Adapting HEP applications to run on GPUs

Alfio Lazzaro (alfio.lazzaro@cern.ch)
CERN/Openlab

2nd Workshop on adapting applications and computing services to multi-core and virtualization

June 21th - 22th, 2010

CERN

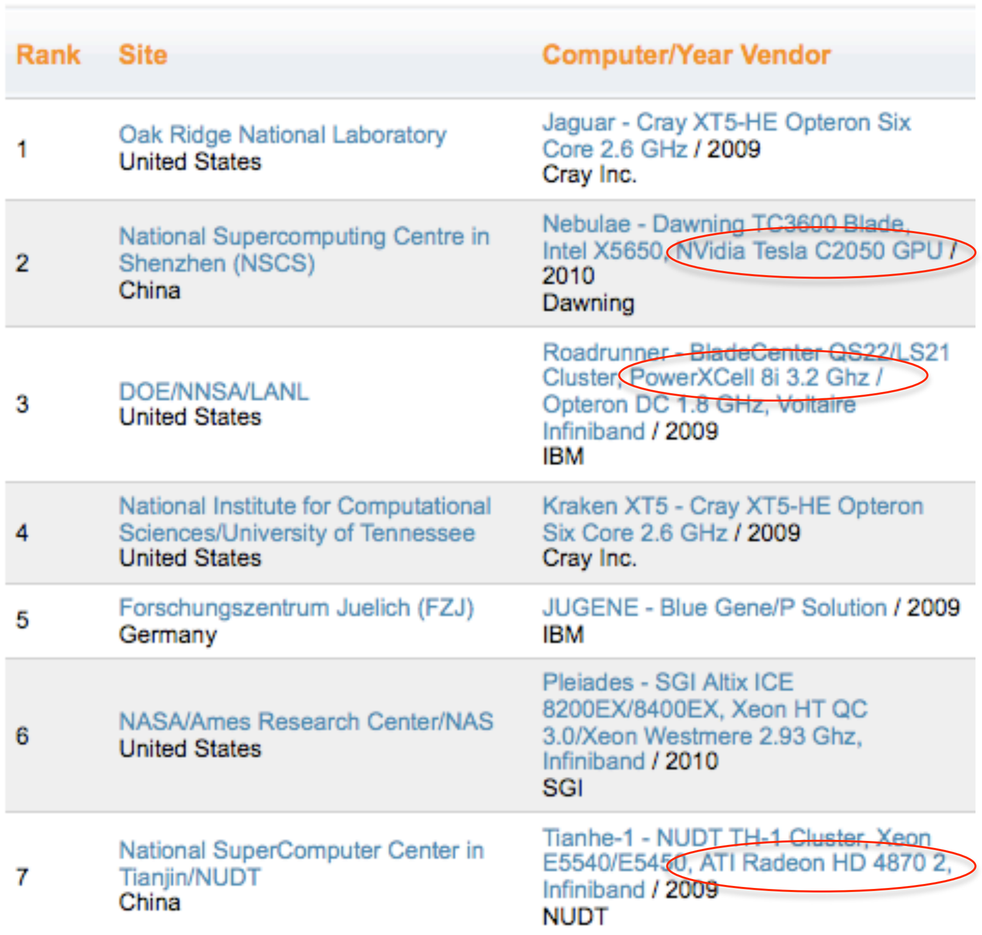
Introduction: Accelerator systems

- GPUs belong to general accelerator systems:
  - Devices that assist a main computer for speeding a specific calculation
    - Cell BE, ClearSpeed, GPU, Intel Larrabee, etc
- Basically we are talking about many-core accelerators:
  - (Massive) Parallel computer on a chip
  - The only way to get benefit from these cards is to write code highly parallelizable
    - Same difficulties raised in parallel computing
    - Very high performance on specific tasks (high throughput)

Taking as reference HPC…

- Several supercomputers are now based on accelerator systems (TOP500, June 2010)

New entry in June 2011!



