Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Programmierung paralleler und verteilter Systeme

GPU Computing with OpenCL

Frank Feinbube

Operating Systems and Middleware
Prof. Dr. Andreas Polze

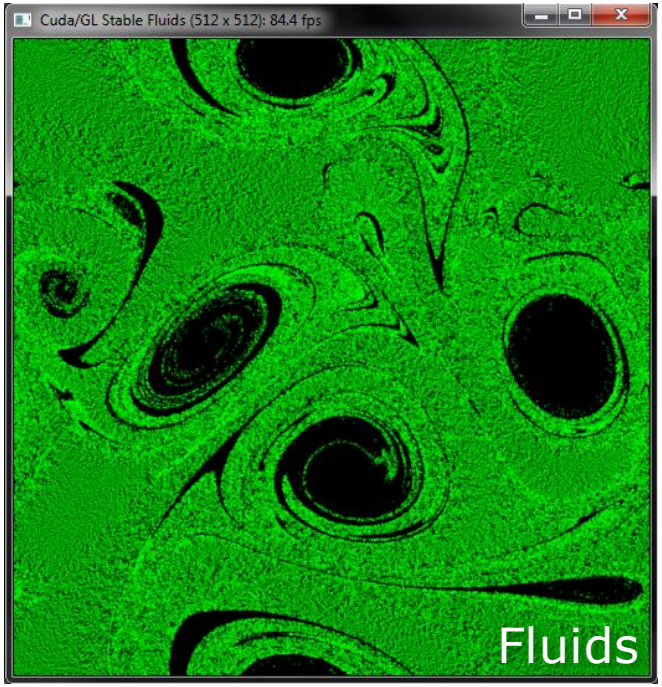
Agenda / Quicklinks

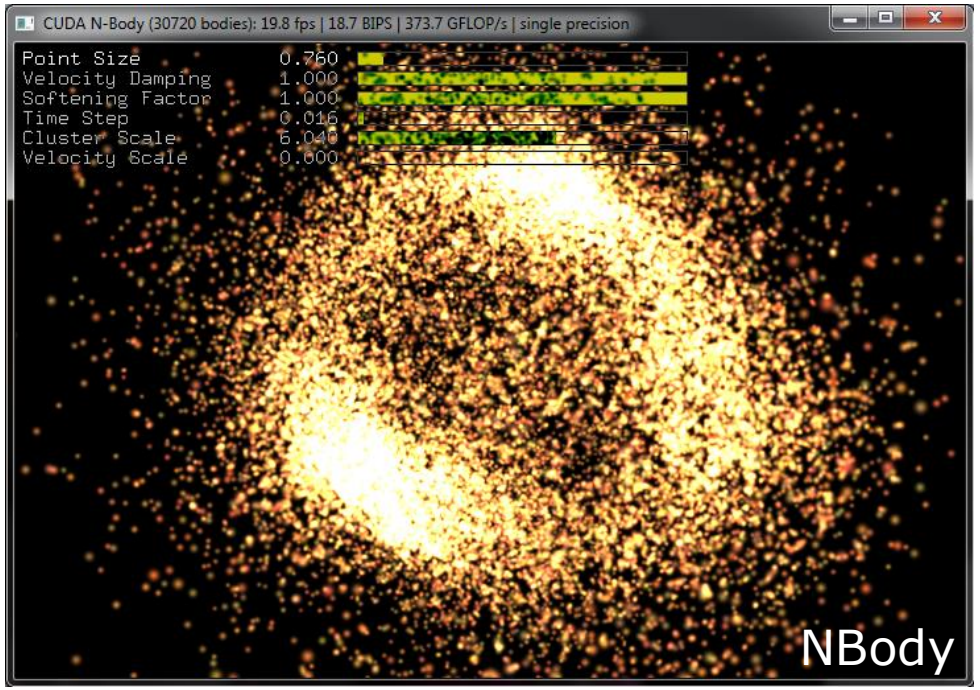
- Motivation
- History of GPU Computing
- Programming Model
- Examples
- Development Support
- Hardware Characteristics
- Performance Tuning
- Further Readings

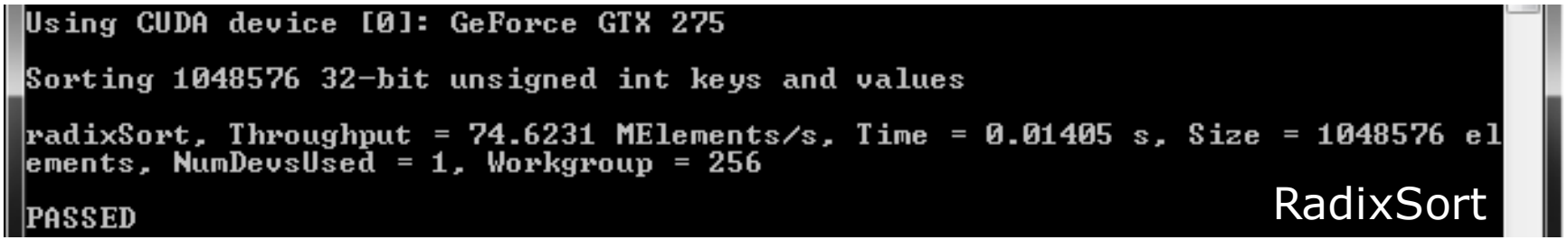
Intel Xeon Phi

NVIDIA Tesla

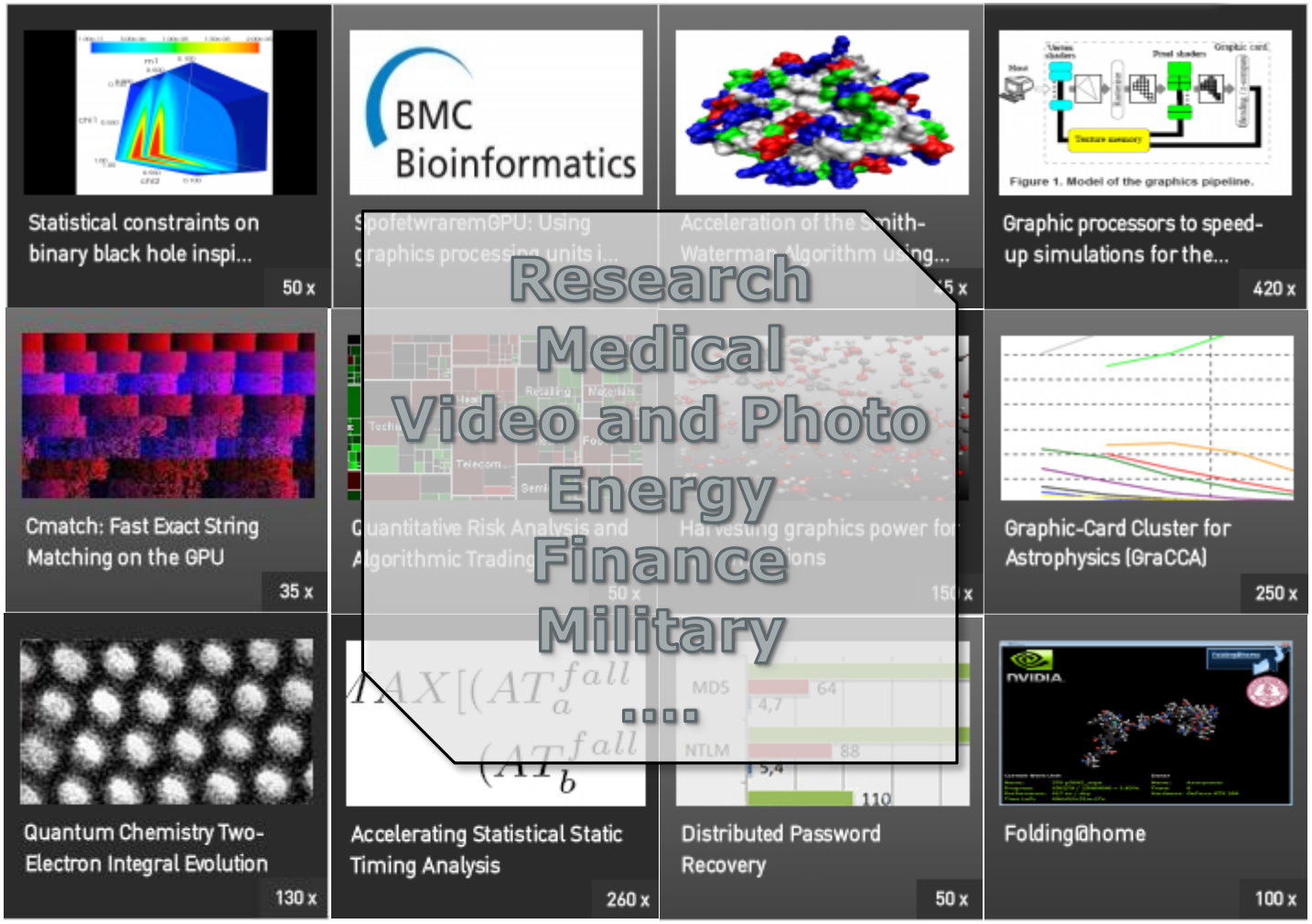
The Power of GPU Compute Devices







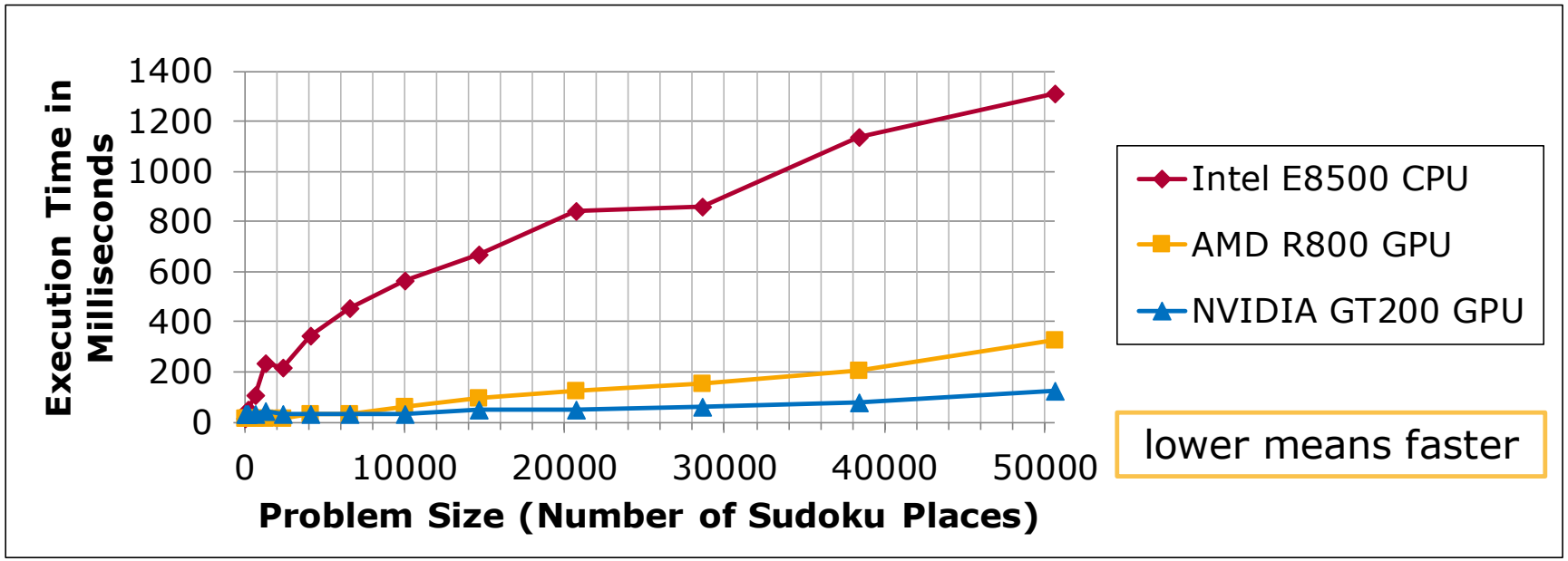
Wide Varity of Application Domains



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

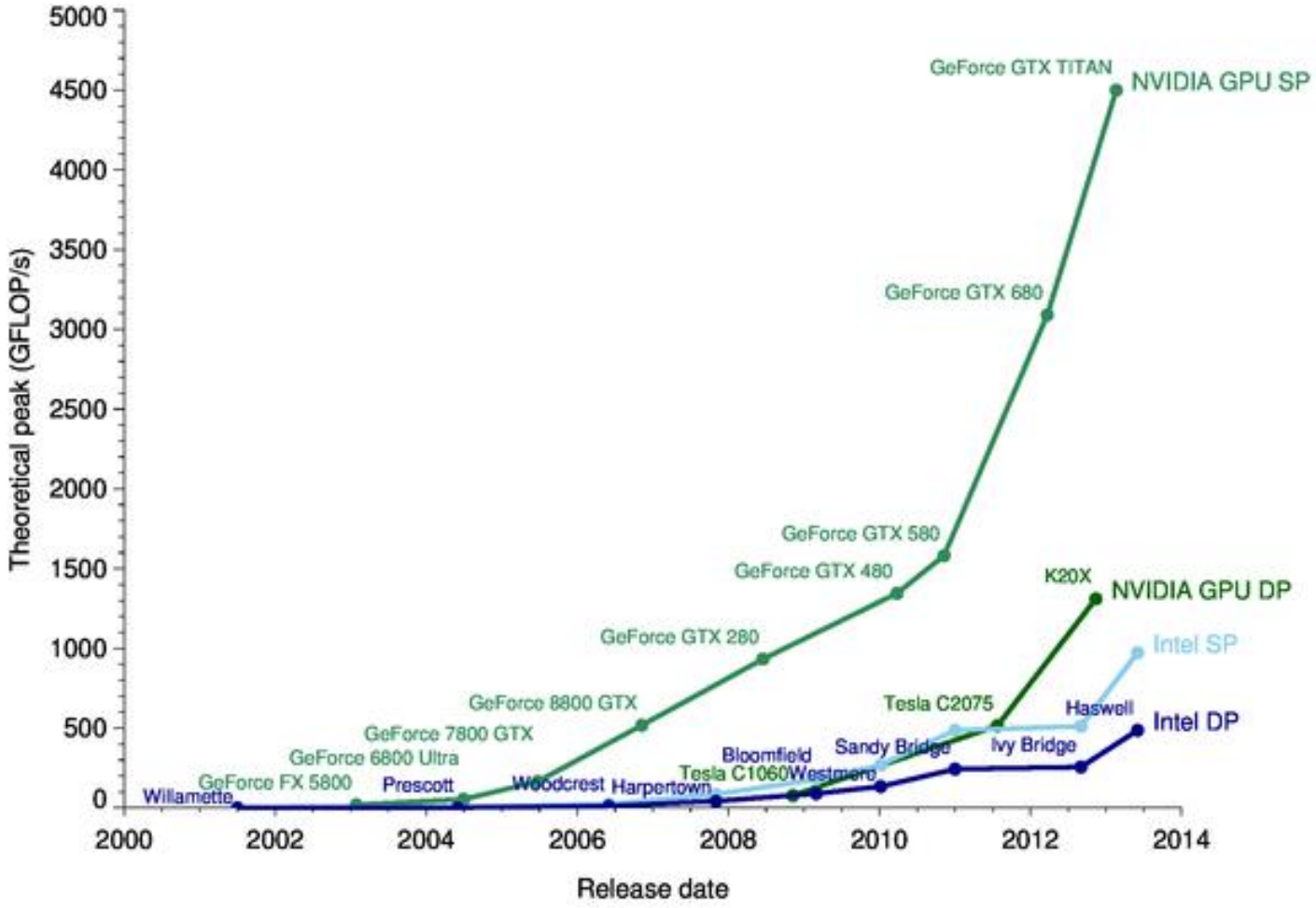
Performance



Energy / Price

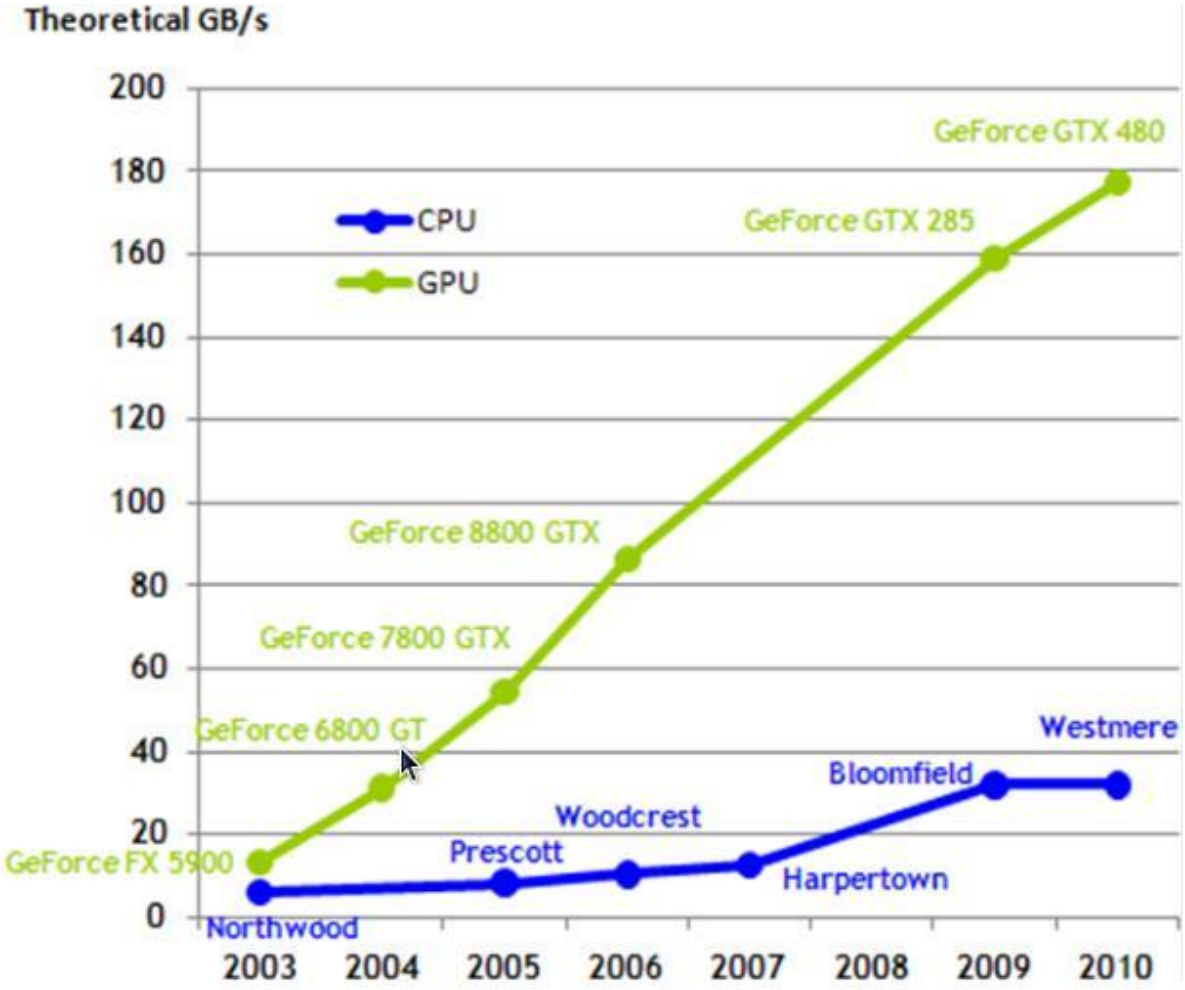
- Cheap to buy and to maintain
- GFLOPS per watt: Fermi 1,5 / Keppler 5 / Maxwell 15

