Applying CUDA computing model to event reconstruction software

MOHAMMAD AL-TURANY

GSI Darmstadt

Outline

- Few words about GPU vs CPU
- CUDA vs GPGPU
- Why CUDA?
- Example 1: Runge-Kutta Track propagation
  (use of texture memory)
- Example 2: Track and vertex fitting
  (use of pinned memory)
- Summary

CPU vs. GPU

CPU	GPU
• CPU is designed to execute one stream of instructions as fast as possible.	• GPU is designed to execute many parallel streams 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 GPU spends transistors in processor arrays, multithreading hardware, shared memory, and multiple memory controllers.

CPU vs. GPU

CPU	GPU
• The CPU uses cache to improve performance by reducing the latency of memory accesses.	• The GPU uses cache (or software-managed shared memory) to amplify bandwidth.
• CPUs support one or two threads per core.	• CUDA capable GPUs support up to 1024 threads per streaming multiprocessor.

CPU vs. GPU

CPU	GPU
• 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.	• 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.
• The cost of a CPU thread switch is hundreds of cycles.	• GPUs have no cost in switching threads. GPUs typically switch threads every clock.

CPU vs. GPU

CPU	GPU
• CPUs use SIMD (single instruction, multiple data) units for vector processing.	• 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 vs GPGPU

CUDA

- work with familiar programming concepts (C language) while developing software that can run on a GPU
- CUDA compile the code directly to the hardware (GPU assembly language, for instance), thereby providing great performance.

GPGPU

Trick the GPU into general-purpose computing by casting problem as graphics

- Turn data into images ("texture maps")
- Turn algorithms into image synthesis ("rending passes")

Drawback:

- Tough learning curve
- potentially high overhead of graphics API
- highly constrained memory layout & access model

CUDA: Features

- Standard C language for parallel application development on the GPU
- Standard numerical libraries for FFT (Fast Fourier Transform) and BLAS (Basic Linear Algebra Subroutines)
- Dedicated CUDA driver for computing with fast data transfer path between GPU and CPU

Why CUDA?

- CUDA development tools work alongside the conventional C/C ++ compiler, so one can mix GPU code with general-purpose code for the host CPU.
- CUDA Automatically Manages Threads:
  - It does NOT require explicit management for threads in the conventional sense, which greatly simplifies the programming model.
- Stable, available (for free), documented and supported for windows, Linux and Mac OS
- Low learning curve:
  - Just a few extensions to C
  - No knowledge of graphics is required

Cuda (2.3) Toolkit

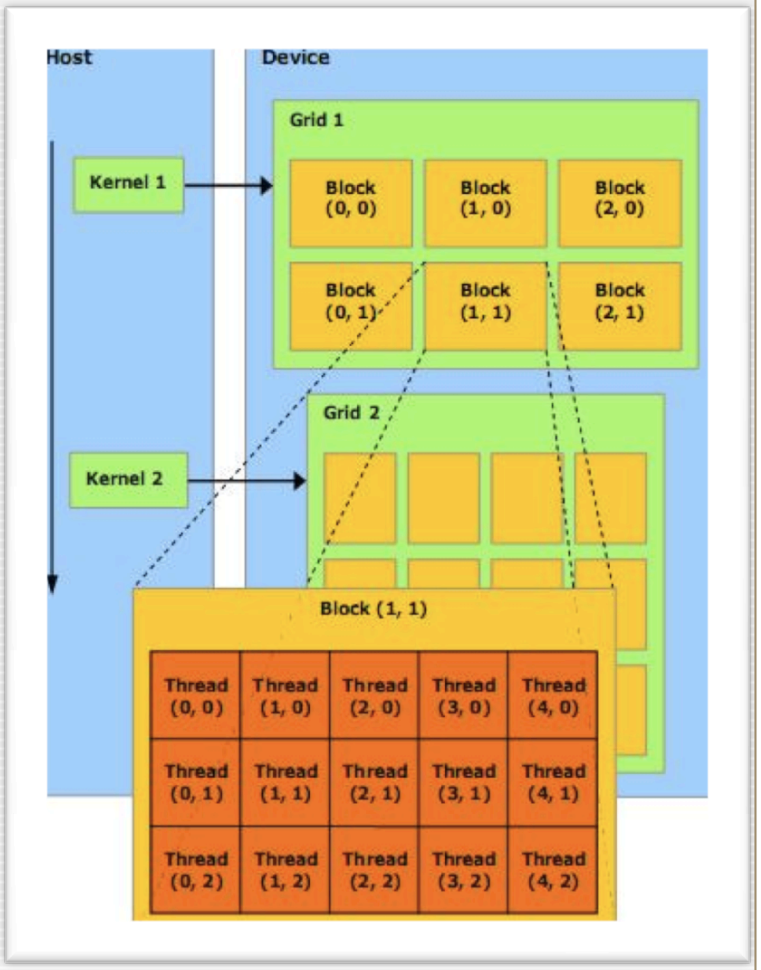
- NVCC C compiler
- CUDA FFT and BLAS libraries for the GPU
- CUDA-gdb hardware debugger
- CUDA Visual Profiler
- CUDA runtime driver (also available in the standard NVIDIA GPU driver)
- CUDA programming manual

CUDA in FairRoot

- FindCuda.cmake (Abe Stephens SCI Institute)
  - Integrate CUDA into FairRoot very smoothly
- CMake create shared libraries for cuda part
- FairCuda is a class which wraps CUDA implemented functions so that they can be used directly from ROOT CINT or compiled code

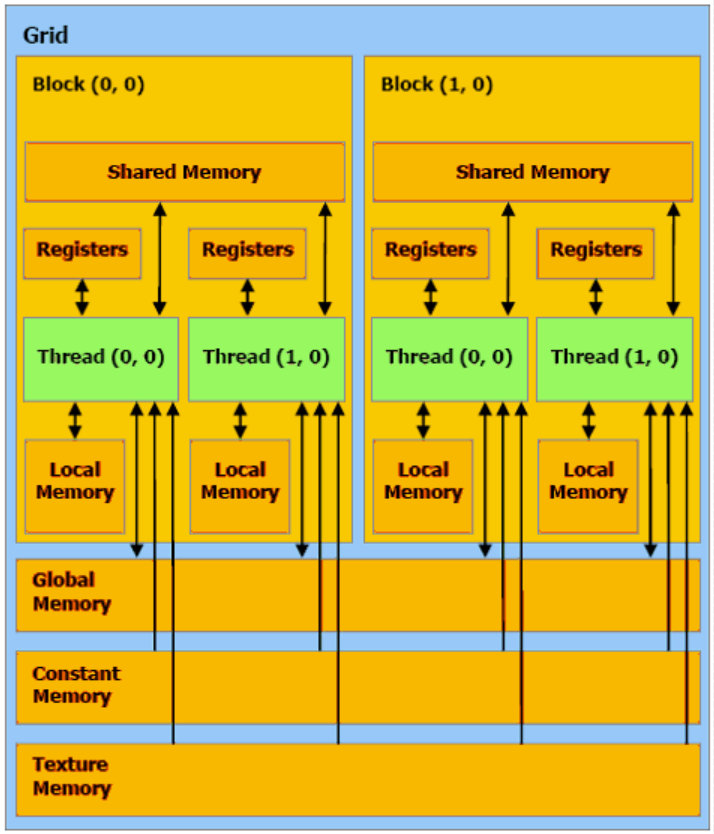
CUDA programming model

- Kernel:
  - One kernel is executed at a time
  - Kernel launches a grid of thread blocks
- Thread block:
  - A batch of thread.
  - Threads in a block cooperate together, efficiently share data.
  - Thread/block have unique id
- Grid:
  - A batch of thread blocks that execute the same kernel.
  - Threads in different blocks in the same grid cannot directly communicate with each other



