



Bimalee Salpitikorala
Thilina Gunarathne




GPGPU & ACCELERATORS

CPU

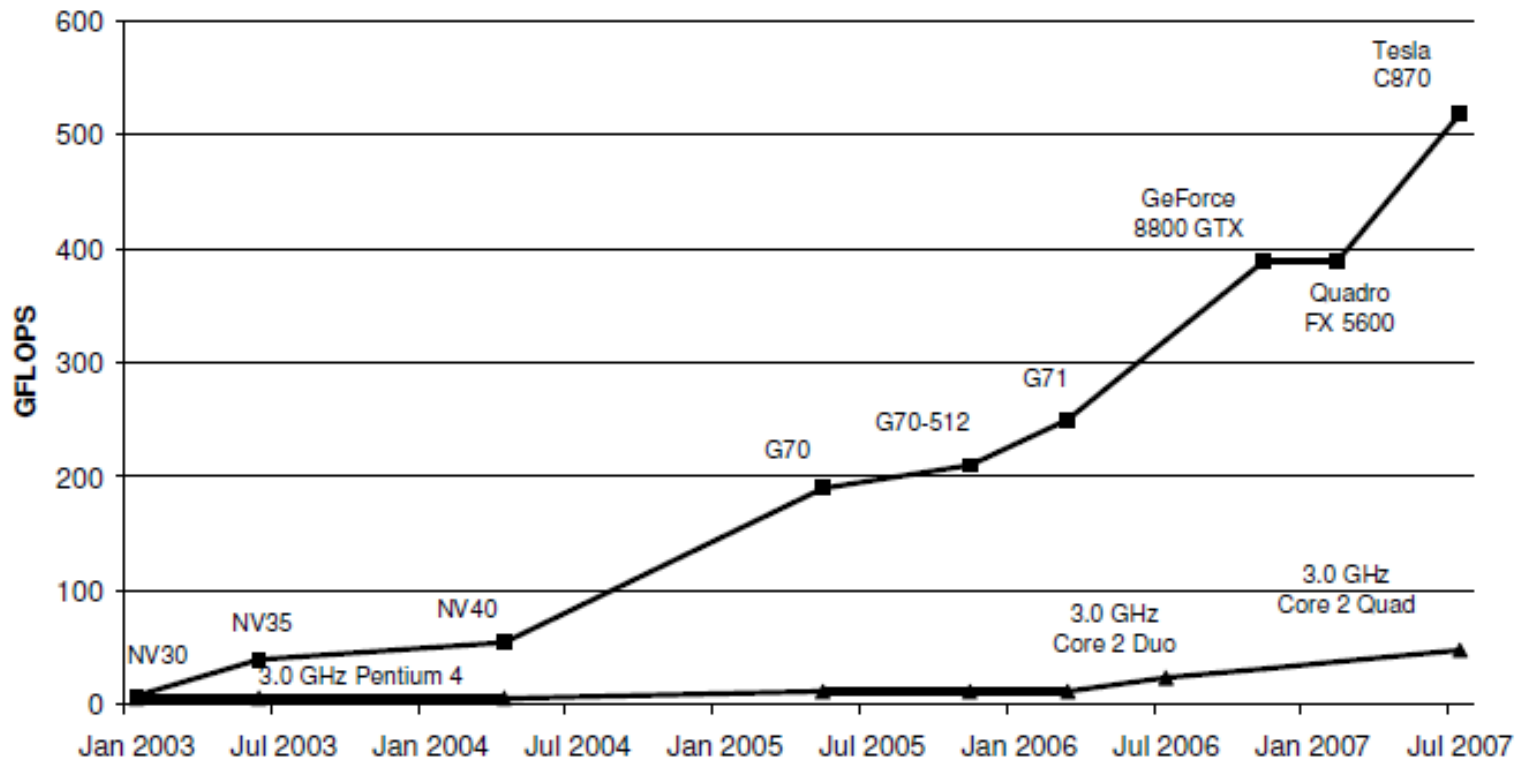
- Optimized for sequential performance
- Extracting Instruction level parallelism is difficult
 - Control hardware to check for dependencies, out-of-order execution, prediction logic etc
- Control hardware dominates CPU
 - Complex, difficult to build and verify
- Does not do actual computation but consumes power
- Pollack's Rule
 - Performance $\sim \sqrt{\text{area}}$
 - To increase sequential performance 2 times, area 4 times
 - Increase performance using 2 cores, area only 2 times



GPU

- High-performance many-core processors
 - Data Parallelized Machine -SIMD Architecture
 - Less control hardware
 - High computational performance
- 

GPU vs CPU GFLOPS graph



¹Based on slide 7 of S. Green, "GPU Physics," SIGGRAPH 2007 GPGPU Course. <http://www.gpgpu.org/s2007/slides/15-GPGPU-physics.pdf>

Figure 1.1. Enlarging Performance Gap between GPUs and CPUs.


GPU - Accelerator?

No, not this

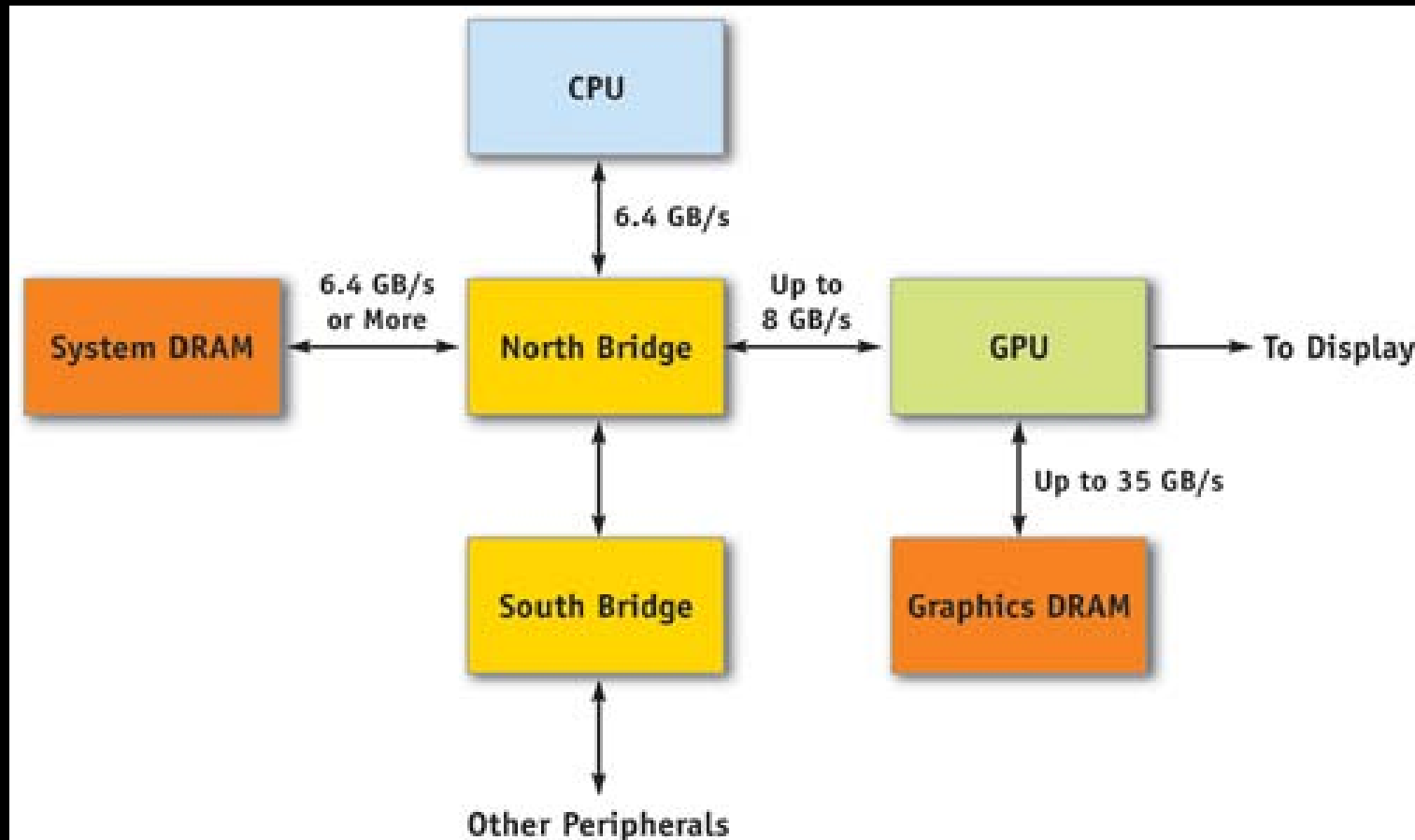




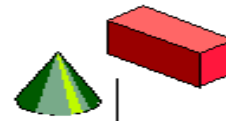
Accelerator?

- Speed up some aspect of the computing workload
 - Implemented as a *coprocessor* to the host-
 - Its own instruction set
 - Its own memory (usually but not always).
 - To the hardware - another IO unit
 - To the software – another computer
 - Today's accelerators
 - Sony/Toshiba/IBM Cell Broadband Engine
 - GPUs
- 

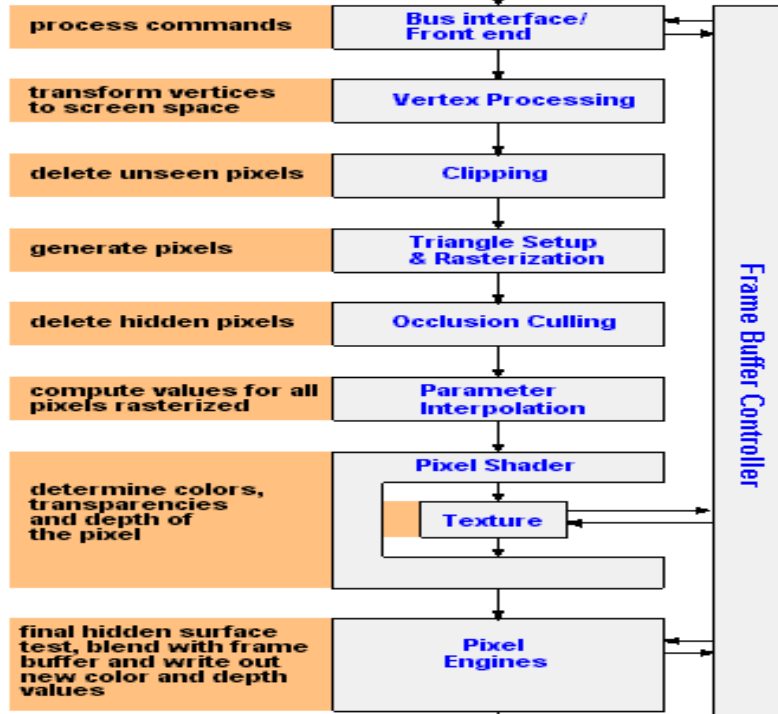
How the GPU Fits into the Overall Computer System



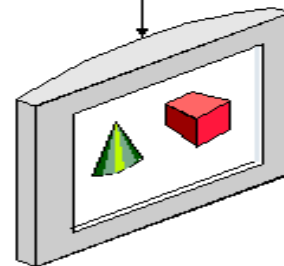
**3-D
OBJECTS
IN THE
GRAPHICS
APPLICATION**