Why GPU Compute Devices?
Middle Term View: More Performance



[8]

Why GPU Compute Devices?
Middle Term View: More Performance



[8]

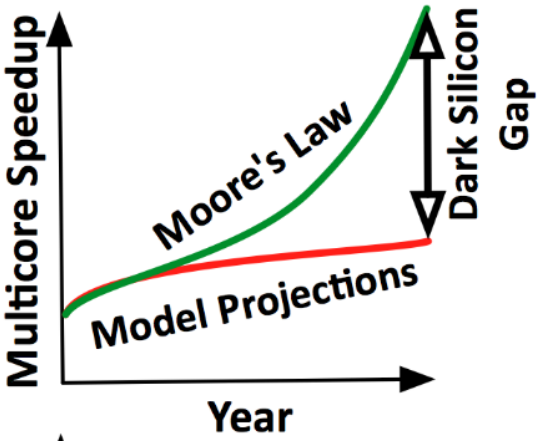
Why GPU Compute Devices?
Long Term View: Hybrid Computing

Dealing with massivly multi-core:

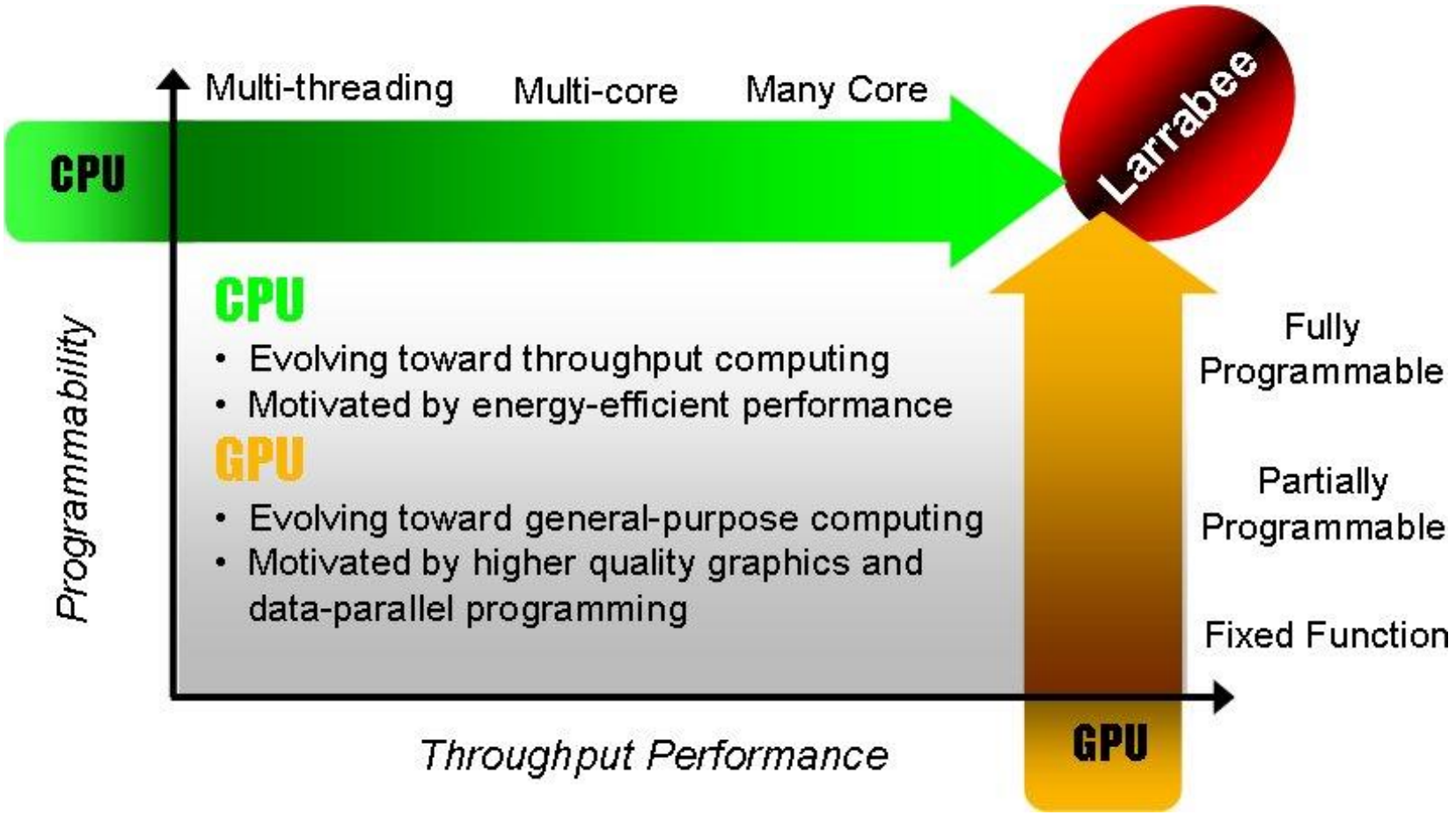
- New architectures are evaluated (Intel, IBM, HP,…)
- Accelerators that accompany common general purpose CPUs (Hybrid Systems)

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



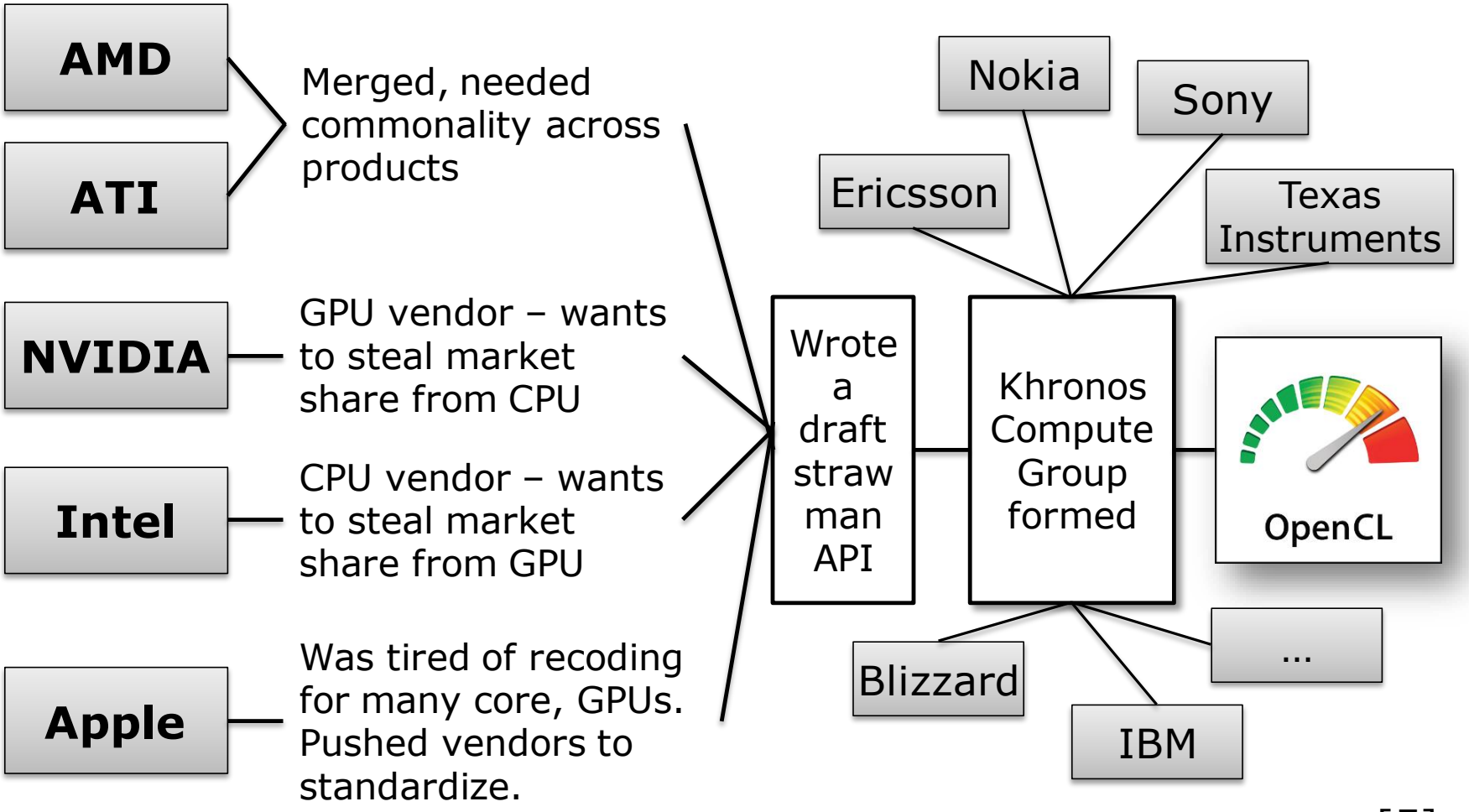
History of GPU Computing



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

Open Compute Language (OpenCL)



[5]

Open Compute Language (OpenCL)

- Hardware vendors, system OEMs, middleware vendors, application developers
- OpenCL became an important standard “on release” by virtue of the market coverage of the companies behind it.
- OpenCL implementations already exist for AMD, NVIDIA, Intel, IBM, ARM, …

- Use all computational resources in system
  - Program GPUs, CPUs, and other processors as peers
  - Efficient C-based parallel programming model
  - Abstract the specifics of underlying hardware
- Abstraction is **low-level, high-performance but device-portable**
  - Approachable – but primarily targeted at expert developers
  - Ecosystem foundation – no middleware or “convenience” functions
- Implementable on a range of embedded, desktop, and server systems
  - HPC, desktop, and handheld profiles in one specification

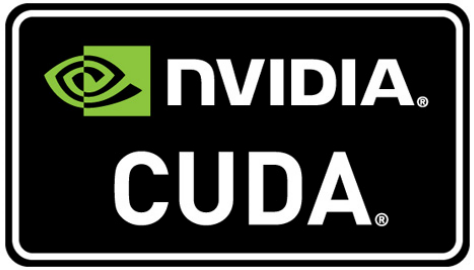
Programming Models

~~AMD: ATI Stream SDK~~

- Today using OpenCL

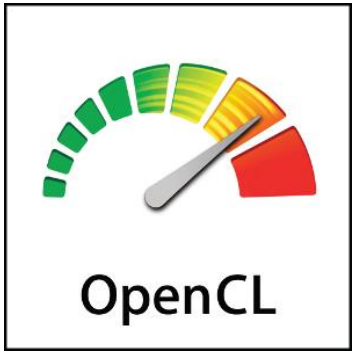
NVIDIA: Common Unified Device Architecture

- CUDA C/C++ compiler, libraries, runtime
- Mature: literature, examples, tool, development support

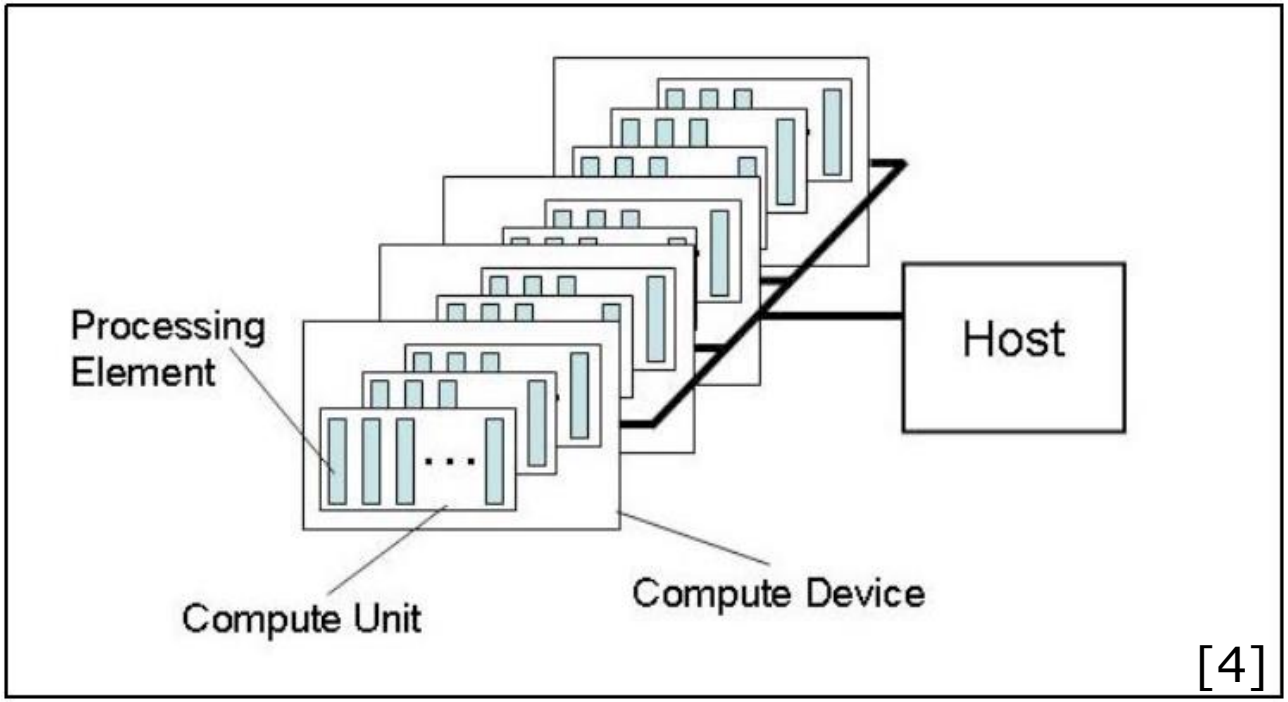


Khronos Group: OpenCL

Open standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors



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”

The BIG idea behind OpenCL

OpenCL execution model … execute a kernel at each point in a problem domain.

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

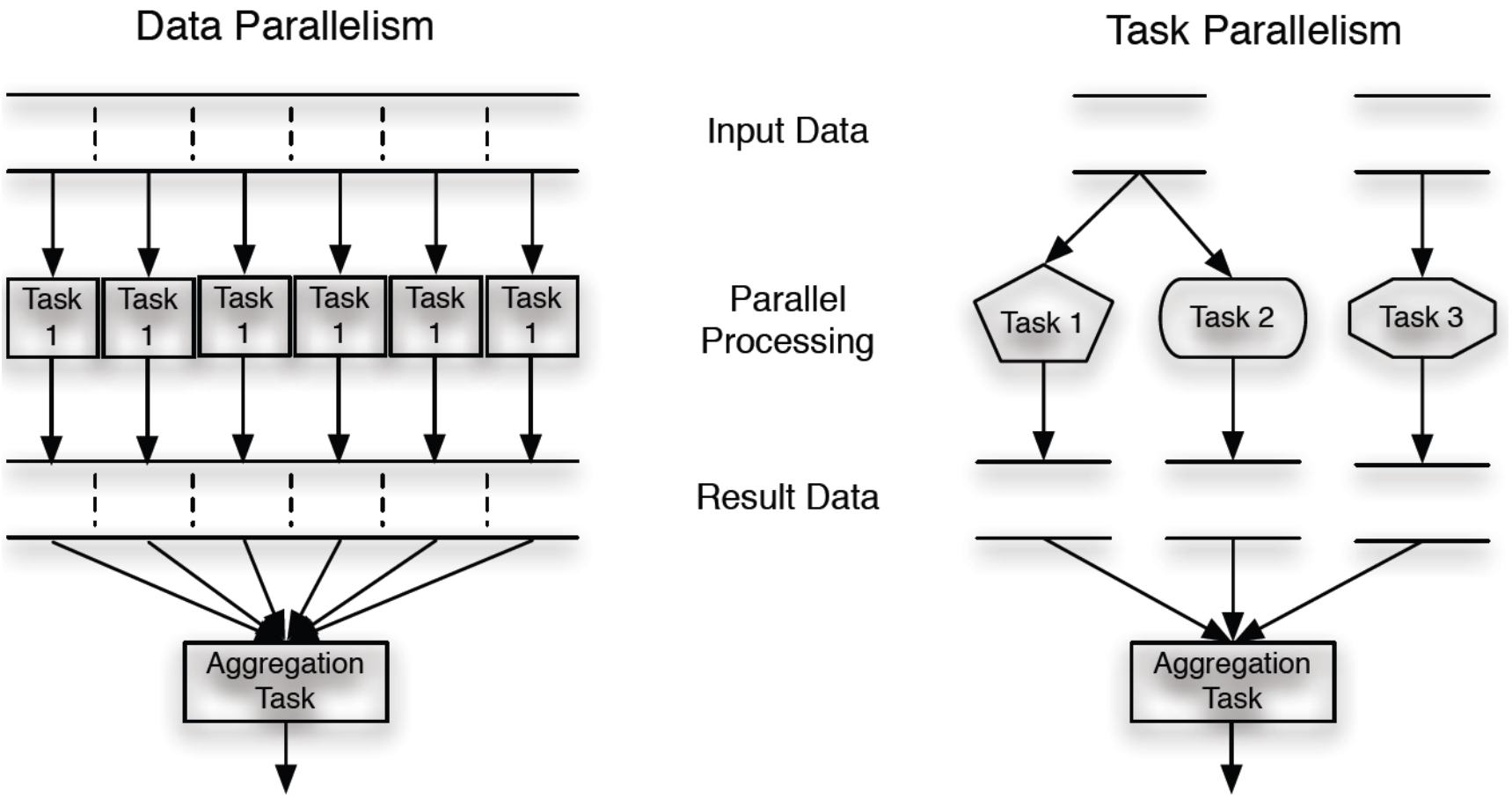
Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

