HLT with GPU

N. Katayama, S. Lange

Trigger/DAQ Workshop

Jan. 26, 2010

Outline

- GPU
- Belle II HLT
- Coding examples

GPU is hot

- GPGPU (general purpose graphics processing unit) is becoming popular
  - In 2008, I was looking at Cell CPUs (used in play station 3) and earlier GPUs
  - Nvidia announced a new architecture called Fermi early October 2009. The products using it will come out soon (Q2/3, 2010)
    - 520-630 double precision Gflops/ GPU(peak) as opposed to 78 Gflops for the current generation
    - These numbers do not include transfer time between memories of CPU and GPU

C1060

C2070

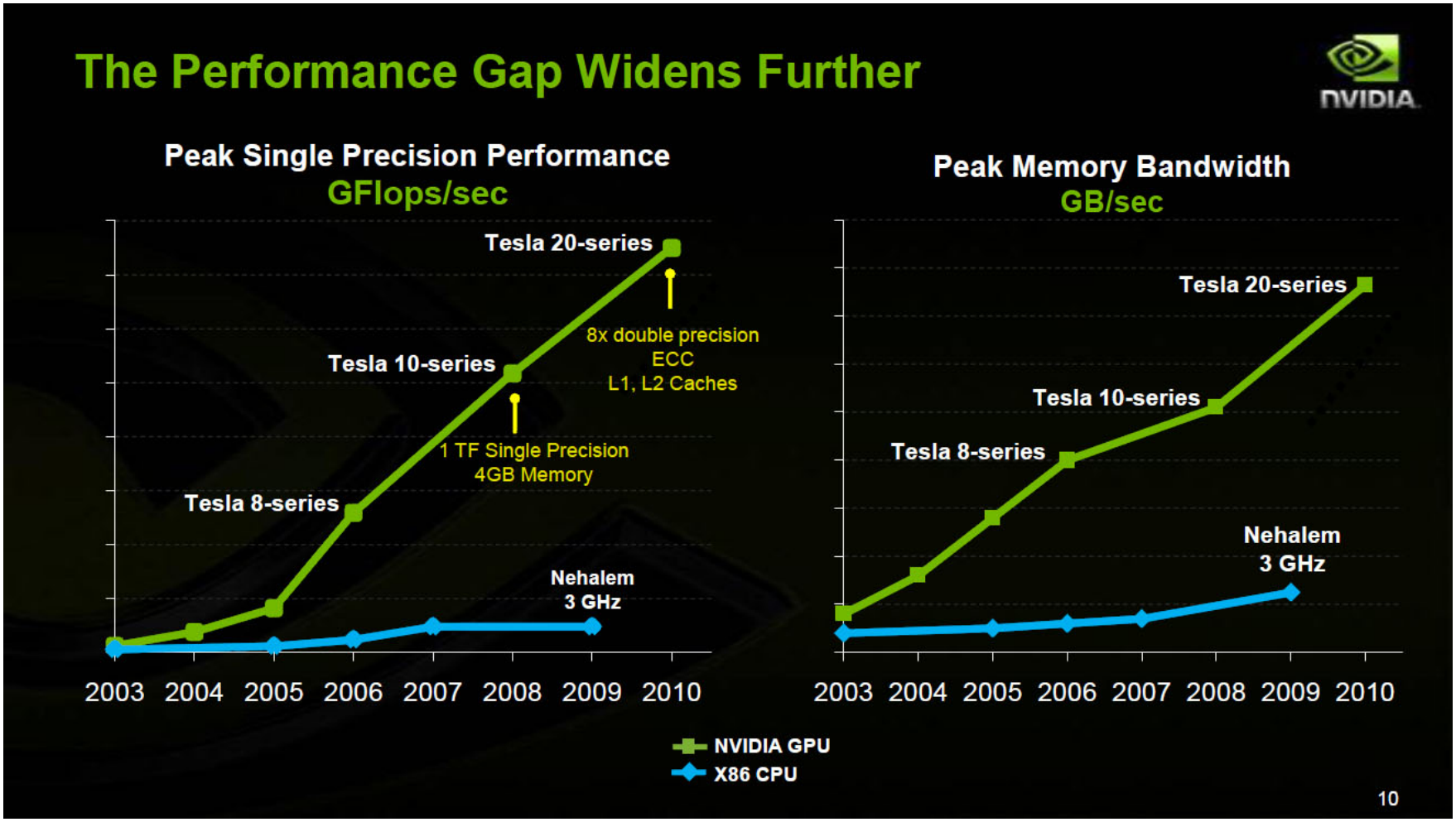
CPU and GPU

GPU promises 1/10 of cost and 1/20 of power consumption

Processor	Intel Core 2 Extreme QX9650	NVIDIA TESLA C2070
Transistors	820 million	1.4 billion
Processor clock	3 GHz	1.3 GHz?
Cores	4	512
Cache / Shared Memory	6 MB x 2	16-48 KB/768KB(L2)
Threads executed per clock	4	512
Hardware threads in flight	4	24576
Memory controllers	Off-die	384bit
Memory Bandwidth	12.8 GBps	64bytes/clock?

