



**Hasso  
Plattner  
Institut**

IT Systems Engineering | Universität Potsdam

## Parallel Programming Concepts

## GPU Compute Devices

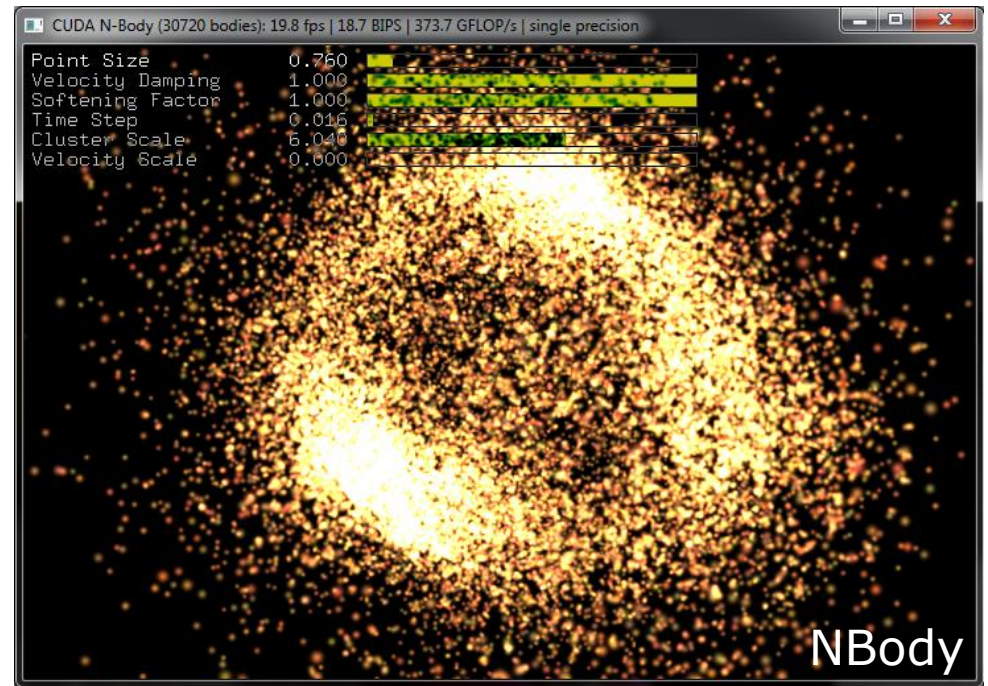
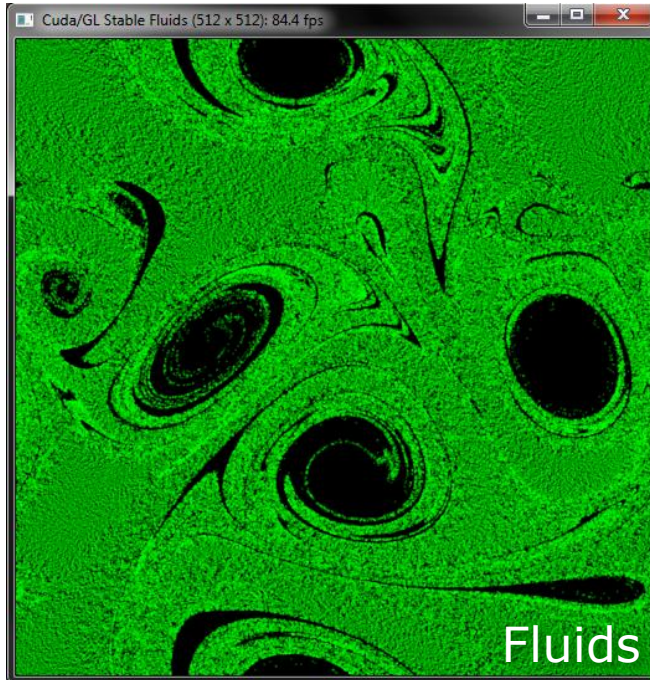
Frank Feinbube

Operating Systems and Middleware

Prof. Dr. Andreas Polze

# The Power of GPU Compute Devices

5



```
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 elements, NumDevsUsed = 1, Workgroup = 256
PASSED
```

RadixSort

# Wide Variety of Application Domains

6

The image displays a grid of 12 application domains, each with a representative image and a title. A central overlay lists the following domains: Research, Medical, Video and Photo, Energy, Finance, and Military.

Thumbnail	Title	Scale
	Statistical constraints on binary black hole inspi...	50 x
	Software GPU: Using graphics processing units i...	5 x
	Acceleration of the Smith-Waterman Algorithm using...	420 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	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

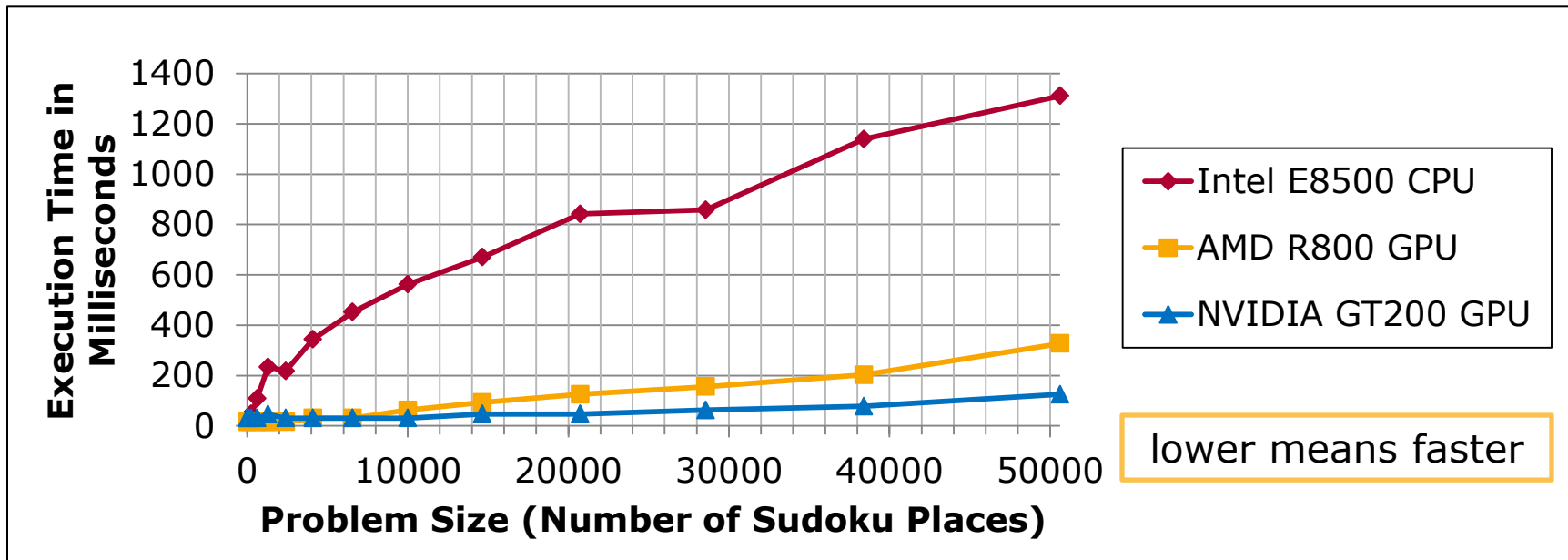
[http://www.nvidia.com/object/cuda\\_apps\\_flash\\_new.html](http://www.nvidia.com/object/cuda_apps_flash_new.html)  
[http://www.nvidia.com/object/tesla\\_testimonials.html](http://www.nvidia.com/object/tesla_testimonials.html)

