

Investigations of applying GPGPUs to HEP

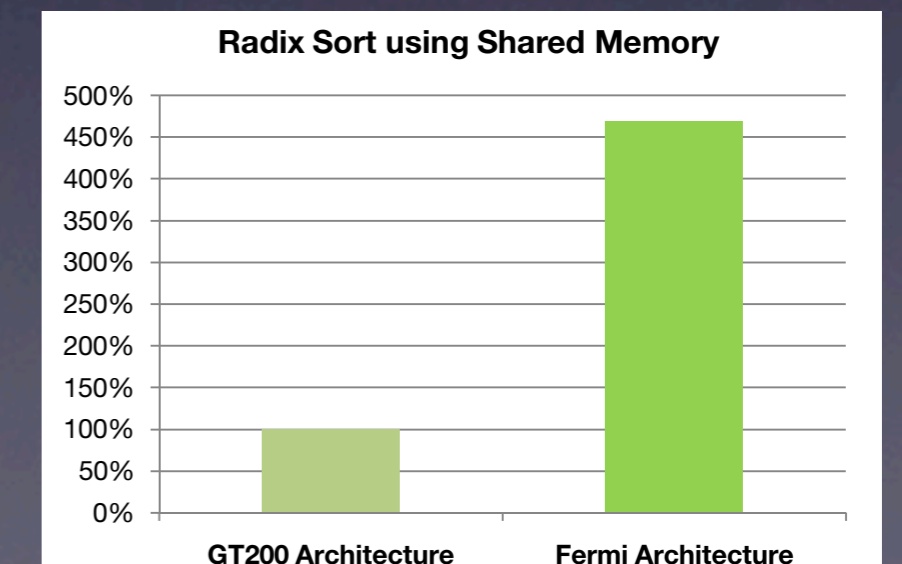
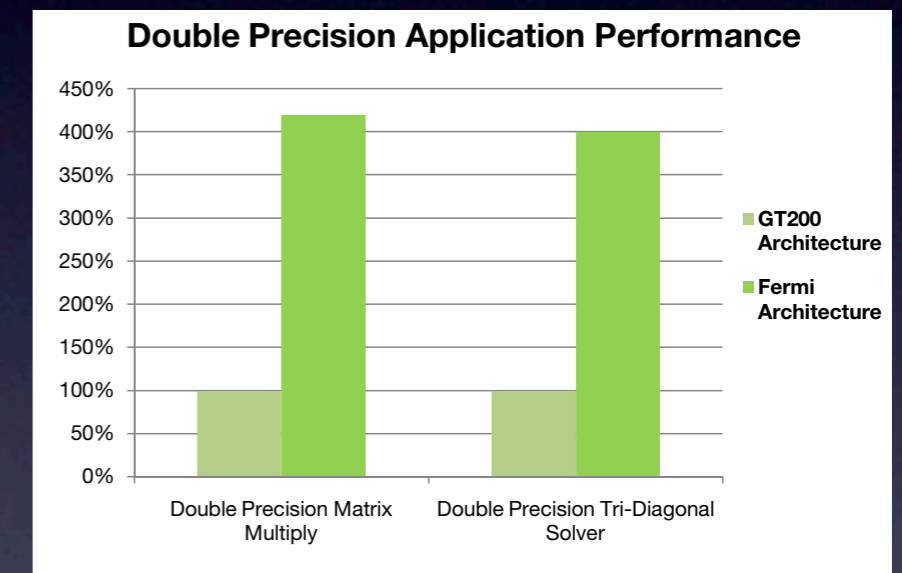
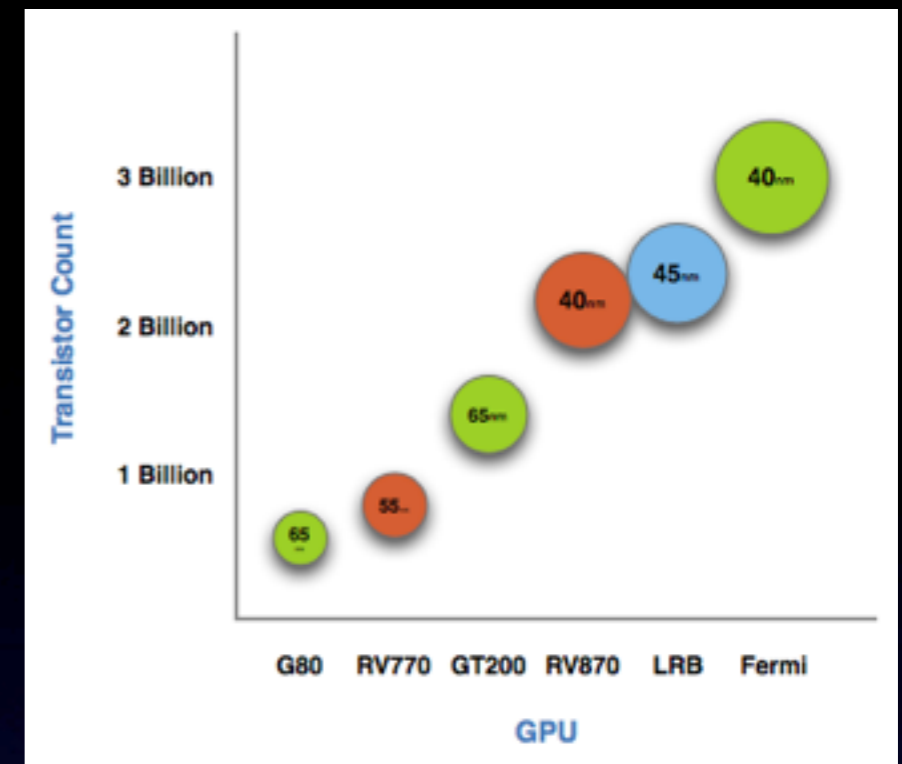
Amir Farbin
University of Texas at Arlington

