 22nd 2015

Overview

- Information (inlcuding online classes, tutorials etc) on CUDA is plentiful on the Web - so the purpose of this slides is not to present something deep, but to hopefully save time for everybody by providing a compressed quick overview tailored to the group. If everyone knows all of this already, all the better. I'll be quick.
- I'm not an expert - opinions expressed here are my own, based on a bit of tinkering with the Nvidia CUDA IDE and running simple tests.
- May help form an initial opinion on usefulness of CUDA for the Wirecell project.
- Also to facilitate comparison with OpenACC and other accelerator tools (to be done)
- I'll show a trivial piece of code I wrote for illustration purposes and ran on my graphics card, and then briefly explain how it relates to the API which in turn relates to GPU architecture.

What is CUDA?

- Wikipedia: CUDA is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. CUDA gives developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.
- You trade the speed of the processor for the sheer number of processors available to your application ($\sim 10^3$). Individual cores in the GPU are slow compared to a CPU in late model PC. According to Wikipedia: "GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly."
- CUDA SDK was released in 2007 - "somewhat" recently. It was much smaller and modest in terms of included functionality (e.g. libraries) compared to contemporary releases.
- Applications (including in physics) followed rather quickly (conference presentations in 2009)!
- Since 2008 supported on Windows, Linux and Mac.
- Supports a vast number of GPU models produced by NVIDIA.
- Various types of NVIDIA hardware are not created equal. There are "levels" of CUDA support in each generation, meaning either more or less functionality depending on the hardware version.

Use in HEP

- Despite quick initial adoption, it took time for this technology to be utilized at scale.
- Hardware and software underwent rather rapid evolution, resulting in multiple versions of both with different functionalities (while retaining a degree of compatibility). Not every CUDA version works with hardware version XYZ.
- Many large and/or important projects were conceived and/or built before CUDA reached a degree of maturity and acceptance in the HEP community.
- Structuring computational problems to fully exploit parallelism is a challenge above and beyond just writing good software. A few important software components (cf. GEANT4) are older than CUDA and other parallel platforms so provisions for parallelism were not an integral part of their design.
- GPU adoption simply didn't reach the critical mass that would result in such facilities to be built at scale. At the same time, there are supercomputing facilities created for other reasons that can be exploited opportunistically. A few of the current and future systems are equipped with CUDA-capable GPUs.
- Finally, a few notable experiment like IceCube and LHCb are in fact utilizing GPUs as an integral part of their systems (in differing capacities).
- It looks increasingly likely that the trend will continue at a faster pace and we don't want to be left behind.

"Device" vs "Host"

- In a typical installation, the GPU (often called "the device") resides inside a more conventional computing hardware, such as a PC, called "the host".
- The host itself can be a node inside a supercomputer.
- Multiple GPUs in a single host are supported:
  - in the PC scenario, this is useful to know since heavy GPU use can interfere with primary function of the graphics card which is being a video adaptor, although in reality it's the other way around - the driver can kick out computational load from GPU if it considers it excessive - all tunable, of course.
  - provides a straightforward way to scale even in a conventional box
- GPUs don't have an operating system in the conventional sense, so basic functions such as memory allocation (also freeing memory and others) are hidden behind an API that can be used from code running on the host.
- Memory is not shared between the host and the device. There is an API which allows copying of the data in either direction. Communication happens over the bus (e.g. PCIe) and therefore is a lot slower than RAM access. It is one of the principal limiting factors for overall performance.
- Functions that are executed on the GPU are termed "kernel".

Elements of CUDA syntax

Declaring functions

- __global__ declares kernel, which is called on host and executed on device
- __device__ declares device function, which is called and executed on device
- __host__ declares host function, which is called and executed on host
- __host__ and __device__ can be combined if the same C (C++) function will need to run on both (obviously compiled differently for each target)

Declaring variables

- __device__ declares device variable in global memory, accessible from all threads, with lifetime of application
- __constant__ declares device variable in constant memory, accessible from all threads, with lifetime of application
- __shared__ declares device varibale in block's shared memory, accessible from all threads within a block, with lifetime of block

- Most routines return an error code of type cudaError_t.