# Why GPU Compute Devices?

## Short Term View: Cheap Performance

7

### Performance

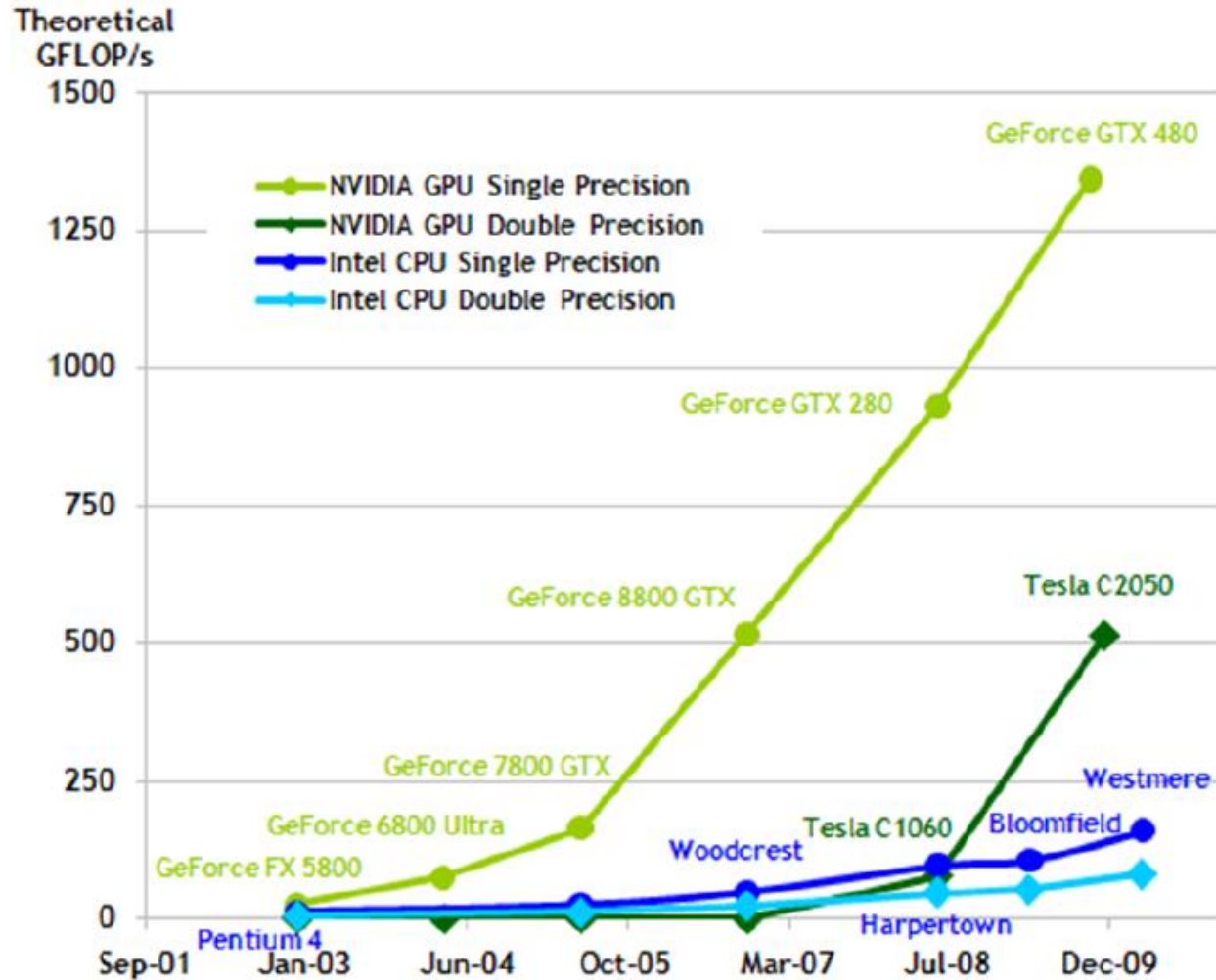


### Energy / Price

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

# Why GPU Compute Devices? Middle Term View: More Performance

8



# Why GPU Compute Devices?

## Long Term View: Hybrid Computing

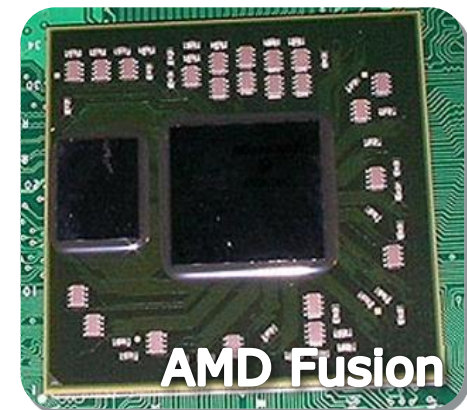
9

Dealing with massively 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

10

## 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)

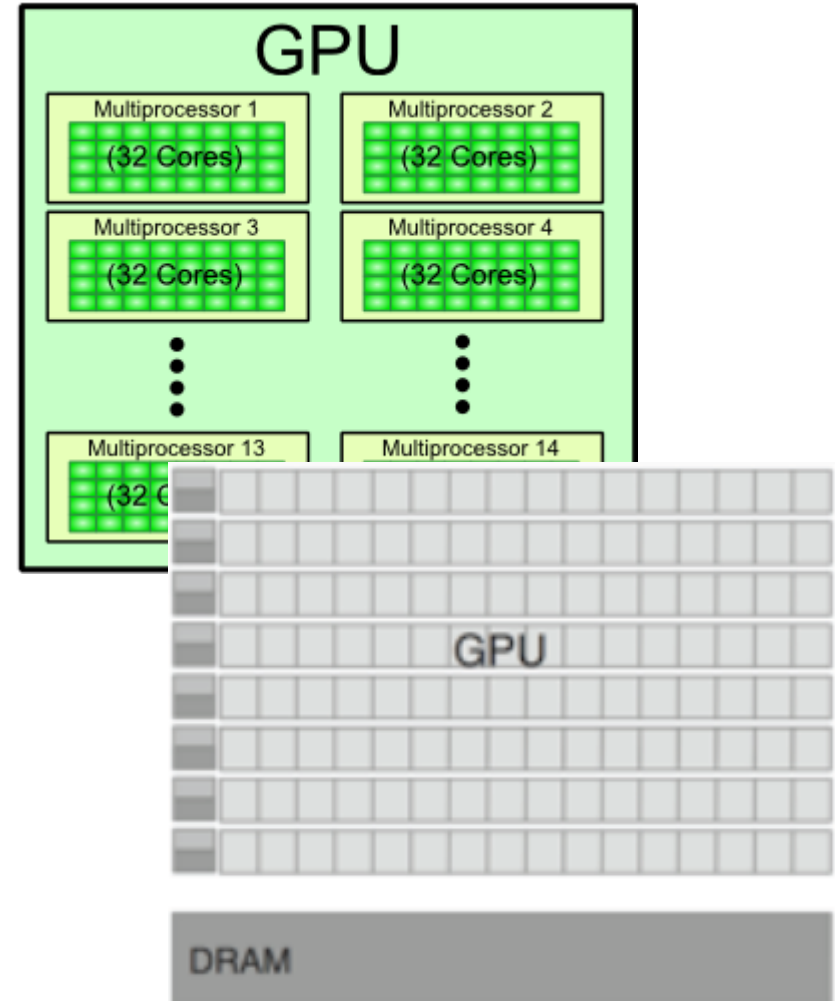
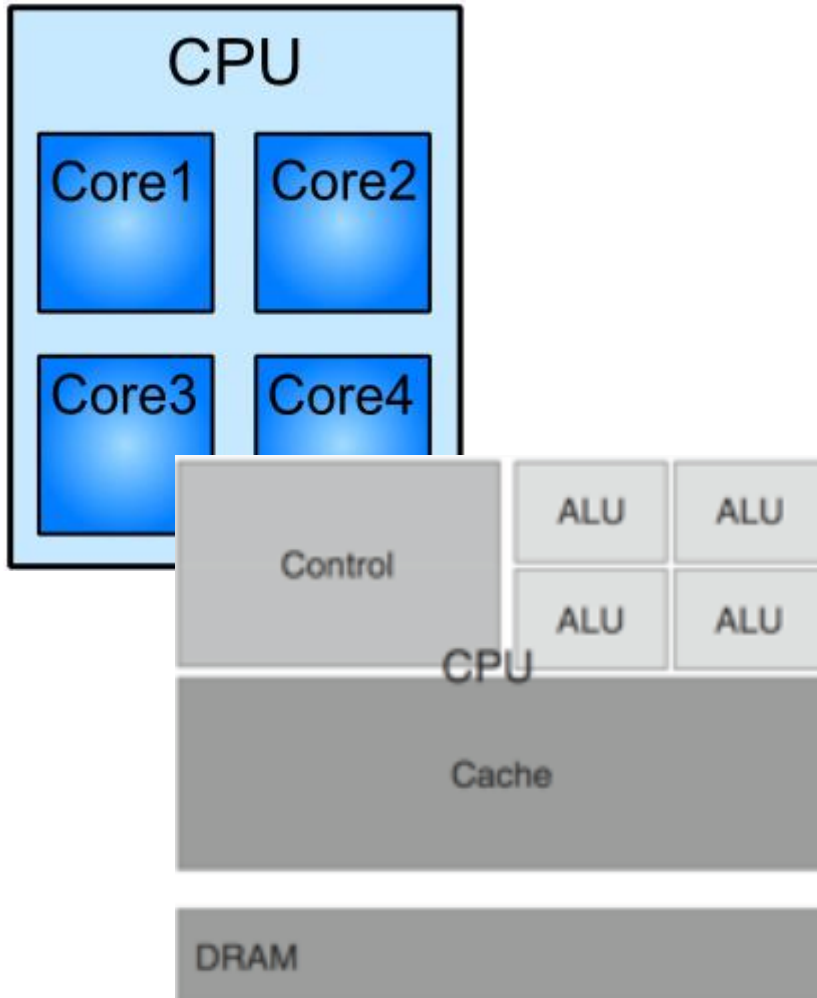
- compute problem as native graphic operations; algorithms as shaders; data in textures

## GPU Computing

- Programming CUDA; shaders programmable; load and store instructions; barriers; atomics

# CPU vs. GPU Architecture

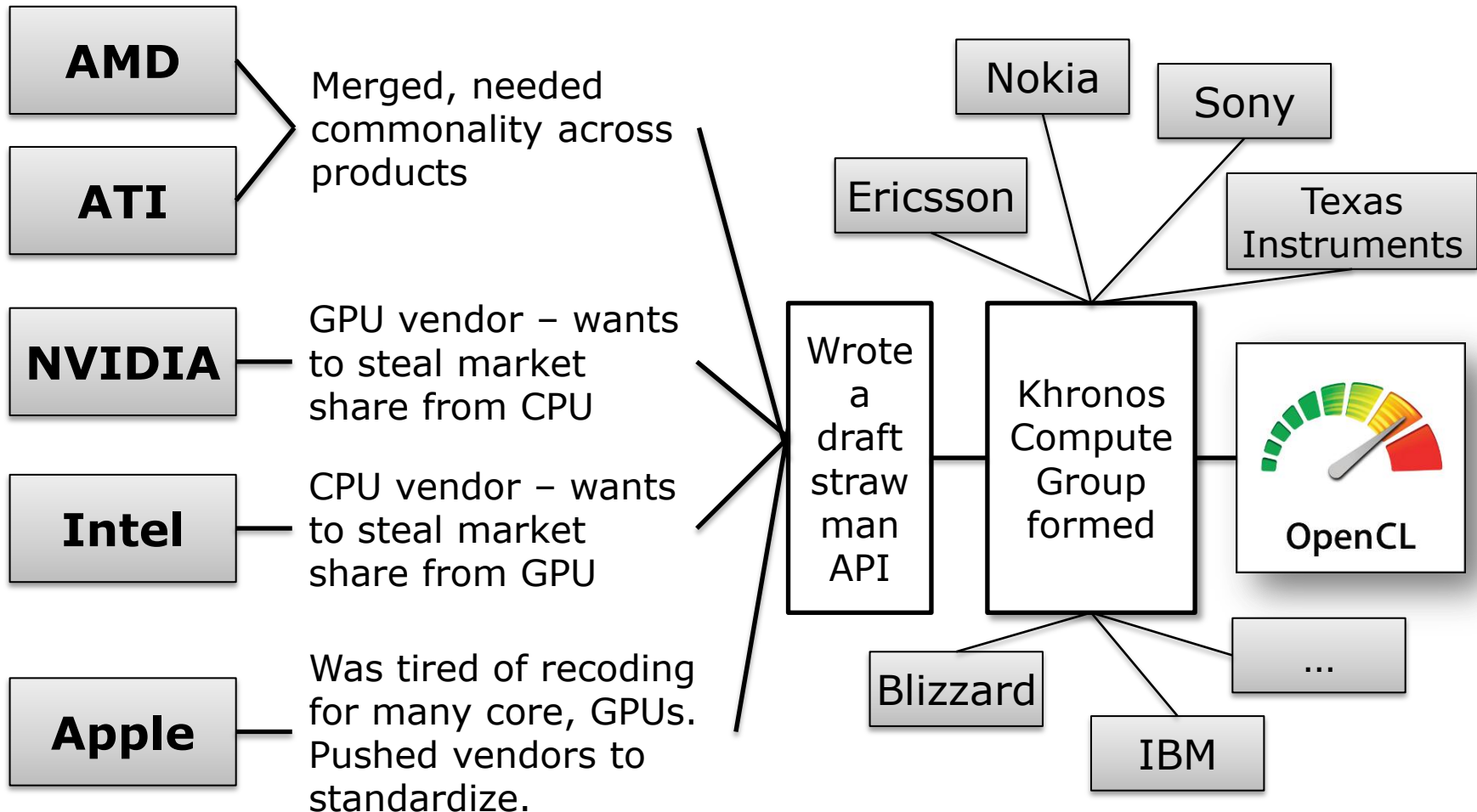
11





# Open Compute Language (OpenCL)

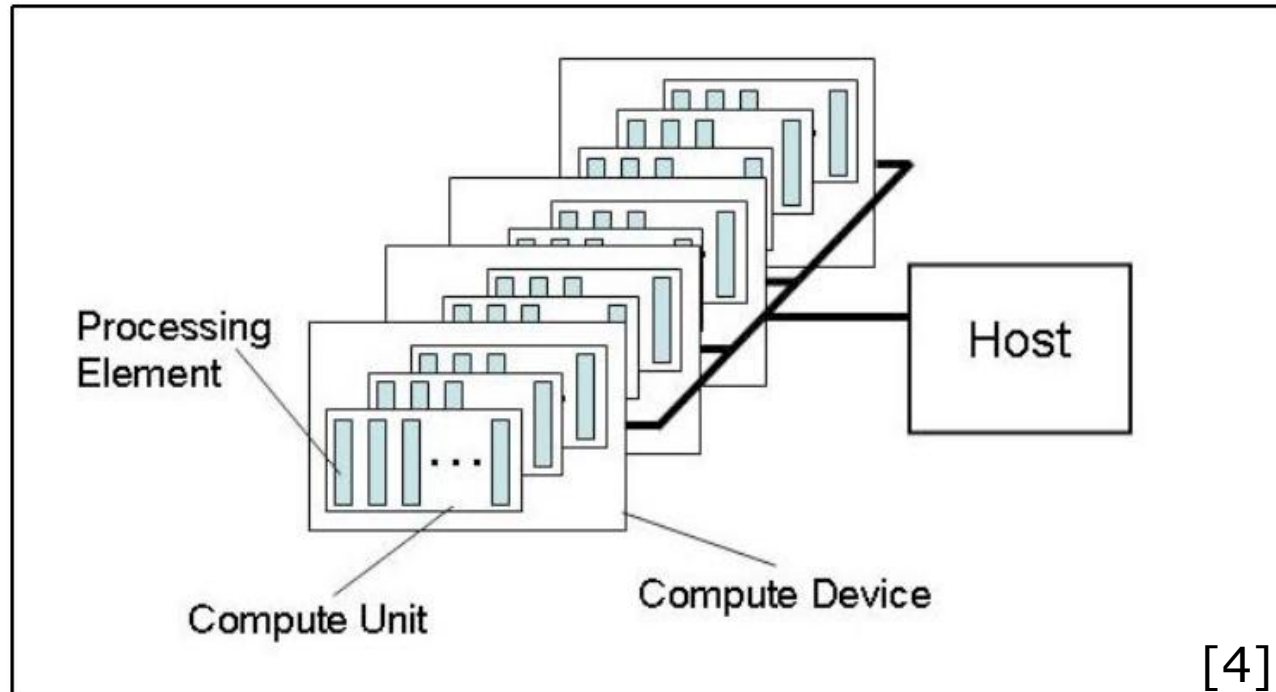
12



[5]