**Peripheral Bus
(PCI, AGP, PCX)**

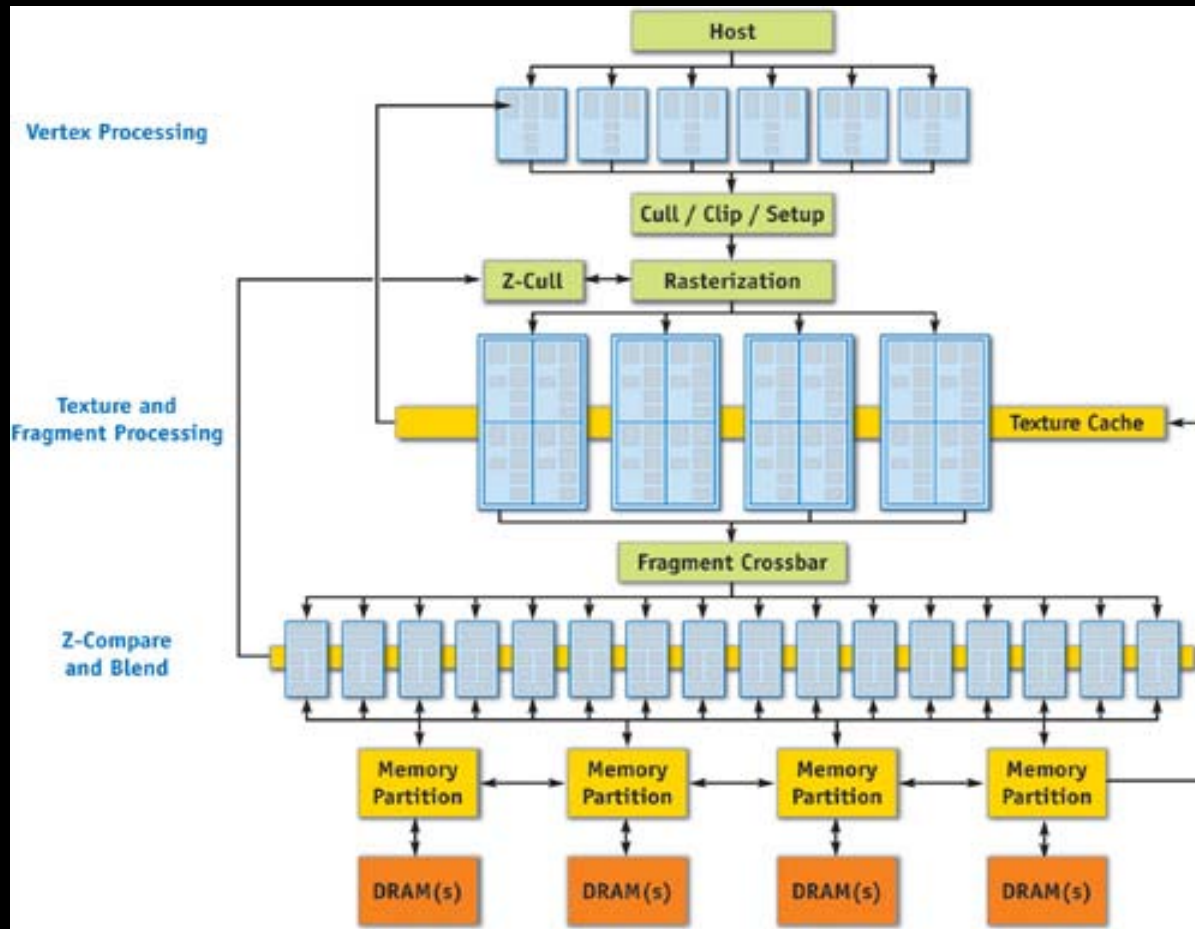


**2-D
IMAGE
ON
SCREEN**



Graphics pipeline

GPU Architecture




GPU Pipeline

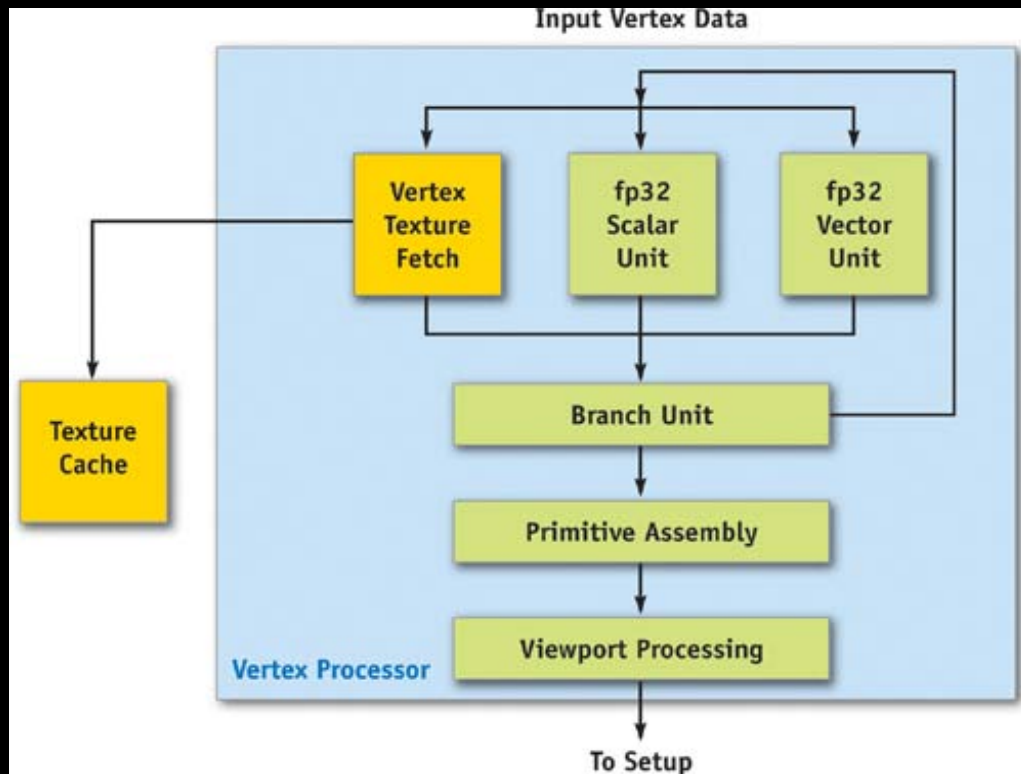
- CPU to GPU data transfer
- Vertex processing
- Cull / Clip / Set up
- Rasterization
- Texture & Fragment processing
- Compare & blend



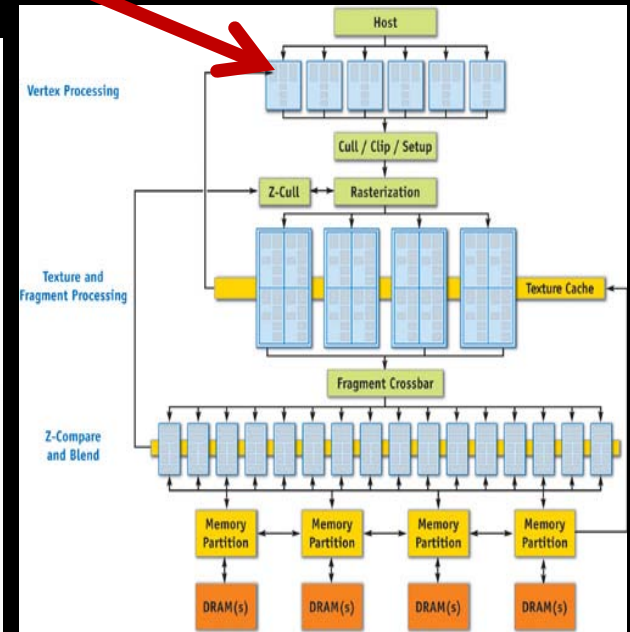
CPU to GPU data transfer

- Through a graphics connector
 - PCI Express
 - AGP slot on the motherboard
 - Graphics connector transfers
 - properties at the end points (vertices) or control points of the geometric primitives (lines and triangles).
 - The type of properties provided per vertex
 - x-y-z coordinates
 - RGB values
 - Texture
 - Reflectivity etc..
- 

Vertex processing




Vertex processor/vertex shaders






Cull/ Clip/ Set up

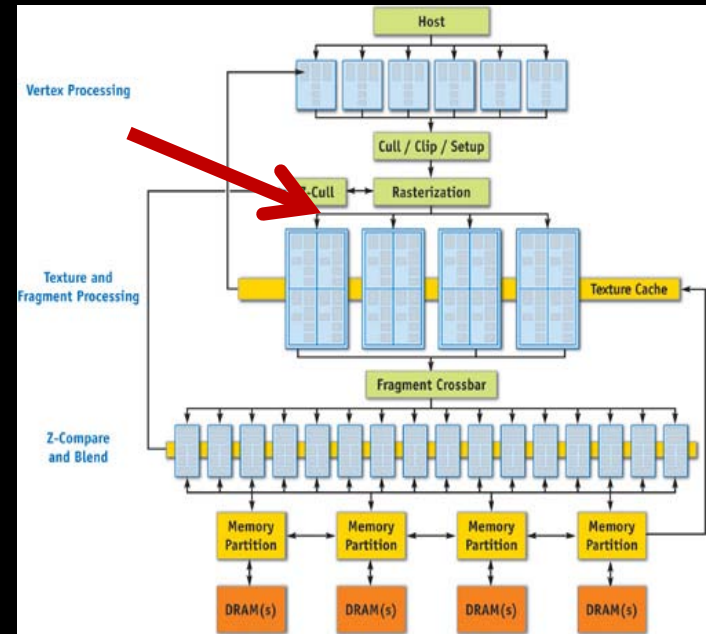
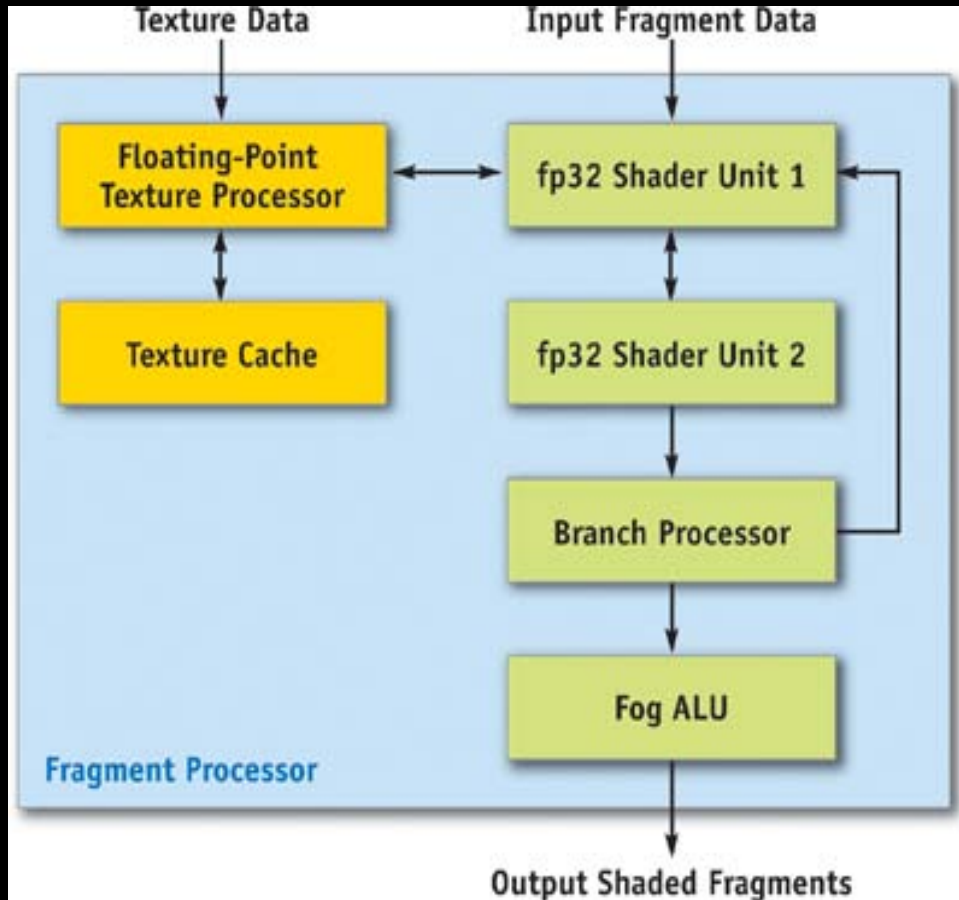
- Remove primitives that aren't visible
 - Clip primitives that intersect the view frustum
 - Performing edge and plane equation setup on the data in preparation for rasterization.
- 



