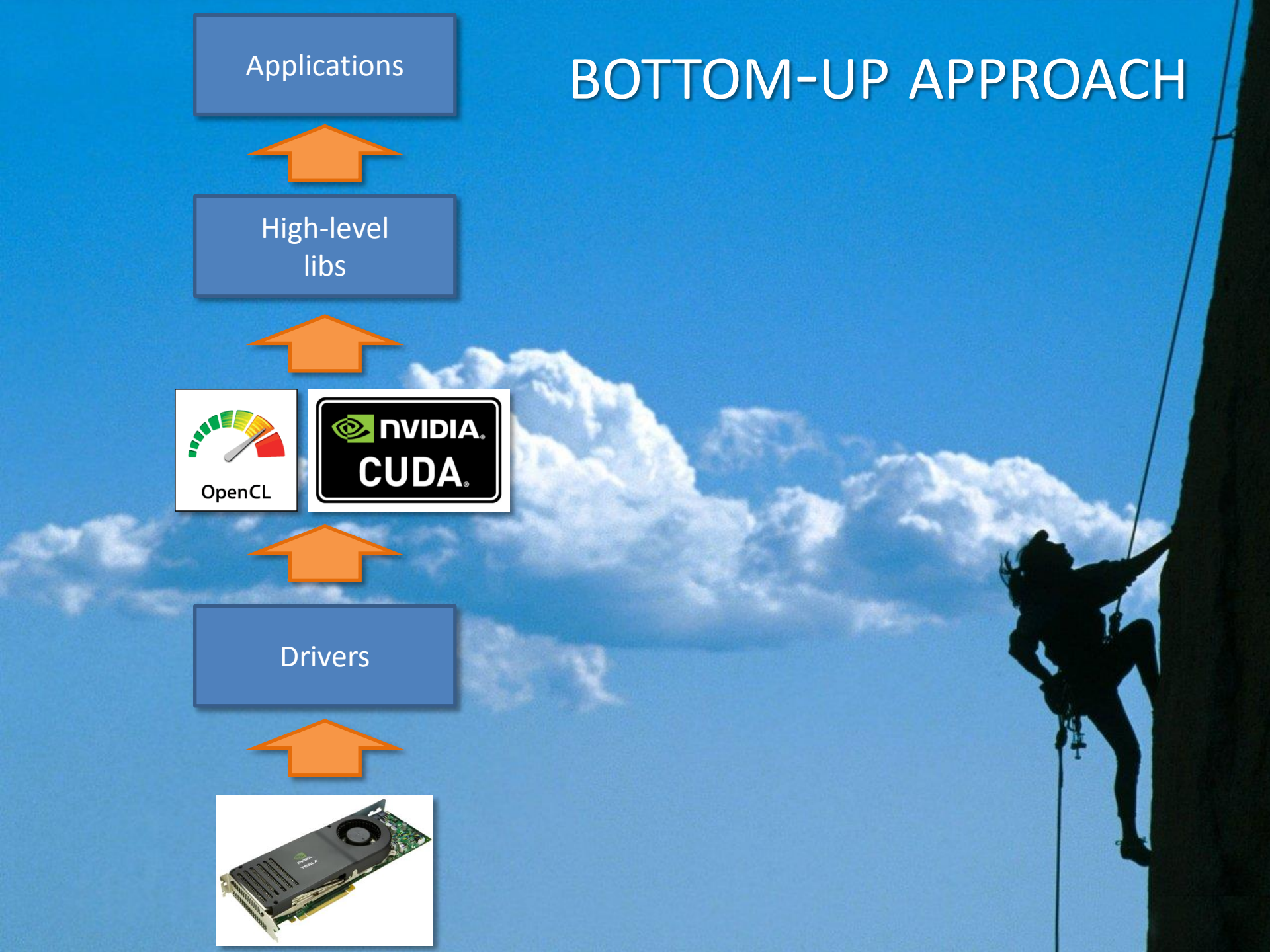
Università della Svizzera italiana
Faculty of Informatics
Institute of Computational Science ICS

