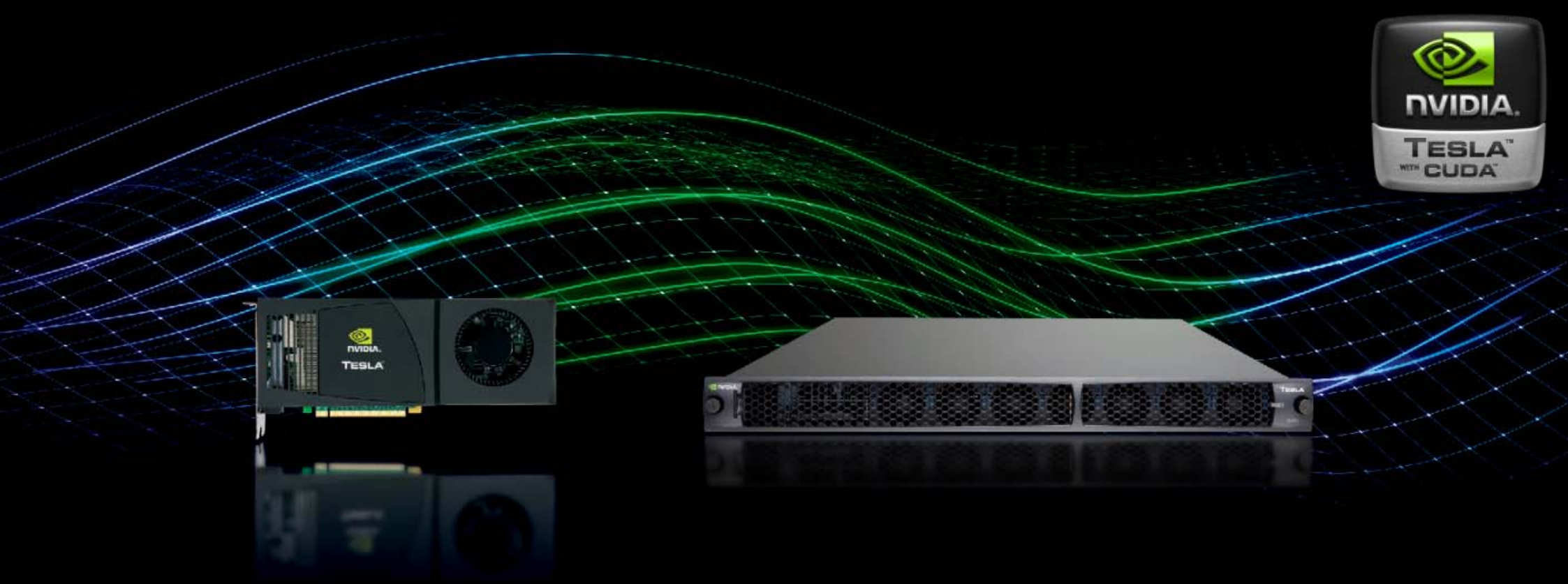


SIGGRAPH Asia 2009 GPU Computing master Class

Future Direction in GPU Computing

Toru Baji / NVIDIA Solution Architect

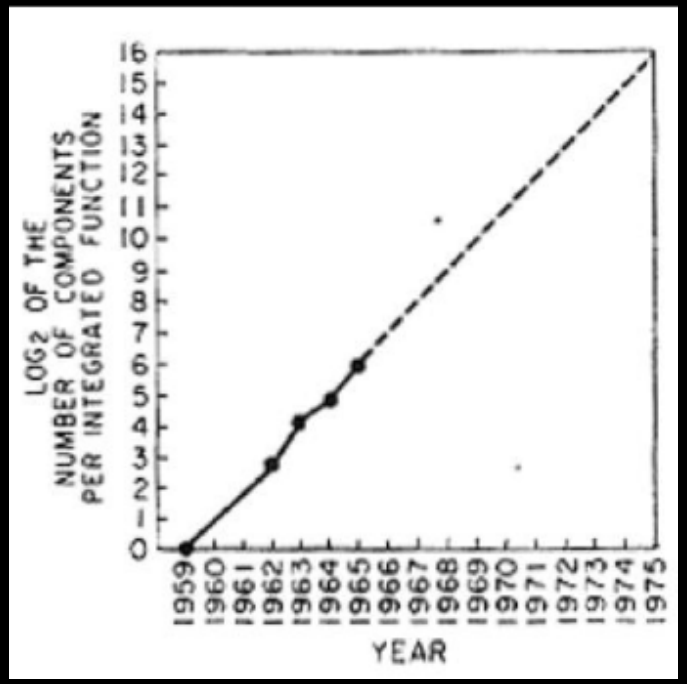
16th Dec., 2009

Content

- Why GPU Computing?
  - Single-threaded CPU performance is no longer scaling
  - CUDA can exploit “ Performance = Parallelism”
- Evolution of NVIDIA GPU
- Key Milestone in GPU Computing ‘Fermi’
  - Processor / Memory Architecture
  - Enhanced Thread Management
  - CUDA3.0
  - Integrated Development Environment ‘Nexus’
- An NVIDIA ExaScale Machine in 2017
- Conclusion

Single-threaded CPU performance is no longer scaling (1)

Moore's Law



Moore, Electronics 38(8) April, 1965

- In 1965, Gordon Moore predicted the number of Tr will grow at the rate of:

  x 2 / year

  later revised to

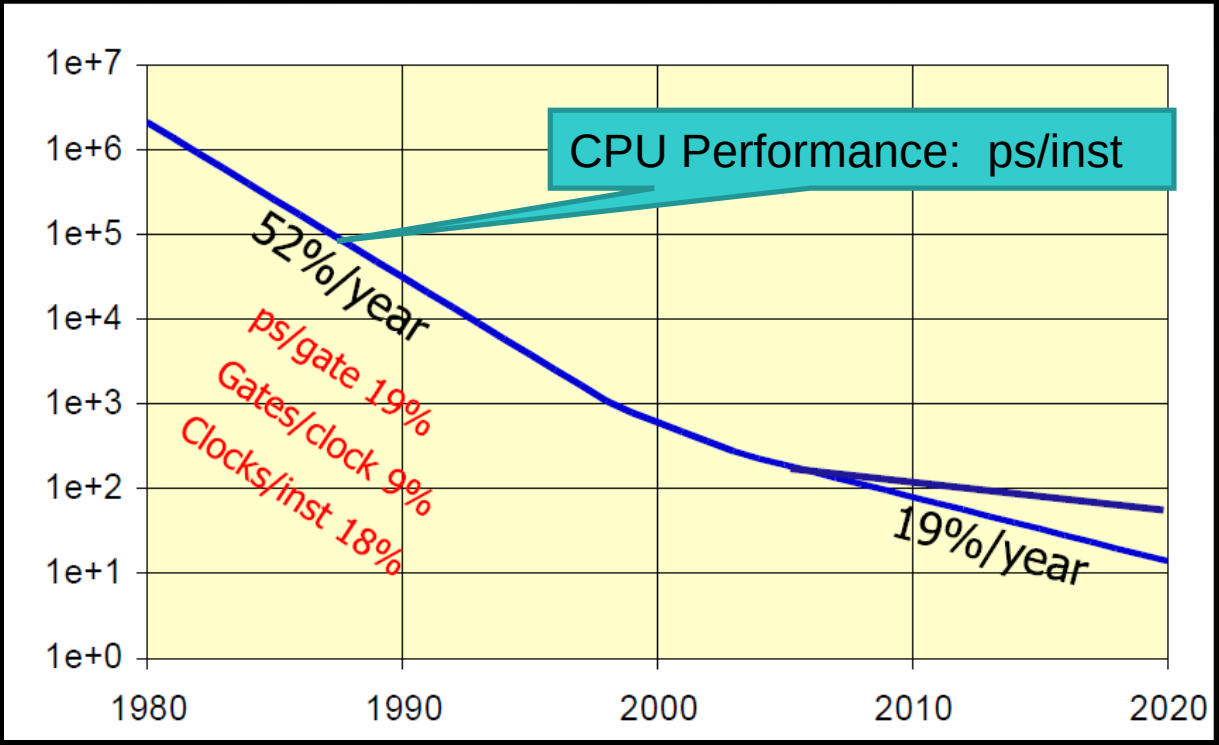
  x 2 / 18-months

- No prediction of CPU performance

Single-threaded CPU performance is no longer scaling (2)

The End of ILP (Instruction Level Parallelism) Scaling

- The increase in Tr counts was used to increasing the parallelism of the CPU instruction execution (e.g. superscalar, pipeline) and other speed-ups