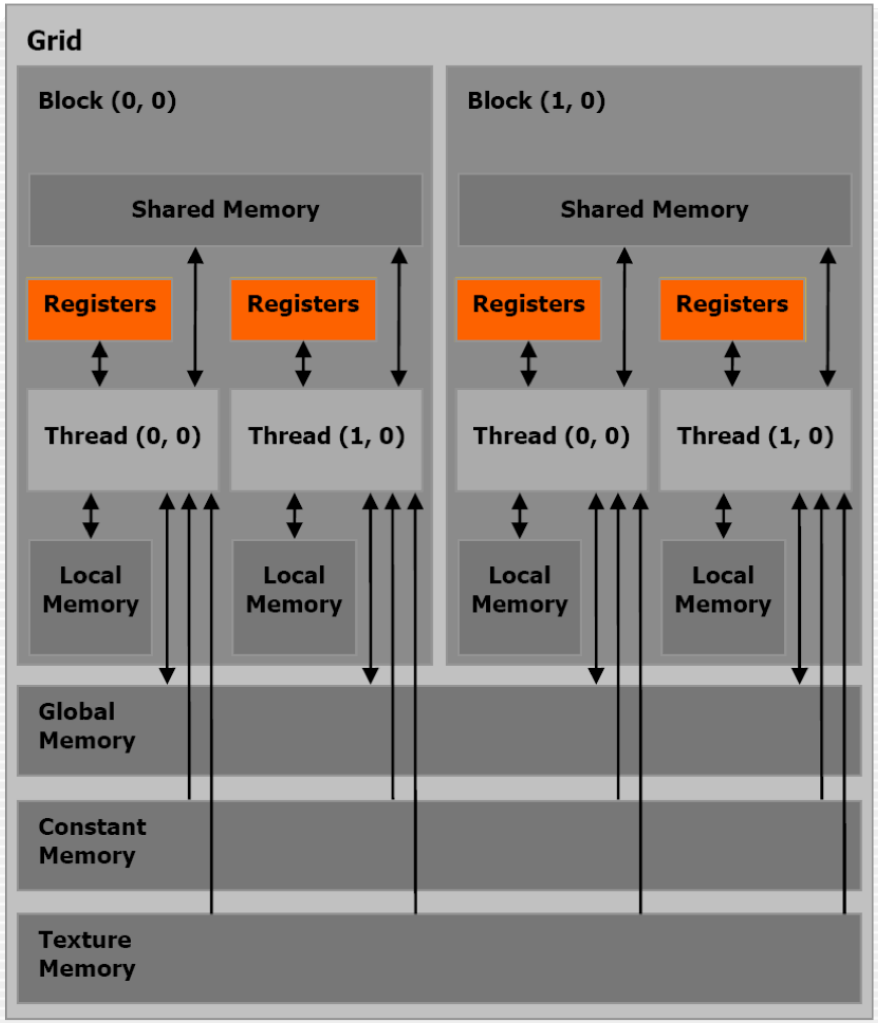
CUDA memory model

- One set of local registers per thread.
- A parallel data cache or shared memory that is shared by all the threads and implements the shared memory space.
- A read-only constant cache that is shared by all the threads and speeds up reads from the constant memory space
- A read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory



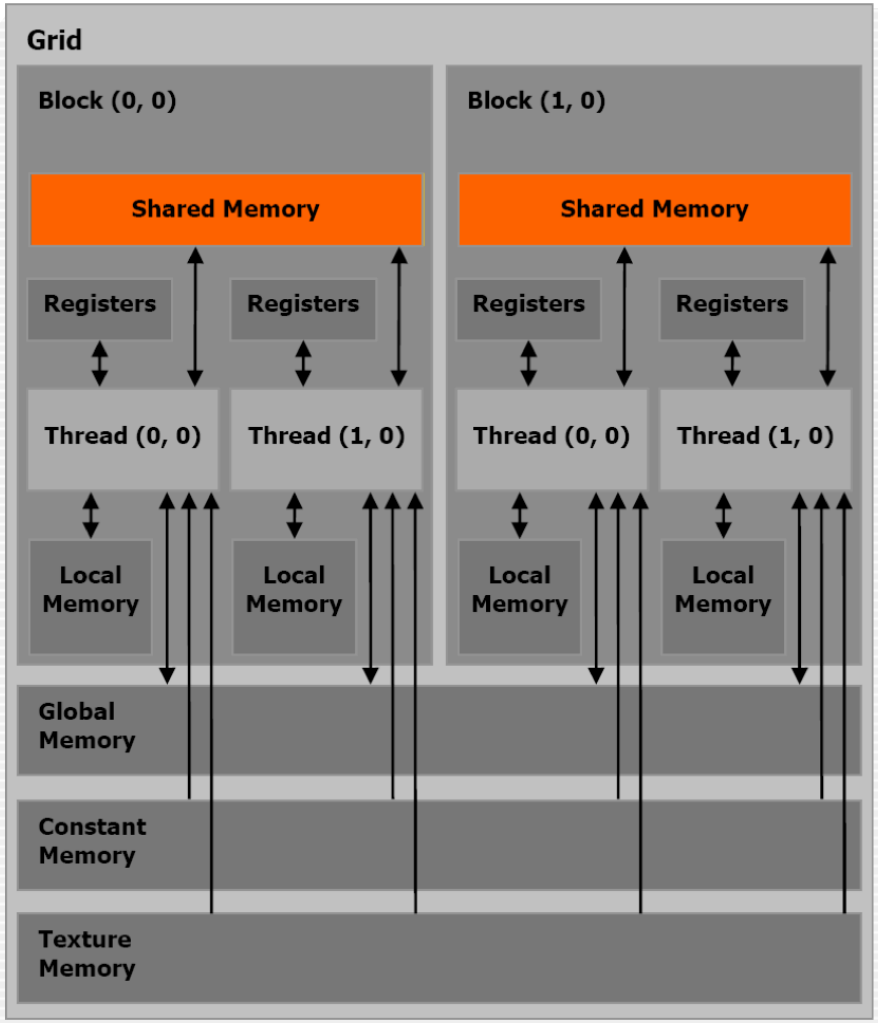
Register Memory

- The fastest form of memory on the multi-processor.
- Is only accessible by the thread.
- Has the lifetime of the thread



