Investigations of applying GPGPUs to HEP

Amir Farbin
University of Texas at Arlington

Post-doc: Alden Stradling
CS Masters: Pavan Kumar, Jitendra Gupta
HEP Computing Problems

- Today: We do not have enough CPU resources for simulation...
- this will come out of the Tier 2 analysis resources.
- higher burden on Tier 3s...
- what if we need to re-simulate?

- Computing Model does not include resources for the CPU-intensive analysis activities which in the past decade have come to characterize analyses of HEP data at the Tevatron and the B factories:
  - sophisticated fits, statistical analysis of large “toy” Monte Carlo models, matrix element calculations, and use of the latest discriminant techniques, such as boosted decision trees.
- Even w/o upgrade we are resource limited...
- Resources requirements for upgrade might out-pace Moore’s law.
Moore’s Law

• Moore’s Law means more transistors. More cores per CPU. Requires parallelization.

• Our problems are embarrassingly parallel, so we parallelize our software by running multiple instances (or threads).

• Current parallelization efforts focus on memory sharing between threads due to concerns with memory costs.

• Even if memory was cheap, CPU to memory bandwidth limits scalability to large # of cores/CPU.

• Maybe need to worry about IO before this... can we feed many core CPU with enough data?

• Are there other ways of parallelizing our software workflow?

• Current: Sequentially processing (all steps), one event at a time.

• Future? Parallel processing many events, one step at a time.
What is GPGPU?

- General Purpose computing on Graphics Processing Units.
- Historically GPU development driven by gaming industry.
  - Specialized co-processors + video memory which simultaneously perform the same operation (generally linear algebra) on multiple chunks of data.
  - Single Instruction Multiple Data (SIMD) computing.
  - Originally GPUs had lots of simple processing units capable of simple instructions.
  - Evolved into execute nearly the same instructions as standard CPUs.
    - GPUs today have 100’s of cores, and are capable of simultaneously running O (10000) threads... achieving TFlops on a single GPU.
- CPUs are optimized for low latency... GPUs are optimized for high throughput.
- Note, not just faster than CPUs
  - Exponential increase in FLOPs
  - Lower power/cost
GPGPU Computing

- An individual core in a GPU is generally not as powerful or fast as a CPU.

- GPU is optimized for parallel computing:
  - It's all about data parallelism.
  - Memory access can be costly. Different types of memory provide different latency.
  - A processing unit with a thread waiting for data switches to another thread. (All in Hardware).
  - Better performance when threads perform same operation (in step).

- Computation Model
  - Data is moved (possibly asynchronously) between CPU and GPU memory.
  - Kernel is a computation initiated by CPU to run on GPU.
  - Data is broken into blocks, which is assigned to a physical unit... Multiple threads process each block, with low latency access to data in that block.
  - Allows transparent scaling to different GPUs.
GPGPU Computing

- An individual core in a GPU is generally not as powerful or fast as a CPU.
- GPU is optimized for parallel computing:
  - It’s all about data parallelism.
  - Memory access can be costly. Different types of memory provide different latency.
  - A processing unit with a thread waiting for data switches to another thread. (All in Hardware).
  - Better performance when threads perform same operation (in step).
- Computation Model
  - Data is moved (possibly asynchronously) between CPU and GPU memory.
  - Kernel is a computation initiated by CPU to run on GPU.
  - Data is broken into blocks, which is assigned to a physical unit... Multiple treads process each block, with low latency access to data in that block.
  - Allows transparent scaling to different GPUs.
NVidia’s Fermi

- Announced just last month.
- Note the name! (Previous gen was “Tesla”)
- Huge set of improvements...
- Supports C++

