Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

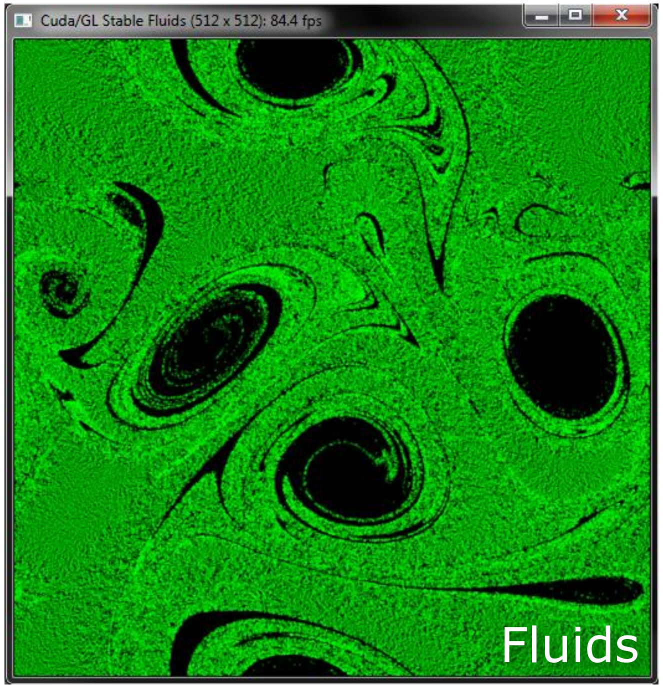
Parallel Programming Concepts

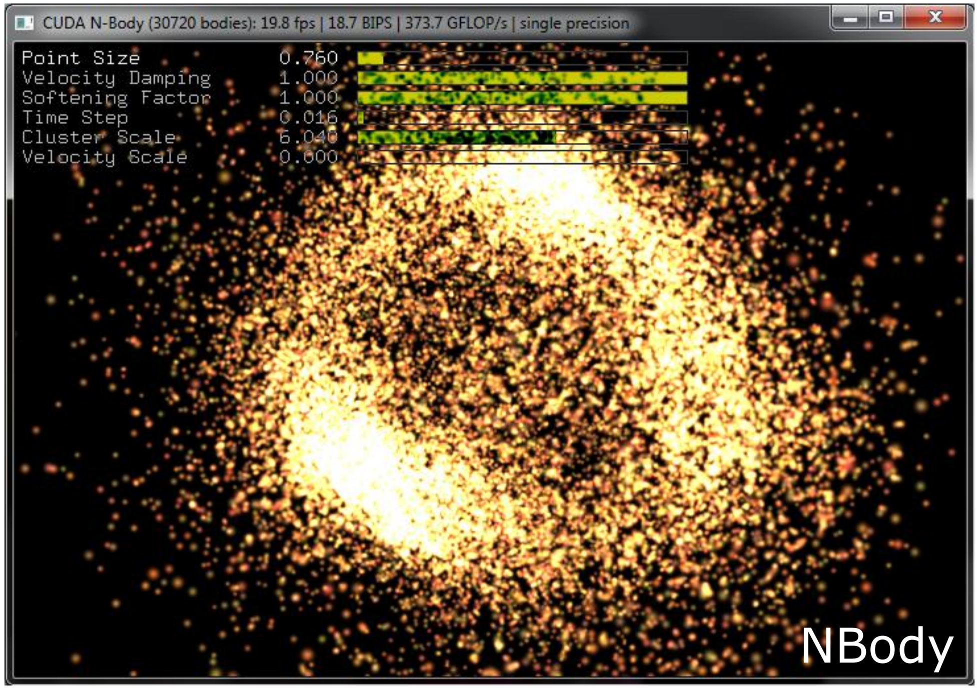
GPU Compute Devices

Frank Feinbube

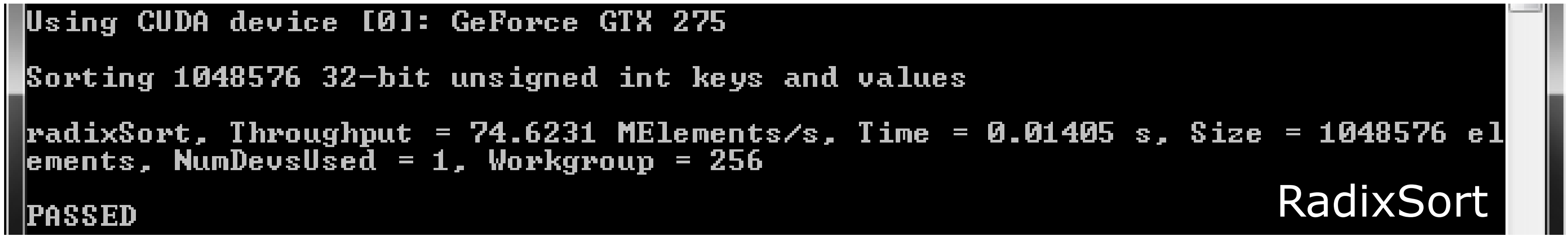
Operating Systems and Middleware
Prof. Dr. Andreas Polze

The Power of GPU Compute Devices




CUDA N-Body (30720 bodies): 19.8 fps | 18.7 BIPS | 373.7 GFLOP/s | single precision

```
Point Size          0.760
Velocity Damping    1.000
Softening Factor    1.000
Time Step           0.016
Cluster Scale       6.040
Velocity Scale      0.000
```

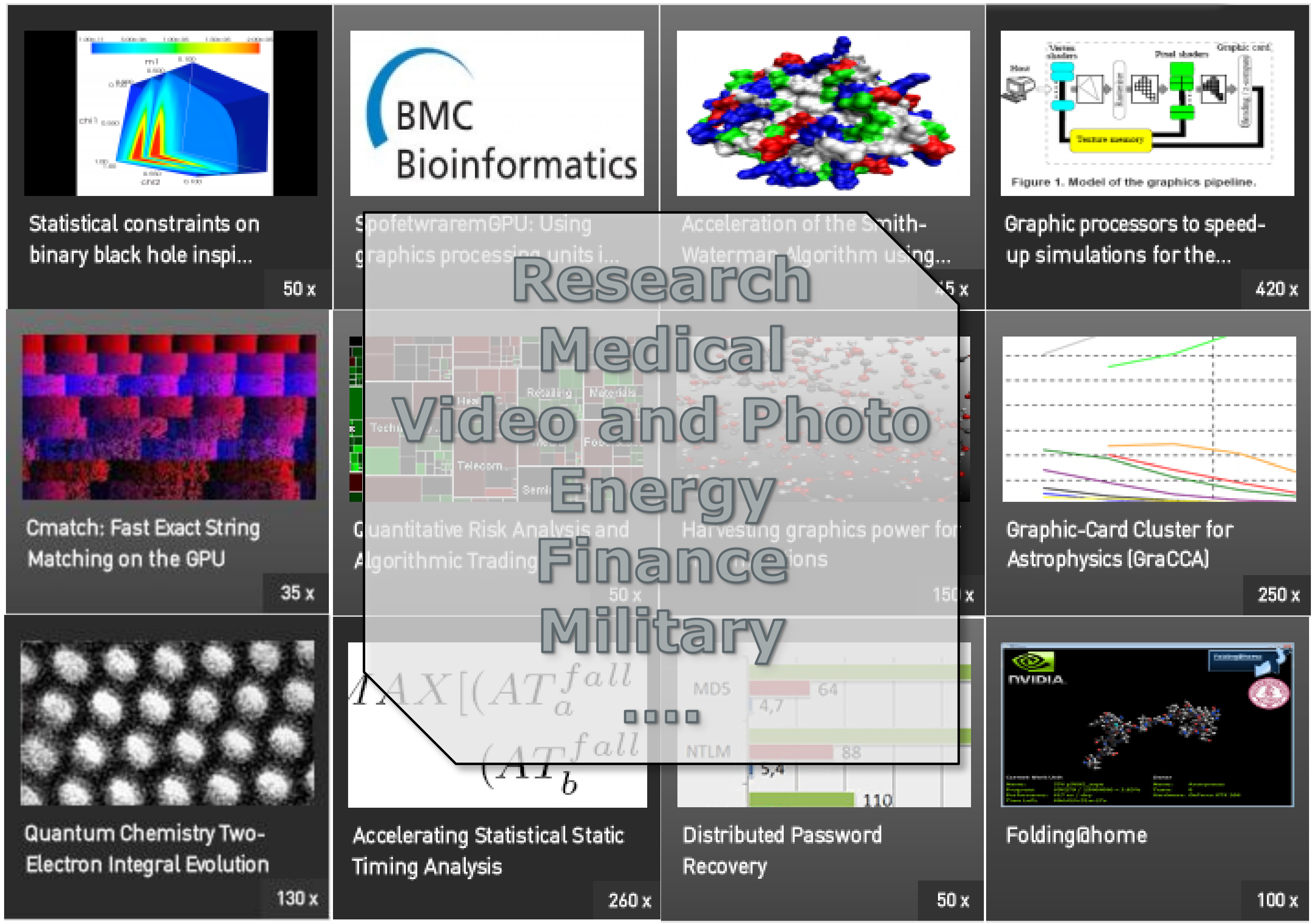

```
Using CUDA device [0]: GeForce GTX 275

Sorting 1048576 32-bit unsigned int keys and values

radixSort, Throughput = 74.6231 MElements/s, Time = 0.01405 s, Size = 1048576 el
ements, NumDevsUsed = 1, Workgroup = 256

PASSED
```

Wide Varity of Application Domains



Statistical constraints on binary black hole inspi... 50 x

SpofetwraremGPU: Using graphics processing units i...

Acceleration of the Smith-Waterman Algorithm using... 45 x

Graphic processors to speed-up simulations for the... 420 x

Cmatch: Fast Exact String Matching on the GPU 35 x

Quantitative Risk Analysis and Algorithmic Trading 50 x

Harvesting graphics power for ...ions 150 x

Graphic-Card Cluster for Astrophysics (GraCCA) 250 x

Quantum Chemistry Two-Electron Integral Evolution 130 x

Accelerating Statistical Static Timing Analysis 260 x

Distributed Password Recovery 50 x

Folding@home 100 x

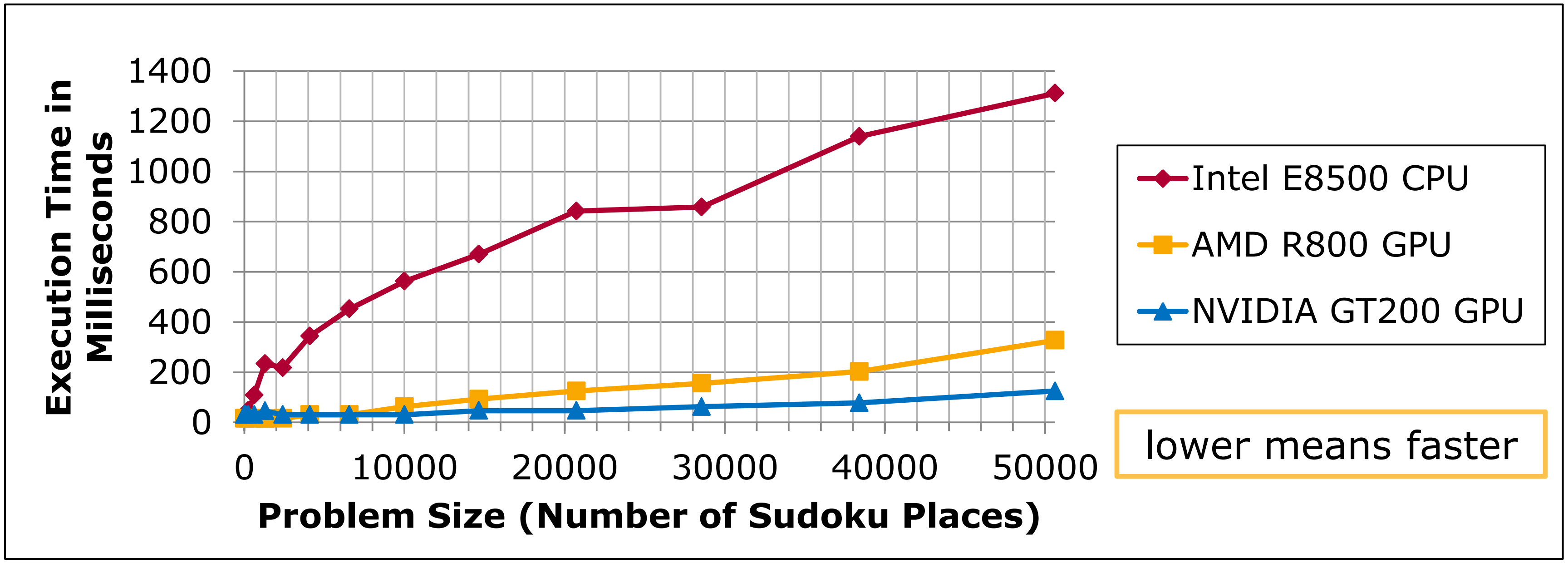
- Research
- Medical
- Video and Photo
- Energy
- Finance
- Military
- ....