Rank	Site	Computer/Year Vendor
1	Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.
2	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning
3	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2009 IBM
4	National Institute for Computational Sciences/University of Tennessee United States	Kraken XT5 - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.
5	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2009 IBM
6	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon Westmere 2.93 Ghz, Infiniband / 2010 SGI
7	National SuperComputer Center in Tianjin/NUDT China	Tianhe-1 - NUDT TH-1 Cluster, Xeon E5540/E5450, ATI Radeon HD 4870 2, Infiniband / 2009 NUDT

GPUs: hardware

- There are two main vendors:
  - NVIDIA
  - ATI/AMD
- Both vendors are in very high competition, providing "monster" cards in performance

Specifications	Westmere X5670	NVIDIA GTX 480	AMD FirePro 3D V8800
Processing elements	6 cores, 2 threads, 2 way SIMD @ 2.93 GHz	16 streaming multiprocessors, 30 cores, 32 threads @ 1.4 GHz	80 units, 32 processors @ 825 MHz
DP GFLOPS	70	336 (4.8x)	528 (7.4x)
Power (Watt) *	95	250	208
GFLOPS/Watt	0.74	1.34	2.54
Price ($)	1,440 for 1k units	499	1,499

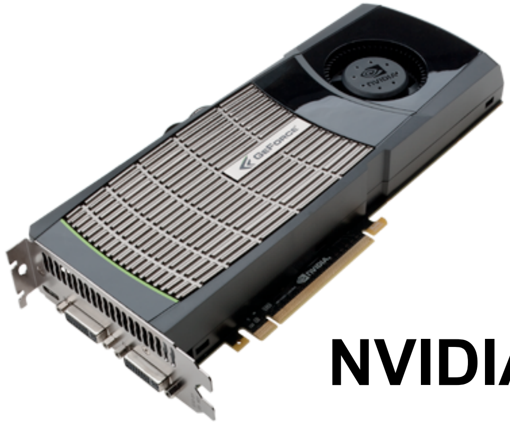
* Note that a GPU requires a host PC with CPU (at least 600 Watts in total)

GPUs: hardware

- I will not give details on the hardware
  - Look at the vendors website
- However, note the difference nomenclature CPU vs GPU for:
  - # processors (cores VS streaming processors)
    - ATI: 1600, NVIDIA: 480
    - CPU: 6-8 (12 for AMD)
  - # threads:
    - GPUs are based on thread parallelism, up to 15630 threads in total (light software thread)
    - 16 on Westmere (2 per each core)
  - # memory access:
    - GPUs have different types of memory (Global, Shared, L1/L2 Cache, Texture Cache): access to global memory is 159 GB/s
    - CPU has cache levels (L1/L2/L3): access to memory 25.6 GB/s
  - Double Precision VS Single Precision:
    - GPUs are optimized for SP (~4x FLOPS than DP)
    - Not all cards (e.g. notebook cards) support DP
    - Not always the cards are IEEE-754 compatible

GPU HPC solution

- The GeForce GTX 480 (code name GF100) is a "normal" GPU (for gamers)
  - It is based on the FERMI architecture (details later), announced last March