Testbed

- I used the graphics card in my home PC (which is high-end but this has no bearing on the test).
- GTX 970 Engine Specs:

CUDA Cores:	1664 (128 cores per 13 Multiprocessors)
Base Clock (MHz):	1050
Streaming Multiprocessors (SMM):	13 (Maxwell class)

- GTX 970 Memory Specs:

Memory Clock:	7.0 Gbps
Standard Memory Config:	4GB (subject to debate due to varying speed)
Memory Interface:	GDDR5
Memory Interface Width:	256-bit
Memory Bandwidth (GB/sec):	224 (subject to debate)

- Misc:
  - The card was released last year
  - 148W nominal power compared to 230-250 in previous generation (770, 780)
  - The 970 is a pretty much a crippled 980 (which has 16 SMM and 2048 cores)

CUDA software

- In CUDA, host and device code coexist in a single source file (*.cu) - for the developer, it looks like normal C++ with a few additional keywords and decorations (for those who use those in Python).
- On any platform, "nvcc" compiler is used to compile the code, and it in turn calls the native host compiler (e.g. gcc on Linux or Visual C++ compiler on Windows) to create runtime binaries.
- You will notice longer compilation times compared to plain vanilla case.
- Quote: "The CUDA development environment relies on tight integration with the host development environment"... (cf. C libraries etc).
- Using an IDE: the only straightforward way to use CUDA on Windows is via a kit that works with a specific version of Visual Studio. On Linux there is an option to integrate with Eclipse, but integration with an IDE is not mandatory – you can compile and run from the command line.

A trivial test of CUDA (1)

- There is a plethora of CUDA tutorials, examples and tests, lots of them covering problems in multiple dimensions. In addition to a few test I ran and tweaked, I decided to write something very simple from scratch to gain initial experience before moving to N-dimensional code. One dimension seemed easy enough, and I chose to implement a dot product routine - if the vectors being multiplied are very long, it is beneficial to split the procedure in multiple threads each iterating over a slice of the vectors.
- Used random numbers but switched to same value for vector components for ease of debugging (fill them with same number).
- In no way should it be considered a GPU performance test or a benchmark as it is not designed for this (as we will see). Scaling with thread count will be measured, though.
- Kernel is a function that will run on the GPU, in this test there are two:

```
__global__ void threadedDotProduct(int load, int *inp1, int *inp2, int *interim) {
    int index = blockIdx.x*blockDim.x + threadIdx.x; // this is the global thread index
    for (int i = 0; i < load; i++) interim[index] += inp1[index + i] * inp2[index + i];
}
__global__ void summation(int *array, int N) {
    int sum = 0;
    for (int i = 0; i < N; i++) sum += array[i];
    array[0] = sum;}
```

A trivial test of CUDA (2)

- Running the kernel - looking in the host code. "Load" is simply an integer specifying the length of the portion of the vector to be processed in one thread, "device arrays" are vectors being multiplied, "interim array" contains partial sums to be combined later.
  - first compute partial dot-product of two vectors in b*t threads... watch for the triple chevron!

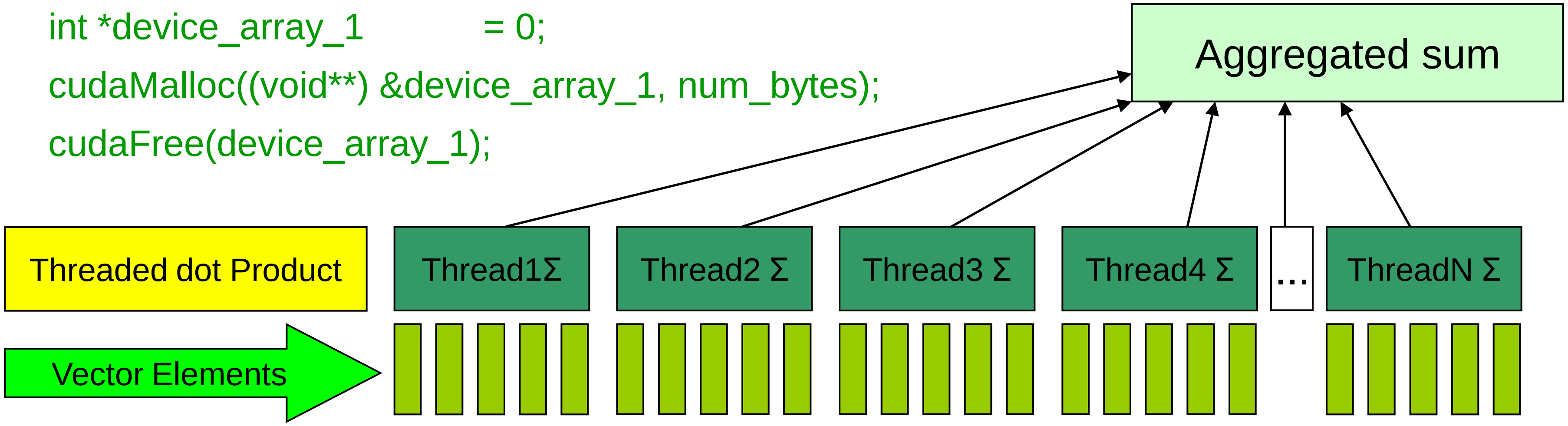
```
threadedDotProduct<<<b,t>>>(load, device_array_1, device_array_2, interim_array);
```