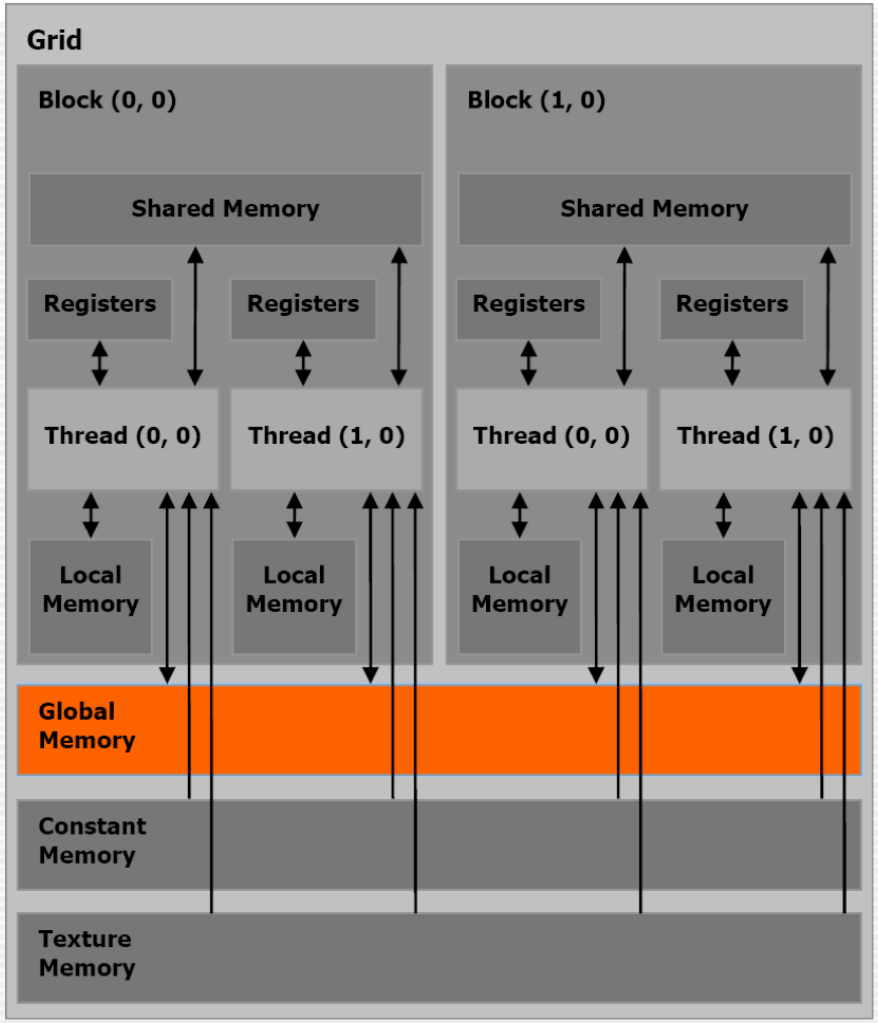
Shared Memory

- Can be as fast as a register when there are no bank conflicts or when reading from the same address.
- Accessible by any thread of the block from which it was created.
- Has the lifetime of the block.



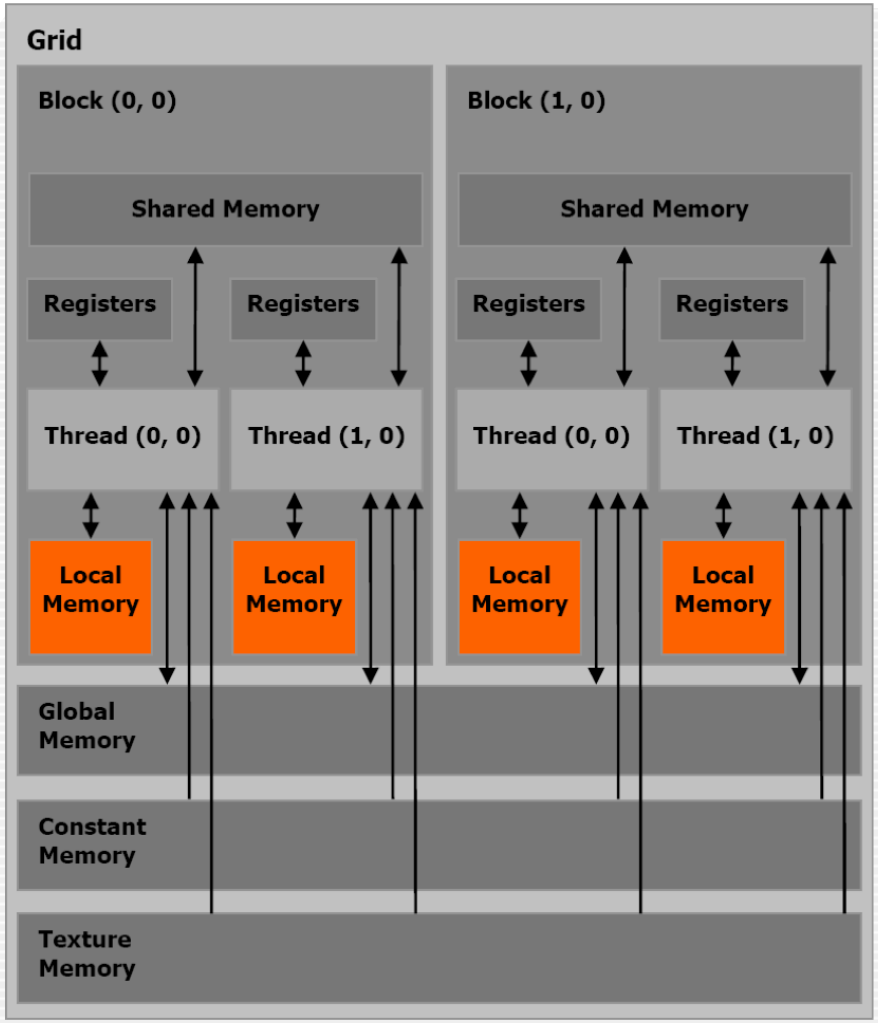
Global Memory

- Potentially 150x slower than register or shared memory .
- Accessible from either the host or device.
- Has the lifetime of the application.



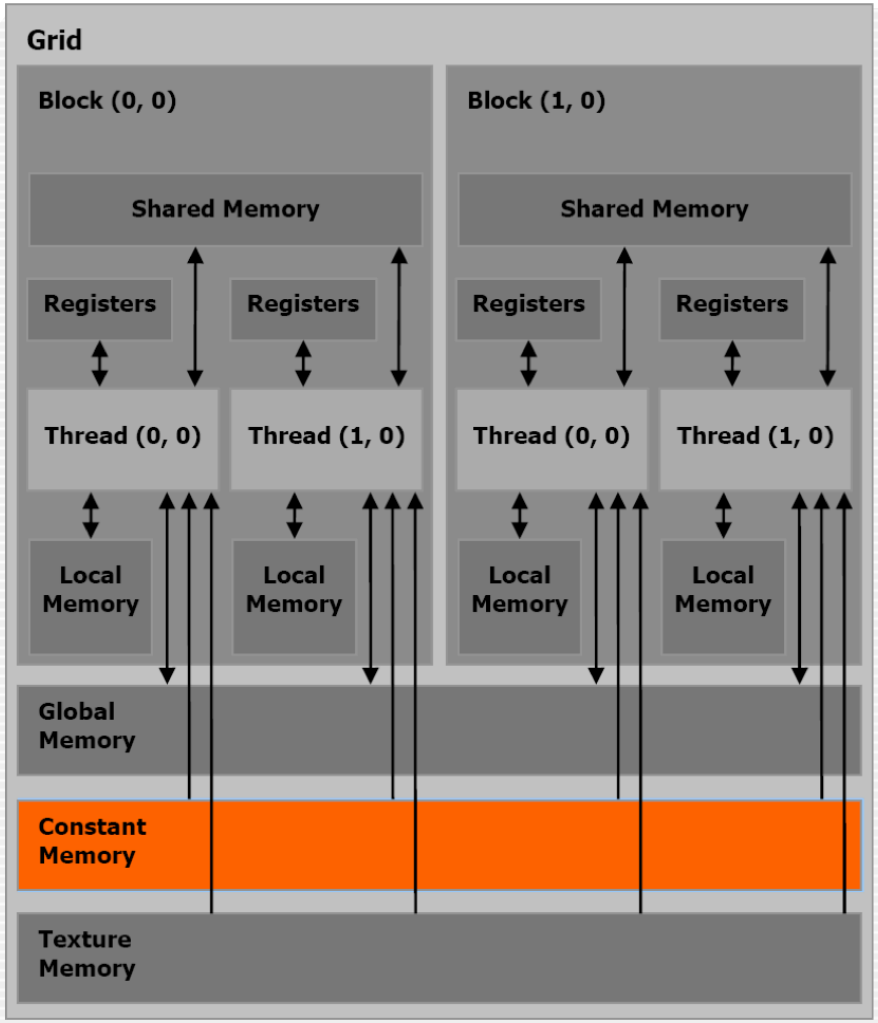
Local Memory

- Resides in global memory and can be 150x slower than register or shared memory
- Is only accessible by the thread
- Has the lifetime of the thread.