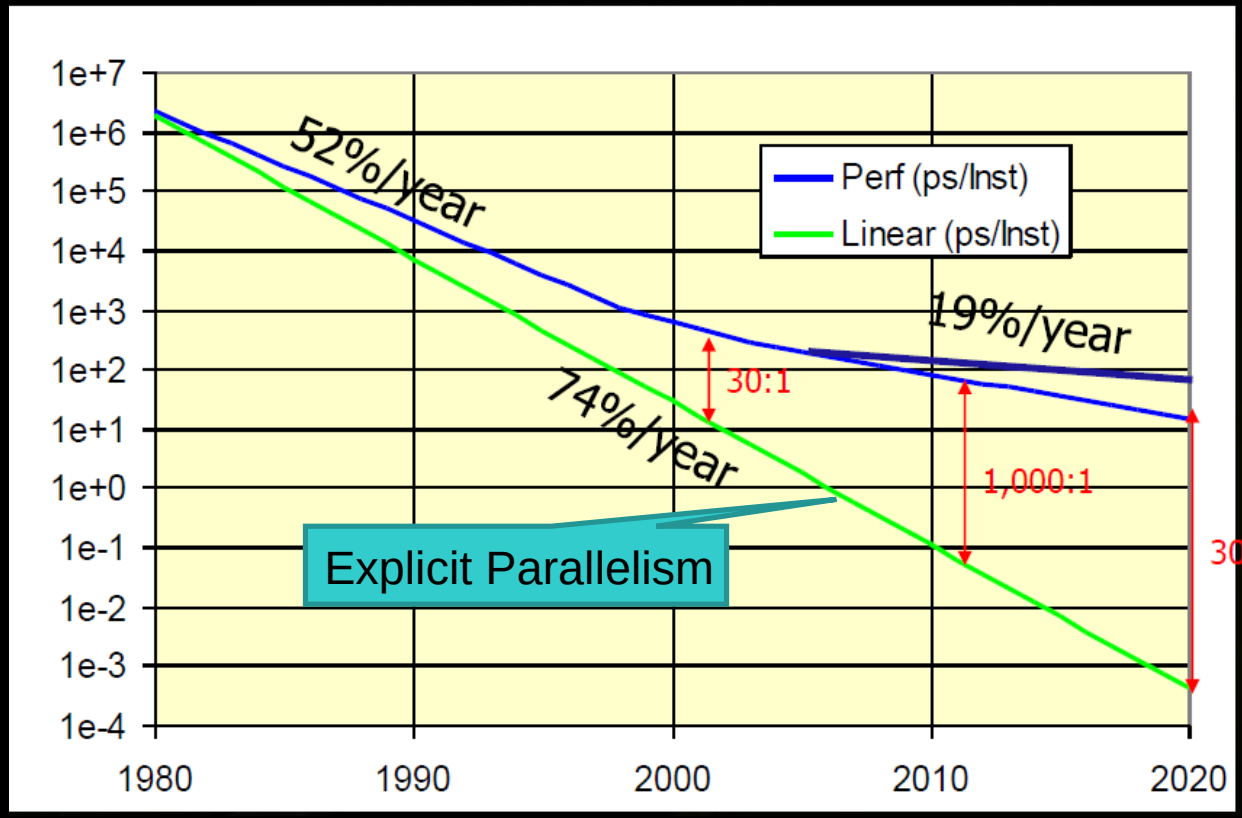
Performance is measured as the time required to execute one instruction (ps/inst)

Bill Dally et al., the Last Classical Computer, ISAT Study, 2001

Single-threaded CPU performance is no longer scaling (3)

Explicit Parallelism is Now Attractive

- The increase in Tr counts is used to increasing the number of processors



Bill Dally et al., the Last Classical Computer, ISAT Study, 2001

CUDA can exploit Performance = Parallelism

Now it is clear that " Performance = Parallelism"

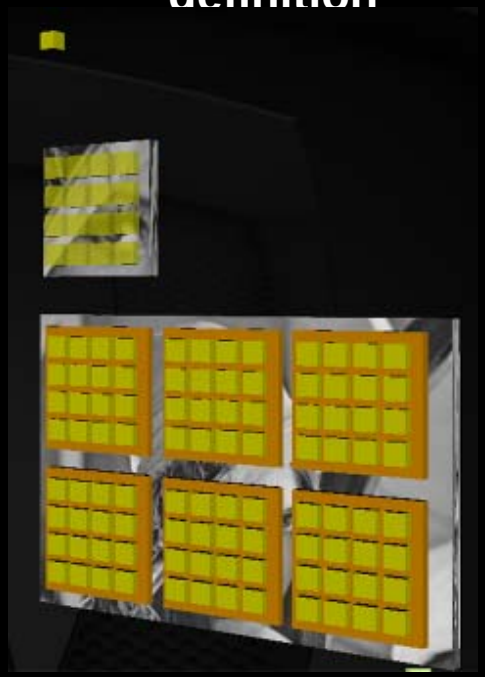
- **Then what is required to exploit this Parallelism?**
  - **Many efficient processors**
  - **A programming system that abstract the parallelism**

- **"CUDA (Compute Unified Device Architecture"**
  - **GPU has hundreds of full-featured processors**
  - **Multi-threading architecture use the best of those massive parallel cores**
  - **CUDA starts from extension to the familiar C-language, and now expanded to Fortran, OpenCL and direct Compute**
  - **CUDA can isolate the programmer from the details of parallel programming**

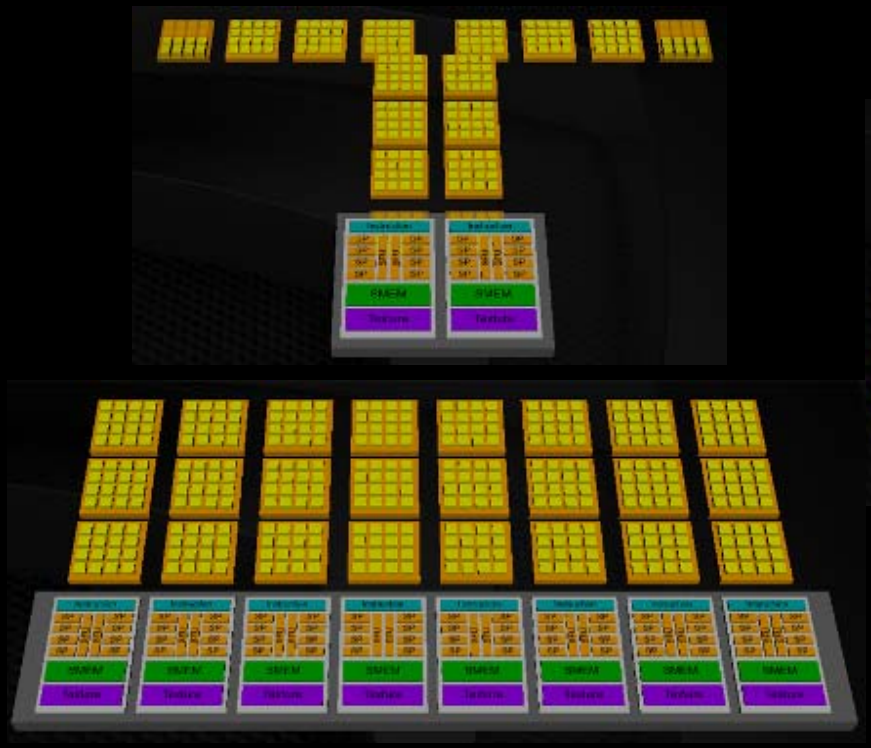
CUDA GPU Computing Visual Demo

separate animation tool used

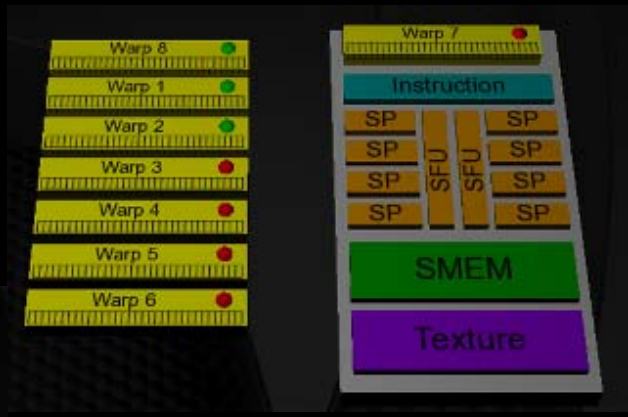
Hierarchical thread definition



Scalability to n-SMs



Hide memory Latency



Ease of Programming and Performance Increase

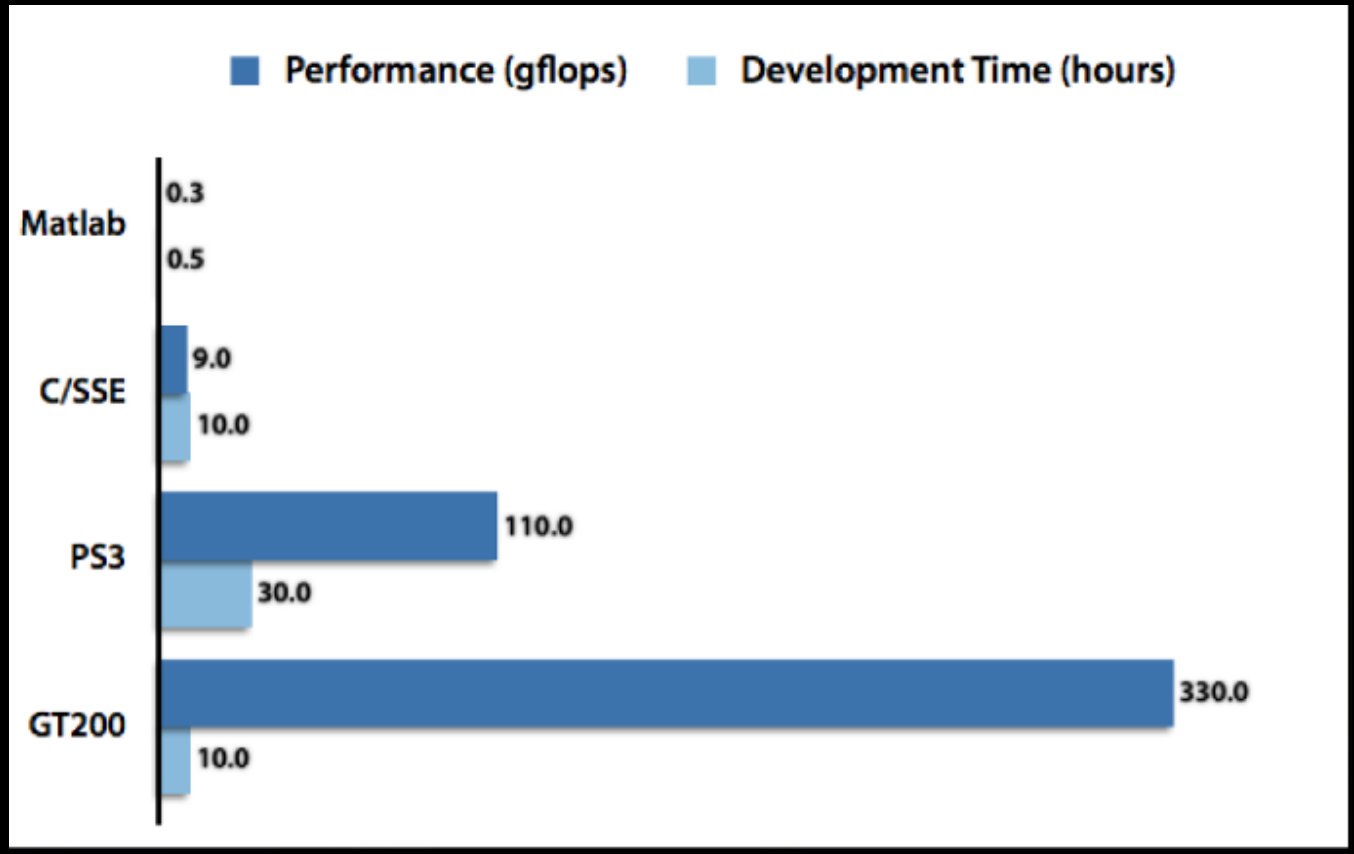
- **Ease of programming represented as “Development time”**
- **Performance achievement measured in GFLOPS**