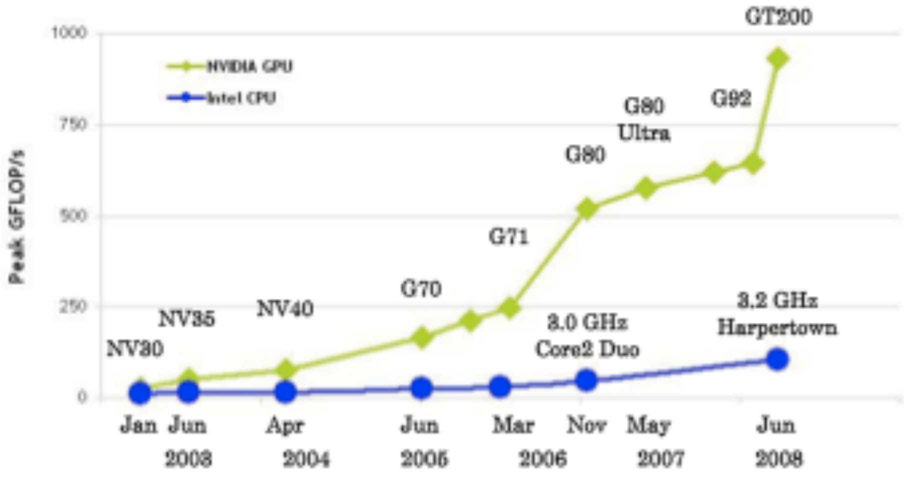


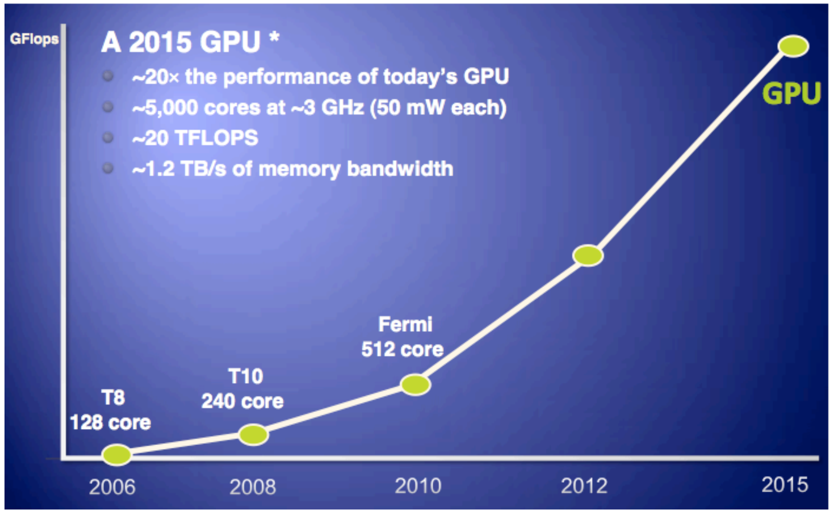
NVIDIA

- NVIDIA has developed specific cards for HPC, code name Tesla C2050/C2070
  - Still not available
  - No video output
  - Up to 4 cards stacked in a single box (rack module)
- AMD has a similar solution for HPC: FireStream architecture

So far so good

- The hardware performance are impressive
  - AMD and NVDIA are “almost” equivalent
  - NVIDIA marketing:





This is a sketch of a what a GPU in 2015 might look like; it does not reflect any actual product plans.

- But this is an half of the story
  - Reality (as usual) is more complicated…

GPUs: Software

- The real challenge is programming the card
  - **GPU programming is definitely different from CPU programming**
- AMD and NVIDIA provide two different solutions (which can be used only for their GPUs):
  - AMD: SDK which includes “Brook+”, an AMD hardware optimized version of the Brook language developed by Stanford University, itself a variant of the ANSI C, open-sourced and optimized for stream computing
  - **NVIDIA: CUDA** provides familiar programming concepts (C/C++ language) while developing software that can run on the GPU. CUDA compile the code directly mapping it on the hardware
- Other solutions, which guarantee portability, are:
  - OpenCL: developed by Khronos Group
  - Compiler companies that have developed language add-ons to simplify writing portable codes, e.g. RapidMind, CAPS HMPP or the new PGI compiler with GPU support

CUDA

- CUDA development tools can be integrated with the conventional C/C++ compiler, so one can mix GPU code with general-purpose code for the host CPU
- Current version (3.0) supports C and C++ languages. However, there are few important limitations for the C++:
  - Does not allow virtual functions
  - Does not allow inheritance
  - (interesting that the White Paper for the Fermi architecture claims that these limitations are now overcome, which means that they are supported by the new hardware. Maybe there will introduced in the software in the next release of CUDA…)
- In my opinion the best way to programming in CUDA is using C (as in previous version of CUDA)

CUDA Toolkit

- NVCC C/C++ compiler
- CUDA FFT and BLAS libraries for the GPU
- CUDA-gdb and CUDA-memcheck hardware debugger
- CUDA Visual Profiler
- There is the interesting possibility to emulate the hardware (but it is deprecated release 3.0)
- Stable, free available, documented and supported for Windows, Linux and MacOS
  - A lot of applications and blogs on the web

CUDA Parallelism

- CUDA automatically manages threads:
  - It does NOT require explicit management for threads in the conventional sense (in the same fashion of OpenMP)
- Fill the card with the maximum number of threads
  - Based on SIMT (Single Instruction Multiple Threads)
- Hierarchy of concurrent threads:
  - Parallel kernels composed of many threads
    - all threads execute the same sequential program
  - Threads are grouped into thread blocks
    - threads in the same block can cooperate (synchronization & data sharing)
    - Blocks must be independent to have the best scalability
  - Threads/blocks have unique IDs
- A CUDA thread has its own PC, registers, processor state, etc and there is no implication about how threads are scheduled
- The GPUs automatically exploits the SIMD vectorization