# OpenCL Platform Model

15

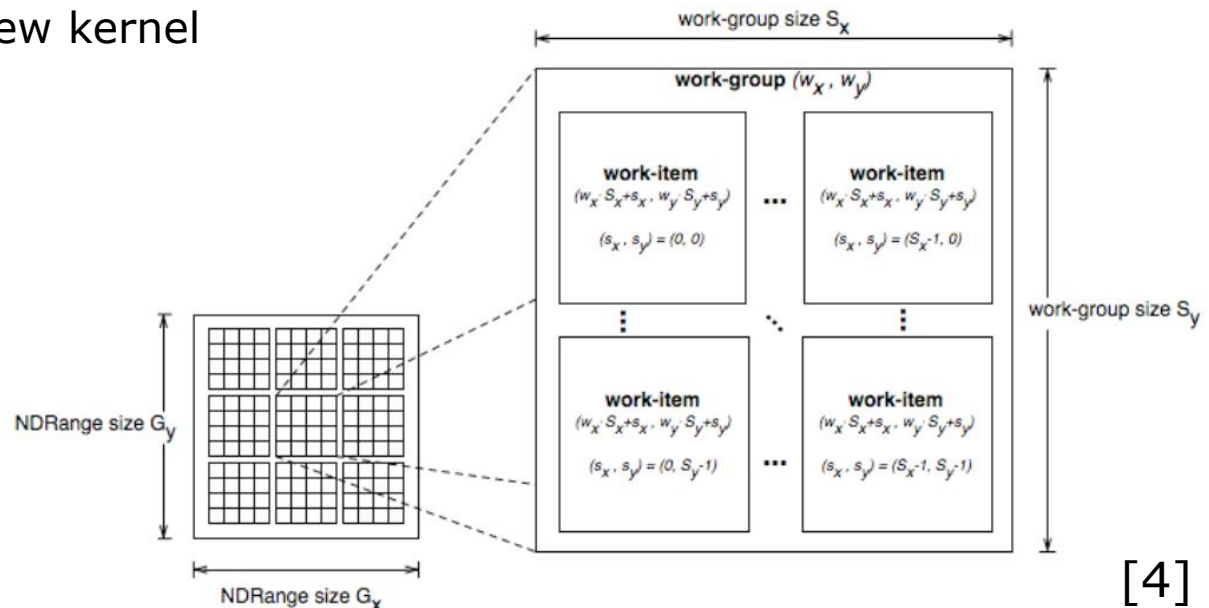


- OpenCL exposes CPUs, GPUs, and other Accelerators as “devices”
- Each “device” contains one or more “compute units”, i.e. cores, SMs,...
- Each “compute unit” contains one or more SIMD “processing elements”

# OpenCL Execution Model

17

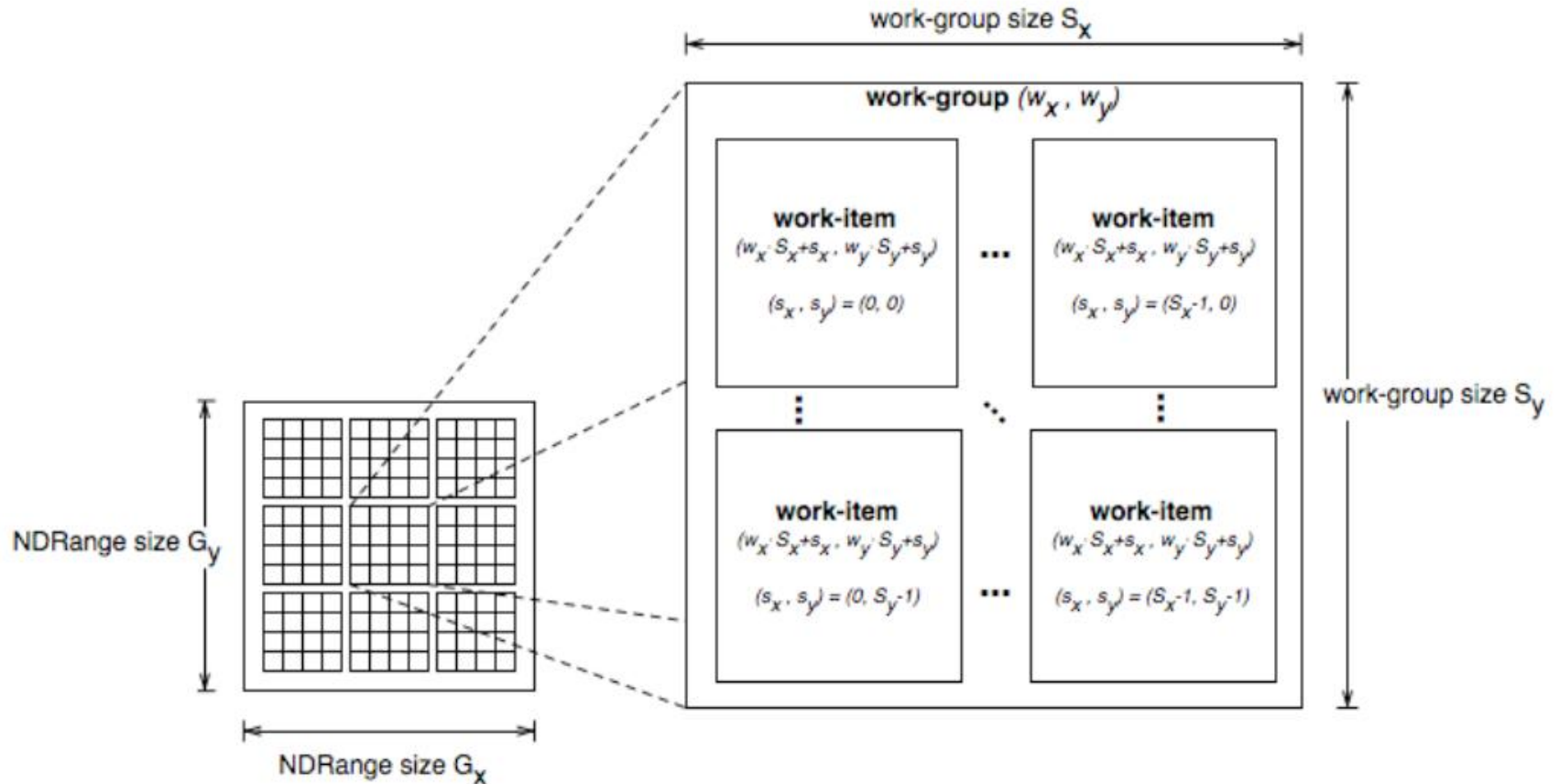
- Parallel work is submitted to devices by launching kernels
- Kernels run over global dimension index ranges (NDRange), broken up into “work groups”, and “work items”
- Work items executing within the same work group can synchronize with each other with barriers or memory fences
- Work items in different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



[4]

# OpenCL Execution Model

18



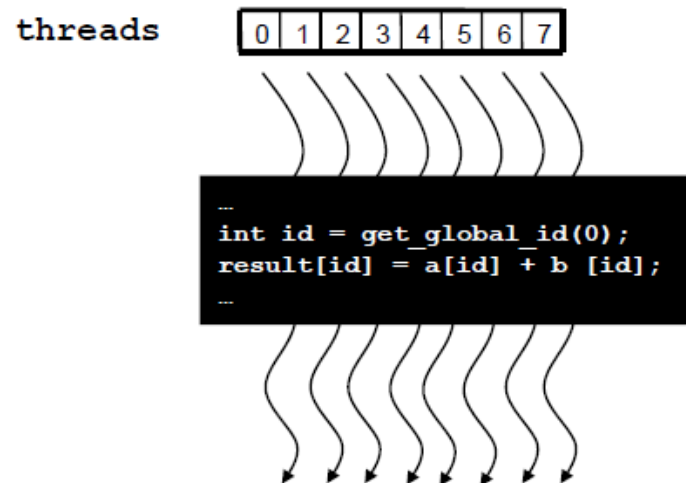
An example of an NDRange index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs. [4]

# OpenCL Execution Model

19

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



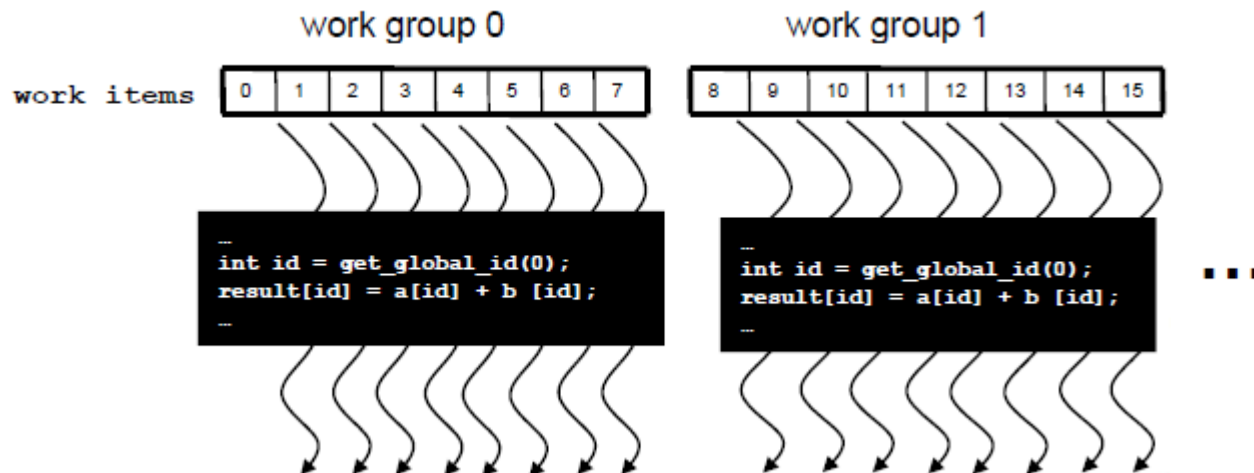
[1]

# Work Groups: Scalable Cooperation

20

Divide monolithic work item array into work groups

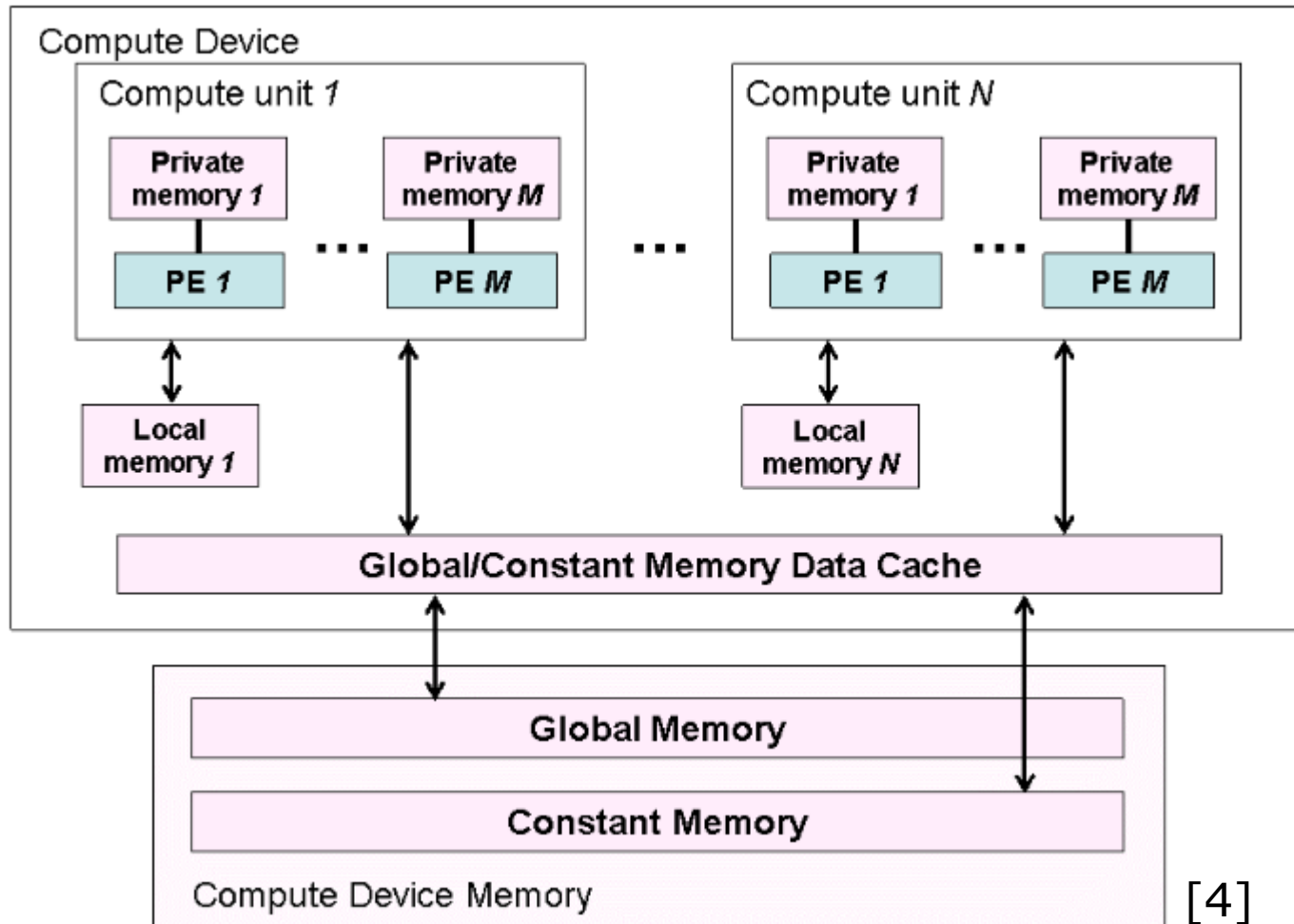
- Work items within a work group cooperate via **shared memory, atomic operations** and **barrier synchronization**
- Work items in different work groups cannot cooperate