Motivation

- ATLAS claims that disk is the primary constraint on simulation production
 - But there is a CPU constraint too (my own estimates)
 - Anyway, with infinite CPU, you don't need disk!
- With enough CPU, we may be able to short circuit the data → generator → simulation → digitization → reconstruction → data cycle.
- Massive parallelism will need to be adopted in core of future software.
- Various CPU bottlenecks in our software...
- Replace CPU with GPGPU above!

NVidia's Fermi

- Out now!
- Note the name! (Previous gen was "Tesla")
- Huge set of improvements...
- Supports C++



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

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.

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/implementation):
 - I. 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)
 - (Proof: “Calculation of HELAS amplitudes for QCD processes using graphics processing unit (GPU)” [K. Hagiwara](#), [J. Kanzaki](#), [N. Okamura](#), [D. Rainwater](#), [T. Stelzer](#)).
 - 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 $O(1 \text{ ms})$ ($\sim 1.2 \times 10^{-6}$ 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, 3.5×10^{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.

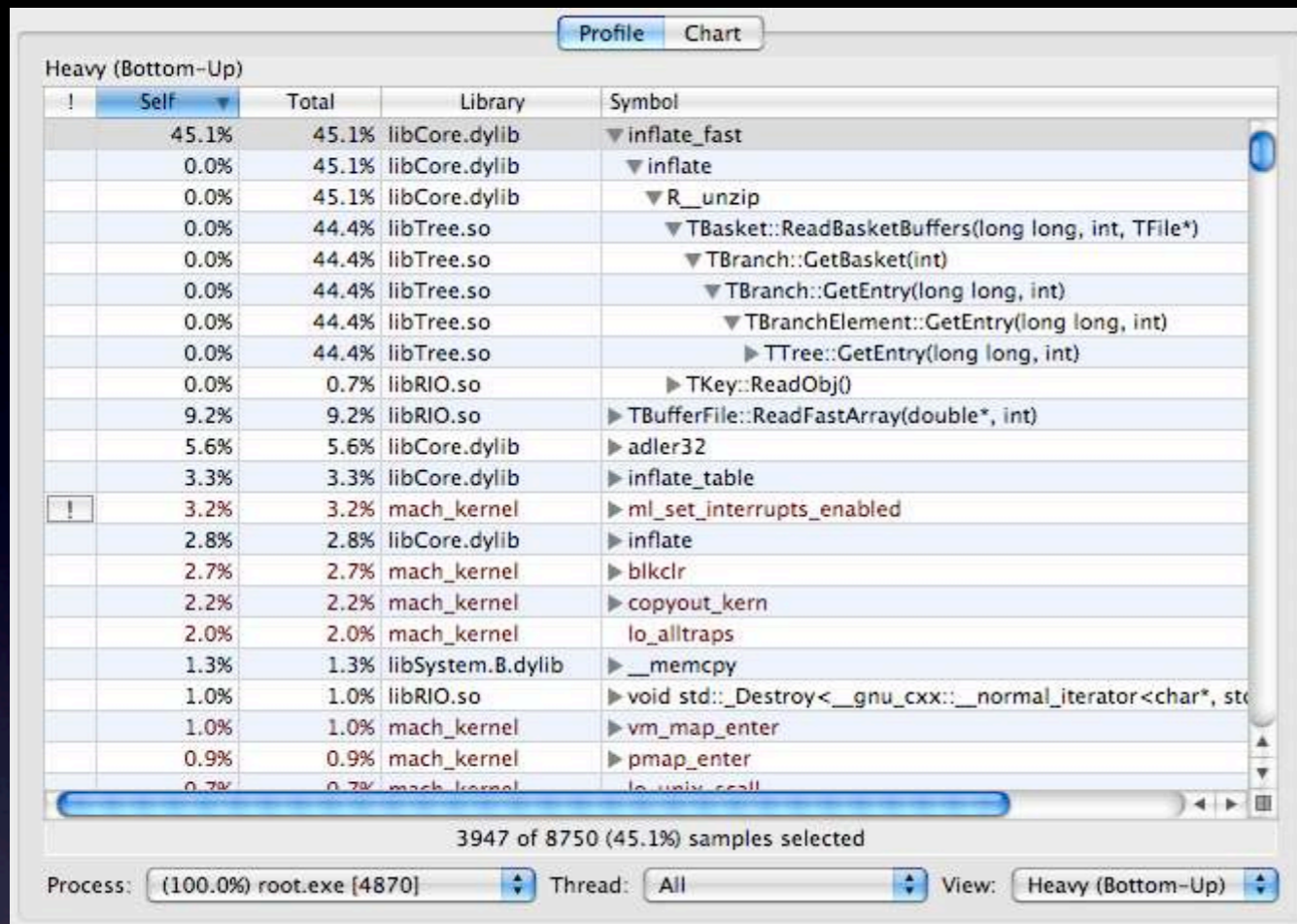
- a. Translating Code- Must be able to automatically take (parts) of existing software and covert to GPGPU code/integrate. Currently only support C, so concentrate on MadGraph (Fortran) and specific parts of ROOT (eg RooFit PDFs). C++ when Fermi arrives.