http://www.nvidia.com/object/cuda_apps_flash_new.html

http://www.nvidia.com/object/tesla_testimonials.html

Why GPU Compute Devices?
Short Term View: Cheap Performance

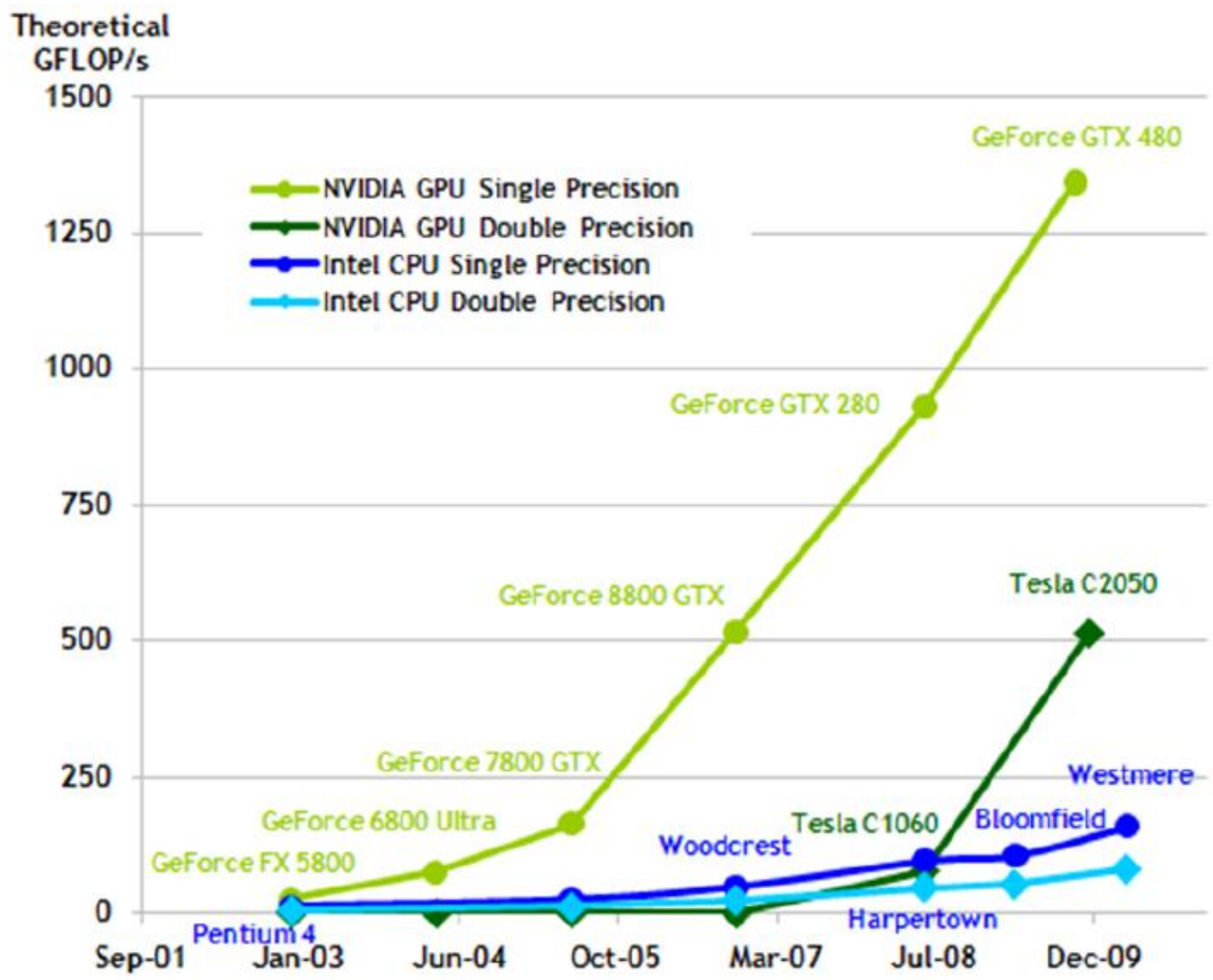
Performance



Energy / Price

- Cheap to buy and to maintain
- GFLOPS per watt: Fermi 1,5 / Keppler 5 / Maxwell 15

Why GPU Compute Devices?
Middle Term View: More Performance



[8]

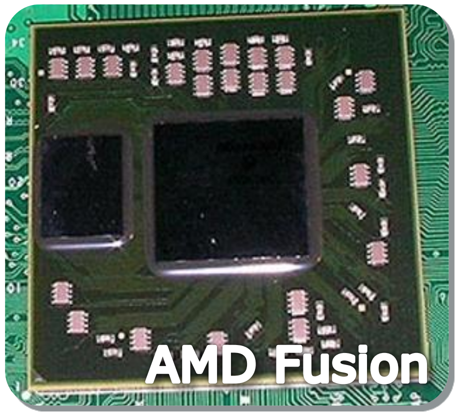
Why GPU Compute Devices?
Long Term View: Hybrid Computing

Dealing with massivly multi-core:

- New architectures are evaluated (Intel SCC)
- Accelerators that accompany common general purpose CPUs (Hybrid Systems)

Hybrid Systems

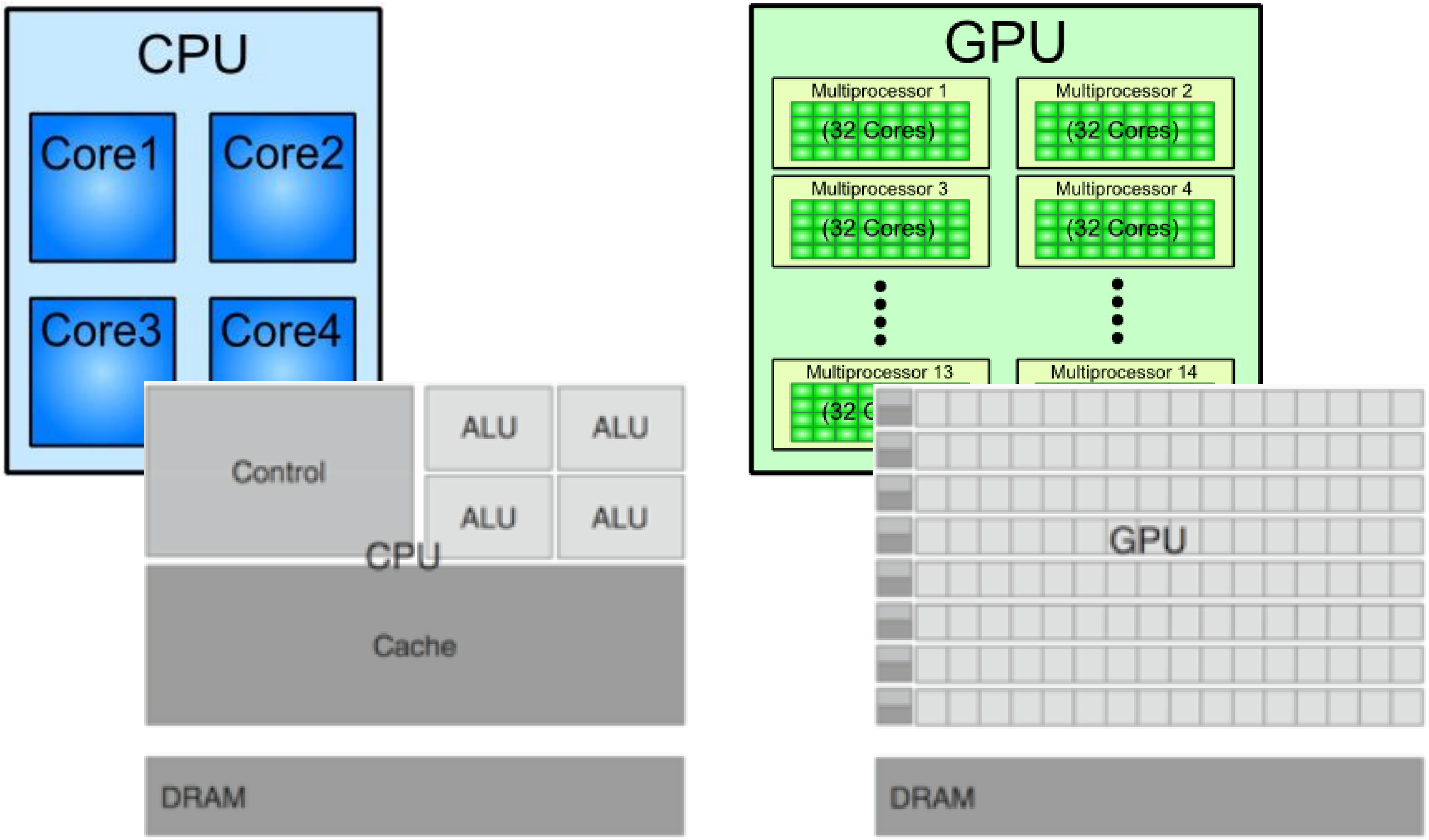
- **GPU Compute Devices:**
  High Performance Computing (3 of top 5 supercomputers are GPU-based!),
  Business Servers, Home/Desktop Computers,
  Mobile and Embedded Systems
- **Special-Purpose Accelerators:**
  (de)compression, XML parsing, (en|de)cryption, regular expression matching



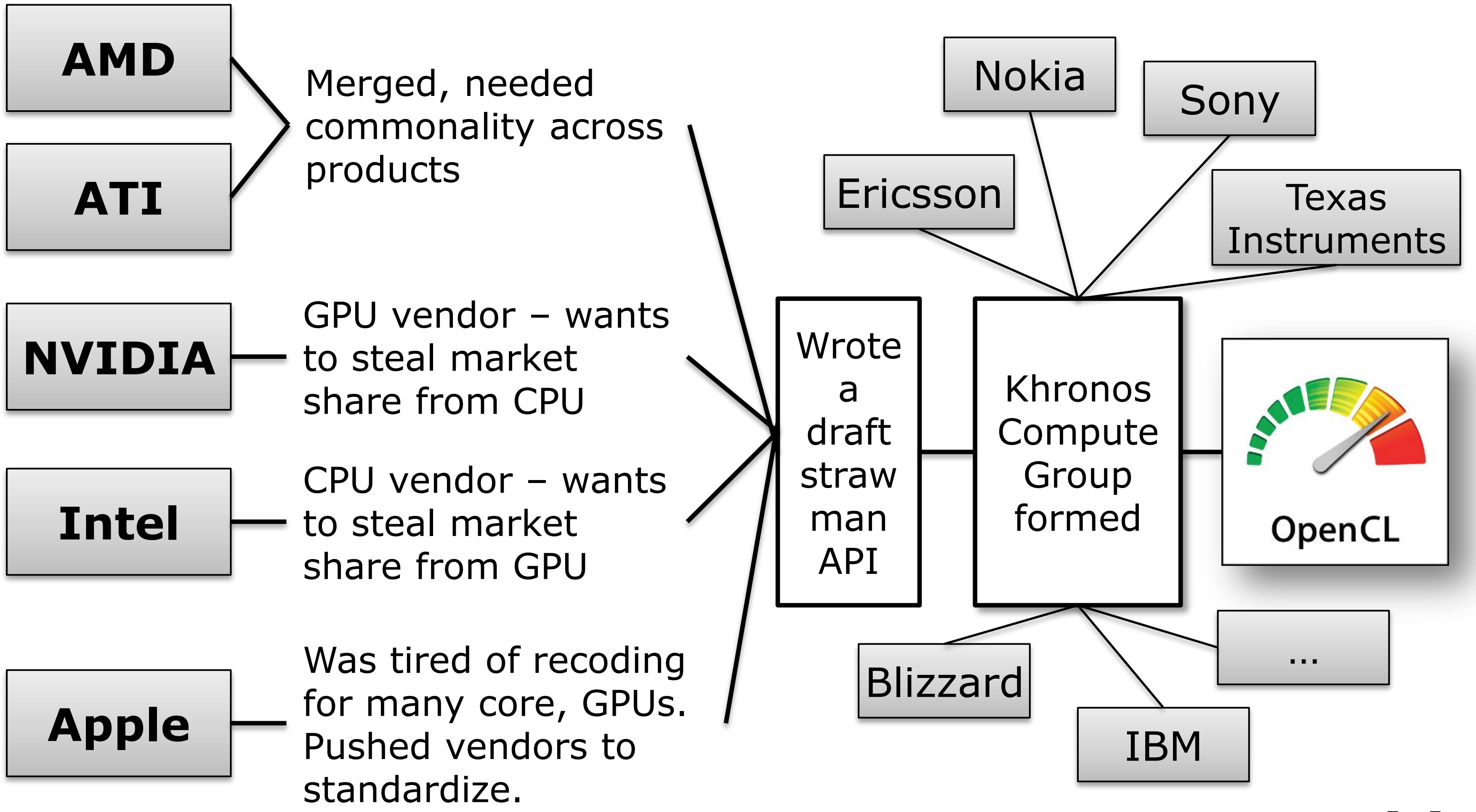
History of GPU Computing

Fixed Function Graphic Pipelines	• 1980s-1990s; configurable, not programmable; first APIs (DirectX, OpenGL); Vertex Processing
Programmable Real-Time Graphics	• Since 2001: APIs for Vertex Shading, Pixel Shading and access to texture; DirectX9
Unified Graphics and Computing Processors	• 2006: NVIDIAs G80; unified processors arrays; three programmable shading stages; DirectX10
General Purpose GPU (GPGPU)	• compute problem as native graphic operations; algorithms as shaders; data in textures
GPU Computing	• Programming CUDA; shaders programmable; load and store instructions; barriers; atomics