E.g., process a 1024 x 1024 image with one kernel invocation per pixel or 1024 x 1024 = 1,048,576 kernel executions

[5]

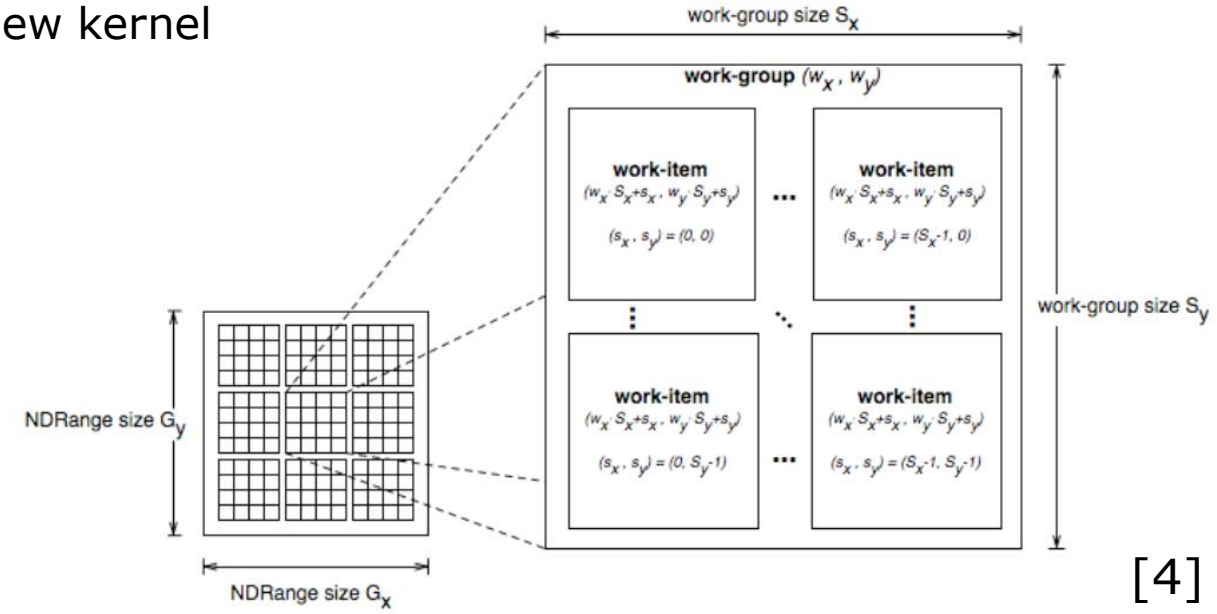
Data Parallelism versus Task Parallelism



OpenCL is designed for SIMD / SPMD approaches

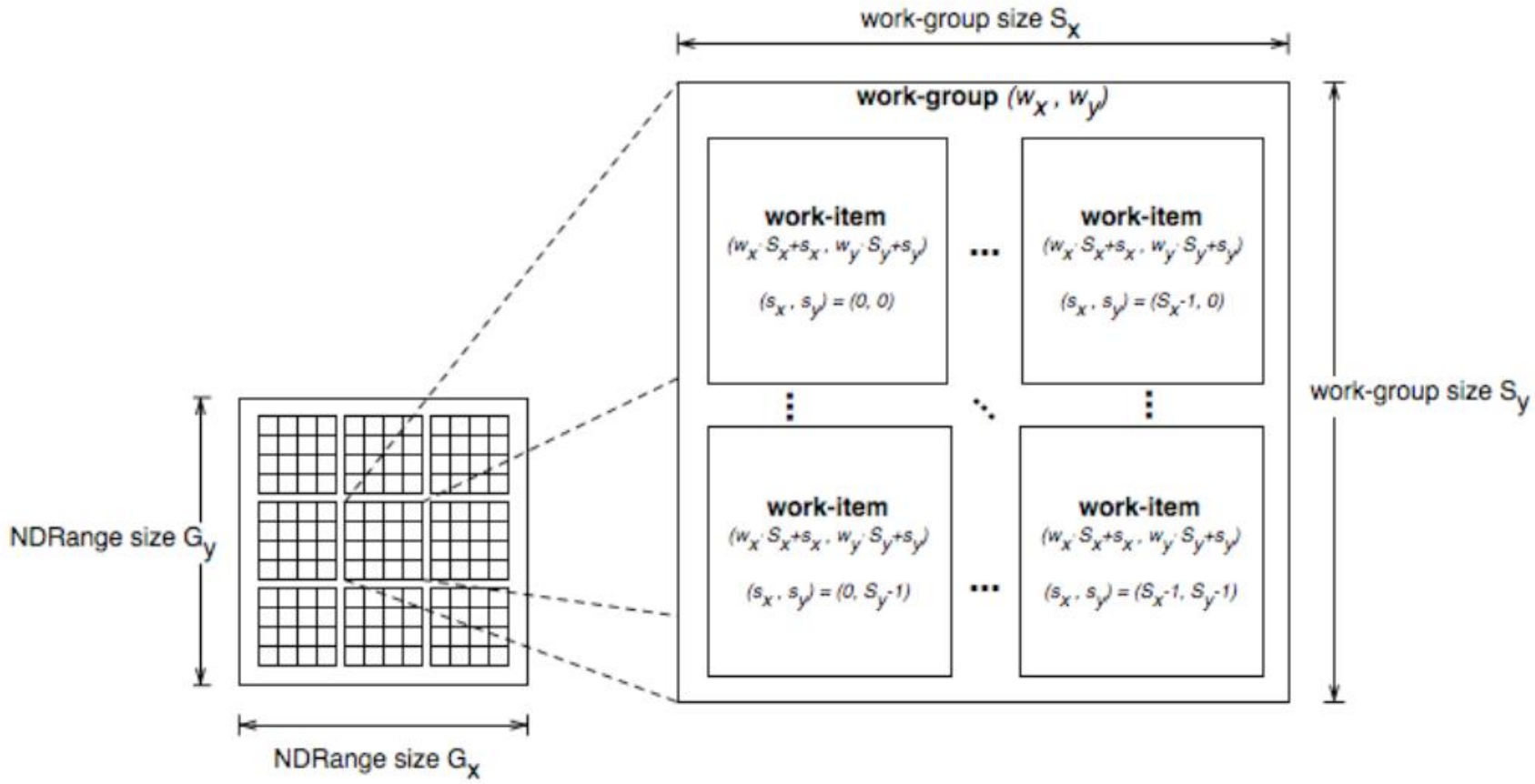
OpenCL Execution Model

- 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



[4]

OpenCL Execution Model

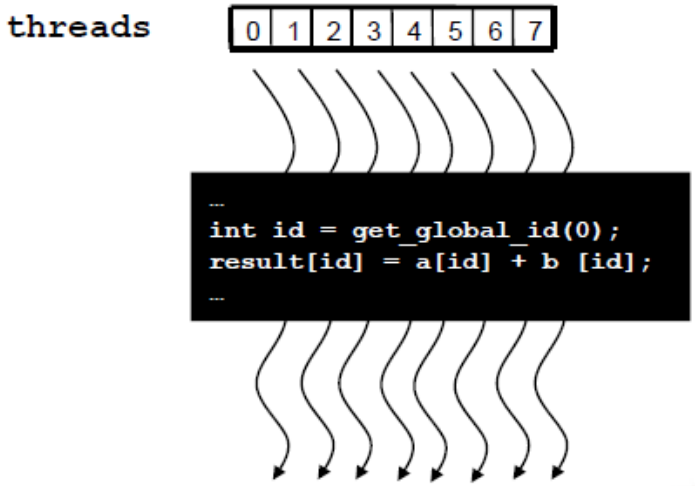


An example of an NDRange index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs. [4]

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

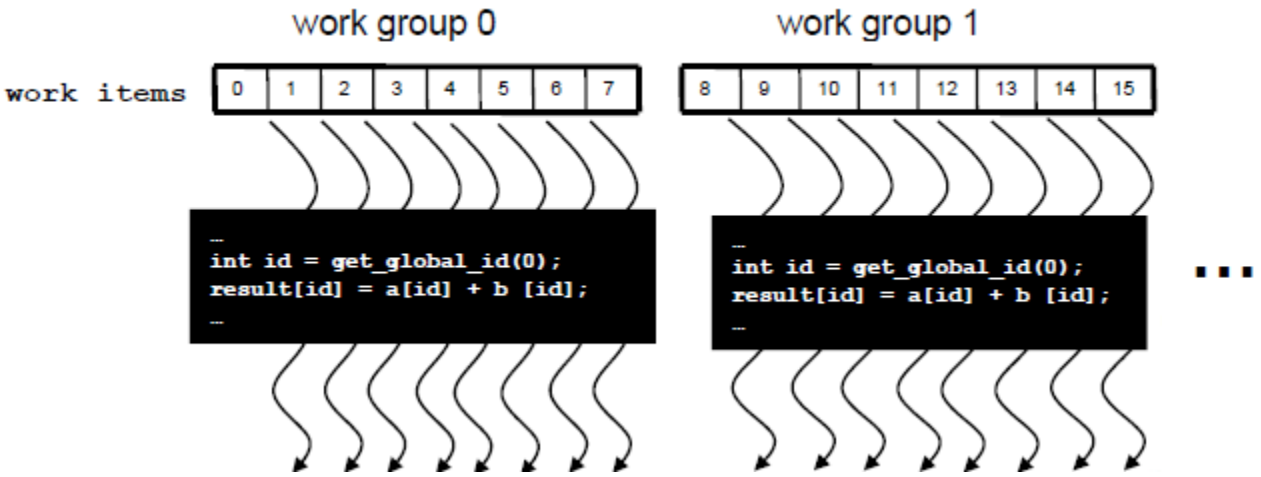


[1]

Work Groups: Scalable Cooperation

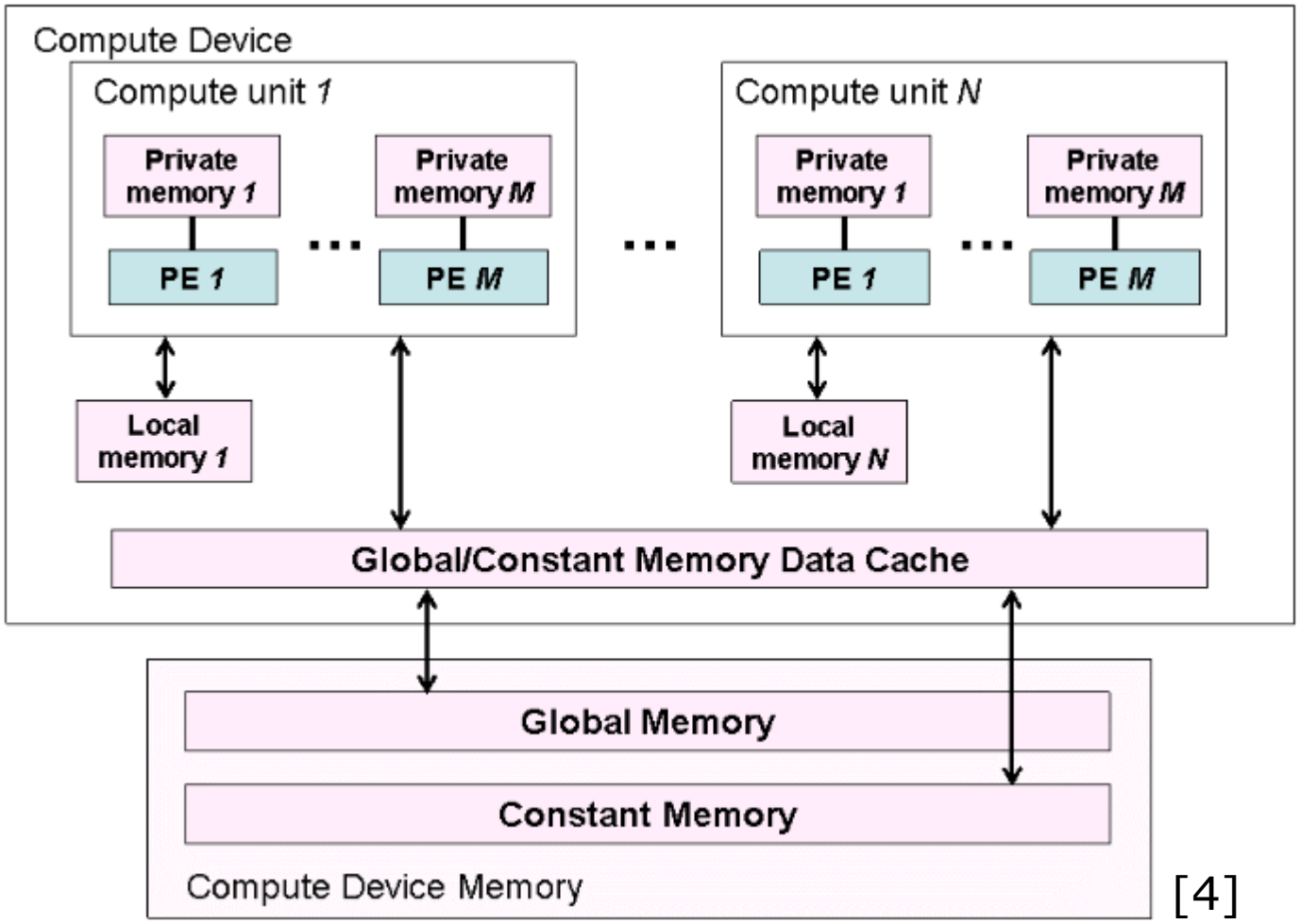
Divide monolithic work item array into work groups

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



[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

- 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

OpenCL Work Item Code

A subset of ISO C99 - without some C99 features

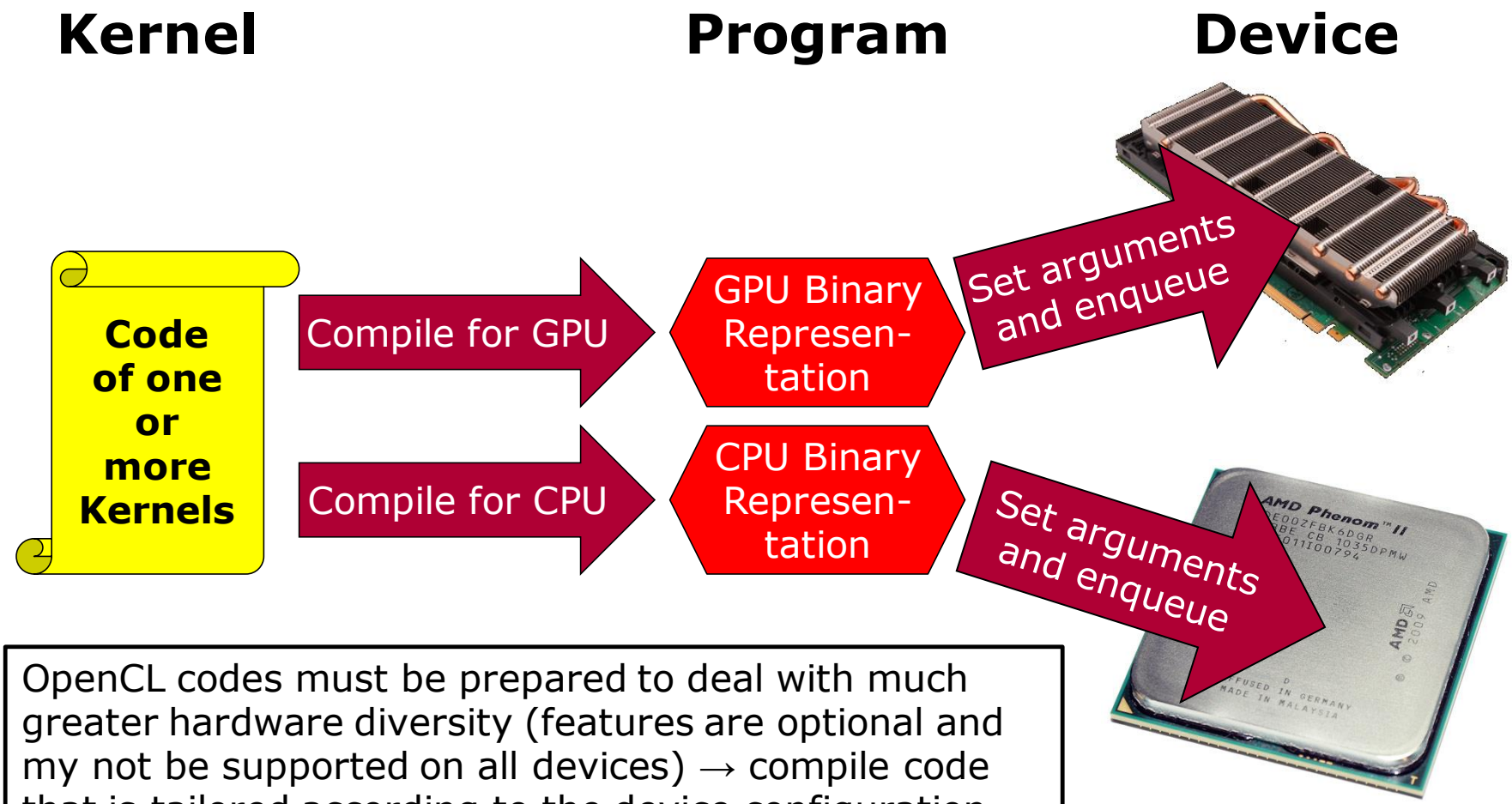
- headers, function pointers, recursion, variable length arrays, and bit fields

A superset of ISO C99 with additions for

- Work-items and workgroups
- Vector types (2,4,8,16): endian safe, aligned at vector length
- Image types mapped to texture memory
- Synchronization
- Address space qualifiers

Also includes a large set of built-in functions for image manipulation, work-item manipulation, specialized math routines, vectors, etc. [5]

Building and Executing OpenCL Code

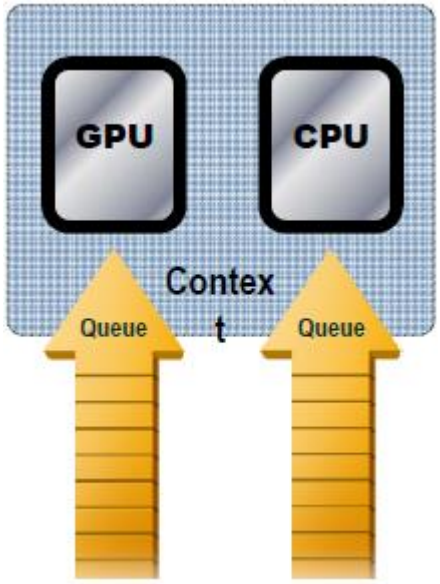


OpenCL codes must be prepared to deal with much greater hardware diversity (features are optional and my not be supported on all devices) → compile code that is tailored according to the device configuration

OpenCL Execution Model

An OpenCL application runs on a host which submits work to the compute devices. Kernels are executed in contexts defined and manipulated by the host.