  - then aggregate partial sums into the final scalar in a single thread

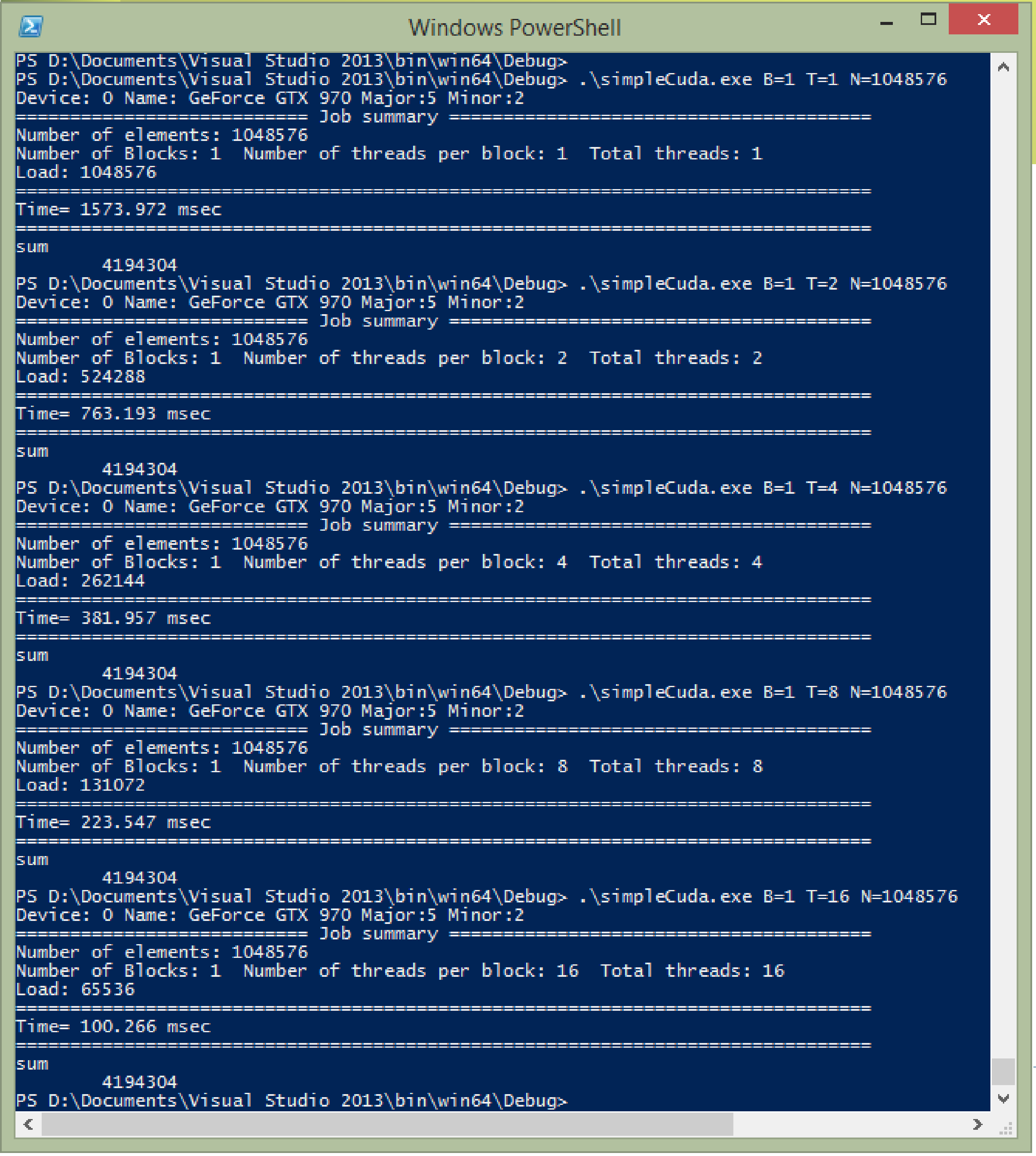
```
summation<<<1,1>>> (interim_array, num_inter);
```

- Memory allocation/deallocation on the device:

```
int *device_array_1          = 0;
cudaMalloc((void**) &device_array_1, num_bytes);
cudaFree(device_array_1);
```