CPU vs. GPU Architecture

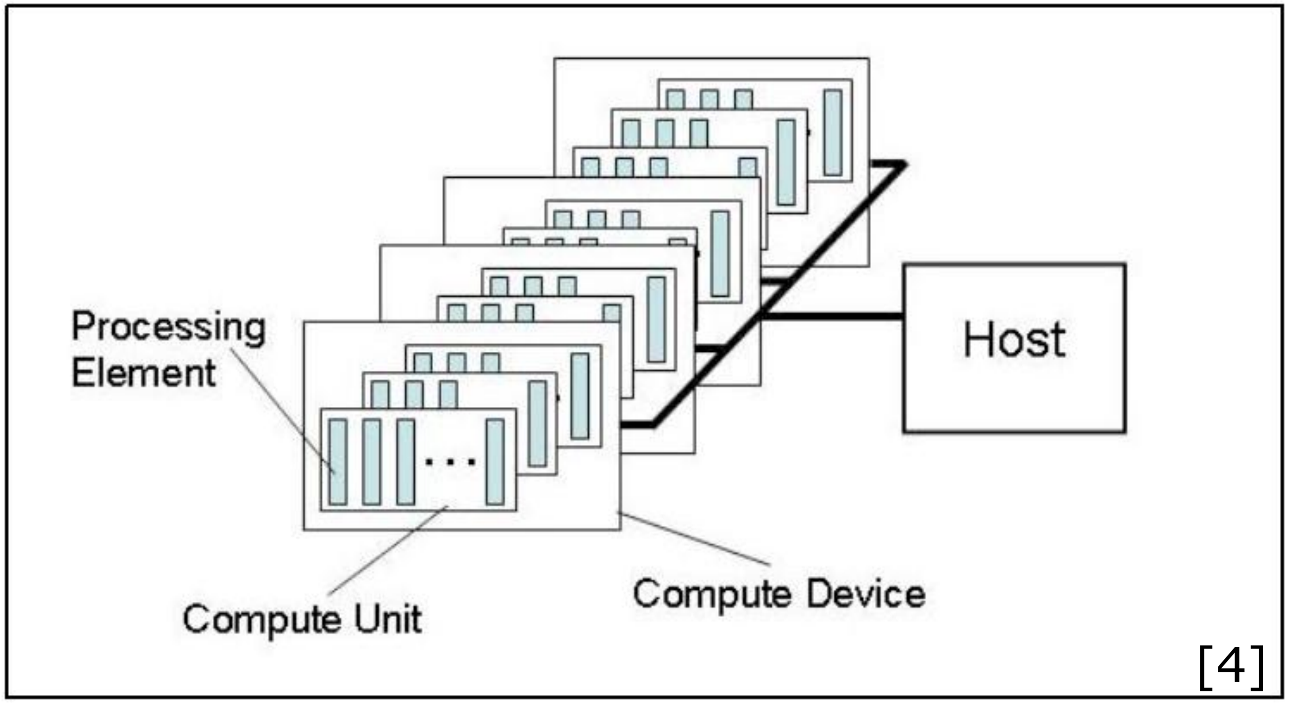


Open Compute Language (OpenCL)


AMD
ATI
Merged, needed commonality across products
NVIDIA
GPU vendor – wants to steal market share from CPU
Intel
CPU vendor – wants to steal market share from GPU
Apple
Was tired of recoding for many core, GPUs. Pushed vendors to standardize.
Wrote a draft straw man API
Khronos Compute Group formed
OpenCL
Nokia
Sony
Ericsson
Texas Instruments
Blizzard
IBM
...

[5]

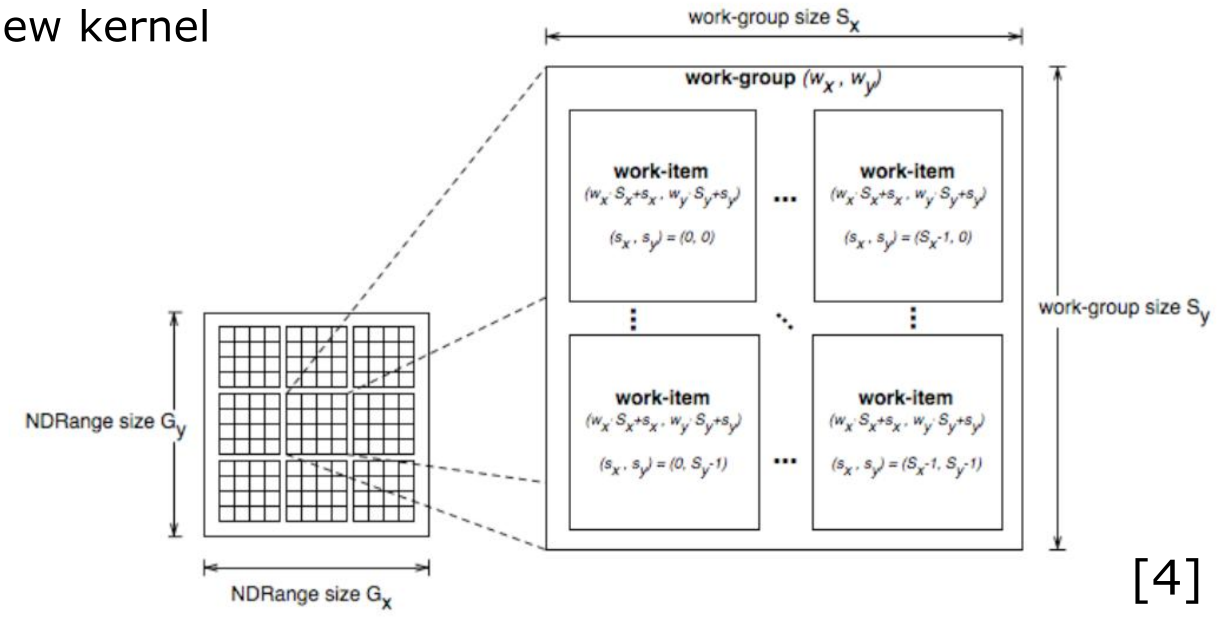
OpenCL Platform Model



- OpenCL exposes CPUs, GPUs, and other Accelerators as “devices”
- Each “device” contains one or more “compute units”, i.e. cores, SMs,...
- Each “compute unit” contains one or more SIMD “processing elements”

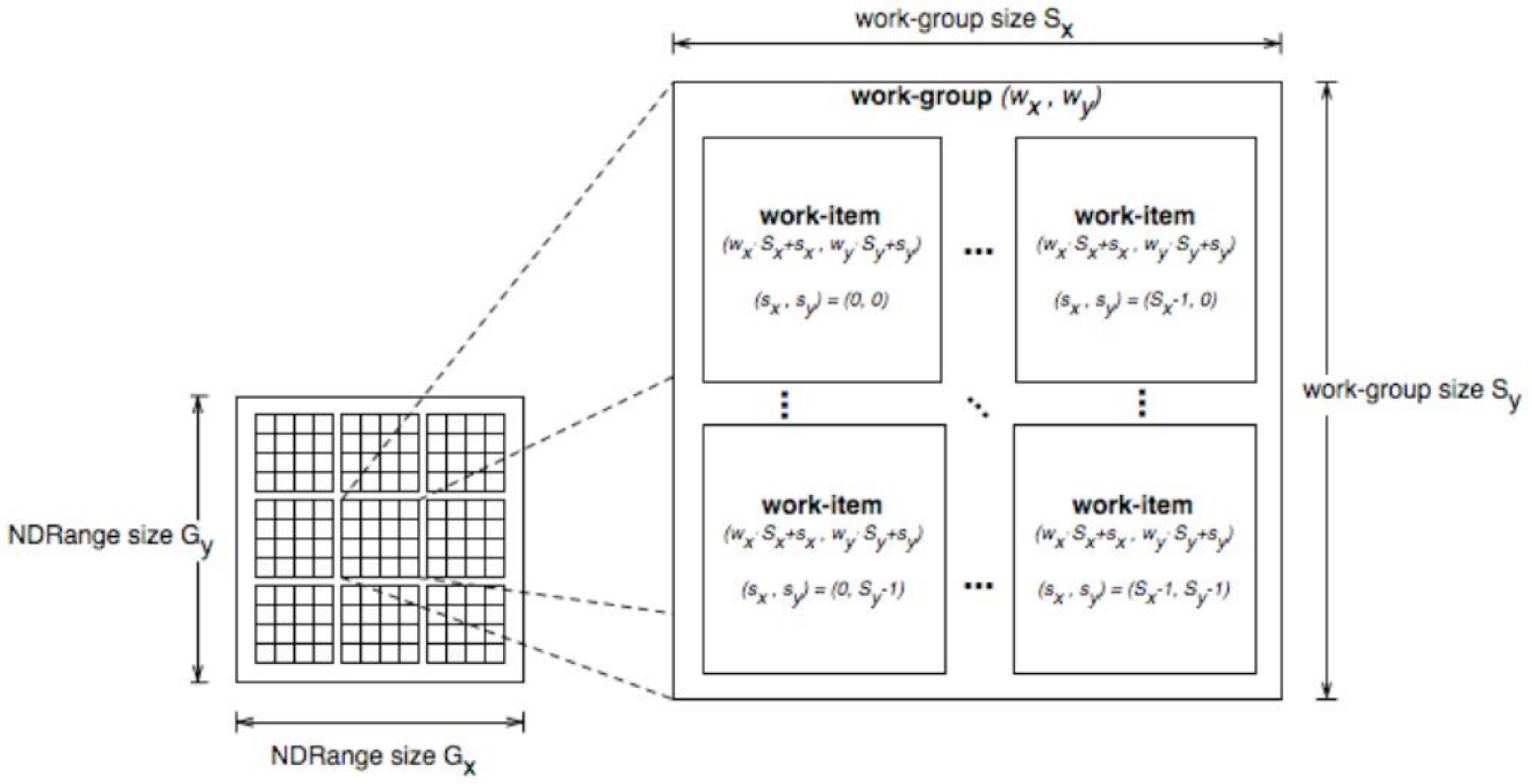
OpenCL Execution Model

- Parallel work is submitted to devices by launching kernels
- Kernels run over global dimension index ranges (NDRange), broken up into “work groups”, and “work items”
- Work items executing within the same work group can synchronize with each other with barriers or memory fences
- Work items in different work groups can’t sync with each other, except by launching a new kernel


work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-group size S_y
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
NDRange size G_y
NDRange size G_x

[4]

OpenCL Execution Model

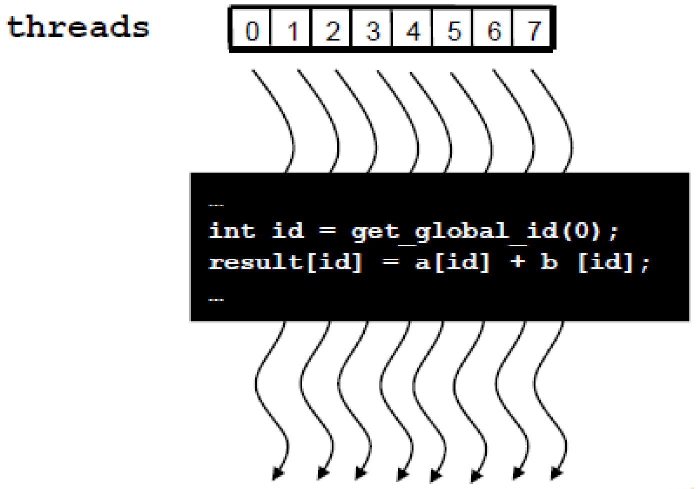

work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

An example of an NDRange index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs. [4]

OpenCL Execution Model

An OpenCL kernel is executed by an array of work items.