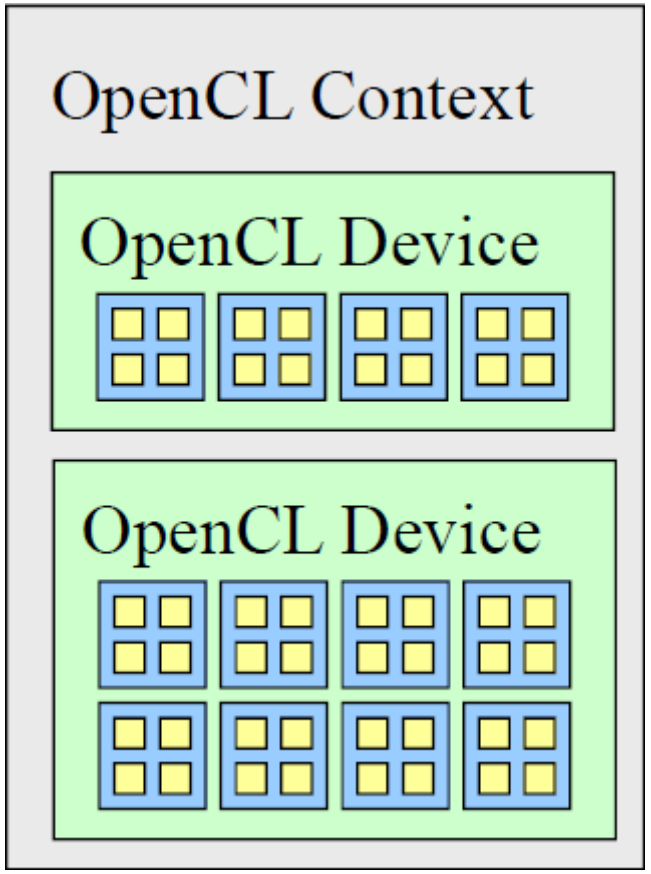
- **Work item**: the basic unit of work on an OpenCL device.
- **Kernel**: the code for a work item. Basically a C function
- **Program**: Collection of kernels and other functions (Analogous to a dynamic library)
- **Context**: The environment within which work-items executes … includes devices and their memories and command queues.
- **Queue**: used to manage a device. (copy memory, start work item, …) In-order vs. out-of-order execution



[5]

OpenCL Context

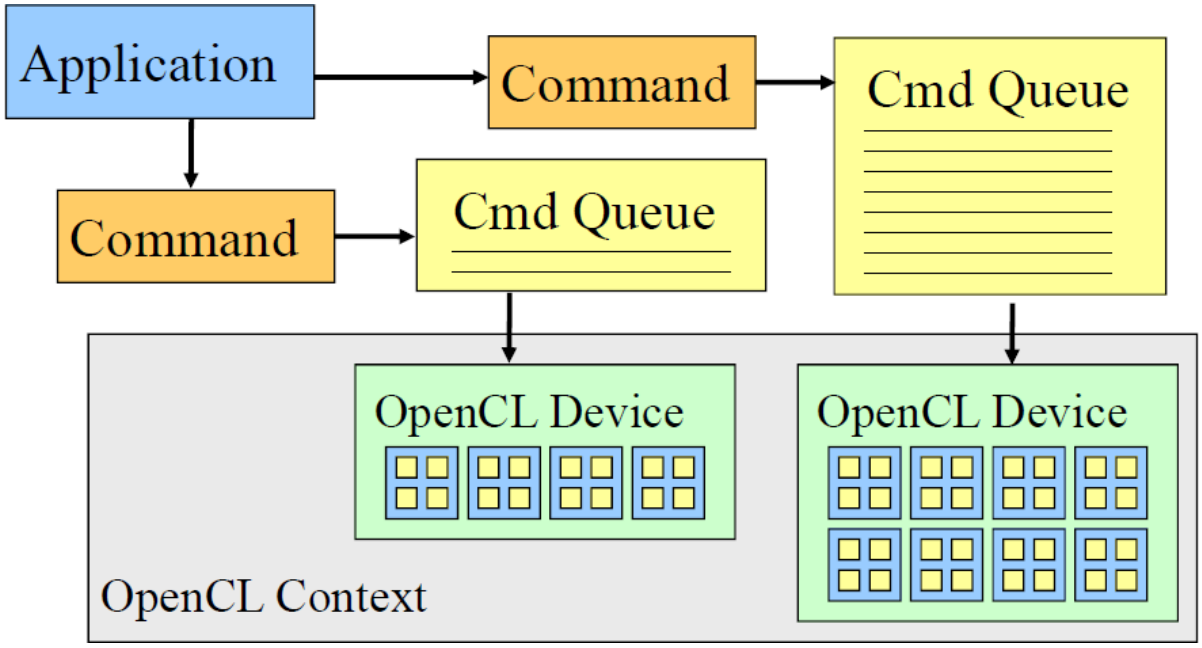
- Contains one or more devices
- OpenCL memory objects are associated with a context, not a specific device
- clCreateBuffer() is the main data object allocation function
  - error if an allocation is too large for any device in the context
- Each device needs its own work queue(s)
- Memory transfers are associated with a command queue (thus a specific device)



[1]

OpenCL Device Command Execution

- Command-queue - coordinates execution of kernels
  - Kernel execution commands
  - Memory commands: transfer or mapping of memory object data
  - Synchronization commands: constrains the order of commands



[1]

Vector Addition: Kernel

- Kernel body is instantiated once for each work item; each getting an unique index

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c)
 {
     int gid = get_global_id(0);
     c[gid] = a[gid] + b[gid];
}
```

» Code that actually executes on target devices

[5]

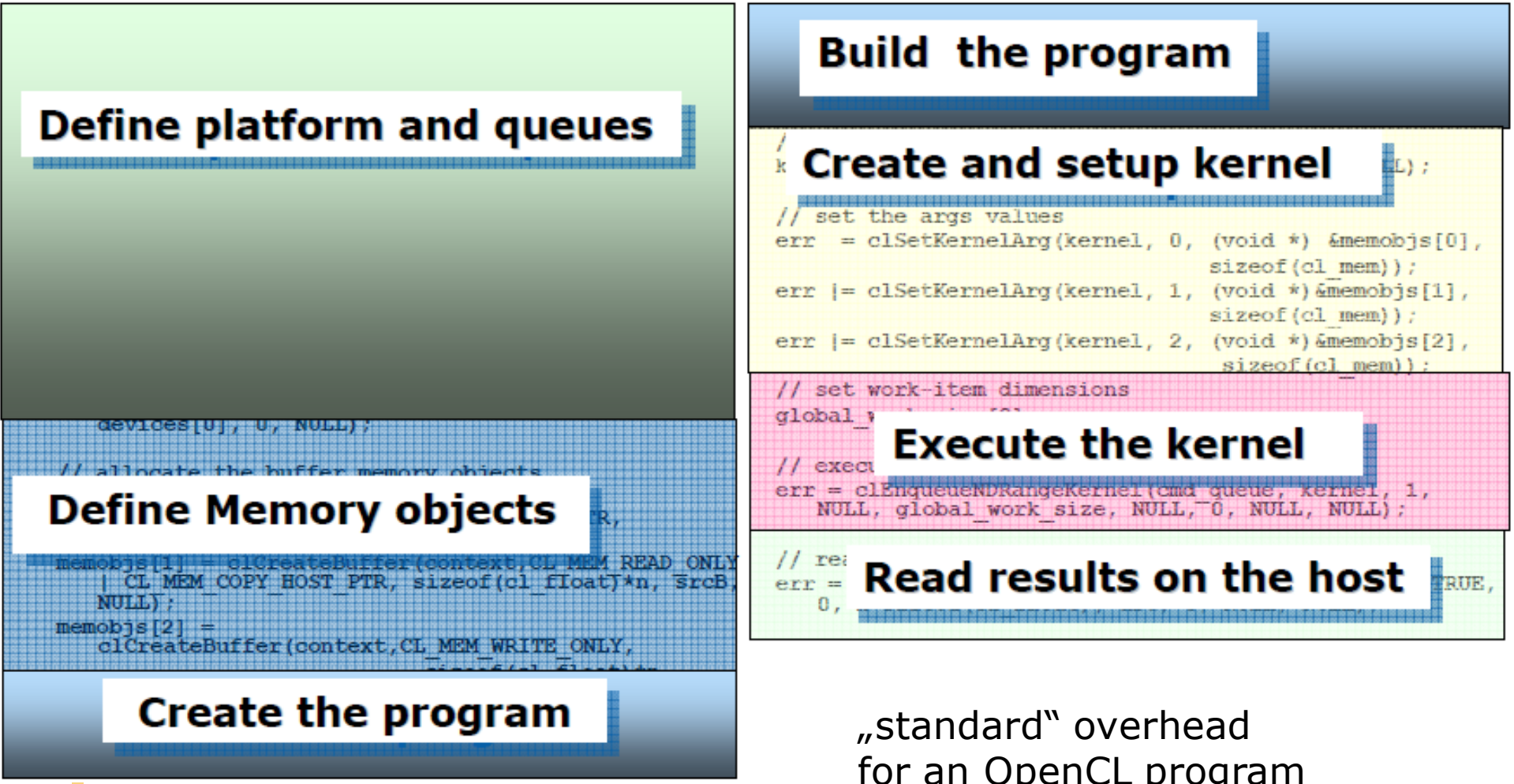
Vector Addition: Host Program

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
   CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with
   context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                        NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,
   devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context,
   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
   sizeof(cl_float)*n, srcA,
                                        NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY
   | CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
                                        NULL);
memobjs[2] =
   clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                          sizeof(cl_float)*n,
   NULL,
                                        NULL);
// create the program
program = clCreateProgramWithSource(context, 1,
   &program_source, NULL, NULL);
```

```
// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
                                              NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                  sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
   CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

[5]

Vector Addition: Host Program



„standard" overhead
for an OpenCL program

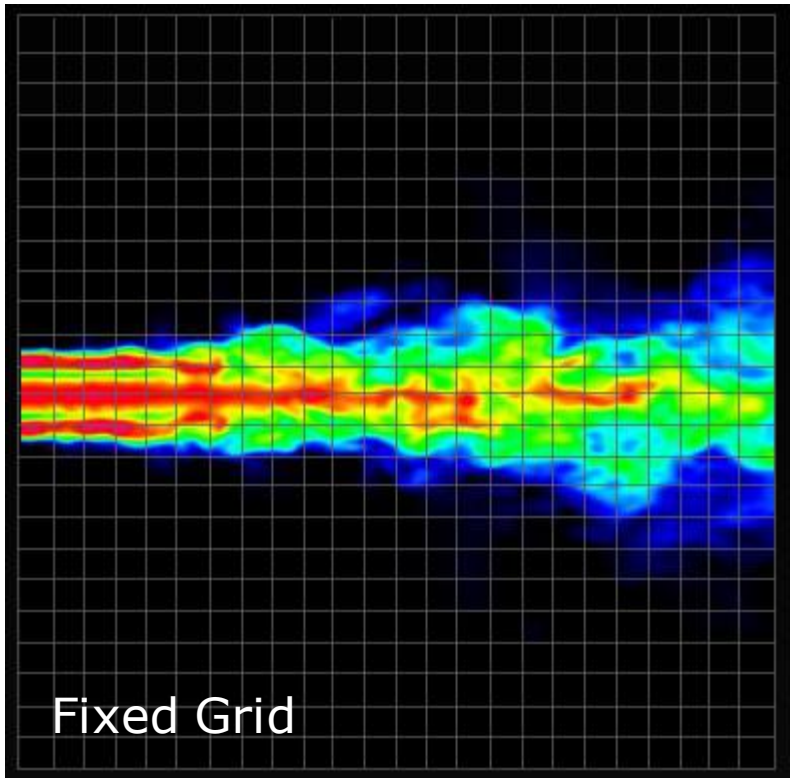
[5]

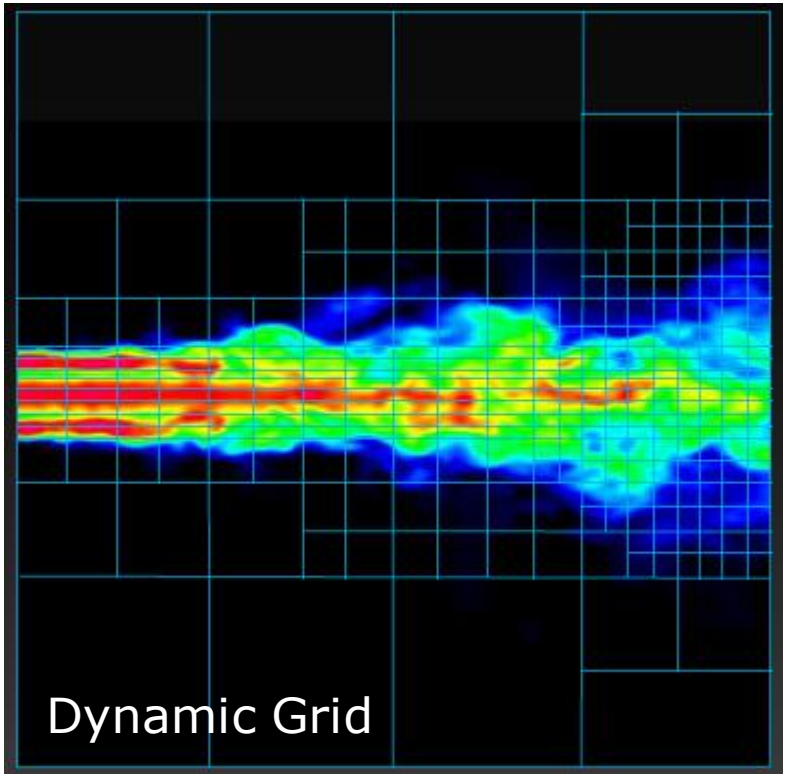
Live Demo

OpenCL “Hello Device”

Java+OpenCL “Fractal Explorer”

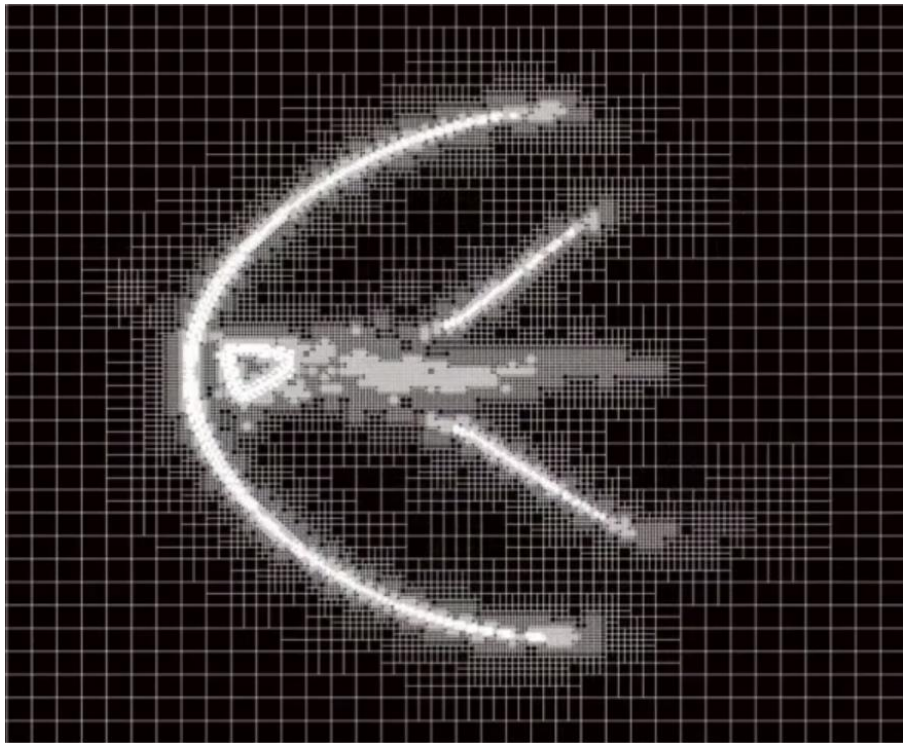
Dynamic Parallelism: The Idea

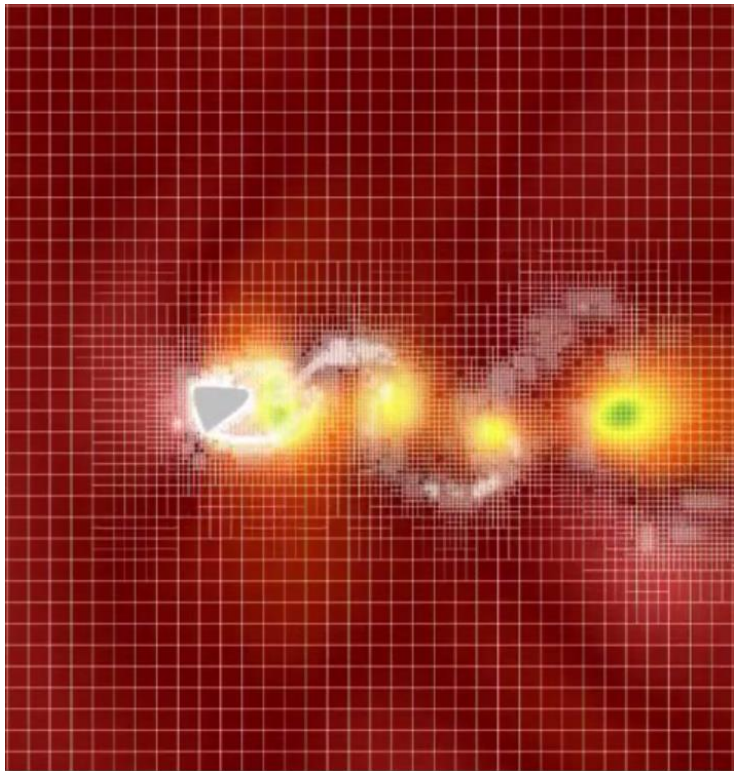




[10]

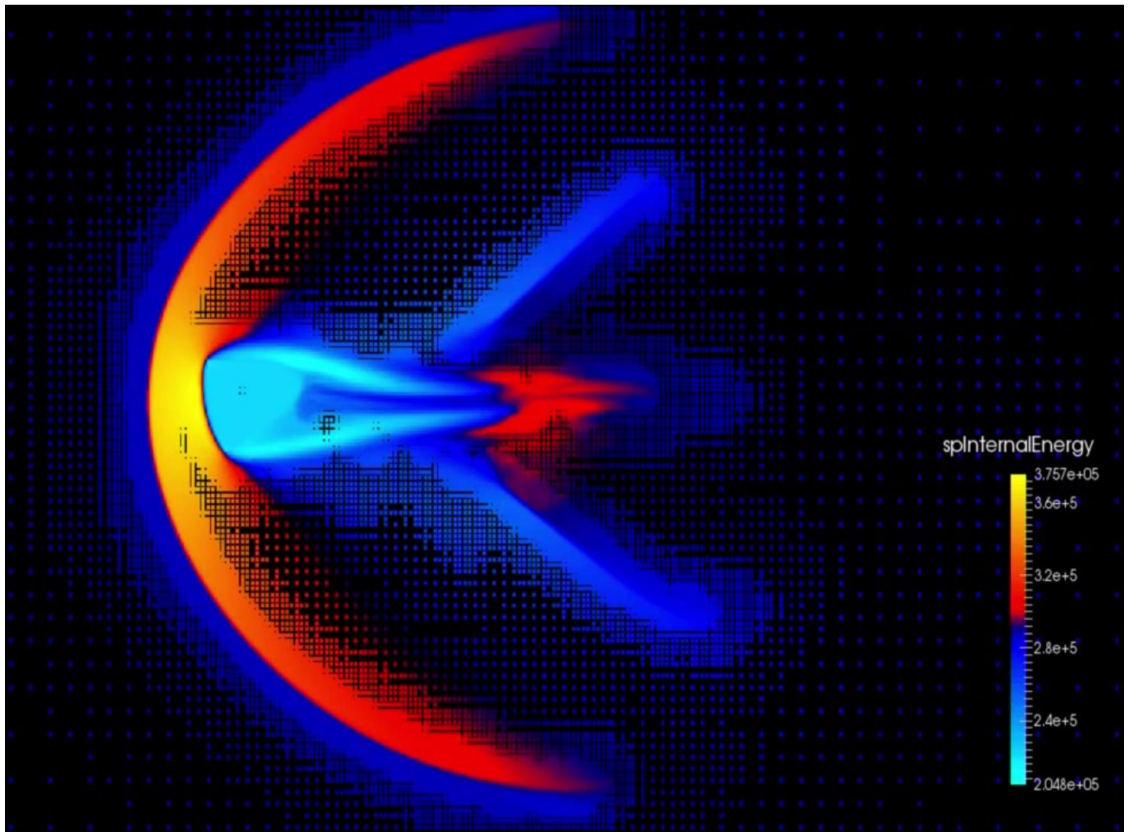
Dynamic Parallelism in Action: SpaceX





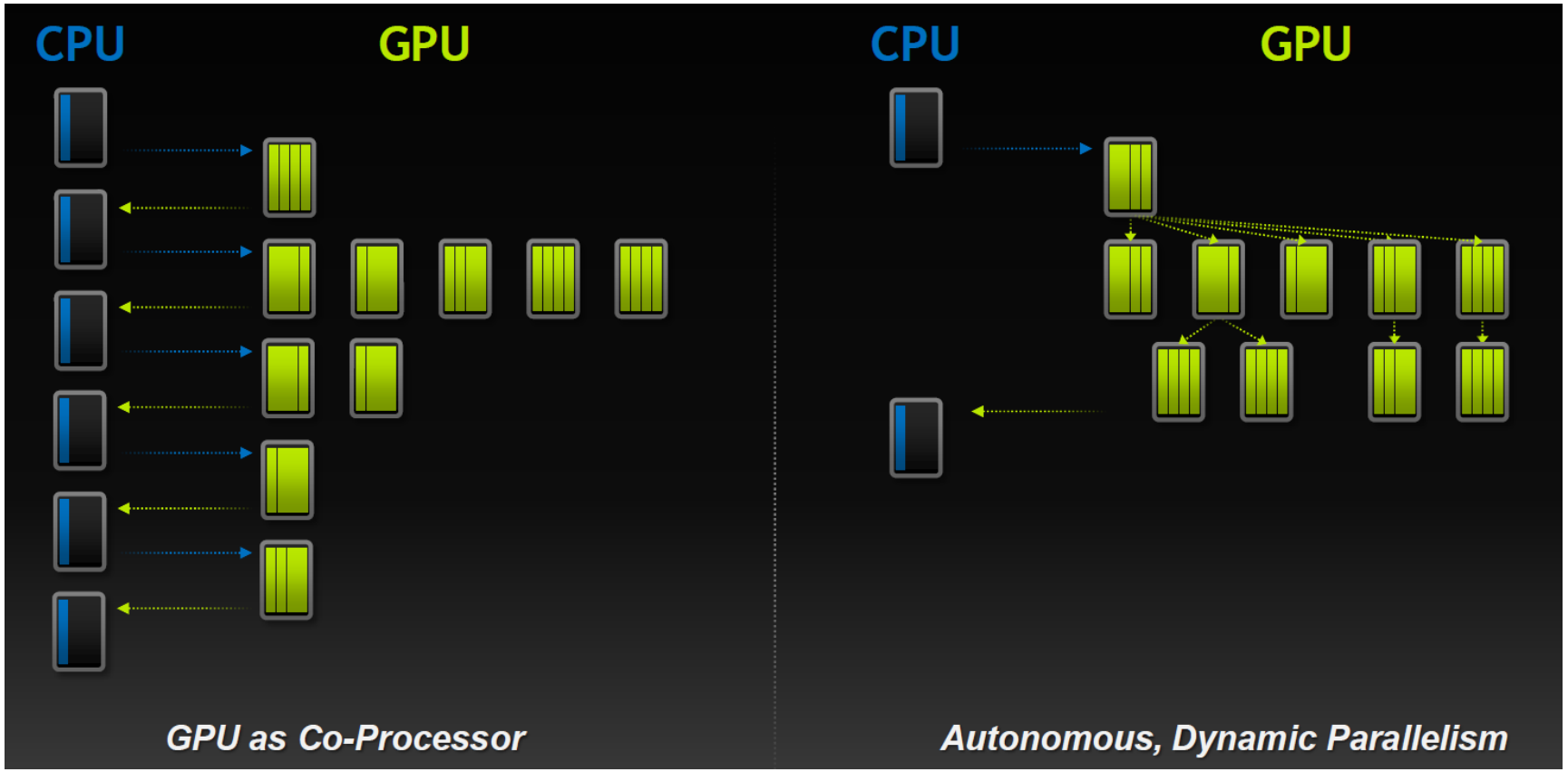
http://on-demand.gputechconf.com/gtc/2015/video/S5398.html

Dynamic Parallelism in Action: SpaceX



http://on-demand.gputechconf.com/gtc/2015/video/S5398.html

Dynamic Parallelism: Task Distribution

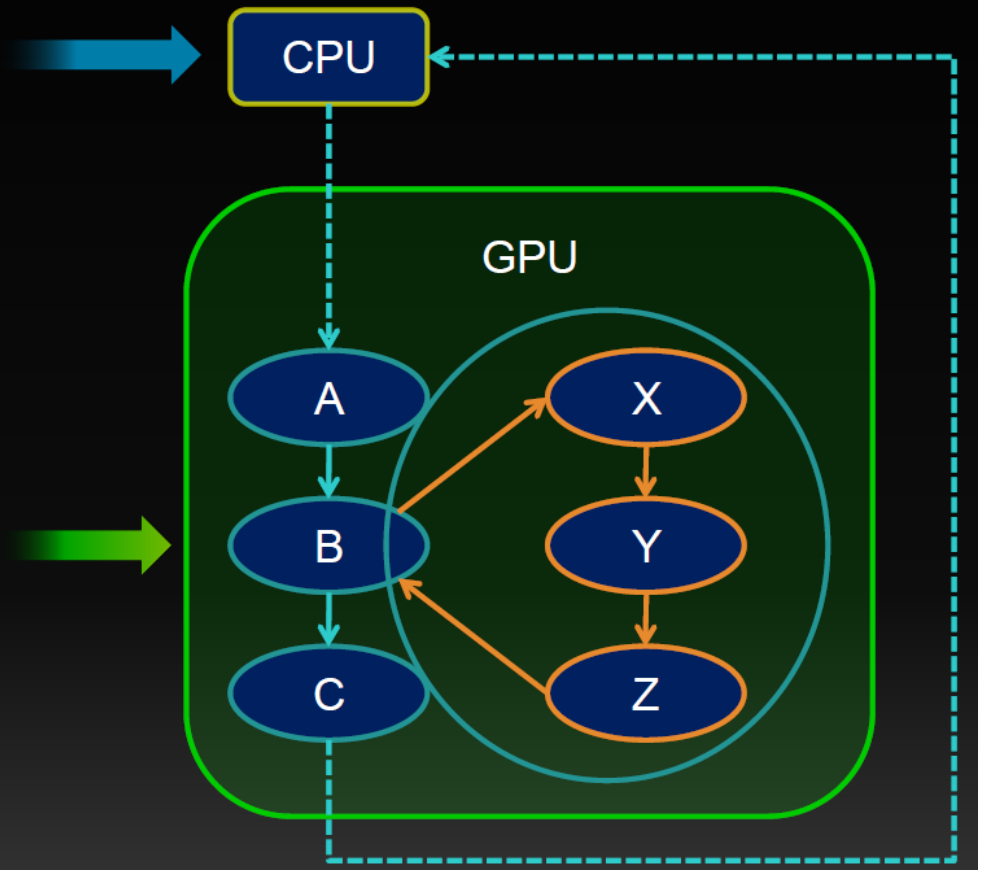


[10]

Dynamic Parallelism: Familiar Syntax