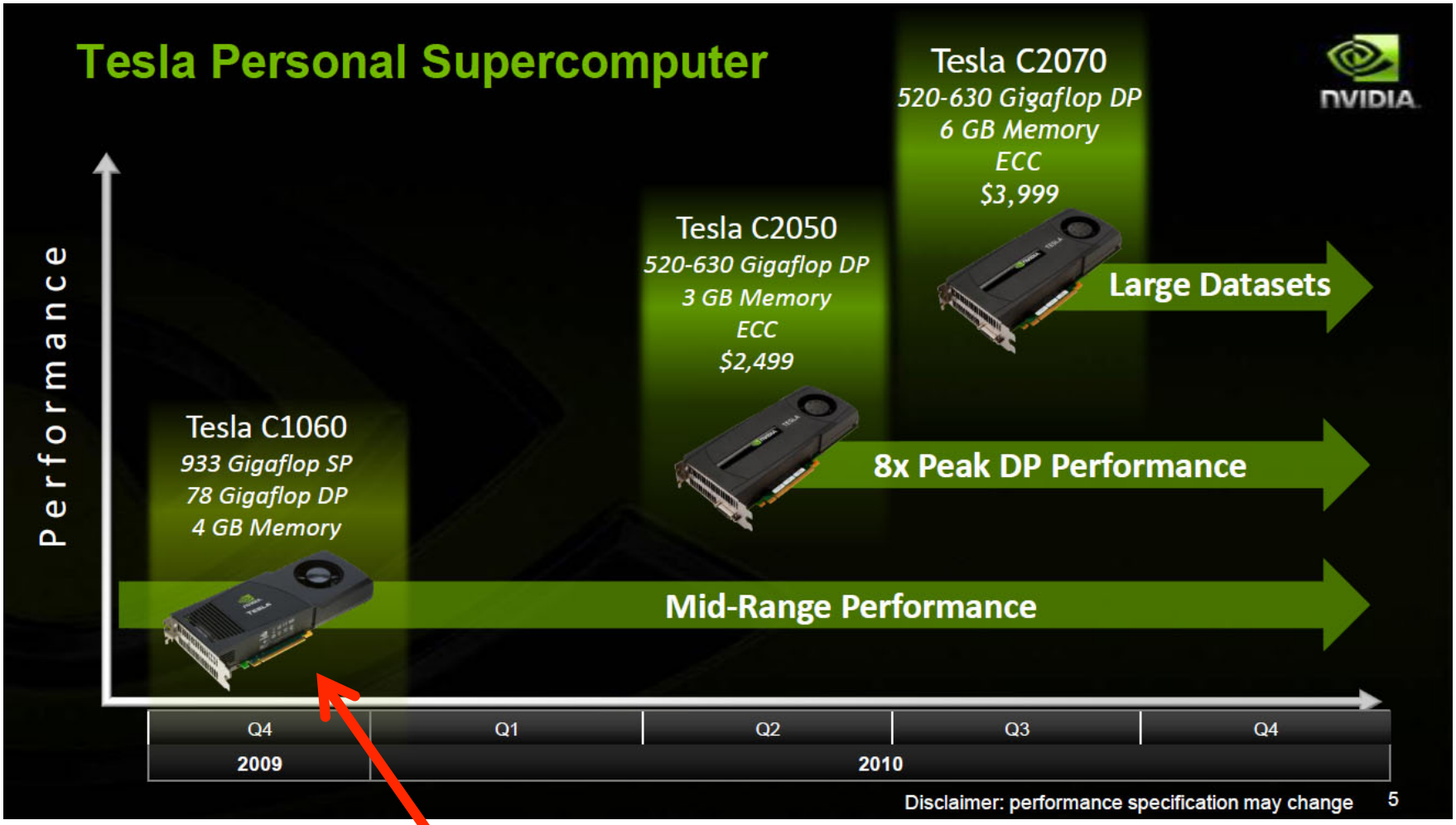
PCIe x16 < 8GB/s aggregate

Fermi GPU



>600 GFLOPS/sec (Double Precision)/chip

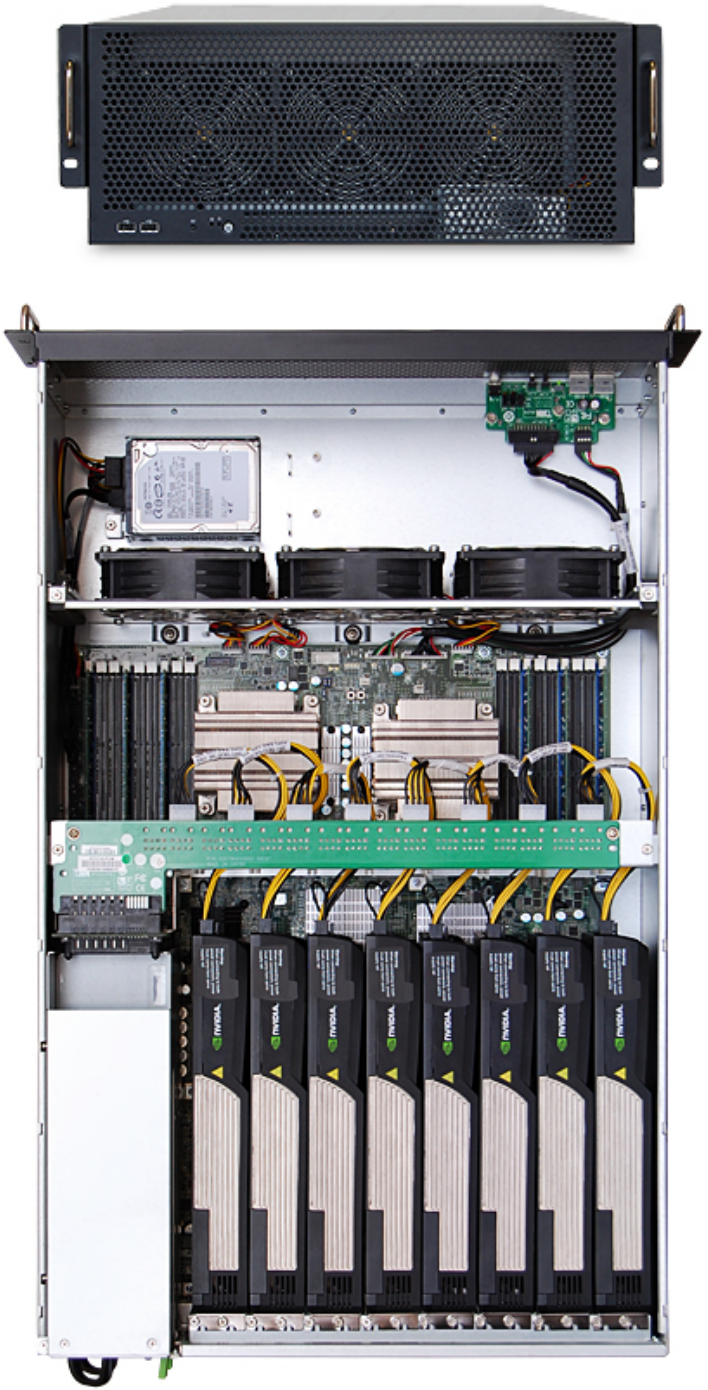
It’s real



Only <$1,000 now

Box with lots of GPUs

- For example, this box has
  - 2 CPU sockets
  - 8 PCIe X16 bus
  - Max. 144 + 48(on GPU) GB memory
- Can install up to 8 Tesla GPU boards (5 TFLOPS with Tesla 20)
- GSI group is now developing a special card (PCIe 4e) that will read the DATA from DAQ
  - We can directly read PXD/SVD data
- Only for less than $10,000/box (basic configuration, one CPU+GPU)
- Box with 4 GPUs/no CPU in 1U chassis