Rasterization

- Filling primitives with pixels known as "fragments,"
 - Calculates which pixels are covered by each primitive.
 - Removes pixels that are hidden (occluded) by other objects in the scene.
 - Compute the value of pixels
 - Color
 - Fog
 - texture
- 


Texture & Fragment processing



Fragment Processor/ pixel shader



Memory

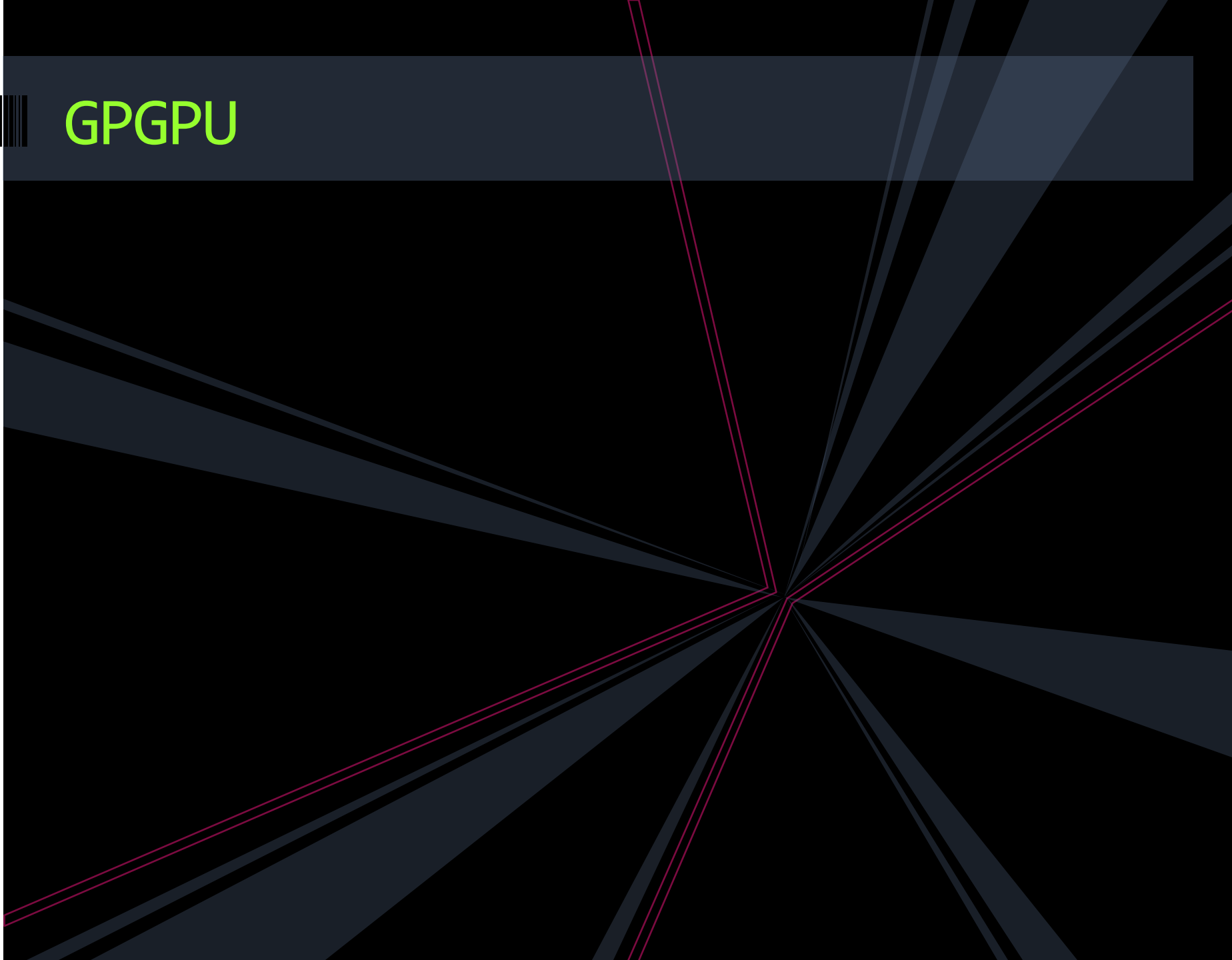
- Partitioned into up to four independent memory partitions each with its own dynamic random-access memories.
 - All rendered surfaces are stored in the DRAMs (Frame buffer)
- 

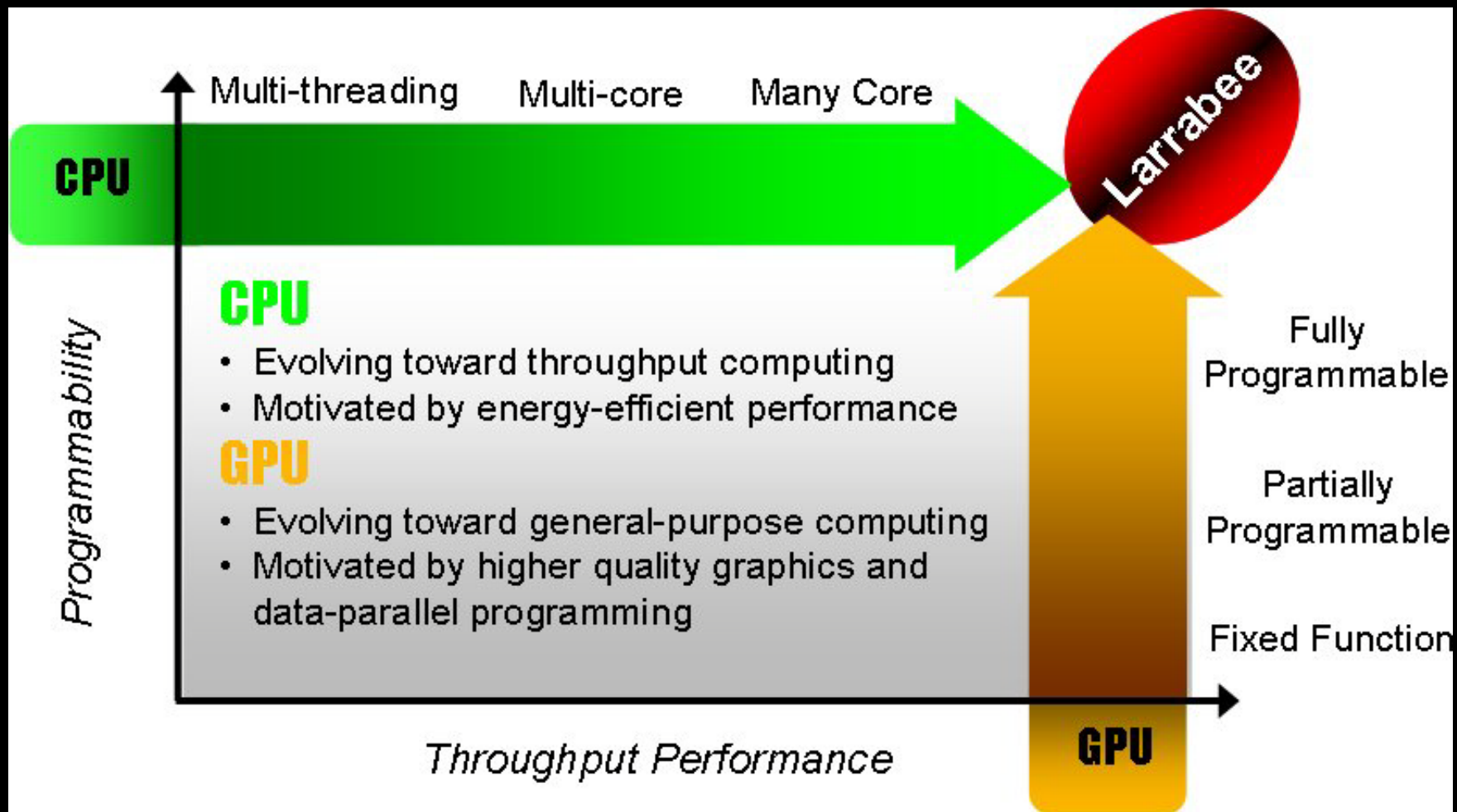
GPU Evolution






GPGPU








Early GPGPU drawbacks

- Require knowledge of graphics APIs and GPU architecture
 - Problems expressed in terms of vertex coordinates, textures and shader programs
 - Random reads and writes to memory were not supported
 - Lack of double precision support
- 



GPU Computing

- Nvidia G80 unified graphics and compute architecture
 - GeForce 8800[®], Quadro FX 5600[®], and Tesla C870[®] GPUs
 - **CUDA**
 - Software and hardware architecture
 - Enables the GPU to be programmed with high level languages
 - Ability to write C programs with CUDA extensions
 - **“GPU Computing”**
 - Broader application support
 - Wider programming language support
 - Clear separation from the early “GPGPU” model of programming
- 

NVIDIA Tesla

- Consists of Nvidia's highest end graphics card, **minus** the video out connector.
- Cuts the cost roughly in half
 - Quadro FX 5800 is ~\$3000
 - Tesla C1060 is ~\$1500.