```
void main() {
    float *data;
    do_stuff(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);
    cudaDeviceSynchronize();

    do_more_stuff(data);
}
```

```
__global__ void B(float *data)
{
    do_stuff(data);

    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);
    cudaDeviceSynchronize();

    do_more_stuff(data);
}
```



[10]

Dynamic Parallelism: Code Example

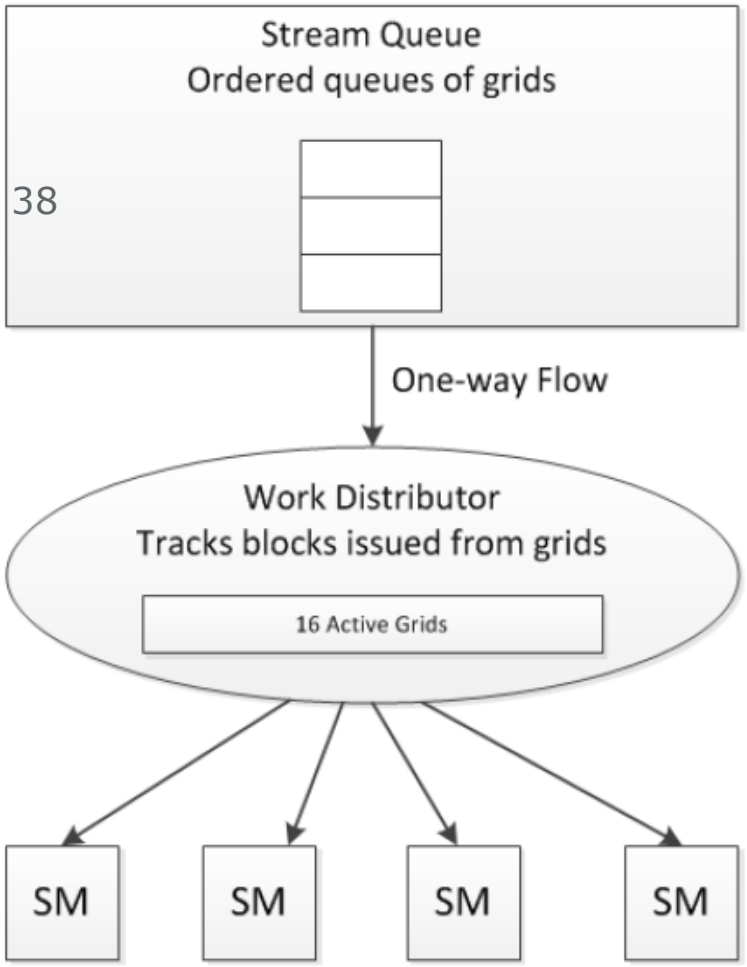
- CUDA Runtime syntax & semantics
- Launch is per-thread
- Sync includes all launches by any thread in the block
- *cudaDeviceSynchronize()* does not imply syncthreads
- Asynchronous launches only (note bug in program, here!)

Code Example

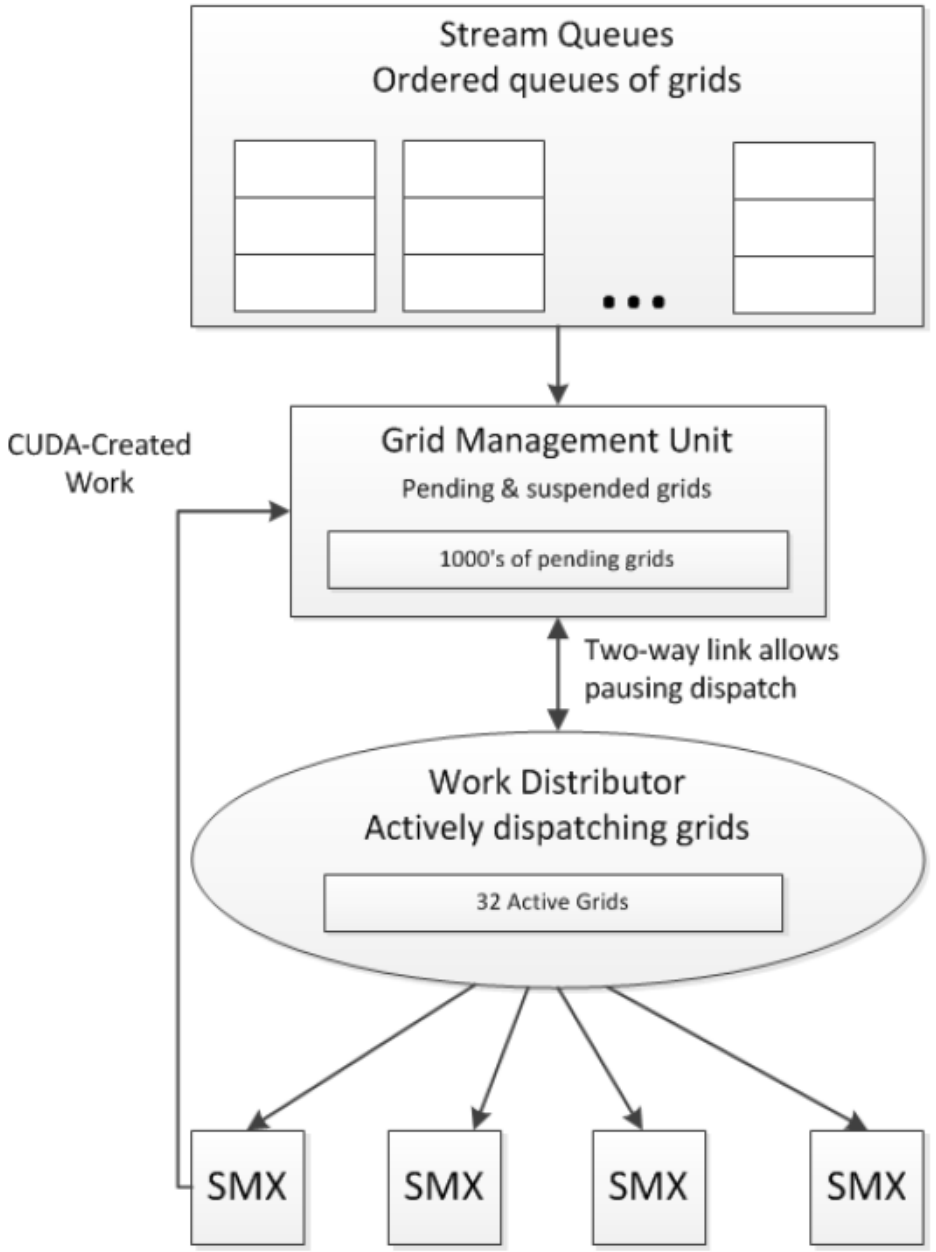
```
__device__ float buf[1024];
__global__ void dynamic(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];
    __syncthreads();

    if(tid == 0) {
        launch<<< 128, 256 >>>(buf);
        cudaDeviceSynchronize();
    }
    __syncthreads();