What's new with Fermi/Cuda 3.0

- Nvidia has changed the strategy a bit so that it is even more usable (programmable)
- Double precision operation is 8 times faster than before
- ECC
- It uses the same address space as CPU so that we can use objects (data part only)
- More C++ support in Cuda 3.0
- Some compiler (PGI for example) can produce GPU code automatically from normal C99/Fortran programs

Belle II HLT

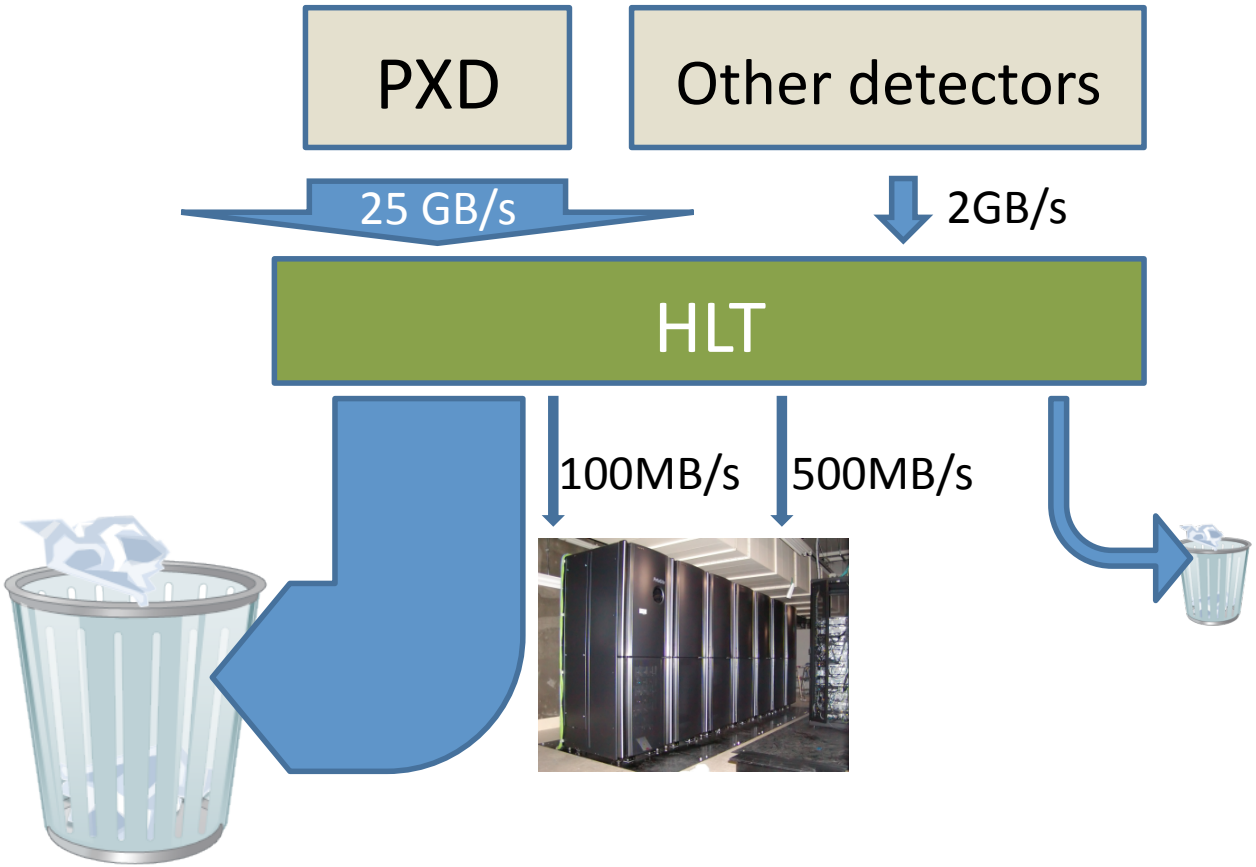
The biggest computing challenge

Belle II HLT

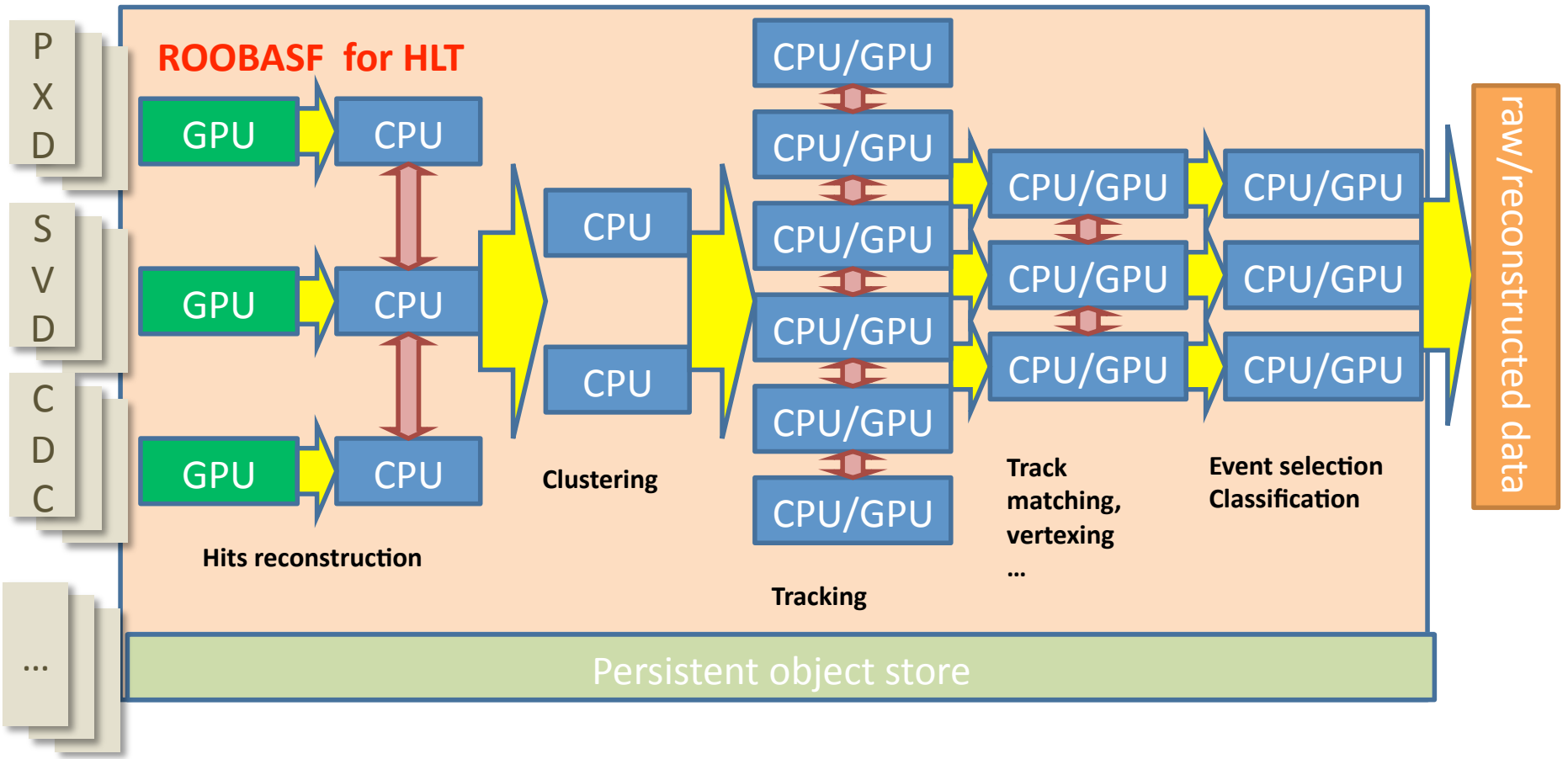
- Belle HLT working well (200 CPU core or so)
  - It takes 0.4 second to analyze one Hadronic event