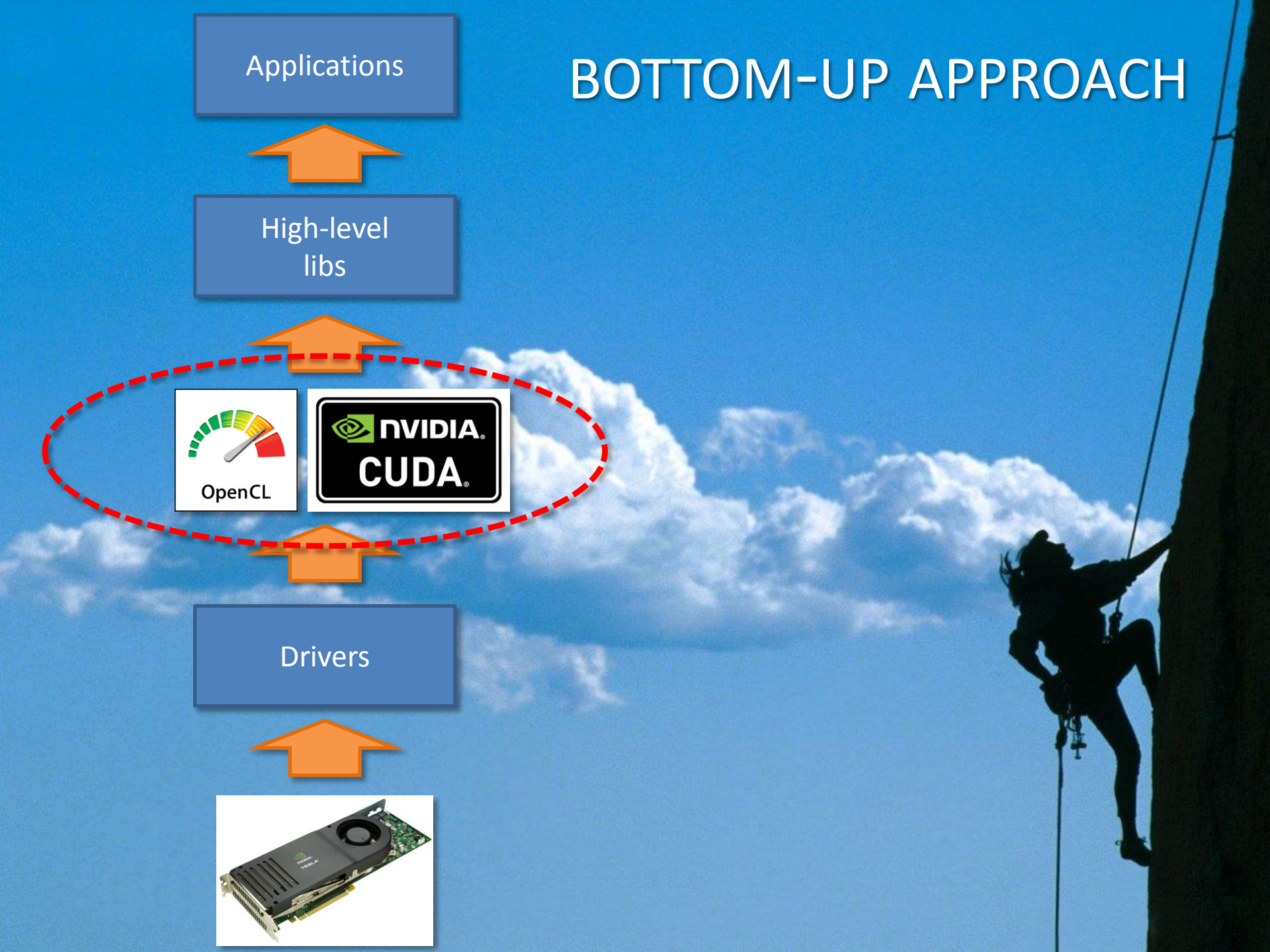
An introduction to OpenCL and CUDA

Achille Peternier,
Dorian Krause,
Rolf Krause

BOTTOM-UP APPROACH

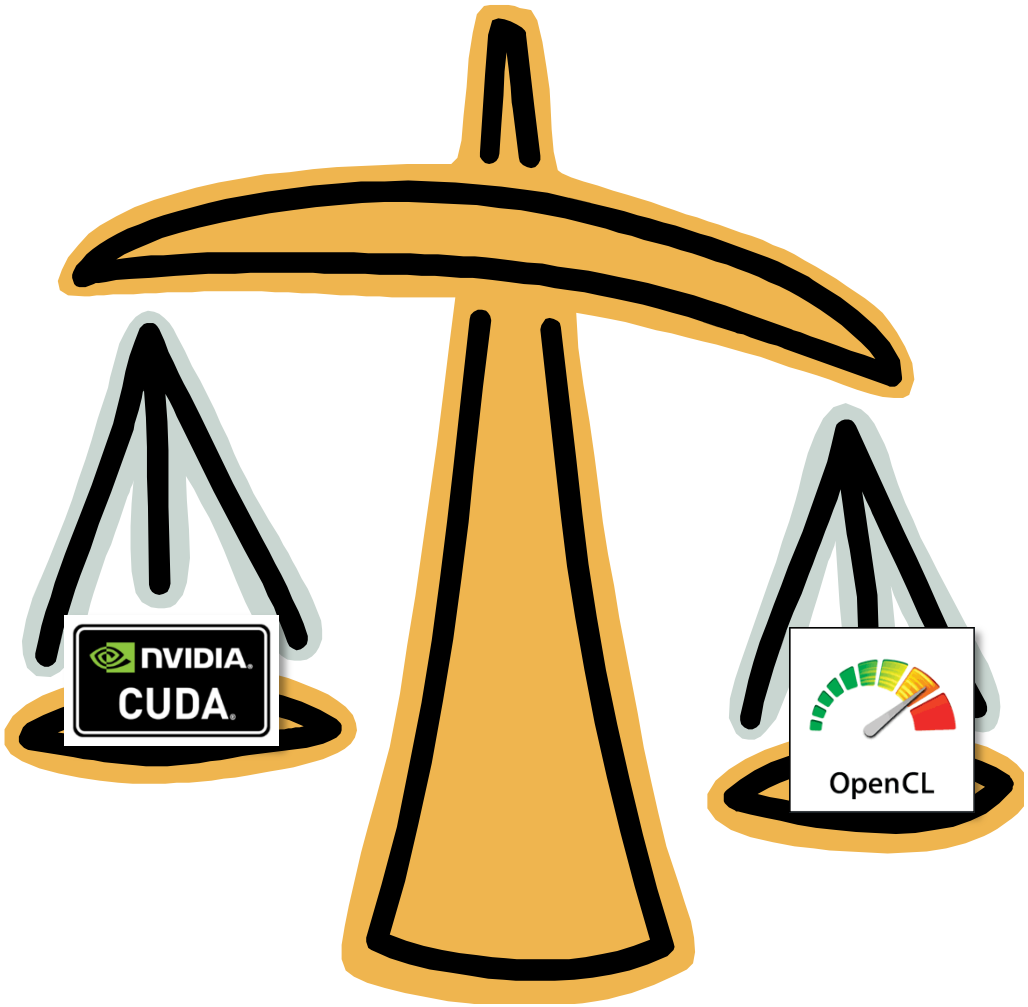


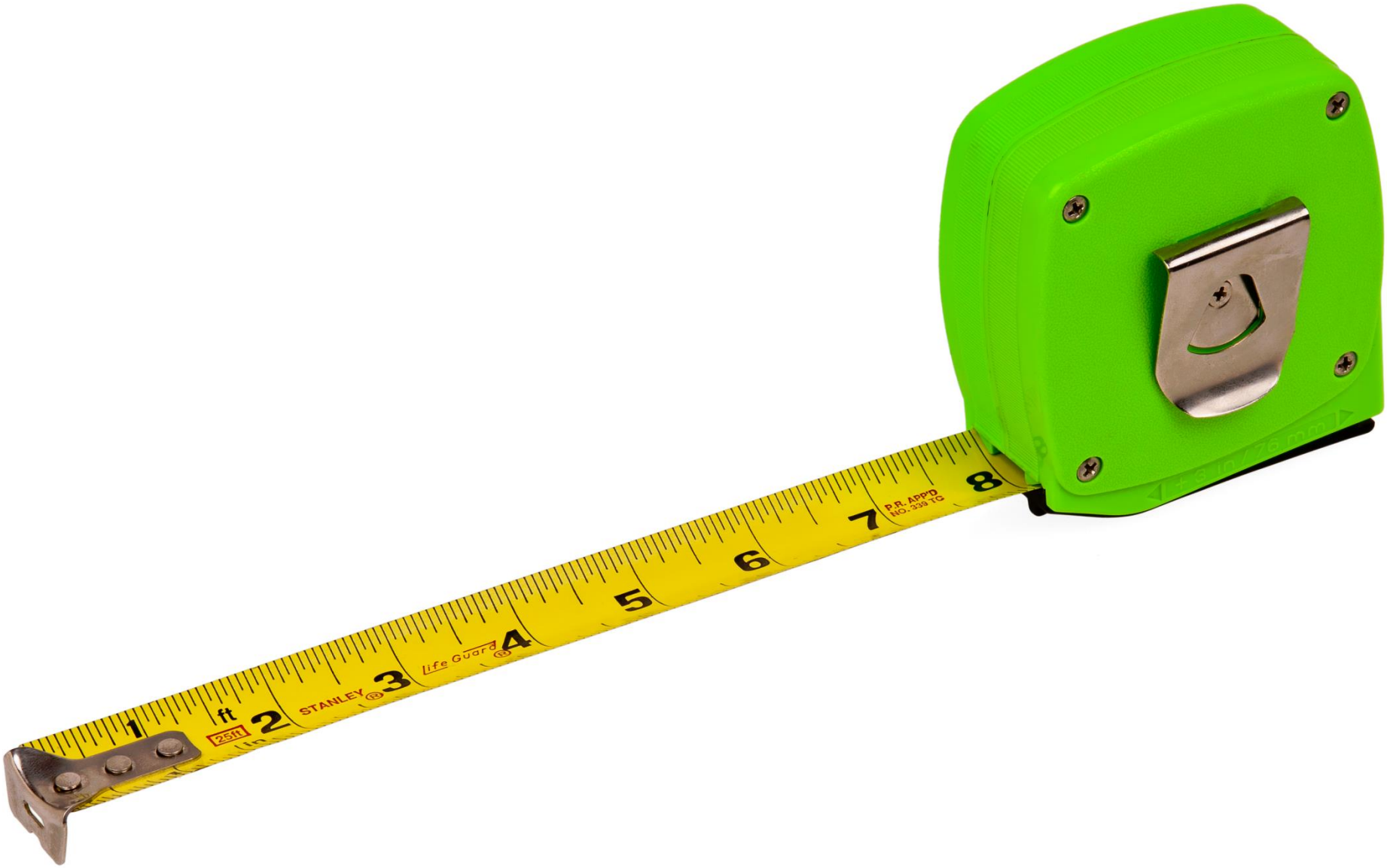
BOTTOM-UP APPROACH



Summary

- Performance measurement
  - exercise #1
- CPU vs. GPU
- OpenCL vs. CUDA
- The CUDA-way
  - exercise #2
- The OpenCL-way
  - exercise #3
- Conclusions
- (maybe) SarXEngine


NVIDIA
CUDA
OpenCL



PERFORMANCE MEASUREMENT

"Today it is impossible to estimate performance: you have to measure it. Programming has become an empirical science."

Performance Anxiety: Performance analysis in the new millennium
Joshua Bloch, Google Inc.

Common pitfalls

- Frequency scaling
  - Power save, turbo mode
  - Governor policies
    - cpufreq-utils, e.g., cpufreq-set -c core# -g policy
- SMT vs. real cores
- NUMA node bindings
- Asynchronous/non-blocking APIs
- Runtime optimizations (ok, that's Java…)

Time APIs

- QueryPerformanceCounter (win)

```
double PCFreq = 0.0;
__int64 CounterStart = 0;

void StartCounter() {
  LARGE_INTEGER li;
  QueryPerformanceFrequency(&li);
