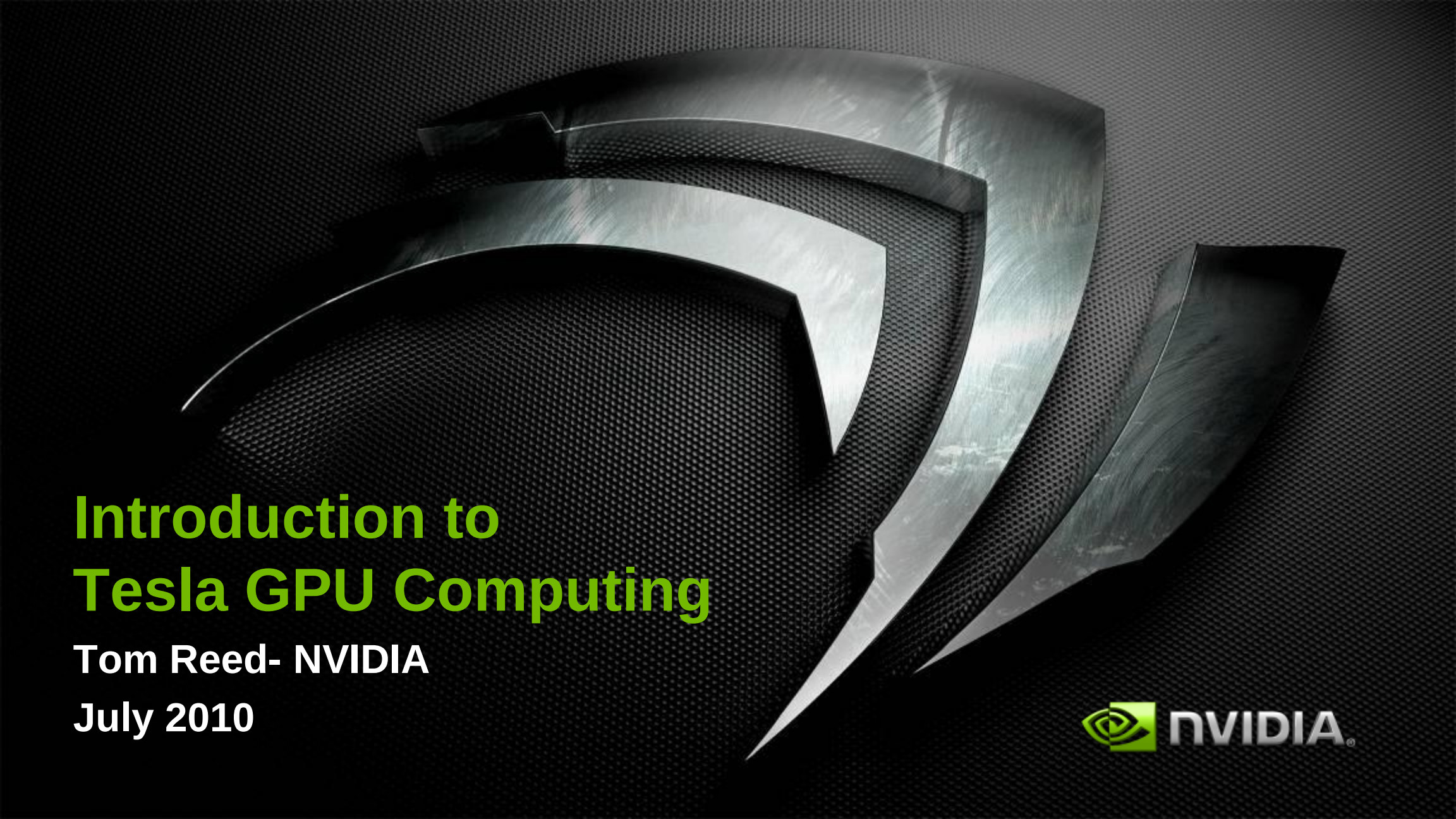
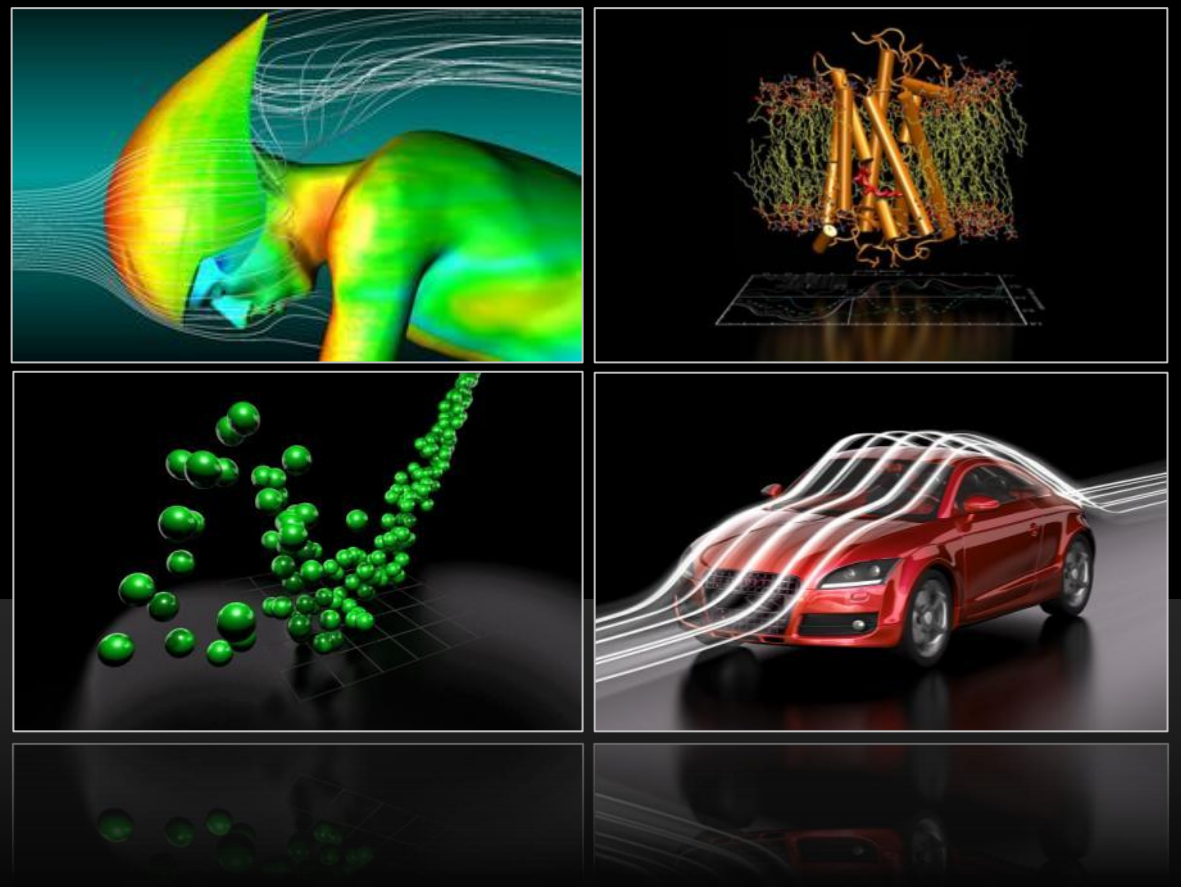

Introduction to
Tesla GPU Computing
Tom Reed- NVIDIA
July 2010
NVIDIA

Tesla GPU Computing



Two Markets on a Common Trajectory

Visualization Challenges:

- **SD -> HD -> HD Stereo -> 4K 3D**
- **Gaming to Visual Realism (Ray Tracing)**
- **Fusing video & geometry**

Traditional GPU
+
Computational Acceleration

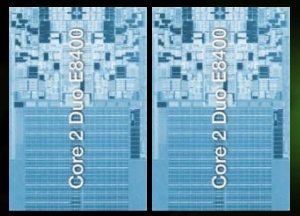
HPC Challenges:

- **Overcoming Peta-scale complexity (node count, power, floor space, uptime)**
- **Explosive Data growth and complexity**
- **Scaling analysis and interpretation tools**

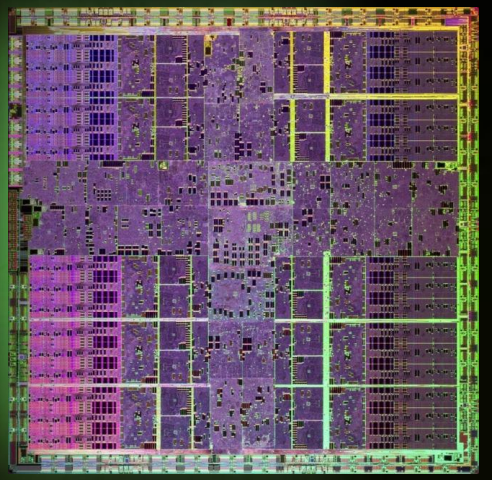
Traditional CPU
+
Computational Acceleration

The GPU and the CPU

“Heterogeneous CPU+GPU”



+



Multi-Core CPU

Parallel-Core GPU

Why Use GPU Computing Processors?

Imagine what can be achieved if …

automated traders can take advantage of arbitrages quicker than their competition.

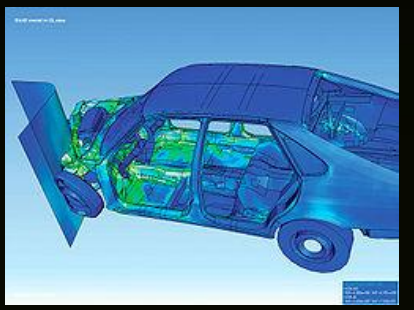
⟹ Increase time to revenues

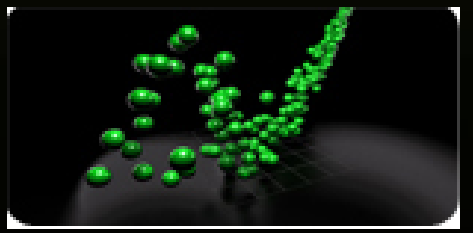
engineers are empowered with a supercomputer at their desktop.

⟹ Accelerate time to results

researchers unlock the mysteries of cancer or other diseases.

⟹ Achieve the impossible

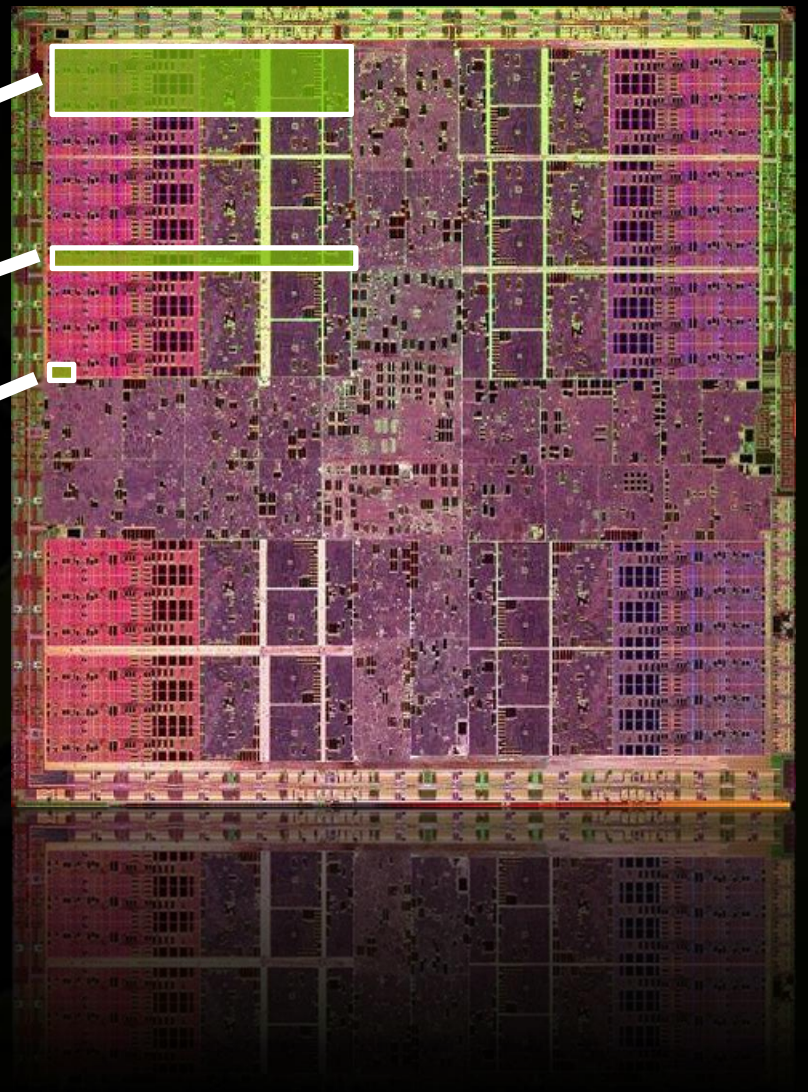




Advantages of GPUs

- Designed for numerical applications not operating systems
- Designed for thousands of threads of execution
- Orders of magnitude more floating point units (single *and* double precision)
- Ideally suited for Imaging or Signal processing work
- Low entry price + free development environment = over 180 million potential GPU computing devices

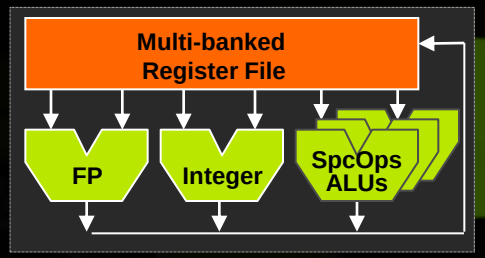
Tesla T10: 1.4 Billion Transistors



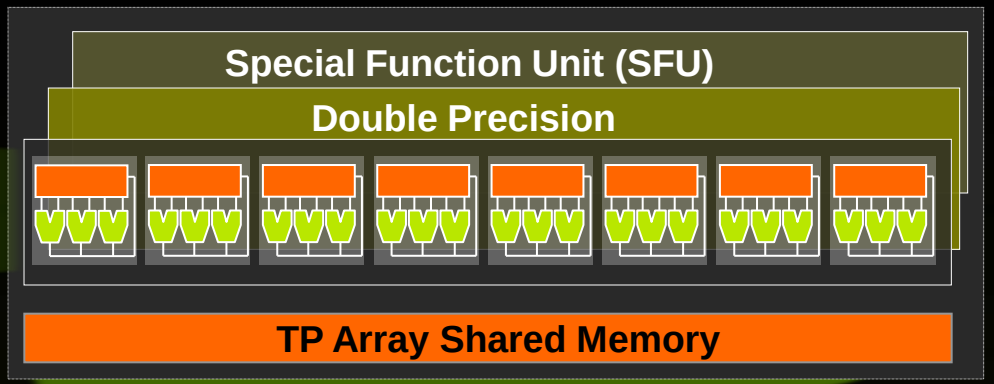
Die Picture of Tesla T10

Tesla T10: The Processor Inside

Thread Processor (TP)