- All work items run the same code (SPMD)
- Each work item has an index that it uses to compute memory addresses and make control decisions



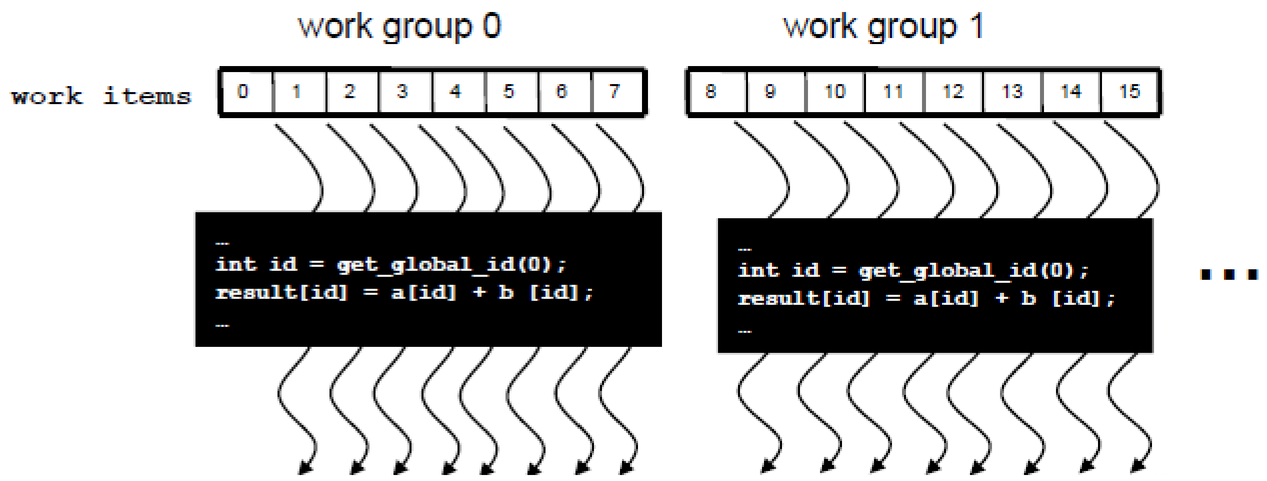
```
…
int id = get_global_id(0);
result[id] = a[id] + b [id];
…
```

[1]

Work Groups: Scalable Cooperation

Divide monolithic work item array into work groups

- Work items within a work group cooperate via **shared memory, atomic operations** and **barrier synchronization**
- Work items in different work groups cannot cooperate

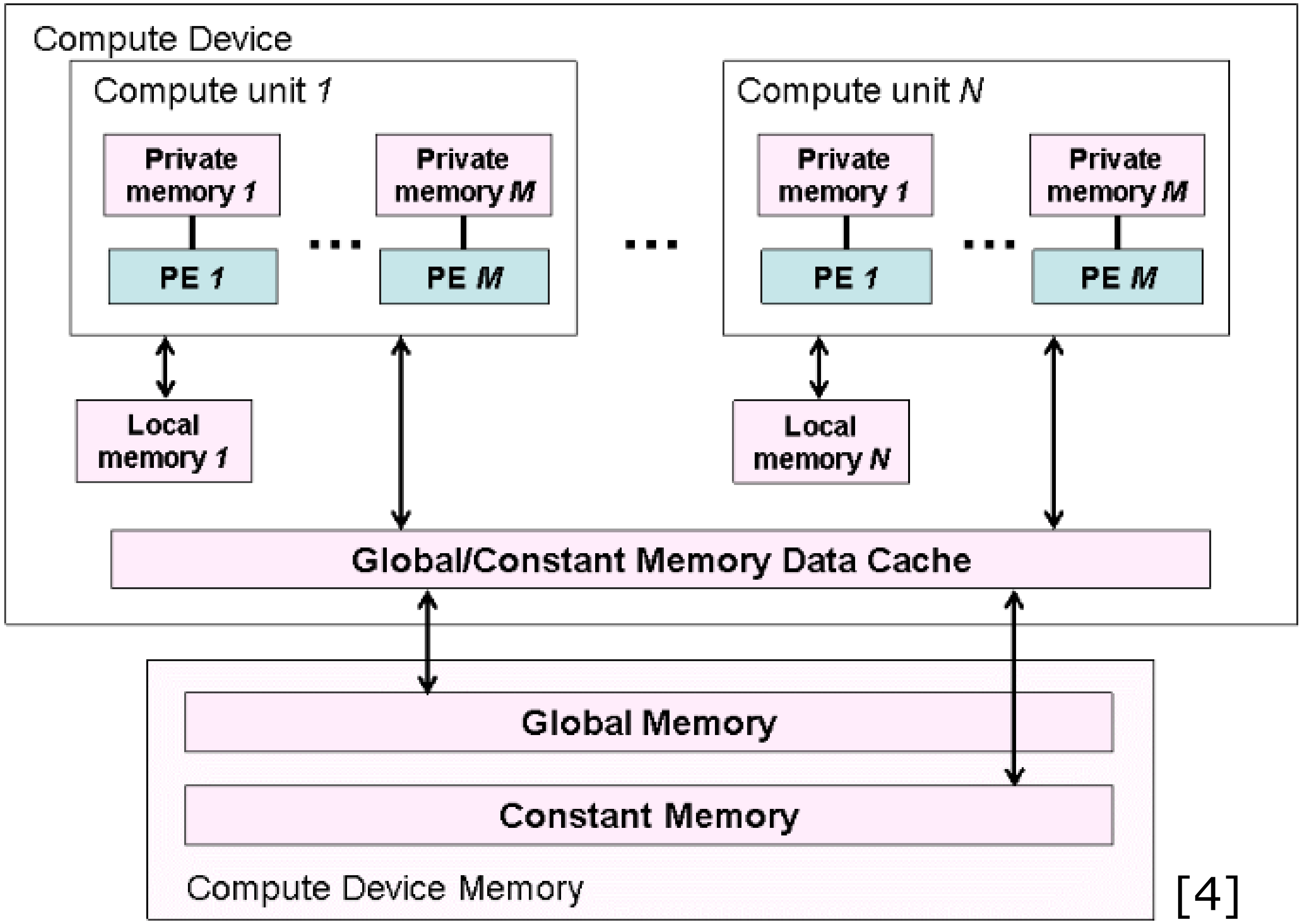


```
…
int id = get_global_id(0);
result[id] = a[id] + b [id];
…
```

```
…
int id = get_global_id(0);
result[id] = a[id] + b [id];
…
```

[1]

OpenCL Memory Architecture



[4]

Private

Per work-item

Local

Shared within a workgroup

Global/ Constant

Visible to all workgroups

Host Memory

On the CPU

OpenCL Memory Architecture

Hasso Plattner Institut

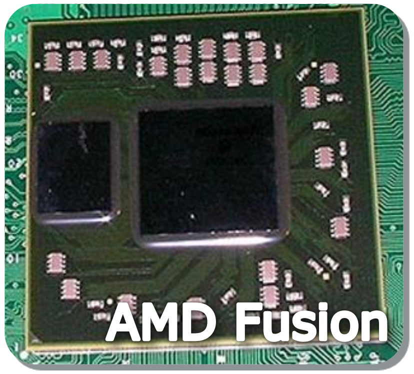
- Memory management is explicit: you must move data from host → global → local… and back

Memory Type	Keyword	Description/Characteristics
Global Memory	__global	Shared by all work items; read/write; may be cached (modern GPU), else slow; huge
Private Memory	__private	For local variables; per work item; may be mapped onto global memory (Arrays on GPU)
Local Memory	__local	Shared between workitems of a work group; may be mapped onto global memory (not GPU), else fast; small
Constant Memory	__constant	Read-only, cached; add. special kind for GPUs: texture memory