http://images.nvidia.com/products/tesla_c1060/Tesla_c1060_3qtr_low.png

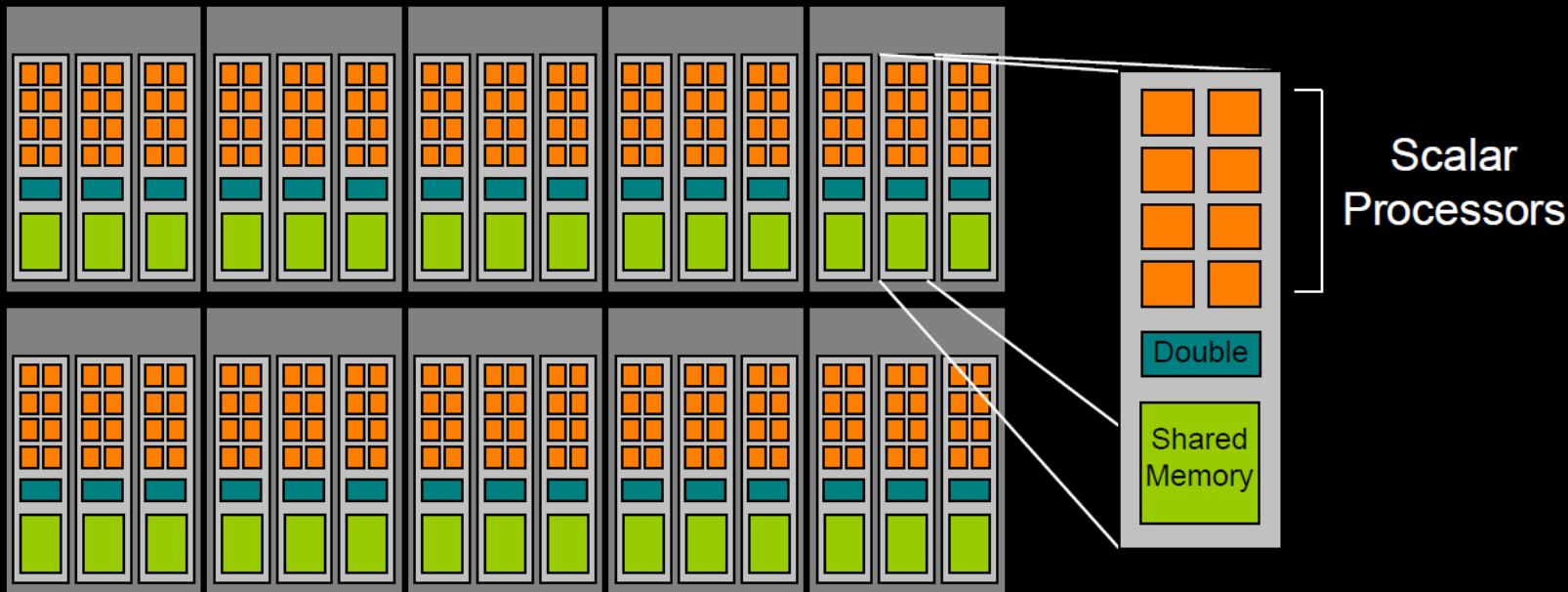
NVIDIA Tesla C1060 Card Specs



- 240 GPU cores
- 1.296 GHz
- Single precision floating point performance
 - **933 GFLOPs** (3 single precision flops per clock per core)
- Double precision floating point performance
 - **78 GFLOPs** (0.25 double precision flops per clock per core)
- Internal RAM: 4 GB
 - Speed: 102 GB/sec (compared 21-25 GB/sec for regular RAM)
- Has to be plugged into a PCIe slot (at most 8 GB/sec)

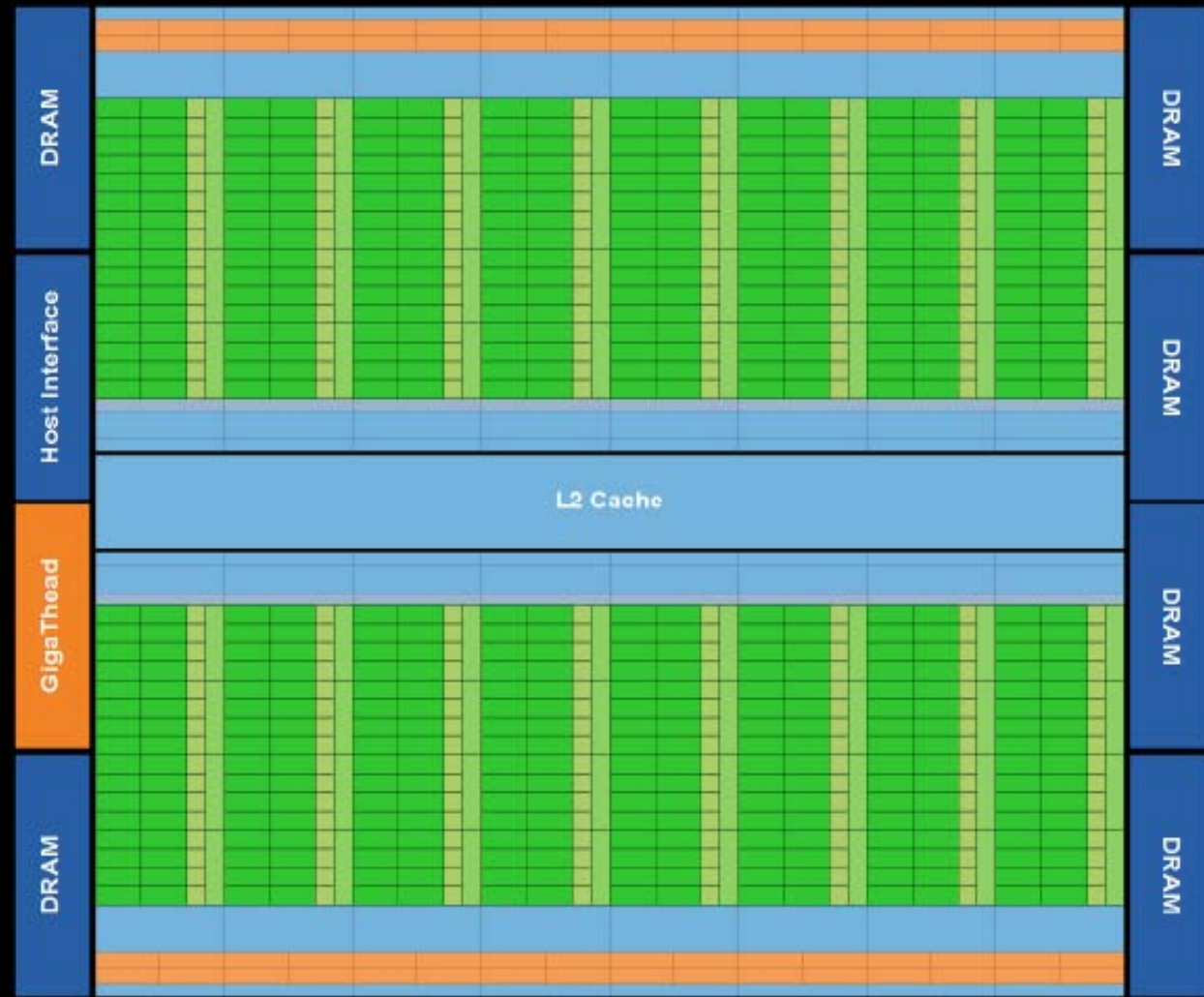
Tesla C1060 (GT 200) Architecture

- 30 Streaming Multiprocessors (SM)
- Each SM contains
 - 8 scalar processors
 - 1 double precision unit
 - 2 special function units
 - shared memory (16 K)
 - registers (16,384 32-bit=64 K)

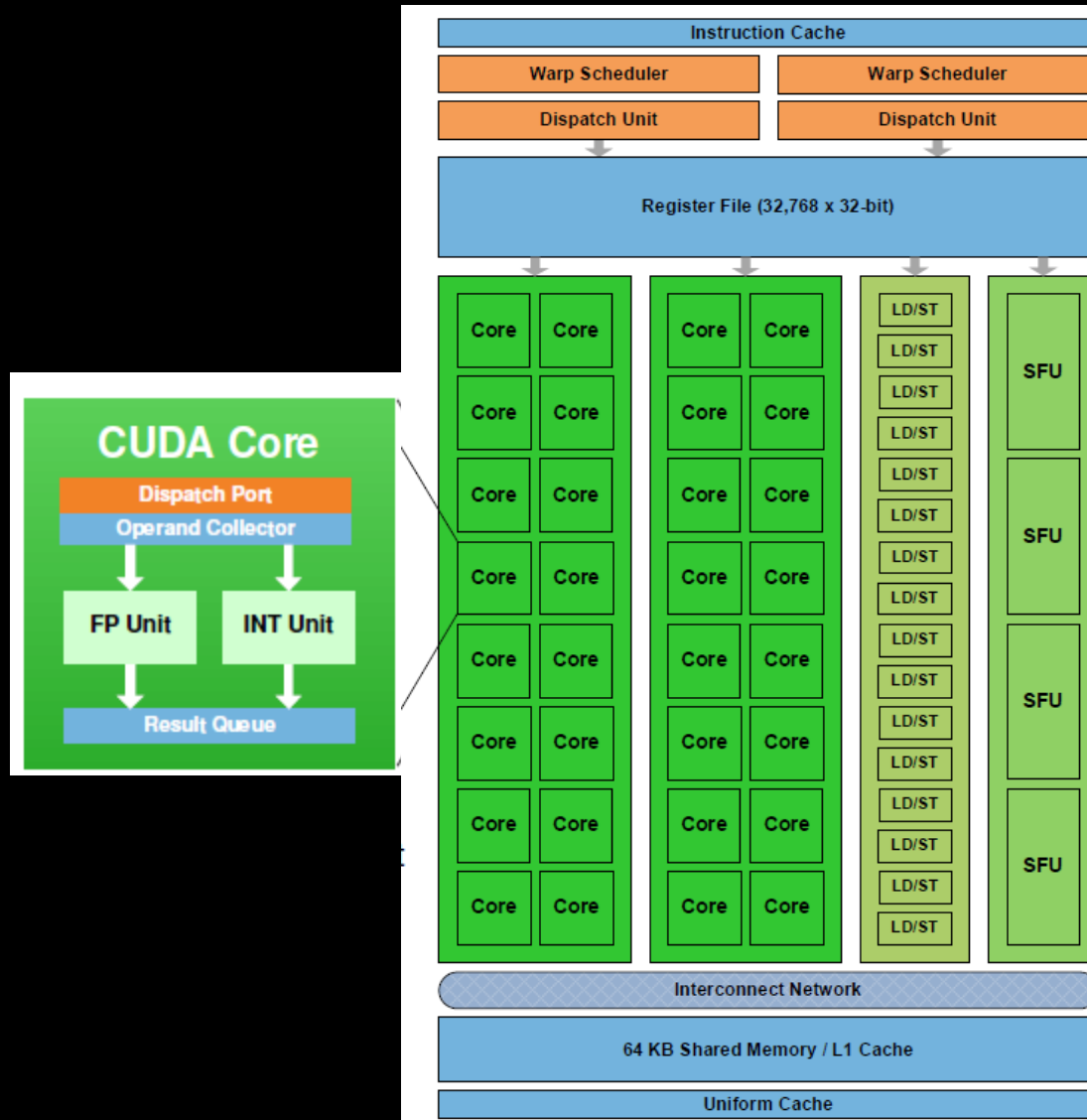





NVIDIA Fermi Architecture


- 16 SMs - 32 cores each
- Single core executes a floating point or an integer instruction per clock
- Six 64-bit memory partitions
- GigaThread global scheduler



Streaming Multiprocessor



- 
- 
- 
- 32 Cores
 - Fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU).
 - 16 Load/Store units
 - Four Special Function Units
 - sin, cosine, reciprocal, and square root.
 - Each SFU executes one instruction per thread, per clock
 - 64 KB Configurable Shared Memory and L1 Cache
 - can be configured as 48 KB of Shared memory with 16 KB of L1 cache or as 16 KB of Shared memory with 48 KB of L1 cache
 - Shared memory
 - Facilitates extensive reuse of on-chip data, and
 - Reduces off-chip traffic



GigaThread™ Thread Scheduler


- Two-level, distributed thread scheduler
 - Chip level , a global work distribution engine schedules thread blocks to various SMs,
 - SM level, each warp scheduler distributes warps of 32 threads to its execution units.
- Greater thread throughput(more than 12,288 threads)
- Faster context switching
- Concurrent kernel execution
- Improved thread block scheduling

Concurrent Kernel Execution

- Concurrent kernel execution
 - Different kernels of the same application context can execute on the GPU at the same time
 - Allows programs that execute a number of small kernels to utilize the whole GPU
- Sequential kernel execution
 - Kernels from different application contexts can run sequentially with great efficiency due to the improved context switching performance



Fermi Improvements

- Double Precision Performance
 - Error Correction Code (ECC) support
 - True Cache Hierarchy
 - More Shared Memory
 - Faster Context Switching
 - Faster Atomic Operations
- 

Double Precision Performance

- Implements the new IEEE 754-2008 floating-point standard.
- Providing the fused multiply-add (FMA) over MAD
- FMA improves over a multiply-add – no loss of precision in the addition