  PCFreq = double(li.QuadPart)/1000.0;
  QueryPerformanceCounter(&li);
  CounterStart = li.QuadPart;
}

double GetCounter() {
  LARGE_INTEGER li;
  QueryPerformanceCounter(&li);
  return double(li.QuadPart-CounterStart)/PCFreq;
}
```

- Gettimeoftheday (Linux)

```
timeval t;
gettimeofday(&t, nullptr);
long long int ticks = t.tv_sec * 1000000 + t.tv_usec;
```

RDTSC

- ReaD Time Stamp Counter
  - Returns the number of CPU cycles since reset
- CPU-level counter
  - Low overhead
  - Fine grain
- Pay attention to:
  - Frequency scaling -> disable it
    - Power saving, Turbo mode
  - Simultaneous MultiThreading
  - Out of order execution -> use RDTSCP
    - core i7+, athlon X2+

CUDA/OpenCL timers

- Event-base (notifications)
- Useful for measuring CUDA/OpenCL internals
- Good to synchronize various kernel pipelines

Exercise #1

- A simple timer class
  - Download and compile the cscs_timer example
  - Guess the CPU frequency speed
  - Add a method for computing Fibonacci's sequence
  - Measure the time required for computing each of the first N numbers

Fibonacci's sequence

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34$, etc.