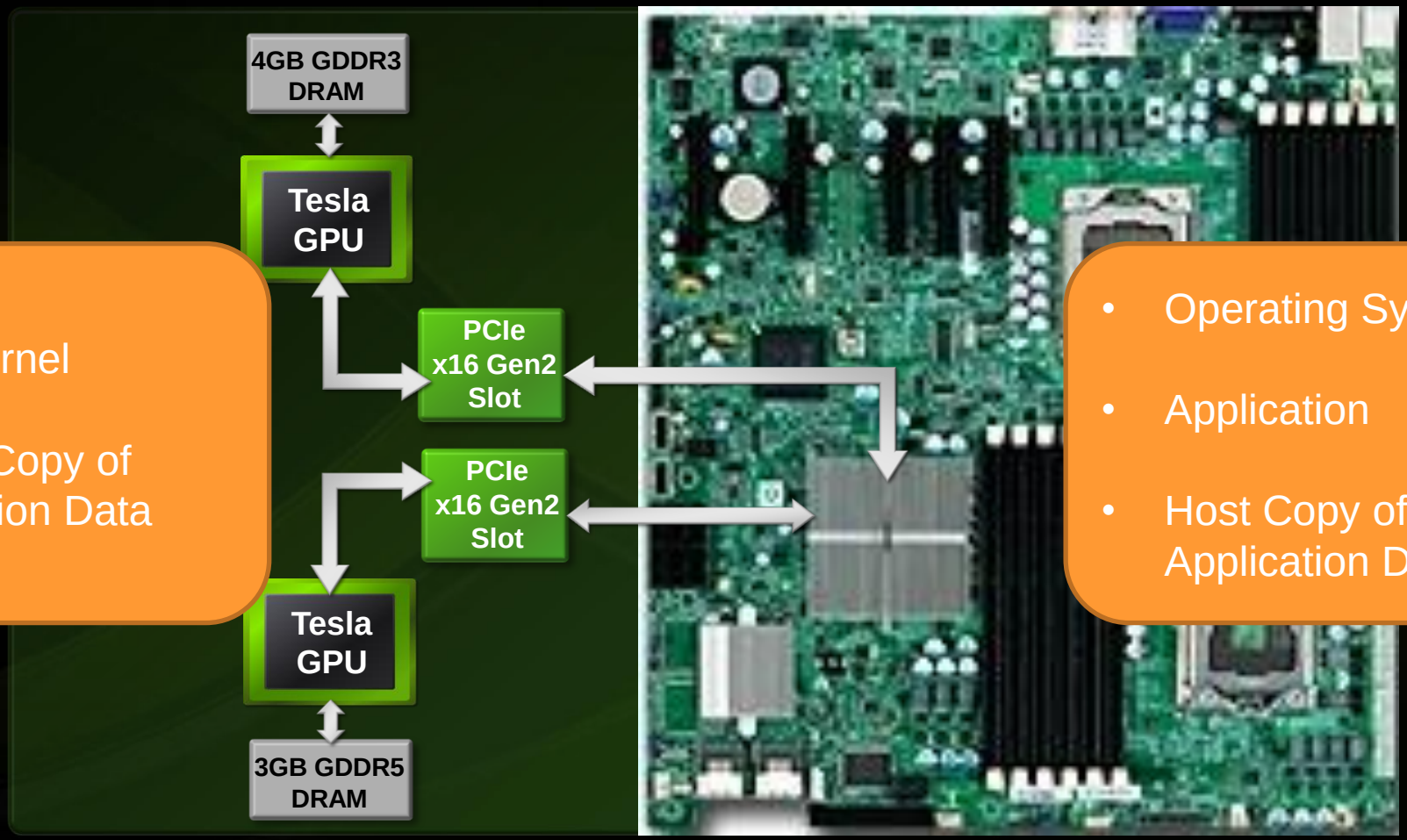
Thread Processor Array (TPA)



- 8 TP per TPA (240 Total)
- Full scalar processor with integer and floating point units
- 16K of RAM for Shared Memory

30 TPAs = 240 Processors

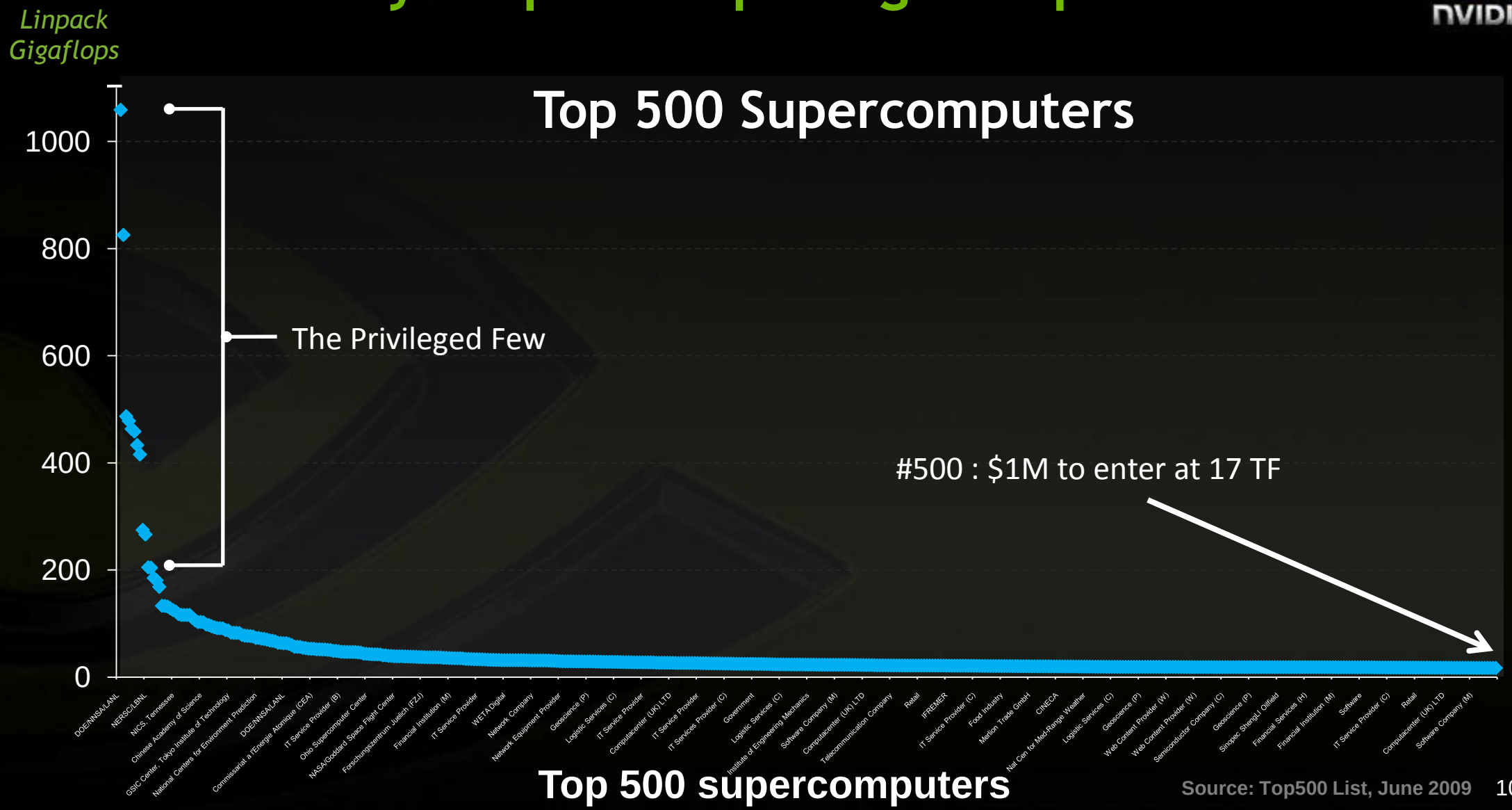
Connecting GPUs to System Architecture



- GPU Kernel
- Device Copy of Application Data

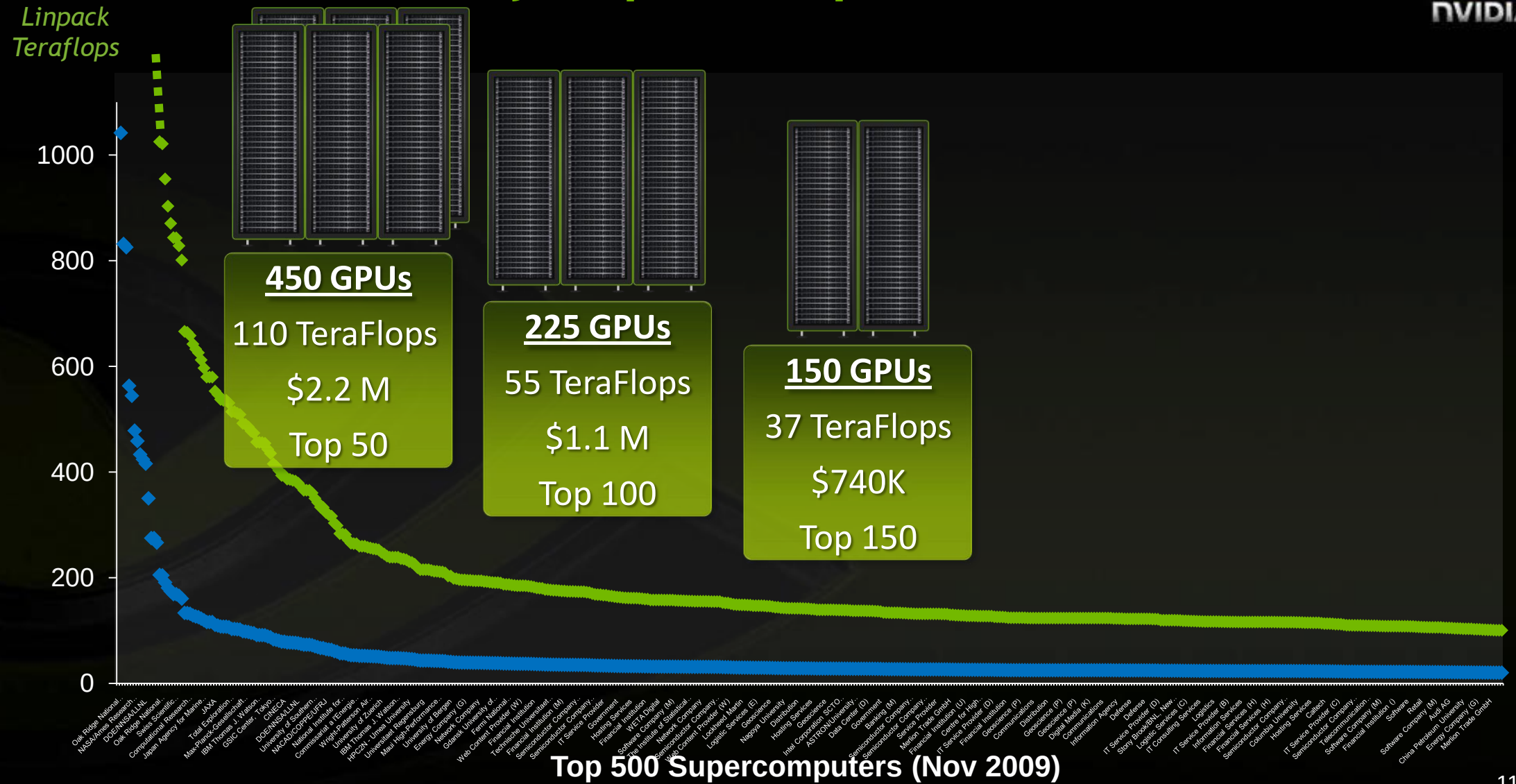
- Operating System
- Application
- Host Copy of Application Data

Today: Supercomputing is Expensive



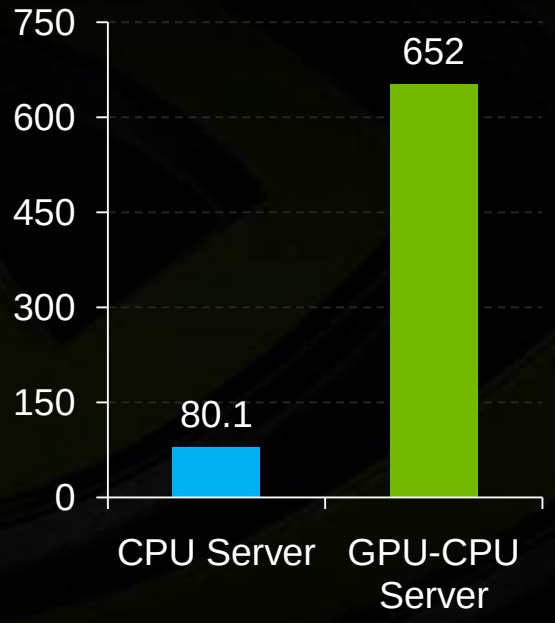
Source: Top500 List, June 2009

What if Every Supercomputer Had Tesla?

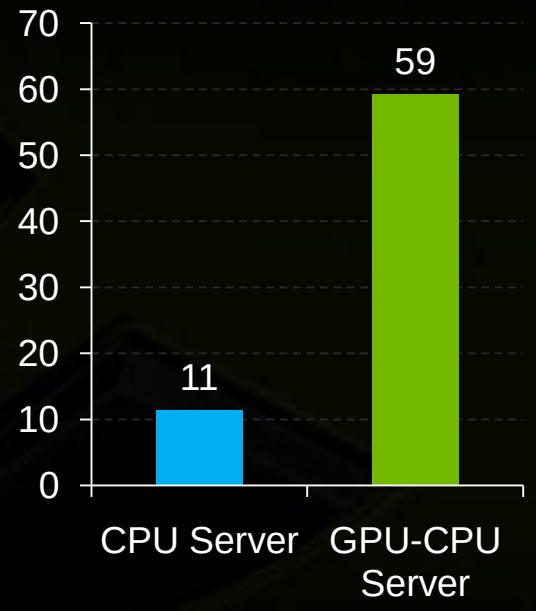


8x Higher Linpack

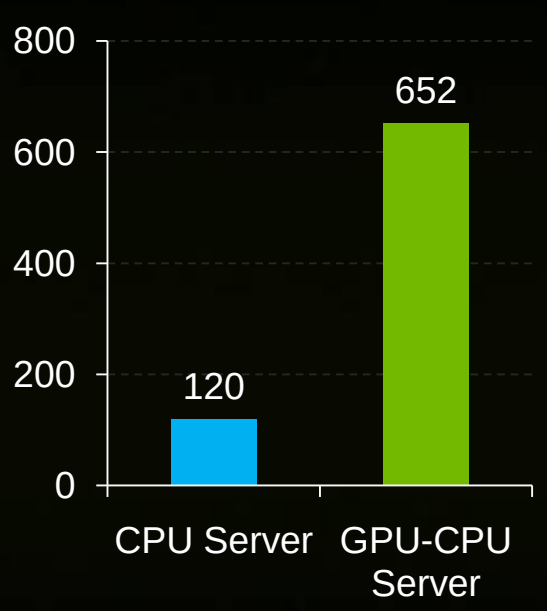
8x



5x



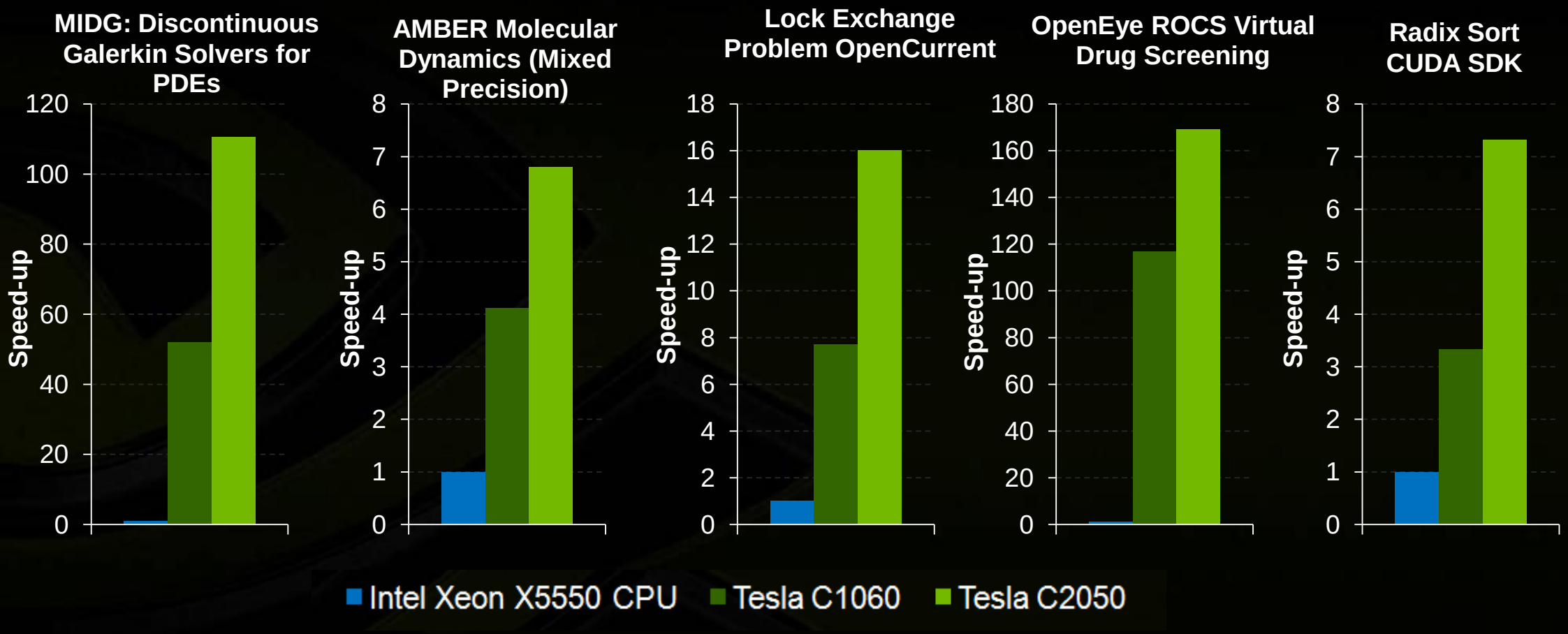
5.5x



CPU 1U Server: 2x Intel Xeon X5550 (Nehalem) 2.66 GHz, 48 GB memory, $7K, 0.67 kw