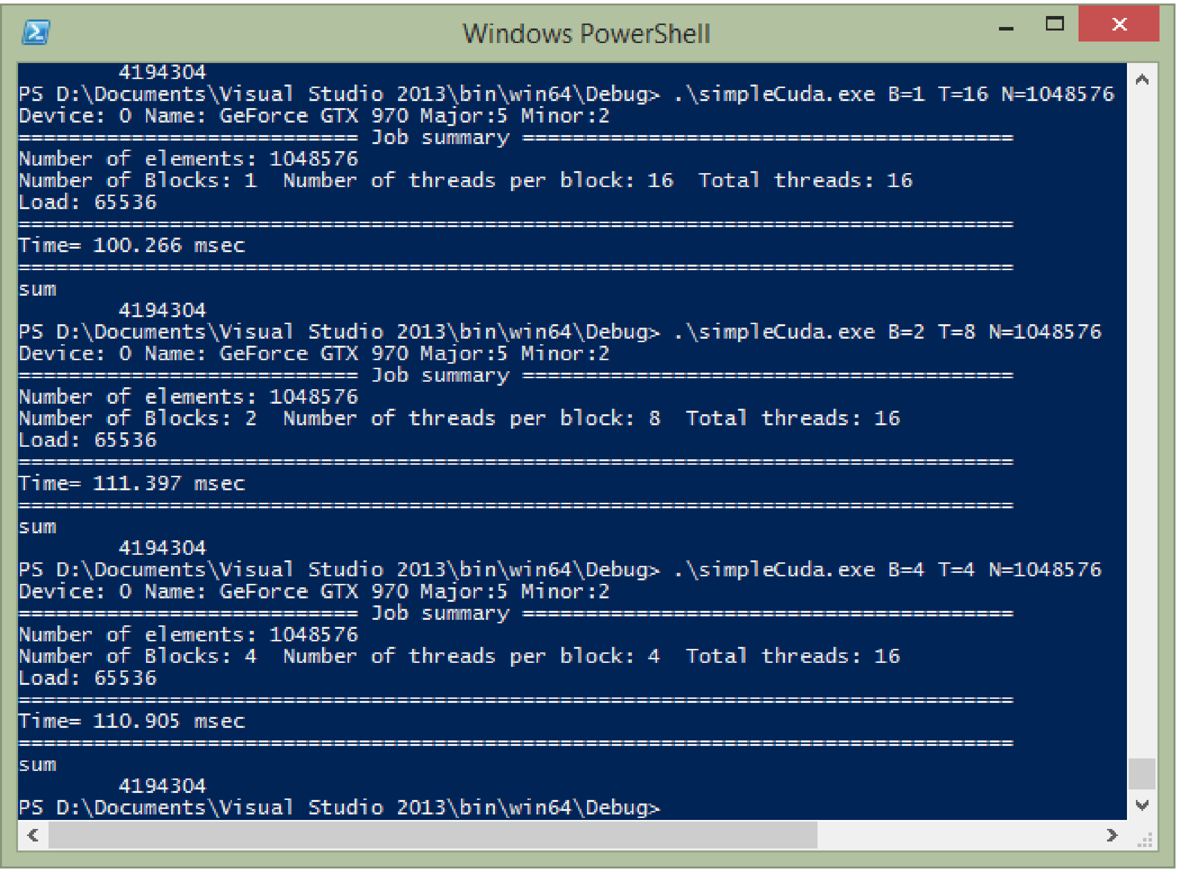
(Unscientific) Scale test 1



```
PS D:\Documents\Visual Studio 2013\bin\win64\Debug>
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=1 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary ======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 1  Total threads: 1
Load: 1048576
==============================================================================
Time= 1573.972 msec
==============================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=2 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary ======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 2  Total threads: 2
Load: 524288
==============================================================================
Time= 763.193 msec
==============================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=4 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary ======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 4  Total threads: 4
Load: 262144
==============================================================================
Time= 381.957 msec
==============================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=8 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary ======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 8  Total threads: 8
Load: 131072
==============================================================================
Time= 223.547 msec
==============================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=16 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary ======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 16  Total threads: 16
Load: 65536
==============================================================================
Time= 100.266 msec
==============================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug>
```