“Hello World” example

```
//Compute vector sum C=A+B
//Each thread performs one pairwise addition
__global__ void vecAdd(float* a, float* b, float* c) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   c[i] = a[i] + b[i];
}

int main() {
   //Run N/256 blocks of 256 threads each
   vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);
}
```

- Some CUDA keywords:

```
__global__ void KernelFunc(...);   // kernel callable from host
__device__ void DeviceFunc(...);   // function callable on device
__device__ int  GlobalVar;         // variable in device memory
__shared__ int  SharedVar;         // in per-block shared memory
```

C++ class definition in CUDA

```
class Add
{
public:
    __device__
    float
    operator() (float a, float b)
    const
    {
        return a + b;
    }
};
```

```
// Device code
template<class O>
__global__
void
VectorOperation(const float * A, const float * B,
                      float * C, unsigned int N, O op)
{
    unsigned int iElement = blockDim.x * blockIdx.x + threadIdx.x;
    if (iElement < N)
    {
        C[iElement] = op(A[iElement], B[iElement]);
    }
}
```

```
// Host code
VectorOperation<<<blocks, threads>>>(v1, v2, v3, N, Add());
```

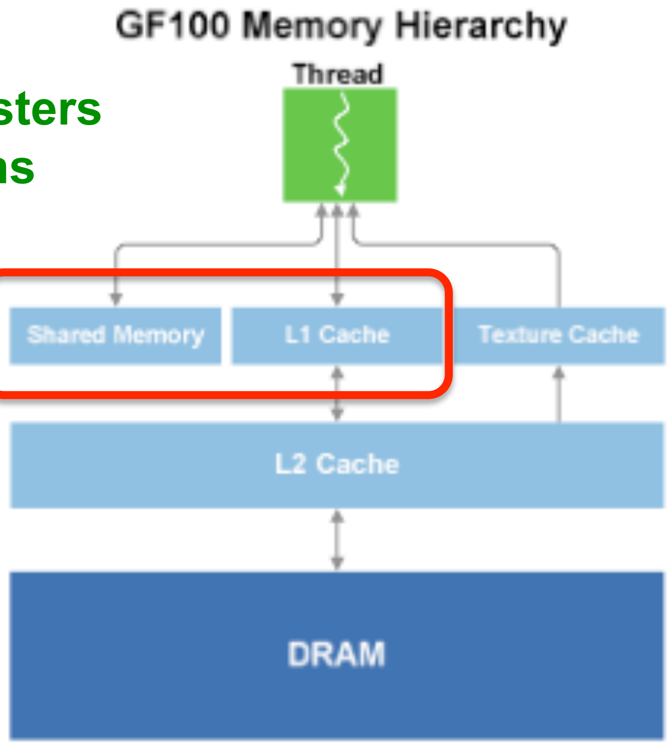
CUDA (Fermi) Memory model

- One of the main problem is the memory management
  - Keep the data close to the threads
  - Move data from main (CPU) memory to GPU memory using the PCI Express

Internal registers for operations

64 KB in total, ~speed of registers

Introduced with Fermi: 768 KB (all clients read/write)



12 KB, read-only

~150x slower than L1 Cache (ECC)

Memory and Performance

- CUDA uses 3 memories definitions on the DRAM (they differ only in caching algorithms and access models):
  - Global memory: has the lifetime of the application
  - Local memory: lifetime of the thread
  - Constant memory (read-only)
  - Texture memory (read-only)
- CPU host can refresh and access only: global, constant, and texture memory
- It is really important to keep data close to threads
  - Correct usage of the memories
  - Reuse your data
  - Reduce communications in the threads
- Move data from CPU memory to GPU memory is a time expensive operation (PCI-E 2.0 x16 has 4 GB/s)
  - The effect can be alleviate with the "zero copy" feature which enables GPU threads to directly access host memory