GPU-CPU 1U Server: 2x Tesla C2050 + 2x Intel Xeon X5550, 48 GB memory, $11K, 1.0 kw

Performance Summary

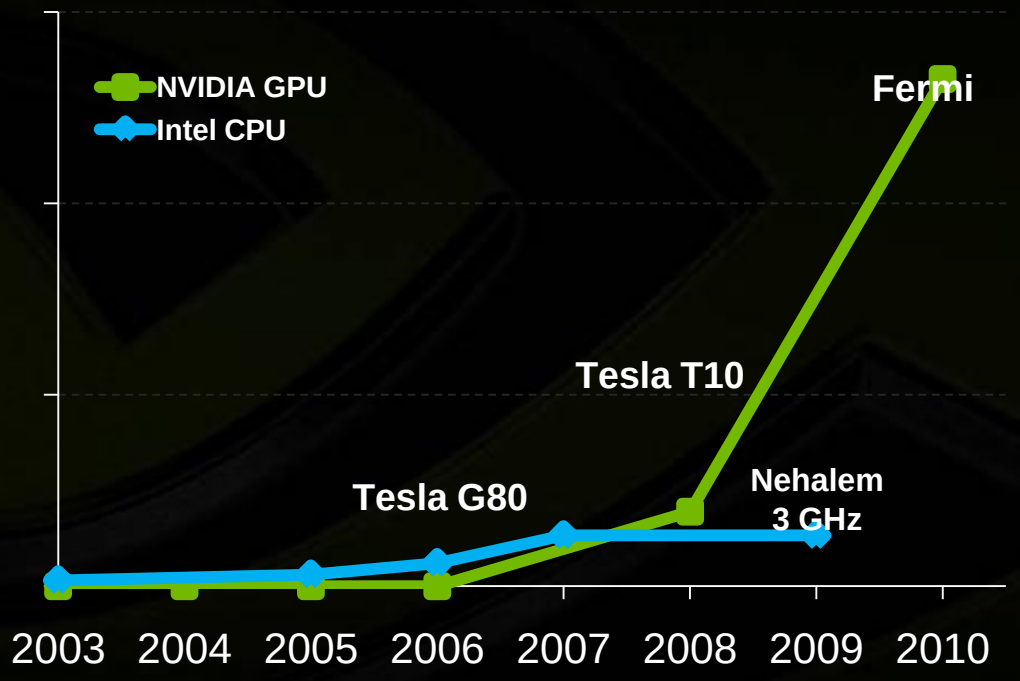


Preliminary data

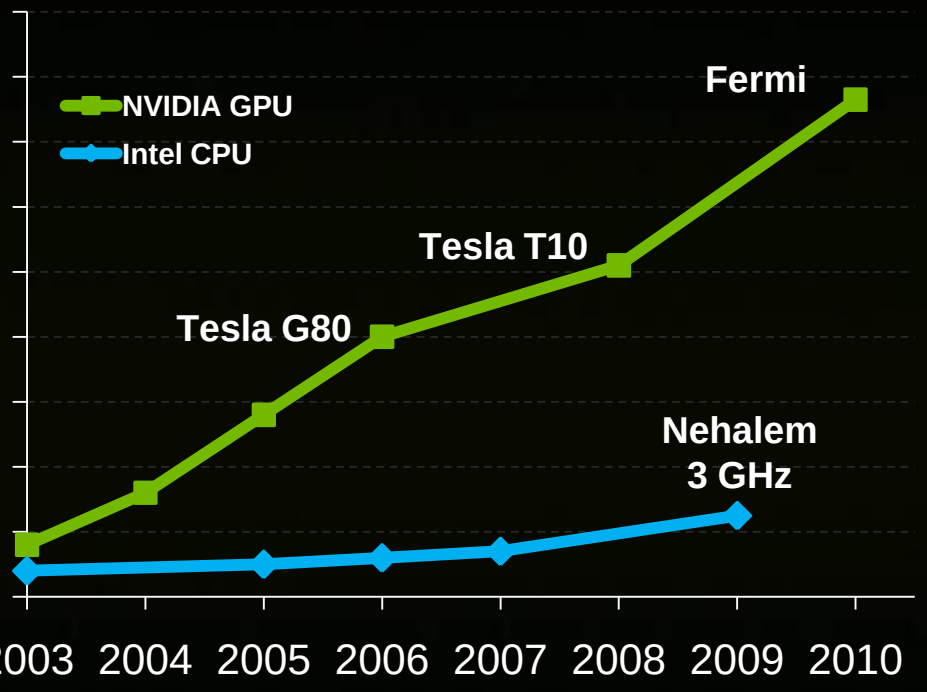

Introducing Fermi
NVIDIA

Not Just Single Precision…

Peak Double Precision Performance
Gflops/sec



Peak Memory Bandwidth
GB/sec



Disclaimer: specification subject to change

Fermi: The Computational GPU

Performance

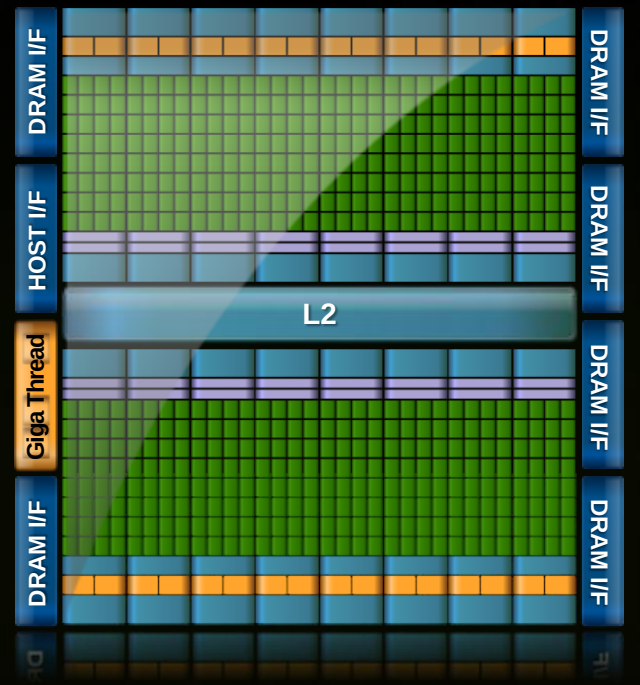
- 13x Double Precision of CPUs
- IEEE 754-2008 SP & DP Floating Point

Flexibility

- Increased Shared Memory from 16 KB to 64 KB
- Added L1 and L2 Caches
- ECC on all Internal and External Memories
- Enable up to 1 TeraByte of GPU Memories
- High Speed GDDR5 Memory Interface

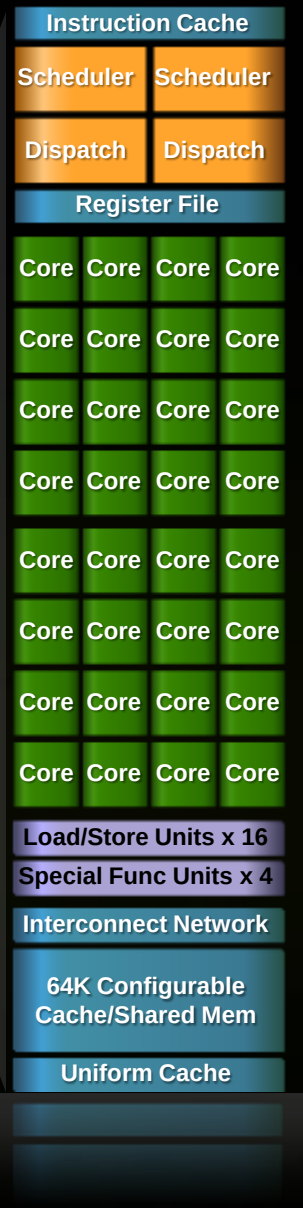
Usability

- Multiple Simultaneous Tasks on GPU
- 10x Faster Atomic Operations
- C++ Support
- System Calls, printf support



SM Architecture

- 32 CUDA cores per SM (448 total)
- 8x peak double precision floating point performance
  - 50% of peak single precision
- Dual Thread Scheduler
- 64 KB of RAM for shared memory and L1 cache (configurable)

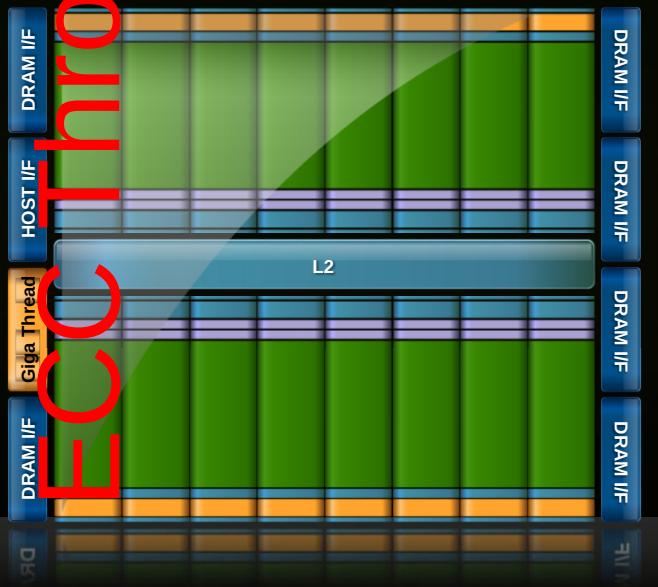


Cached Memory Hierarchy

First GPU architecture to support a true cache hierarchy in combination with on-chip shared memory

- L1 Cache per SM (16KB/48KB X 16 SMs)
  - Improves bandwidth and reduces latency
- Unified L2 Cache (768 KB)
  - Fast, coherent data sharing across all cores in the GPU
- GDDR5 Global Memory
  - 2x speed of GDDR3
  - Up to 1 Terabyte of memory attached to GPU
  - Operate on large data sets

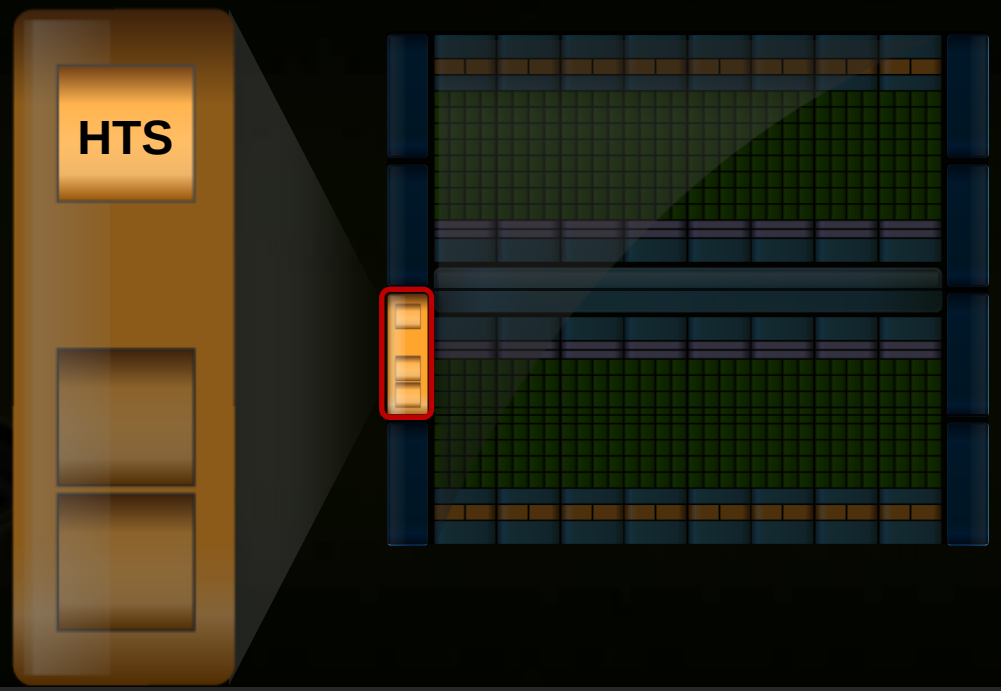
ECC throughout!



Parallel DataCache™ Memory Hierarchy

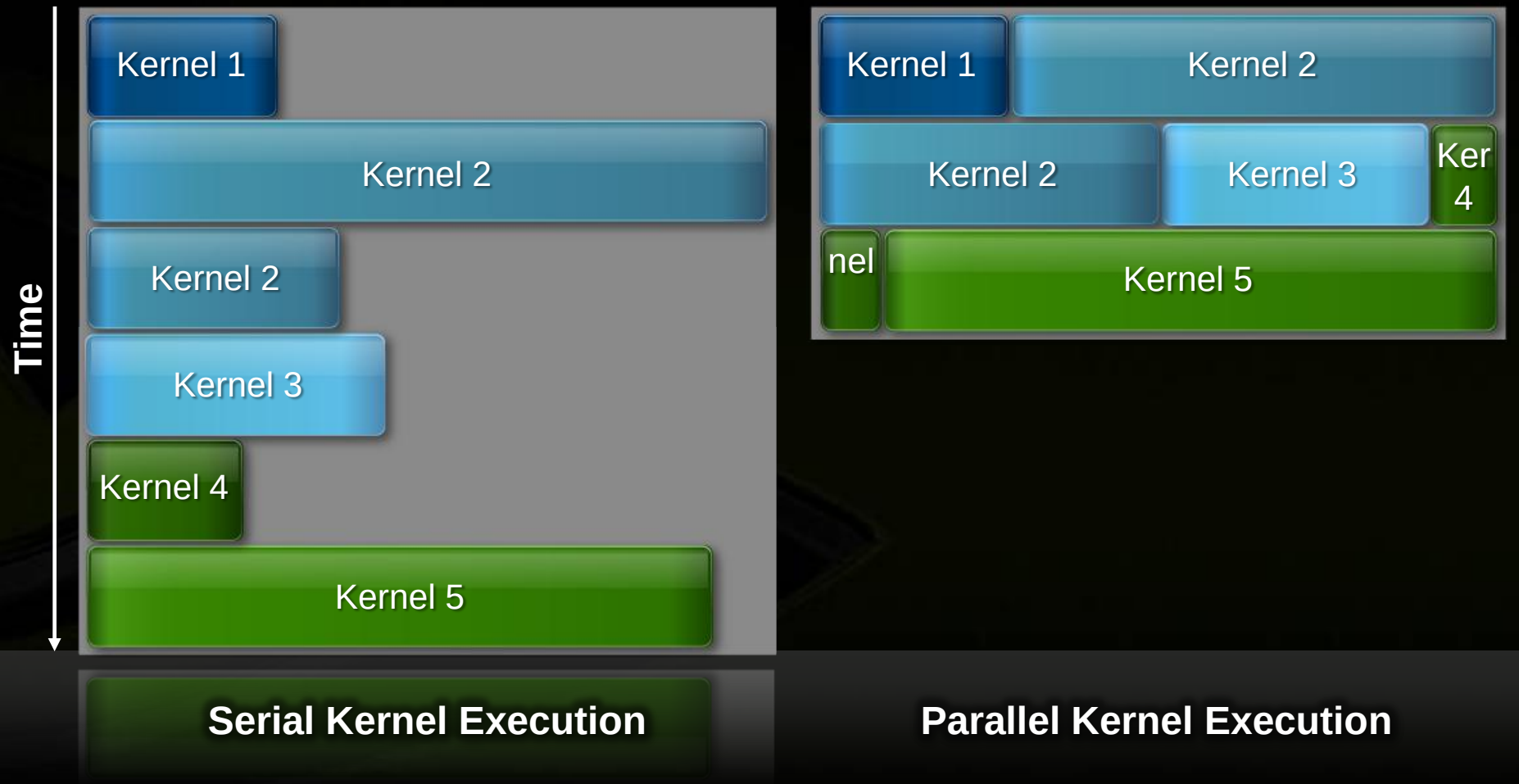
GigaThread™ Hardware Thread Scheduler (HTS)