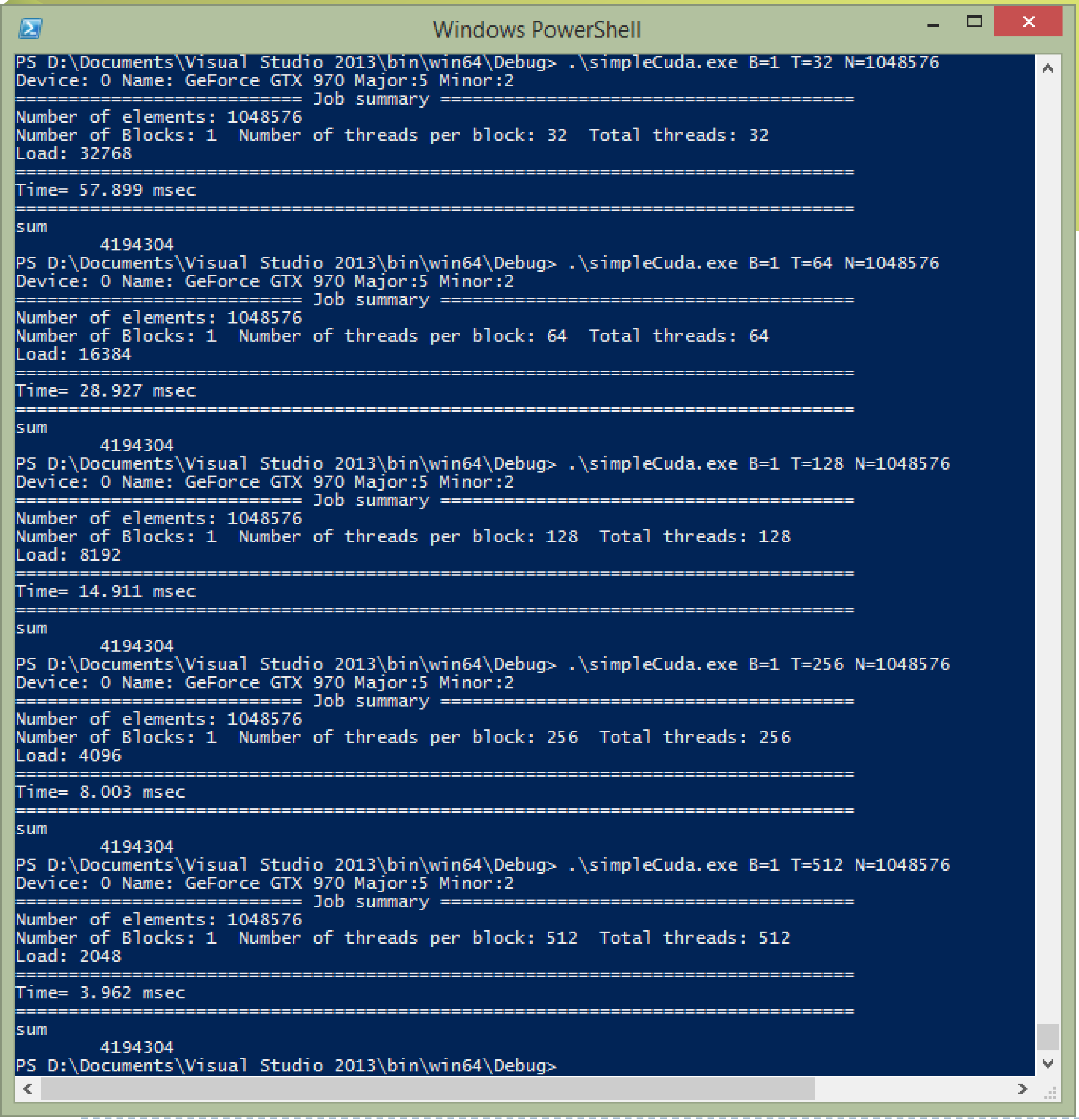
- Test case - a vector of 1M (1024^2) integers
- Keep 1 block of threads and vary their number
- Use built-in event timer in CUDA to measure execution time
- Under these conditions, we observe an almost linear with respect to the number of threads speed-up due to parallel execution.

Scale test 2



```
                4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=16 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary ========================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 16  Total threads: 16
Load: 65536
================================================================================
Time= 100.266 msec
================================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=2 T=8 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary ========================================
Number of elements: 1048576
Number of Blocks: 2  Number of threads per block: 8  Total threads: 16
Load: 65536
================================================================================
Time= 111.397 msec
================================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=4 T=4 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary ========================================
Number of elements: 1048576
Number of Blocks: 4  Number of threads per block: 4  Total threads: 16
Load: 65536
================================================================================
Time= 110.905 msec
================================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug>
```

- Keep total number of threads constant but vary the number of thread blocks.
- We observe nearly constant execution time. Something is being obviously done in parallel!