Putting everything together

- A GPU requires:
  - High degree of task parallelism
  - High degree of data parallelism (vectorization)
  - Correct use of memory (and cache levels)
- There are all characteristics which are fulfill by few applications in HPC, for example:
  - Algebra (BLAS)
  - FFT
  - (note that these are provided with CUDA. Note that Linpack benchmark used for the Top500 is an algebraic package…)
- But what about more general applications?
  - In HPC the discussion is ongoing:
    - Recent IBM paper: “Believe it or Not! Multicore CPUs can Match GPUs for FLOP-intensive Applications!”
    - Few interesting articles in http://www.hpcwire.com/:
      - http://www.hpcwire.com/blogs/GPU-Computing-The-Inevitable-Transition-96611294.html
      - http://www.hpcwire.com/news/Supercomputings-Future-Is-it-CPU-or-GPU-96482939.html

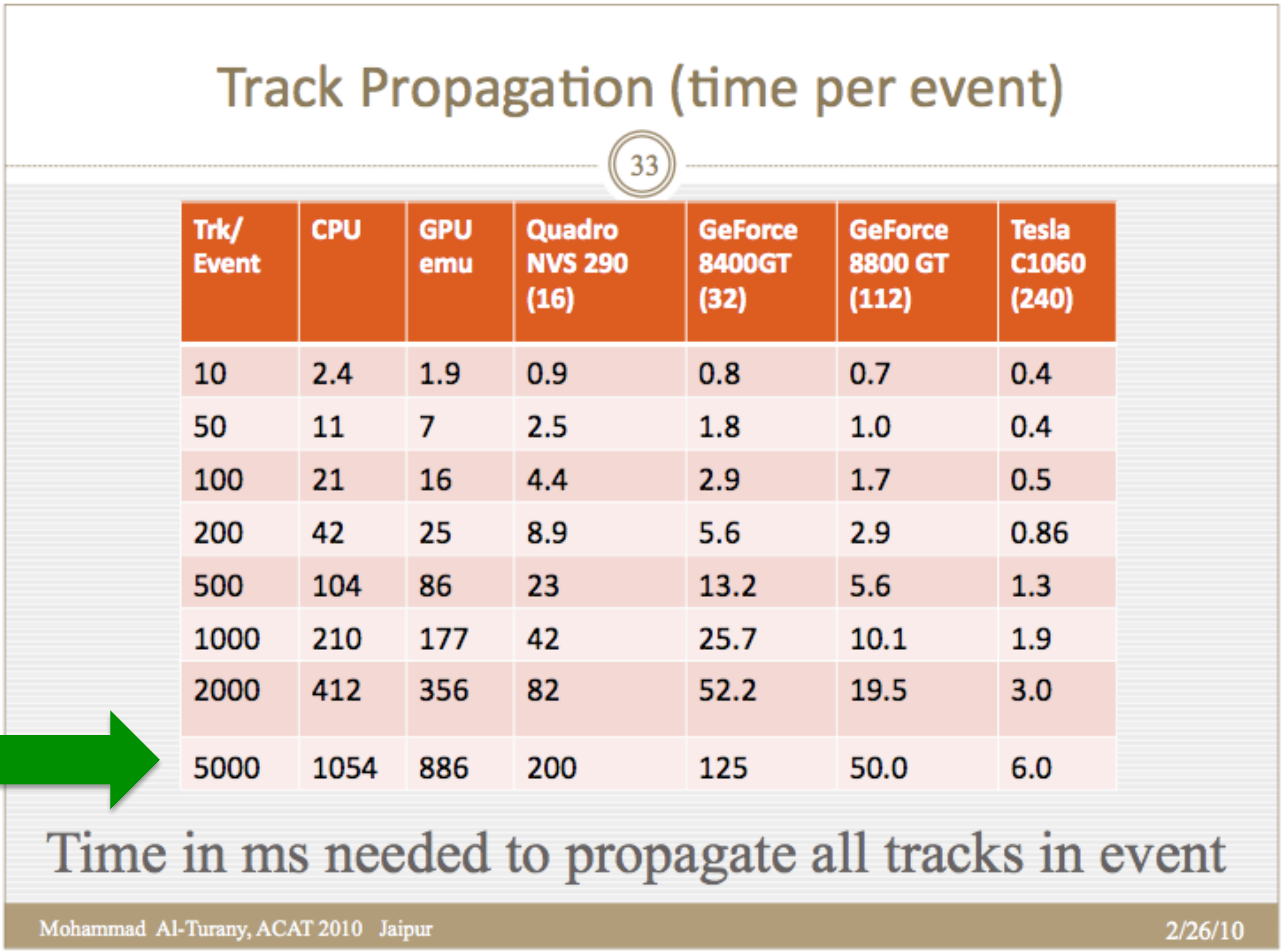
What about HEP?