High-level algorithm Development tool on CPU

CPU & SSE-SIMD

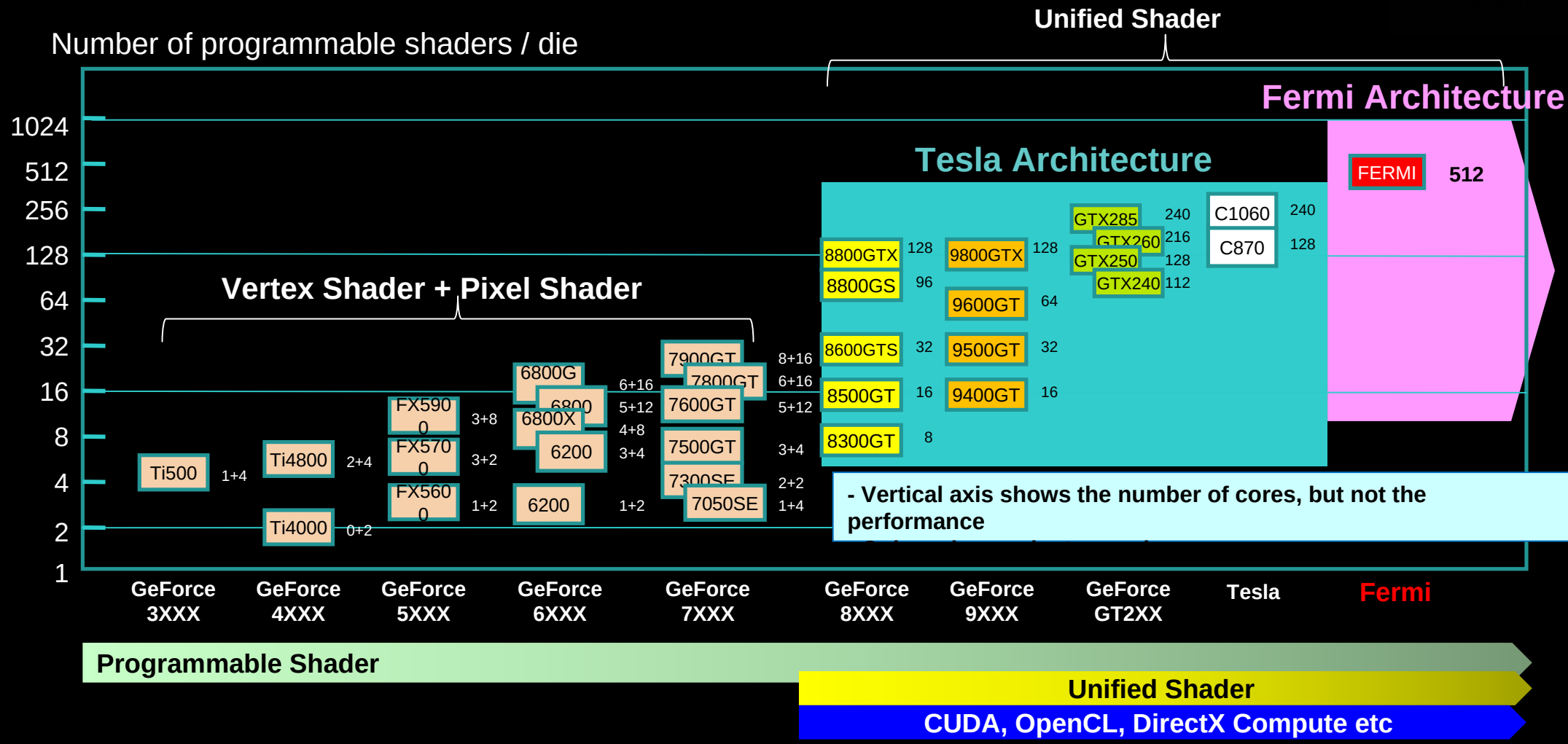
Cell Processor

GPU with CUDA

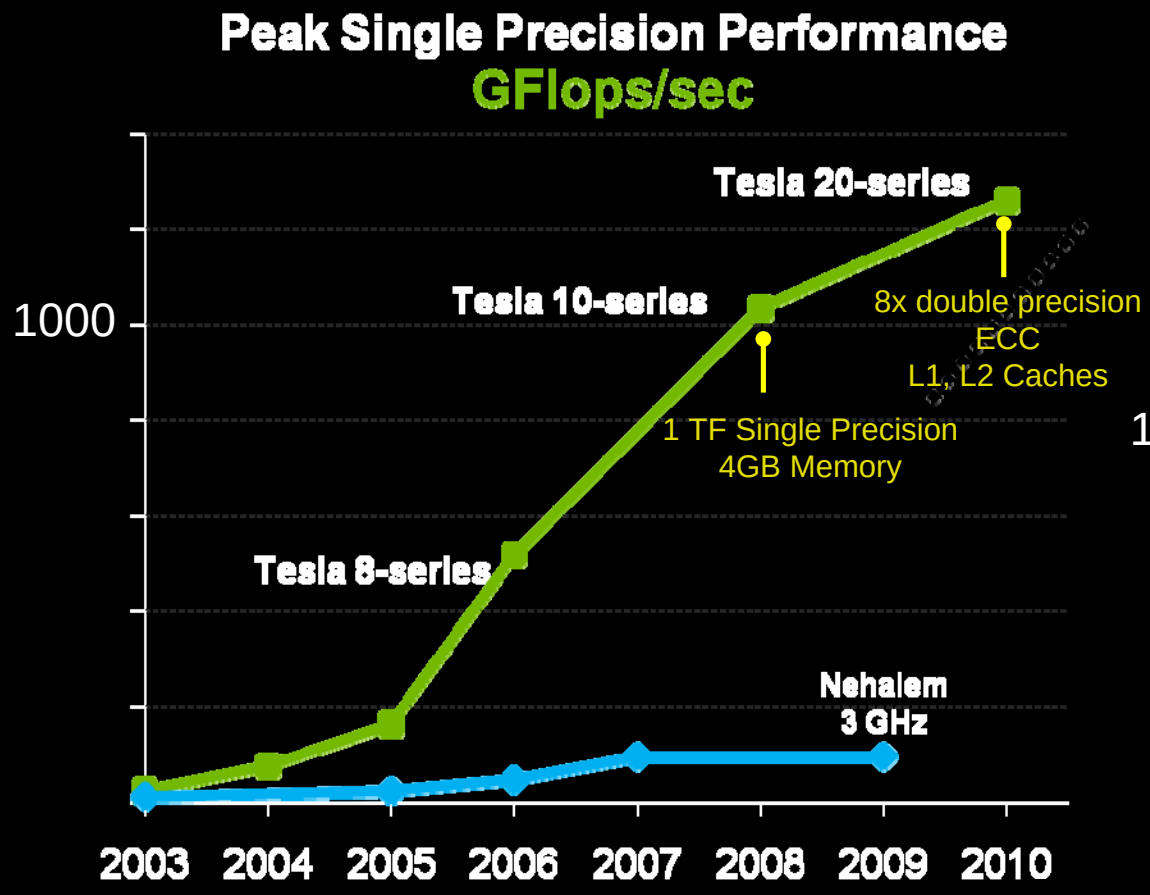


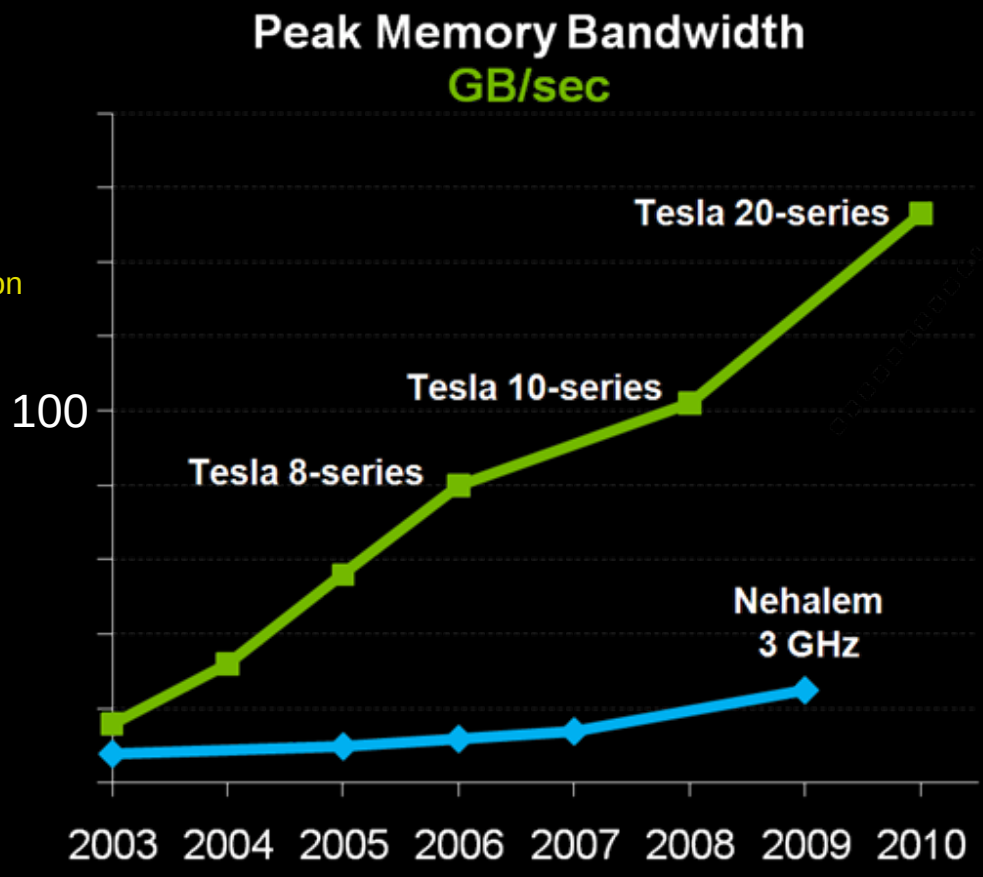
Source: Nicolas Pinto, MIT

Evolution of NVIDIA GPU - Number of Cores



Gap of GPU-CPU Performance, Mem-BW is growing

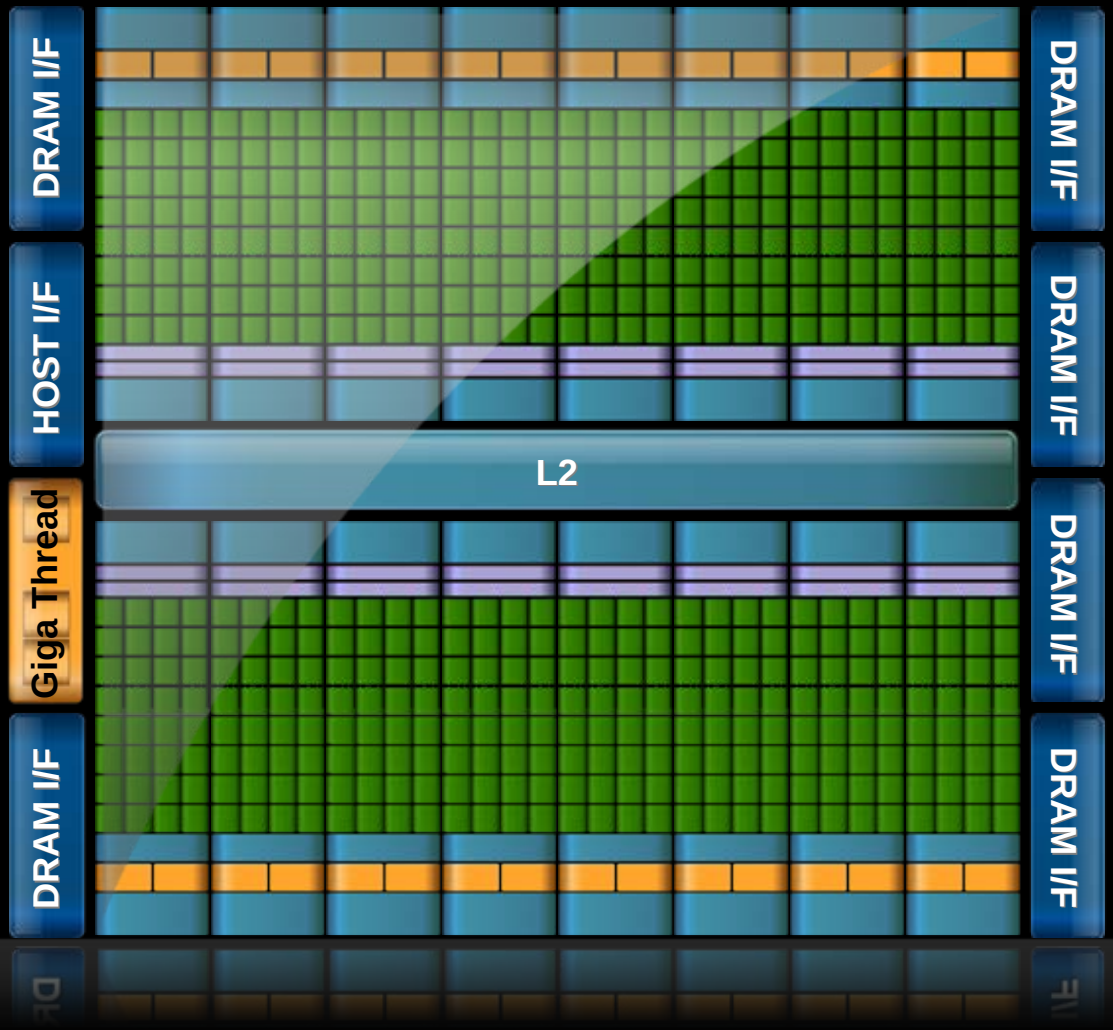




NVIDIA GPU
X86 CPU

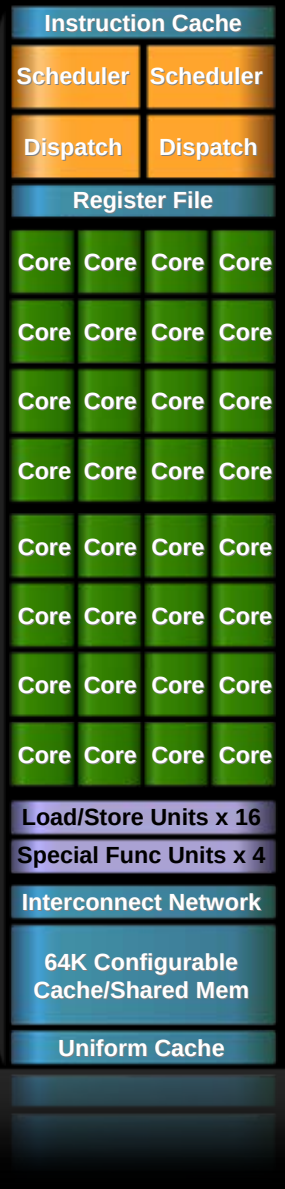
Key Milestone in GPU Computing 'Fermi'

- Improved peak performance
- Improved throughput efficiency
- Broader applicability
- Full integration within modern software development environment



SM Multiprocessor Architecture

- 32 CUDA cores per SM (512 total)
- 8x peak FP64 performance