  - L1 trigger rate is ~500Hz, HLT output rate is ~200Hz
  - Real Hadronic events: 100Hz
- Belle II HLT: much harder problem
  - L1 trigger rate 20KHz
  - HLT will reduce to 5KHz
    - Real Hadronic events:5KHz
  - Pixel detector produces 25GB/s data
    - We need to reduce down to 100MB/s or so

10,000 cores needed

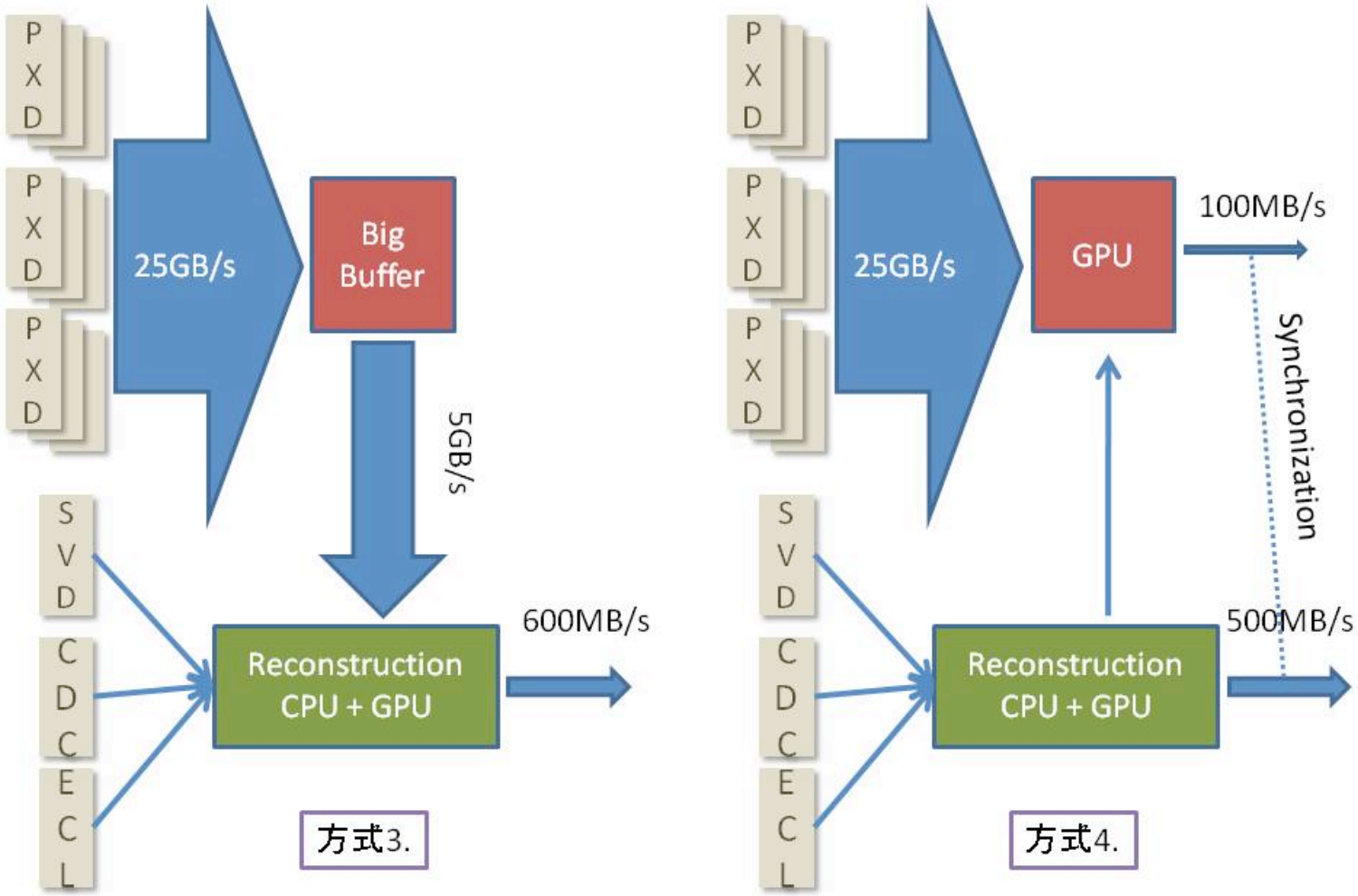
Belle II High Level Trigger



Roobasf for HLT



Two ways


P
X
D
P
X
D
P
X
D
25GB/s
Big
Buffer
5GB/s
S
V
D
C
D
C
E
C
L
Reconstruction
CPU + GPU
600MB/s
方式3.
P
X
D
P
X
D
P
X
D
25GB/s
GPU
100MB/s
Synchronization
S
V
D
C
D
C
E
C
L
Reconstruction
CPU + GPU
500MB/s
方式4.