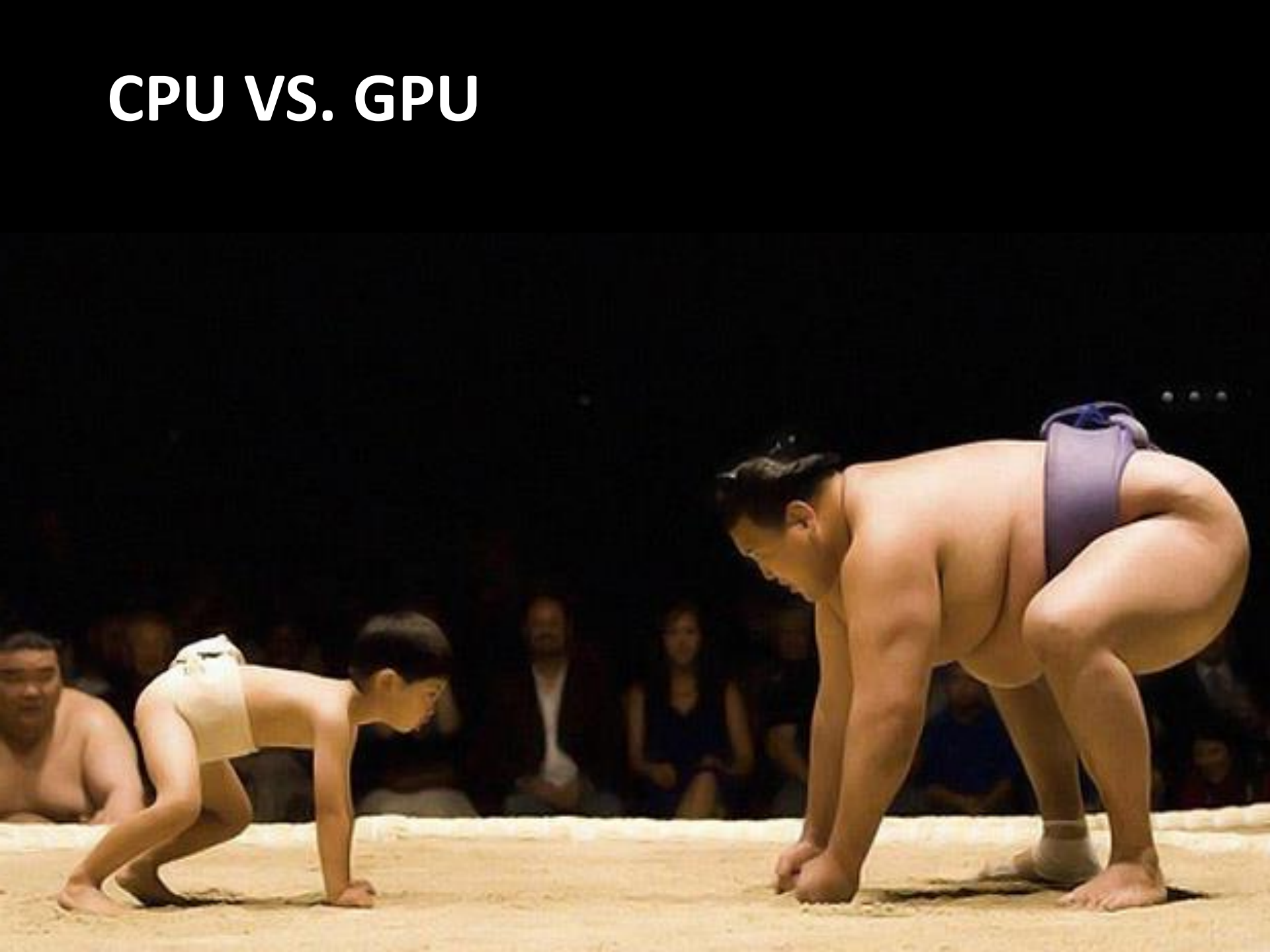
$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0, F_1 = 1$$

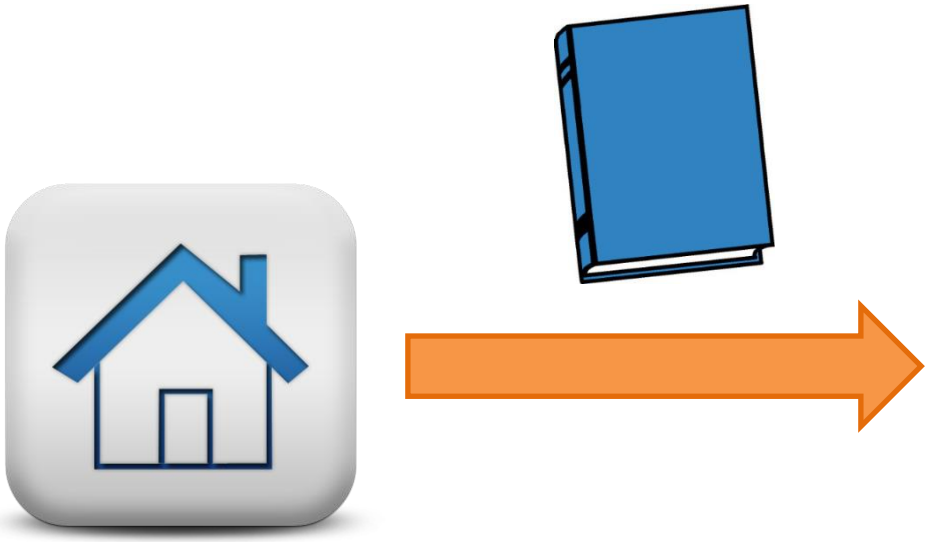
1170 – 1250

```
int Fibonacci(int n)
{
    if (n == 0)
        return 0;
    else
        if (n == 1)
            return 1;
        else
            return (Fibonacci(n-1) + Fibonacci(n-2));
}
```

CPU VS. GPU



Latency vs. throughput



Latency vs. throughput

Optimized for latency

Latency vs. throughput

Latency vs. throughput

Optimized for throughput

Latency VS. throughput

Optimized for both ☺

Latency vs. throughput

CPU

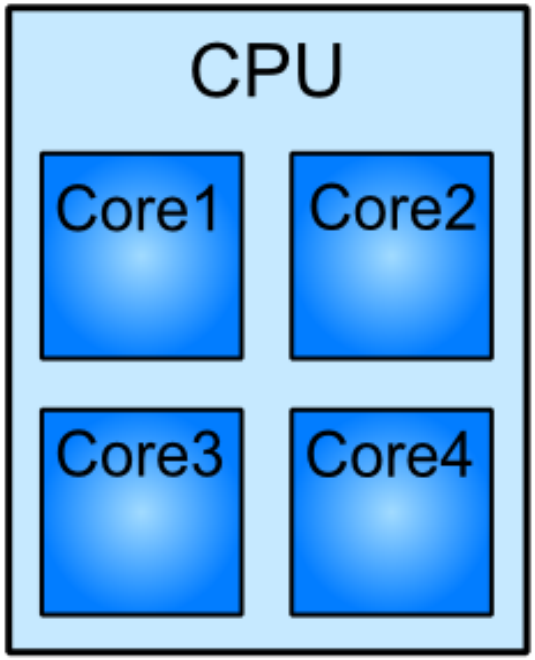
- Optimized for latency
- Out of order
- Multiple cache levels
- Extended instruction set
- Higher frequency
- Limited parallel resources
  - HT
  - SIMD

GPU

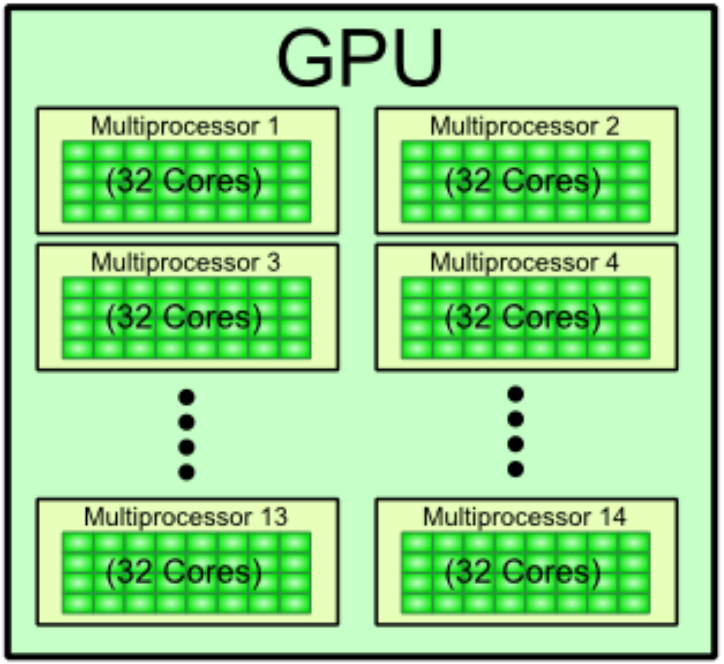
- Optimized for throughput
- No out of order
- Specialized instruction set
- Lower frequency
- Massive parallel architecture
  - SIMD, SIMT