Multiply-Add (MAD):




Fused Multiply-Add (FMA)

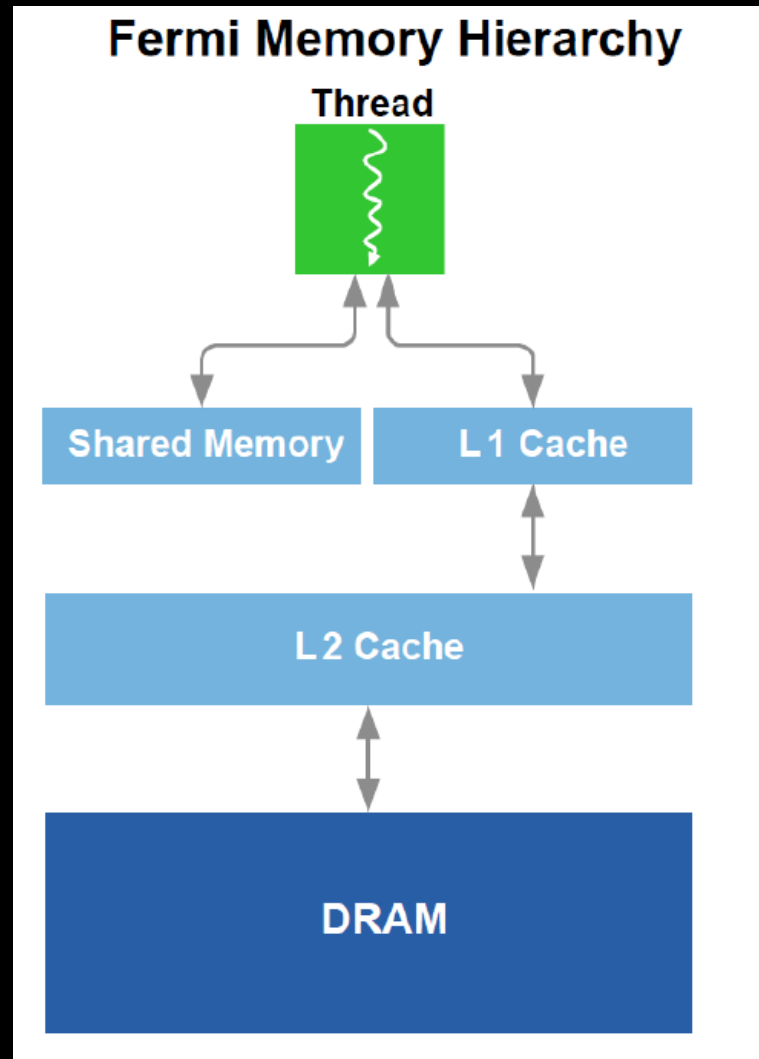




Error Correction Code (ECC) support

- Detects and corrects single-bit soft errors before they affect the system.
 - Single-Error Correct Double-Error Detect (SECCDED) ECC codes that correct any single bit error in hardware as the data is accessed.
 - Register files, shared memories, L1 caches, L2 cache, and DRAM memory are ECC protected
- 

True Cache Hierarchy



True Cache Hierachy

- Implement a single unified memory request path for loads and stores
 - An L1 cache per SM multiprocessor
 - Unified L2 cache that services all operations (load, store and texture).
- The per-SM L1 cache is configurable (64 KB)
 - 48 KB of Shared memory with 16 KB of L1 cache
 - 16 KB of Shared memory with 48 KB of L1 cache.

Parallel Thread Execution ISA

- Stable ISA that spans multiple GPU generations
- Achieve full GPU performance in compiled applications
- A machine-independent ISA for C, C++, Fortran, and other compiler targets.
- Common ISA for optimizing code generators and translators, which map PTX to specific target machines.
- Facilitate hand-coding of libraries and performance kernels
- Provide a scalable programming model that spans GPU sizes from a few cores to many parallel cores



Fast Atomic Memory Operations

- Allowing concurrent threads to correctly perform read-modify-write operations on shared data structures
- Used for
 - parallel sorting
 - reduction operations
 - building data structures
- Combination of more atomic units in hardware and the addition of the L2 cache, atomic operations performance is up to 20x faster



DEMO

Device Query



Fermi

- Real Floating Point in Quality and Performance
- Error Correcting Codes (ECC) on Main Memory and Caches
- 64bit Virtual Address Space
 - Parallel Thread Execution (PTX) layer allowed seamless migration
- Caches
 - 64KB L1
 - Ability to split as shared memory & Cache
 - 768KB L2
 - Total registers are larger than L1 & L2
 - Total L1 equals to L2
- Fast Context Switching



Fermi

- **Unified Address Space**
- **Debugging Support**
- **Faster Atomic Instructions to Support TaskBased Parallel Programming**
- **A Brand New Instruction Set**
- **Fermi is Faster than G80**



GPGPU Programming






Programming model

- The GPU is viewed as a compute device that:
 - Is a coprocessor to the CPU (host)
 - Has its own GDRAM (device memory)
 - Runs many threads in parallel
- Single Instruction Multiple Thread (SIMT)
- Data-parallel portions of an application are executed on the device as **kernels** which run in parallel on many threads




Hardware Differences

CPU VS GPU

- Threading Resources (parallel threads)
 - Handful in CPU vs Thousands in GPU
 - Threads
 - GPU threads are extremely lightweight
 - Very little creation overhead
 - GPU needs 1000s of threads for full efficiency
 - Multi-core CPU needs only a few
 - RAM
 - Single address space vs many levels of programmer managed
- 



What's Best For GPU's

- Massively Parallel computations
 - Data parallel
 - Coherence memory access by kernels
 - Less data moving
 - Enough computations to justify moving data
 - Keep data on the device as long as possible
- 



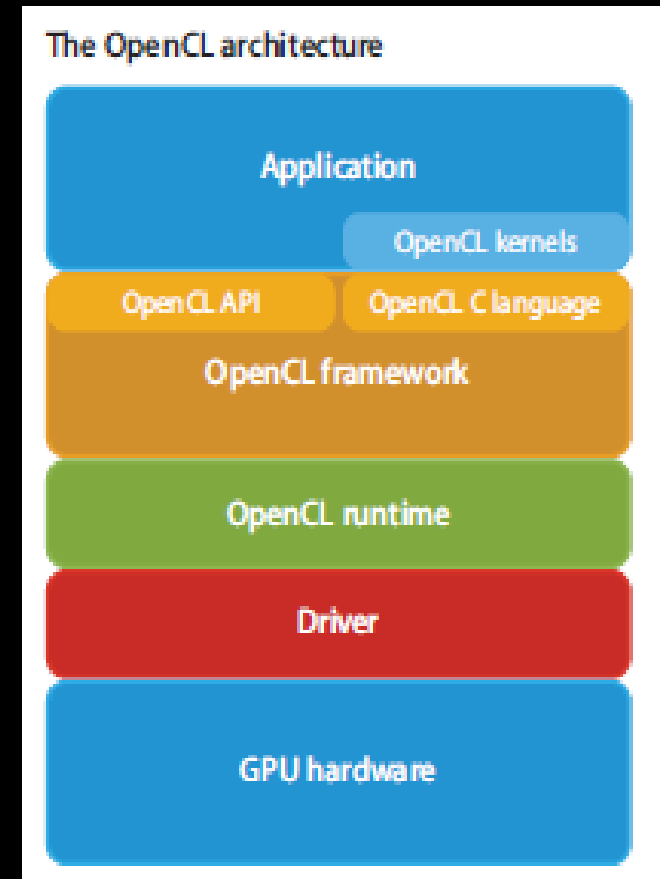
Programming Languages

- OpenCL
 - CUDA
 - ATI Stream SDK
 - OpenGL
 - DirectX
- 



OpenCL

- **Portable** code across multiple devices
 - GPU, CPU, Cell, Mobiles, Embedded systems, ..
- OpenCL platform model
 - **Host** connected to multiple **compute devices**
 - Compute device -> multiple compute units
 - Compute unit -> multiple processing units

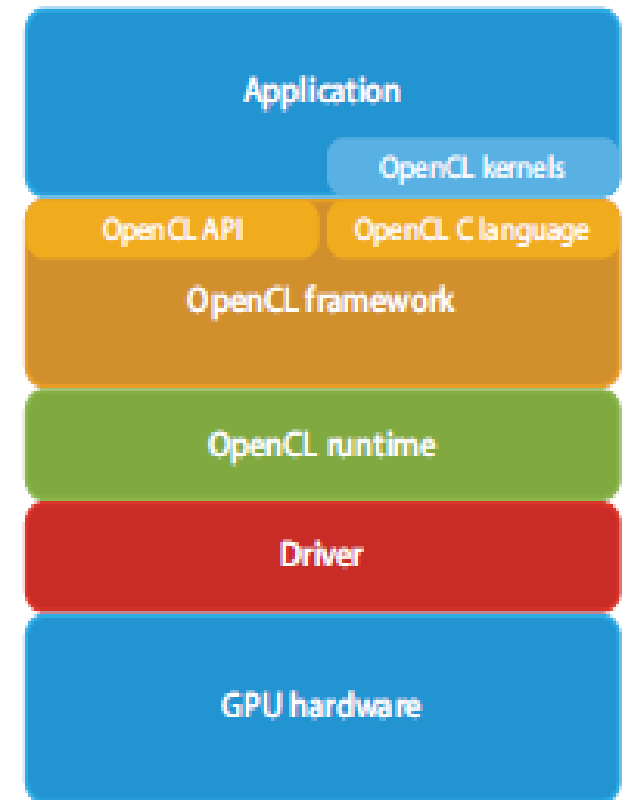




OpenCL

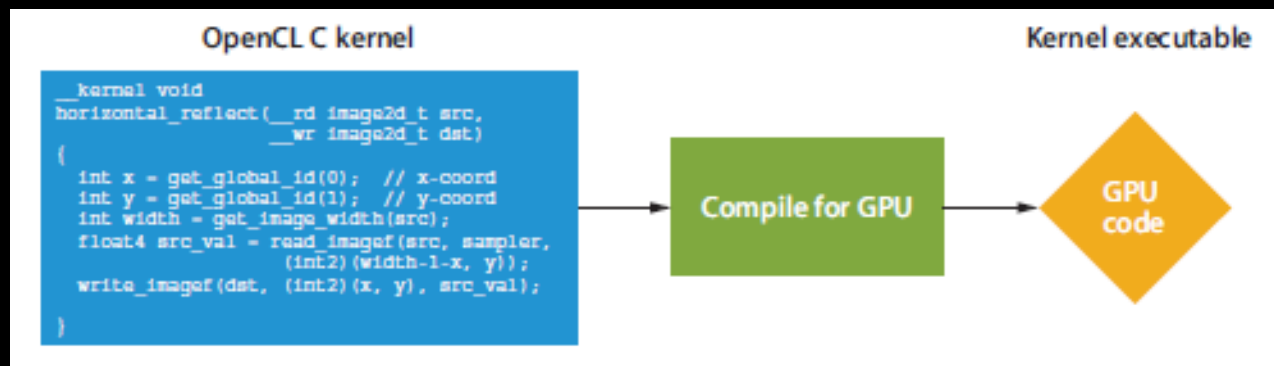
- **Host Code** -> c/ c++
 - Transfer data host memory <-> device memory
 - Execute device code
- Device code (**Kernels**) - > OpenCL C
 - Basic unit of executable code
- Serial parts (CPU) interleaved by parallel parts(GPU)