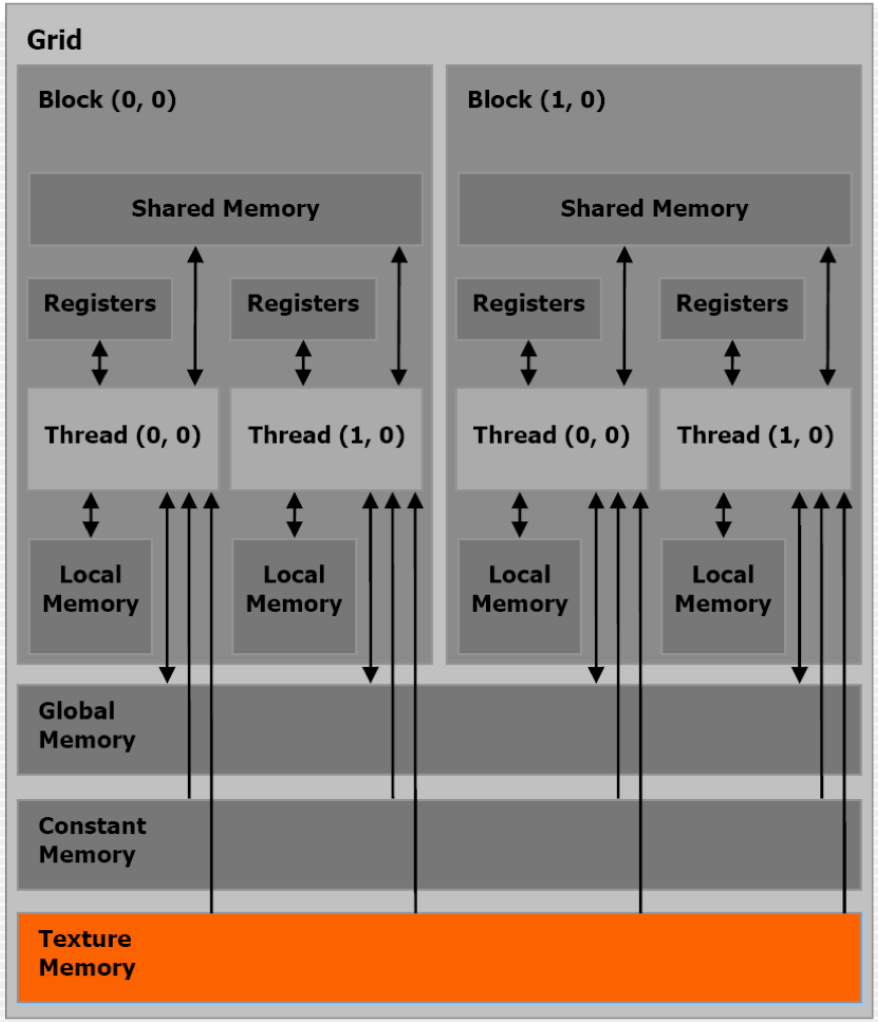
Constant Memory

- in DRAM
- cached
- per grid
- read-only



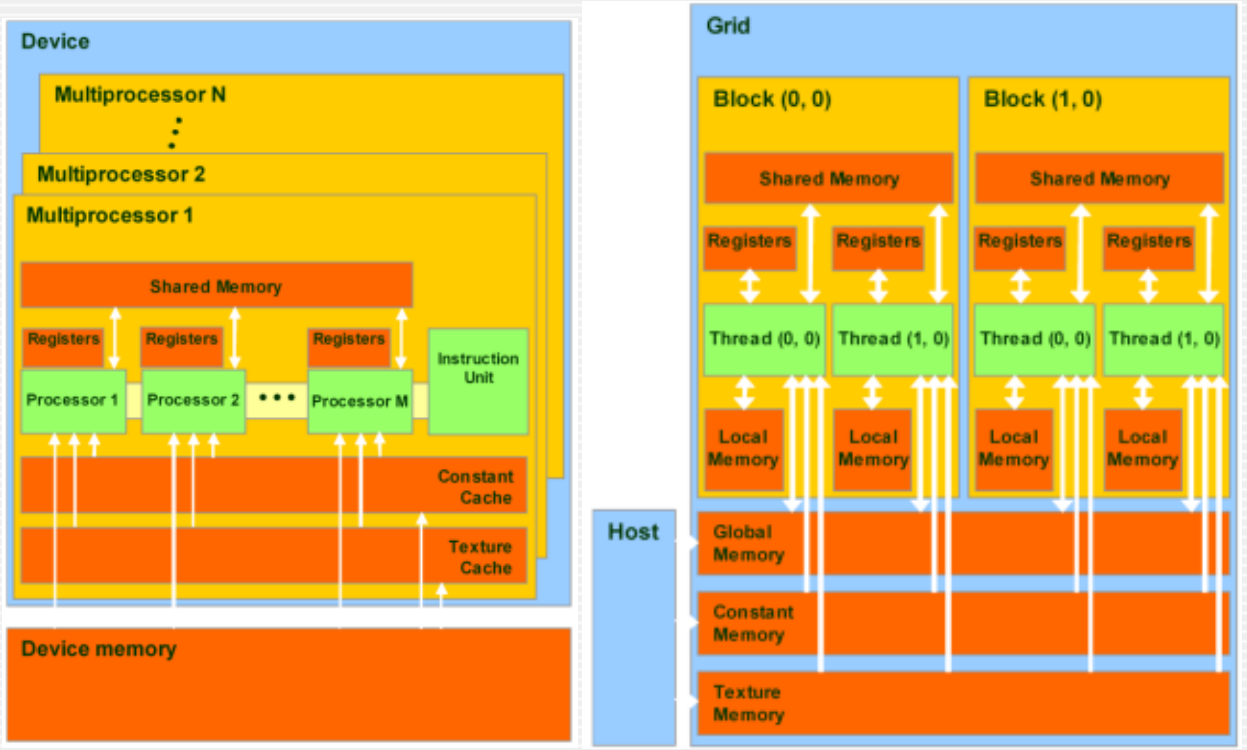
Texture Memory

- in DRAM
- cached
- per grid
- read-only



Global, local, texture, and constant memory are physically the same memory.

They differ only in caching algorithms and access models.



CPU can refresh and access only: global, constant, and texture memory.