GPU Computing Platforms

AMD

R700, R800, R900



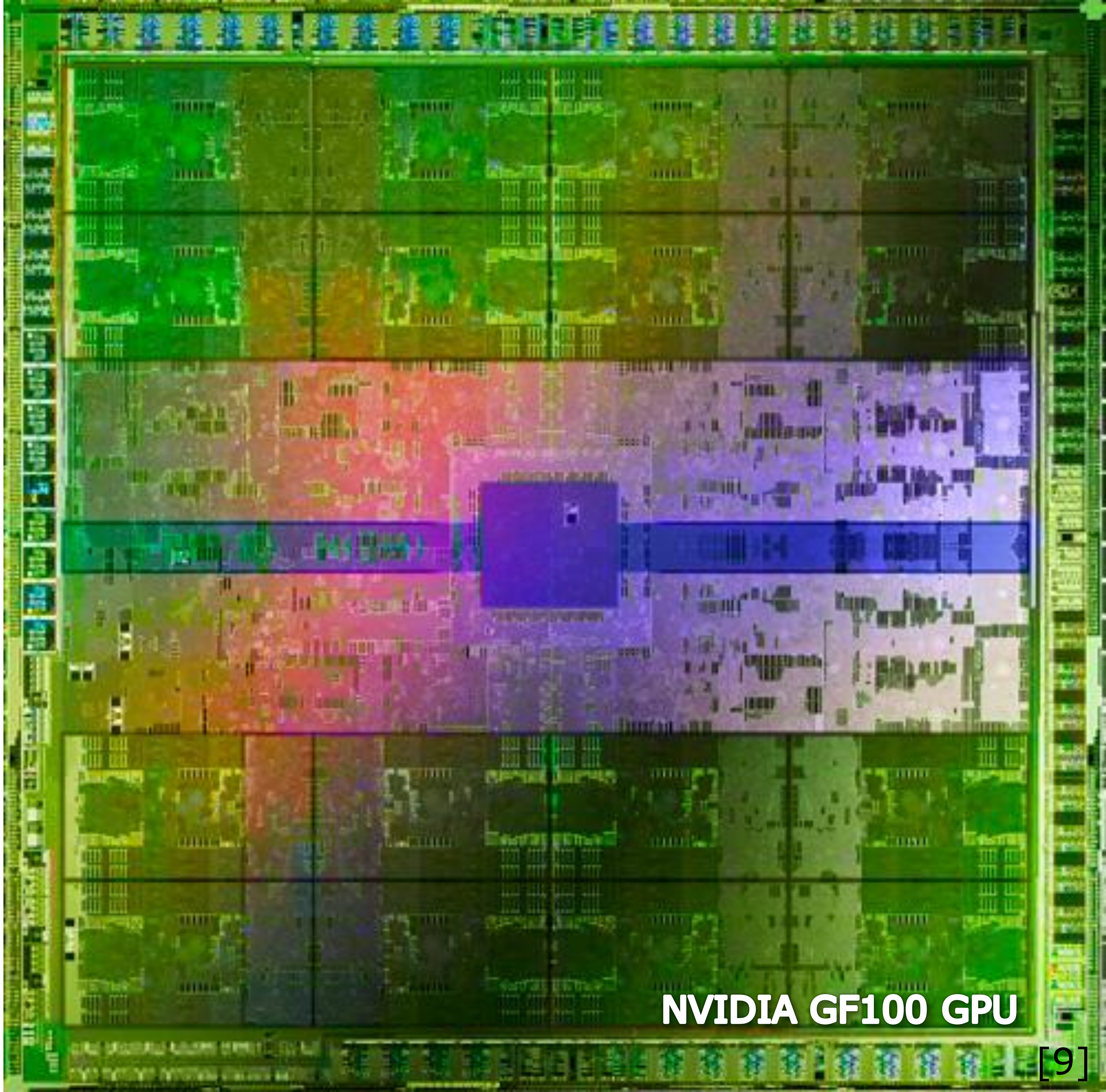
NVIDIA

G80, G92, GT200, GF100, GF110

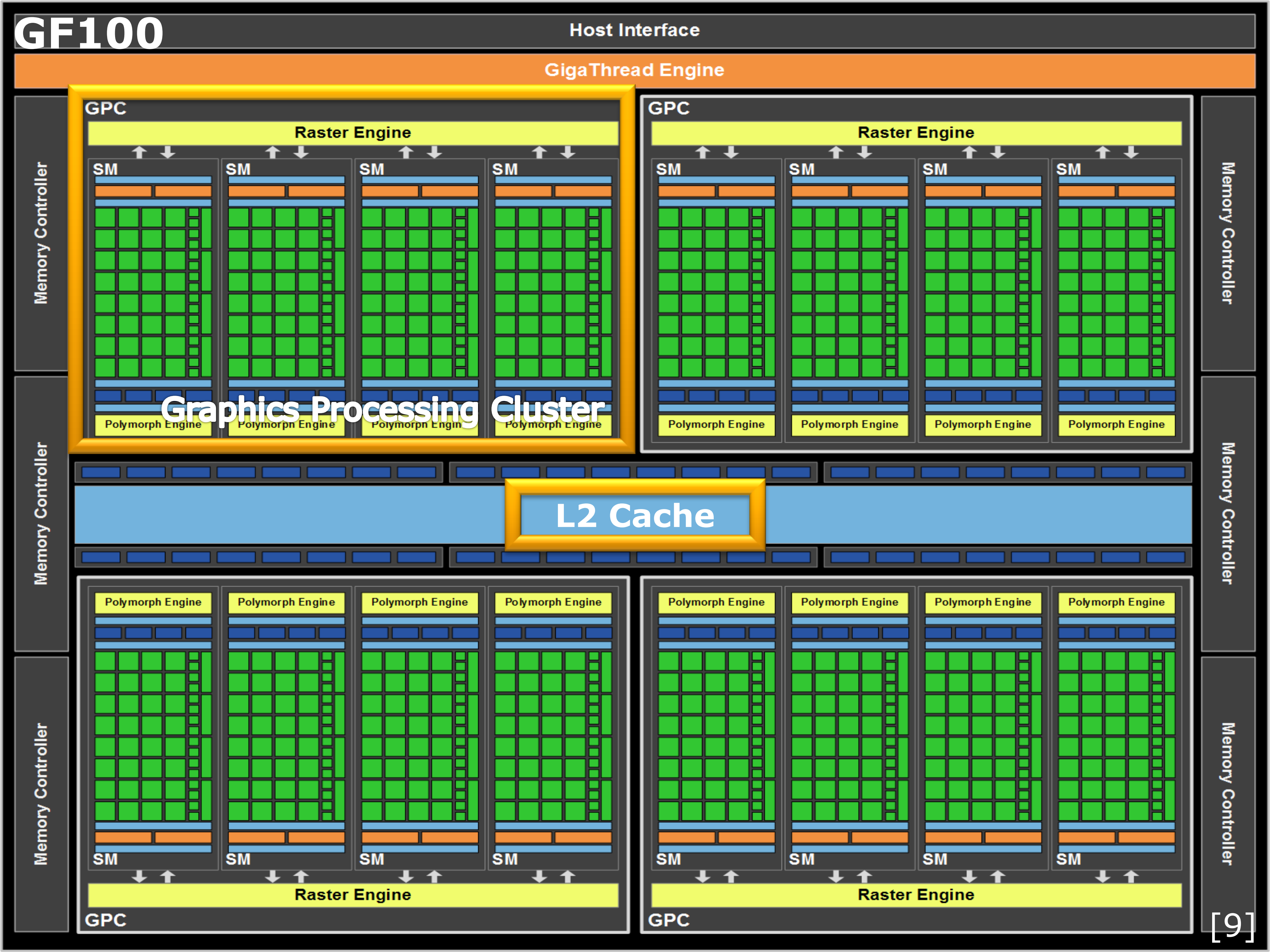
Geforce, Quadro,

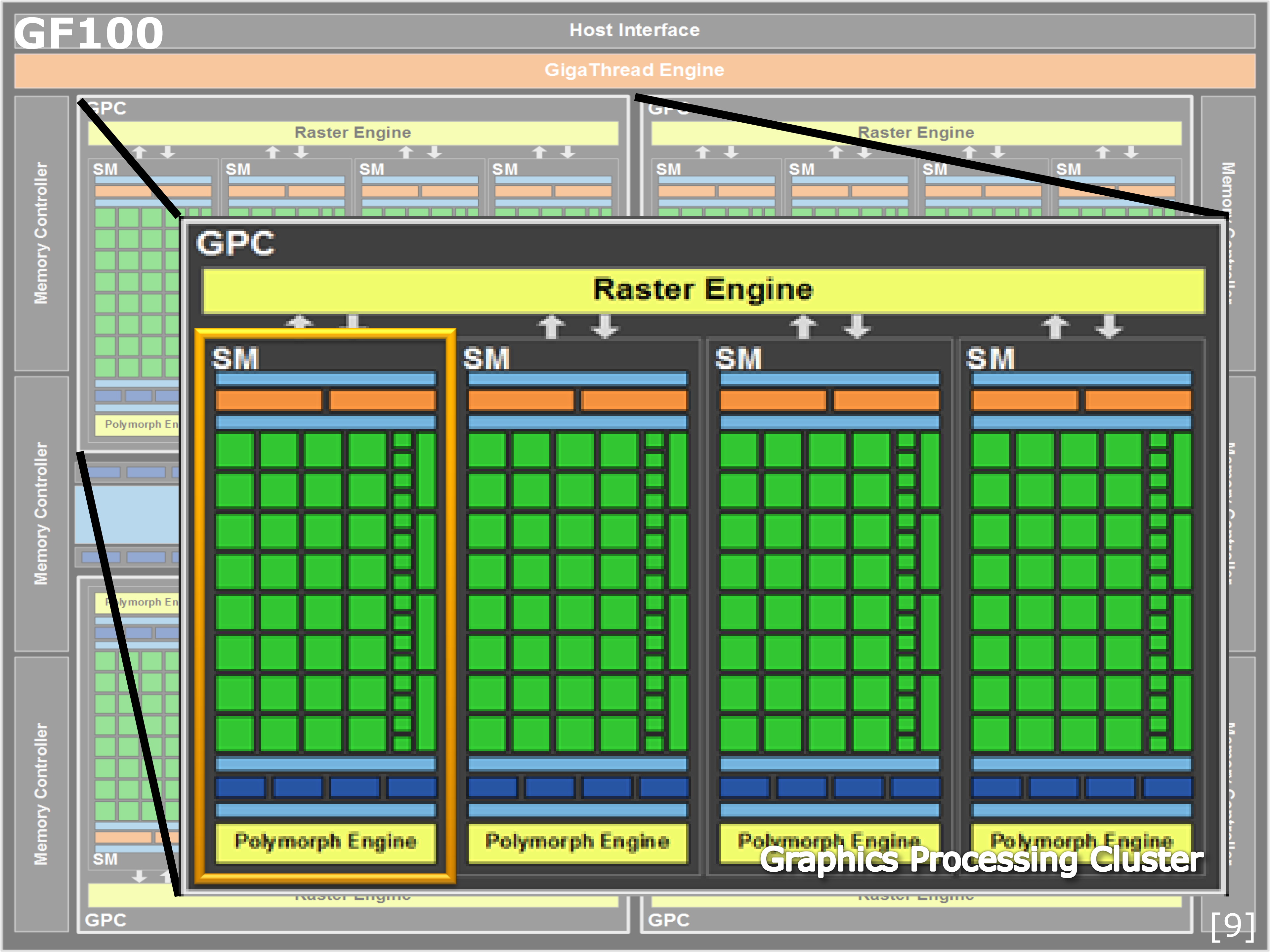
Tesla, ION

GPU Hardware in Detail



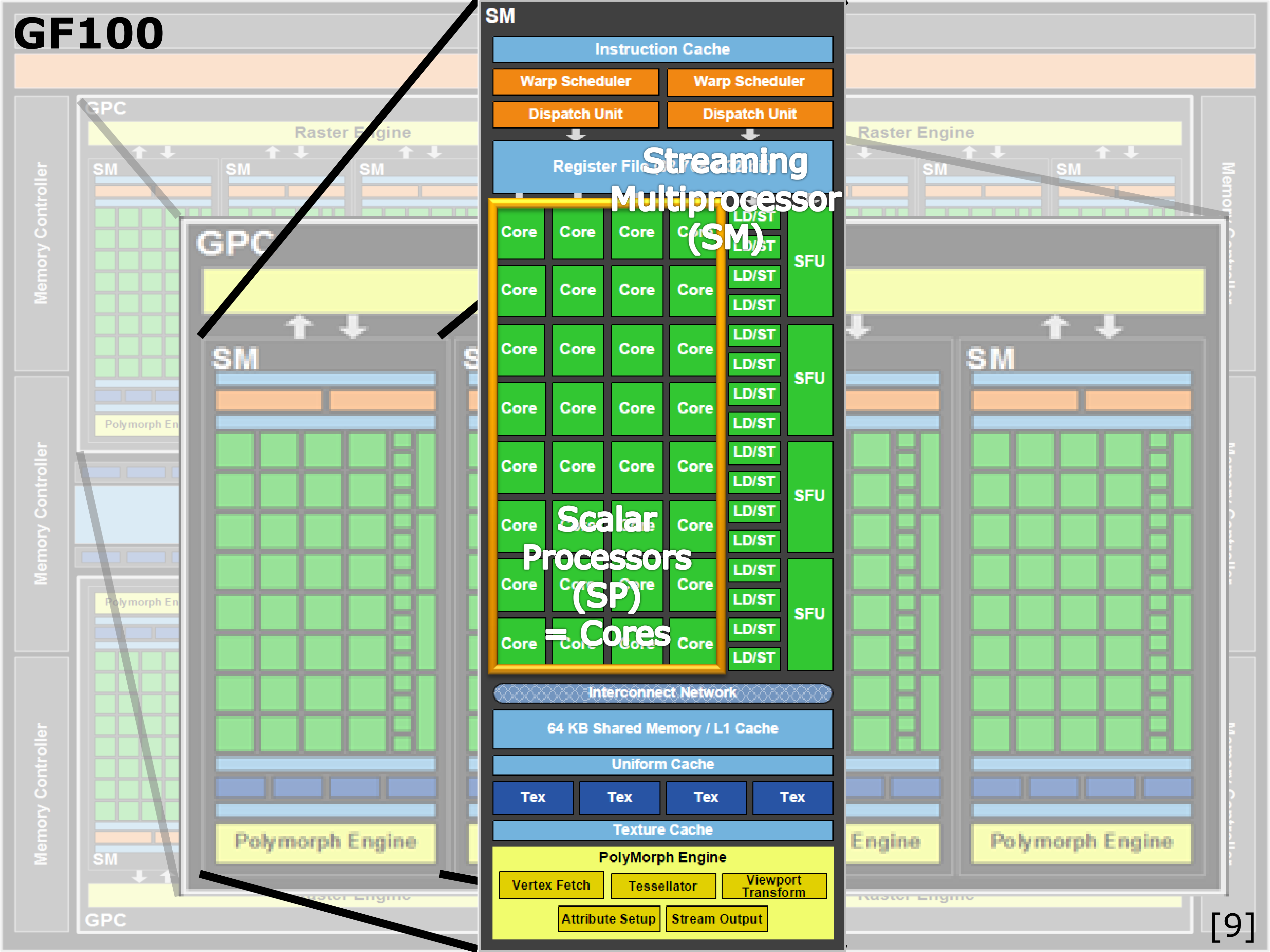
[9]


GF100
Host Interface
GigaThread Engine
GPC
Raster Engine
SM
Polymorph Engine
Graphics Processing Cluster
L2 Cache
Memory Controller
[9]


GF100
Host Interface
GigaThread Engine
GPC
Raster Engine
SM
Polymorph Engine
Memory Controller
GPC
Raster Engine
SM
SM
SM
SM
Polymorph Engine
Polymorph Engine
Polymorph Engine
Polymorph Engine
Graphics Processing Cluster