### Summary Table

<table>
<thead>
<tr>
<th>GPU</th>
<th>G80</th>
<th>GT200</th>
<th>Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>681 million</td>
<td>1.4 billion</td>
<td>3.0 billion</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Double Precision Floating Point Capability</td>
<td>None</td>
<td>30 FMA ops / clock</td>
<td>256 FMA ops /clock</td>
</tr>
<tr>
<td>Single Precision Floating Point Capability</td>
<td>128 MAD ops /clock</td>
<td>240 MAD ops /clock</td>
<td>512 FMA ops /clock</td>
</tr>
<tr>
<td>Special Function Units (SFUs) / SM</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Warp schedulers (per SM)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shared Memory (per SM)</td>
<td>16 KB</td>
<td>16 KB</td>
<td>Configurable 48 KB or 16 KB</td>
</tr>
<tr>
<td>L1 Cache (per SM)</td>
<td>None</td>
<td>None</td>
<td>Configurable 16 KB or 48 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>None</td>
<td>None</td>
<td>768 KB</td>
</tr>
<tr>
<td>ECC Memory Support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrent Kernels</td>
<td>No</td>
<td>No</td>
<td>Up to 16</td>
</tr>
<tr>
<td>Load/Store Address Width</td>
<td>32-bit</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
</tbody>
</table>

---

**Double Precision Application Performance**

- Double Precision Matrix Multiply
- Double Precision Tri-Diagonal Solver

**Radix Sort using Shared Memory**

- GT200 Architecture
- Fermi Architecture
Intel is also working the other way... Larrabee will be lots of simple x86 CPUs on one chip.
Hardware/Software Landscape

- **Hardware Vendors**: NVidia, ATI (AMD now), ClearSpeed, ...
- **Software Development Environments**: CUDA (NVidia only), BrookGPU, AMD Stream Computing (AMD only), Sh, OpenCL

- In general, NVidia has been pressing GPGPU the most.
  - $O(100)$ of scientific and commercial applications already developed in CUDA.
  - Lots of examples on NVidia website.
  - Many commonly used libraries already have CUDA implementations (eg LAPACK).
- Most promising development environment is OpenCL (Open Computing Language).
  - Initiated by Apple, passed to open consortium, endorsed by nearly everyone.
  - Architecture independent parallel computing for CPU and GPUs... use all resources!
  - Specs release late 2008. 1st implementation in Snow Leopard (Mac OS X).
  - AMD has shown OpenCL example using their ATI GPUs. NVidia recently released their own drivers.
  - Sufficient similarity to CUDA... develop in CUDA now... migrate to OpenCL later.
GPGPU Application Development

- Ideal GPGPU Applications exhibit following characteristics:
  - **Compute Intensity** – Large number of arithmetic operations per IO or global memory reference.
  - **Data Parallelism** – this exists in a dataset if the same function is applied to all records of an input stream and a number of records can be processed simultaneously without waiting for results from previous records.
  - **Data Locality** – a specific type of temporal locality common, where data is produced once, read once or twice later in the application, and never read again.
  - In general, code developed for CPU must be explicitly parallelized to run on GPU.
  - Must consider how to break data into chunks, with simple algorithm processing each chunk.
  - Model: data is prepared on CPU, shipped to GPU, processed, returned.
  - Development environments generally provide extensions to C which allow integration of CPU/GPU code.
  - Can’t just recompile your software to run on GPU... use GPU as a coprocessor.
GPU Programming Experience

- **OpenCL**: Compiles GPU code for current architecture at run-time while setting up the calculation. (Just in time compilation)
- Treats CPU/GPU on same footing...
- Clearly the way to go... but we experience strange bugs... Apple's JIT compiler seems not mature.
- **CUDA**: Single Architecture allows compiling at compile time.
- Faster execution than OpenCL.
- Code running on GPU must be very simple... (until Fermi?).
- **Data**: May only use simple types + arrays of simple types (and structs)... No recursion... Limitations on the complexity.
  - Must think of appropriate programming model.
- Data transfer between CPU/GPU memory can be time consuming.
- Performance highly dependent on how data is organized... high price for poorly accessing memory. (Global, shared, local memory).
- Optimization is critical...
Developing a GPGPU in HEP Strategy

• Though potential gain is large, path is not clear.

• HEP software doesn’t just apply the same simple calculation over and over again.

• Data/algorithms very complicated, developed by large community of non-expert user/developers, lots of legacy code ... and validation is critical.