[1]

# OpenCL Memory Architecture

21



[4]

**Private**  
Per work-item

**Local**  
Shared within  
a workgroup

**Global/  
Constant**  
Visible to  
all workgroups

**Host Memory**  
On the CPU

# OpenCL Memory Architecture

22

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

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



**OpenCL "Hello Device"**

**OpenCL "Sudoku Validator"**

# Development Support

32

**Software development kits:** NVIDIA and AMD; Windows and Linux

**Special libraries:** AMD Core Math Library, BLAS and FFT libraries by NVIDIA, OpenNL for numerics and CULA for linear algebra; NVIDIA Performance Primitives library: a sink for common GPU accelerated algorithms

**Profiling and debugging tools:**

- NVIDIAs Parallel Nsight for Microsoft Visual Studio
- AMDs ATI Stream Profiler
- AMDs Stream KernelAnalyzer:  
displays GPU assembler code, detects execution bottlenecks
- gDEDebugger (platform-independent)

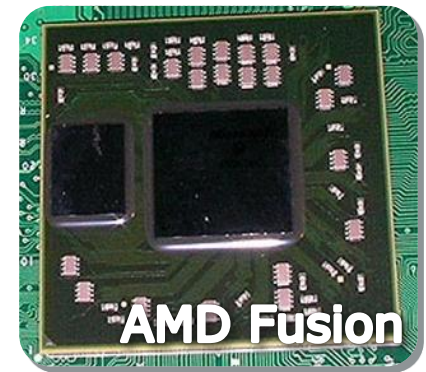
Big knowledge bases with tutorials, examples, articles, show cases, and developer forums

# GPU Computing Platforms

33

**AMD**

R700, R800, R900

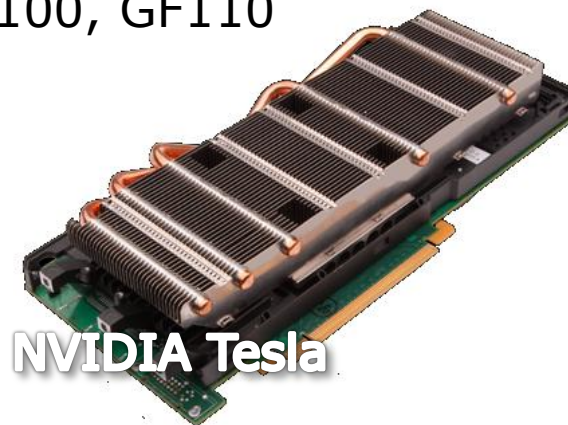


**NVIDIA**

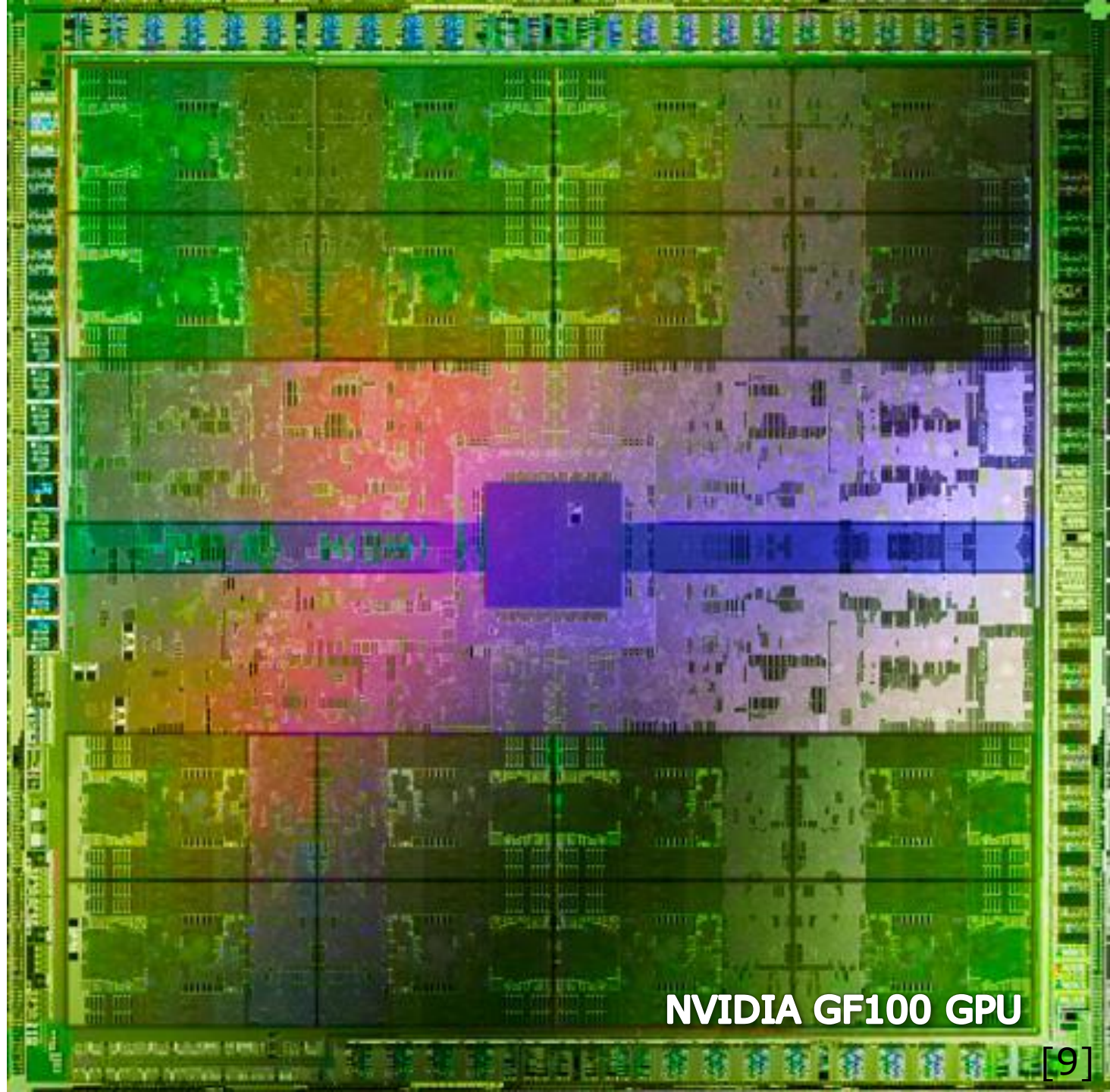


G80, G92, GT200, GF100, GF110

Geforce, Quadro,  
Tesla, ION



# GPU Hardware in Detail



**NVIDIA GF100 GPU**

# GF100

Host Interface

GigaThread Engine

Memory Controller

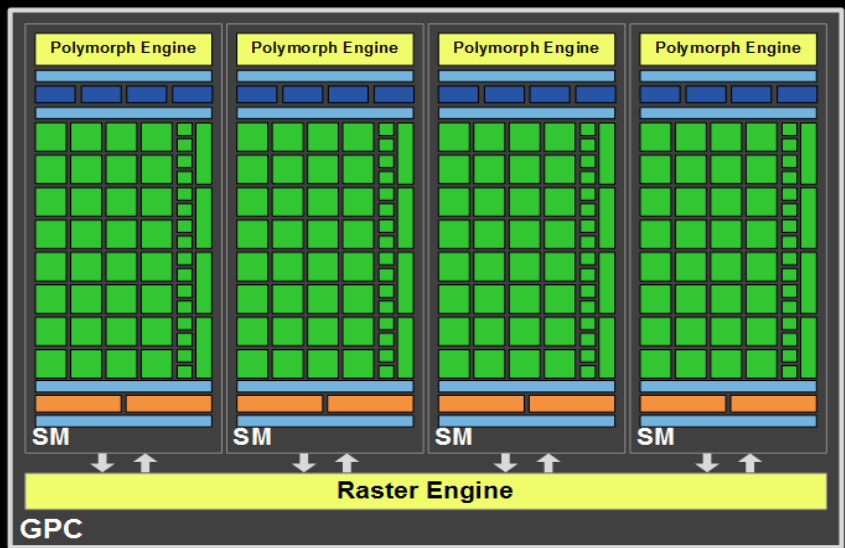
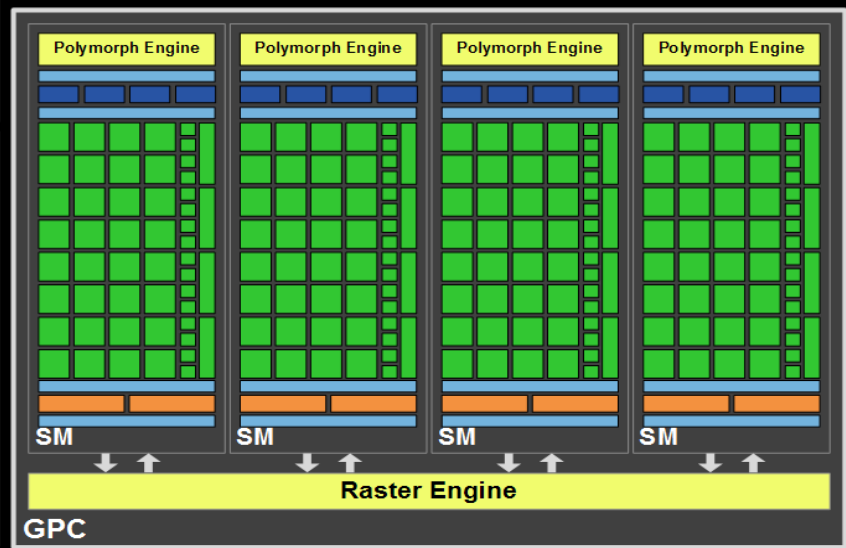
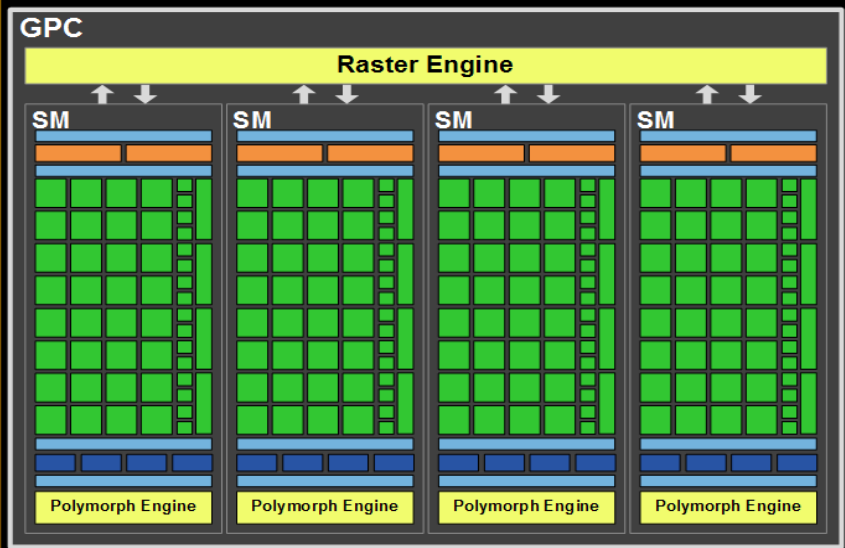
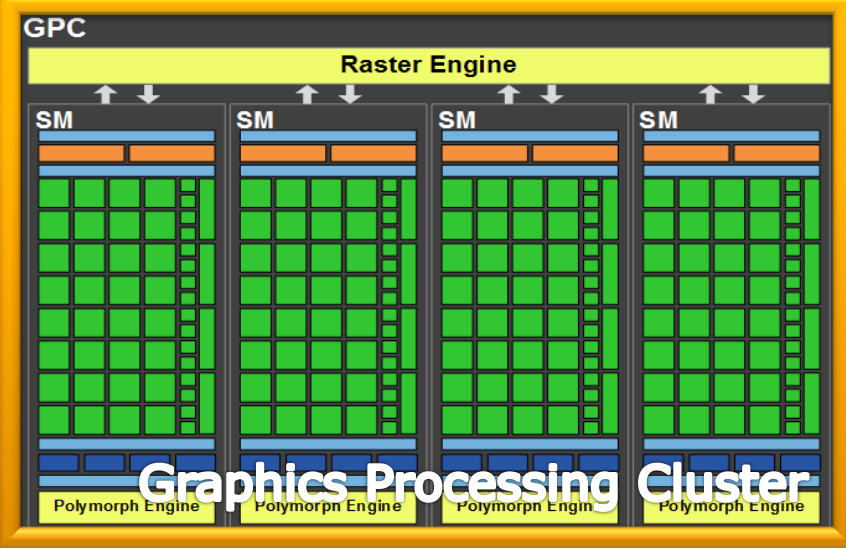
Memory Controller