- 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

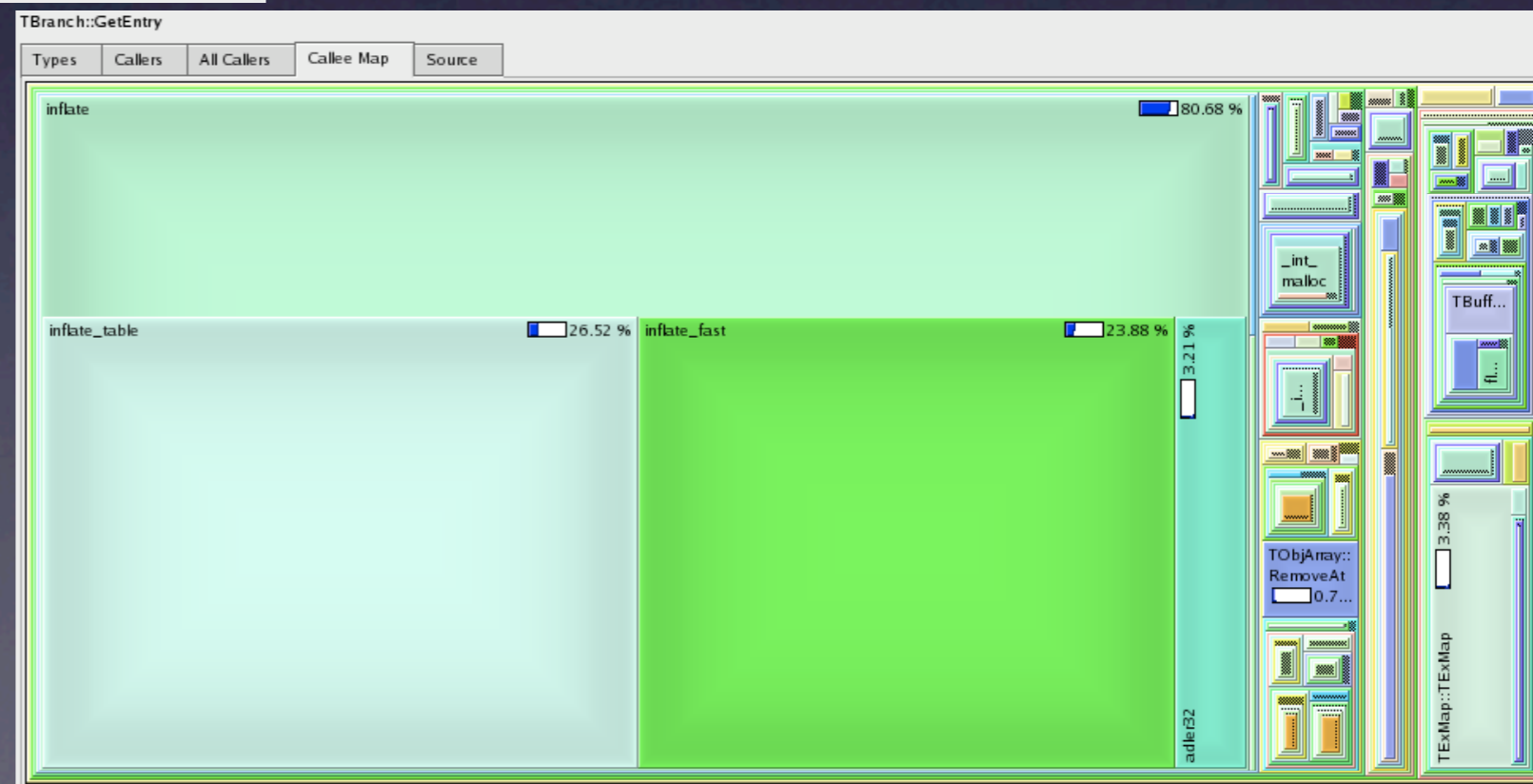
- 2 CS Masters students... very slow progress.
- They have learned some C++ programming basics, ROOT, CUDA, OpenCL
- Read lots of code... eg gzip.
- Plan has been to build experience through simple problems which are relevant to our physics goals
 - Need an approach which is computationally intensive 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 spend 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.