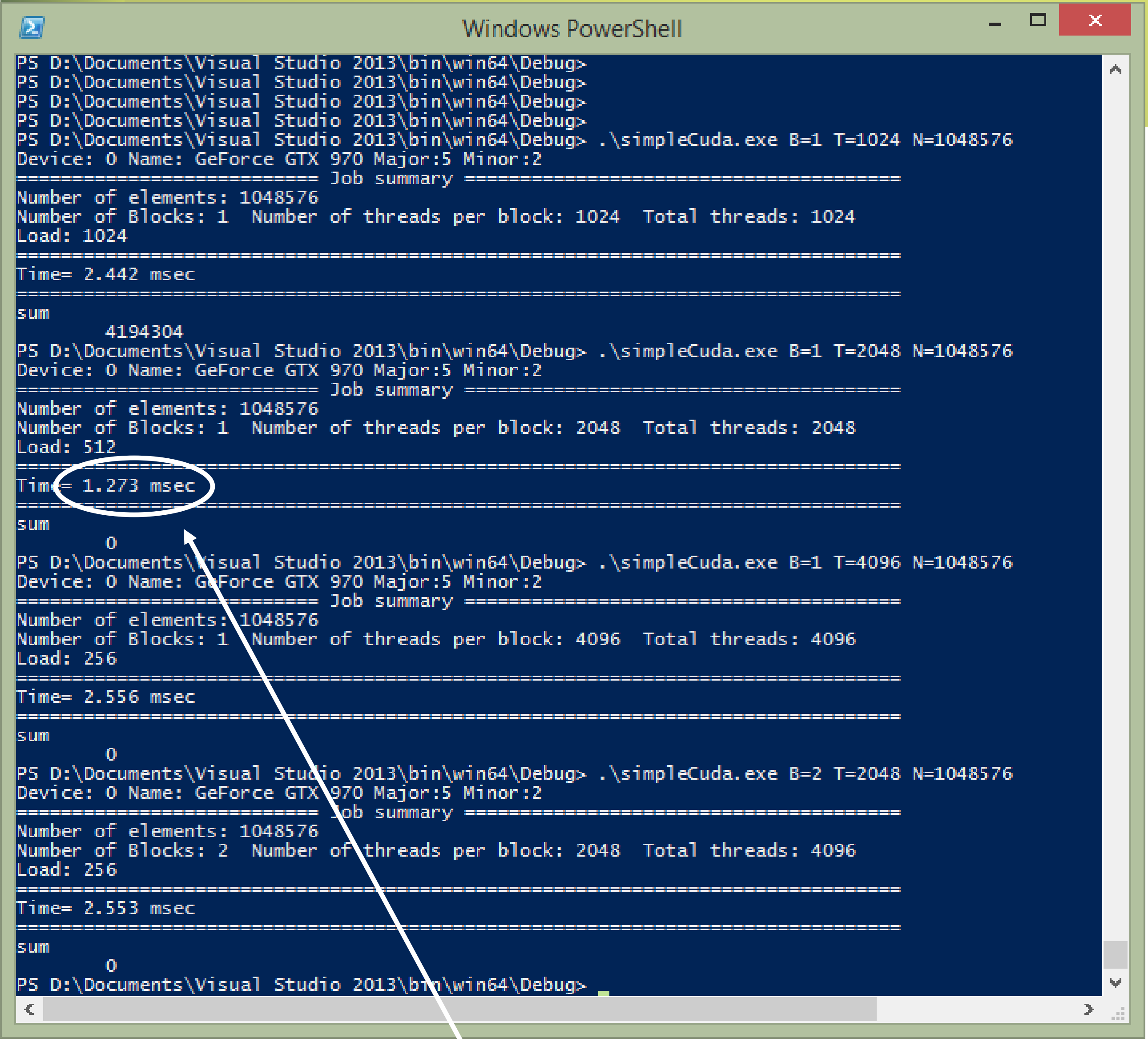


```
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=32 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary =======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 32  Total threads: 32
Load: 32768
===============================================================================
Time= 57.899 msec
===============================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=64 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary =======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 64  Total threads: 64
Load: 16384
===============================================================================
Time= 28.927 msec
===============================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=128 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary =======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 128  Total threads: 128
Load: 8192
===============================================================================
Time= 14.911 msec
===============================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=256 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary =======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 256  Total threads: 256
Load: 4096
===============================================================================
Time= 8.003 msec
===============================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=512 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary =======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 512  Total threads: 512
Load: 2048
===============================================================================
Time= 3.962 msec
===============================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug>
```

Scale test 3

- Now contunue to raise the number of threads from 16 to 512, linear (or almost linear) increase of speed continues.

Scale test 4



```
PS D:\Documents\Visual Studio 2013\bin\win64\Debug>
PS D:\Documents\Visual Studio 2013\bin\win64\Debug>
PS D:\Documents\Visual Studio 2013\bin\win64\Debug>
PS D:\Documents\Visual Studio 2013\bin\win64\Debug>
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=1024 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary =======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 1024  Total threads: 1024
Load: 1024
================================================================================
Time= 2.442 msec
================================================================================
sum
        4194304
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=2048 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary =======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 2048  Total threads: 2048
Load: 512
================================================================================
Time= 1.273 msec
================================================================================
sum
        0
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=1 T=4096 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary =======================================
Number of elements: 1048576
Number of Blocks: 1  Number of threads per block: 4096  Total threads: 4096
Load: 256
================================================================================
Time= 2.556 msec
================================================================================
sum
        0
PS D:\Documents\Visual Studio 2013\bin\win64\Debug> .\simpleCuda.exe B=2 T=2048 N=1048576
Device: 0 Name: GeForce GTX 970 Major:5 Minor:2
=========================== Job summary =======================================
Number of elements: 1048576
Number of Blocks: 2  Number of threads per block: 2048  Total threads: 4096
Load: 256
================================================================================
Time= 2.553 msec
================================================================================
sum
        0
PS D:\Documents\Visual Studio 2013\bin\win64\Debug>
```