The OpenCL architecture




OpenCL API's

- Platform layer API (host)
 - Functions to manage parallel computing tasks
 - Abstraction for diverse compute resources
- Runtime
 - Task execution
 - Resource management
- OpenCL C Kernels
 - Compiled on the fly, optimized for user's hardware



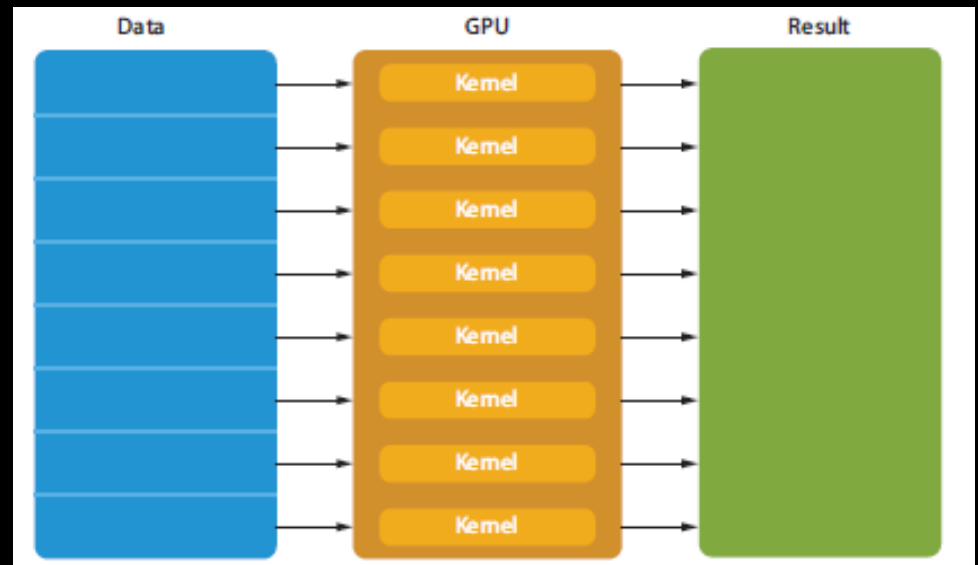
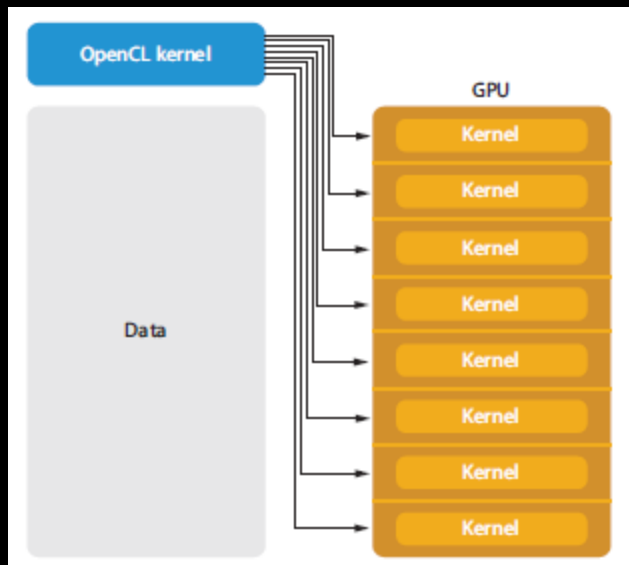


Few Keywords

- Devices
 - CPUs, GPUs
 - Contexts
 - Sharing data between devices
 - Queues
 - All work submitted through queues
 - A queue for each device
- 

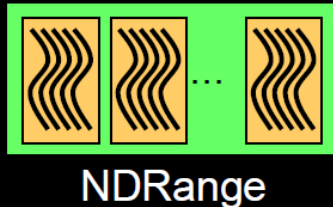
How it works..

- Device Query
- Select devices & create command queues
- Load & compile kernels
- Specify data & number of kernels
- Move data GPU VRAM
- Executes kernels

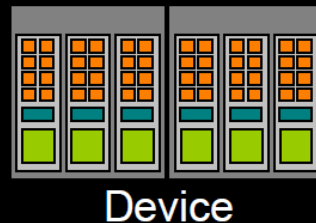
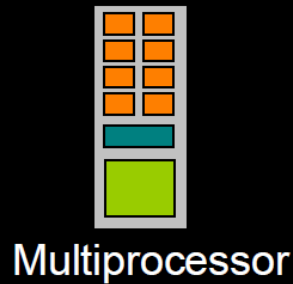


Execution Model

Software

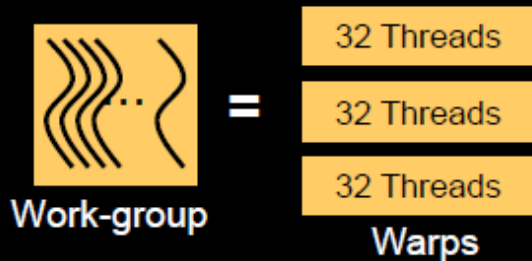


Hardware

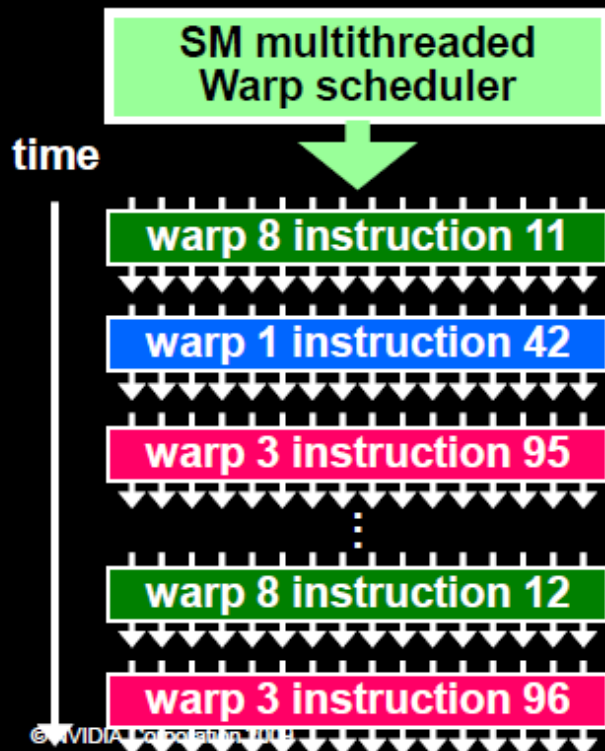


- Work-items are executed by streaming processors
- Maps directly to HW managed threads
- Work-groups are executed on multiprocessors
- They do not migrate
- Several concurrent work-groups can reside on one multiprocessor - limited by multiprocessor resources
- A kernel is launched as an N-D Range of work-groups
- Work-groups are dynamically load-balanced by HW scheduler

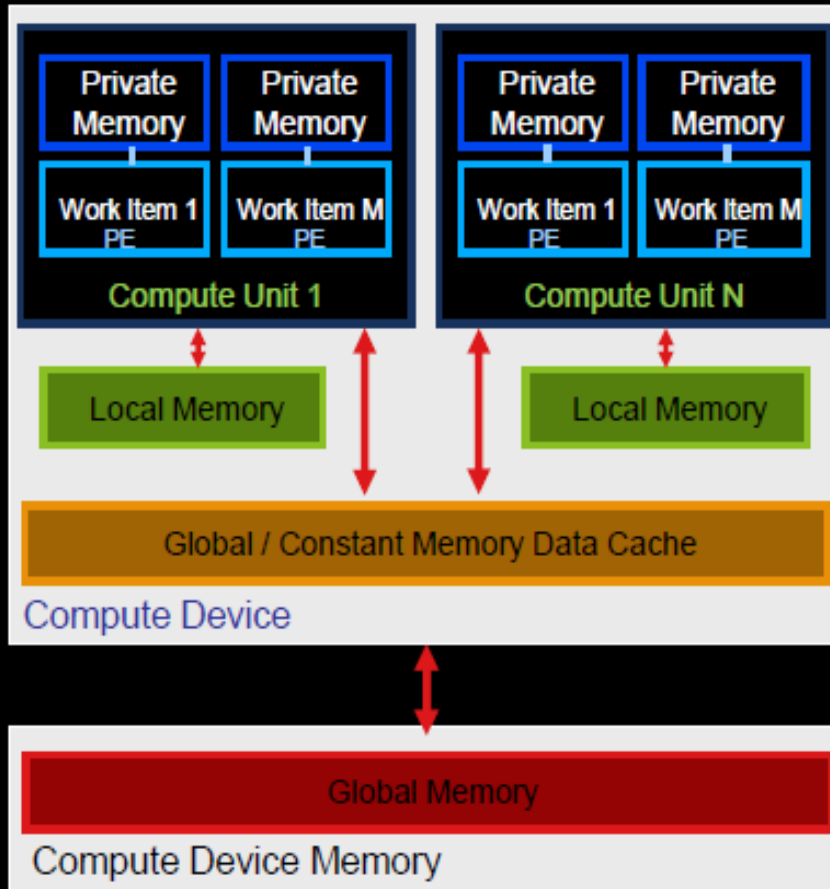
Warps & SIMT



- Basic scheduling unit
- Work-groups divide into groups of 32 threads called warps.
- Warps always perform same instruction (SIMT)
- A lot of warps can hide memory latency



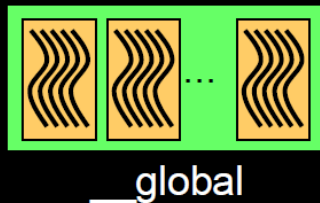
OpenCL Memory Hierarchy



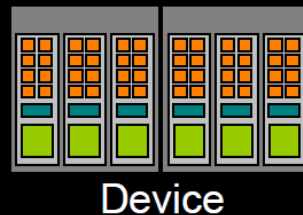
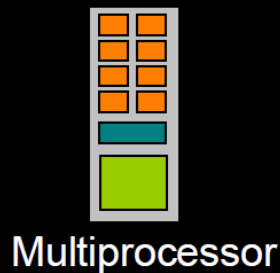
- **Global: R/W per-kernel**
- **Constant : R per-kernel**
- **Local memory: R/W per-group**
- **Private: R/W per-thread**

Memory Model

Software



Hardware




- Each hardware thread has a dedicated `__private` region for stack

- Each multiprocessor has dedicated storage for `__local` memory and `__constant` caches
- Work-items running on a multiprocessor can communicate through `__local` memory

- All work-groups on the device can access `__global` memory
- Atomic operations allow powerful forms of global communication



Memory best practices

- Minimize host<->device data transfer
 - Batch Transfers
 - Overlap IO with computation
 - Coalesced memory access
 - Use local memory as a cache
 - ~100x smaller latency
- 

Performance Optimizations

- Compiling programs can be expensive
 - Reuse or precompiled binaries
- Starting a kernel can be expensive
- Large global work sizes hide memory latencies
- Trade-off precision & performance with less precise data types (half_ & native_)
- Divergent execution can be bad
- Data reuse through local memory

A vertical bar on the left side of the slide, composed of several colored segments: a small pink square at the top, a grey rectangle, a yellow rectangle, and a long pink rectangle at the bottom.