Memory Controller

Memory Controller

Memory Controller

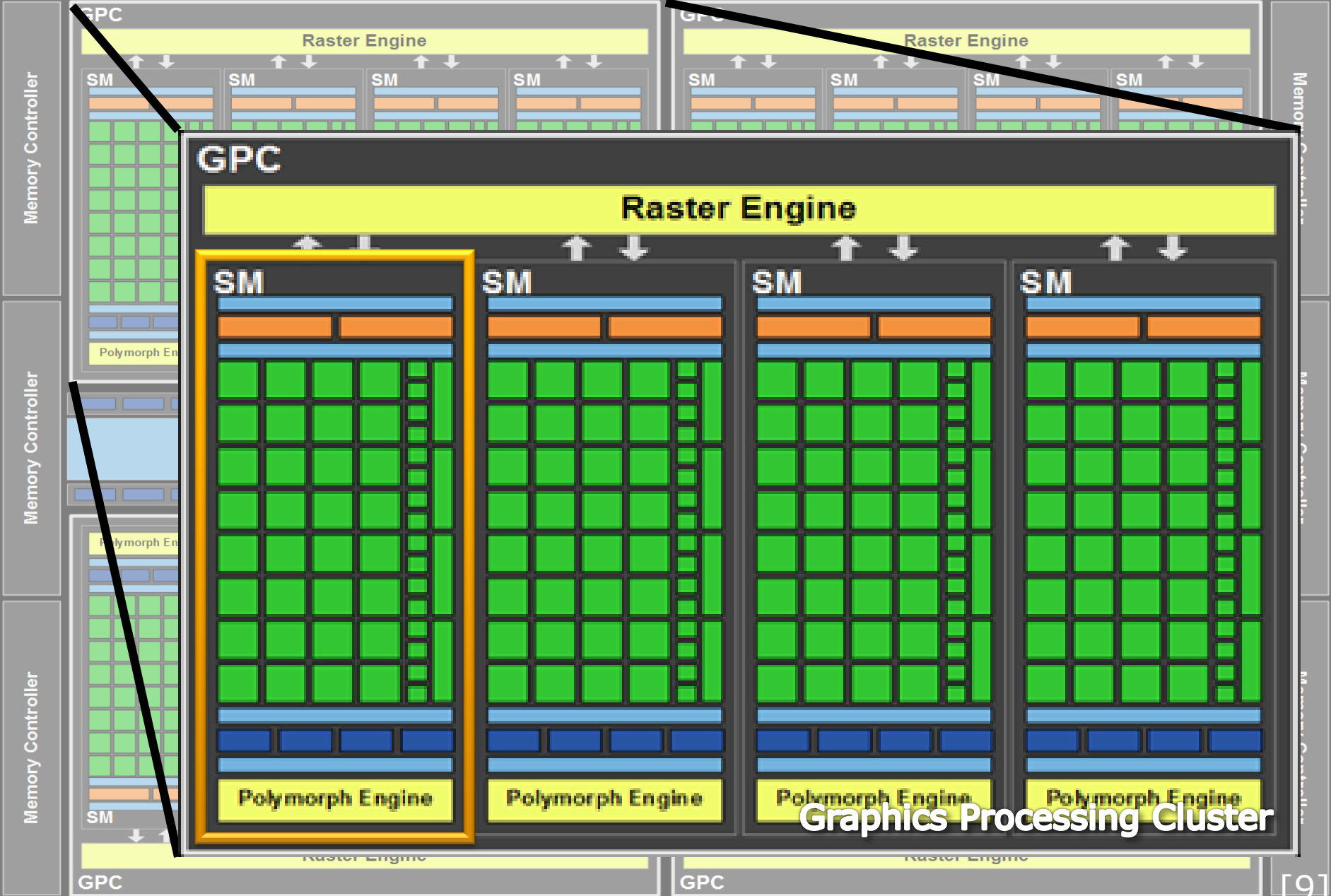
Memory Controller



# GF100

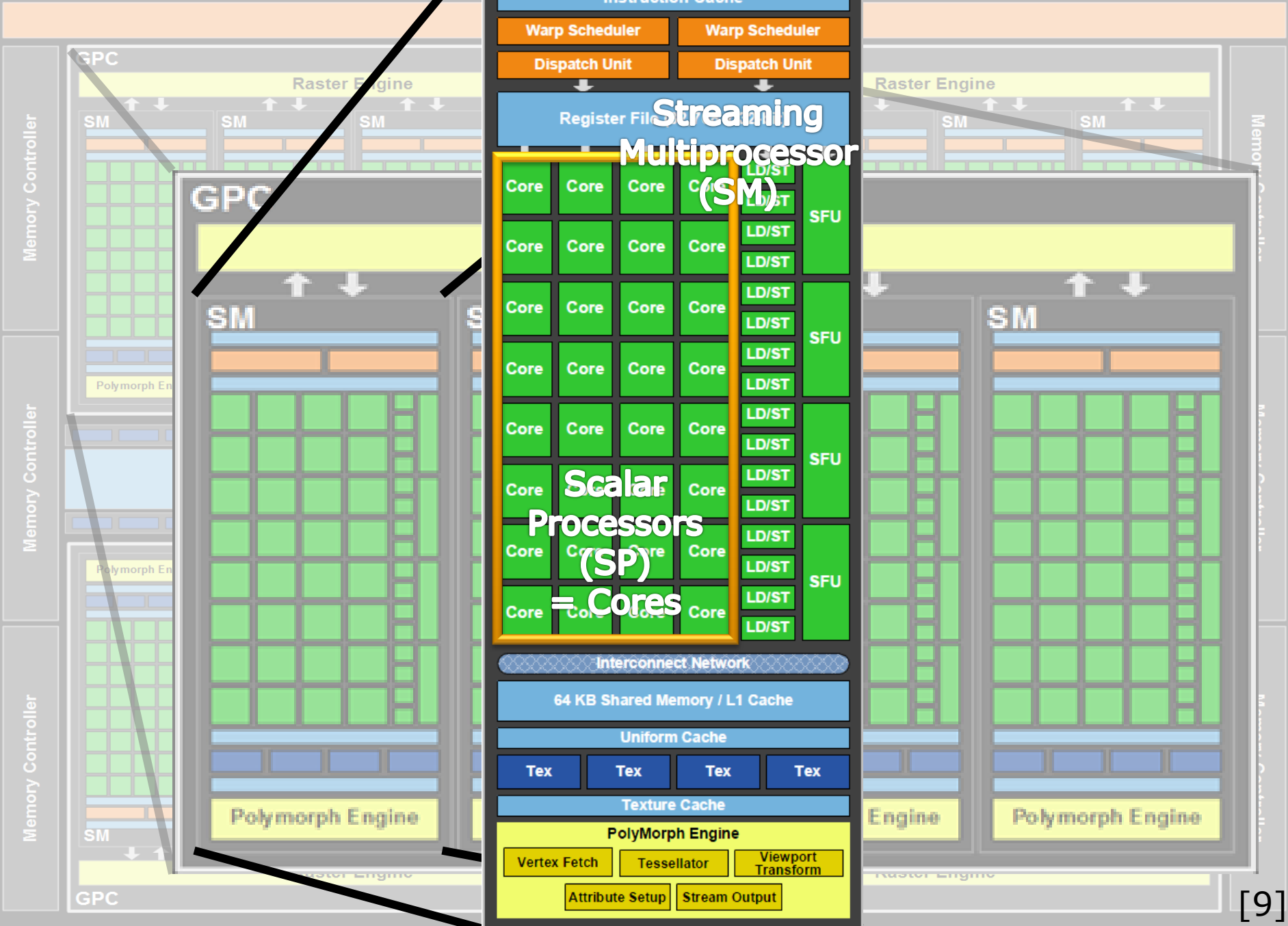
Host Interface

GigaThread Engine

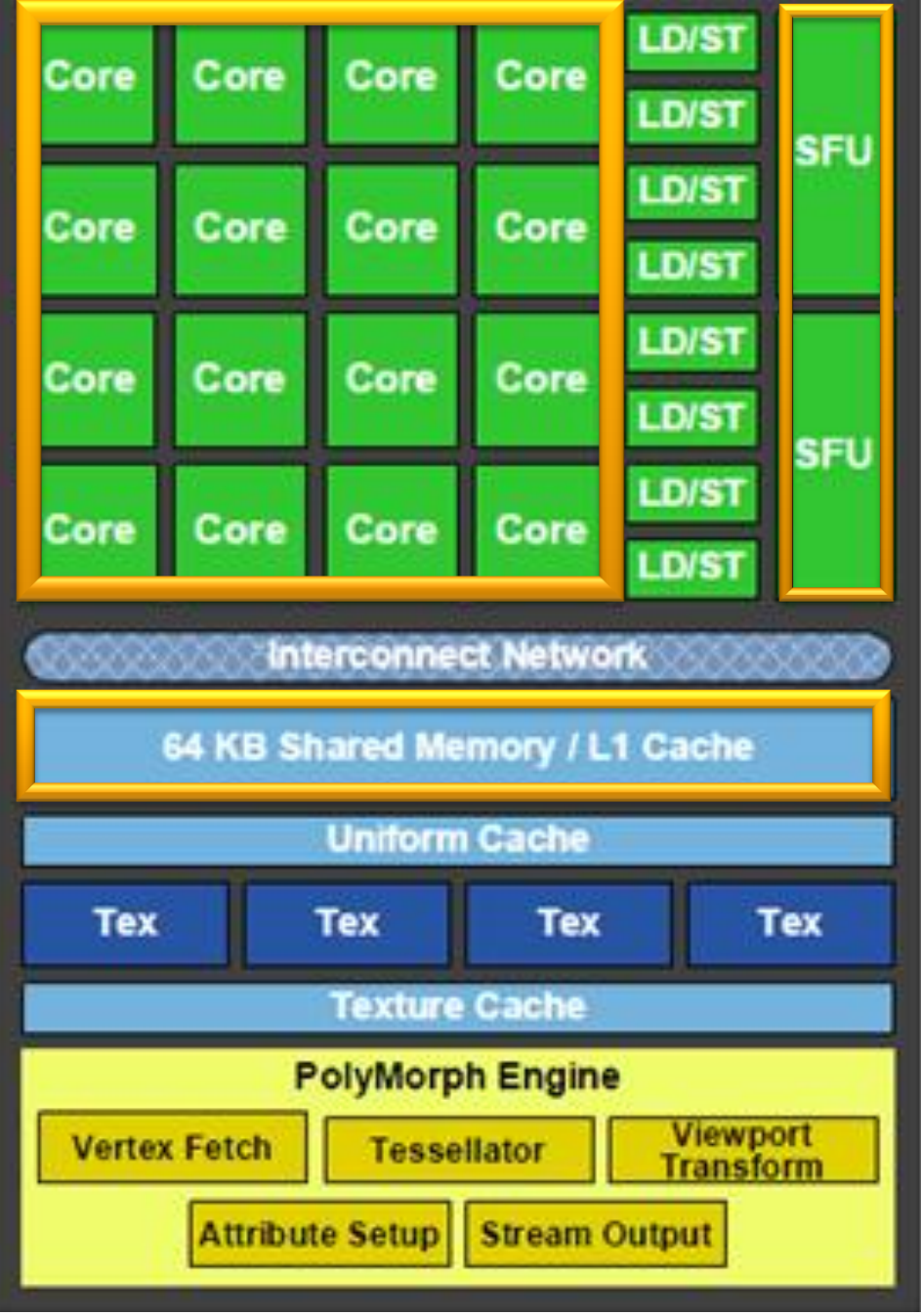
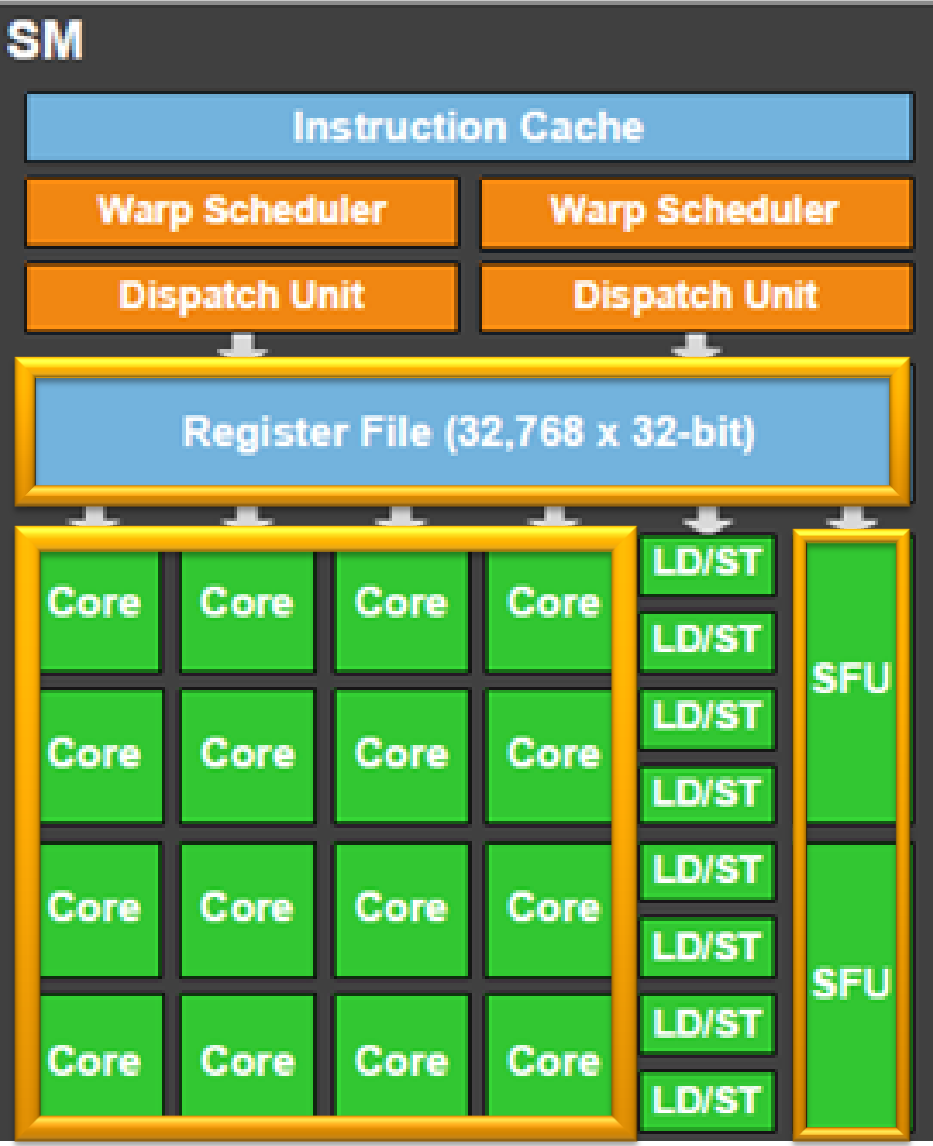