- Hierarchically manages thousands of simultaneously active threads
- 10x faster application context switching
- Concurrent kernel execution



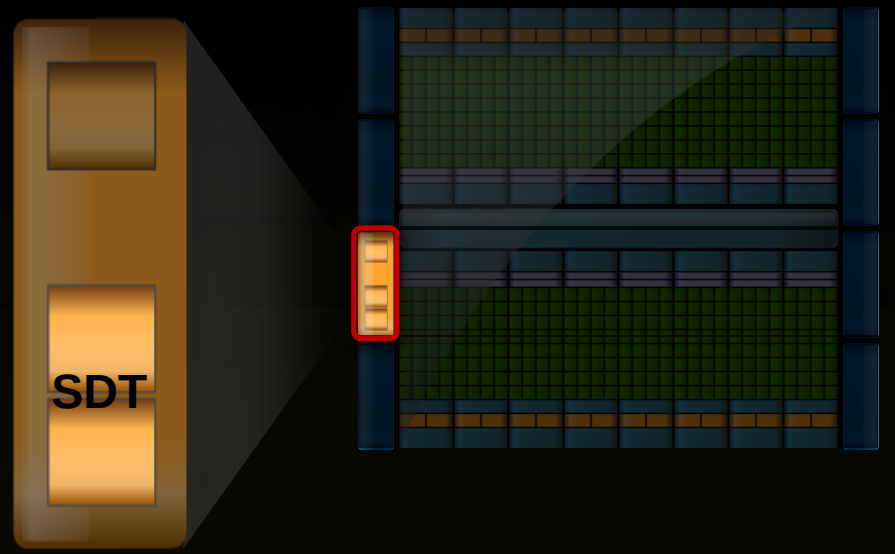
GigaThread Hardware Thread Scheduler

Concurrent Kernel Execution + Faster Context Switch

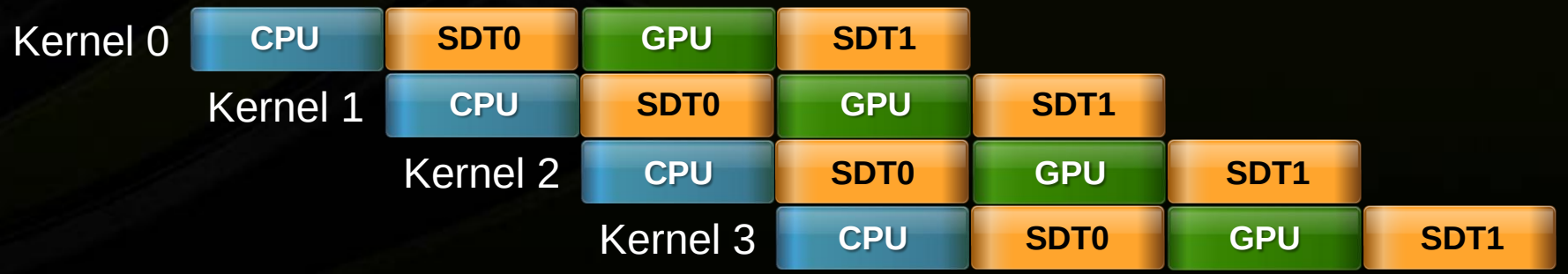


GigaThread Streaming Data Transfer Engine

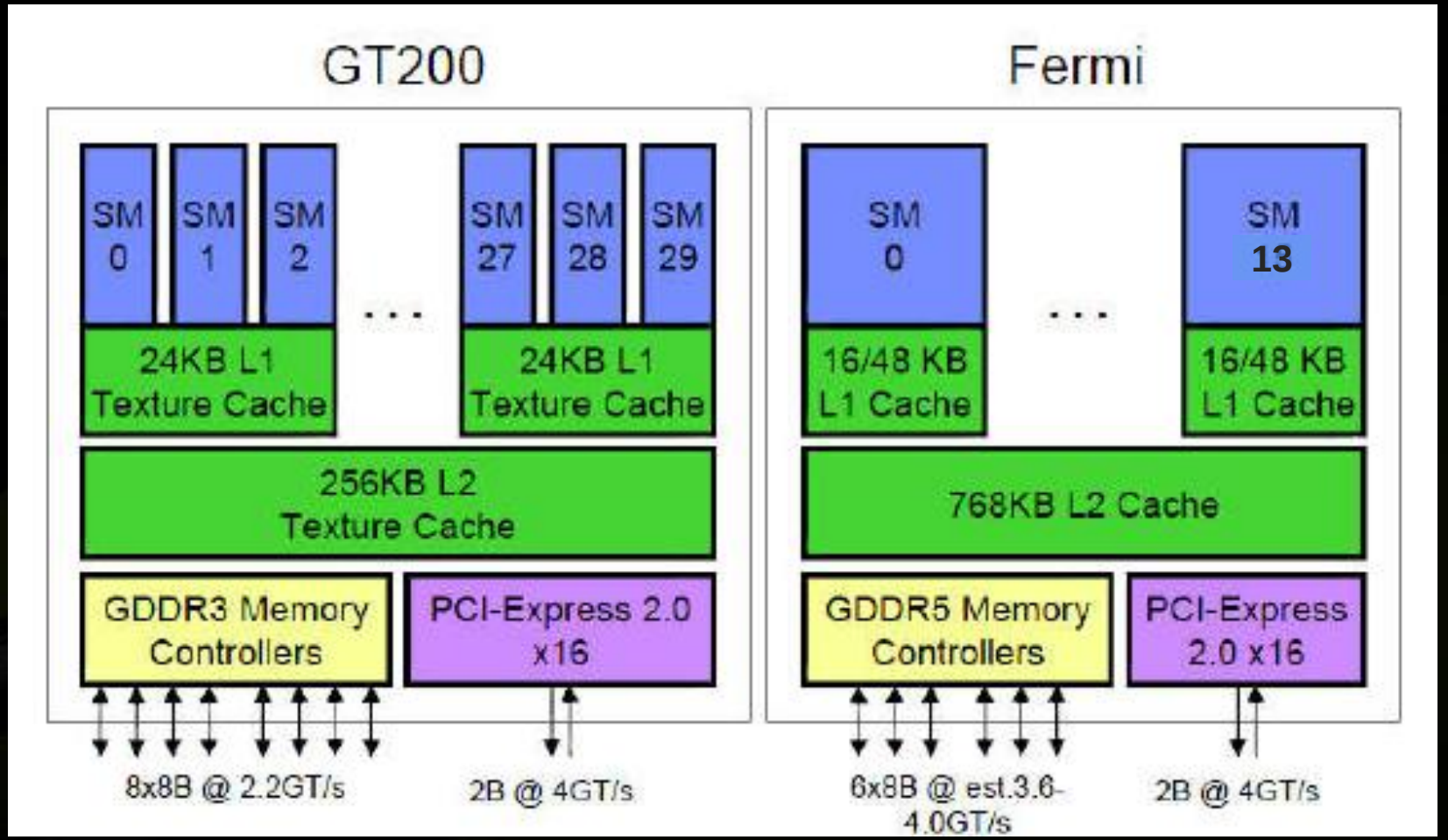
- **Dual DMA engines**
  - Simultaneous CPU→GPU and GPU→CPU data transfer
  - Fully overlapped with CPU and GPU processing time