  - 50% of peak FP32 performance
- Dual Thread Scheduler
- 64 KB of RAM for shared memory and L1 cache (configurable)



	FP32	FP64	INT	SFU	LD/ST
Ops / clk	32	16	32	4	16

IEEE 754-2008 Floating Point

- **IEEE 754-2008 results**
  - 64-bit double precision
  - 32-bit single precision
  - full-speed denormal operands & results
  - NaNs, +/- Infinity
- **IEEE 754-2008 rounding**
  - nearest even, zero, +inf, -inf
- **IEEE 754-2008 Fused Multiply-Add (FMA)**
  - D = A*B + C;
  - No loss of precision
  - IEEE divide & sqrt use FMA

Multiply-Add (MAD): D = A*B + C;

A × B

= Product (truncate digits)

+ C

= D

Fused Multiply-Add (FMA): D = A*B + C;

A × B

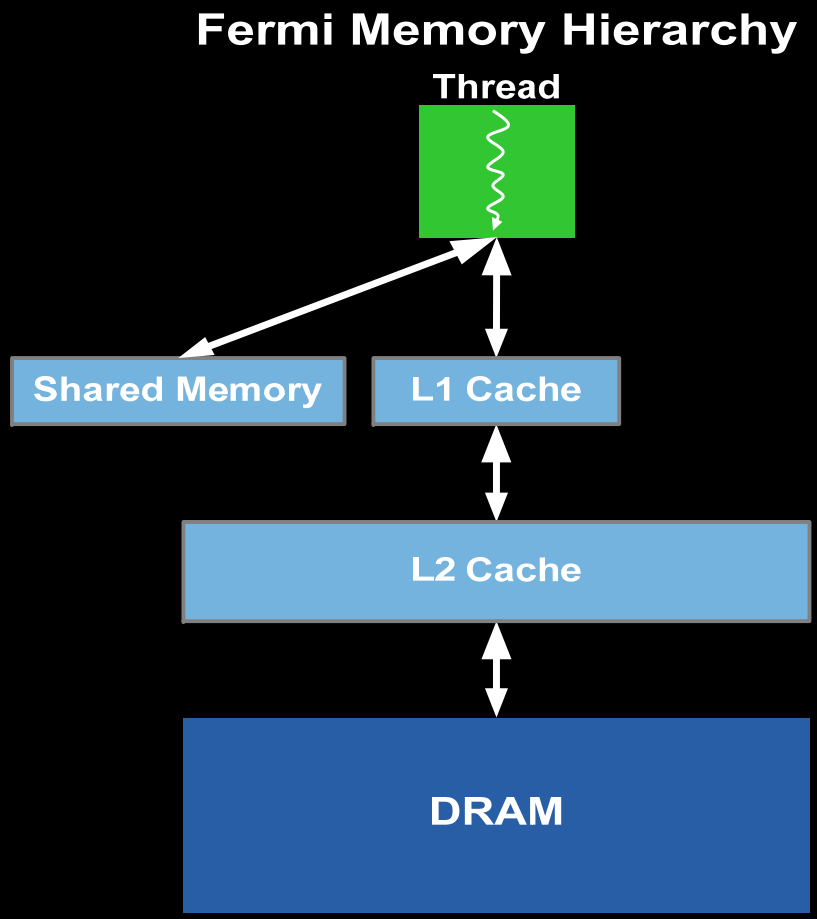
= Product (retain all digits)