- I think there are few possible applications of GPUs in HEP (giving current difficulties to move our frameworks to parallel versions):
  - Online reconstruction
  - Data analysis (?)
- They requires to rethink the algorithms
  - I must say also to rewrite them in C…
- I cannot image a situation where we use GPUs on a distributed system like GRID (sorry for that)…
- I think the possible use of GPUs is at centralized level (cluster of GPUs for online) and at user level (GPU on your laptop/desktop)

Online applications on GPUs

- Fill the GPUs with tracks
- Usually they use Single Precision (take the maximum from the card)
- Take a look at the talk by Mohammad Al-Turany at ACAT10 for GSI FairROOT:
  - http://indico.cern.ch/contributionDisplay.py?contribId=147&confId=59397
  - Nice examples:
    - Using texture memory for field maps
    - Geant3 Runge-Kutta propagator rewritten as a CUDA kernel
- Alice people are ready to use GPUs for heavy ions collisions reconstruction
  - See David Rohr at the International Tracking Workshop, GSI, June 7-11, 2010 (https://www.gsi.de/documents/FOLDER-9871272014070.html)
- I know that there is a work ongoing in Atlas to use GPUs for trigger
  - I'm not aware of other projects. Sorry if your project is missing in my list…

Mohammad's results



Track Propagation (time per event)

33

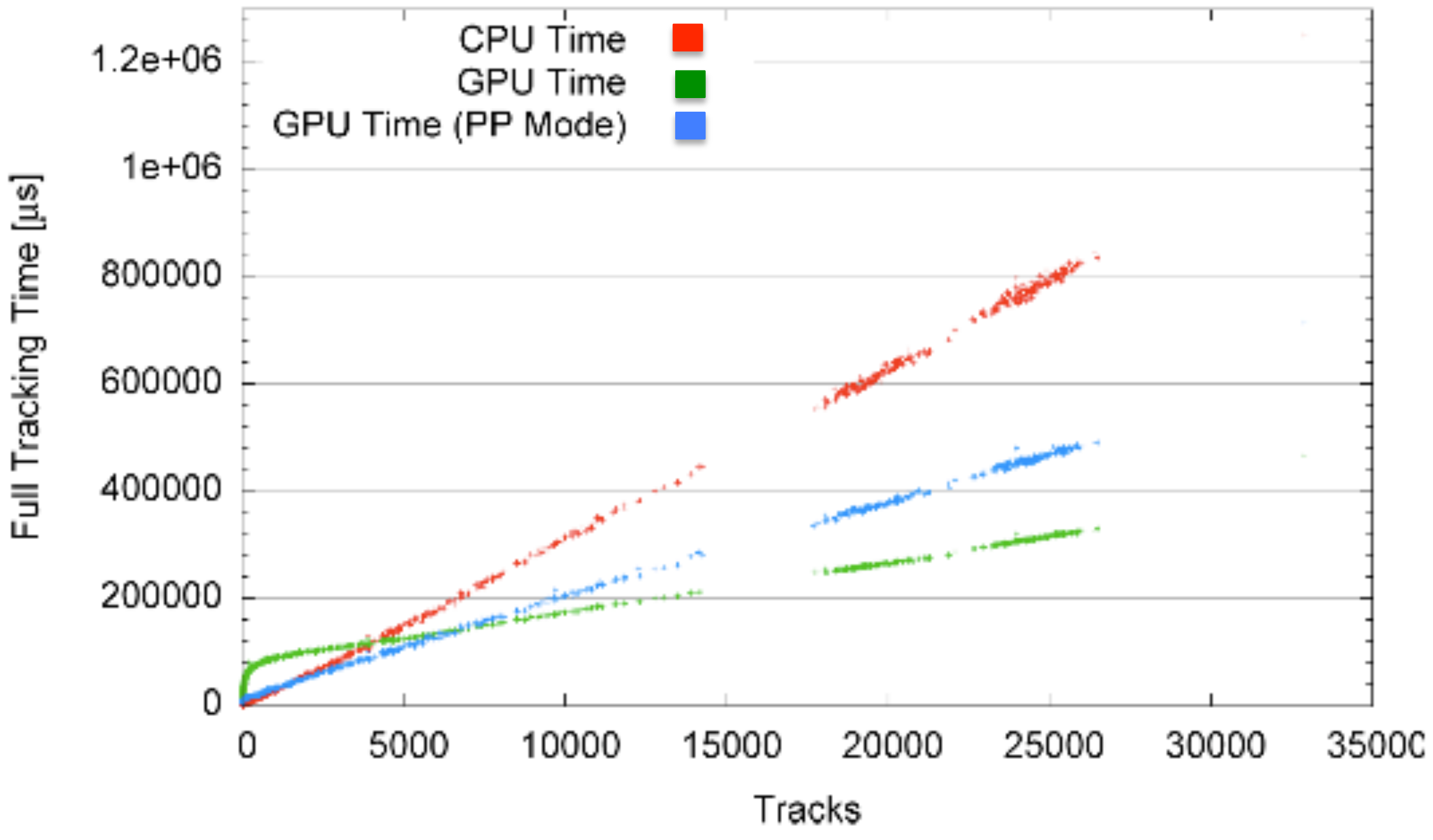
Trk/ Event	CPU	GPU emu	Quadro NVS 290 (16)	GeForce 8400GT (32)	GeForce 8800 GT (112)	Tesla C1060 (240)
10	2.4	1.9	0.9	0.8	0.7	0.4
50	11	7	2.5	1.8	1.0	0.4
100	21	16	4.4	2.9	1.7	0.5
200	42	25	8.9	5.6	2.9	0.86
500	104	86	23	13.2	5.6	1.3
1000	210	177	42	25.7	10.1	1.9
2000	412	356	82	52.2	19.5	3.0
5000	1054	886	200	125	50.0	6.0