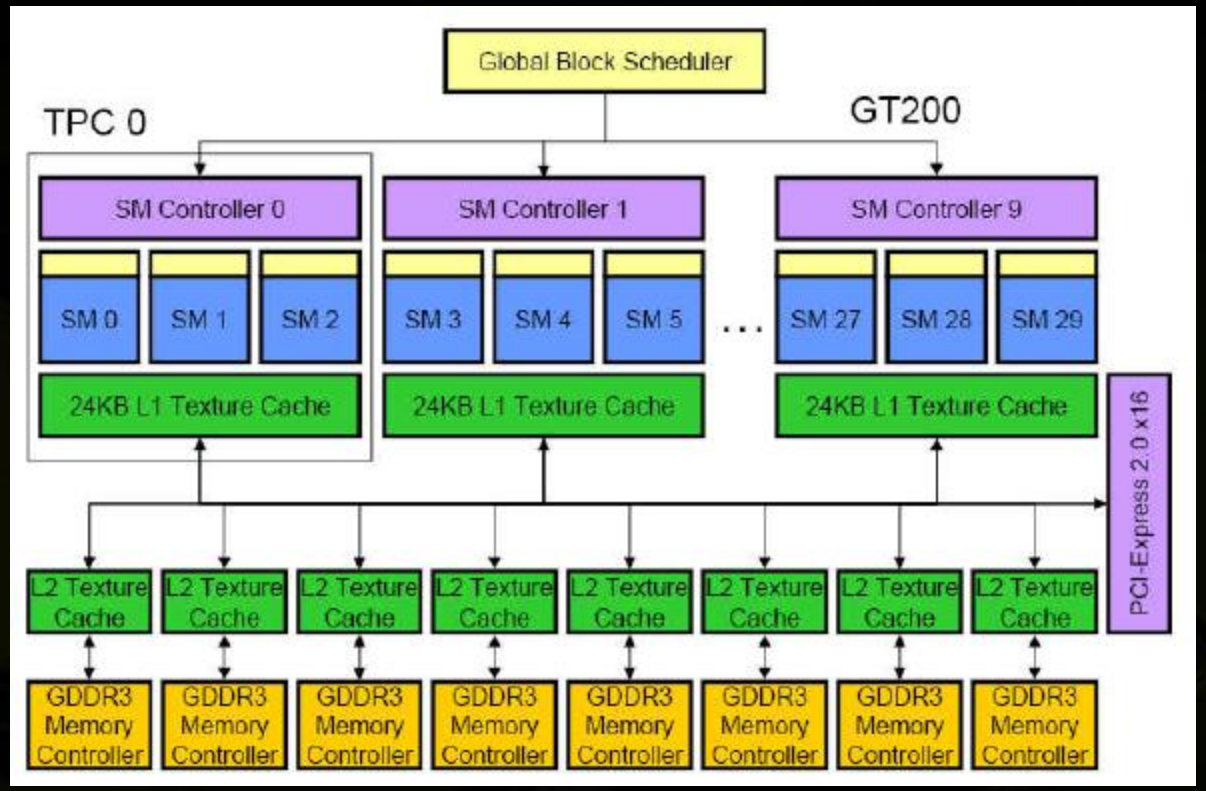
- **Activity Snapshot:**

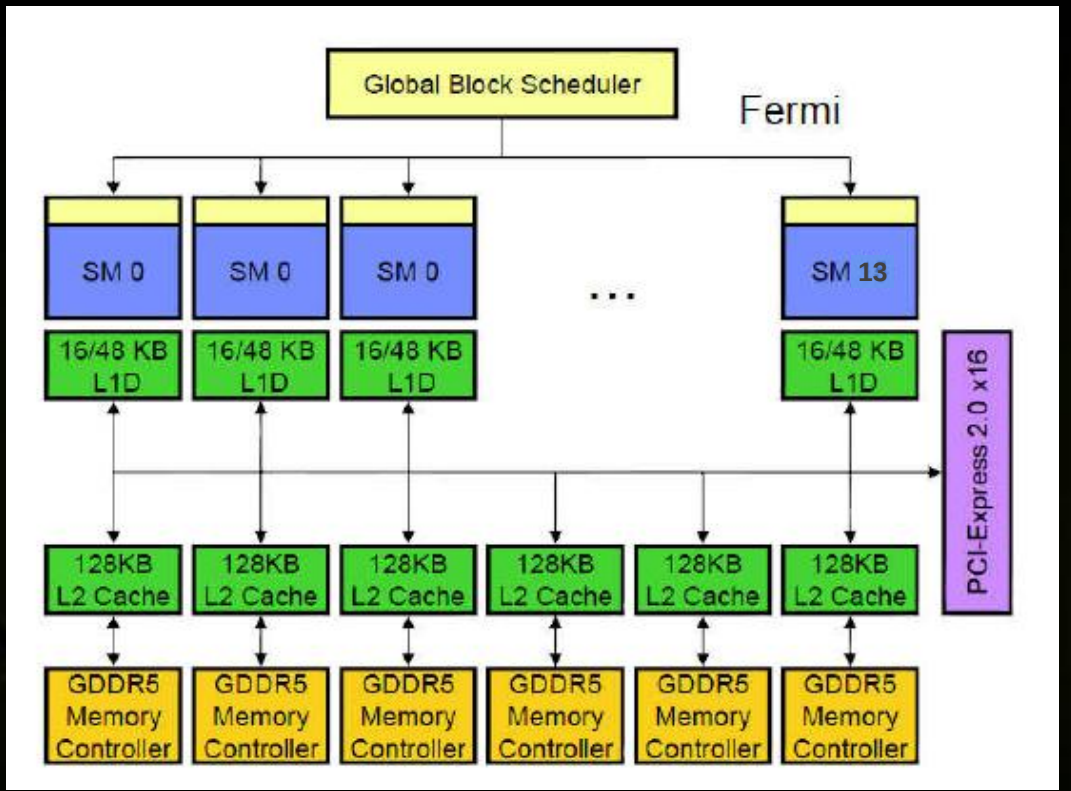


Comparison of GT200 & Fermi

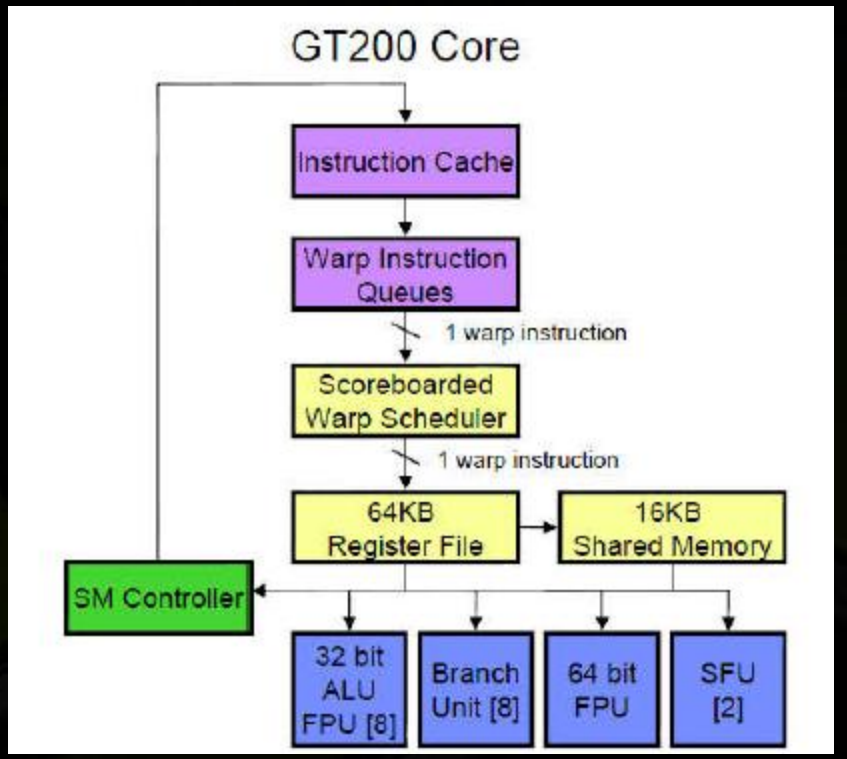


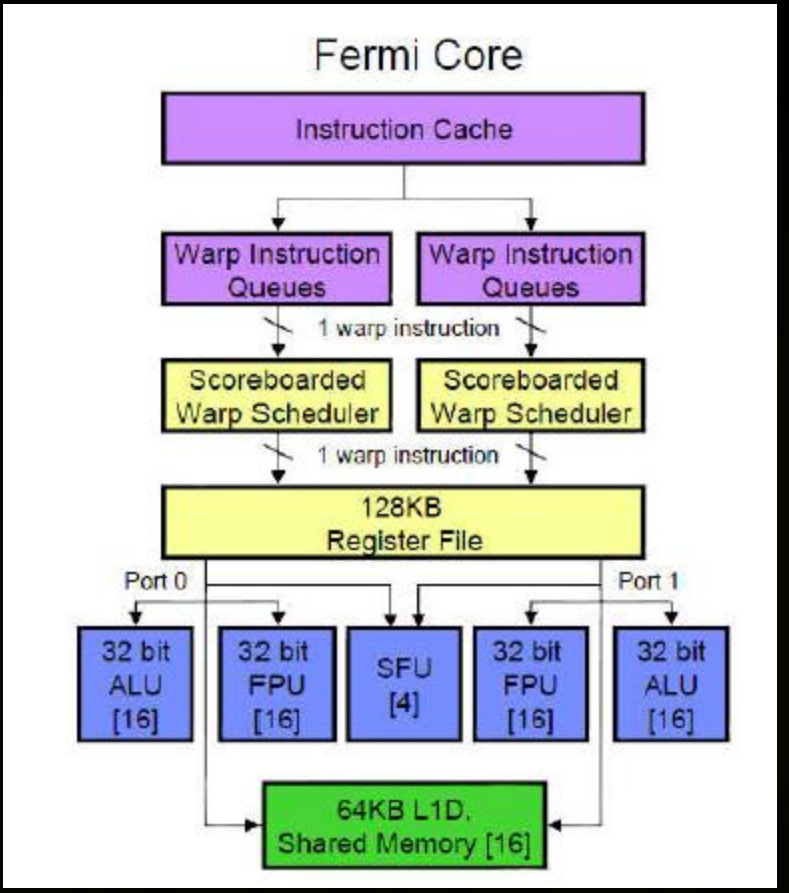
Comparison of GT200 & Fermi



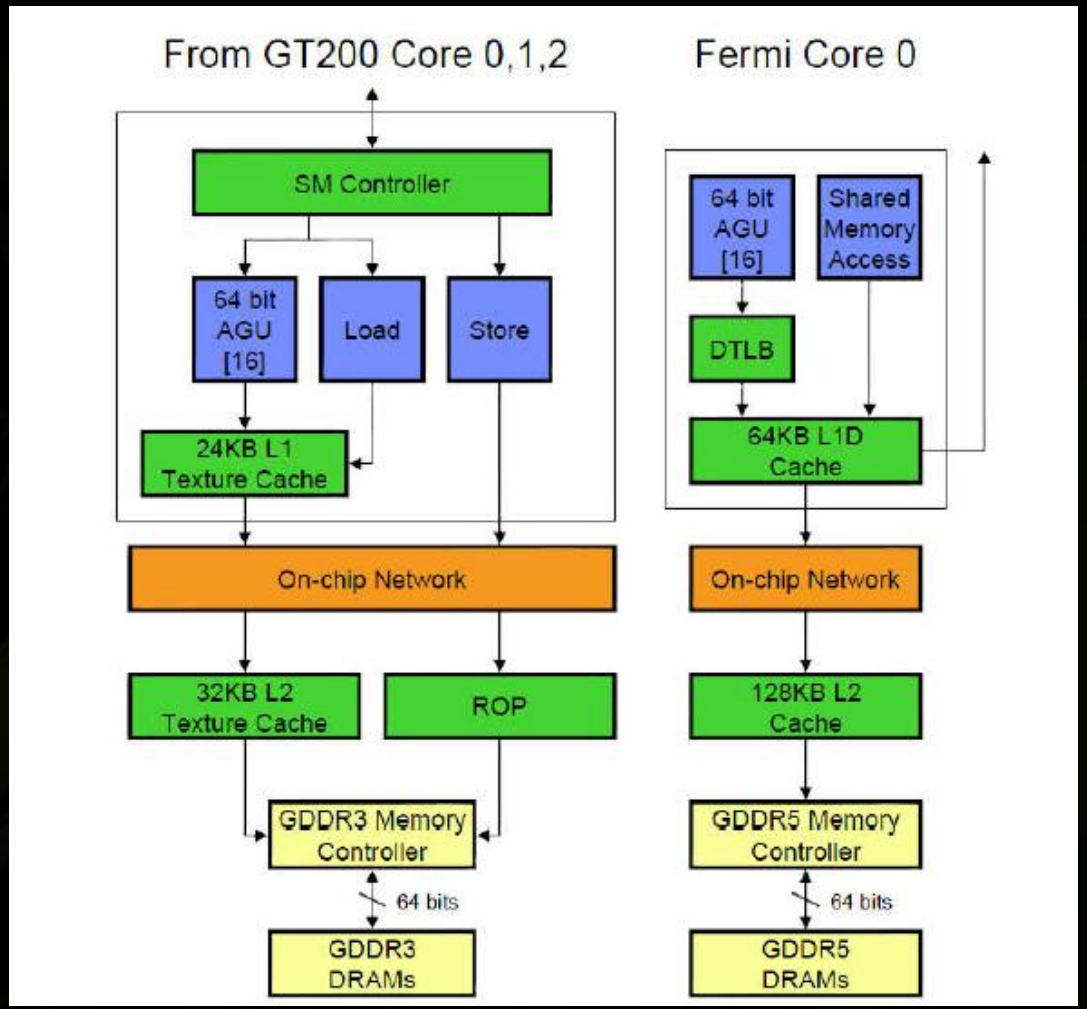


Comparison of Core Architecture

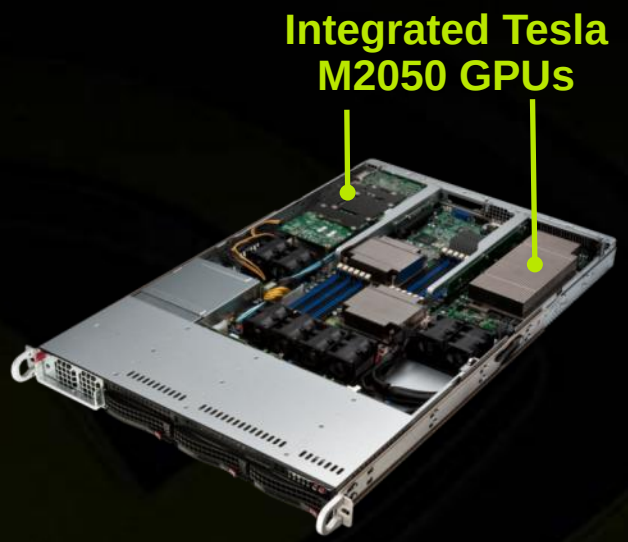




Comparison of Memory Hierarchy



Data Center / Workstation GPU Solutions



Integrated
CPU-GPU Servers
with Tesla M2050 GPUs

OEM CPU Server +
Tesla S2050
4 Tesla GPUs in 2U

Workstations
Up to 4x
Tesla C2050 GPUs

Tesla C-Series Workstation GPUs



	Tesla C1060	Tesla C2050	Tesla C2070
Architecture	Tesla 10-series GPU	Tesla 20-series GPU	
Number of Cores	240	448	
Caches	16 KB Shared Memory / 8 cores	64 KB L1 cache + Shared Memory / 32 cores, 768 KB L2 cache	
PCI-e DMA Engines	1	2	
Floating Point Peak Performance	933 Gflops (single) 78 Gflops (double)	1030 Gflops (single) 515 Gflops (double)	
GPU Memory	4 GB	3 GB 2.625 GB w/ ECC on	6 GB 5.25 GB w/ ECC on
Memory Bandwith	102 GB/s (GDDR3)	144 GB/s (GDDR5)	
System I/O	PCIe x16 Gen2	PCIe x16 Gen2	
Power	188 W (max)	247 W (max)	225 W (max)
Available	Available now	Shipping in May	Q3 2010

Data Center and Workstation Comparison

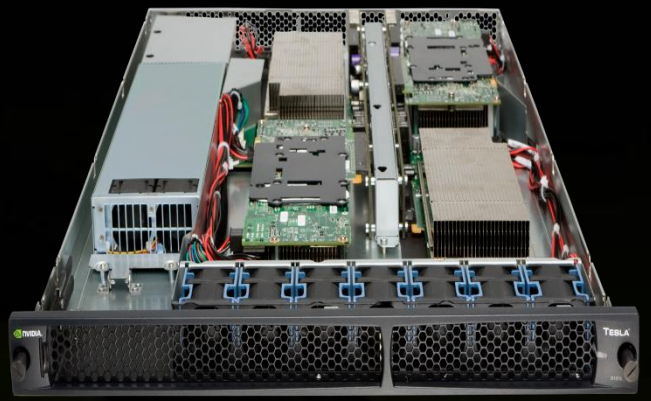
	Data Center Products		Workstation
	Tesla M2050 Module	Tesla S2050 1U System	Tesla C2050 Workstation Board
			
GPUs	**1 Tesla GPU**	**4 Tesla GPUs**	**1 Tesla GPU**
Single Precision Performance	1030 Gigaflops	4.12 Teraflops	1030 Gigaflops
Double Precision Performance	515 Gigaflops	2.06 Teraflops	515 Gigaflops
Memory	3 GB	12 GB (3 GB / GPU)	3 GB

Tesla S2050 1U GPU Systems

	S2050
Processors	**4 Tesla 20-series GPUs**
Number of Cores	**1792 per 1U (448 / GPU)**
Single precision performance	**4120 Gigaflops per 1U** ***1030 Gigaflops / GPU***
Double precision performance	**2060 Gigaflops per 1U** ***515 Gigaflops / GPU***
GPU Memory	**12 GB per 1U** ***3 GB / GPU*** ***2.625 GB with ECC on***
Memory Interface	**GDDR5**
System I/O	**2x PCIe x16 Gen2 Host interface cards (HICs)** **OR 2x PCIe x8 Gen2 Host interface cards (HICs)** **OR 1x PCIe x16 Gen2 Dual Host interface cards (DHIC)**
Power	**900 W (Typical)**
Available	**June 2010**

Tesla 1U Computing Systems and Accessories

Systems



S1070

S2050

Accessories

PCIe cable

DHIC

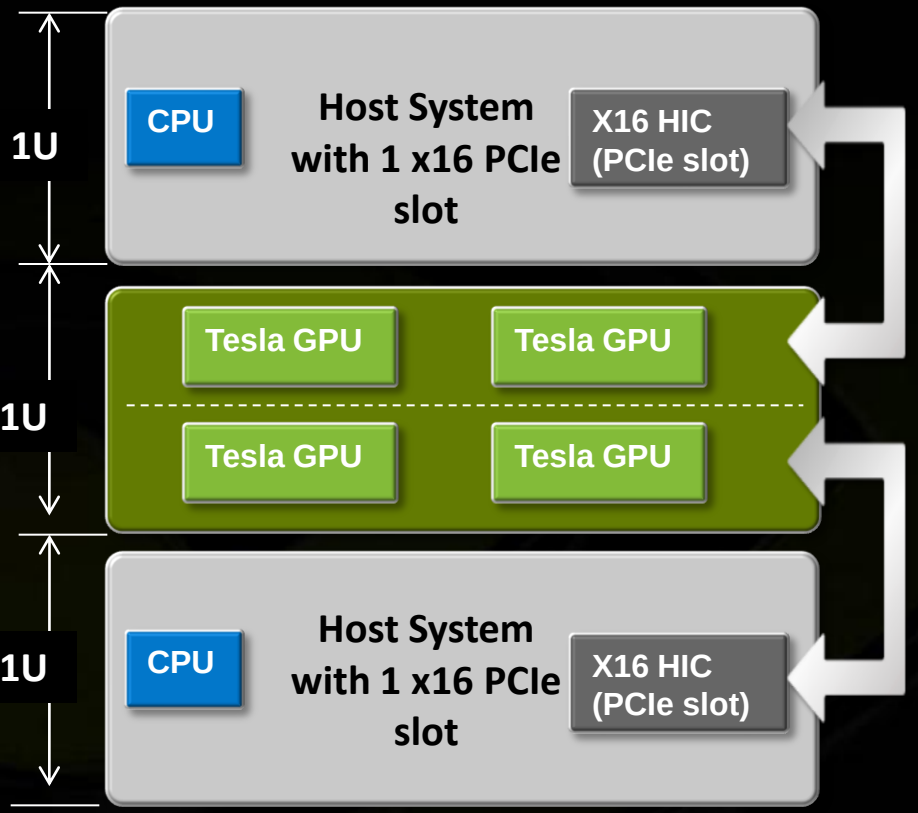
HIC

GHIC

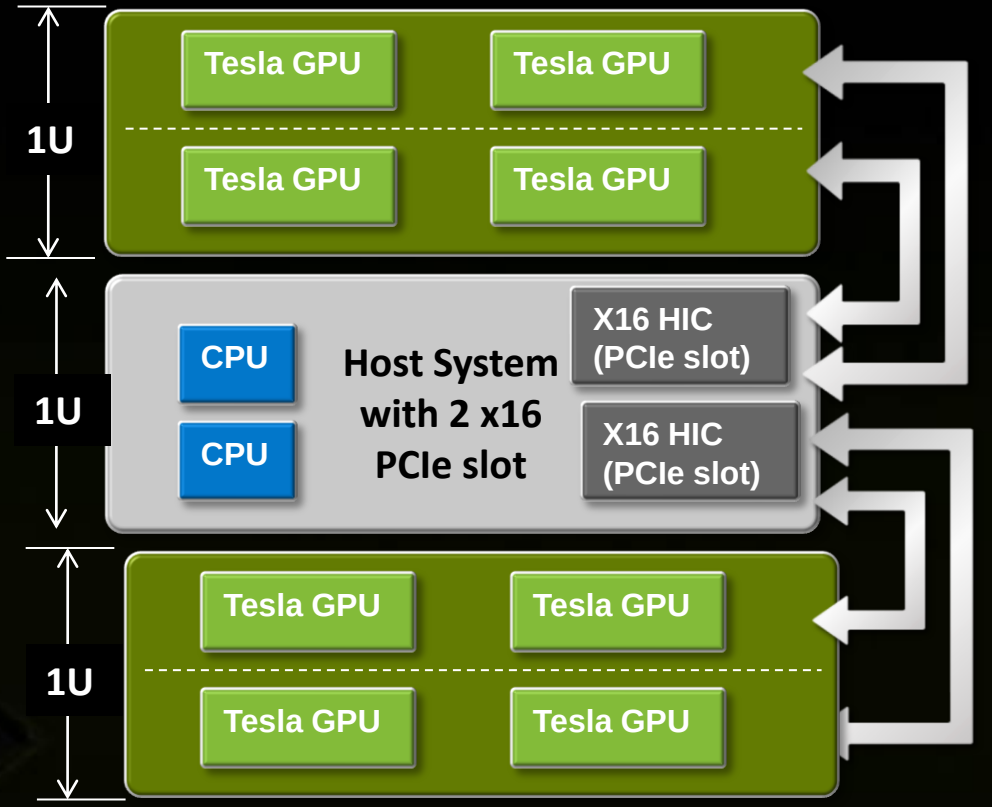
DVI Dongle for GHIC

S20xx uses same accessories as S1070

GPU density in a 3U configuration



GPU: CPU = 1:2
2 x16 HICs
Rack space = 3U

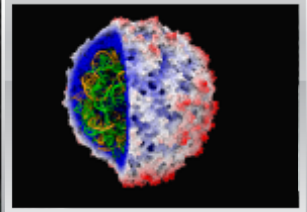


GPU: CPU = 1:4
2 x16 DHICs
Rack space = 3U

Range of Applications benefit from GPUs

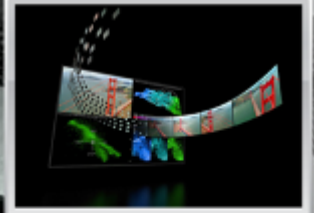
146X

Interactive visualization of volumetric white matter connectivity



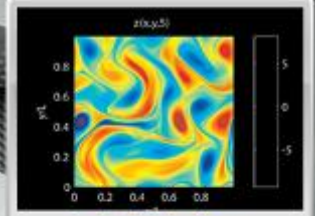
36X

Ionic placement for molecular dynamics simulation on GPU



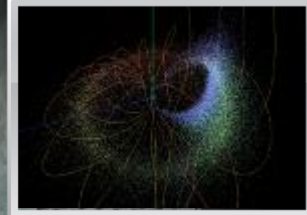
19X

Transcoding HD video stream to H.264



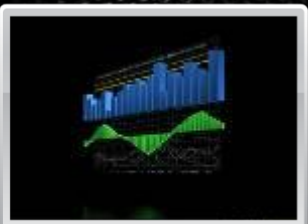
17X

Simulation in Matlab using .mex file CUDA function



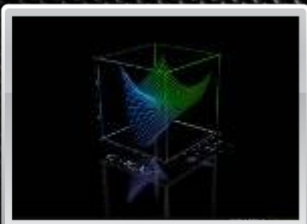
100X

Astrophysics N-body simulation



149X

Financial simulation of LIBOR model with swaptions



47X

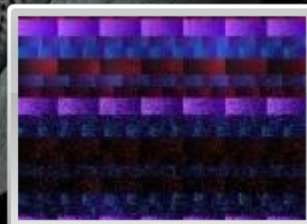
GLAME@lab: An M-script API for linear Algebra operations on GPU