Graphics Processing Cluster

# GF100

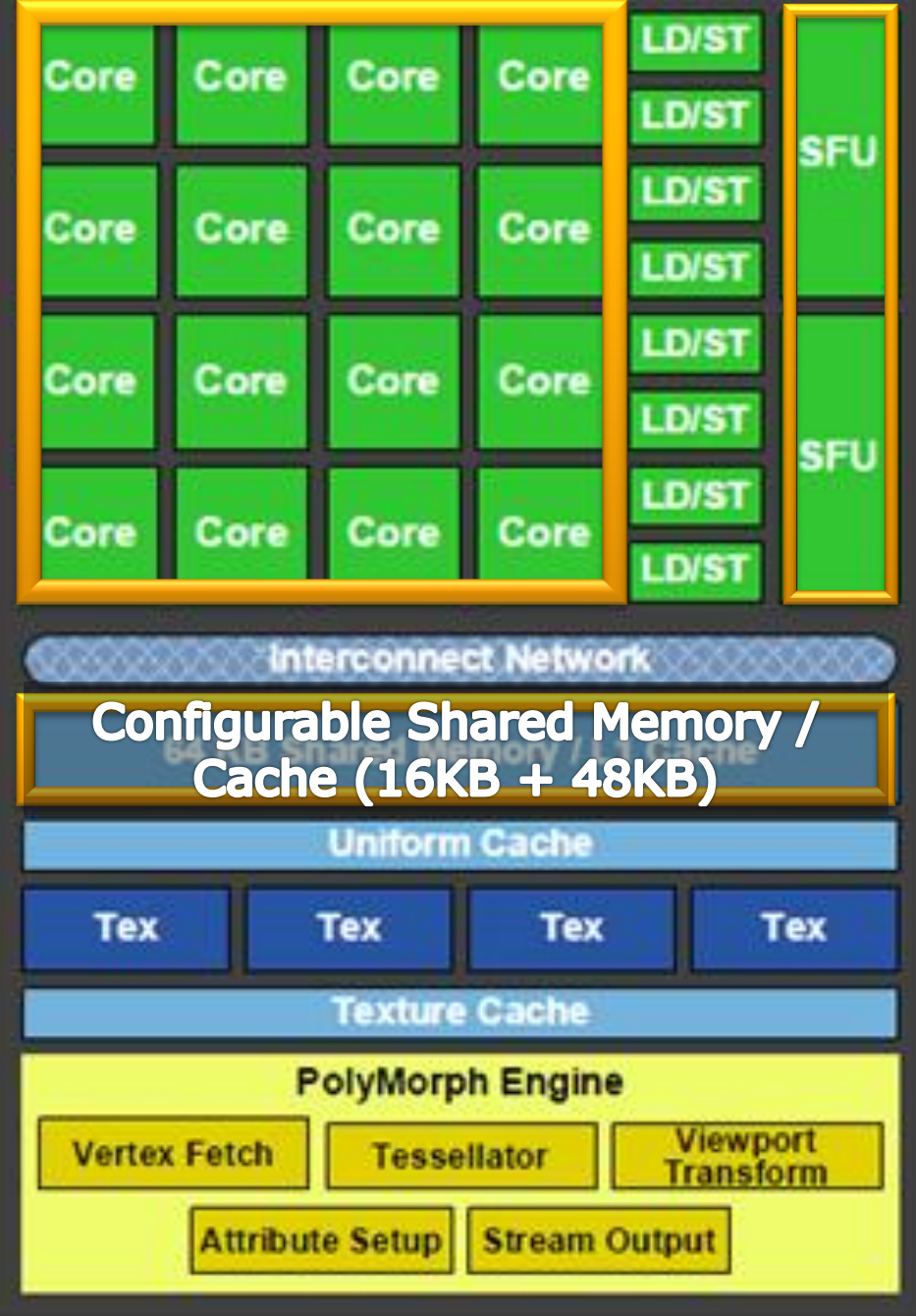
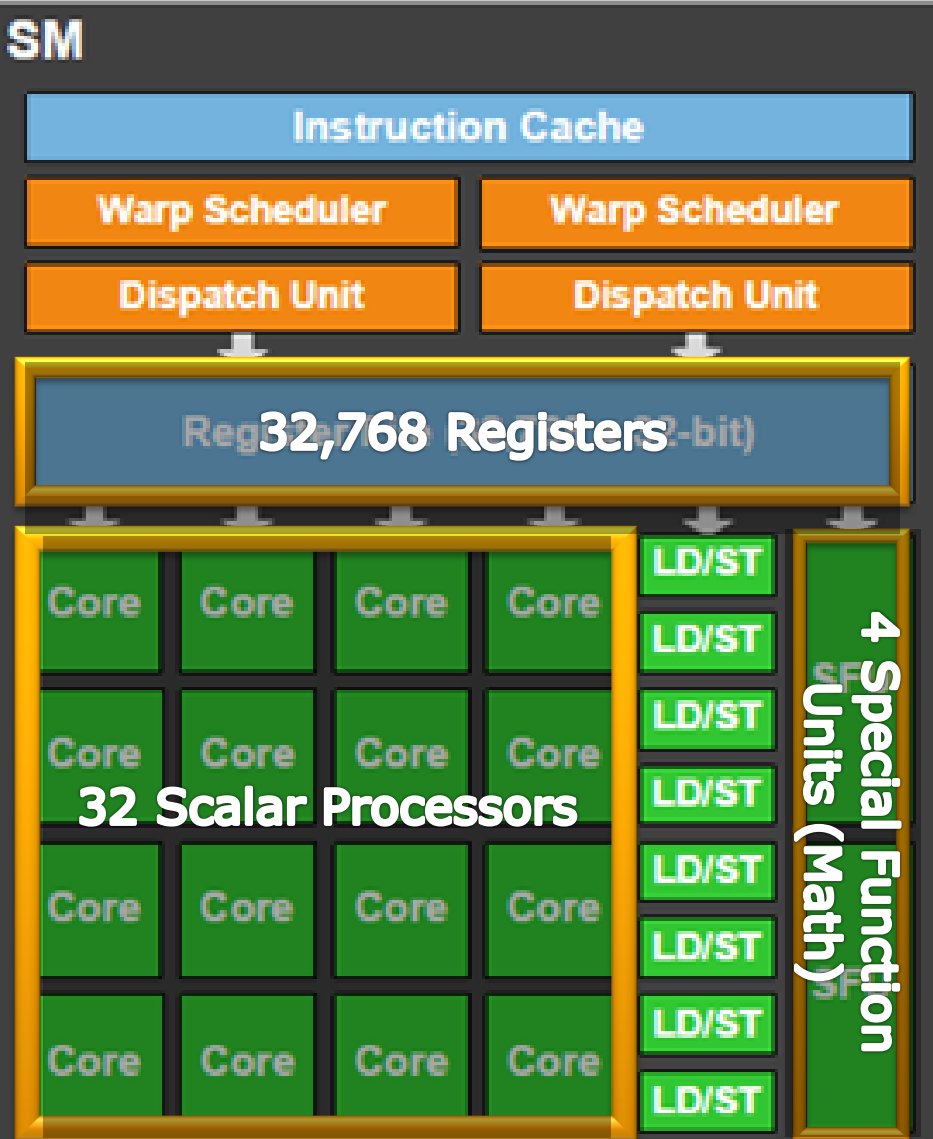


# GF100





# GF100



# GT200 – previous architecture

40

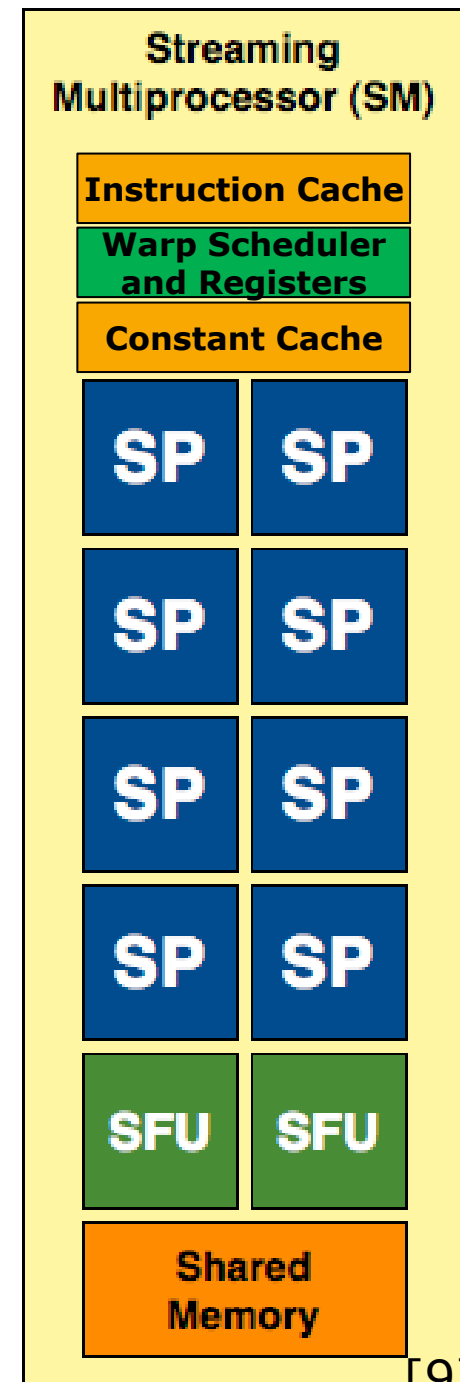
Simpler architecture, but same principles