CPU/GPU architectures

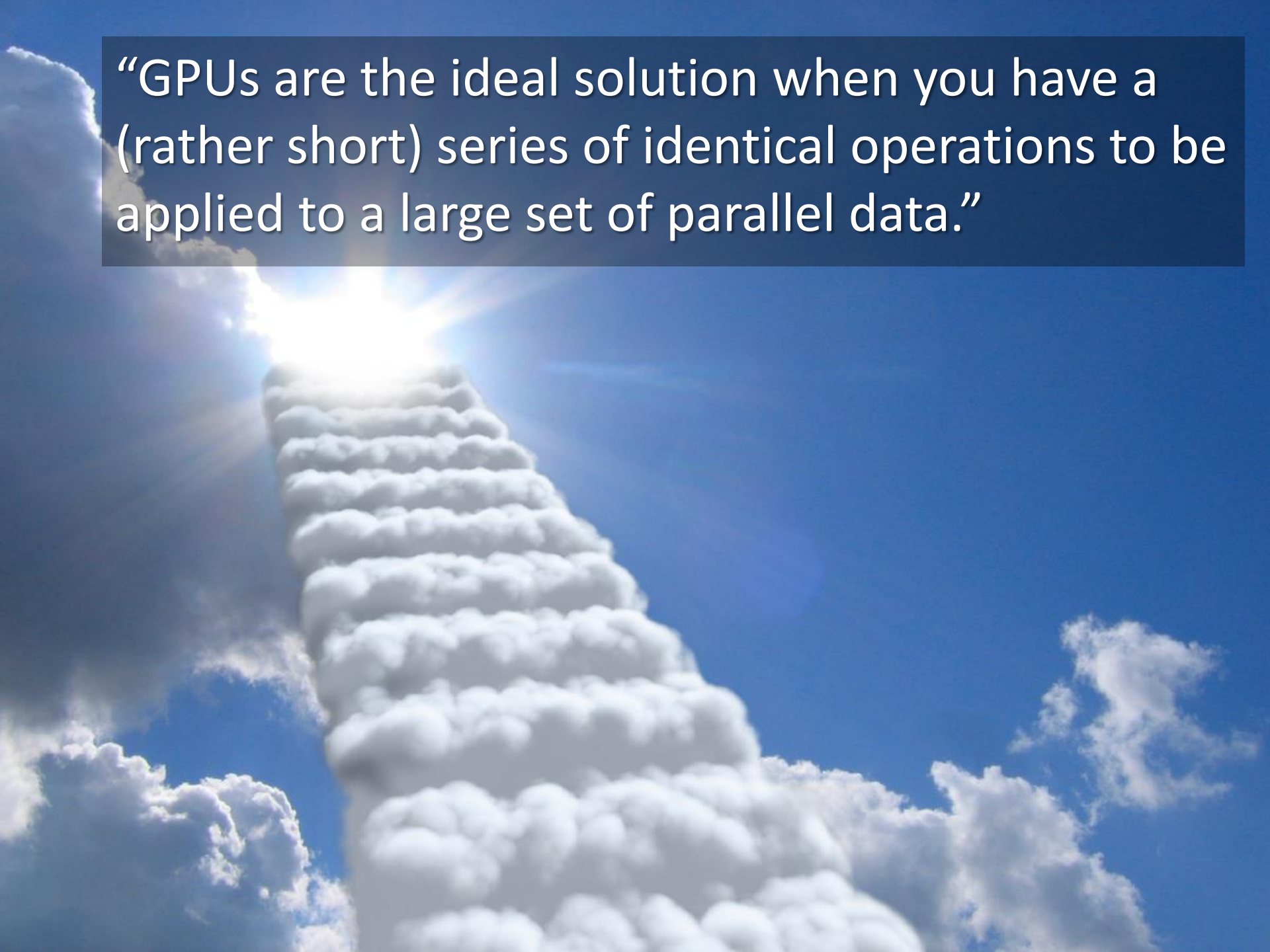
A few sophisticated and fast cores



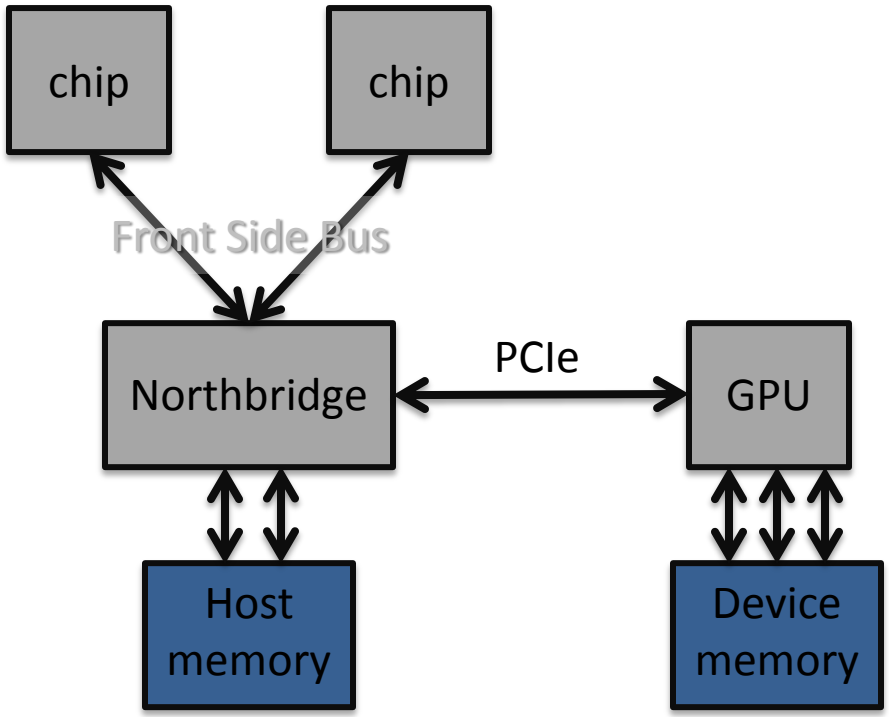
A plethora of simpler and slower cores



(M. Thiesen, Golden Helix)


“GPUs are the ideal solution when you have a (rather short) series of identical operations to be applied to a large set of parallel data.”