20X

Ultrasound medical imaging for cancer diagnostics

24X

Highly optimized object oriented molecular dynamics



30X

Cmatch exact string matching to find similar proteins and gene sequences


The CUDA Environment
NVIDIA

GPU Parallel Computing Developer Eco-System

Numerical Packages

MATLAB
Mathematica
NI LabView
pyCUDA

Debuggers & Profilers

cuda-gdb
NV Visual Profiler
Parallel Nsight
Allinea
TotalView

GPU Compilers

C
C++
Fortran
OpenCL
DirectCompute
Java
Python

Parallelizing Compilers

PGI Accelerator
CAPS HMPP
mCUDA
OpenMP

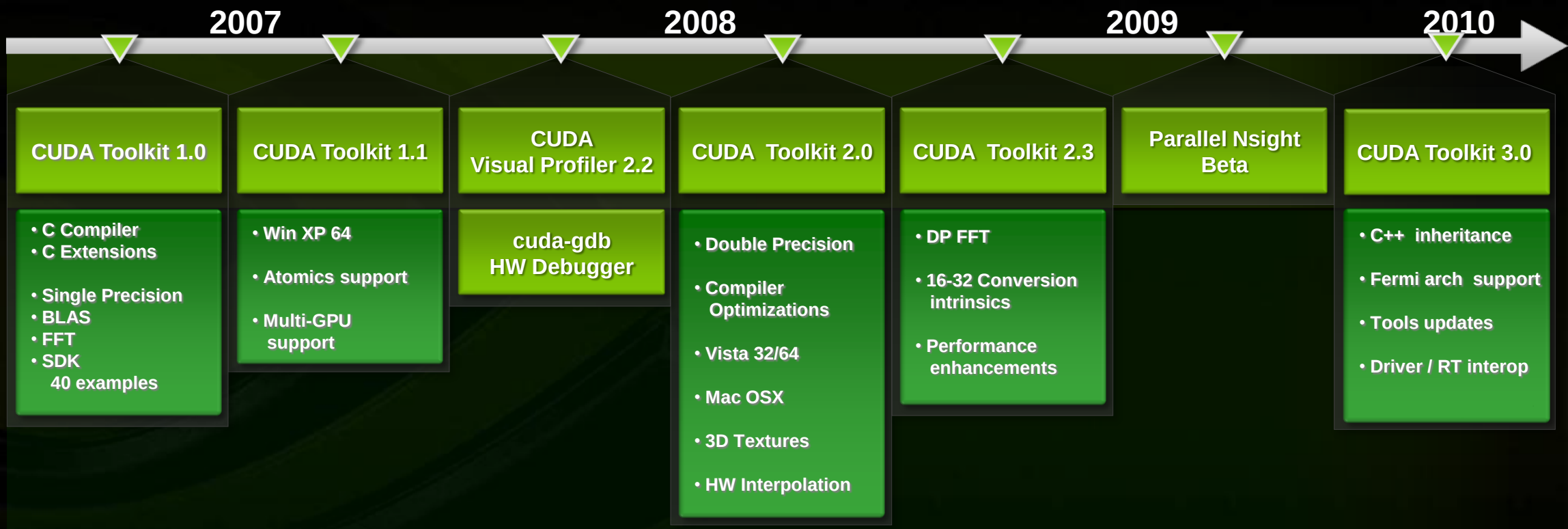
Libraries

BLAS
FFT
LAPACK
NPP
Video
Imaging

CUDA Consultants & Training

Solution Providers

CUDA C/C++ Leadership



What is CUDA?

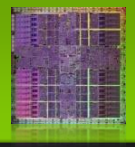
Broad Adoption

- Over 180,000,000 installed CUDA-Architecture GPUs
- Over 190k Toolkit downloads (v2.3)
- Supports Windows, Linux and MacOS
- GPU Computing spans HPC to Consumer
- 300+ Universities teaching GPU Computing on the CUDA Architecture

NVIDIA's Architecture for GPU Computing

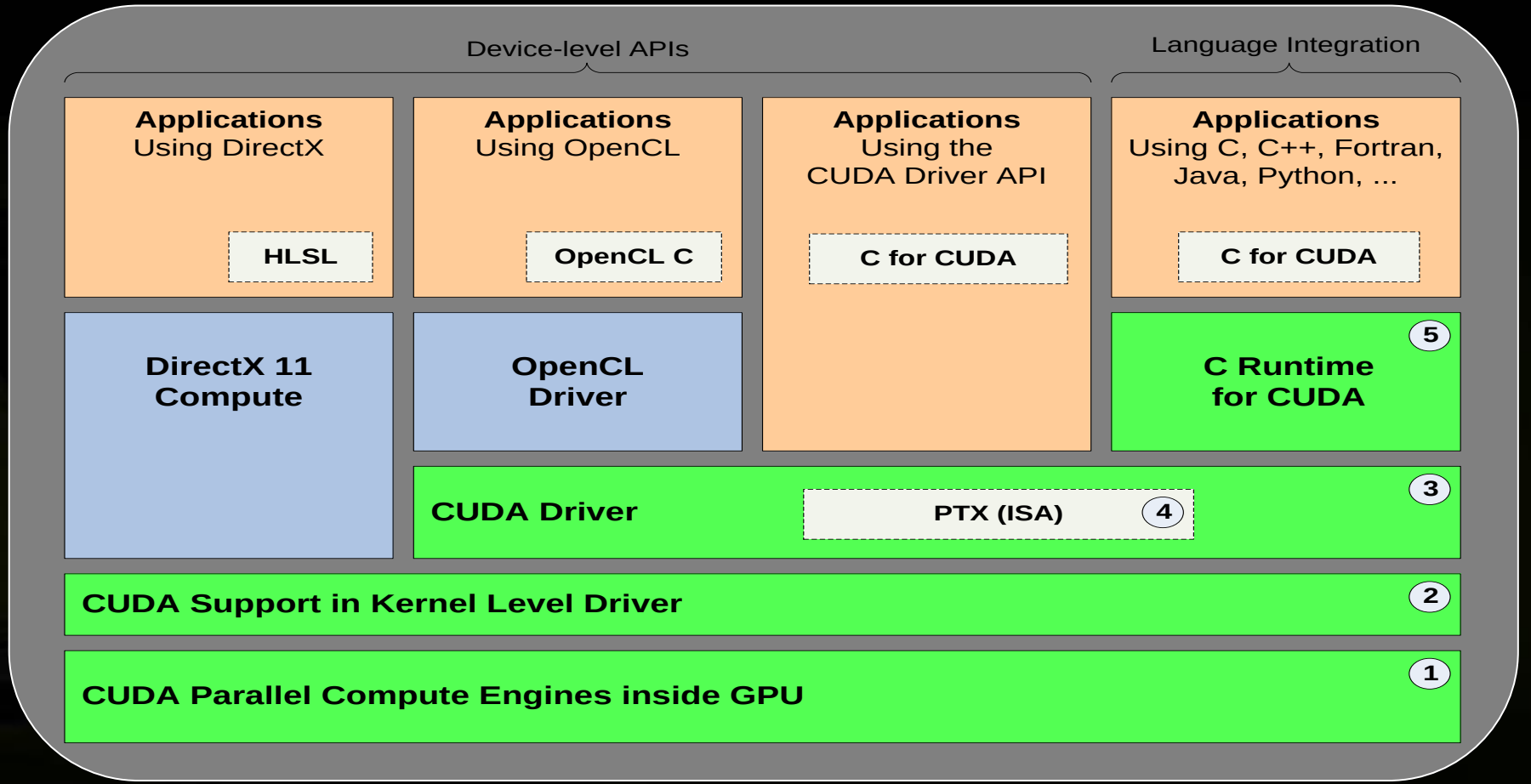
GPU Computing Applications

CUDA C/C++	OpenCL	Direct Compute	Fortran	Python, Java, .NET, …
• Over 60,000 developers • Running in Production since 2008 • SDK + Libs + Visual Profiler and Debugger	• 1st GPU demo • Shipped 1st OpenCL Conformant Driver • Public Availability • SDK + Visual Profiler	• Microsoft API for GPU Computing • Supports all CUDA-Architecture GPUs (DX10 and DX11)	• PGI Accelerator • PGI CUDA Fortran	• PyCUDA • jCUDA • CUDA.NET • OpenCL.NET

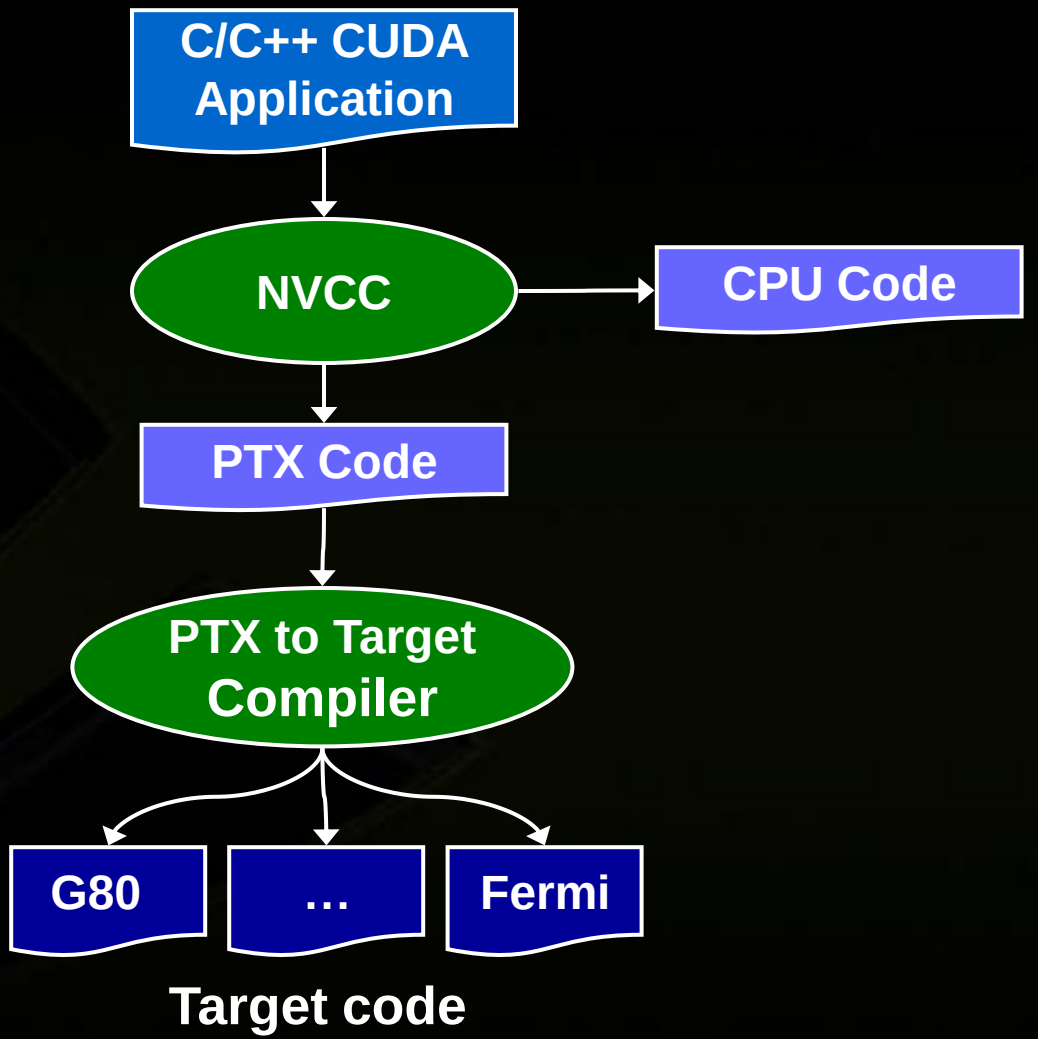


NVIDIA GPU
with the CUDA Parallel Computing Architecture

CUDA Tools Overview



CUDA Compilation Flowchart



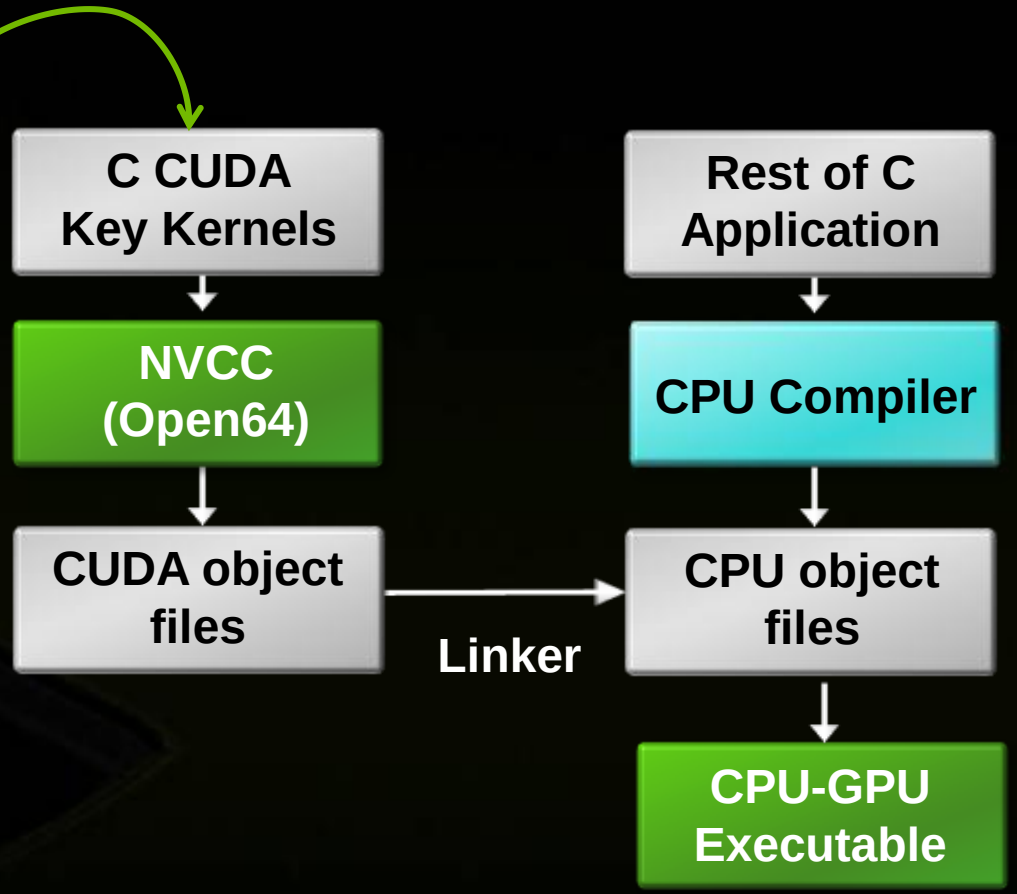
Compiling C for CUDA Applications

```
void serial_function(… ) {
  ...
}
void other_function(int ... ) {
  ...
}
```

```
void saxpy_serial(float ... ) {
   for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
```

```
void main( ) {
  float x;
  saxpy_serial(..);
  ...
}
```