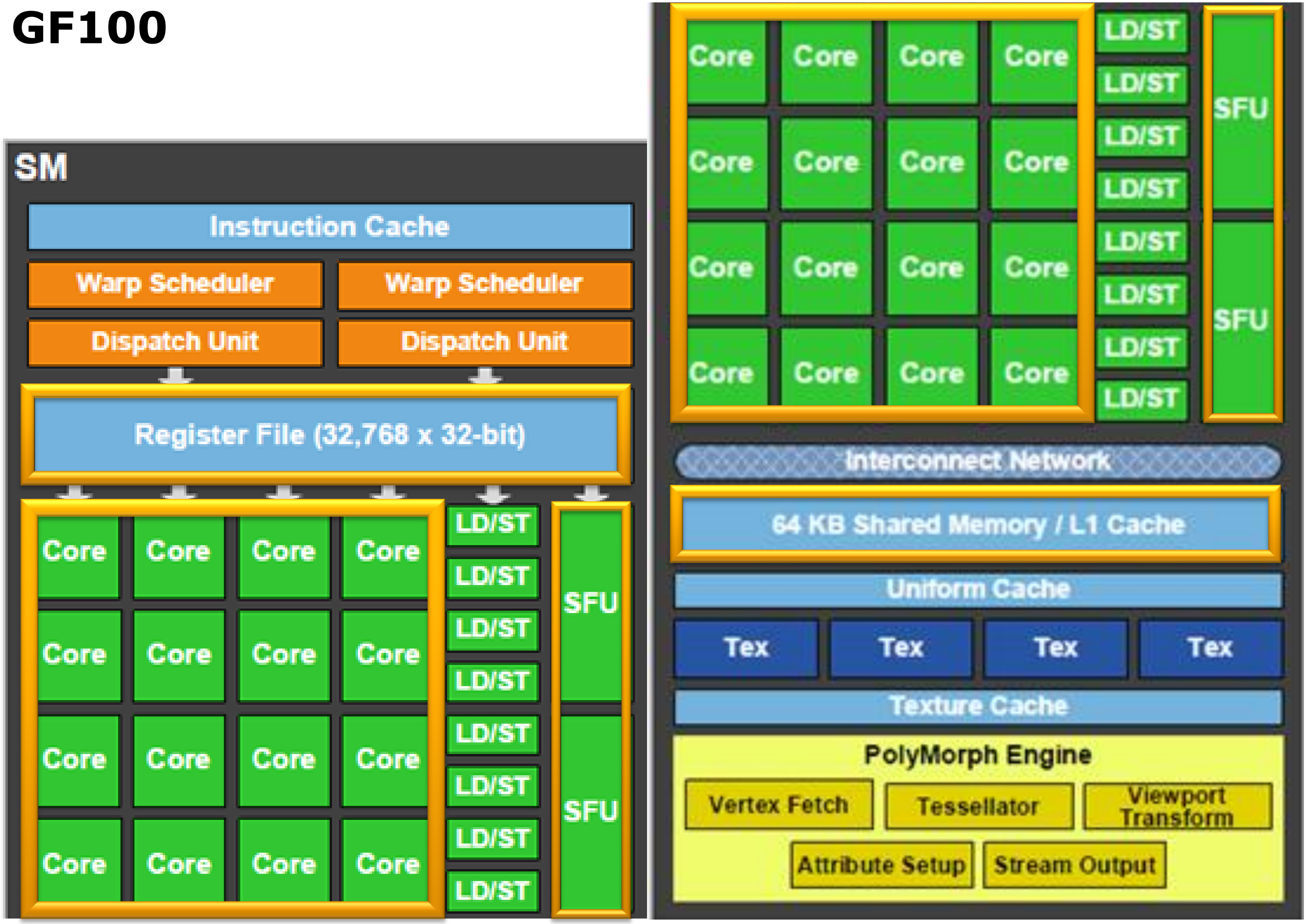
[9]

GF100


GPC
Raster Engine
SM
Memory Controller
Polymorph Engine
SM
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File
Streaming Multiprocessor (SM)
Core
LD/ST
SFU
Scalar Processors (SP) = Cores
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache
Tex
Tex
Tex
Tex
Texture Cache
PolyMorph Engine
Vertex Fetch
Tessellator
Viewport Transform
Attribute Setup
Stream Output

[9]

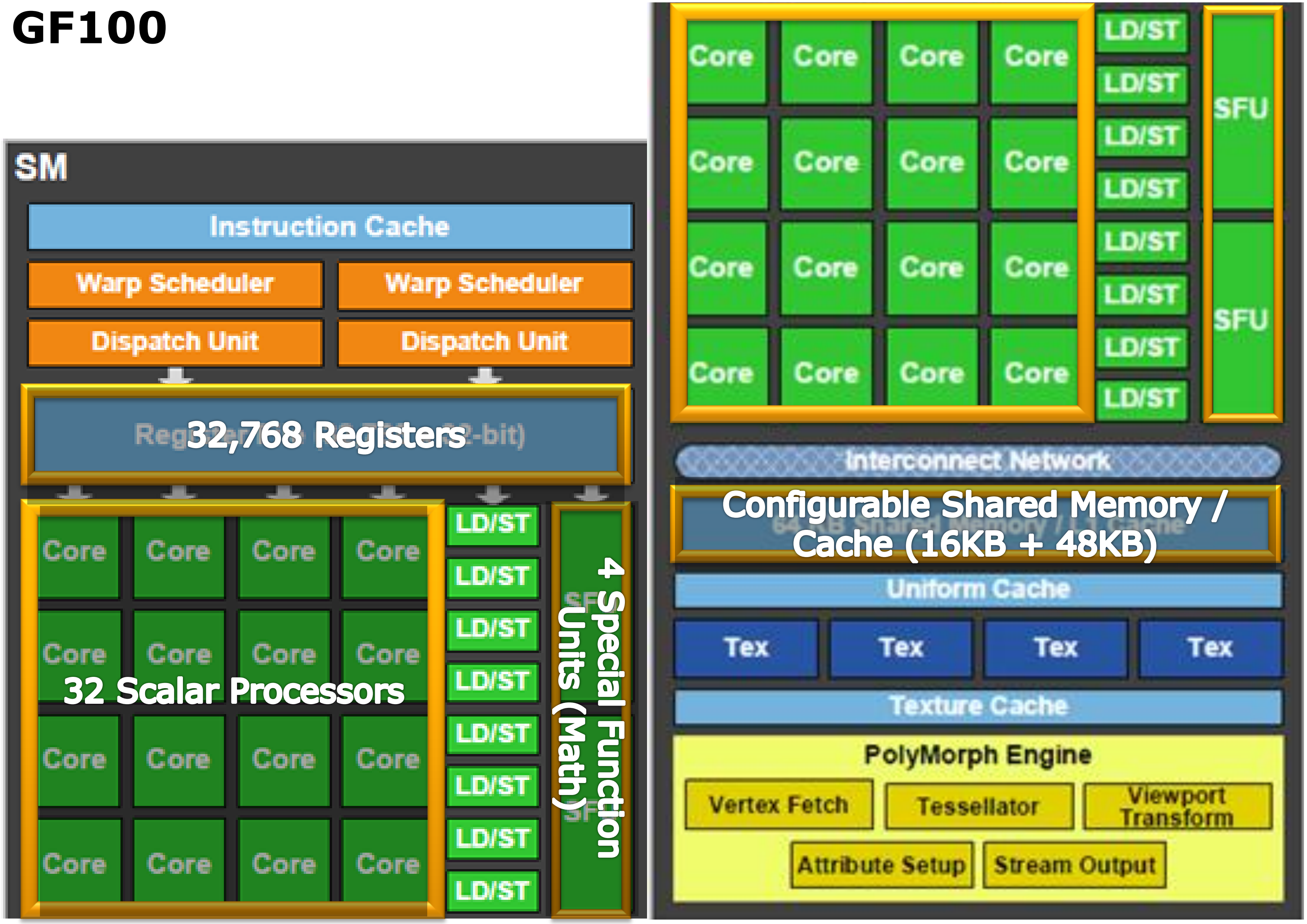
GF100


SM
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
Core
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache
Tex
Tex
Tex
Tex
Texture Cache
PolyMorph Engine
Vertex Fetch
Tessellator
Viewport Transform
Attribute Setup
Stream Output

...

[9]

GF100


SM
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
32,768 Registers
Core
32 Scalar Processors
LD/ST
4 Special Function Units (Math)
SFU
Interconnect Network
Configurable Shared Memory / Cache (16KB + 48KB)
Uniform Cache
Tex
Texture Cache
PolyMorph Engine
Vertex Fetch
Tessellator
Viewport Transform
Attribute Setup
Stream Output

...

[9]

GT200 – previous architecture

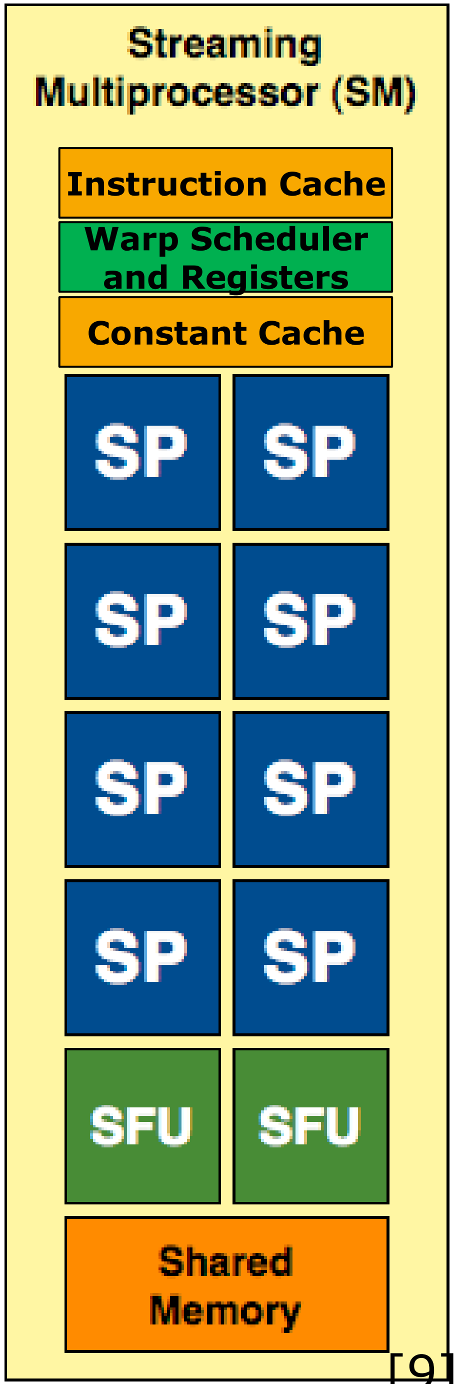
Simpler architecture, but same principles

Several Work Groups reside on one SM

- Amount depends on available resources (Shared Memory (=Local Memory in OpenCL), Registers)
- More Work Groups → better latency hiding
  - Latencies occur for memory accesses, pipelined floating-point arithmetic and branch instructions

Thread execution in “Warps” (called “wavefronts” on AMD)

- Native execution size (32 Threads for NVIDIA)
- Zero-Overhead Thread Scheduling: If one warp stalls (accesses memory) next warp is selected for execution



[9]

Warp Execution Example

Application creates 200.000 „Tasks"

→ Global Work Group Size: 200.000 Work Items

Programmer decides to use a Local Work Group Size of 100 Work Items

→ Number of Work Groups: 2.000 Work Groups

One Work Item requires 10 registers and 20 byte of Shared Memory; a SM has 16 KB of Shared Memory and 16.384 registers

→ Number of Work Items per SM: 16KB/20B = 819 Work Items

→ Number of Work Groups per SM: 819/100 = 8 Work Groups per SM

Even if 7 Work Groups are waiting for memory, 1 can be executed.

Warp Execution Example

Each of the Work Groups contains 100 Work Items; the Warp Size (native execution size of a SM) is 32

→ Number of Threads Executed in parallel:	32 Threads
→ Number of „Rounds“ to execute a Work Group:	100/32 = 4
→ Threads running in the first 3 rounds:	32 Threads
→ Threads running in the last round:	100-32*4=4 Threads

If one of the threads accesses memory: whole warp stalls

If one of the threads follows a differing execution path: it is executed in an additional seperate round