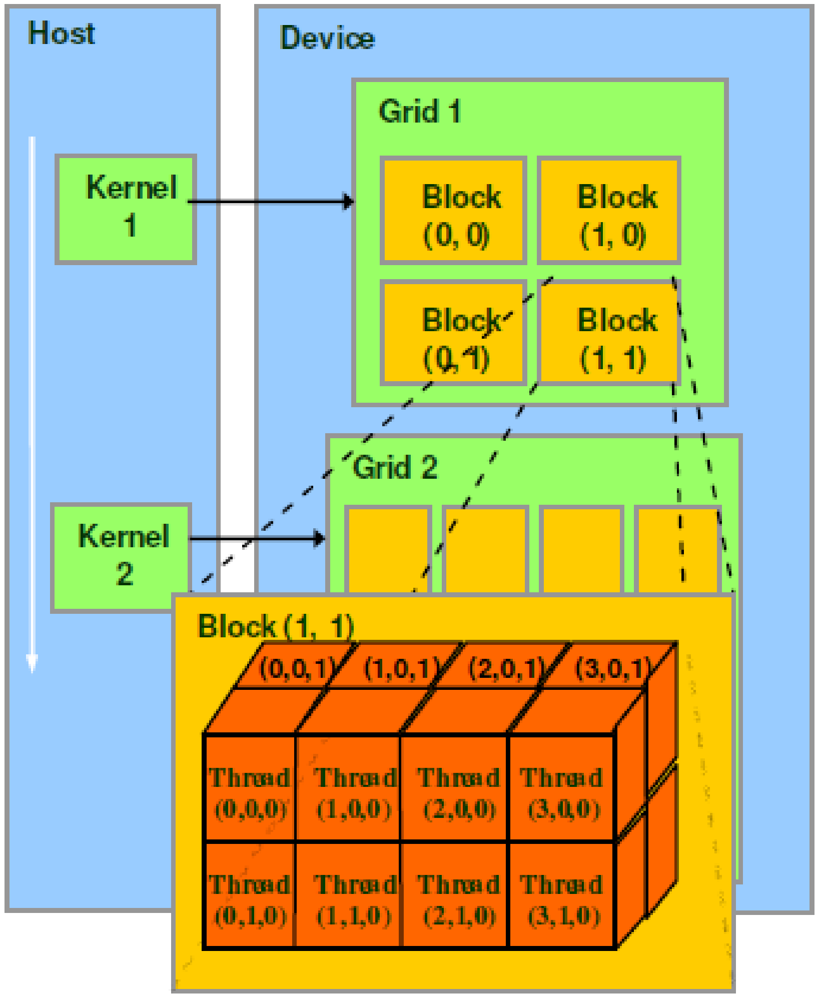
- Finally, with 2 to 4 thousand threads, we see saturation and then deterioration of performance.
- This is to be expected, since the thread load becomes very short while overhead grows (we also do final summation in a single thread).
- Total number of threads seem to matter the most, tried same with varying number of blocks.

PS. Same operation takes 9ms on a typical RACF interactive node

More on blocks and threads

- Why blocks? A block of threads is assigned to a single multiprocessing unit (SMM) within the GPU, so grouping threads like this allows better management of GPU workload. There will be queing of threads in the GPU.
- All threads in a block will run the same kernel.
- In the example considered so far, we used a scalar object (number) for blocks and same for threads: blah<<<b,t>>>(foo);
- This, however, is just a limiting case. Blocks can form a 2D grid, and each block can handle a 3D array of threads . This is achieved transparently (more or less) for the user by declaring and defining "b" and "t" accordingly.
- Blocks in the grid must have same size and cannot be resized at runtime.
- Why do people need more than one dimension in these data structures? That is because the challenge in this type of computing is optimal distribution of workload over threads, and it assumes segmentation of data which must be mapped to threads accordingly. Example: when solving a 3D problem, it will be often optimal to map portions of the 3D volume being processed to a 3D space of threads to exploit the data layout in a way that leads to computational parallelism.
- OTOH comment - some problems just beg to be solved on such grid, cf. Poisson equation.