    cudaMemcpyAsync(data, buf, 1024);
    cudaDeviceSynchronize();
}
```

[10]

Fermi Workflow



Kepler Workflow



Development Support

Software development kits: NVIDIA and AMD; Windows and Linux

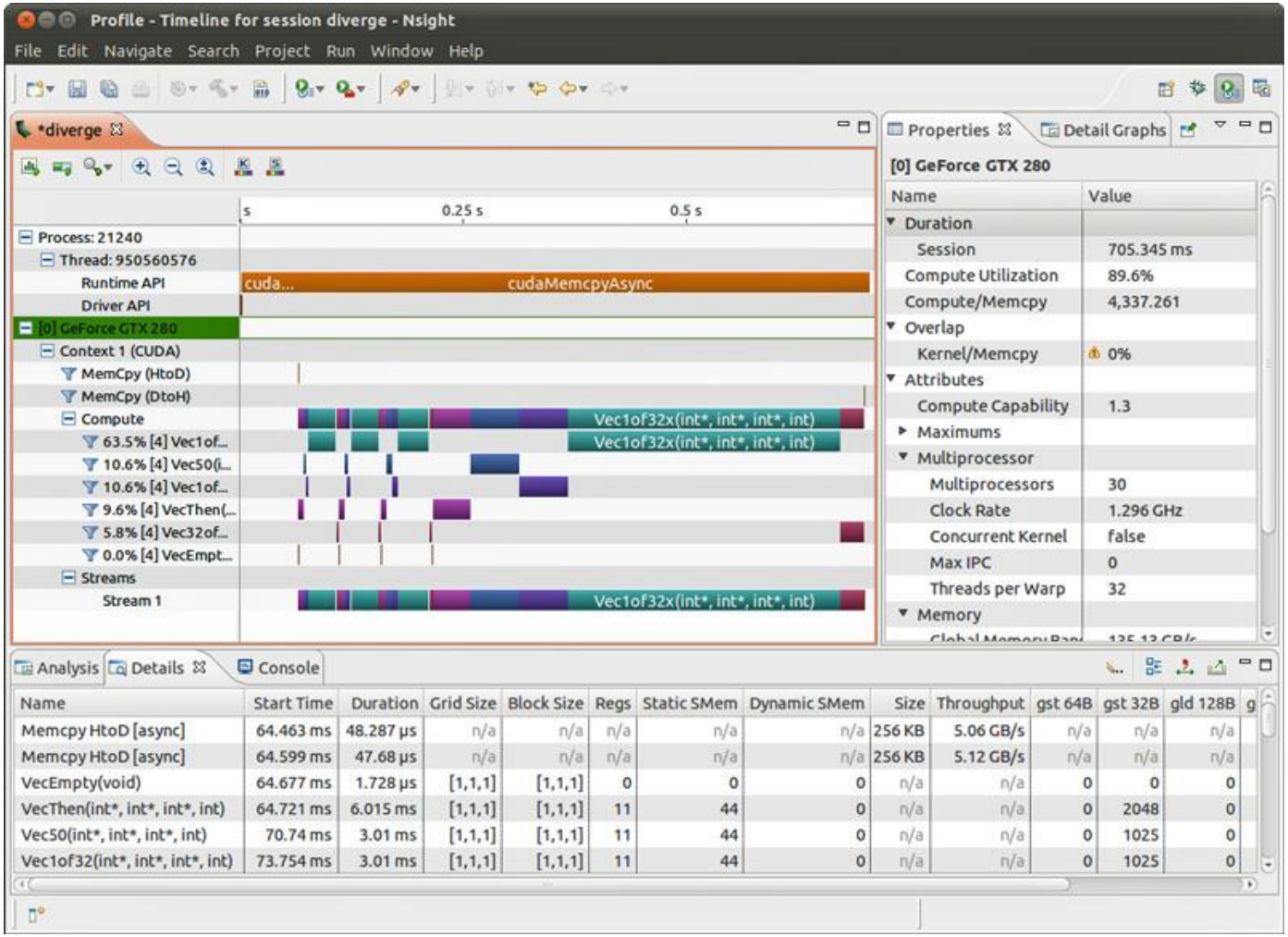
Special libraries: AMD Core Math Library, BLAS and FFT libraries by NVIDIA, OpenNL for numerics and CULA for linear algebra; NVIDIA Performance Primitives library: a sink for common GPU accelerated algorithms

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
- gDEBugger (platform-independent)

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

Nsight


Profile - Timeline for session diverge - Nsight
File Edit Navigate Search Project Run Window Help
*diverge
Properties
Detail Graphs
[0] GeForce GTX 280
Name
Value
Duration
Session
705.345 ms
Compute Utilization
89.6%
Compute/Memcpy
4,337.261
Overlap
Kernel/Memcpy
0%
Attributes
Compute Capability
1.3
Maximums
Multiprocessor
Multiprocessors
30
Clock Rate
1.296 GHz
Concurrent Kernel
false
Max IPC
0
Threads per Warp
32
Memory
0.25 s
0.5 s
Process: 21240
Thread: 950560576
Runtime API
cuda...
cudaMemcpyAsync
Driver API
[0] GeForce GTX 280
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
Vec1of32x(int*, int*, int*, int)
63.5% [4] Vec1of...
10.6% [4] Vec50(i...
10.6% [4] Vec1of...
9.6% [4] VecThen(...
5.8% [4] Vec32of...
0.0% [4] VecEmpt...
Streams
Stream 1
Analysis
Details
Console
Name
Start Time
Duration
Grid Size
Block Size
Regs
Static SMem
Dynamic SMem
Size
Throughput
gst 64B
gst 32B
gld 128B
Memcpy HtoD [async]
64.463 ms
48.287 µs
256 KB
5.06 GB/s
Memcpy HtoD [async]
64.599 ms
47.68 µs
256 KB
5.12 GB/s
VecEmpty(void)
64.677 ms
1.728 µs
[1,1,1]
VecThen(int*, int*, int*, int)
64.721 ms
6.015 ms
2048
Vec50(int*, int*, int*, int)
70.74 ms
3.01 ms
1025
Vec1of32(int*, int*, int*, int)
73.754 ms

Towards new Platforms

WebCL [Draft] http://www.khronos.org/webcl/

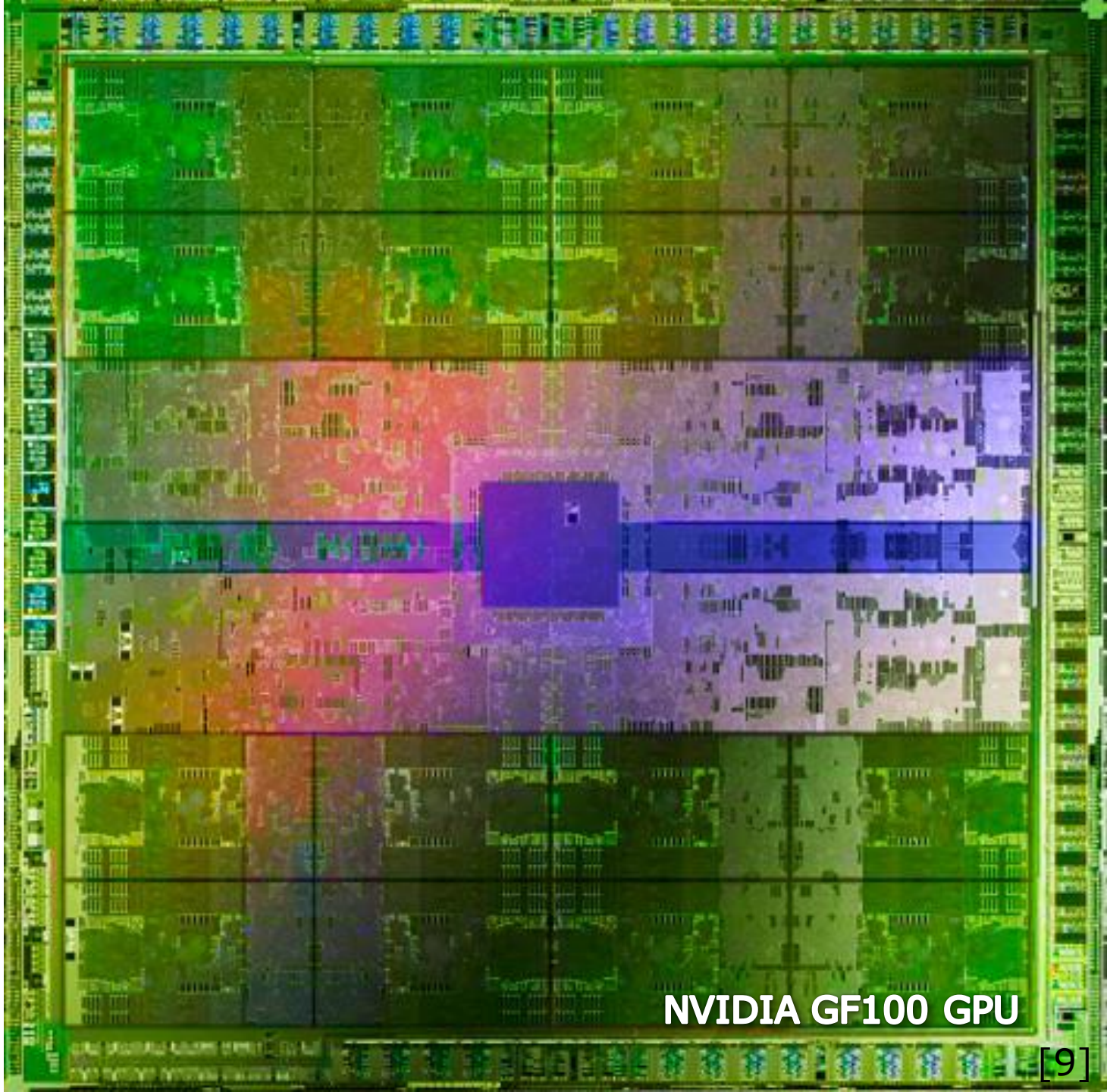
- JavaScript binding to OpenCL
- Heterogeneous Parallel Computing (CPUs + GPU) within Web Browsers
- Enables compute intense programs like physics engines, video editing…
- Currently only available with add-ons (Node.js, Firefox, WebKit)

Android installable client driver extension (ICD)

- Enables OpenCL implementations to be discovered and loaded as a shared object on Android systems.

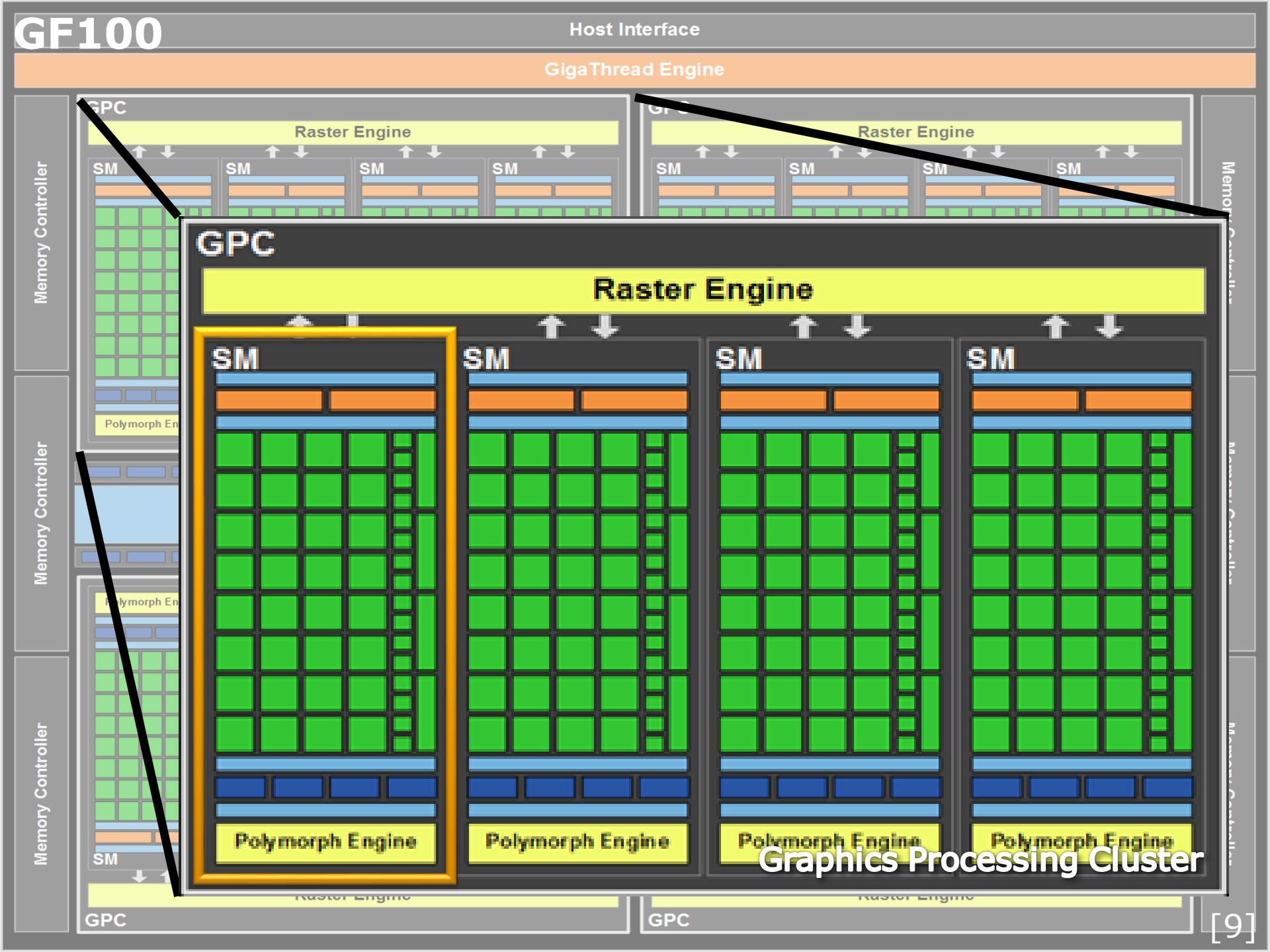