GPU programming

- There are many ways to make use of GPUs
  - Write your own code using
    - CUDA (easier?)
    - OpenCL (promising?)
  - Using compilers that produce code running on GPUs
    - PGI (no C++)
  - Use existing libraries for GPU (cublas, magma…)
    - there might not be one
  - Hybrid approach
    - Use CUDA but use existing packages wherever possible

pyCUDA

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1))

print dest-a*b
```

You can embed GPU program in python.

It is dynamically compiled and executed on the fly

pyCUDA (gpuarray)

- Can manipulate GPU memory in python

```
import pycuda.gpuarray as gpuarray
a_gpu =
   gpuarray.to_gpu(numpy.random.randn(4,4).astype(numpy.float32))
#
#computation (X 2.0)is done in GPU memory
#
a_doubled = (2.0*a_gpu).get()
```

- pyCUDA supports packed sparse array (Garland and Bell, integrates with scipy.sparse)
  - maybe useful for pixel data reduction

pyUblas + boost.ublas + magma

- magma stands for “matrix algebra for GPU and multicore architectures”
  - It is not really for our problem of pixel data reduction
  - But the library is available for single GPU and we can compare the result with the standard implementation of BLAS routines
- In python (on CPU), one writes

  c = magma.gemm(a, b)
  - and the computation (matrix multiplication) is carried out on GPU
- d(s)gemm gives 60-75(200-370) GFLops/s on a C1060

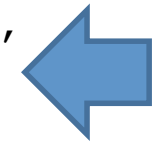
gemm code (C++)

```
double *d_A_m , *d_B_m , *d_C_m;
cublasAlloc( size_A1, sizeof(double), (void**)&d_A_m );
cublasAlloc( size_B1, sizeof(double), (void**)&d_B_m ) ;
cublasAlloc( size_C1, sizeof(double), (void**)&d_C_m ) ;

cublasSetMatrix( M, K, sizeof( double ), A, lda, d_A_m, lda ) ;
cublasSetMatrix( K, N, sizeof( double ), B, ldb, d_B_m, ldb ) ;
cublasSetMatrix( M, N, sizeof( double ), C, ldc, d_C_m, ldc ) ;

magmablas_dgemm (//Order,
                 TransA, TransB, M, N, K,
                 alpha, d_A_m, lda,
                 d_B_m, ldb,
                 beta, d_C_m, ldc);

cublasGetMatrix( M, N, sizeof( double ), d_C_m, ldc, C, ldc ) ;
cublasFree(d_A_m);
cublasFree(d_B_m);
cublasFree(d_C_m);
```



Tuned cuda code is inside magmablass_dgemm

magmablas_dgemm.cpp

I can't show you… I have not gotten permission but something like:

```
__shared__ double Bb[16][17];
const int tx = threadIdx.x;
const int ty = threadIdx.y;

int iby = ((blockIdx.y + blockIdx.x )
    % (n/16))*16;
const int idt = ty * 16 + tx;
int ibx = blockIdx.x *64+idt;
//int iby = blockIdx.y *16;

A += ibx ;
B+=tx+__mul24(iby+ty,ldb);
//A += ibx + idt;
//C += ibx +idt +__mul24( iby,ldc);
//C += __mul24(ibx +idt,ldc) + iby;
C += __mul24(ibx ,ldc) + iby;

const double *Bend = B + k;
```

```
double Cb[16] =
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

do {
    double Ab[4] = {A[0], A[lda],
          A[2*lda], A[3*lda]};
    Bb[tx][ty+0] = B[0*ldb];
    Bb[tx][ty+4] = B[4*ldb];
    Bb[tx][ty+8] = B[8*ldb];
    Bb[tx][ty+12] = B[12*ldb];