Modify into Parallel CUDA code



C for CUDA : C with a few keywords

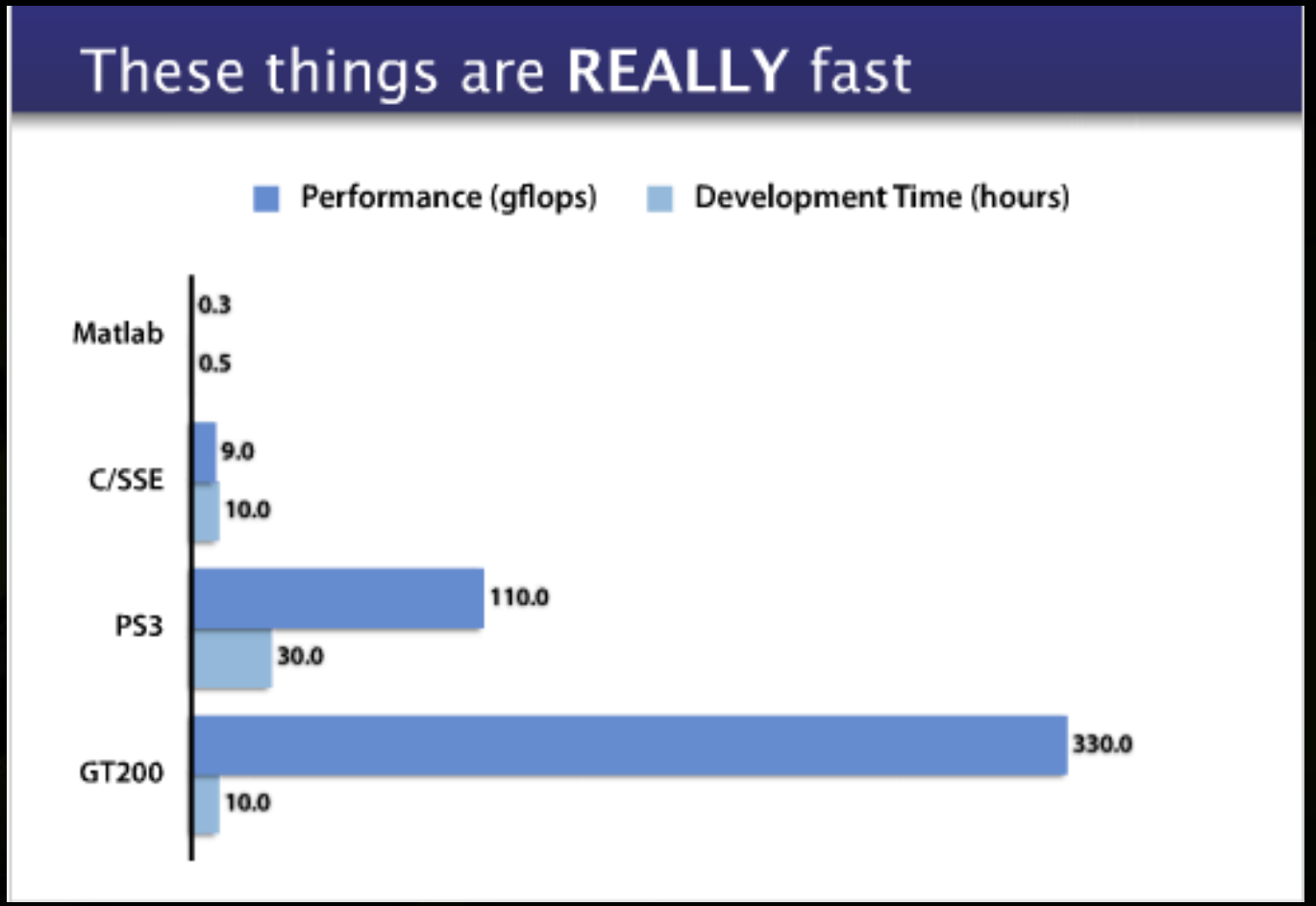
```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Parallel C Code

CUDA Programming Effort / Performance



Source : MIT CUDA Course

OpenCL

http://www.khronos.org/opencl

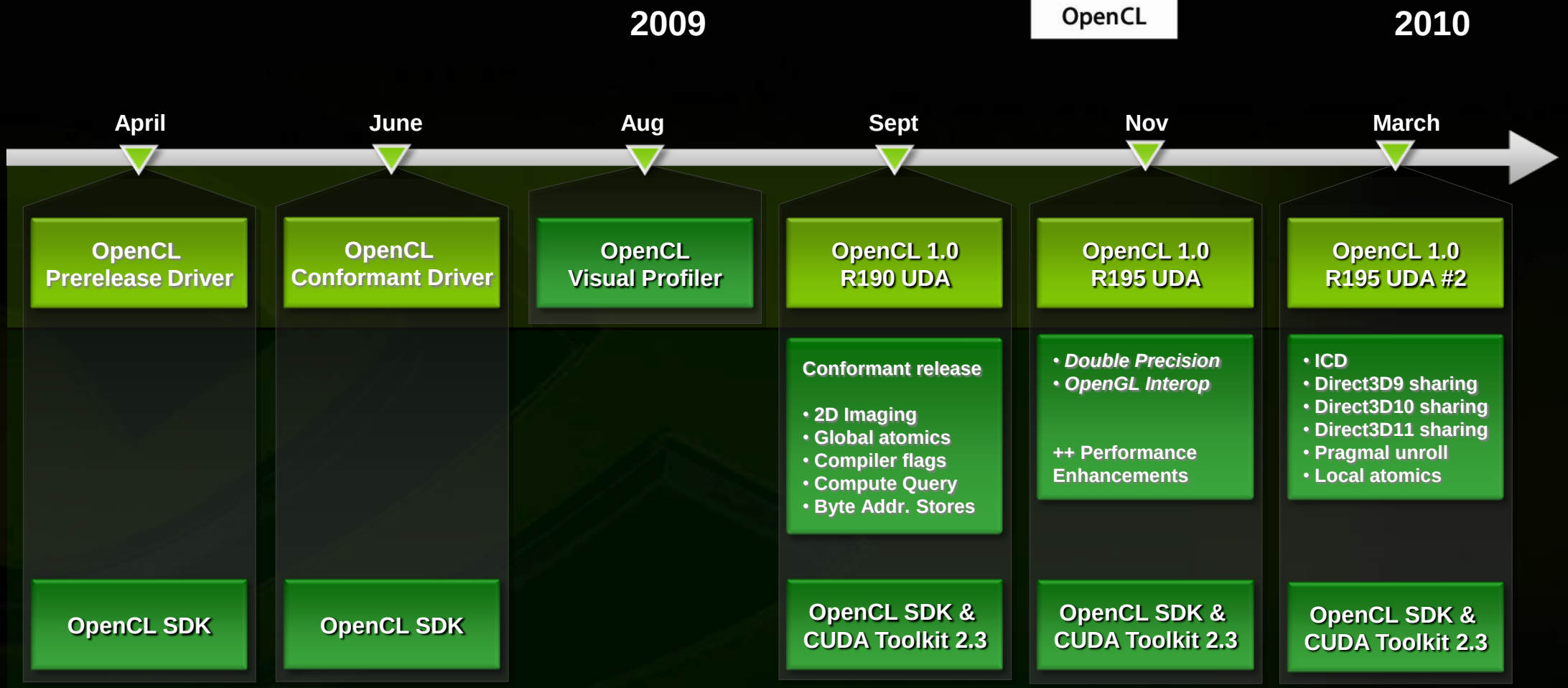
- Cross-vendor industry standard
  - Managed by the Khronos Group
- Low-level API for device management and launching kernels
  - Close-to-the-metal programming interface
  - JIT compilation of kernel programs
- C-based language for compute kernels
  - Kernels must be optimized for each processor architecture

NVIDIA released the first OpenCL v1.0 conformant driver for Windows and Linux to thousands of developers in June 2009

OpenCL & CUDA

- NVIDIA Supports all GPU programming languages
  - CUDA C/C++ is our platform of innovation
- Language integration, tools and libraries yield higher productivity
  - CUDA C Runtime has them today, OpenCL device driver API does not
- OpenCL designed for “write once, optimize everywhere”
  - Kernels must be optimized for each processor architecture
- NVIDIA is still the leading vendor delivering support for OpenCL
  - Conformant in June 2009, in production drivers since September

NVIDIA OpenCL Execution



OpenCL Extensions Support

Introduction to CUDA C programming


Intro to CUDA C
Memory Management
NVIDIA

Memory Spaces

CPU and GPU have separate memory spaces

- Data is moved across PCIe bus
- Use functions to allocate/set/copy memory on GPU
  - Very similar to corresponding C functions

Pointers are just addresses

- Can’t tell from the pointer value whether the address is on CPU or GPU
- Must exercise care when dereferencing:
  - Dereferencing CPU pointer on GPU will likely crash
  - Dereferencing GPU pointer on CPU will likely crash

GPU Memory Allocation / Release

Host (CPU) manages device (GPU) memory

- cudaMalloc (void ** pointer, size_t nbytes)
- cudaMemset (void * pointer, int value, size_t count)
- cudaFree (void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Note: Device memory from GPU point of view is also referred to as global memory.

Data Copies

cudaMemcpy(void *dst, void *src, size_t nbytes,
  enum cudaMemcpyKind direction);

- returns after the copy is complete
- blocks CPU thread until all bytes have been copied
- doesn't start copying until previous CUDA calls complete

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

Non-blocking memcopies are provided

Code Walkthrough 1

- Allocate CPU memory for *n* integers
- Allocate GPU memory for *n* integers
- Initialize GPU memory to 0s
- Copy from GPU to CPU
- Print the values

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
```

Code Walkthrough 1

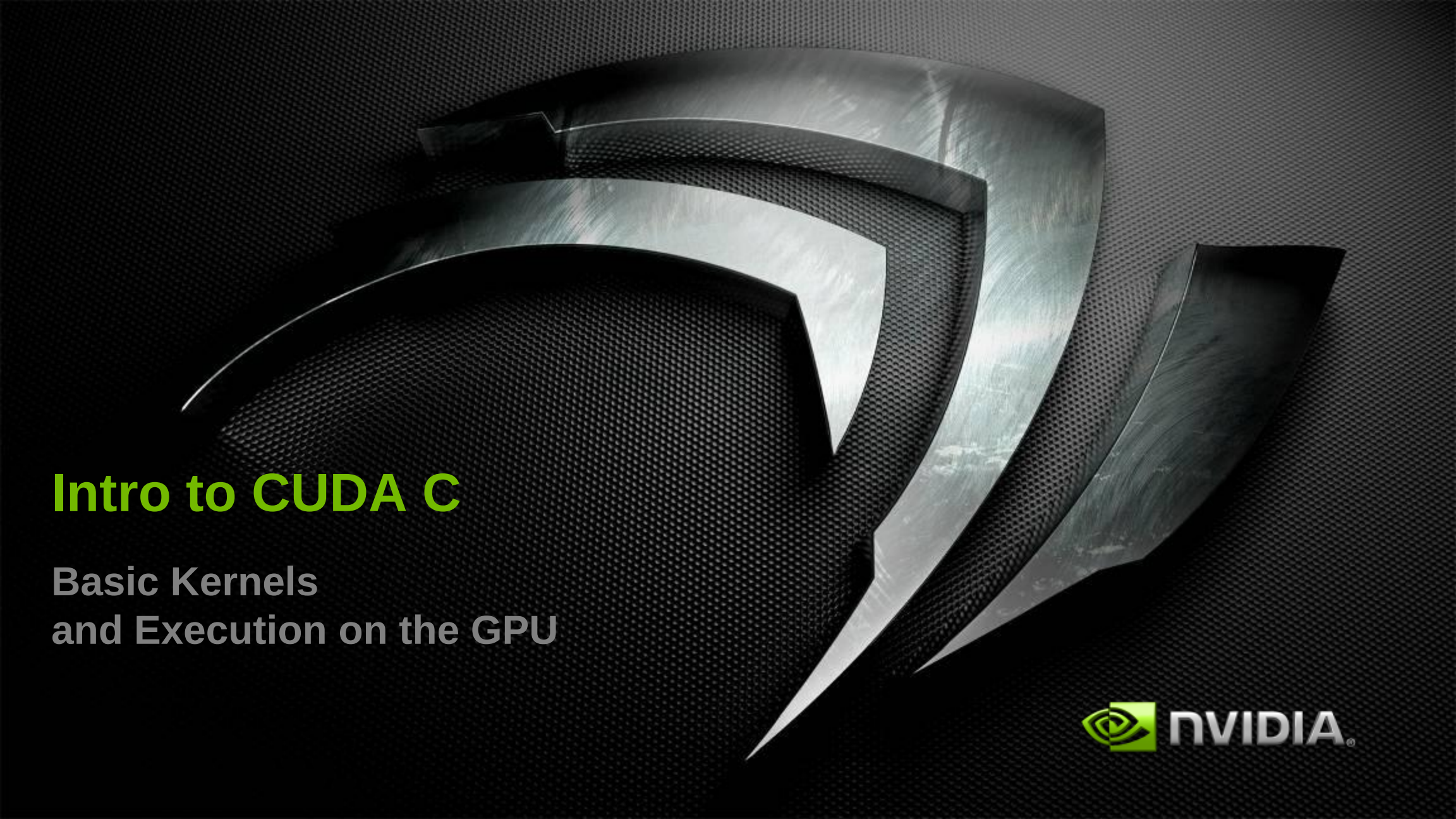
```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )  {
        printf("couldn't allocate memory\n"); return 1;
    }
```

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )  {
        printf("couldn't allocate memory\n"); return 1;
    }

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
```

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a ) {
        printf("couldn't allocate memory\n"); return 1;
    }

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```


Intro to CUDA C
Basic Kernels
and Execution on the GPU
NVIDIA

CUDA Programming Model

- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables

Thread Hierarchy

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory

IDs and Dimensions

Threads

- 3D IDs, unique within a block

Blocks

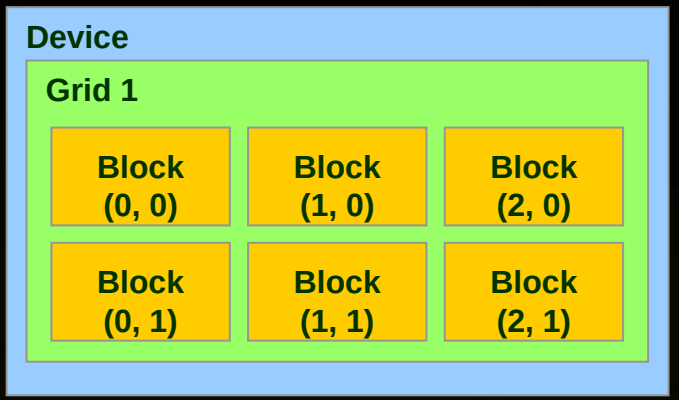
- 2D IDs, unique within a grid

Dimensions set at launch time

- Can be unique for each grid

Built-in variables

- threadIdx, blockIdx
- blockDim, gridDim



(Continued)

IDs and Dimensions

Threads

- 3D IDs, unique within a block

Blocks

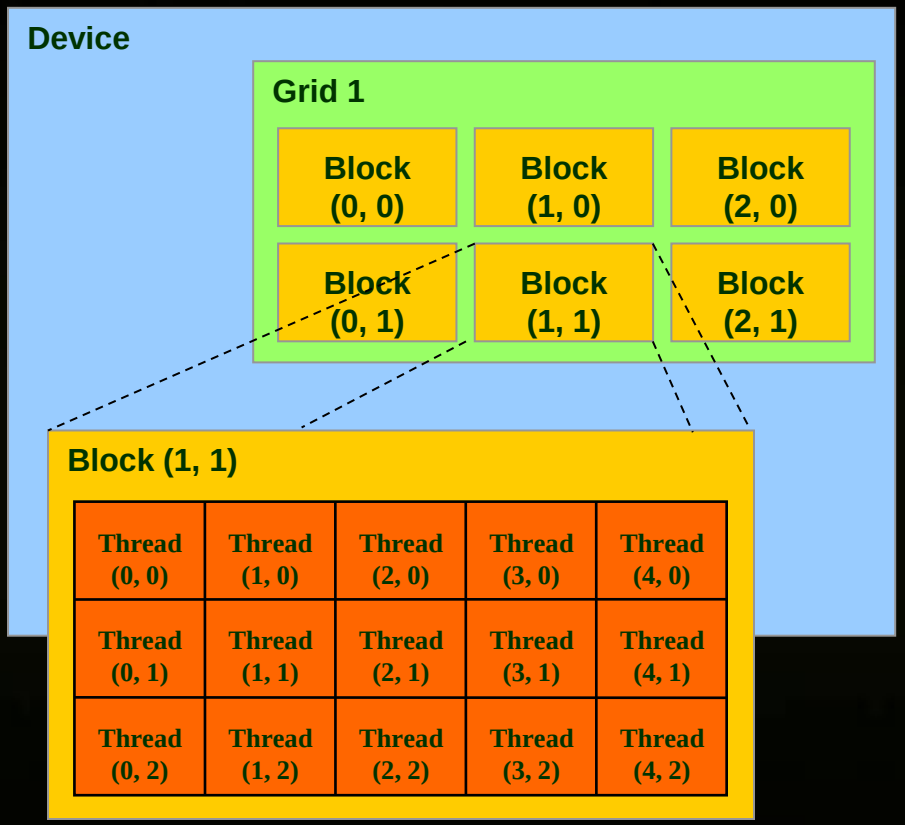
- 2D IDs, unique within a grid

Dimensions set at launch time

- Can be unique for each grid

Built-in variables

- threadIdx, blockIdx
- blockDim, gridDim



Code executed on GPU

C function with some restrictions:

- Can only access GPU memory (0-copy is the exception)
- No variable number of arguments
- No static variables
- No recursion

Must be declared with a qualifier:

- __global__ : launched by CPU, cannot be called from GPU must return void
- __device__ : called from other GPU functions, cannot be launched by the CPU
- __host__ : can be executed by CPU
- __host__ and __device__ qualifiers can be combined
  - Sample using mathematical functions

Code Walkthrough 2

- Build on Walkthrough 1
- Write a kernel to initialize integers
- Copy the result back to CPU
- Print the values

Kernel Code (executed on GPU)

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Launching Kernels on GPU

Launch parameters

- grid dimensions (up to 2D), dim3 type
- thread-block dimensions (up to 3D), dim3 type
- shared memory: number of bytes per block
  - for extern smem variables declared without size
  - Optional, 0 by default
- stream ID
  - Optional, 0 by default

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```