Example (Texture Memory)

USING TEXTURE MEMORY FOR FIELD MAPS

Field Maps

- Usually a three dimensional array (XYZ, Rθϕ, etc)
- Used as a lockup table with some interpolation
- For performance and multi-access issues, many people try to parameterize it.

Texture Memory for field maps

- Three dimensional arrays can be bind to texture directly
- Accessible from all threads in a grid
- Linear interpolation is done by dedicated hardware
- Cashed and allow multiple random access

Using Texture Memory

- Host (CPU) code:
  - Allocate/obtain memory (global linear/pitch linear, or CUDA array)
  - Create a texture reference object (Currently must be at file-scope)
  - Bind the texture reference to memory/array
  - When done: Unbind the texture reference, free resources
- Device (kernel) code:
  - Fetch using texture reference
    - Linear memory textures: tex1Dfetch()
    - Array textures: tex1D() or tex2D() or tex3D()
    - Pitch linear textures: tex2D()

Texture Filtering

- CudaFilterModePoint: The returned value is the texel (Texture Element) whose texture coordinates are the closest to the input texture coordinates;

- CudaFilterModeLinear: The returned value is the linear interpolation of the two (for a one-dimensional texture), four (for a two-dimensional texture), or eight (for a three-dimensional texture) texels whose texture coordinates are the closest to the input texture coordinates

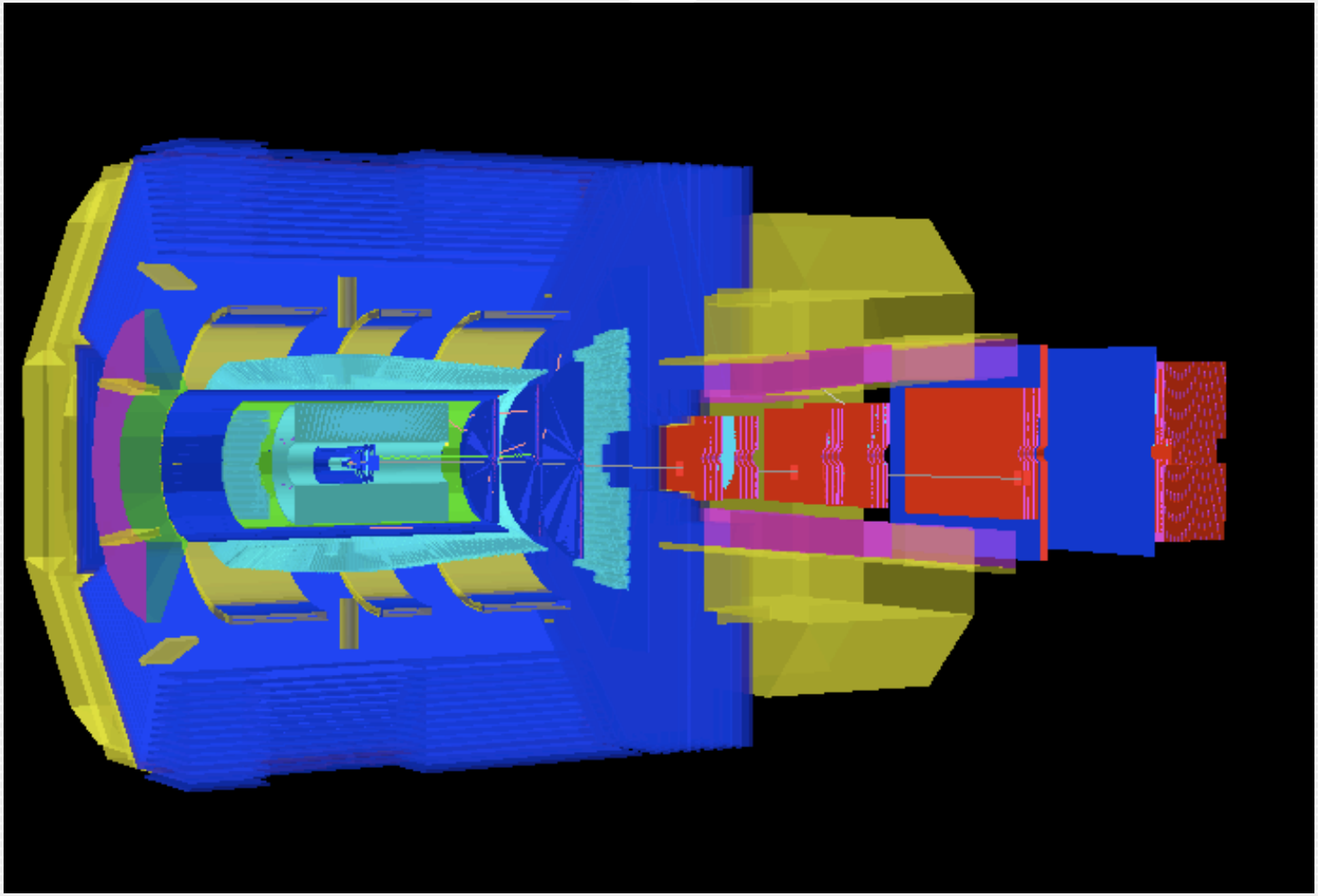
Texture Address Mode

- How out-of-range texture coordinates are handled;
  - Clamp: Out-of-range texture coordinates are clamped to the valid range. (Values below 0 are set to 0 and values greater or equal to N are set to N-1)
  - Wrap: Out-of- range texture coordinates are wrapped to the valid range (only for normalized coordinates). Wrap addressing is usually used when the texture contains a periodic signal. It uses only the fractional part of the texture coordinate; for example, 1.25 is treated the same as 0.25 and -1.25 is treated the same as 0.75