PGI for multicore ARM processors

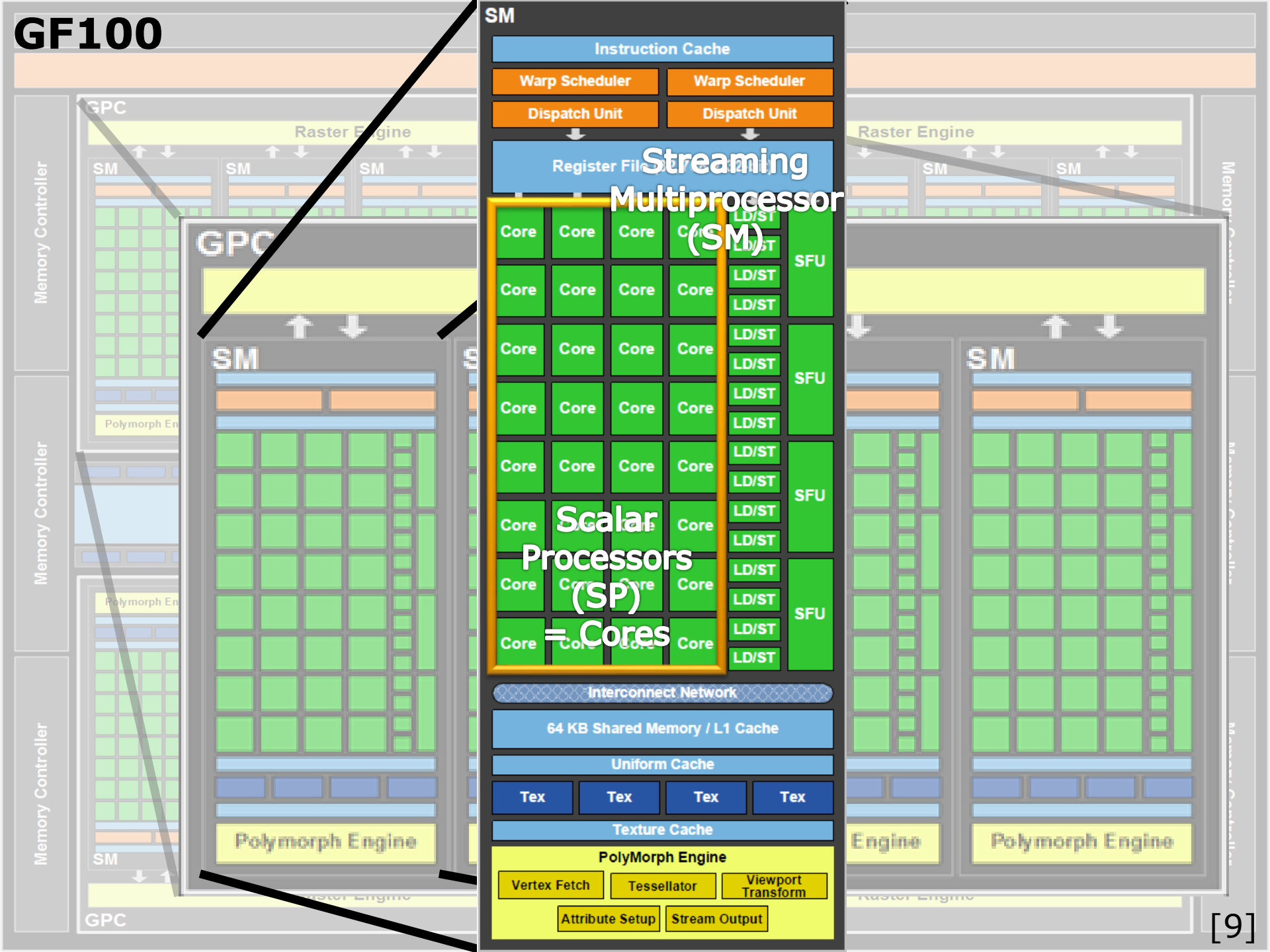
GPU Hardware in Detail



[9]

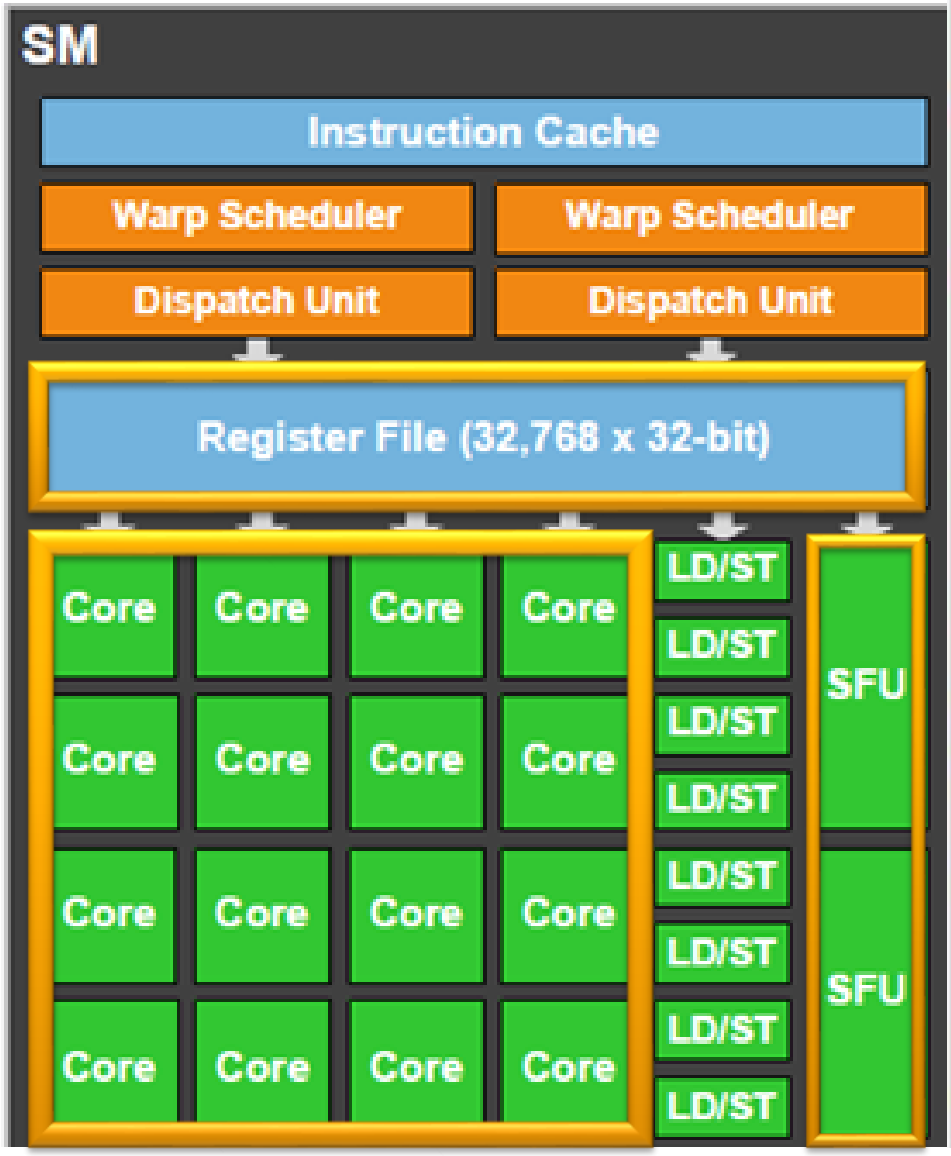

GF100
Host Interface
GigaThread Engine
GPC
Raster Engine
SM
Polymorph Engine
Graphics Processing Cluster
Memory Controller
L2 Cache

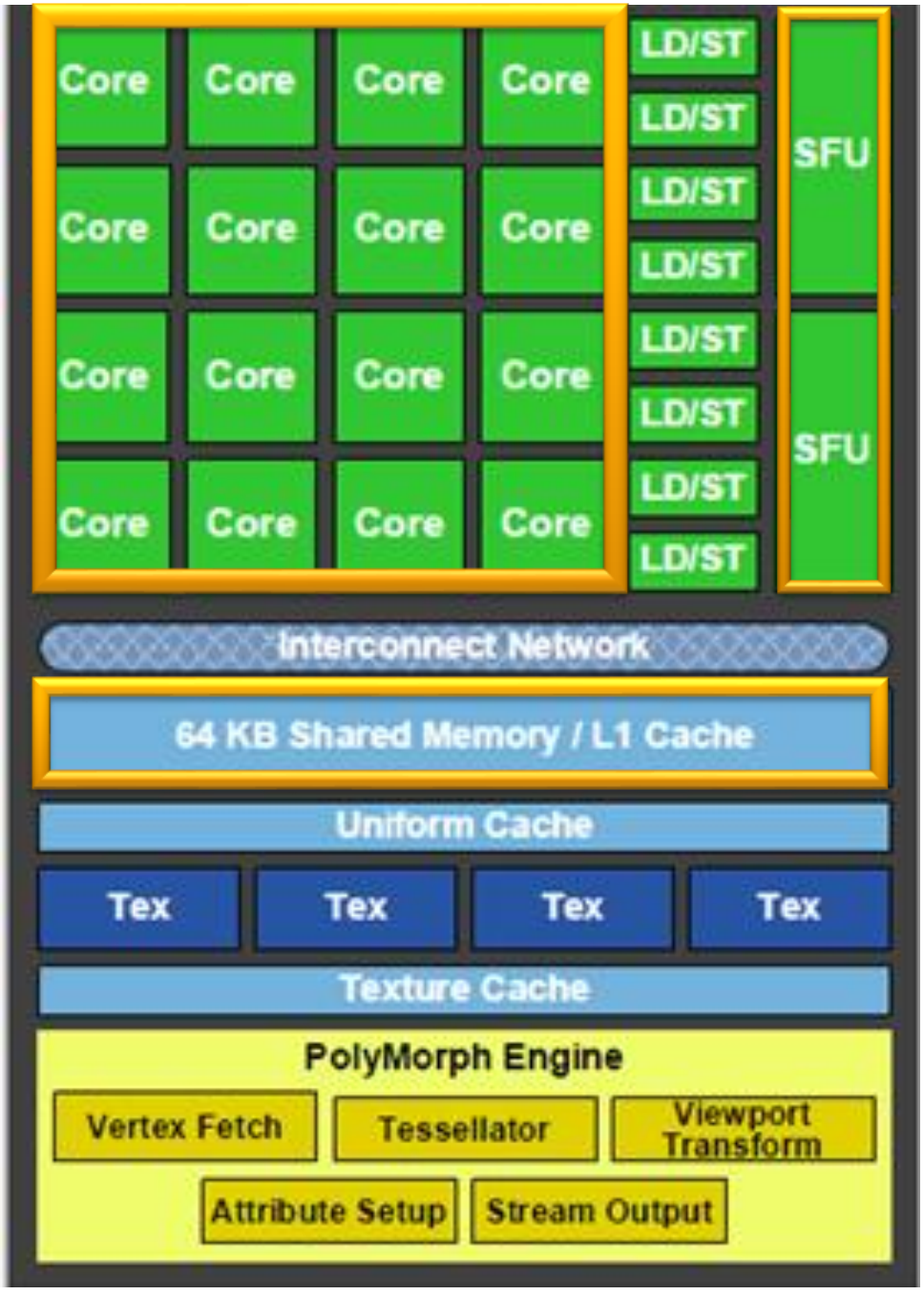

GF100
Host Interface
GigaThread Engine
Memory Controller
Memory Controller
Memory Controller
GPC
Raster Engine
SM
SM
SM
SM
Polymorph Engine
Polymorph Engine
Polymorph Engine
Polymorph Engine
Graphics Processing Cluster
GPC
GPC


GF100
SM
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File
Core
LD/ST
SFU
Streaming Multiprocessor (SM)
Scalar Processors (SP) = Cores
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache
Tex
Texture Cache
PolyMorph Engine
Vertex Fetch
Tessellator
Viewport Transform
Attribute Setup
Stream Output
GPC
Raster Engine
Polymorph Engine
Memory Controller

[9]

GF100

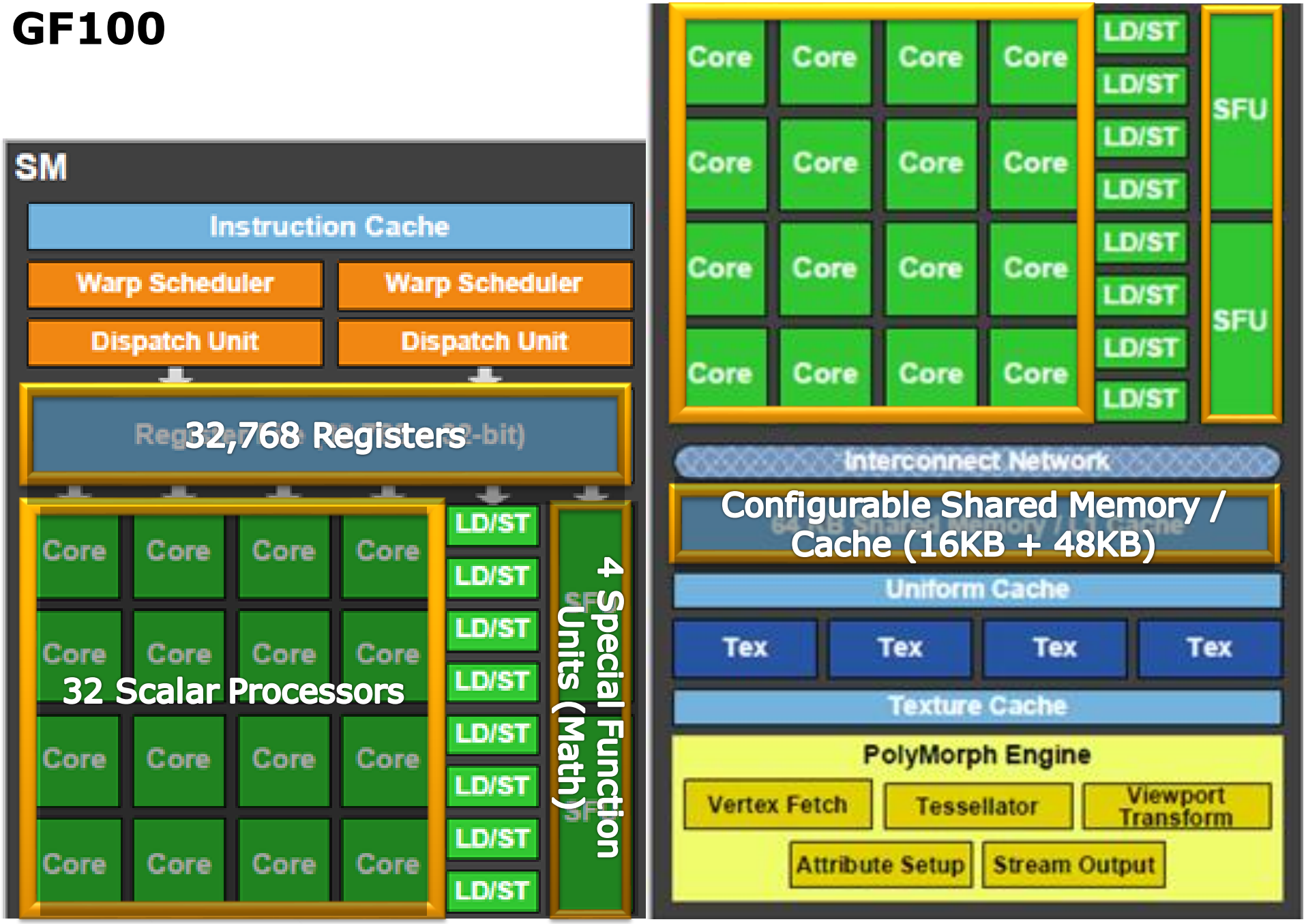

SM
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
Core
LD/ST
SFU


Core
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache
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
LD/ST
32 Scalar Processors
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 – early architecture

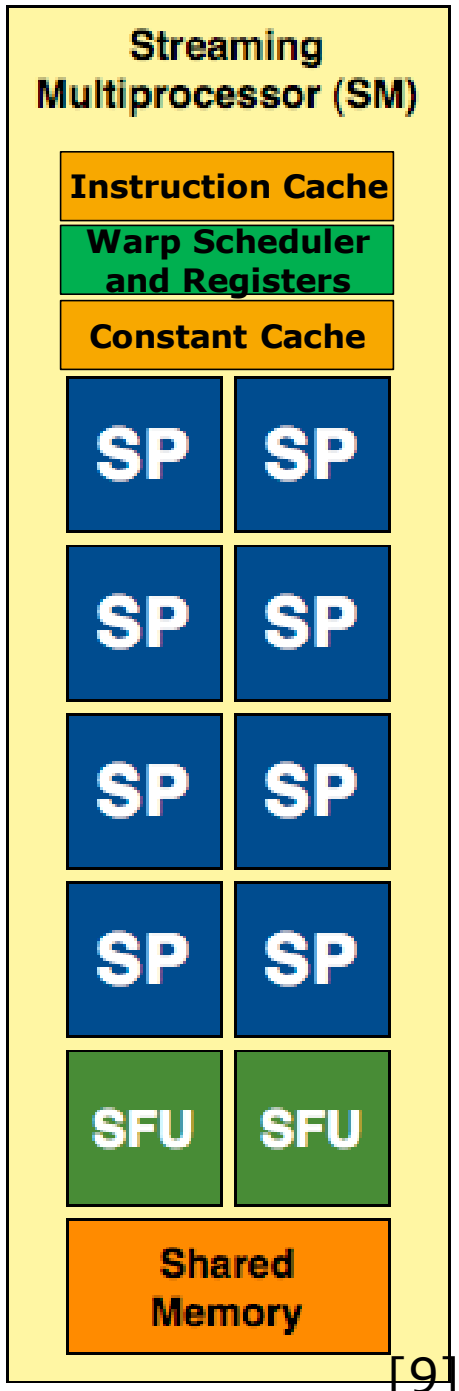
Simpler architecture, but same principles

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:

Max(16.384/10, 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

OpenCL Platforms

AMD APP SDK (supports OpenCL CPU and accelerated processing unit Devices)	X86 + SSE2 (or higher) compatible CPUs 64-bit & 32-bit;[77] Linux 2.6 PC, Windows Vista/7 PC	AMD Fusion E-350, E-240, C-50, C-30 with HD 6310/HD 6250	AMD Radeon/Mobility HD 6800, HD 5x00 series GPU, iGPU HD 6310/HD 6250	ATI FirePro Vx800 series GPU
Intel SDK for OpenCL Applications 2013[78] (supports Intel Core processors and Intel HD Graphics 4000/2500)	Intel CPUs with SSE 4.1, SSE 4.2 or AVXsupport.[79][80] Microsoft Windows, Linux	Intel Core i7, i5, i3; 2nd Generation Intel Core i7/5/3, 3rd Generation Intel Core Processors with Intel HD Graphics 4000/2500	Intel Core 2 Solo, Duo Quad, Extreme	Intel Xeon 7x00,5x00,3x00 (Core based)
IBM Servers with OpenCL Development Kit for Linux on Power running on Power VSX[81][82]	IBM Power 755 (PERCS), 750	IBM BladeCenter PS70x Express	IBM BladeCenter JS2x, JS43	IBM BladeCenter QS22