Traditional architecture



PCIe

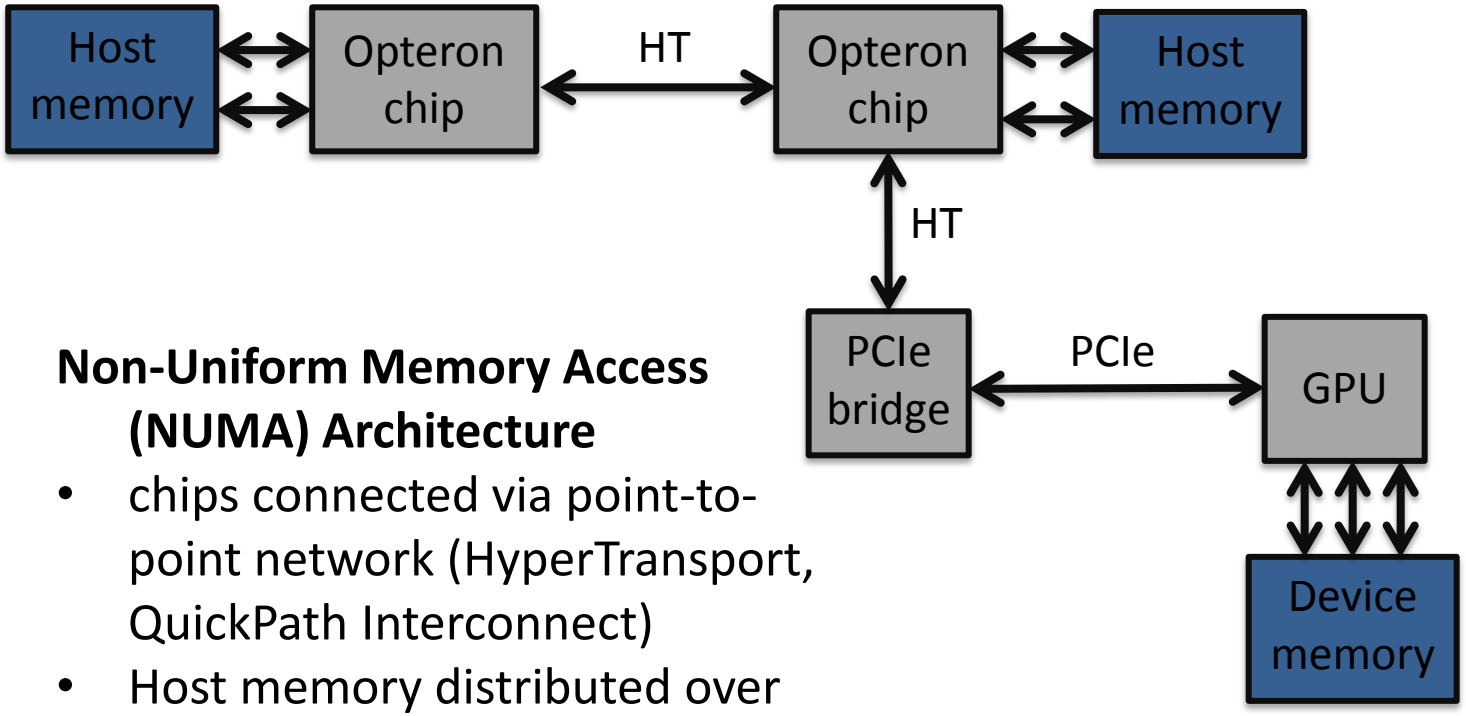
- 200 Mbytes/sec/lane unidir. (Gen1)
- 400 Mbytes/sec/lane unidir. (Gen2)

For PCIe x16 Gen2

- 6.4 Gbytes/sec bandwidth

Data-transfer bottleneck

NUMA architecture



Non-Uniform Memory Access (NUMA) Architecture

- chips connected via point-to-point network (HyperTransport, QuickPath Interconnect)
- Host memory distributed over NUMA nodes