• Long history behind the current solutions... often not optimal.

• GPGPU technology is still evolving.

• Suggestion:

• Go from simple/immediately useful (eg for data analysis) to complex/long-term applications (Geant4/SLHC).

• Start with simple problems which allow you to develop strategies for tackling foreseen challenges
Breaking it down

• 2 types of problems (optimization/implementaiton):

1. Optimization - optimal performance requires understanding of GPU and fine-tuning.
   a. Problems with essentially no data in/out
      - Monte Carlo Integration: RooFit, MadGraph (for matrix-element method analyses)
   b. Problems which read data once and out essentially no data
      - Discriminators: TMVA. Concentrating on Probability Density Estimators. Boosted Decision Trees and Neural Network training on big samples also good candidates.
      - ATLAS B Field: 40-200 MB (depending on whether they assume symmetry or not). Evaluation takes $O(1 \text{ ms}) \sim 1.2 \times 10^{-6} \text{ kSI2K}$. Evaluated trillions of times in simulation. No B-field Simulation goes $\sim 20\%$ faster. It is evaluated lots of times in reconstruction too.
c. Problems with continuous data in/out

- ROOT I/O (decompression): Kumar worked out how to do it...

- Collaborating with Box Leungsuksun (CS Faculty, LaTech). He has methods for hiding memory copy latency.

- Digitization. Relative time for mixing in pile-up: $10^{32}:1.0$, $10^{33}:2.3$, $3.5 \times 10^{33}$: $5.8$, $10^{34}:160$... but mostly an IO problem.

- Reconstruction: Tracking (CBM heavy ion experiment showed 60x faster GPU tracking fitting at CHEP), 200,000 Calo Cells.

- Relevant for HLT trigger, reco for SLHC.

- Read Out Drivers? These are easier to program than DSPs and FPGAs.

2. Implementation/Validation- HEP software is massive, developed by lots of people, and not built for parallelization. And you can’t just recompile your code... requires rewriting.

a. Translating Code- Must be able to automatically take (parts) of existing software and covert to GPGPU code/integrate. Currently only support C, so concentrate on MadGraph (Fortran) and specific parts of ROOT (eg RooFit PDFs). C++ when Fermi arrives.
b. Taking Advantage of GPU in Embarrassingly Parallel, but not Parallelized Application (basically all HEP applications).

- Problem is that max GPU efficiency achieved when perform MANY parallel operations.
- Our software processes data sequentially.
- Idea: Create thread-parallel versions of applications... run 1000s of threads... each thread requests computation from a GPU service and waits... GPU service performs computation in bulk.
- Great deal of effort to make Athena thread parallel + memory efficient... prerequisite for this approach.
- Good first application is B field in reco.
- Geant4 is the “holy grail”... but to get truly GPU version must rewrite (Geant5?)
  - Gene Cooperman et al (Northeastern) have machinery to turn Geant4 code into thread parallel (clever trick)...
  - Begin with B field again... then the B field stepper... then other computations.
UTA GPGPU Activity

- 2 CS Masters students... very slow progress.
- They have learned some C++ programming basics, ROOT, CUDA, OpenCL
- Read lots of code... eg gzip.
- Plan has been to build experience through simple problems which are relevant to our physics goals
  - Need an approach which is computationally intensive but underlying algorithm is simple.
  - Goal: full maximum likelihood fits using matrix-element $\otimes$ detector response.
    - Perhaps a way to avoid full Geant simulation!
- 2 simple relevant projects:
  - MC integration...
  - Probability Density Estimators...
ROOT I/O

- Compression save of order 20-50% in space.
- The limiting factor in ROOT I/O is not disk speed... it’s decompression (CPU limited).
- > 60% time reading Simple Data (ntuples) is spent in decompression.

- Majority of function calls are in decompression algorithms.
- We can parallelize gzip’s Huffman code compression, if we store the start of blocks (change format).
- Or pick another compression algorithm.
**Hard vs Solid State Drive**

- Sequential Read (Write): ~70 (70) MB/s vs 250 (180) MB/s...
- Random depends on data size and pattern.