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



# Warp Execution Example

41

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:  $16\text{KB}/20\text{B} = 819$  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

42

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 separate round

# Compute Capability by version

43

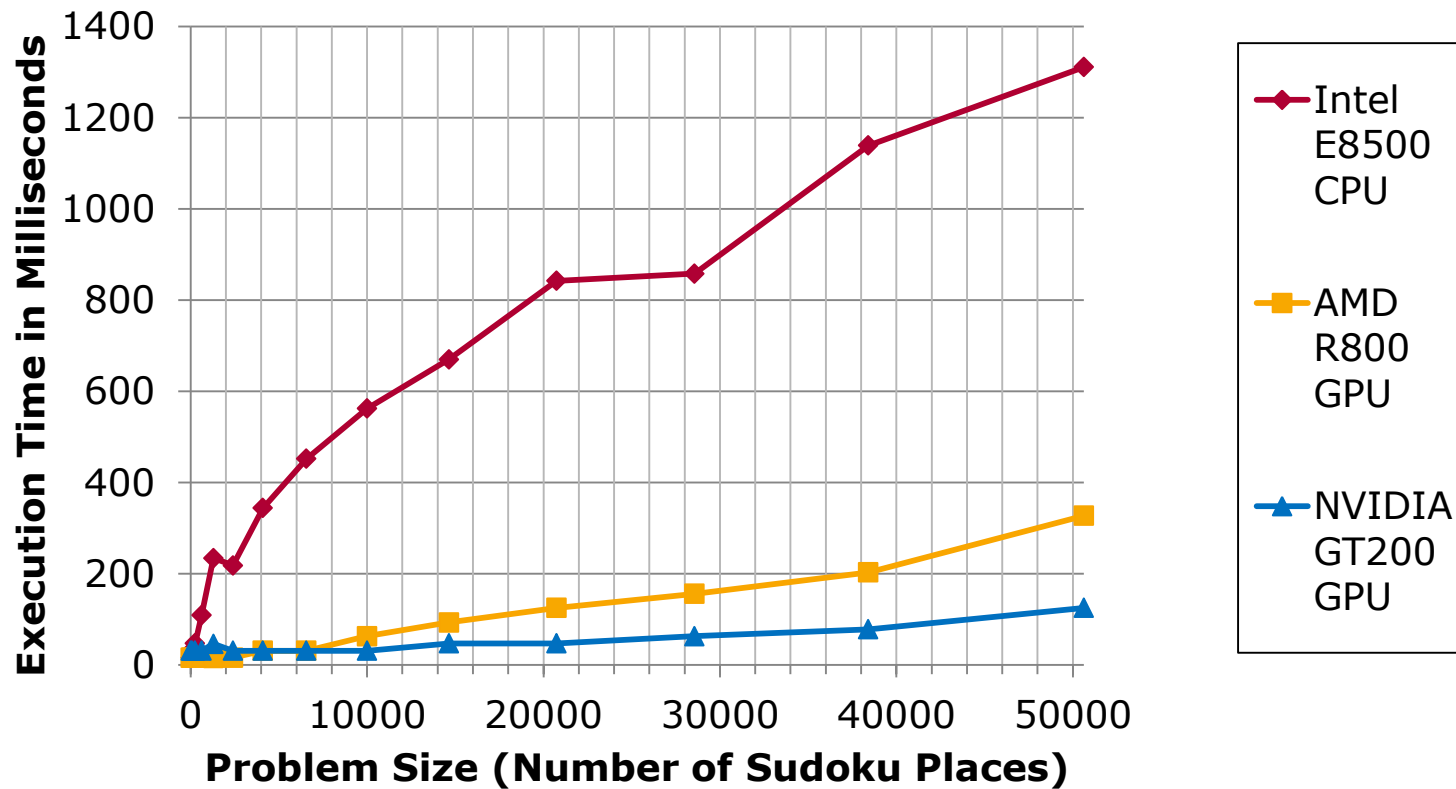
	1.0	1.1	1.2	1.3	2.0
double precision floating point operations	No			Yes	
caches	No			Yes	
max # concurrent kernels	1			8	
max # threads per block	512			1024	
max # Warps per MP	24	32		48	
max # Threads per MP	768	1024		1536	
register count (32 bit)	8192	16384		32768	
max shared mem per MP	16KB			48KB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

44

big performance gains for small problem sizes

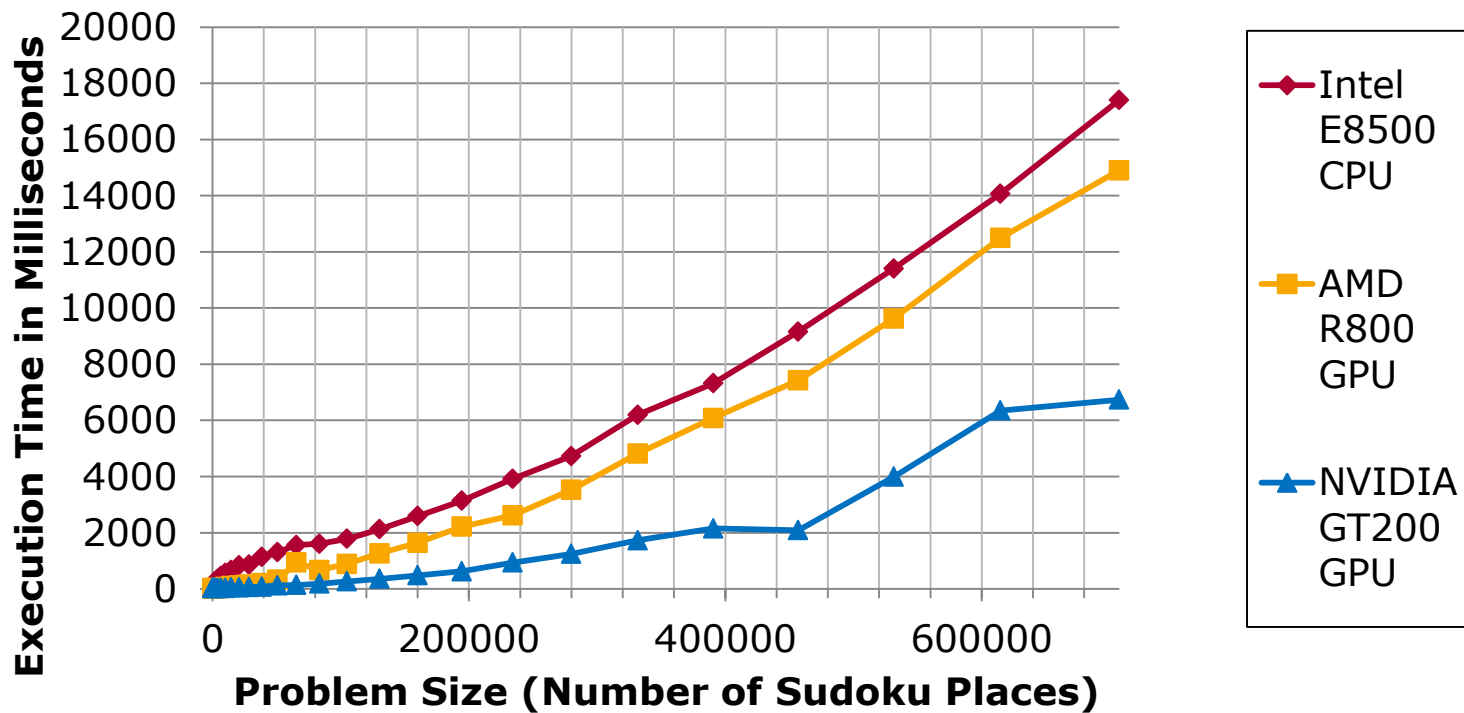


\* less is better

# The Power of GPU Computing

45

small/moderate performance gains for large problem sizes  
 → further optimizations needed



\* less is better

# Best Practices for Performance Tuning

46

## 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

47

## 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

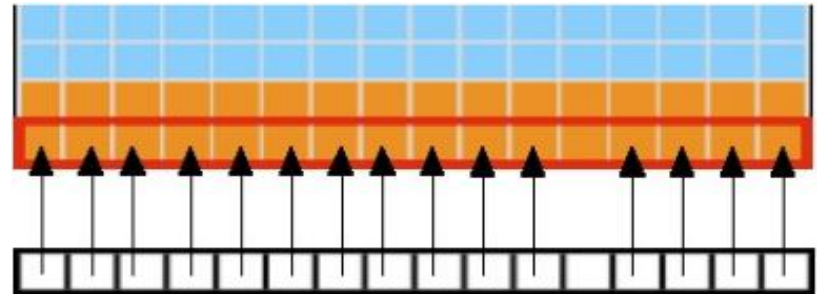
- Instructions are associated with a per-thread condition code (predicate)
  - All instructions are scheduled for execution
  - Predicate true: executed normally
  - Predicate false: do not write results, do not evaluate addresses, do not read operands
- Compiler may use branch predication for `if` or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