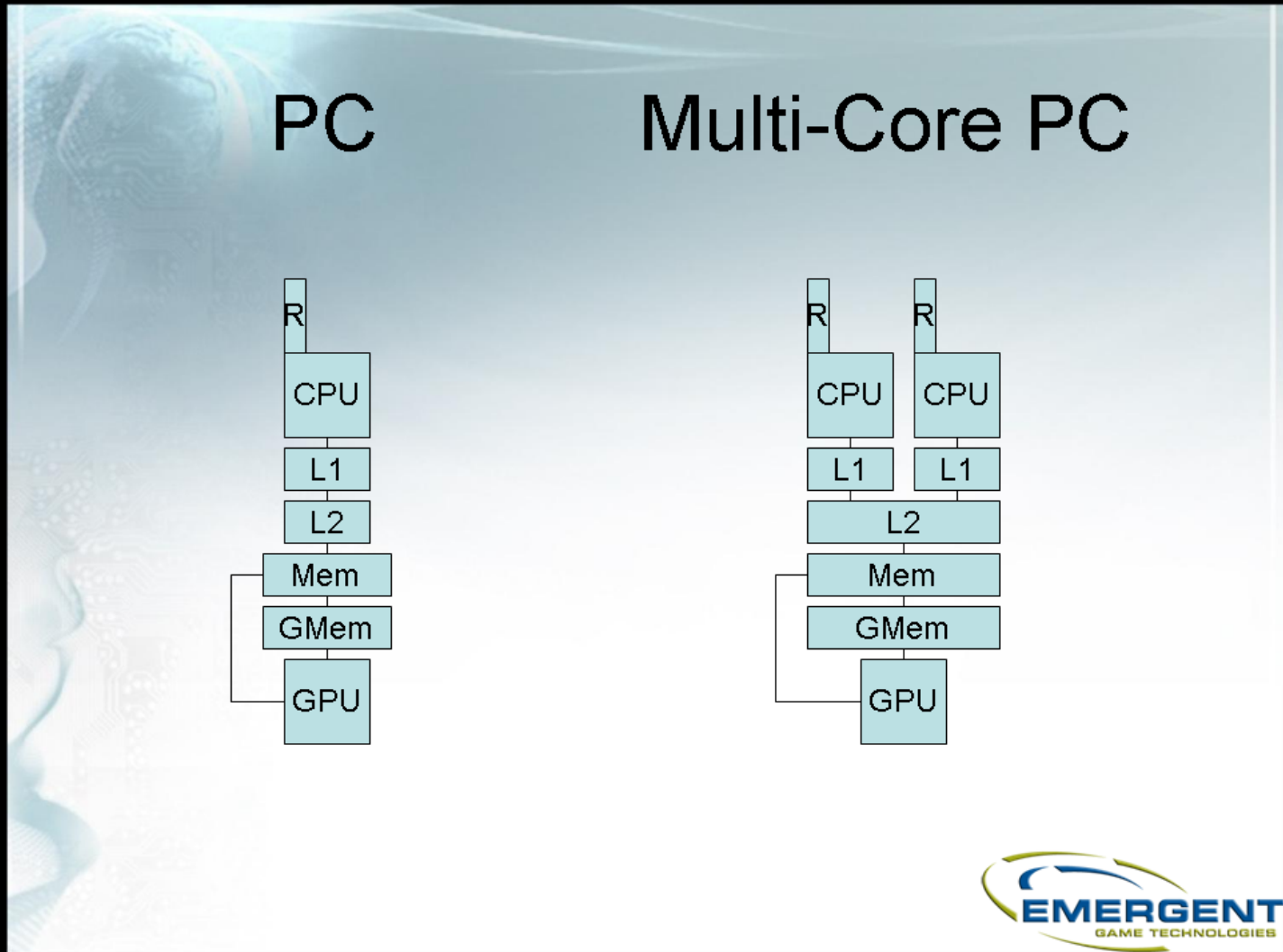
Demo

- Bandwidth Test

Accelerator RECALL

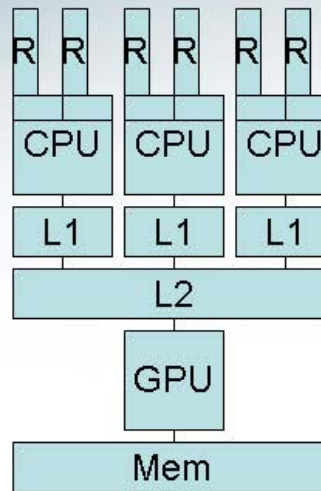


CPU's



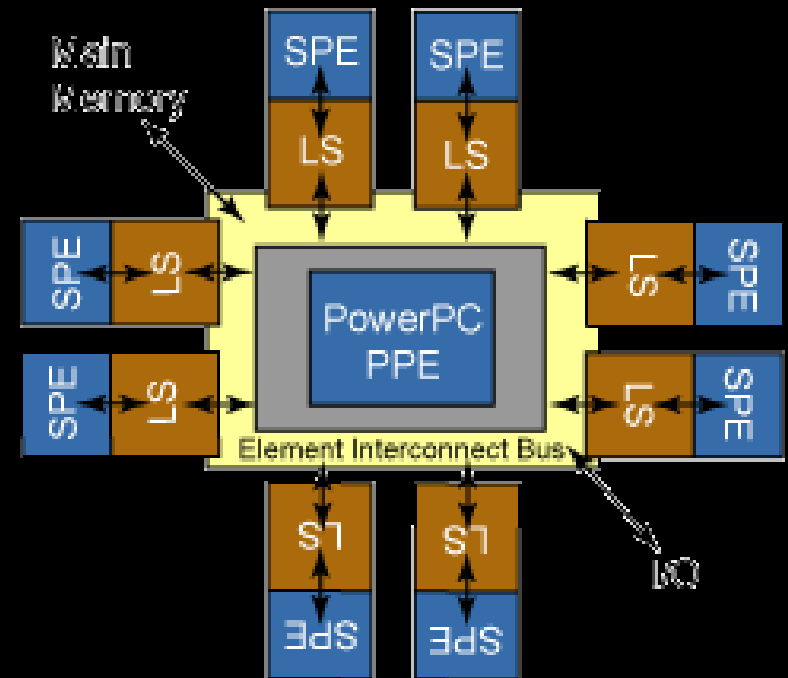
Xbox 360

Xbox 360



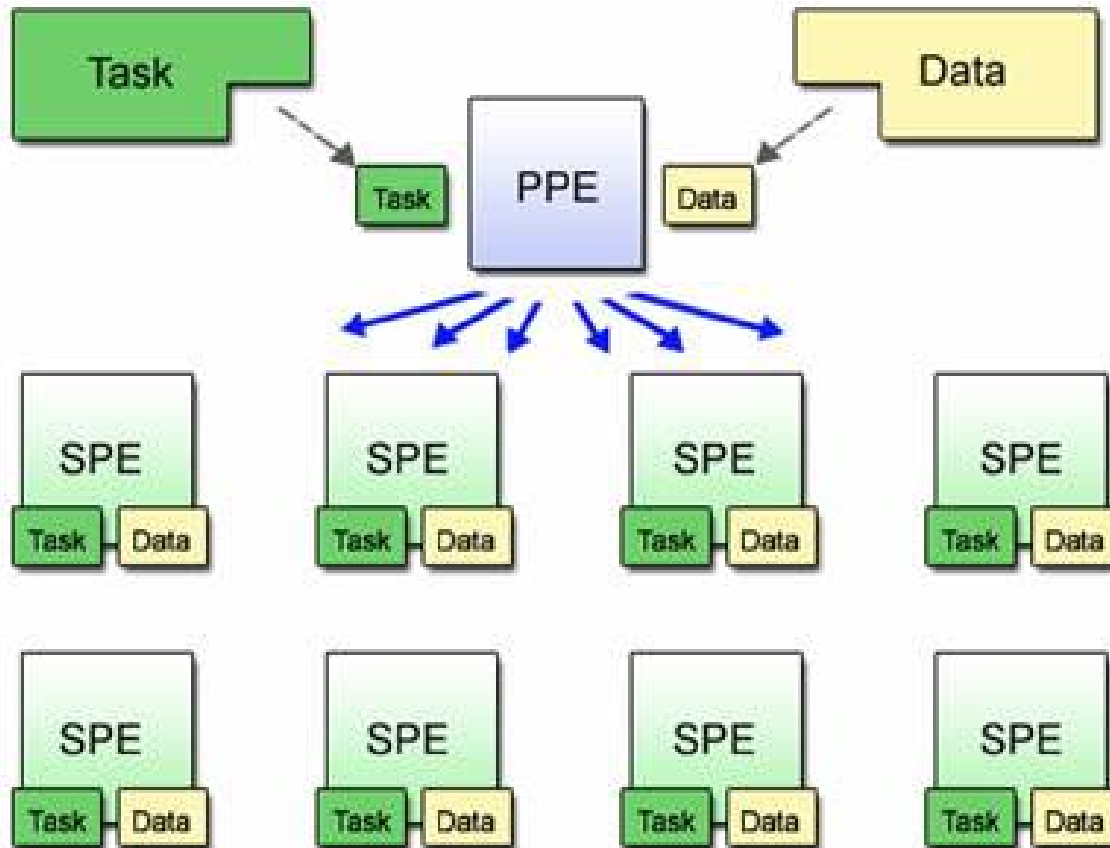
Cell Broadband Engine

- Novel memory coherence architecture
 - emphasizes efficiency/watt
 - prioritizes bandwidth over latency
 - favors peak computational throughput over simplicity of program code.
 - Challenging to developers
- Architecture
 - Power processing element (PPE)
 - 8 co-processors (SPEs)
 - High bandwidth circular data bus
 - Element interconnect Bus
 - Cache coherent DMA engine in each processor
 - Overlap computation with I/O
- IBM RoadRunner in Los Alamos
 - Opteron and Powercell 8i based