    __syncthreads();

    A += 4 * lda;
    saxpy(Ab[0], &Bb[0][0], Cb); Ab[0]
    = A[0*lda];
    saxpy(Ab[1], &Bb[1][0], Cb); Ab[1]
    = A[1*lda];
    saxpy(Ab[2], &Bb[2][0], Cb); Ab[2]
    = A[2*lda];
    saxpy(Ab[3], &Bb[3][0], Cb); Ab[3]
    = A[3*lda];
…
```

Performance measurements

- Started to measure timings
  - Copy from CPU to GPU: 5.2 GB/s
  - Copy from GPU to CPU: 1.8 GB/s
    - These measurements are with cublasSet/GetMatrix which I guess is not a straight copy as it has many arguments
  - Compute speed is fine
    - 60-75Gflops/s for double precision and 200-360 Gflops for single for C1060

Many other applications in HEP

- Reconstruction
  - Track finding/fitting
  - Cluster/pi0 finding/reconstruction
- Physics analysis
- Likelihood fits
- …
- Plan: Hope to investigate more
  - If you want to join, please let me know

backup slides

CPU vs. GPU

CPU

- CPU is designed to execute one stream of instructions as fast as possible.
- The CPU spends transistors on hardware features like instruction reorder buffers, reservation stations, branch prediction hardware, and large on-die cache.
- The CPU uses cache to improve performance by reducing the latency of memory accesses.

GPU

- GPU is designed to execute many parallel streams of instructions as fast as possible.
- The GPU spends transistors in processor arrays, multithreading hardware, shared memory, and multiple memory controllers.
- The GPU uses cache (or software-managed shared memory) to amplify bandwidth.

CPU vs. GPU

CPU