# Coalesced Memory Accesses

48

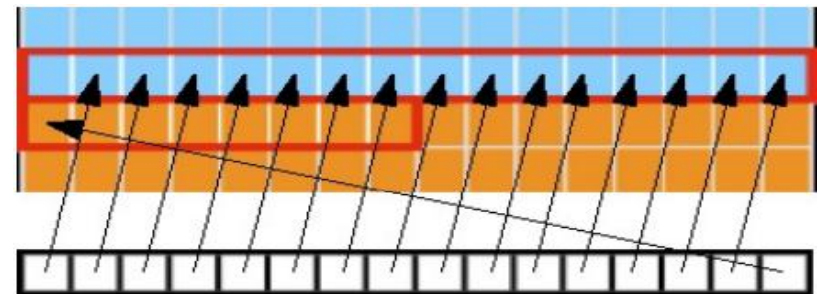
## Simple Access Pattern

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



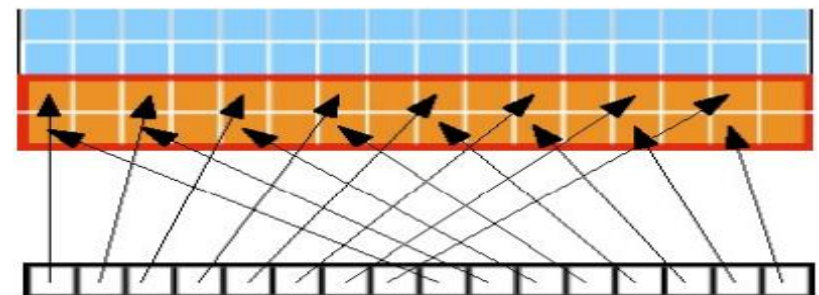
## Sequential but Misaligned Access

- 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

# Use Caching: Local, Texture, Constant

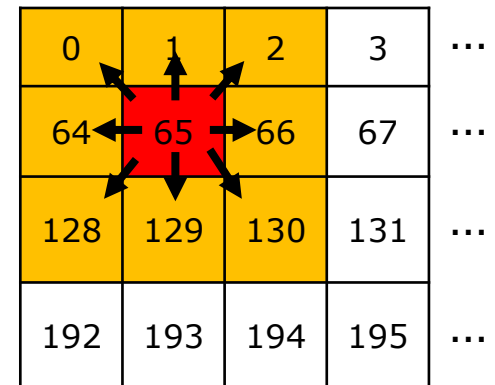
49

## 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 from global memory
- Used with the image data type



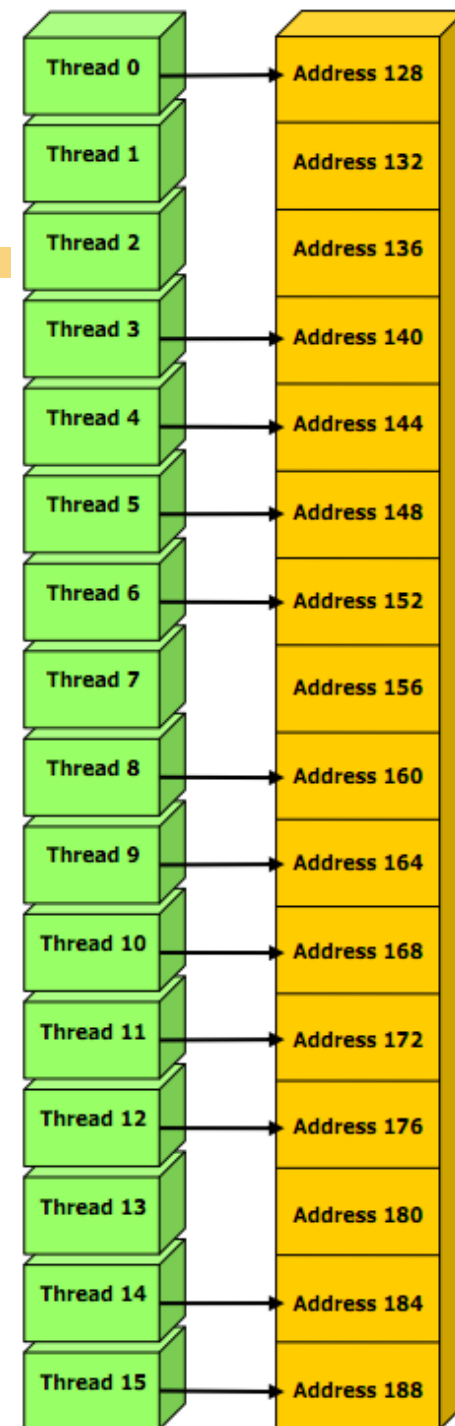
## Constant Memory

- 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

50

- 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

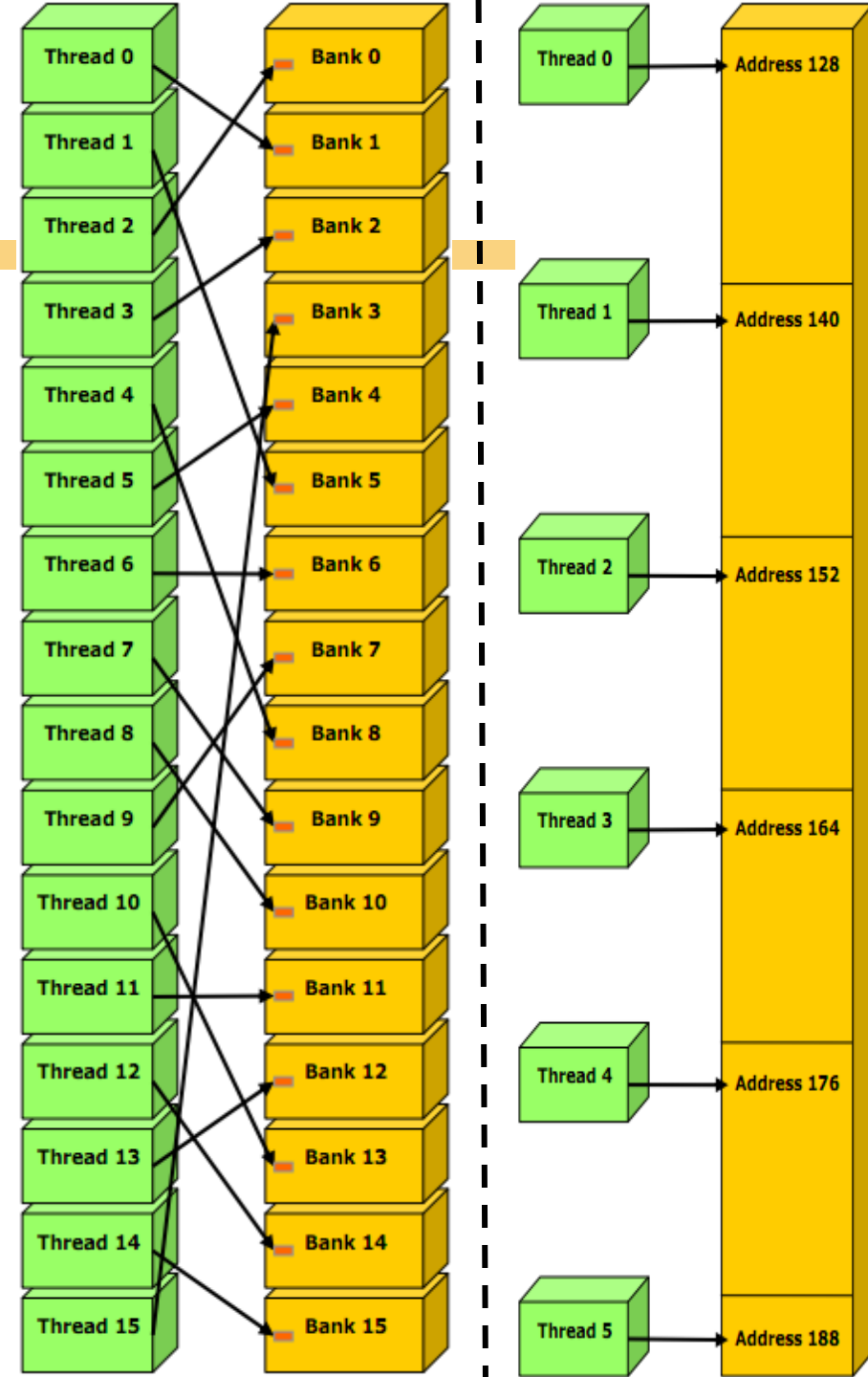
51

## Permuted Memory Access (left)

- Still one transaction on cards with compute capability  $\geq 1.2$ ; otherwise 16 transactions are required

## Strided Memory Access (right)

- Still one transaction on cards with compute capability  $\geq 1.2$ ; otherwise 16 transactions are required

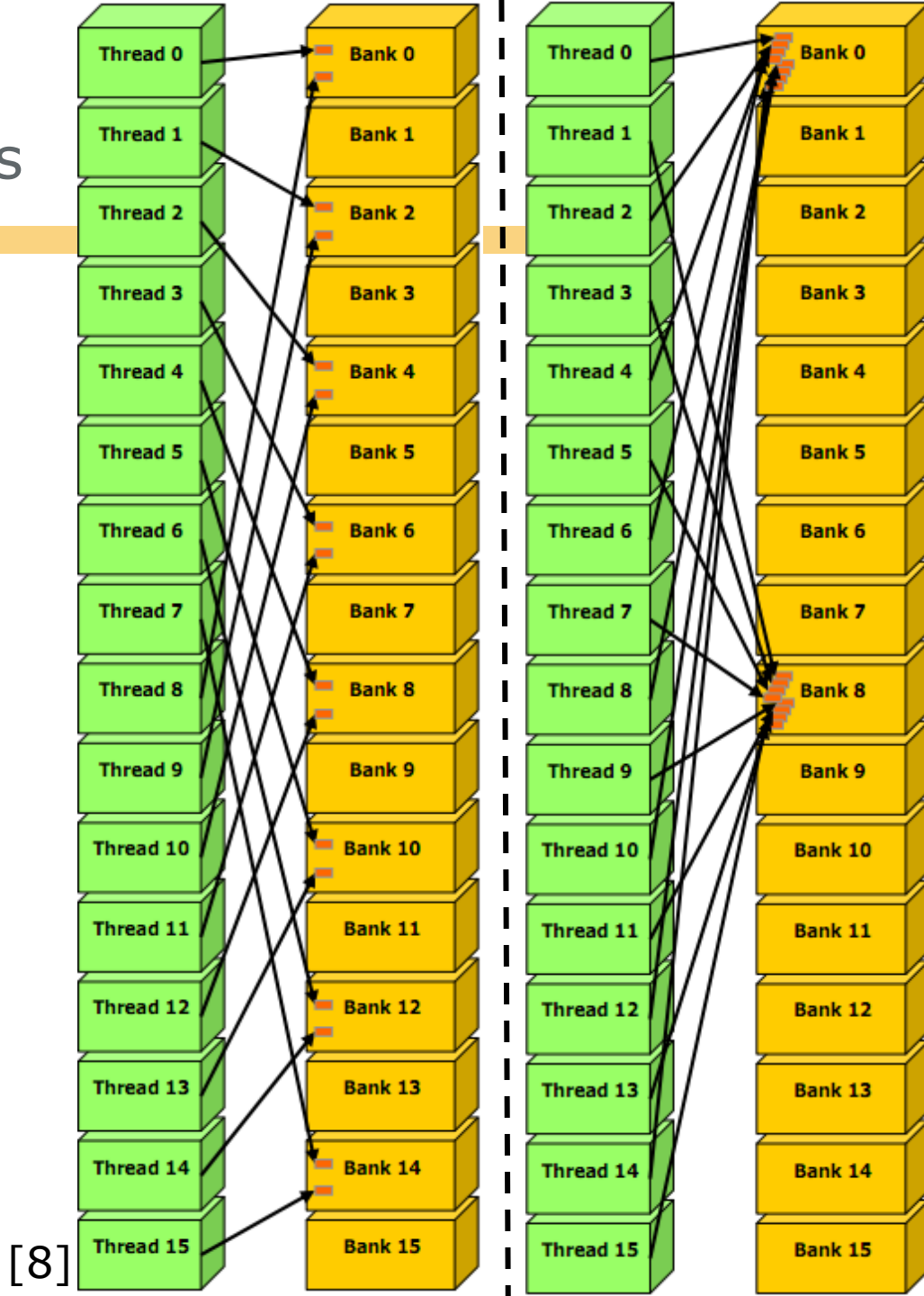


# Memory Bank Conflicts

52

## Bank conflicts

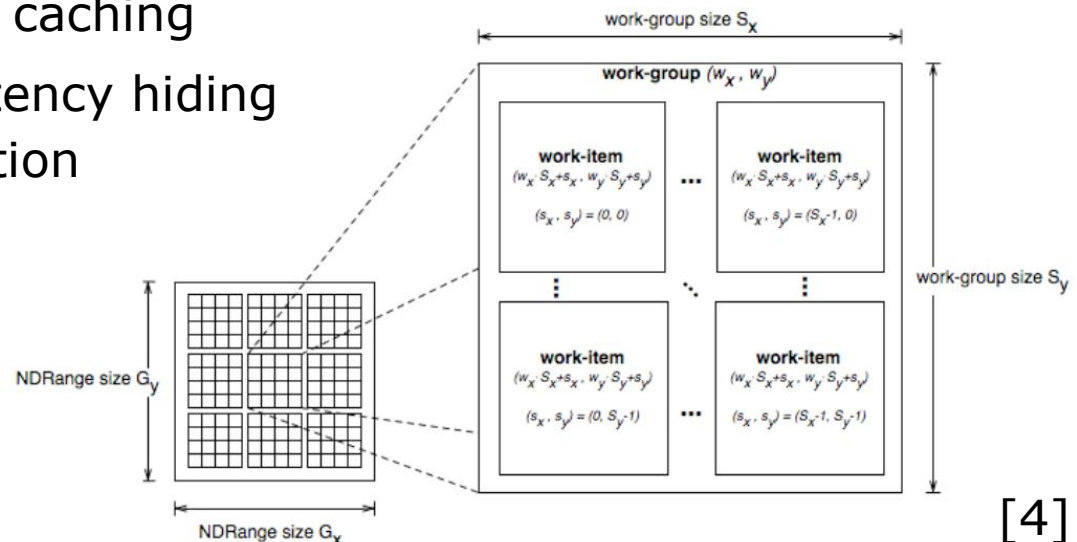
- Left figure: 2 bank conflicts → resulting bandwidth is  $\frac{1}{2}$  of the original bandwidth
- Right figure: 8 bank conflicts → resulting bandwidth is  $\frac{1}{8}$  of the original bandwidth



# Sizing: What is the right execution layout?

53

- Local work item count should be a multiple of native execution size (NVIDIA 32, AMD 64), but not too 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!



[4]

# Instructions and Precision

54

- 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	<b>8</b> operations per clock cycle
single-precision reciprocal, reciprocal square root, and native_logf(x)	<b>2</b> operations per clock cycle
native_sin, native_cos, native_exp	<b>1</b> operation per clock cycle



## Further Readings

55

<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?