Sustained host-device bandwidth can differ between NUMA nodes

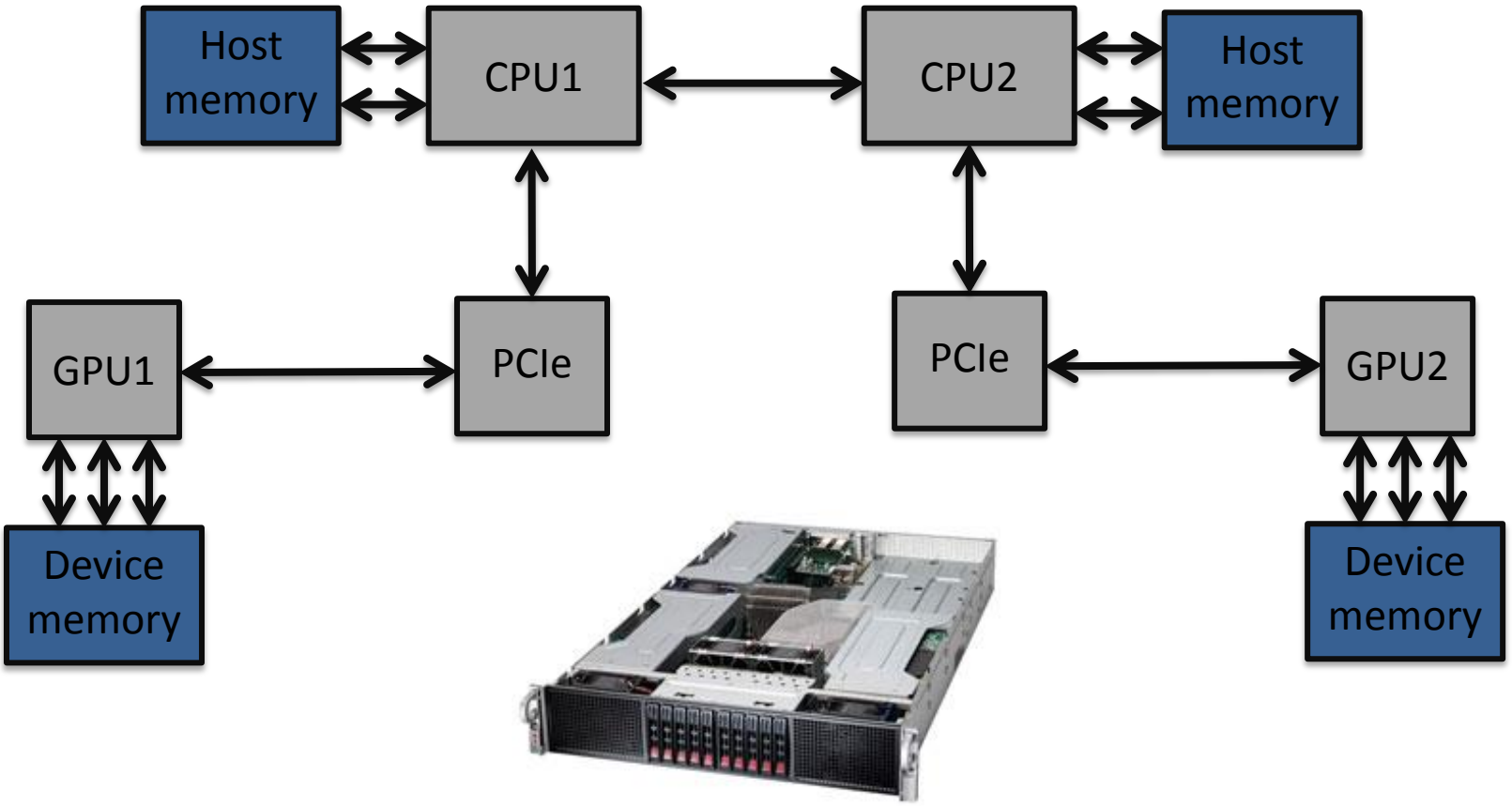
NUMA architecture

- Control it with *numactl*
  - *e.g., numactl –cpubind=0 –membind=0 app.exe*
- 4-CPU Nehalem server:

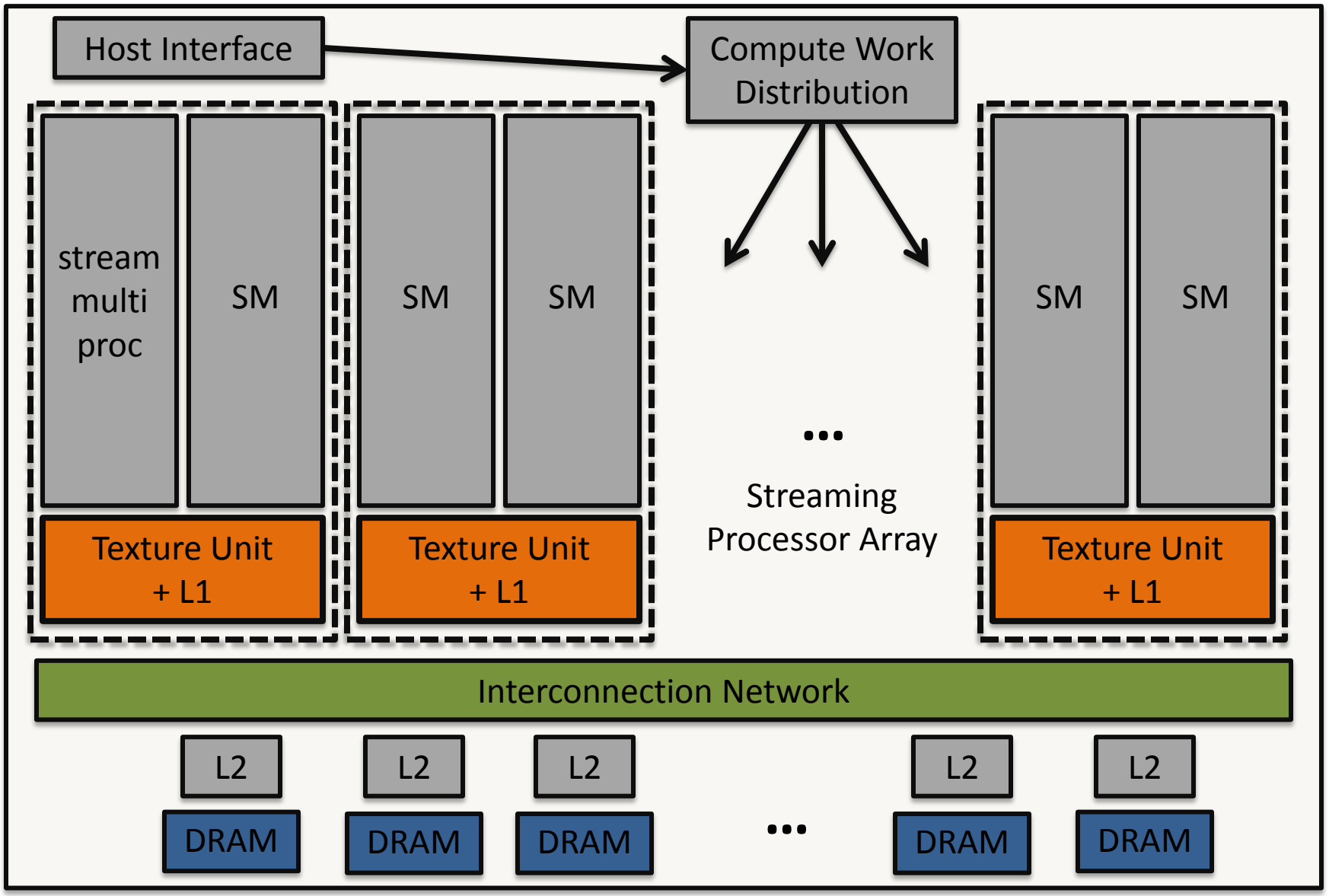
```
peternia@neha:~$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 4 8 12 16 20
node 0 size: 32754 MB
node 0 free: 18844 MB
node 1 cpus: 1 5 9 13 17 21
node 1 size: 32768 MB
node 1 free: 28617 MB
node 2 cpus: 2 6 10 14 18 22
node 2 size: 32768 MB
node 2 free: 26706 MB
node 3 cpus: 3 7 11 15 19 23
node 3 size: 32768 MB
node 3 free: 29788 MB
node distances:
node   0   1   2   3
  0:  10  20  20  20
  1:  20  10  20  20
  2:  20  20  10  20
  3:  20  20  20  10
```