+ C

= D (no loss of precision)

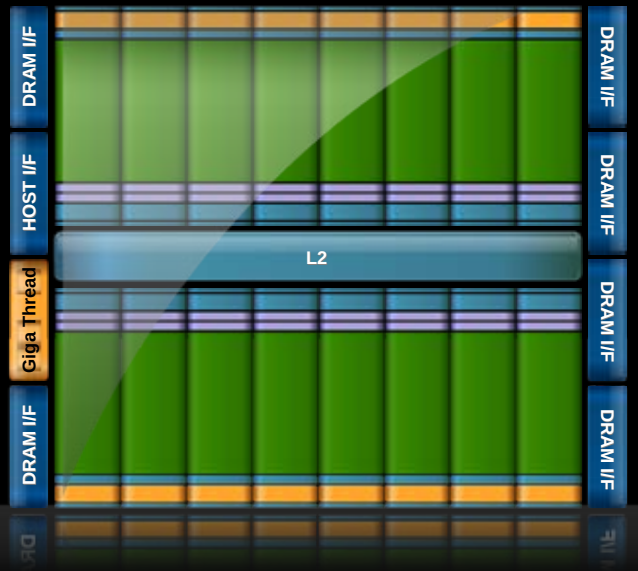
Cached Memory Hierarchy

- Configurable L1 cache per SM
  - 16KB L1$ / 48KB Shared Memory
  - 48KB L1$ / 16KB Shared Memory
- Shared 768KB L2 cache
- Motivation:
  - Caching captures locality, amplifies bandwidth
  - Caching more effective than Shared Memory RAM for irregular or unpredictable access
    - Ray tracing, sparse matrix multiply, physics kernels …
  - Caching helps latency sensitive cases

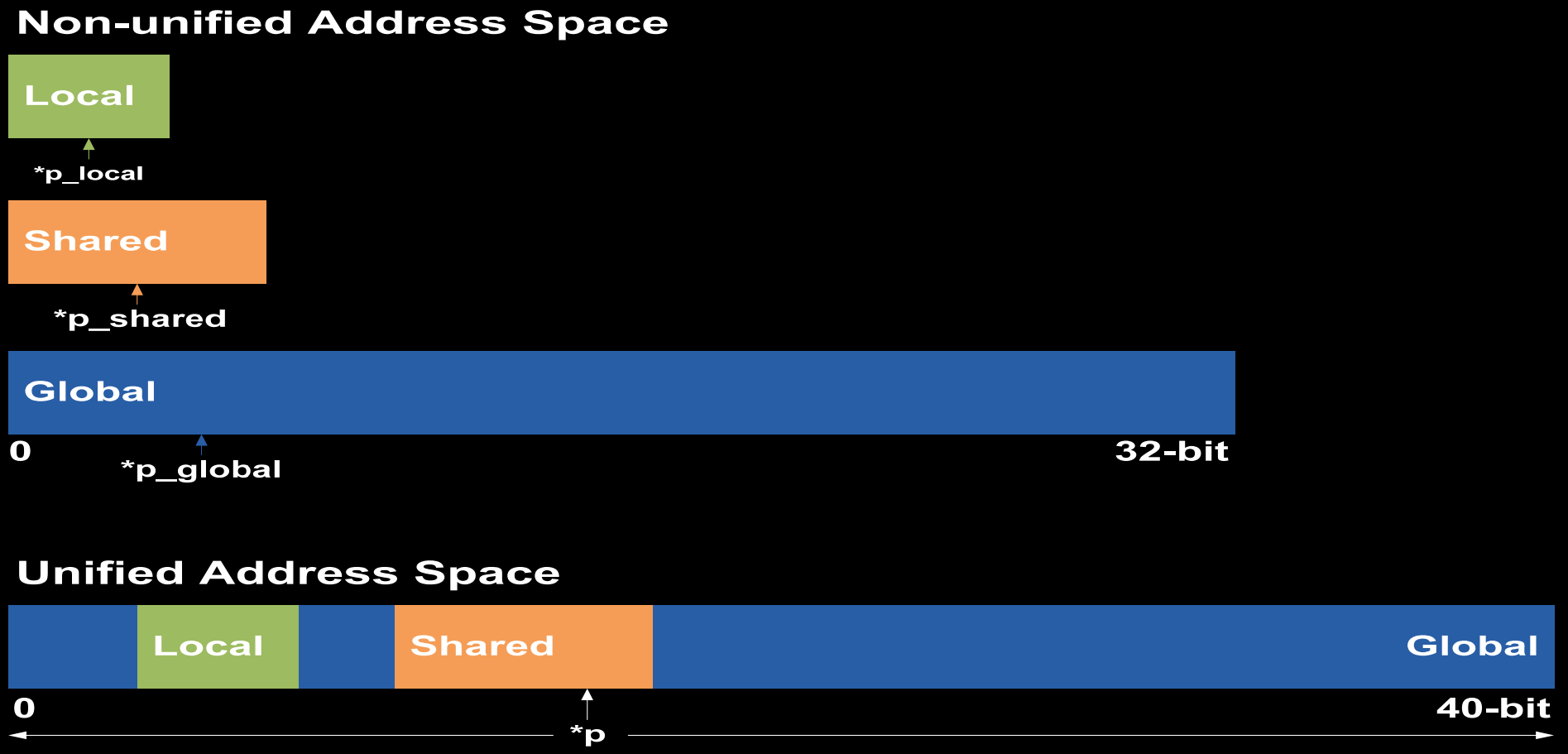


Larger, Faster Memory Interface

- GDDR5 memory interface
  - 2x improvement in peak speed over GDDR3
- Up to 1 Terabyte of memory attached to GPU
  - Operate on large data sets