Time in ms needed to propagate all tracks in event

Mohammad Al-Turany, ACAT 2010 Jaipur 2/26/10

ALICE TPC Online Tracking on GPU

Final Speedup in Contrast to Event Size



(PP Mode: Special variant optimized for small scale events)

Data analysis applications

- Data analysis can be considered at user level
  - Provide a common interface to compute some algorithms on the CPU/GPU
  - **Use Double Precision**
- There are some cases where we can "directly" use GPUs
  - Algebra computation
  - Convolutions using FFT
- In other cases we need to (deeply) rethink the code
  - Code is based on C++ virtual classes (basically ROOT classes)
  - In some cases we need to "downgrade" to C
  - In any case I think the first step is to optimize and parallelize the current version code for CPU and then move to GPU (for very specific cases)
    - Most of data analysis will never fill a GPU (parallelization on CPU is more than enough)

Data analysis applications

- In Openlab we are starting a project to exploit the GPUs for RooFit/RooStats, which are package for data analysis in ROOT:
  - Working with the authors
  - Taking complex example of joint Atlas-CMS analysis
- We will use CUDA (and OpenCL as reference)
- Basically we have a function and we calculate it on several data
  - First difficulty is that the function is a composition of C++ classes with virtual methods
  - Try to have a plain "C" implementation
  - However, it can be the case that we will never able to fill the card
    - Try different algorithms, like Genetic algorithm, which are highly parallelizable
- I know there are other works in HEP with similar goals, but I don't know of any available code

Conclusions

- GPUs are promising devices for the future in case of intensive calculations
- However they are not easy to program
  - Requires massive parallelization (not all applications can benefit)
- NVIDIA provides a SDK with CUDA, but it is device-dependent
  - OpenCL will provide an hardware-free implementation (OpenCL is similar to the CUDA programming style)
- There are good results for online track reconstruction and trigger

Conclusions

- Data analysis can be another field suitable for GPUs
  - Work in progress in Openlab
- Note that in any case it is mandatory to migrate our code to parallel version already for CPU
  - Use GPUs parallelization could not be trivial
  - Of course, this not prevent to try the porting of our code on GPUs (sooner is better than later)
- There are many HPC applications already running on GPUs (see http://gpucomputing.net/)
  - Is the GPUs a real solutions in HPC? Discussion is ongoing… For sure they will play an important role
  - In future other accelerator solutions will be available, like Intel Knight Ferry, based on Larrabee architecture: x86 compatible, much easier to program…