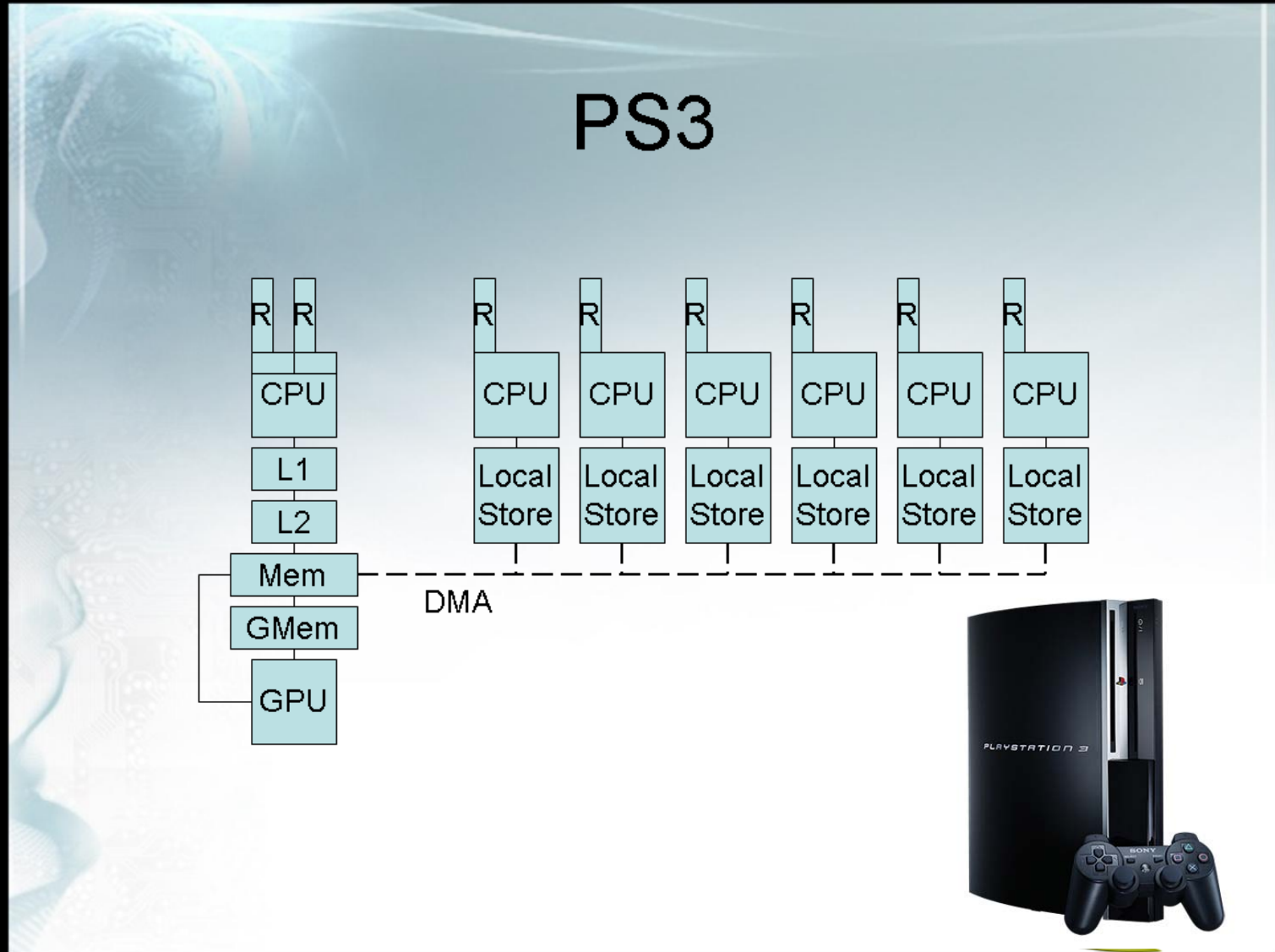


Cell Broadband Engine Abstract Overview

Cell



Sony PlayStation 3



Sony PlayStation 3

- US Military purchase of 2000 PS3
 - <http://scitech.blogs.cnn.com/2009/12/09/military-purchases-2200-ps3s/>
 - Army uses Call of Duty to train soldiers ;-)

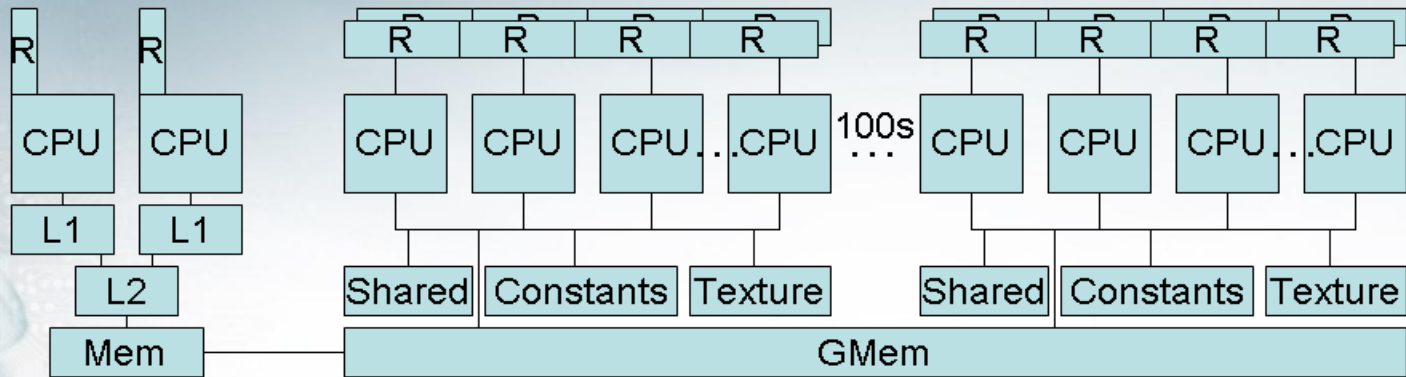


Sony PlayStation 3

- US Military purchase of 2000 PS3
 - <http://scitech.blogs.cnn.com/2009/12/09/military-purchases-2200-ps3s/>
 - ~~Army uses Call of Duty to train soldiers ; -)~~
 - For HPC
 - $\$299 * 2200 \sim .65M \$$
 - $150 \text{ Gflops} * 2200 \sim = 320 \text{ TFlops}$

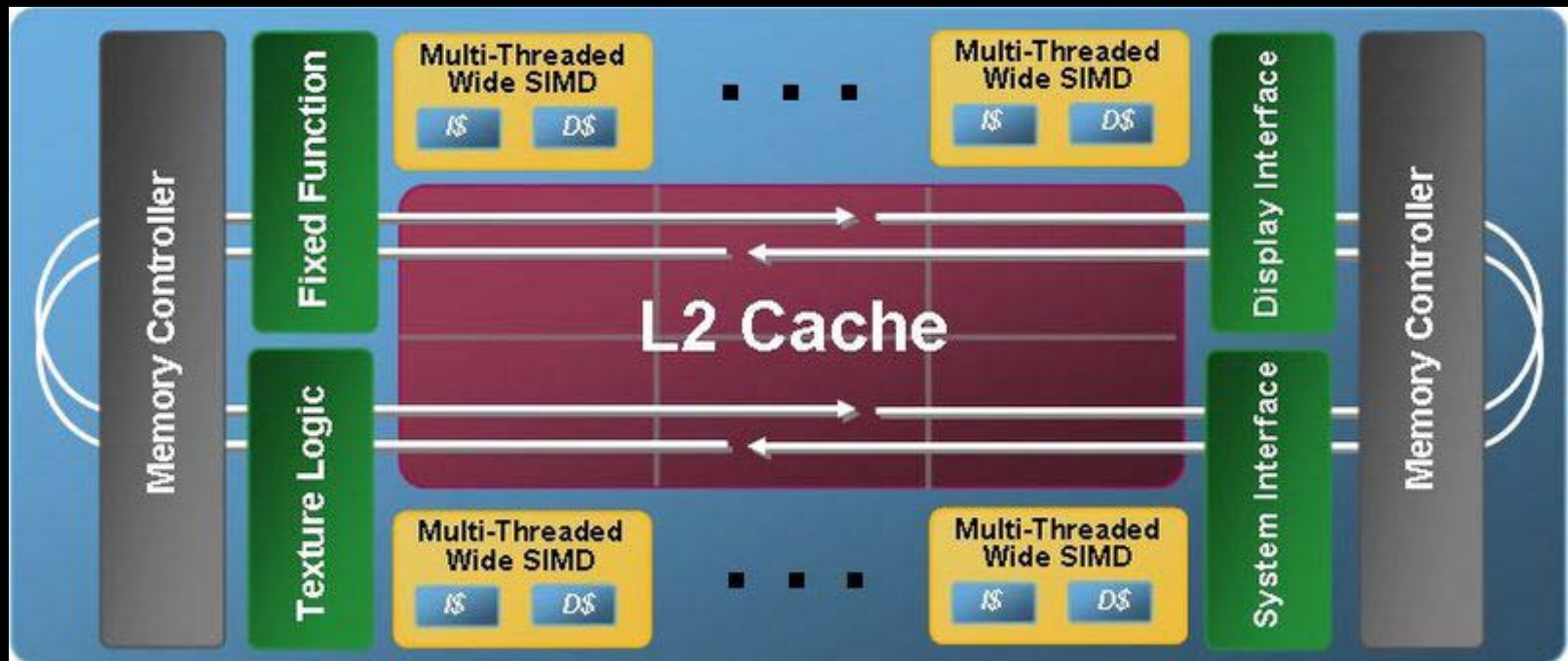
CUDA

CUDA



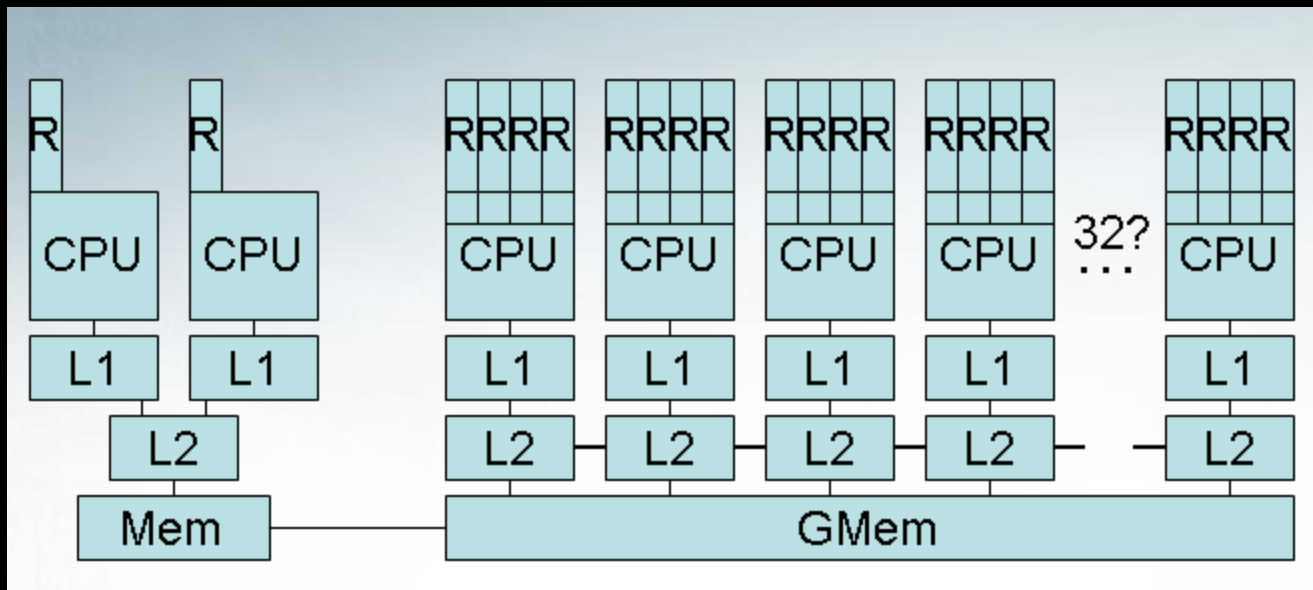
Intel Larabee

- Hybrid
 - Multi-core CPU
 - Coherent Cache
 - x86 compatibility with Larabee extensions
 - GPU
 - SIMD



Intel Larabee

- L2 broken in to cores
 - Faster access to **local** L2
 - High speed ring bus connecting L2 caches






DEMO

Matrix Multiplication





Current Landscape

- Hot area, surprising amount of work happening
 - Many applications
 - <http://developer.download.nvidia.com/compute/cuda/docs/GTCogMaterials.htm>
 - Many libraries
 - CUBLAS
 - Still many possibilities too..
 - Many cores seems to be the future...
- 

References

- <http://beautifulpixels.blogspot.com/2008/08/multi-platform-multi-core-architecture.html>
- <http://developer.nvidia.com/object/opengl.html>
- http://developer.nvidia.com/object/gpu_computing_online.html
- <http://www.intel.com/technology/visual/microarch.htm>
- http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
- http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf
- http://images.apple.com/macosx/technology/docs/OpenCL_TB_brief_20090903.pdf
- http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf
- http://developer.nvidia.com/object/gpu_gems_2_home.html

We extracted some slides from..

- http://www.khronos.org/developers/library/overview/opengl_overview.pdf
- http://www.hotchips.org/archives/hc21/1_sun/H_C21.23.2.OpenCLTutorial-Epub/H_C21.23.250.Lamb-NVIDIA-OpenCL--for-NVIDIA-GPUs.pdf
- http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Best_Practises_For_OpenCL_Programming.pdf
- <http://coachk.cs.ucf.edu/courses/CDA6938/Nvidia%20G80%20Architecture%20and%20CUDA%20oProgramming.pdf>