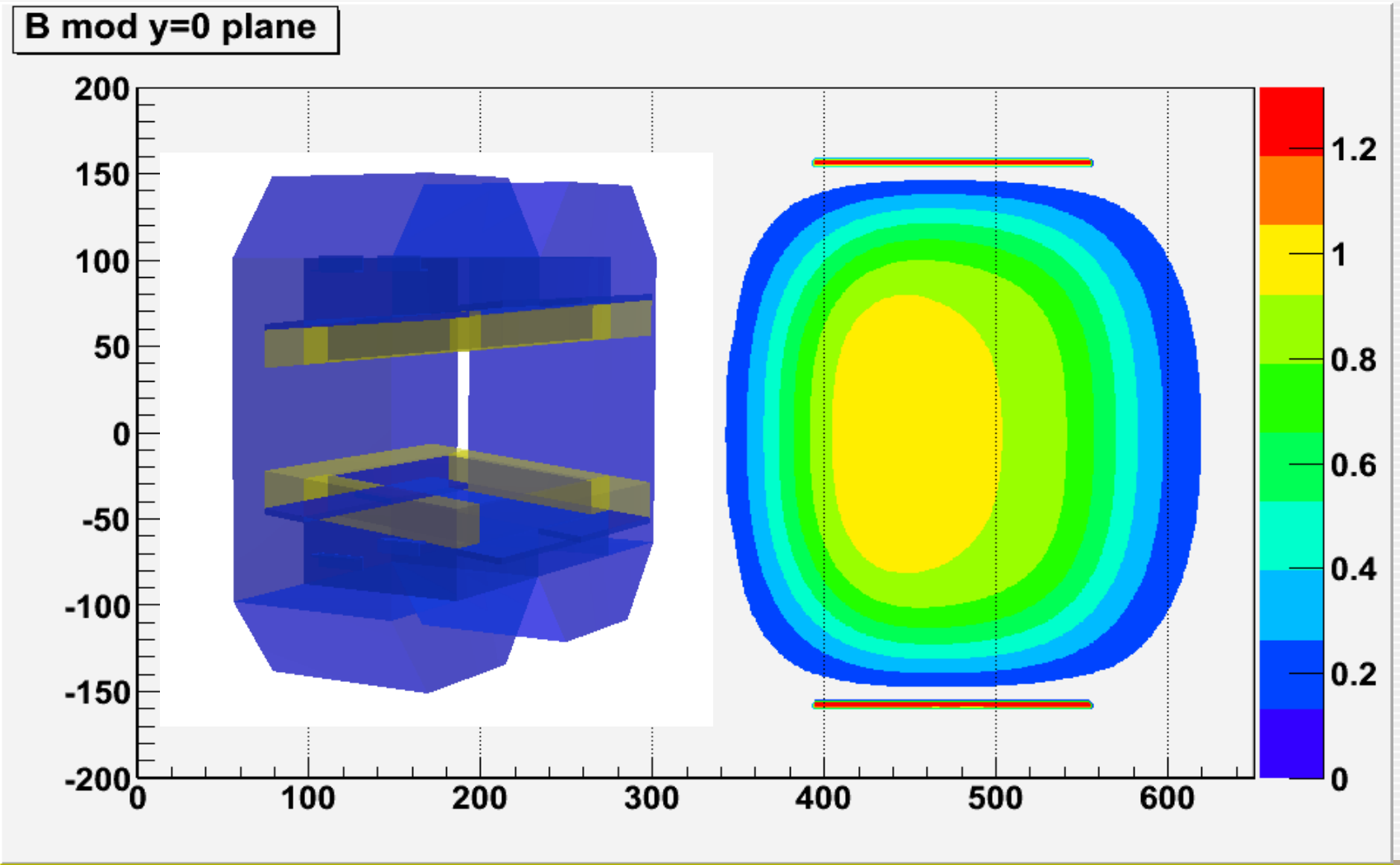
Runge-Kutta propagator

- The Geant3 Runge-Kutta propagator was re-written inside a cuda kernel
  - Runge-Kutta method for tracking a particle through a magnetic field. Uses Nystroem algorithm (See Handbook Nat. Bur. Of Standards, procedure 25.5.20)
- The algorithm it self is hardly parallelizable, but one can propagate all tracks in an event in parallel
- For each track, a block of 8 threads is created, the particle data is copied by all threads at once, then one thread do the propagation

Panda Detector



Magnet and Field



Field Map

Field map grid : Bx, By, Bz

- x = 0.0 to 158 cm, 80 points, Δx = 2.0 cm
- y = 0.0 to 51 cm, 52 points, Δy = 1.0 cm
- z = 342.0 to 602 cm, 131 points, Δz = 2.0 cm

4-fold symmetry

Cards used in this Test

	Qaudro NVS 290	GeForce 8400 GT	GeForce 8800 GT	Tesla C1060
CUDA cores	16 (2 x 8)	32 (4 x 8)	112 (14 x 8)	240 (30 x 8)
Memory (MB)	256	128	512	4000
Frequency of processor cores (GHz)	0.92	0.94	1.5	1.3
Compute capability	1.1	1.1	1.1	1.3
Warps/Multiprocessor	24	24	24	32
Max. No. of threads	1536	3072	10752	30720
Max Power Consumption (W)	21	71	105	200

Features available only in 1.3 computing capabilities

- Support for atomic functions operating in shared memory and on 64-bit words in global memory (for 1.1 only 32-bit words)
- Support for warp vote functions
- The number of registers per multiprocessor is 16384 (8192 in 1.1)
- The maximum number of active warps per multiprocessor is 32 (24 in 1.1)
- The maximum number of active threads per multiprocessor is 1024 (768 in 1.1)
- Support for double-precision floating-point numbers

Track Propagation (time per event)

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

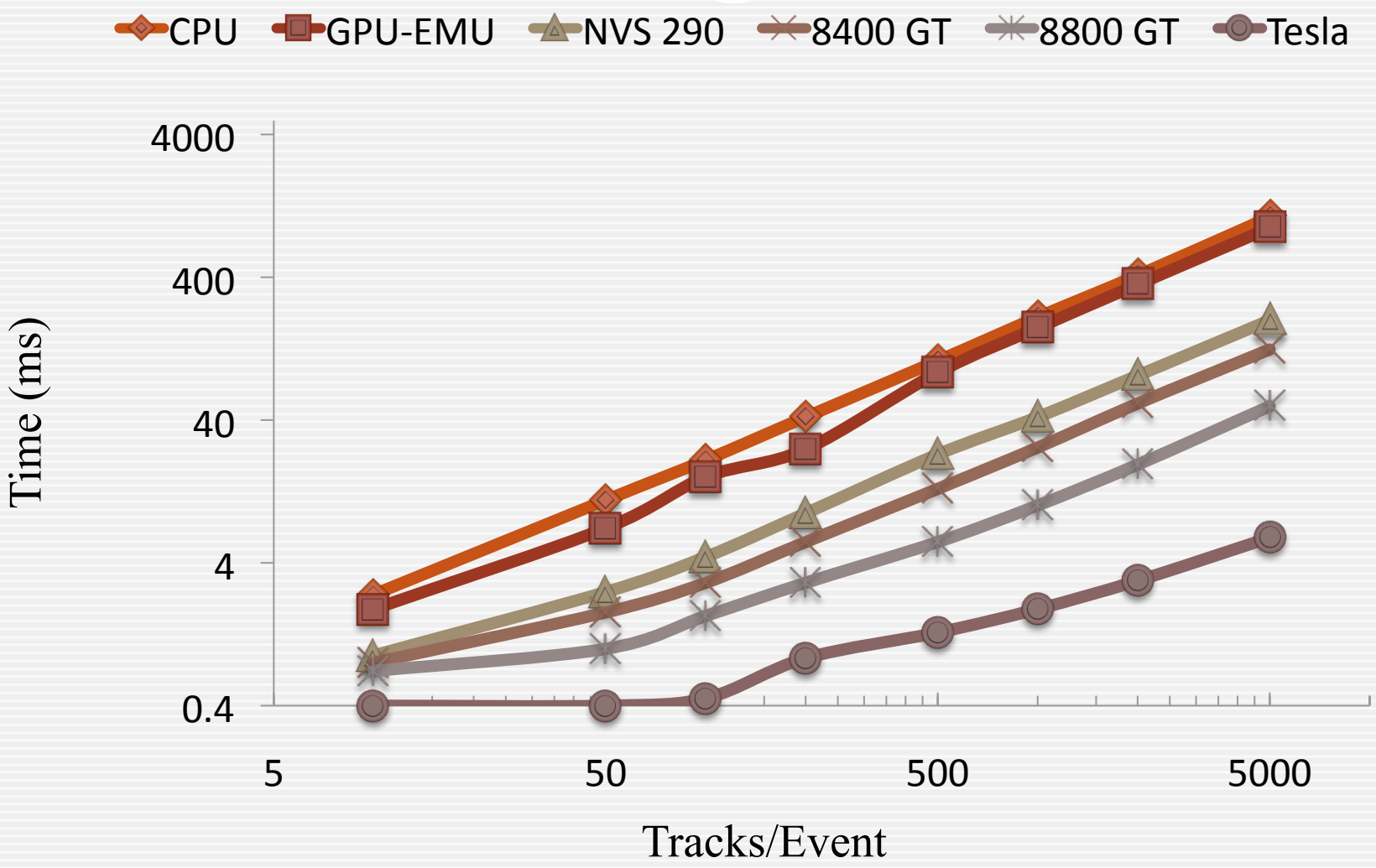
Time in ms needed to propagate all tracks in event