- The CPU handles memory latency by using large caches and branch prediction hardware. These take up a large deal of die-space and are often power hungry.
- CPUs support one or two threads per core.
- The cost of a CPU thread switch is hundreds of cycles.

GPU

- The GPU handles latency by supporting thousands of threads in flight at once. If a particular thread is waiting for a load from memory, the GPU can switch to another thread with no delay.
- CUDA capable GPUs support up to 1,024 threads per streaming multiprocessor.
- GPUs have no cost in switching threads. GPUs typically switch threads every clock.

CPU vs. GPU

CPU

- CPUs use SIMD (single instruction, multiple data) units for vector processing.
- Intel CPUs have no on-die memory controllers.

GPU

- GPUs employ SIMT (single instruction multiple thread) for scalar thread processing. SIMT does not require the programmer to organize the data into vectors, and it permits arbitrary branching behavior for threads.
- CUDA capable GPUs employ up to eight on-die memory controllers. As a result, GPUs typically have 10× the memory bandwidth of CPUs.

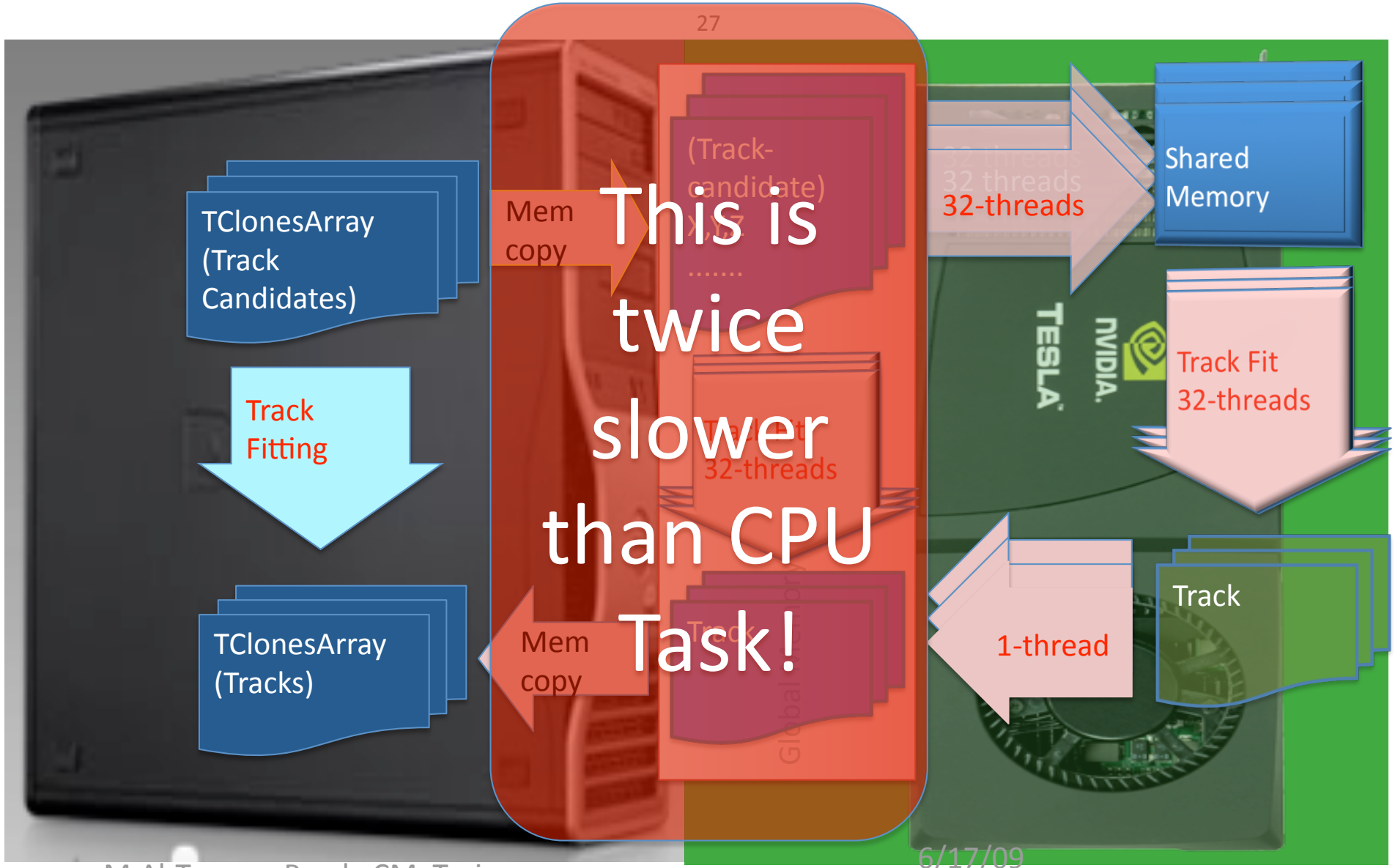
Working example
Tracking for Panda experiment

Mohammad Al-Turany

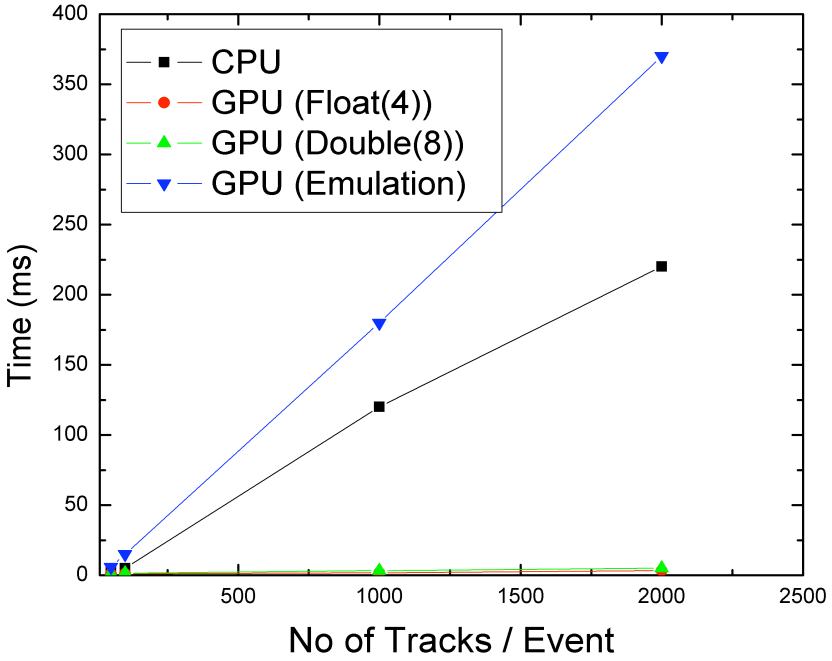
Panda

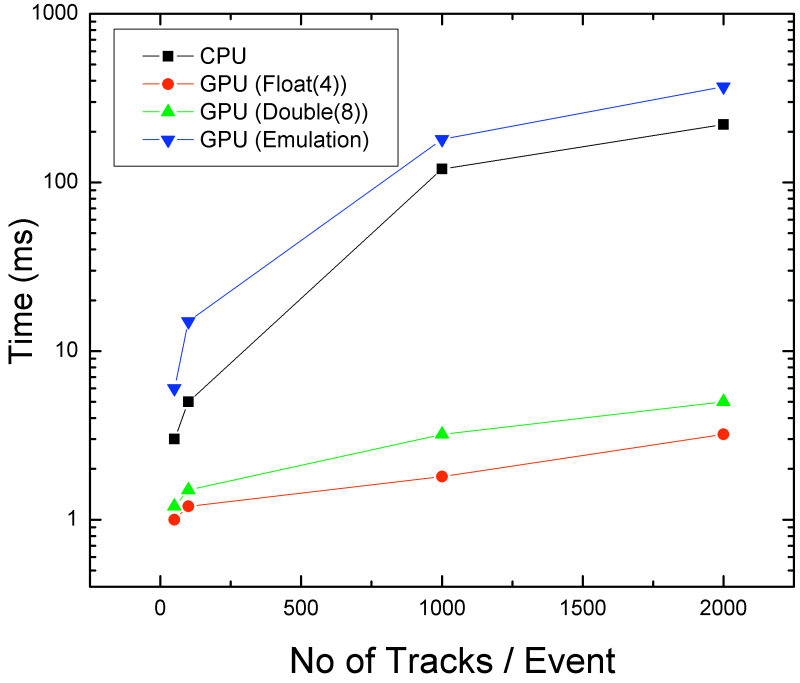
GSI

Track fitting: Implementation in CUDA (2.1)



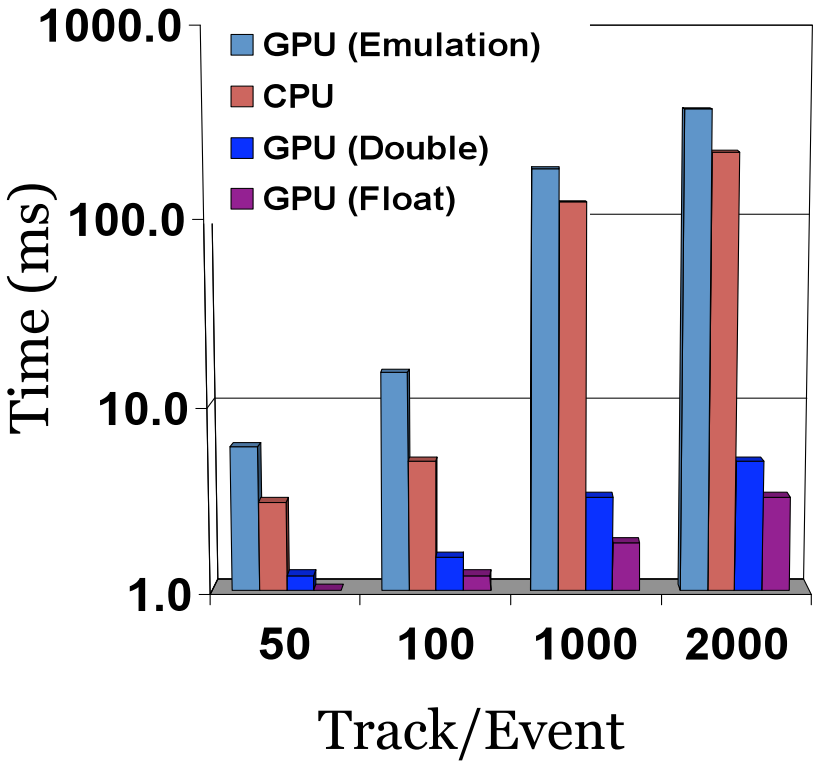
Track fitting on CPU and GPU

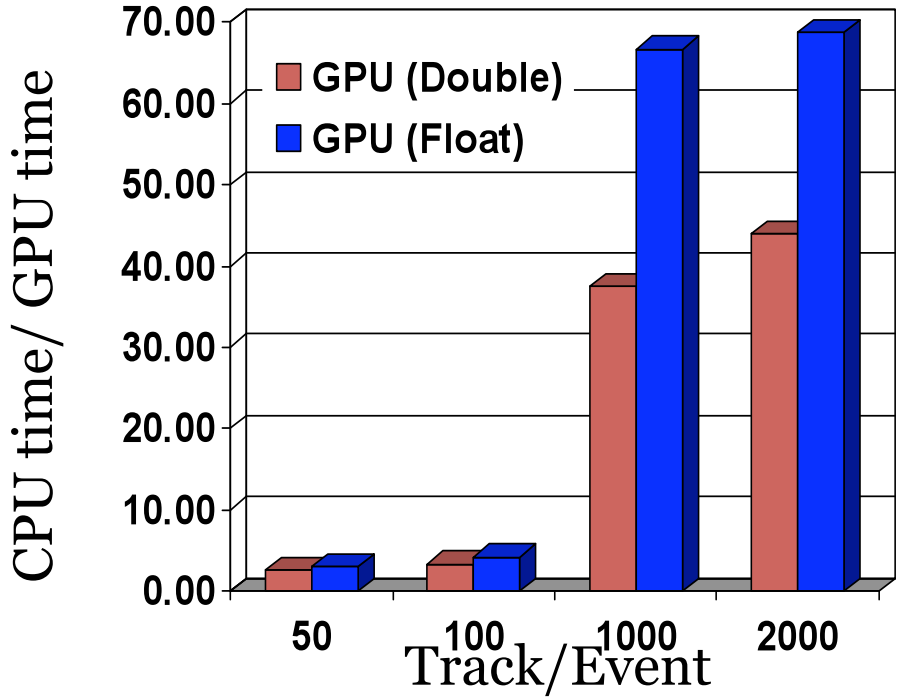




	50	100	1000	2000
GPU (Emu)	6.0	15.0	180	370
CPU	3.0	5.0	120	220
GPU (D)	1.2	1.5	3.2	5.0
GPU (F)	1.0	1.2	1.8	3.2

What we gain?





Track/Event	50	100	1000	2000