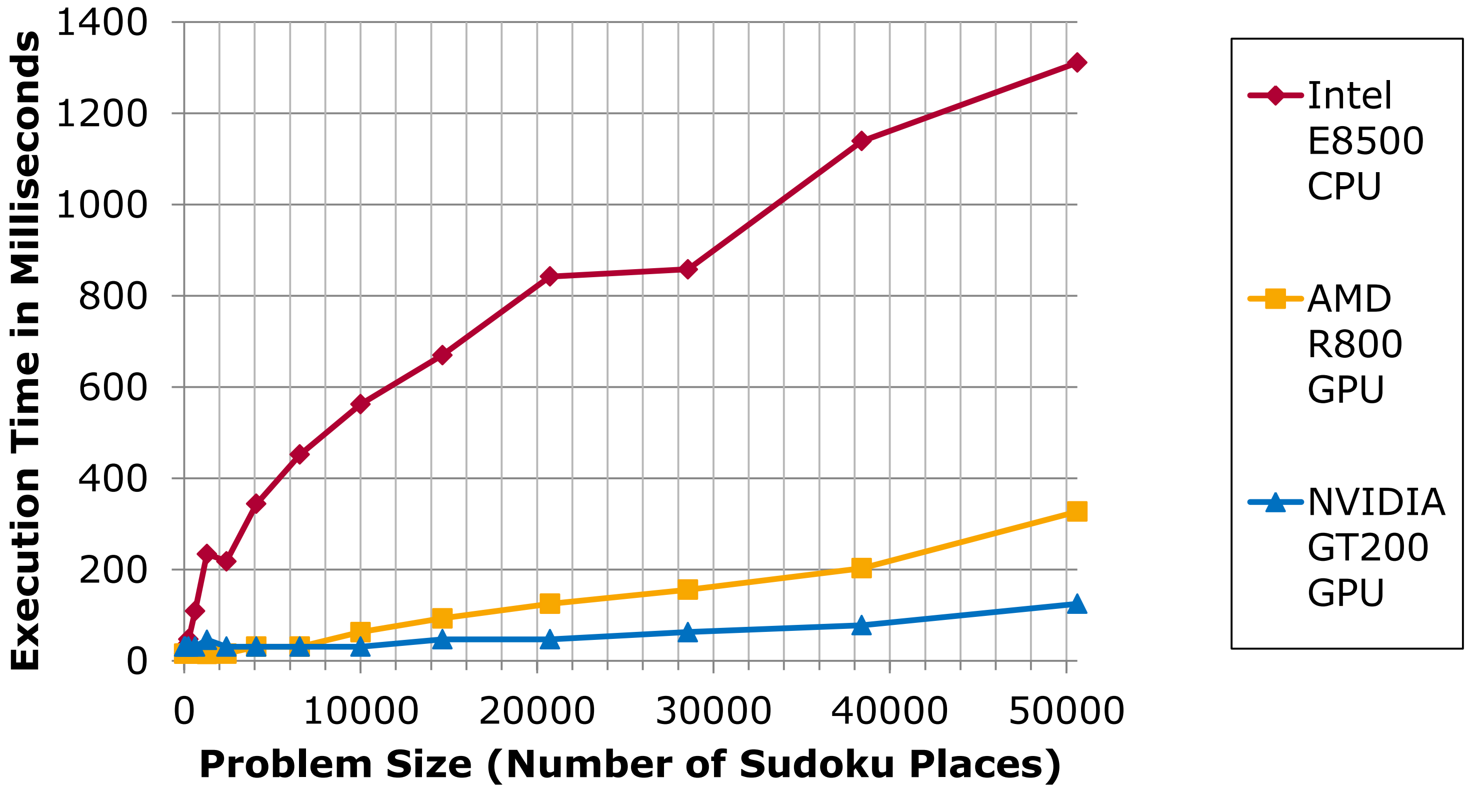
Compute Capability by version

	1.0	1.1	1.2	1.3	2.0
double precision floating point operations	No			Yes	
caches	No				Yes
max # concurrent kernels	1				8
max # threads per block	512				1024
max # Warps per MP	24		32		48
max # Threads per MP	768		1024		1536
register count (32 bit)	8192		16384		32768
max shared mem per MP	16KB				48KB
# shared memory banks	16				32

Plus: varying amounts of cores, global memory sizes, bandwidth, clock speeds (core, memory), bus width, memory access penalties …

The Power of GPU Computing

big performance gains for small problem sizes

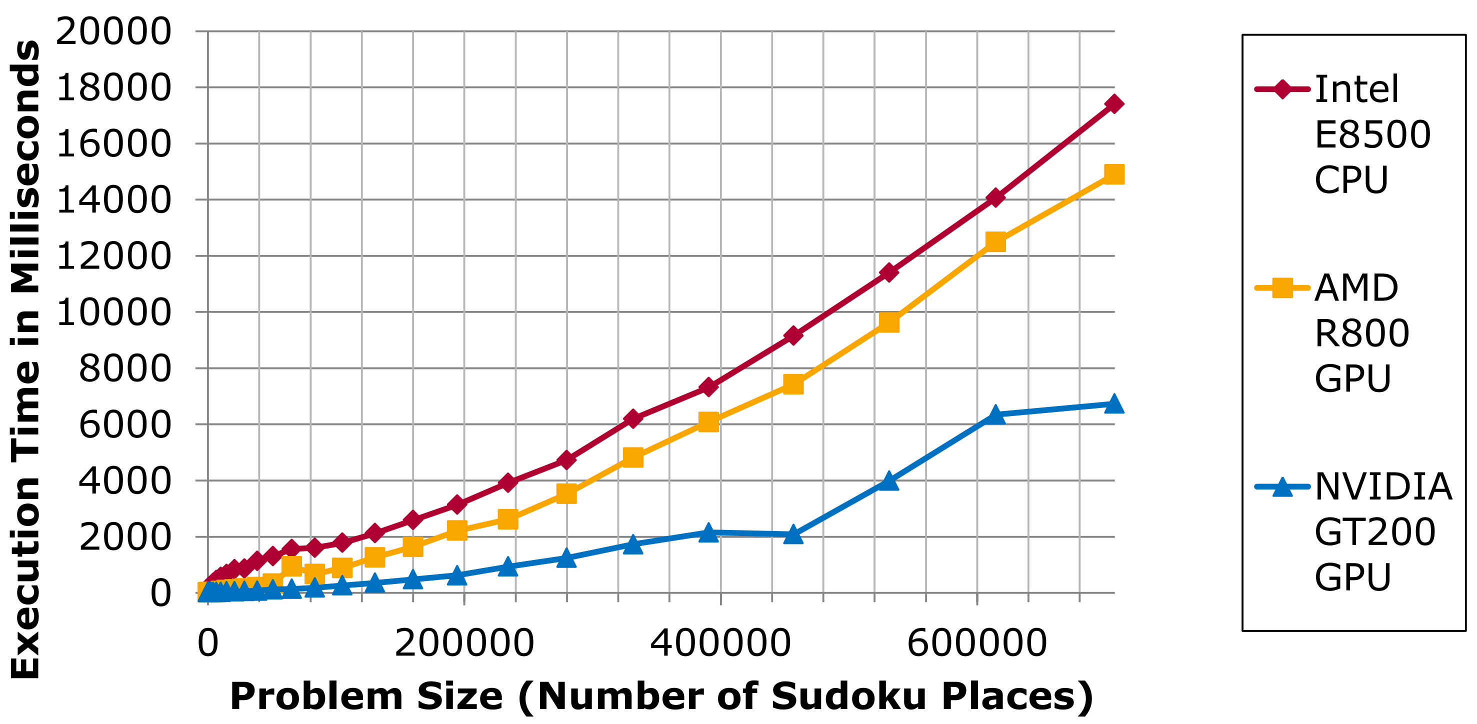


* less is better

The Power of GPU Computing

small/moderate performance gains for large problem sizes

→ further optimizations needed



* less is better

Best Practices for Performance Tuning

Algorithm Design	• Asynchronous, Recompute, Simple
Memory Transfer	• Chaining, Overlap Transfer & Compute
Control Flow	• Divergent Branching, Predication
Memory Types	• Local Memory as Cache, rare resource
Memory Access	• Coalescing, Bank Conflicts
Sizing	• Execution Size, Evaluation
Instructions	• Shifting, Fused Multiply, Vector Types
Precision	• Native Math Functions, Build Options

Divergent Branching and Predication

Divergent Branching

- Flow control instruction (`if`, `switch`, `do`, `for`, `while`) can result in different execution paths
- ➢ Data parallel execution → varying execution paths will be serialized
- ➢ Threads converge back to same execution path after completion

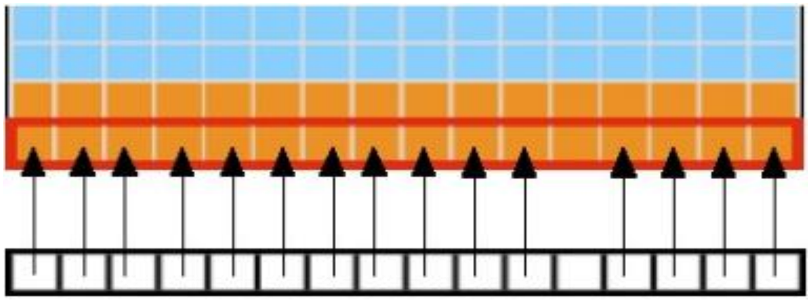
Branch Predication

- Instructions are associated with a per-thread condition code (predicate)
  - All instructions are scheduled for execution
  - Predicate true: executed normally
  - Predicate false: do not write results, do not evaluate addresses, do not read operands
- Compiler may use branch predication for `if` or `switch` statements
- Unroll loops yourself (or use `#pragma unroll` for NVIDIA)

Coalesced Memory Accesses

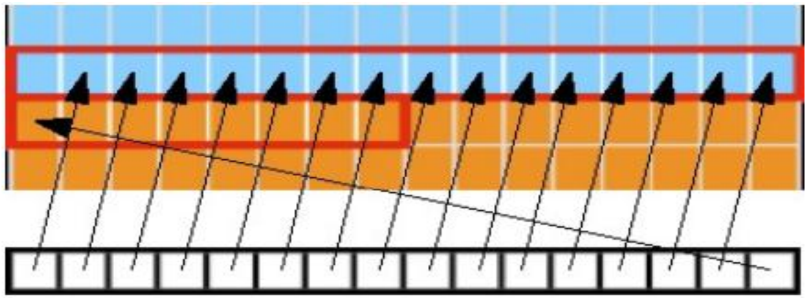
Simple Access Pattern

- Can be fetched in a single 64-byte transaction (red rectangle)
- Could also be permuted *



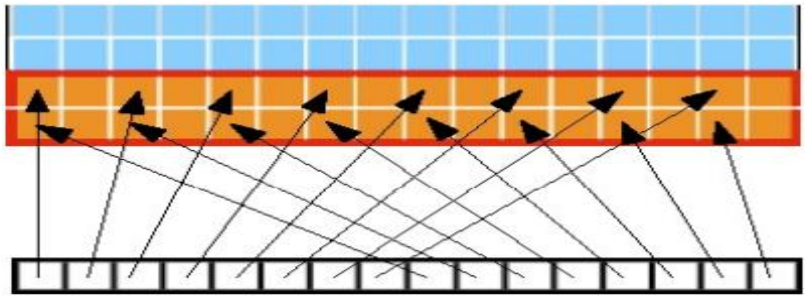
Sequential but Misaligned Access

- Fall into single 128-byte segment: single 128-byte transaction, else: 64-byte transaction + 32-byte transaction *



Strided Accesses

- Depending on stride from 1 (here) up to 16 transactions *



* 16 transactions with compute capability 1.1

[6]

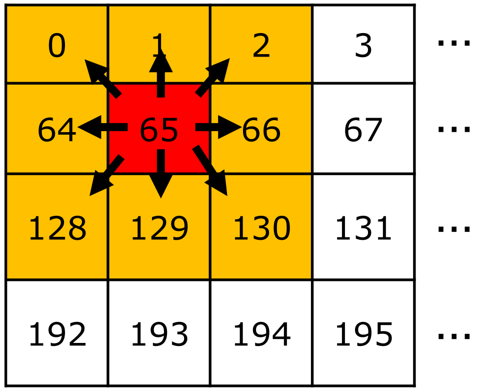
Use Caching: Local, Texture, Constant

Local Memory

- Memory latency roughly 100x lower than global memory latency
- Small, no coalescing problems, prone to memory bank conflicts

Texture Memory

- 2-dimensionally cached, read-only
- Can be used to avoid uncoalesced loads form global memory
- Used with the image data type

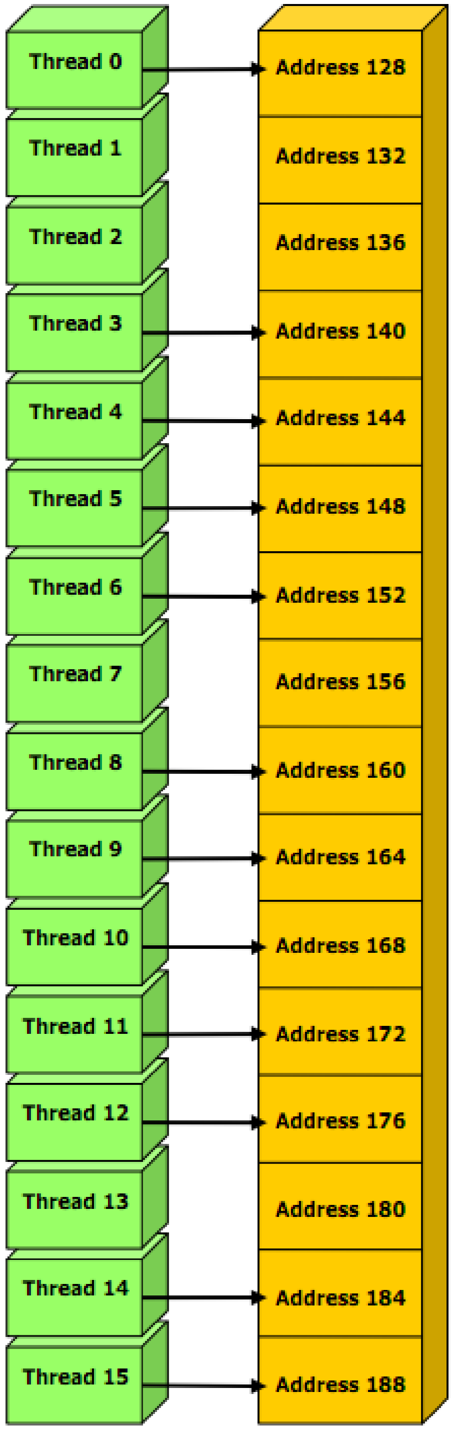


Constant Memory

- Lineary cached, read-only, 64 KB
- as fast as reading from a register for the same address
- Can be used for big lists of input arguments