Track Propagation (time per track)

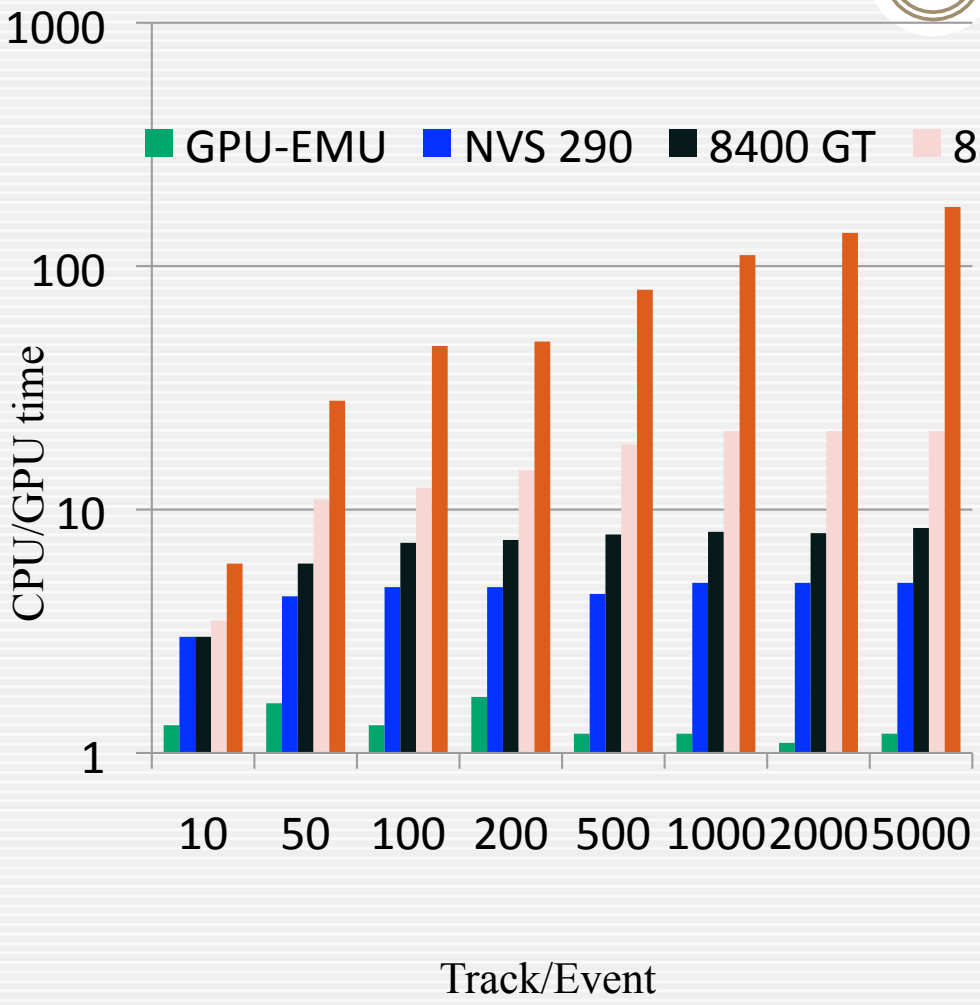
Trk/ Event	CPU	GPU emu	Quadro NVS 290 (16)	GeForce 8400GT (32)	GeForce 8800 GT (112)	Tesla C1060 (240)
10	240	190	90	80	70	40
50	220	140	50	36	20	8
100	210	160	44	29	17	5
200	210	125	45	28	15	4.3
500	208	172	46	26	11	2.6
1000	210	177	42	26	10	1.9
2000	206	178	41	26	10	1.5
5000	211	177	40	25	10	1.2

Time in μs needed to propagate one track 1.5 m in a dipole field

Time needed to analyze one event in ms



Gain for different cards



Trk/ Event	GPU emu	NVS 290	8400 GT	8800 GT	Tesla
10	1.30	3	3	3.5	6
50	1.60	4.4	6	11	28
100	1.30	4.8	7.3	12.3	47
200	1.70	4.8	7.5	14.5	49
500	1.20	4.5	7.9	18.5	80
1000	1.20	5	8.1	21	111
2000	1.10	5	8	21	137
5000	1.20	5	8.4	21	175

Emulation Mode

When running an application in device emulation mode, the programming model is emulated by the runtime. For each thread in a thread block, the runtime creates a thread on the host. The programmer needs to make sure that:

- The host is able to run up to the maximum number of threads per block, plus one for the master thread.
- Enough memory is available to run all threads, knowing that each thread gets 256 KB of stack.

Emulation Mode

- In this example we have 8 threads per block
  - Data is copied from global (or Host) Memory by 8 threads
  - One thread perform the propagation
- On 4 core machine the system can start 9 threads
- In the CPU native code each time one get the field value we have to check for the boundary, but the GPU code do not need this check
  - This explain the speed up in emulation mode against the native CPU code.

Resource usage in this Test

	Qaudro NVS 290	GeForce 8400 GT	GeForce 8800 GT	Tesla C1060
Warps/Multiprocessor	24	24	24	32
Occupancy	33%	33%	33%	25%
Active Threads	128	256	896	1920
Limited by Max Warps / Multiprocessor	8	8	8	8

Active threads = Warps x 32 x multiprocessor x occupancy

Active threads in Tesla = 8x32x30x0.25 = 1920

Example (Zero Copy)

USING THE PINNED (PAGED-LOCKED) MEMORY TO MAKE THE DATA AVAILABLE TO THE GPU

Zero Copy

- Zero copy was introduced in CUDA Toolkit 2.2
- It enables GPU threads to directly access host memory, and it requires mapped pinned (non-pageable) memory
- Zero copy can be used in place of streams because kernel-originated data transfers automatically overlap kernel execution without the overhead of setting up and determining the optimal number of streams

Pinned Memory

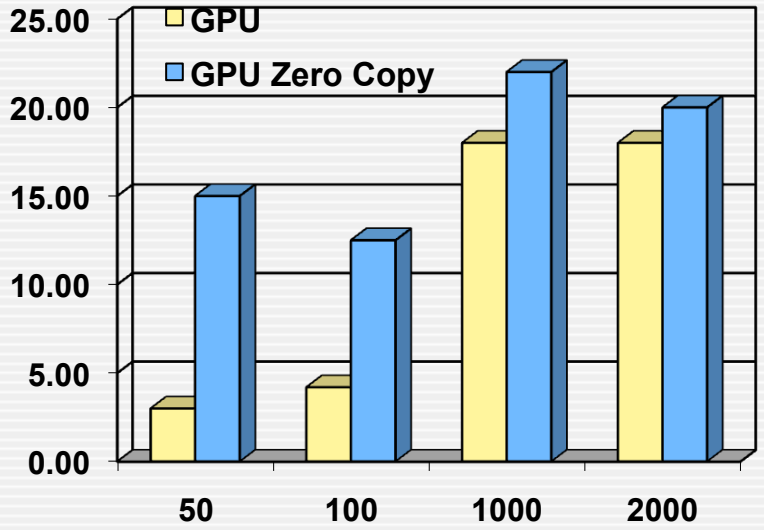
- On discrete GPUs, mapped pinned memory is advantageous only in certain cases. Because the data is not cached on the GPU, mapped pinned memory should be read or written only once, and the global loads and stores that read and write the memory should be coalesced.