Unified Load/Store Addressing

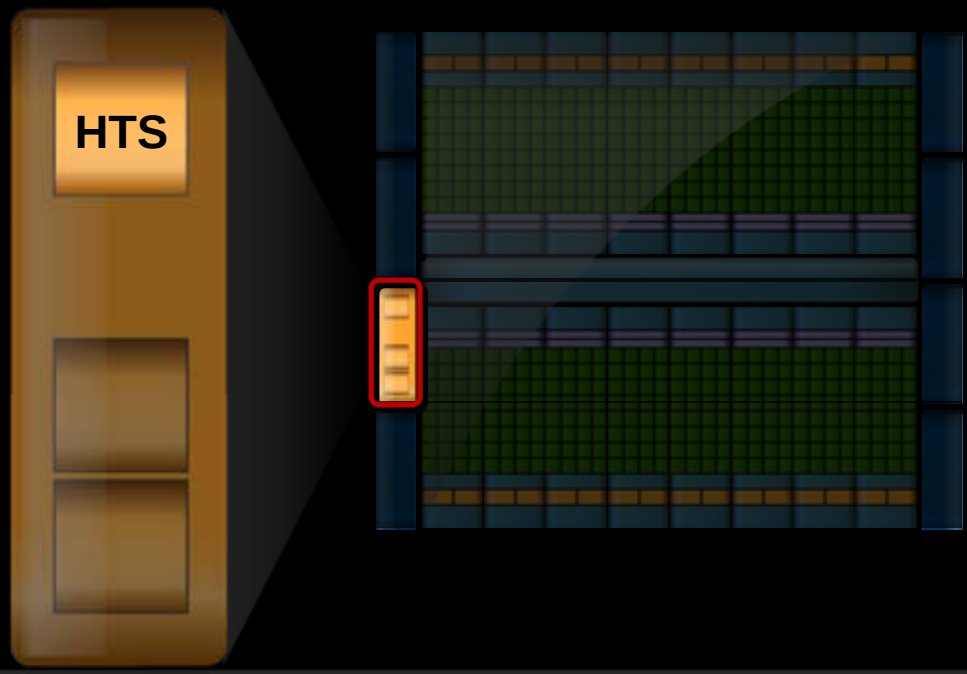


ECC Memory Protection

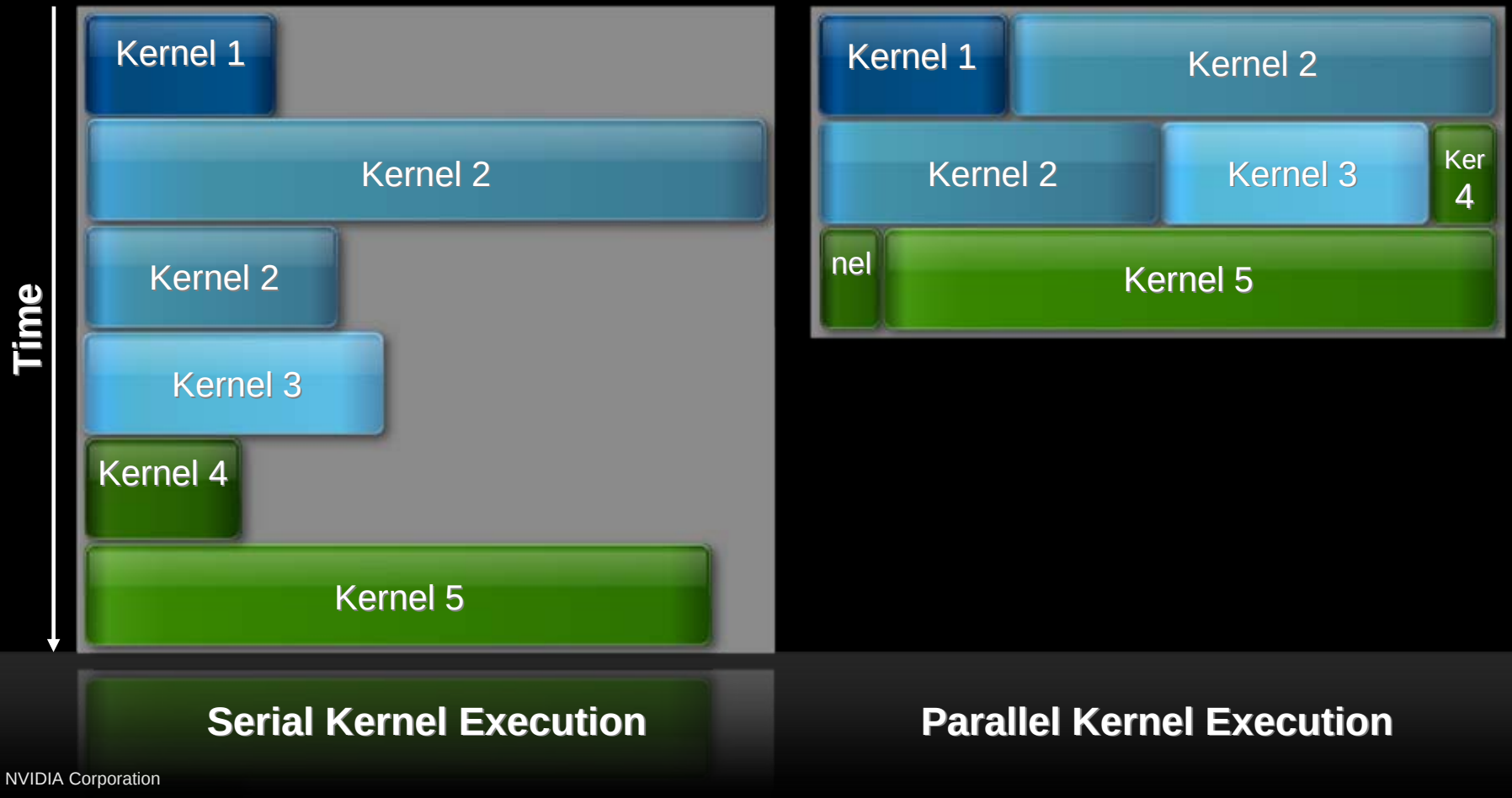
- **All major internal memories protected by ECC**
  - Register file
  - L1 cache
  - L2 cache
- **External DRAM protected by ECC**
- **ECC is a must have for many computing applications**
  - Clear customer feedback

GigaThread™ Hardware Thread Scheduler

- Hierarchically manages tens of thousands of simultaneously active threads
- 10x faster context switching on Fermi
- Overlapping kernel execution

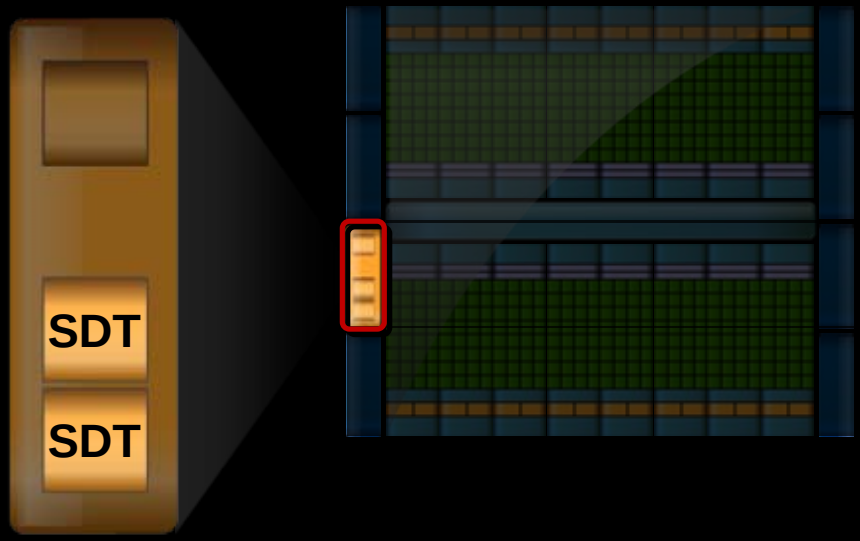


Overlapping Kernel Execution

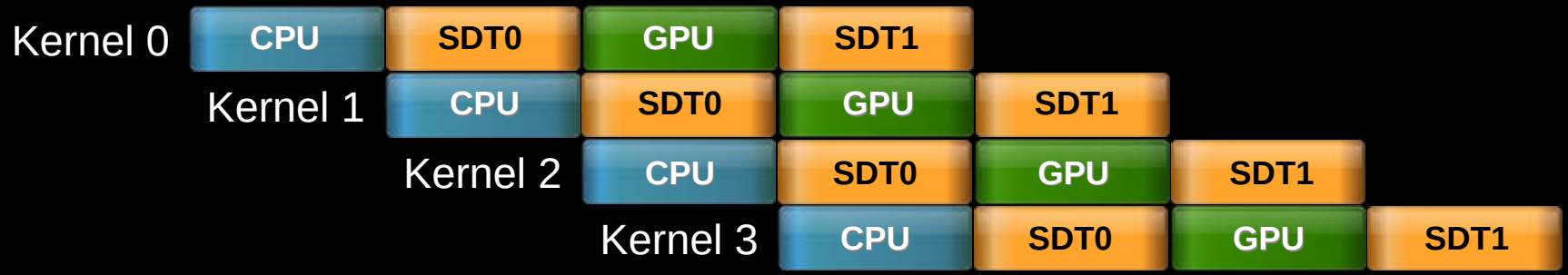


GigaThread Streaming Data Transfer Engine

- **Dual DMA engines**
- **Simultaneous CPU→GPU and GPU→CPU data transfer**

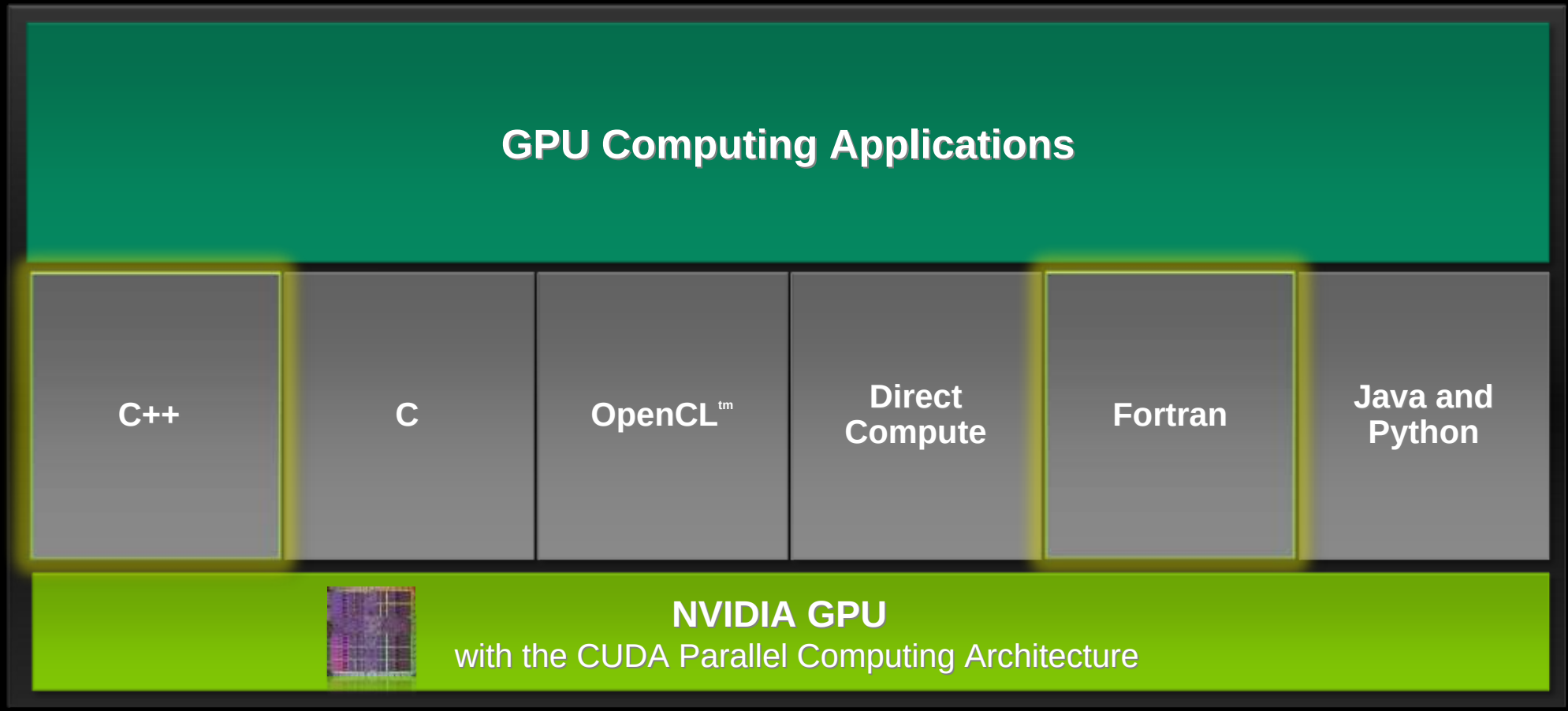


- **Fully overlapped with CPU/GPU processing**



	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point	-	30 FMA ops/clock	256 FMA ops/clock
Single Precision Floating Point	128 MAD ops/clock	240 MAD ops/clock	512 FMA ops/clock
Special Function Units (per SM)	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48/16 KB
L1 Cache (per SM)	-	-	Configurable 16/48 KB
L2 Cache	-	-	768 KB
ECC Memory Support	-	-	Yes
Concurrent Kernels	-	-	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