Memory Bank Conflicts

- Access to (Shared) Memory is implemented via hardware memory banks
- If a thread accesses a memory address this is handled by the responsible memory bank
- Simple Access Patterns like this one are fetched in a single transaction



[8]

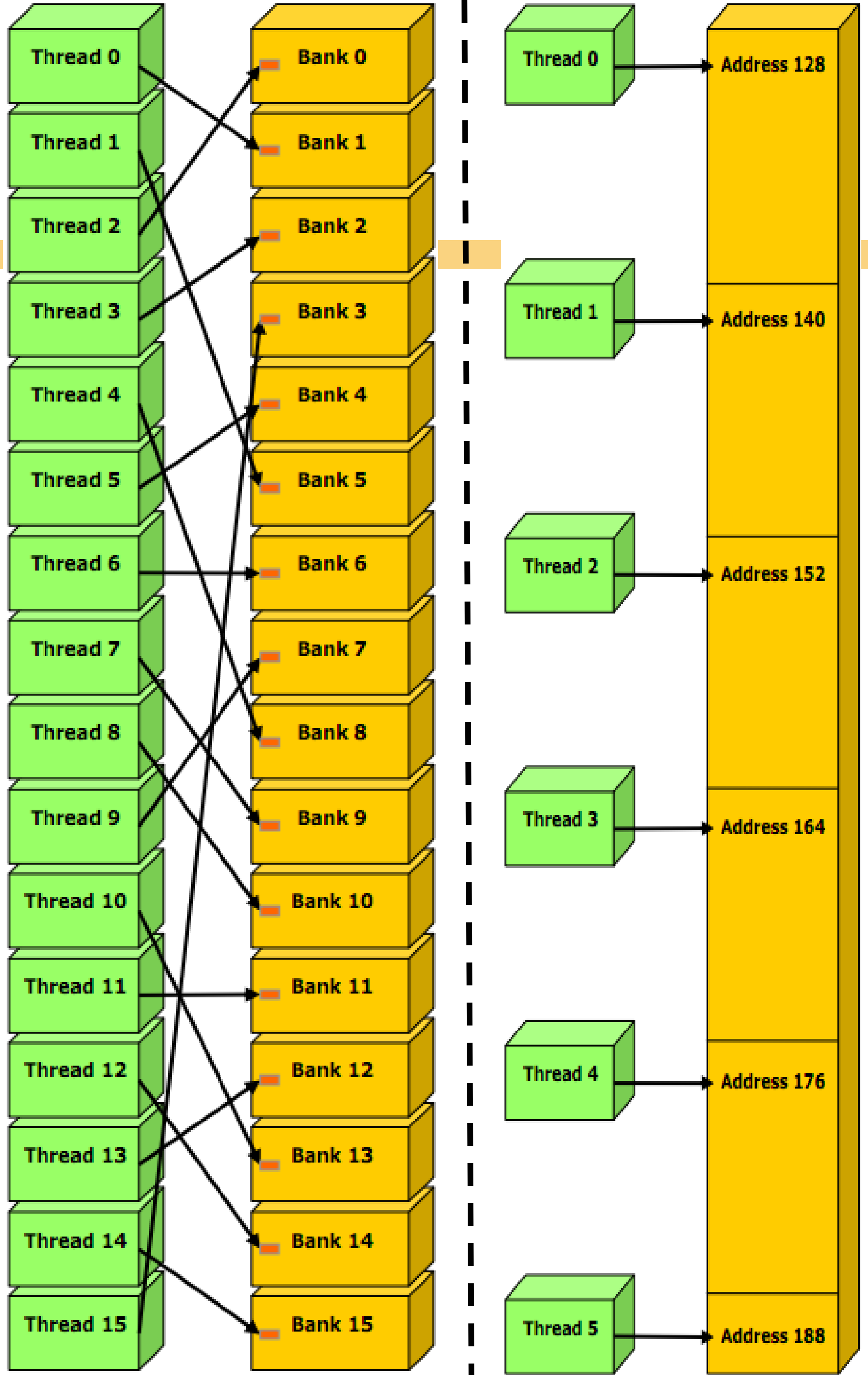
Memory Bank Conflicts

Permuted Memory Access (left)

- Still one transaction on cards with compute capability >=1.2; otherwise 16 transactions are required

Strided Memory Access (right)

- Still one transaction on cards with compute capability >=1.2; otherwise 16 transactions are required



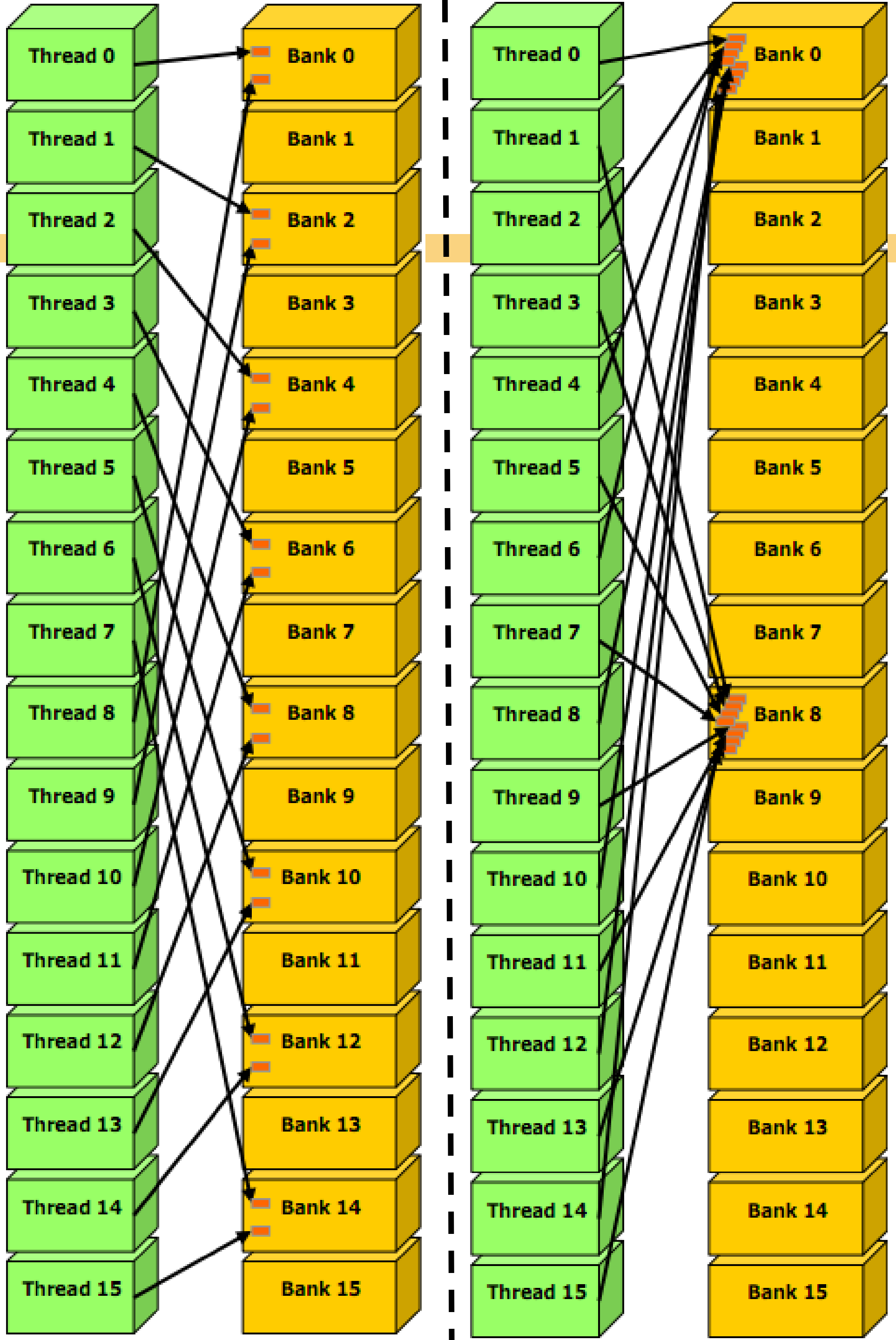
[8]

Memory Bank Conflicts

Bank conflicts

- Left figure: 2 bank conflicts → resulting bandwidth is ½ of the original bandwidth

- Right figure: 8 bank conflicts → resulting bandwidth is 1/8 of the original bandwidth



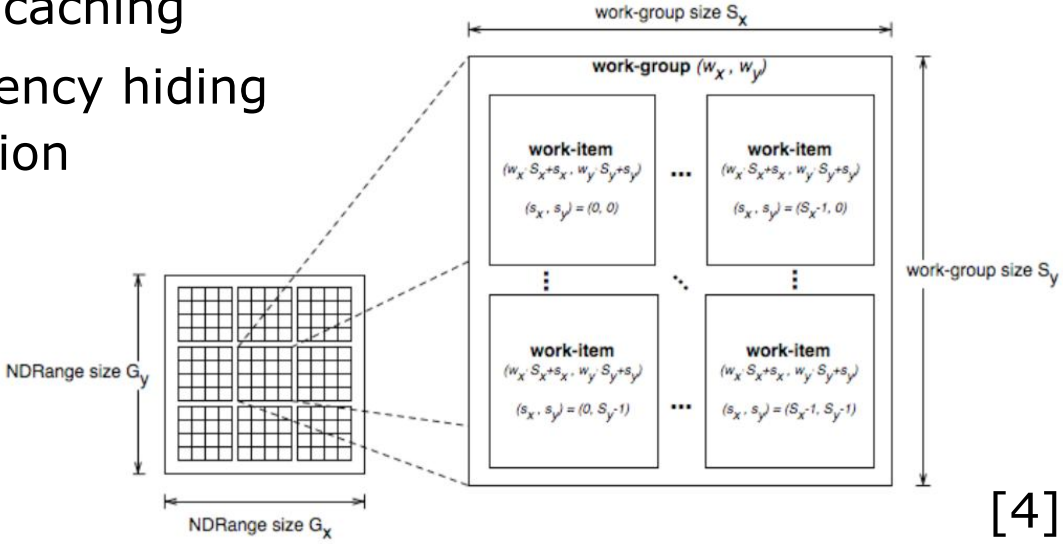
[8]

Sizing: What is the right execution layout?

- Local work item count should be a multiple of native execution size (NVIDIA 32, AMD 64), but not to big
- Number of work groups should be multiple of the number of multiprocessors (hundreds or thousands of work groups)

- Can be configured in 1-, 2- or 3-dimensional layout: consider access patterns and caching
- Balance between latency hiding and resource utilization

- Experimenting is required!


work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (S_x - 1, 0)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (0, S_y - 1)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (S_x - 1, S_y - 1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

[4]

Instructions and Precision

- Single precision floats provide best performance
- Use shift operations to avoid expensive division and modulo calculations
- Special compiler flags
- AMD has native vector type implementation; NVIDIA is scalar
- Use the native math library whenever speed trumps precision

Functions	**Throughput**
single-precision floating-point add, multiply, and multiply-add	**8** operations per clock cycle
single-precision reciprocal, reciprocal square root, and native_logf(x)	**2** operations per clock cycle
native_sin, native_cos, native_exp	**1** operation per clock cycle

Further Readings

http://www.dcl.hpi.uni-potsdam.de/research/gpureadings/

- **[1] Kirk, D. B. & Hwu, W. W., 2010. *Programming Massively Parallel Processors: A Hands-on Approach*. 1 ed. Morgan Kaufmann.**
- [2] Herlihy, M. & Shavit, N., 2008. *The Art of Multiprocessor Programming*.
- **[3] Sanders, J. & Kandrot, E., 2010. *CUDA by Example: An Introduction to General-Purpose GPU Programming* . 1 ed. Addison-Wesley Professional.**
- [4] Munshi, A. (ed.), 2010. The OpenCL Specification - v1.1. The Khronos Group Inc.
- **[5] Mattson, T., 2010. The Future of Many Core Computing: Software for many core processors.**
- [6] NVIDIA, 2009. NVIDIA OpenCL Best Practices Guide - Version 2.3.
- [7] Rob Farber, 2008. CUDA, Supercomputing for the Masses. Dr. Dobb's
- [8] NVIDIA, 2010. OpenCL Programming for the CUDA Architecture - Version 3.1
- [9] Ryan Smith, NVIDIA's GeForce GTX 480 and GTX 470: 6 Months Late, Was It Worth the Wait?