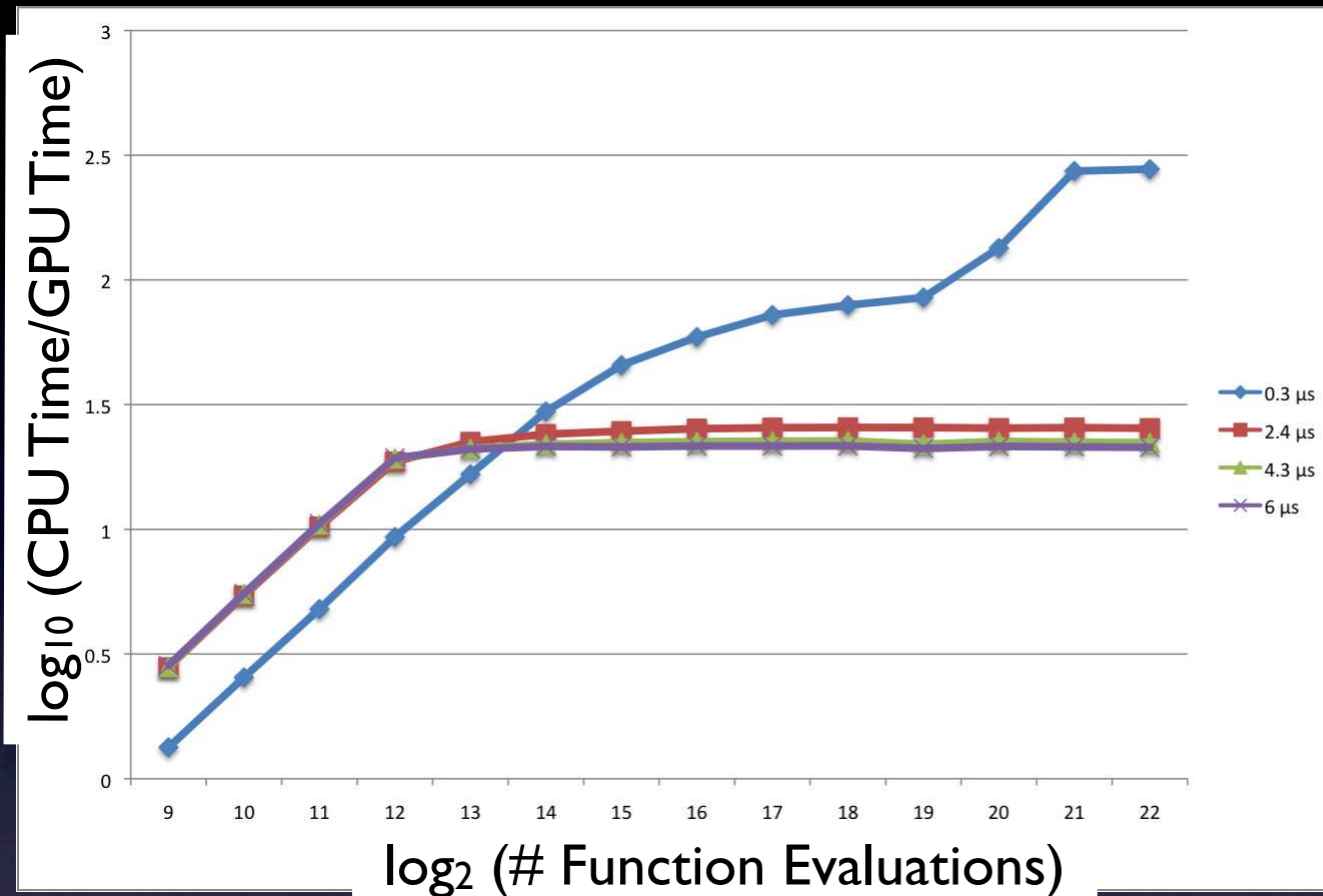
CPU Limited	Simultaneous Jobs	Compression Level	Rate (MB/s)	
Disk Limited			Hard Drive	Solid State Drive
Writing	1	0	25	25
	8	0	50	75
	1	9	4	4
	8	9	25	25
Reading	1	0	24	32
	8	0	20	28
	1	9	16	20
	8	9	14	17

- 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 bkg, this can take hours/event.

MC Integration



- 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.

MC Integrator

- Simple implementation... GPU optimization decouples from complexity of technique.
- Minimizing data in/out between the 3 step:
 - Random Number Generator Kernel
 - Integrator Kernel- Moving part of Random number gen and reduction here!
 - Reduction Kernel
- Expect less global memory access and therefore performance improvement.
- Instead we observe performance loss... profiling show increased global memory access. We don't understand how.