CUDA Parallel Computing Architecture



CUDA C 3.0

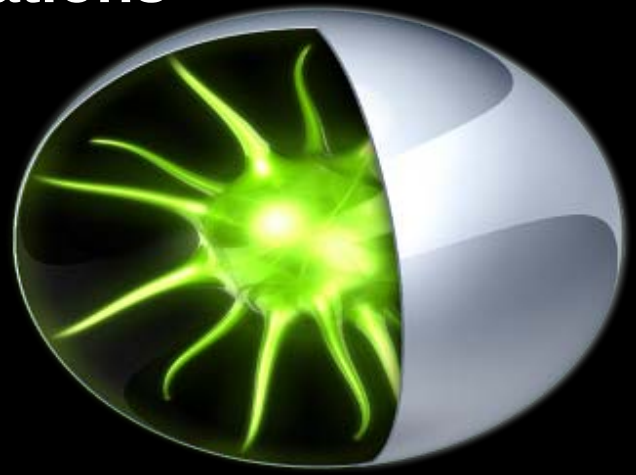
- **Unified addressing for C and C++ pointers**
- Global, shared, local addresses
- Enables 3rd party GPU callable libraries, dynamic linking
- One 40-bit address space for load/store instructions

- **Compiling for native 64-bit addressing**

- **IEEE 754-2008 single & double precision**
- C99 math.h support

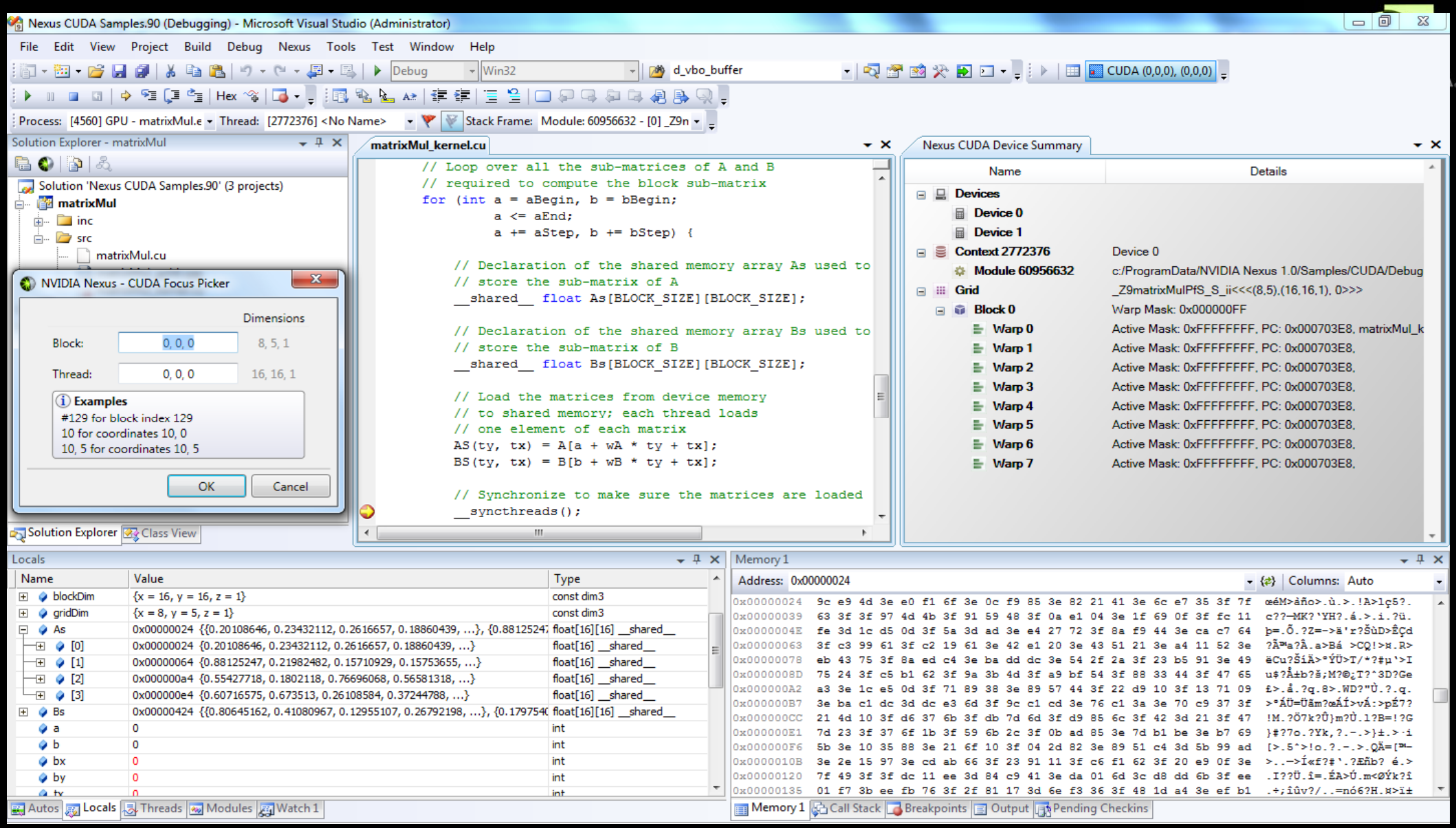
NVIDIA Integrated Development Environment Code-named ‘Nexus’

- Industry’s 1st IDE for massively parallel applications



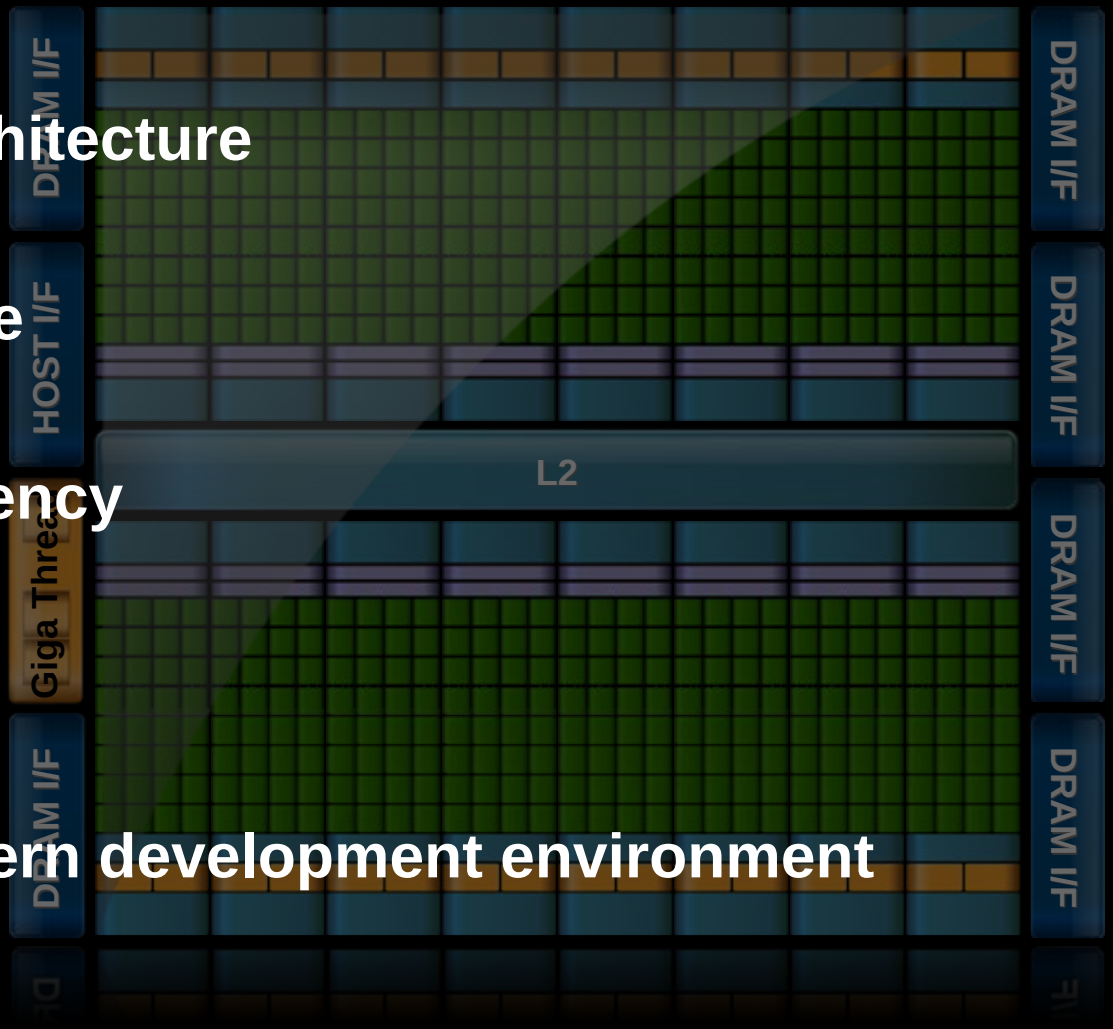
- Accelerate development of CPU + GPU co-processing applications
- Complete Visual Studio-integrated development environment
  - C, C++, OpenCL, & DirectCompute platforms
  - DirectX and OpenGL graphics APIs