<table>
<thead>
<tr>
<th>CPU Limited</th>
<th>Simultaneous Jobs</th>
<th>Compression Level</th>
<th>Rate (MB/s)</th>
<th>Rate (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk Limited</td>
<td></td>
<td></td>
<td>Hard Drive</td>
<td>Solid State Drive</td>
</tr>
<tr>
<td>Writing</td>
<td>1</td>
<td>0</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0</td>
<td>50</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Reading</td>
<td>1</td>
<td>0</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0</td>
<td>20</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>9</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>14</td>
<td>17</td>
</tr>
</tbody>
</table>

- Generally CPU limited. ~ 5x write speed reduction with compression.
- With 8x simultaneous access, only uncompressed write is clearly disk limited.
- SSDs help with reading in all cases (improved random access?)
- 2x speed improvement w/ no compression + SSD w/ 8 simultaneous jobs.
Matrix Element Method

• Instead of fancy theories which explicitly tackle big problems (eg SUSY) write down simple effective theory for new physics (eg using MadGraph).

• Provides a “signal” model... unlike SUSY which has huge parameter space.

• We can use this to generate MC samples for optimization/reporting of results.

• But we can now also apply matrix element methods, developed at TeVatron for top.

• Idea, calculate probability of an event being signal/background from convolution of matrix-element for sig/bkg process with detector response.

• The most sensitive method... knows about all correlations, uses all observables.

• 2 types of analyses:

• Use probabilities to create a discriminate.

• Construct a likelihood fit... TeVatron fits the top mass this way.

• Must integrate over unobserved quantities and convolute with detector resolution.

• Problem... it is slow. Including bkgs, this can take hours/event.
2 Relevant parameters in MC integration:
- Time per evaluation of integrand function
- Number of evaluations of the function
- We see that the CPU/GPU time plateaus for longer evaluation times.
- Current implementation suffers from memory access bandwidth limits.

Use 3 different kernels for: random number generation, function evaluation, summing (reduction) of results.

Each kernel must read results from global memory, works in shared memory, puts results back into global memory.

We are currently implementing a solution with one kernel, using mostly local and some shared memory.

Application: MadGraph, RooFit (PDF normalization).

Easy problem... good for understanding optimization, developing automatic GPU re-coding.
Probability Density Estimator

- When you can’t use Matrix Element (eg need parton showers, or prefer using real data)
- Determine the probability that an event is signal/background by comparing to large sample of signal/background events.
- Other kernel estimators (eg Keys) are approximations to this... but you are often limited in number of dimensions.
- CPU Method:
  - Calculate distance (metric) from test event to all events.
  - Sort them.
  - Count number of signal/background in some region.
- 2 GPU versions of this algorithm in literature... testing now.
- We have developed an algorithm for GPU which avoid sorting. CPU version of this algorithm is done... GPU version in the works.
Final Remarks

• Premature to look at CPU/GPU TFlops plots and say our computation problems are solved.

• HEP software is too complicated.

• 2 types of problems:
  • optimization of algorithms
  • implementation (within our current software)

• Requires real R & D...

• 3 Early areas where we may be able to make impact now, and learn for future:
  • Matrix Element Method: MC Integrator/NN-PDE (learn how to optimize and translate code)
  • ROOT decompression: Learn about data copy latency hiding.
  • B-Field: Learn how to integrate CPU/GPU computing.
Extra Slides
Matrix Element Method

• As proof of principle, consider \(-2 \log (L)\) for single SUSY events.

• \(L\) is computing using effective theory.

• SUSY events are forced

\[
pp \rightarrow s\bar{s} \rightarrow q\bar{q}X\bar{X}
\]

• Compare: If we measured everything in the event versus not measuring the \(X\) particle 3-vector. (Assuming knowledge of \(m_s\))

• In practice we fit in \(m_s / m_X\) plane... and there would be bkg... and detector resolution.

Thanks to Daniel Whiteson