GPU (Double)	2.5	3.3	37.5	44.0
GPU (Float)	3.0	4.2	66.7	68.8

CUDA GPU occupancy calculator

- Tesla C1060: 30 Multiprocessor
  2048 Bytes shared memory per block

- In this test: **32** threads per block

Active Threads per Multiprocessor	**256**
Active Thread Blocks per Multiprocessor	**8**
Occupancy of each Multiprocessor	**25%**

- To Get the most of this card we should use **256** threads per block

Active Threads per Multiprocessor	**1024**
Active Thread Blocks per Multiprocessor	**4**
Occupancy of each Multiprocessor	**100%**

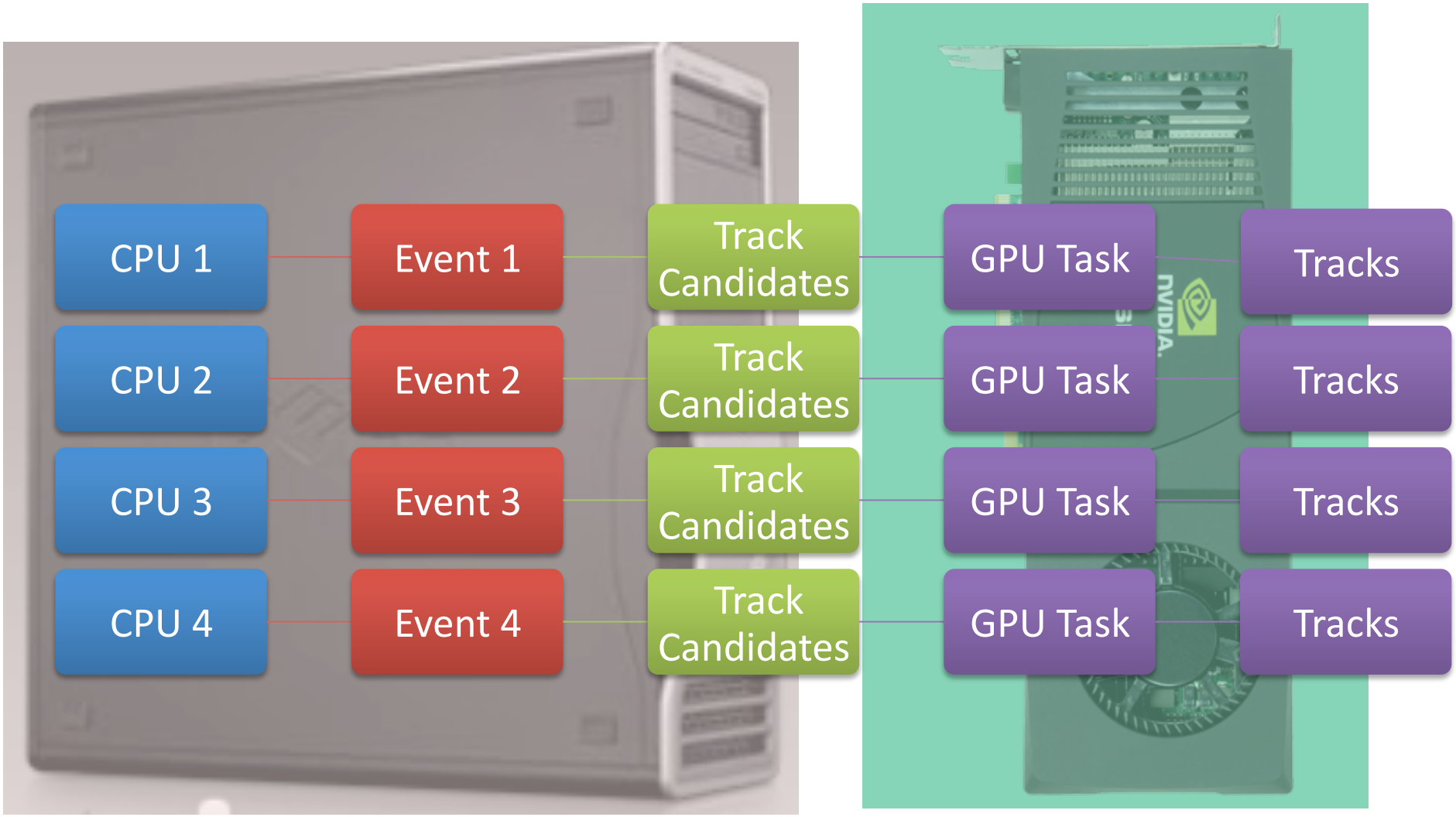
How to optimize the usage of the Tesla card in this example?

- Problem:
  - Only 25% of the hardware is used!
  - The limitation we have in this example is the number of points in the track candidate (average =32 points). So starting more threads and/or less blocks will not help!

This is for current generation GPU. Fermi could well be different. In Fermi each Warp (32threads) can execute different instruction. (NK)

- Solution:
  - Tesla support concurrent access which means that different CPU threads (Or Processes) can start different kernels on the device.

Parallelization on CPU/GPU



Results

Track/Event Process	50 (Float)	2000 (Float)	2000 (Double)
1	1.0 ms	3.2 ms	5.0 ms
4	1.7 ms/Process	3.3 ms /Process	6.3 ms/Process

No. of Process	Track/Event	50 (Float)	2000 (Float)	2000 (Double)
1 CPU		1.7 E4 Track/s	9.1 E2 Track/s	9.1 E2 Track/s
1 CPU + GPU (Tesla)		5.0 E4 Track/s	6.3 E5 Track/s	4.0 E5 Track/s
4 CPU + GPU (Tesla)		1.2 E5 Track/s	2.2 E6 Track/s	1.3 E6 Track/s