Microsoft Visual Studio PARTNER


Nexus CUDA Samples.90 (Debugging) - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Nexus Tools Test Window Help
Debug
Win32
d_vbo_buffer
CUDA (0,0,0), (0,0,0)
Process: [4560] GPU - matrixMul.e
Thread: [2772376] <No Name>
Stack Frame: Module: 60956632 - [0] _Z9n
Solution Explorer - matrixMul
Solution 'Nexus CUDA Samples.90' (3 projects)
matrixMul
inc
src
matrixMul.cu
NVIDIA Nexus - CUDA Focus Picker
Dimensions
Block: 0, 0, 0 8, 5, 1
Thread: 0, 0, 0 16, 16, 1
Examples
#129 for block index 129
10 for coordinates 10, 0
10, 5 for coordinates 10, 5
OK
Cancel
Solution Explorer
Class View
matrixMul_kernel.cu
// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin, b = bBegin;
a <= aEnd;
a += aStep, b += bStep) {
// Declaration of the shared memory array As used to
// store the sub-matrix of A
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
// Declaration of the shared memory array Bs used to
// store the sub-matrix of B
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
// Load the matrices from device memory
// to shared memory; each thread loads
// one element of each matrix
AS(ty, tx) = A[a + wA * ty + tx];
BS(ty, tx) = B[b + wB * ty + tx];
// Synchronize to make sure the matrices are loaded
__syncthreads();
Nexus CUDA Device Summary
Name
Details
Devices
Device 0
Device 1
Context 2772376
Device 0
Module 60956632
c:/ProgramData/NVIDIA Nexus 1.0/Samples/CUDA/Debug
Grid
_Z9matrixMulPfS_S_ii<<<(8,5),(16,16,1), 0>>>
Block 0
Warp Mask: 0x000000FF
Warp 0
Active Mask: 0xFFFFFFFF, PC: 0x000703E8, matrixMul_k
Warp 1
Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 2
Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 3
Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 4
Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 5
Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 6
Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 7
Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Locals
Name
Value
Type
blockDim {x = 16, y = 16, z = 1} const dim3
gridDim {x = 8, y = 5, z = 1} const dim3
As 0x00000024 {{0.20108646, 0.23432112, 0.2616657, 0.18860439, ...}, {0.88125247 float[16][16] __shared__
[0] 0x00000024 {0.20108646, 0.23432112, 0.2616657, 0.18860439, ...} float[16] __shared__
[1] 0x00000064 {0.88125247, 0.21982482, 0.15710929, 0.15753655, ...} float[16] __shared__
[2] 0x000000a4 {0.55427718, 0.1802118, 0.76696068, 0.56581318, ...} float[16] __shared__
[3] 0x000000e4 {0.60716575, 0.673513, 0.26108584, 0.37244788, ...} float[16] __shared__
Bs 0x00000424 {{0.80645162, 0.41080967, 0.12955107, 0.26792198, ...}, {0.1797540 float[16][16] __shared__
a 0 int
b 0 int
bx 0 int
by 0 int
Autos Locals Threads Modules Watch 1
Memory 1
Address: 0x00000024
Columns: Auto
0x00000024 9c e9 4d 3e e0 f1 6f 3e 0c f9 85 3e 82 21 41 3e 6c e7 35 3f 7f
0x00000039 63 3f 3f 97 4d 4b 3f 91 59 48 3f 0a e1 04 3e 1f 69 0f 3f fc 11
0x0000004E fe 3d 1c d5 0d 3f 5a 3d ad 3e e4 27 72 3f 8a f9 44 3e ca c7 64
0x00000063 3f c3 99 61 3f c2 19 61 3e 42 e1 20 3e 43 51 21 3e a4 11 52 3e
0x00000078 eb 43 75 3f 8a ed c4 3e ba dd dc 3e 54 2f 2a 3f 23 b5 91 3e 49
0x0000008D 75 24 3f c5 b1 62 3f 9a 3b 4d 3f a9 bf 54 3f 88 33 44 3f 47 65
0x000000A2 a3 3e 1c e5 0d 3f 71 89 38 3e 89 57 44 3f 22 d9 10 3f 13 71 09
0x000000B7 3e ba c1 dc 3d dc e3 6d 3f 9c c1 cd 3e 76 c1 3a 3e 70 c9 37 3f
0x000000CC 21 4d 10 3f d6 37 6b 3f db 7d 6d 3f d9 85 6c 3f 42 3d 21 3f 47
0x000000E1 7d 23 3f 37 6f 1b 3f 59 6b 2c 3f 0b ad 85 3e 7d b1 be 3e b7 69
0x000000F6 5b 3e 10 35 88 3e 21 6f 10 3f 04 2d 82 3e 89 51 c4 3d 5b 99 ad
0x0000010B 3e 2e 15 97 3e cd ab 66 3f 23 91 11 3f c6 f1 62 3f 20 e9 0f 3e
0x00000120 7f 49 3f 3f dc 11 ee 3d 84 c9 41 3e da 01 6d 3c d8 dd 6b 3f ee
0x00000135 01 f7 3b ee fb 76 3f 2f 81 17 3d 6e f3 36 3f 48 1d a4 3e ef b1
Memory 1 Call Stack Breakpoints Output Pending Checkins

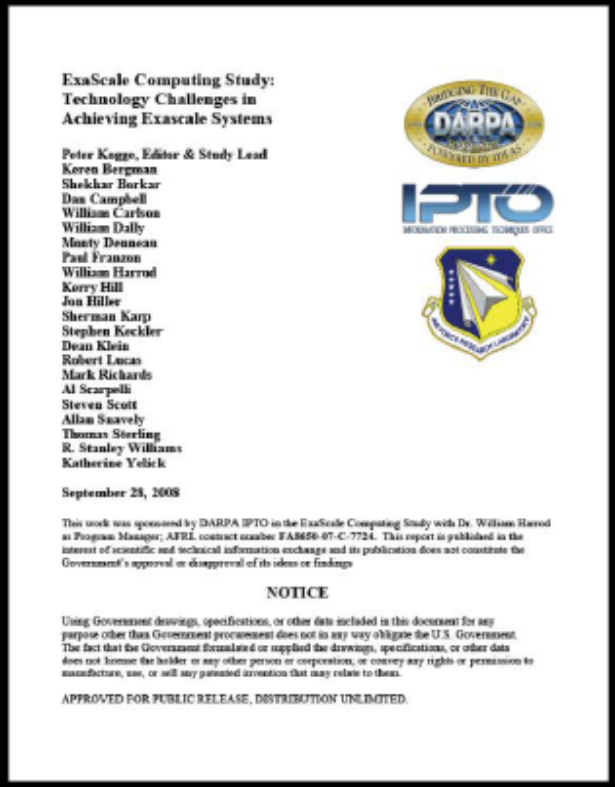
Fermi Summary

- Third generation CUDA architecture
- Improved peak performance
- Improved throughput efficiency
- Broader applicability
- Full integration within modern development environment



Future: ExaScale Computing

- ExaScale (10^18) is the next milestone following PetaScale (10^15) computing
- DARPA, HP, France and others are planning for ExaScale computing facilities.
- DARPA report (28th Sept., 2008) described four major challenges (Power consumption, memory, Concurrency/Locality, Resiliency)
- DARPA reports that the number one issue is Power. Extrapolation of Power indicates over 100MW for Exaflop.



ExaScale Computing Study:
Technology Challenges in
Achieving Exascale Systems

Peter Kogge, Editor & Study Lead
Keren Bergman
Shekhar Borkar
Dan Campbell
William Carlson
William Dally
Monty Denneau
Paul Franzon
William Harrod
Kerry Hill
Jon Hiller
Sherman Karp
Stephen Keckler
Dean Klein
Robert Lucas
Mark Richards
Al Scarpelli
Steven Scott
Allan Snavely
Thomas Sterling
R. Stanley Williams
Katherine Yelick

September 28, 2008

This work was sponsored by DARPA IPTO in the ExaScale Computing Study with Dr. William Harrod as Program Manager; AFRL contract number FA8650-07-C-7724. This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED.

Available at
www.darpa.mil/personnel/docs/ExaScale_Study_Initial.pdf

A Possible NVIDIA ExaScale Machine in 2017

by Bill Dally: Chief Scientist & Sr. VP of Research, NVIDIA

A projection based on Moore's Law and does not represent a committed roadmap

- GPU Node (GPU + CPU + memory + supply) ~300W
  - 2,400 throughput-cores (7,200FPUs), 16 CPUs – single chip
  - 40TFLOPS (SP), 13TFLOPS (DP)
- Node Memory
  - 128GB DRAM, 2TB/s bandwidth
  - 512GB Phase-change Flash for checkpoint and scratch
- Cabinet ~100kW
  - 384-Nodes 15.7PFLOPS (SP), 50TB DRAM
  - Dragonfly network – 1TB/sec node bandwidth
- System ~ 10MW
  - 128 cabinets – 2 ExaFLOPS (SP), 6.8PB DRAM
  - Dragonfly network with active optical links

Conclusion

- Performance of Single-threaded processor is saturating
- Performance = Parallelism
- NVIDIA are committed to offer massive parallel GPUs for now and beyond
- CUDA GPU computing abstracts parallel programming
- ‘Fermi’ is the key milestone in GPU Computing


Thank you for your attention
NVIDIA