```
#include <stdio.h>

__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )  {
        printf("couldn't allocate memory\n"); return 1;
    }

    cudaMemset( d_a, 0, num_bytes );

    dim3 grid, block;
    block.x = 4;
    grid.x  = dimx / block.x;

    kernel<<<grid, block>>>( d_a );

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Kernel Variations and Output

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

GPU Kernel Execution

How the HW executes kernels

- GPU consists of multiple cores (Multiprocessors, up to 30)
- Blocks are launched on MPs
- Each MP can have multiple concurrent blocks executing
- Once a block is started it will not migrate to another MP

Blocks Must Be Independent

Any possible interleaving of blocks should be valid

- presumed to run to completion without pre-emption
- can run in any order
- can run concurrently OR sequentially

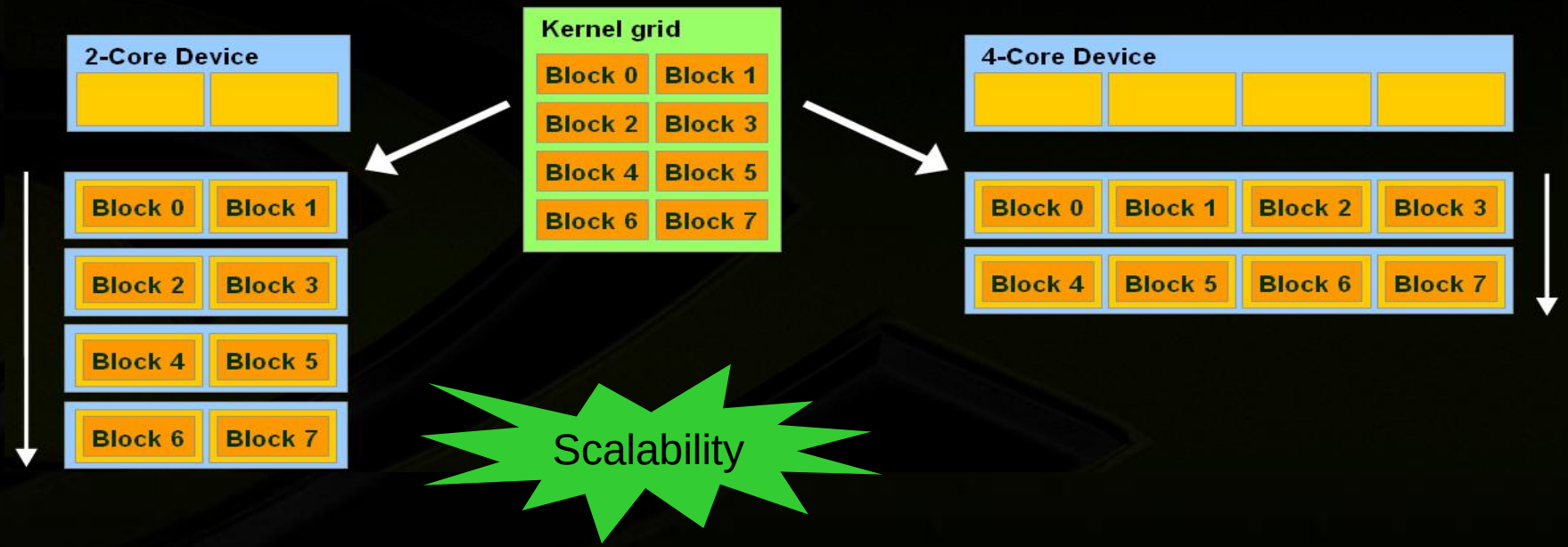
Blocks may coordinate but not synchronize

- shared queue pointer: OK
- shared lock: BAD … any dependence on order easily deadlocks

Independence requirement gives scalability

Blocks Must Be Independent

- Facilitates scaling of the same code across many devices




Intro to CUDA C
Coordinating CPU
and GPU Execution
NVIDIA

Synchronizing GPU and CPU

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel starts executing once all previous CUDA calls have completed
- cudaMemcpy() is synchronous
  - control returns to CPU once the copy is complete
  - copy starts once all previous CUDA calls have completed
- cudaThreadSynchronize()
  - blocks until all previous CUDA calls complete
- Outlook: Asynchronous CUDA calls
  - non-blocking memcopies
  - ability to overlap memcopies and kernel execution

CUDA Error Reporting to CPU

- All CUDA calls return error code:
  - except kernel launches
  - cudaError_t type
- cudaError_t cudaGetLastError(void)
  - returns the code for the last error ("no error" has a code)
- char* cudaGetErrorString(cudaError_t code)
  - returns a null-terminated character string describing the error

```
printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );
```

CUDA Event API

- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - measure elapsed time for CUDA calls (clock cycle precision)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to the event are completed
  - asyncAPI sample in CUDA SDK

CUDA Event API

```
cudaEvent_t start, stop;
cudaEventCreate(&start);        cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start);                cudaEventDestroy(stop);
```

Device Management

- CPU can query and select GPU devices
  - **cudaGetDeviceCount(int* count)**
  - **cudaSetDevice(int device)**
  - **cudaGetDevice(int *current_device)**
  - **cudaGetDeviceProperties(cudaDeviceProp* prop, int device)**
  - **cudaChooseDevice(int *device, cudaDeviceProp* prop)**
- **Outlook: Multi-GPU setup**
  - **device 0 is used by default**
  - **one CPU thread can control one GPU**
    - **multiple CPU threads can control the same GPU**
    - **SDK sample simpleMultiGPU**

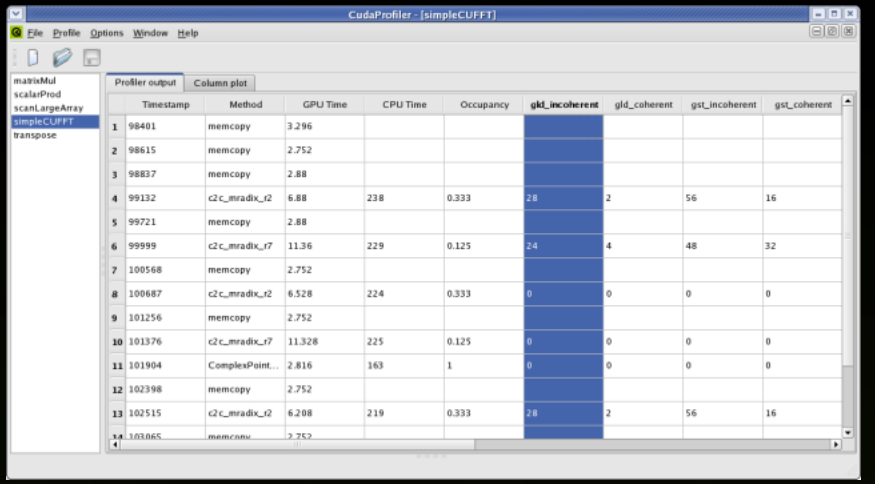

Intro to CUDA C
Development Resources
NVIDIA

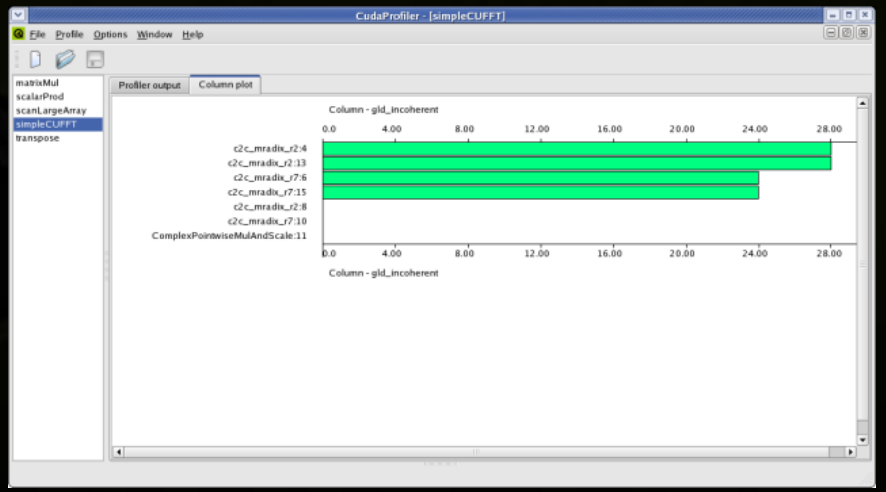
CUDA Programming Resources

CUDA toolkit
- Compiler, libraries, and documentation
- Support for Windows, Linux, and MacOS

CUDA SDK
- code samples
- whitepapers

Instructional materials on CUDA Zone
- slides and audio
- webinars
- tutorials
- forums





cuda-gdb: CUDA Application Debugging

- Included with CUDA toolkit
  - Superset of GDB commands to support GPU programming
  - Specific “cuda” commands to navigate threads
  - Support for instruction level debugging within a CUDA kernel
- Compile with “`nvcc -g -G`” flags for symbols
- Invoke on command line or as backend to DDD, Emacs, etc…
- Documentation: `/usr/local/cuda/doc/CUDA_GDB_v3-0.pdf`

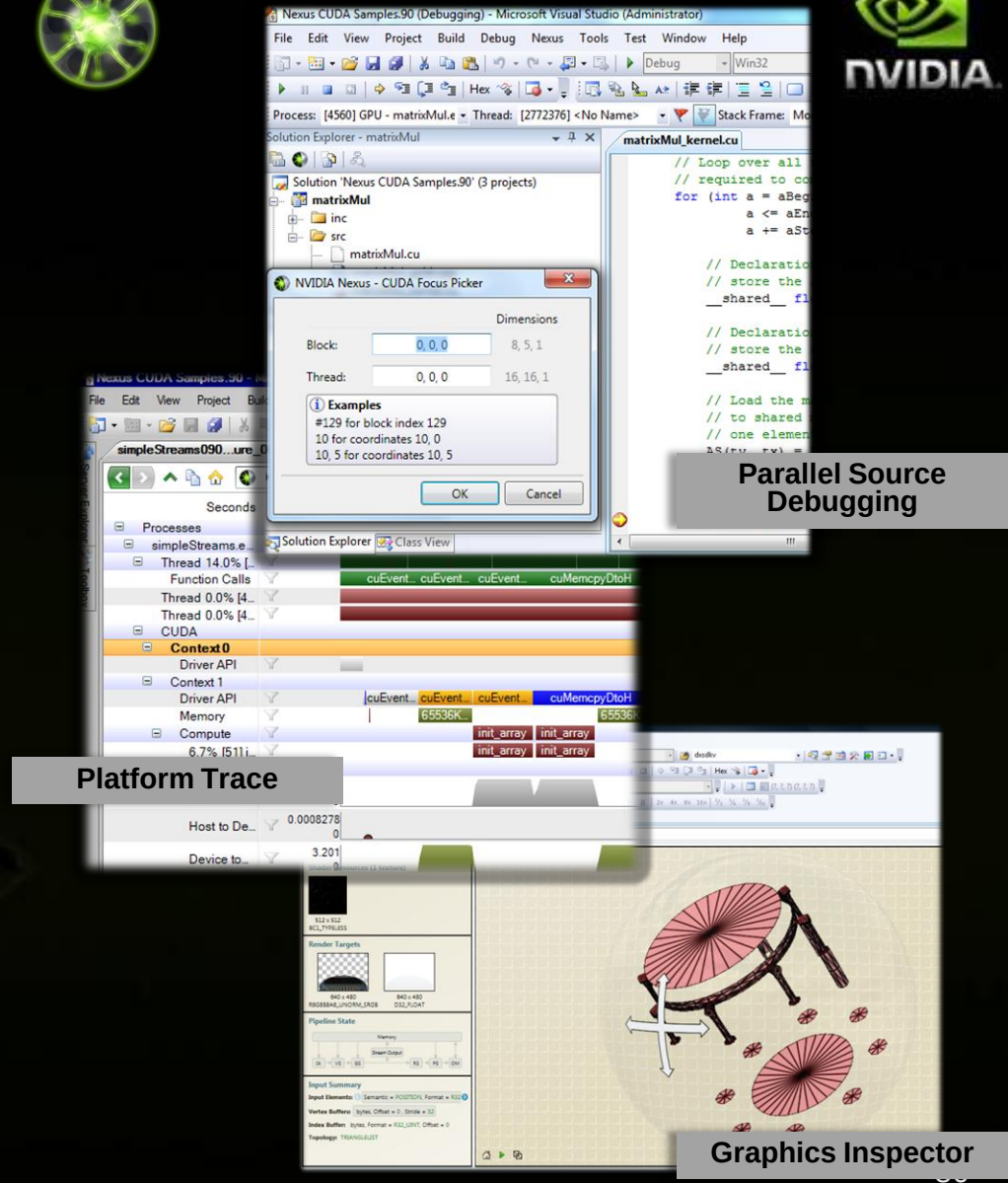
cudaprof: CUDA Application Profiling

- Included as part of CUDA toolkit
  - Command line usage with no re-compile
  - Configurable through environment variables
  - Low overhead hardware counters
  - Measures both instruction and memory operations
- Set environmental variable: `CUDA_PROFILE=1`
  - Run application as normal
  - Examine profile output: `cuda_profile_0.log`
  - Configure options and four active profile signals via a configuration file: `CUDA_PROFILE_CONFIG=configuration-file`
- Visual profiler: `cudaprof` provides ease of use and enhanced reporting
- Documentation: `/usr/local/cuda/docs/CUDA_Profiler_3.0.txt`

NVIDIA Parallel Nsight

The first development environment for massively parallel applications.

- Hardware GPU Source Debugging
- Platform-wide Analysis
- Complete Visual Studio integration



http://developer.nvidia.com/object/nsight.html

CUDA Webinars

Offering Specialized Training from Introductory to Advanced Levels

- Over 5000 students reached in 2009
- Webinars are scheduled for at least once a month
- Easily accessible current course on CUDA, CUDA optimization and OpenCL
- New Webinars have been hosted for Parallel Nsight and scheduled PGI Fortran
- For schedules: http://developer.nvidia.com/object/gpu_computing_online.html

Partner Call To Action

- Encourage customers who request training to attend webinars
- For strategic customers special webinars could be hosted
- Onsite training only in very special cases

Recommended Books