An illustration



Cheat sheet

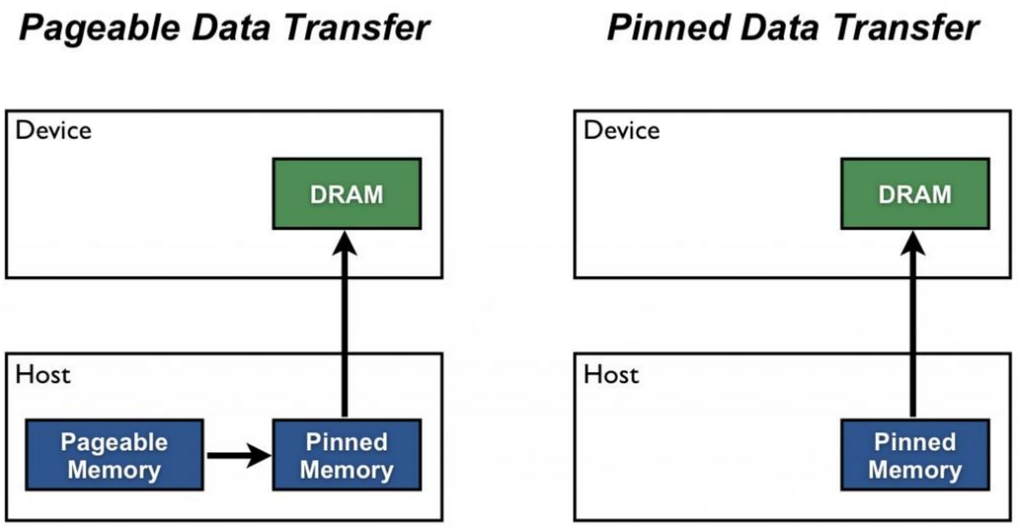
- Computing thread indices in multidimensional cases is prone to human error and can be confusing
- Thread indexing "cheat sheet" can be found (in particular) at http://www.martinpeniak.com/. Examples:

```
1D grid of 1D blocks
__device__
int getGlobalIdx_1D_1D() {return blockIdx.x *blockDim.x + threadIdx.x;}

1D grid of 2D blocks
__device__
int getGlobalIdx_1D_2D() {return blockIdx.x * blockDim.x * blockDim.y+ threadIdx.y * blockDim.x +
    threadIdx.x;}
....
2D grid of 3D blocks
__device__
int getGlobalIdx_2D_3D() {
int blockId = blockIdx.x + blockIdx.y * gridDim.x;
int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
+ (threadIdx.z * (blockDim.x * blockDim.y))
+ (threadIdx.y * blockDim.x) + threadIdx.x;
return threadId;
}
```

Host-device transfer considerations

- Host memory allocation - standard "malloc" will allocate memory which is pageable, which is not suitable for transfer to the device - it must be locked. This leads to an additional copy operation from pageable to "pinned" (locked) memory and a performance penalty. Up to a factor of ~2 difference.
- Actual speed will still depend on hardware.
- This can be avoided using specialized memory allocation function calls included in CUDA libraries: cudaMallocHost() etc.
- Use with caution - this will reduce available system memory.



Quick notes on memory access

- Since host-to-device communications are a critical bottleneck, it pays to deploy functions which don’t necessarily need parallelism on the device if they use device-resident data in some processing step, instead of copying it back and forth.
- "Global" memory resides in the GPU DRAM, and it can be accessed and modified from both the host and the device (e.g. during transfers)
- Global memory can be declared in global (variable) scope using the __device__ declaration specifier as in the first line of the following code snippet, or dynamically allocated using cudaMalloc().
- Arrays allocated in device memory are aligned to 256-byte memory segments by the CUDA driver.
- Misaligned access to memory will result in performance penalty (which can be severe sometimes).

Summary

- For many of us this is the first real encounter with GPUs. It does not mean that we won't consider Xeon and other available parallel computing platforms.
- With CUDA, the barrier to entry into massively parallel computing is extremely low since NVIDIA GPUs are very common and for the most part inexpensive. This is a big plus. Oftentimes even laptops are equipped with CUDA-capable GPUs. CUDA software is readily available and a large knowledge base exists in the developer community and documentation provided by NVIDIA.
- Apparently despite this platform being rather complex, the initial learning curve is not too steep (another big plus) and is conducive to experimentation and creativity in general.
- There is a selection of useful libraries provided with CUDA, which were not discussed in this presentation. This means that useful results can be achieved quickly and without low-level programming for some popular classes of problems (FFT, matrix manipulation etc).
- In addition to consumer or business-grade GPUs installed in workstations and laptops, there are supercomputing installations where CUDA technology is applied (cf. TITAN at Oak Ridge and some other Leadership Class Facilities). That indicates potential for running parallelized DUNE software on such platforms (with DOE's blessing).

Plans

- It may be a good idea to explore a few approaches to the current Wirecell framework from the standpoint of parallel architectures. Optimization problems can be solved numerically in different ways, and some of these methods may benefit more than others from deployment on GPUs. This needs to be understood.
- Should we look at fully 3D event model?