- On integrated GPUs, mapped pinned memory is always a performance gain because it avoids superfluous copies as integrated GPU and CPU memory are physically the same.

Track + vertex fitting on CPU and GPU

CPU Time/GPU Time

Track/Event	50	100	1000	2000
GPU	3.0	4.2	18	18
GPU (Zero Copy)	15	13	22	20

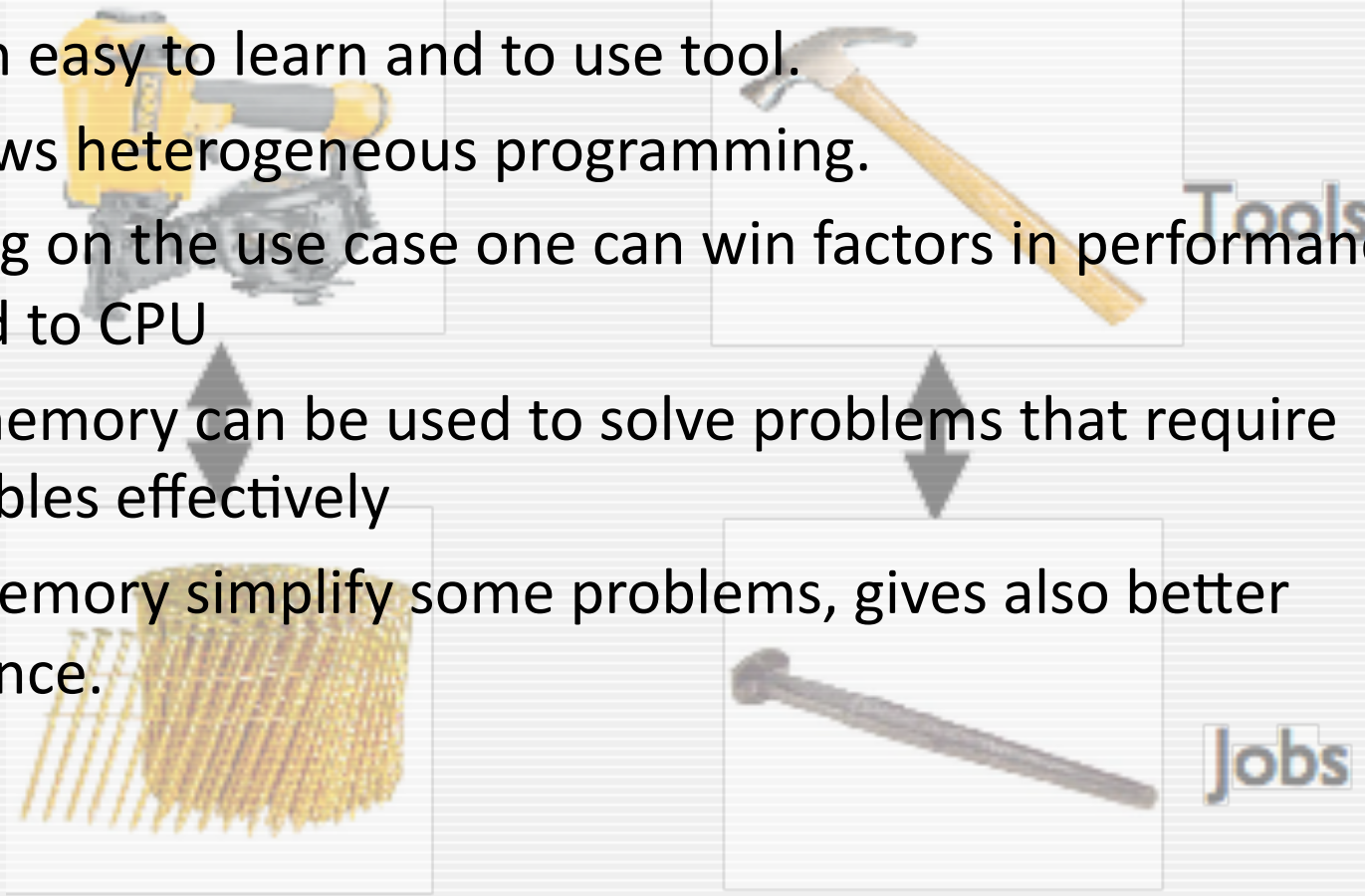


Time needed per event (ms)

	50	100	1000	2000
CPU	3.0	5.0	120	220
GPU	1.0	1.2	6.5	12.5
GPU (Zero Copy)	0.2	0.4	5.4	10.5

Summary

- Cuda is an easy to learn and to use tool.
- Cuda allows heterogeneous programming.
- Depending on the use case one can win factors in performance compared to CPU
- Texture memory can be used to solve problems that require lookup tables effectively
- Pinned Memory simplify some problems, gives also better performance.



Backup Slides

CPU vs GPU code


```
Double_t h2, h4, f[4];
Double_t xyzt[3], a, b, c, ph,ph2;
Double_t secxs[4],secys[4],seczs[4],hxp[3];
Double_t g1, g2, g3, g4, g5, g6, ang2, dxt,
   dyt, dzt;
Double_t est, at, bt, ct, cba;
Double_t f1, f2, f3, f4, rho, tet, hnorm, hp,
   rho1, sint, cost;
Double_t x;
Double_t y;
Double_t z;
Double_t xt;
Double_t yt;
Double_t zt;
Double_t maxit = 10;
Double_t maxcut = 11;
const Double_t hmin   = 1e-4;
const Double_t kdlt   = 1e-3;
const Double_t kdlt32 = kdlt/32.;
const Double_t kthird = 1./3.;
……
```

```
__shared__ float4 field;
float h2, h4, f[4];
float xyzt[3], a, b, c, ph,ph2;
float secxs[4],secys[4],seczs[4],hxp[3];
float g1, g2, g3, g4, g5, g6, ang2, dxt,
   dyt, dzt;
float est, at, bt, ct, cba;
float f1, f2, f3, f4, rho, tet, hnorm, hp,
   rho1, sint, cost;
float x;
float y;
float z;
float xt;
float yt;
float zt;
float maxit= 10;
float maxcut= 11;
float hmin = 1e-4;
float kdlt = 1e-3;
float kdlt32 = kdlt/32.;
float kthird = 1./3.;
 ….
```

CPU vs GPU code


```
 do {
rest  = step - tl;
if (TMath::Abs(h) > TMath::Abs(rest))
                        h = rest;
fMagField->GetFieldValue( vout, f);
    f[0] = -1.0*f[0];
    f[1] = -1.0*f[1];
    f[2] = -1.0*f[2];
……….
if (step < 0.) rest = -rest;
if (rest < 1.e-5*TMath::Abs(step)) return;
} while(1);
```

```
 do {
rest  = step - tl;
if (fabs(h) > fabs(rest))
                        h = rest;
field=GetField(vout[0],vout[1],vout[2]);
    f[0] = -field.x;
    f[1] = -field.y;
    f[2] = -field.z;
……….
 if (step < 0.) rest = -rest;
 if (rest < 1.e-5*fabs(step)) return;
} while(1);
```