- Hopefully Box and I will get a chance to meet.

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.

$$R = \left(\sum_{i=1}^{n_{\text{var}}} |x_i - y_i|^2 \right)^{\frac{1}{2}}$$

- Sort them.

- Count number of signal/background in some region.

$$P_S = \frac{k_S}{k_S + k_B}$$

- 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 is implemented and being validated.

Final Remarks

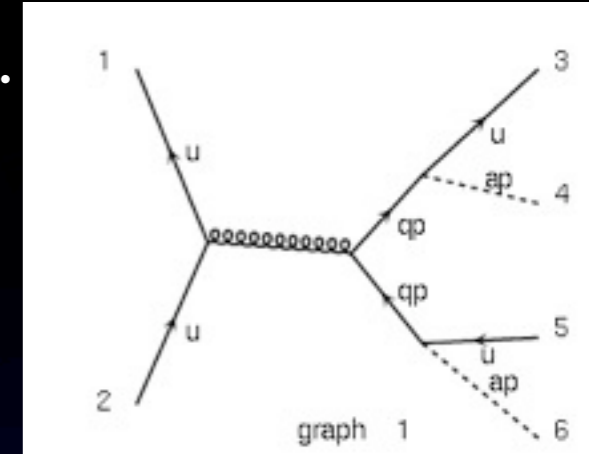
- Premature to look at CPU/GPU TFlops plots and say our computation problems are solved.
 - HEP software is too complicated.
 - 2 types of problems:
 - optimization of algorithms
 - implementation (within our current software)
 - Requires real R & D...
- 3 Early areas where we may be able to make impact 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.