IBM OpenCL Common Runtime (OCR)[83]	X86 + SSE2 (or higher) compatible CPUs 64-bit & 32-bit;[84] Linux 2.6 PC	AMD Fusion, Nvidia Ion and Intel Core i7, i5, i3; 2nd Generation Intel Core i7/5/3	AMD Radeon, Nvidia GeForce and Intel Core 2 Solo, Duo, Quad, Extreme	ATI FirePro, Nvidia Quadro and Intel Xeon 7x00,5x00,3x00 (Core based)
Nvidia OpenCL Driver and Tools[85]	Nvidia Tesla C/D/S	Nvidia GeForce GTS/GT/GTX	Nvidia Ion	Nvidia Quadro FX/NVX/Plex

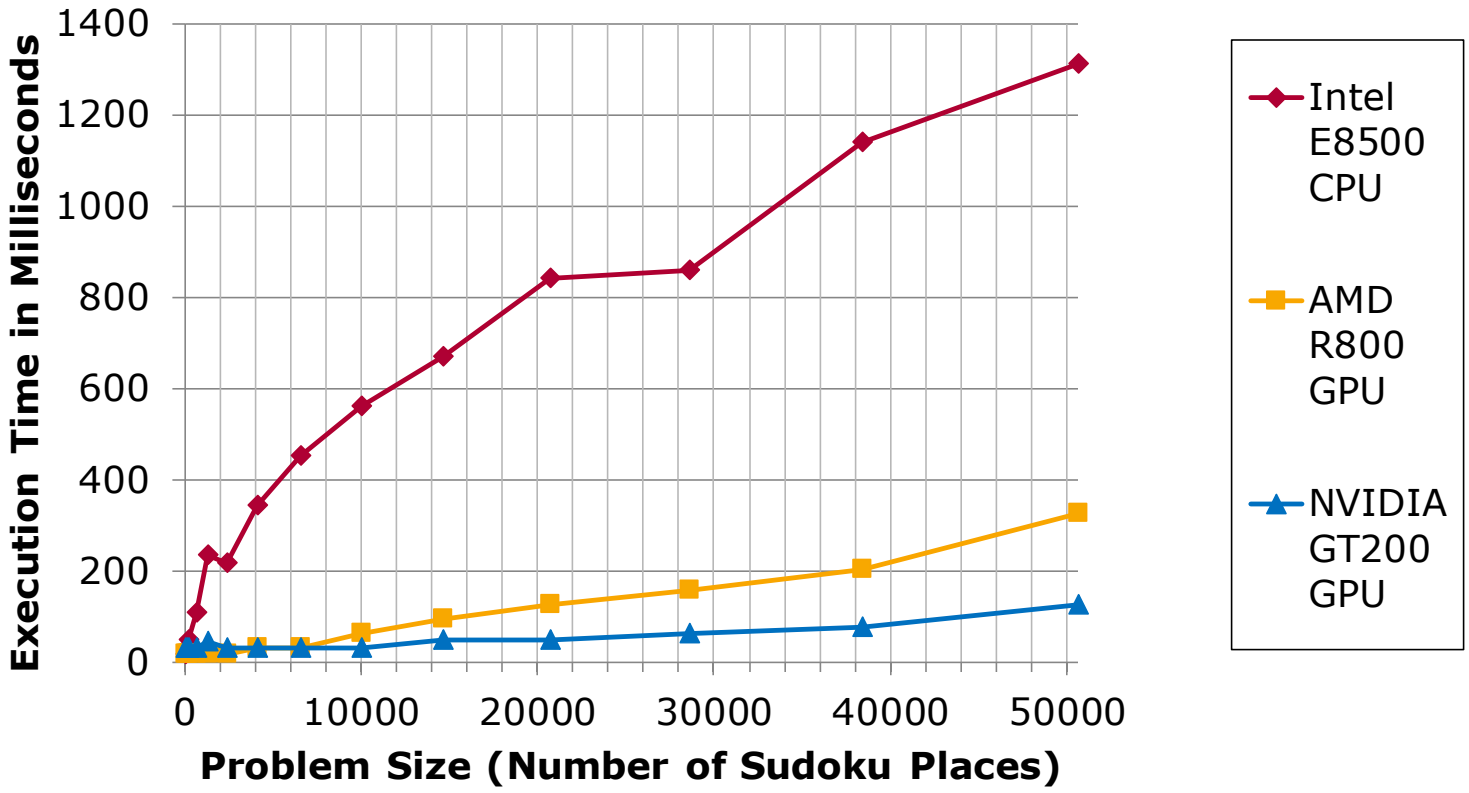
Compute Capability by version

	1.0	1.1	1.2	1.3	2.x	3.0	Since 3.5
double precision floating point operations	No			Yes			
caches	No				Yes		
max # concurrent kernels	1				8		Dynamic Parallelism
max # threads per block	512				1024		
max # Warps per MP	24		32		48	64	
max # Threads per MP	768		1024		1536	2048	
register count (32 bit)	8192		16384		32768	65536	
max shared mem per MP	16KB				16/48KB	*48-112KB*	
# 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

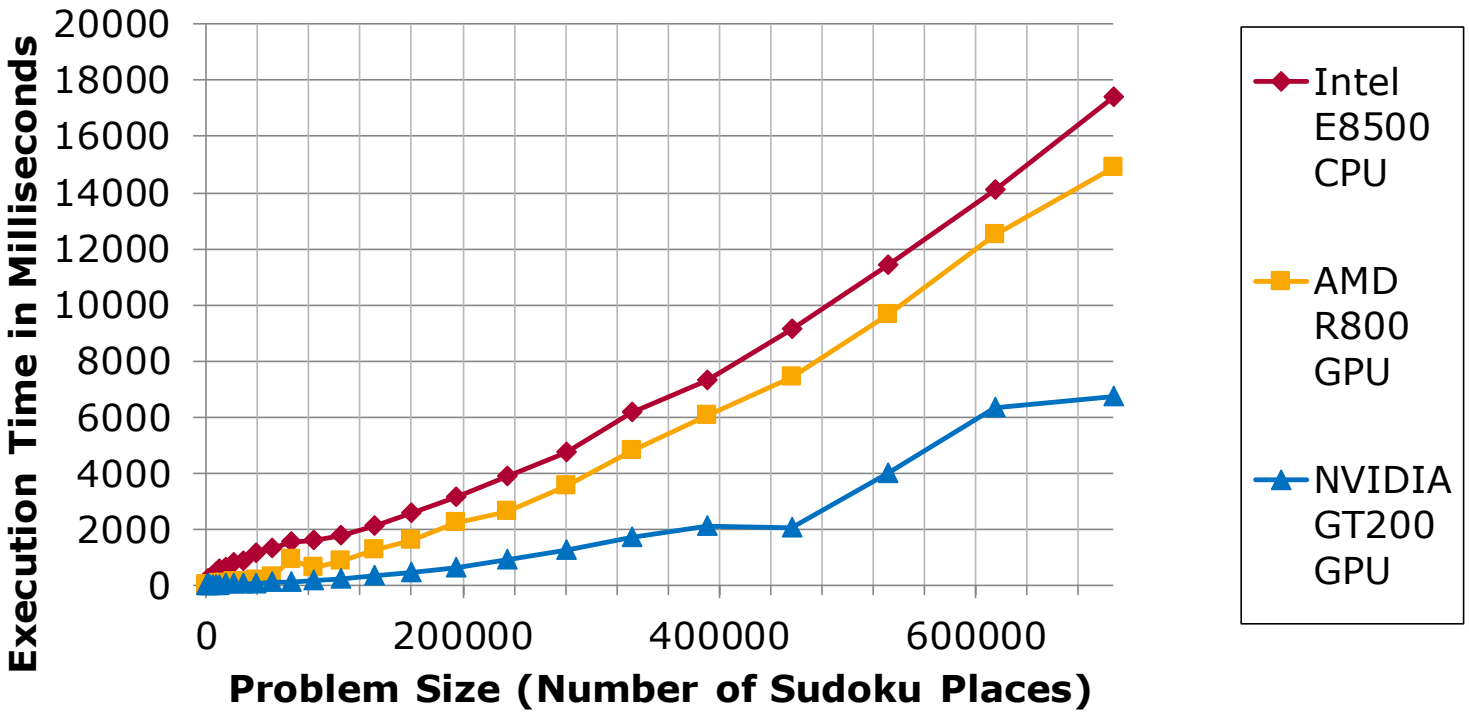


* less is better

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

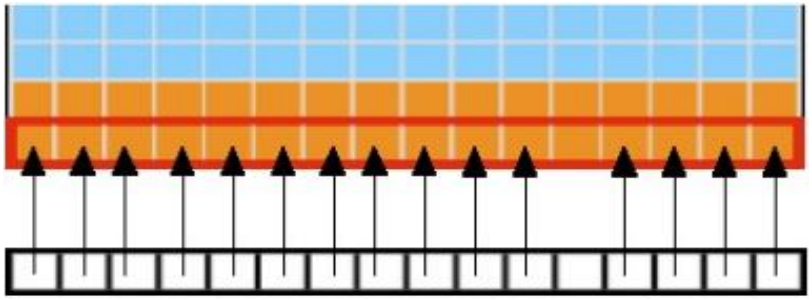
Branch Predication

- 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

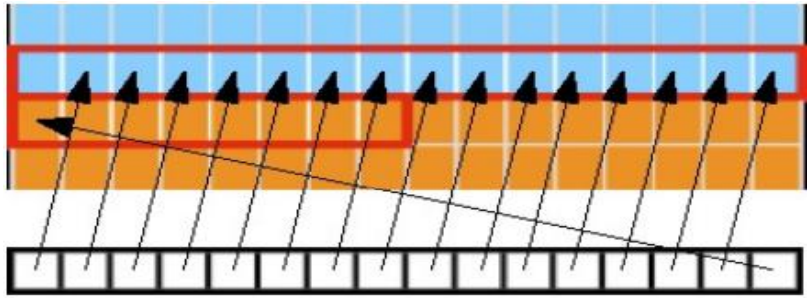
Simple Access Pattern

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



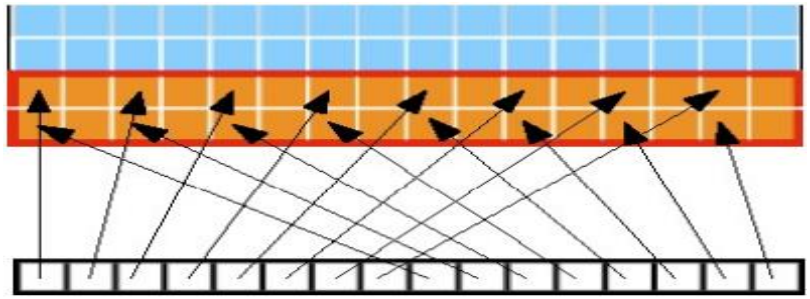
Sequential but Misaligned Access

- 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

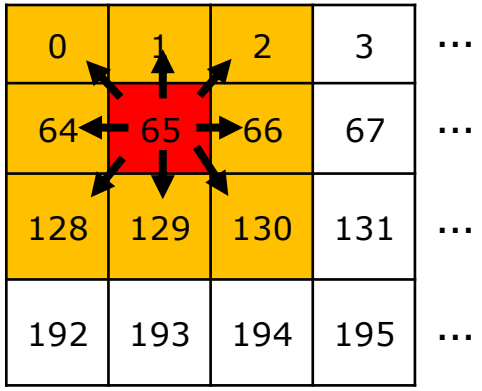
Use Caching: Local, Texture, Constant

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

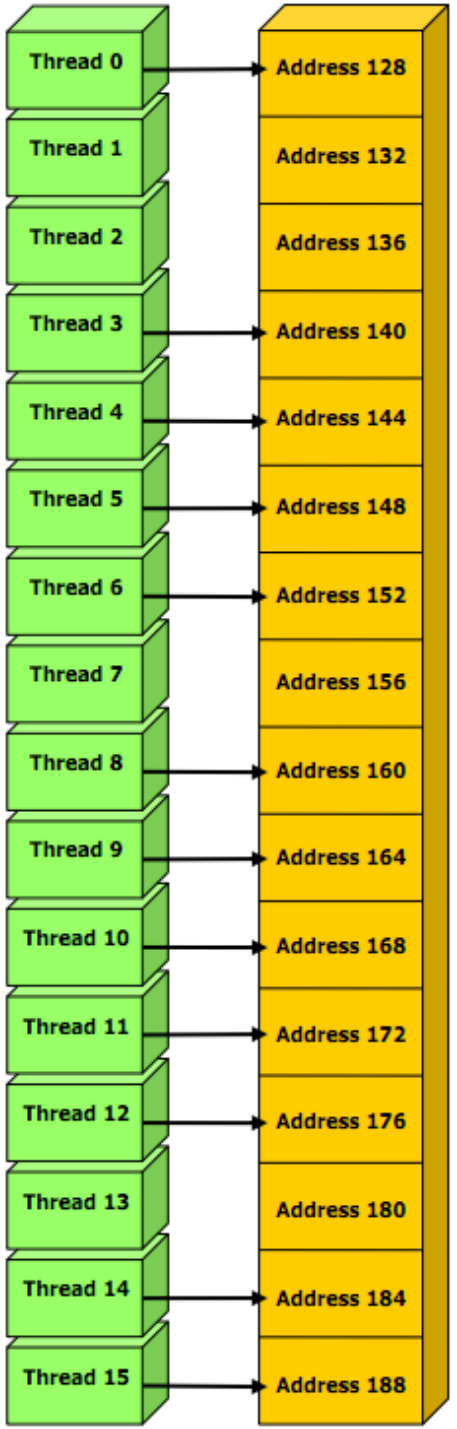


Constant Memory

- 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



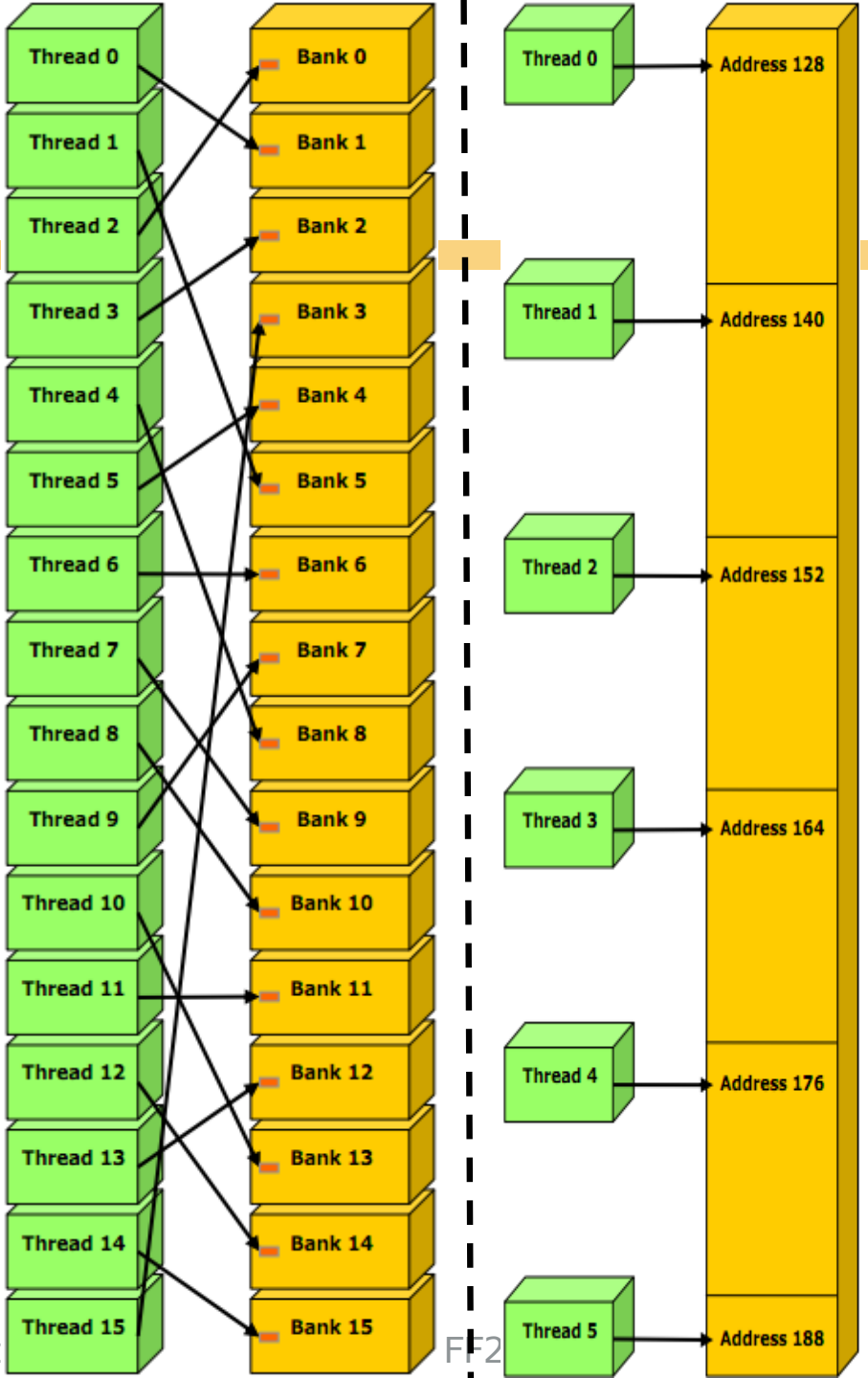
Memory Bank Conflicts

Permuted Memory Access (left)

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

Strided Memory Access (right)

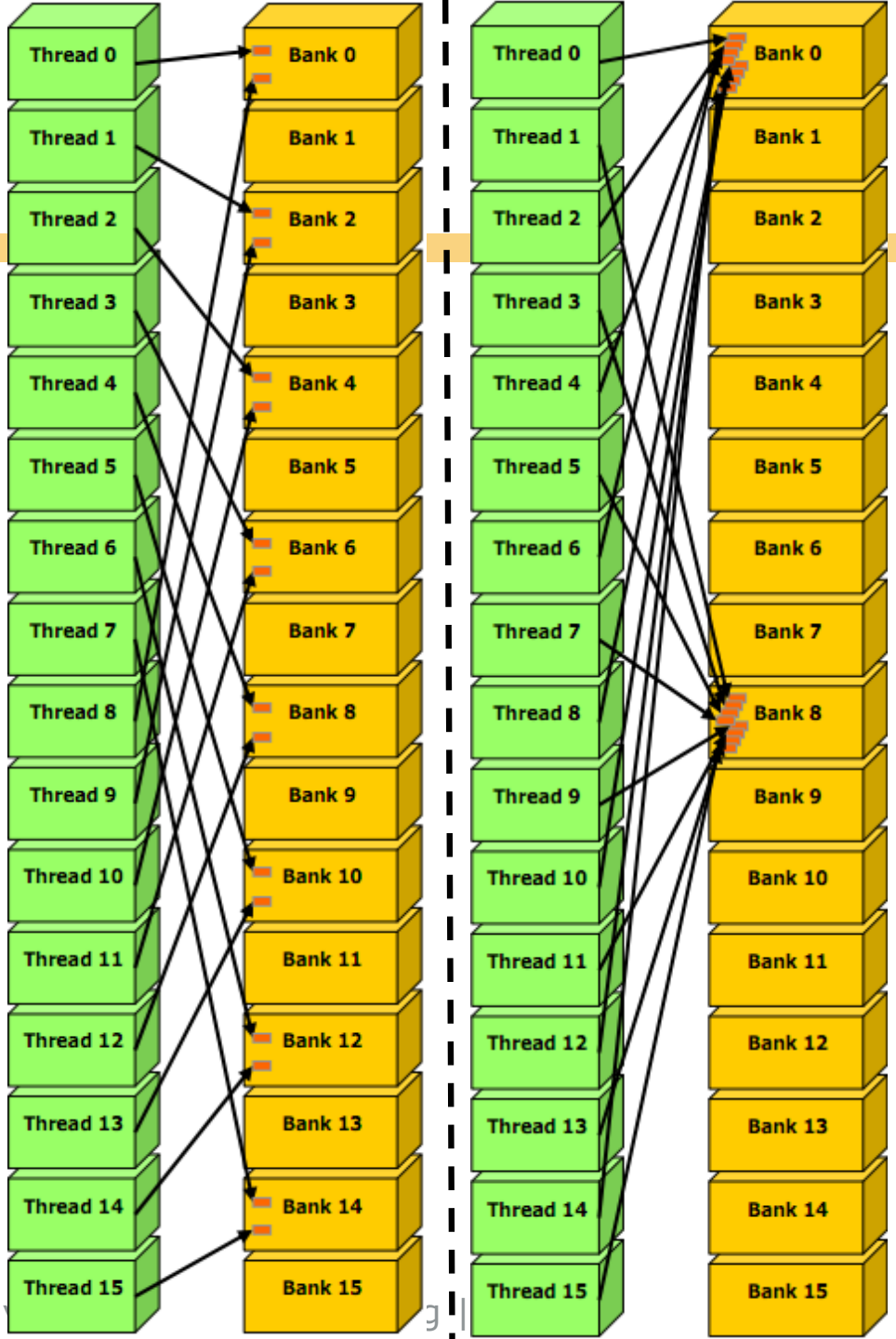
- Still one transaction on cards with compute capability >=1.2; otherwise 16 transactions are required



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

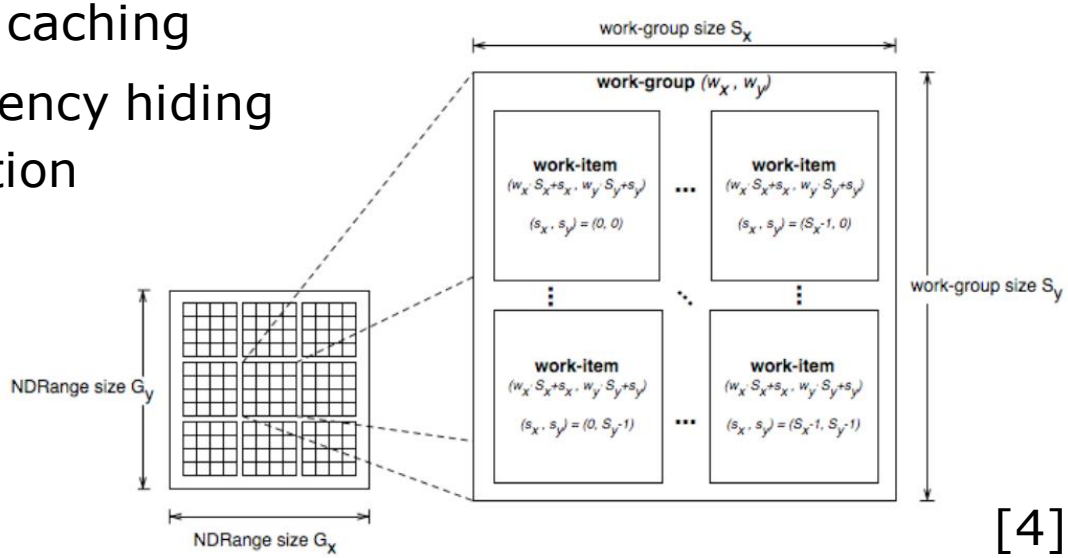


Sizing:
What is the right execution layout?

- 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!



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