NUMA architecture

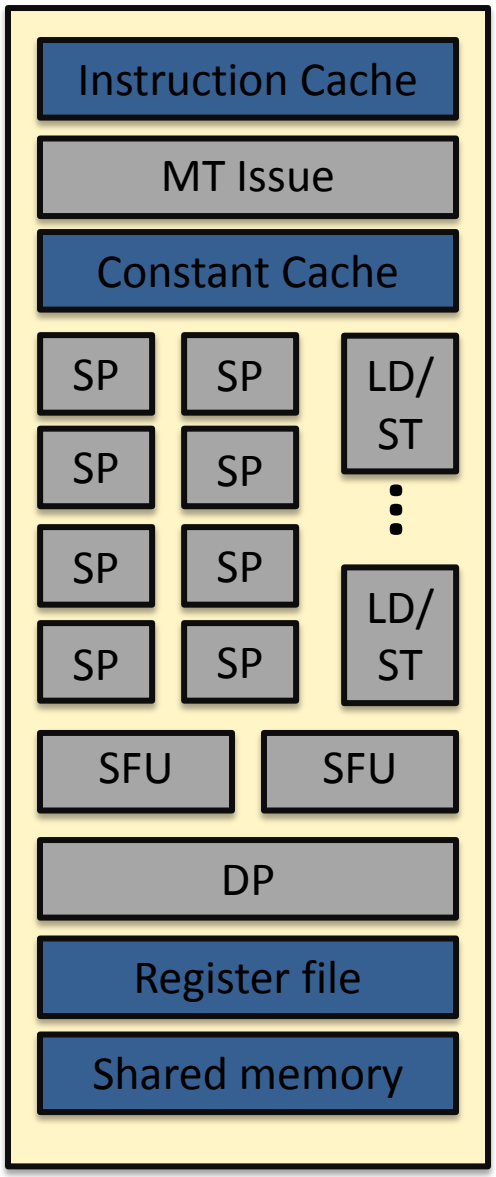
- It works also on multi-GPU systems!



Tesla architecture



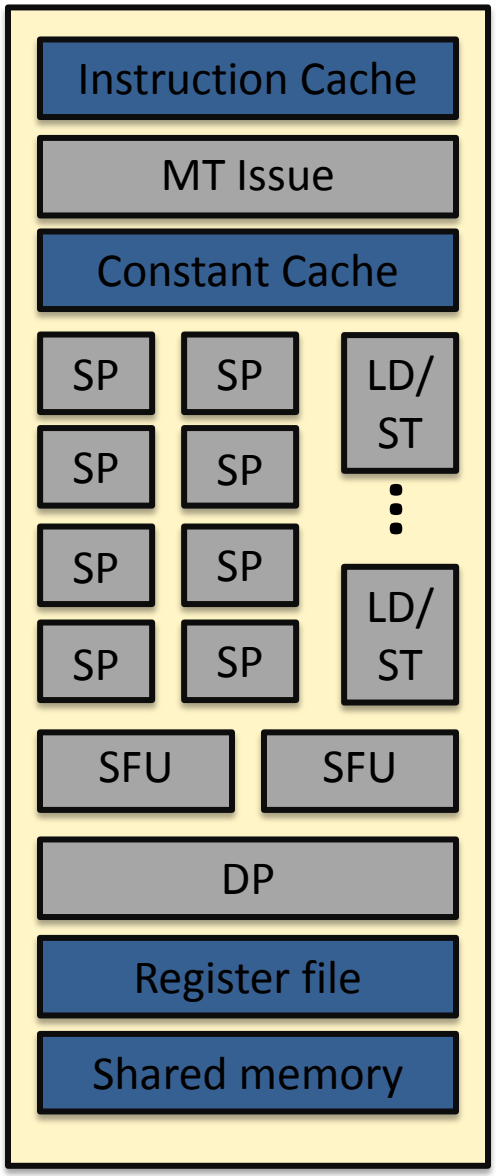
Tesla architecture



Stream Multiprocessor (SM)

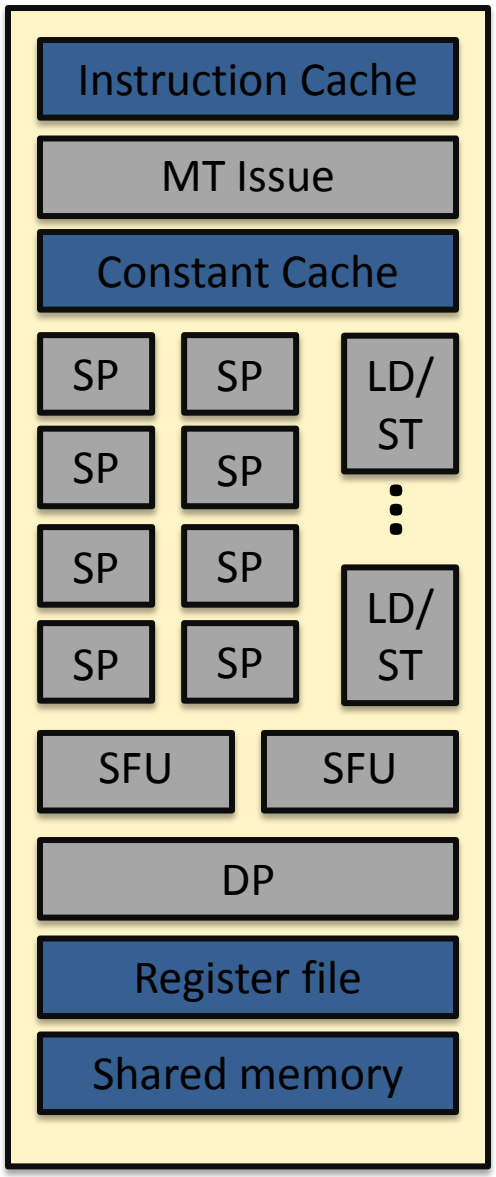
- **8 SP** (streaming processor or CUDA core): 32-bit MAD pipelines
- **2 SFU** (special function units)
  Compute transcendental functions (sin, cos, …) with a throughput of one instruction per clock
  Each SFU contains four floating-point multipliers
- **1 DP** (double precision unit, GT200 only)
- 16 Kbytes **shared memory**
- 32 Kbytes (G80) – 64 Kbytes (GT200) **register file**
- Read-only **constant cache**
- LD/ST instructions & texture fetches can be overlapped with SP instructions
- Compute units clocked at 1 – 1.5 Ghz

Tesla architecture



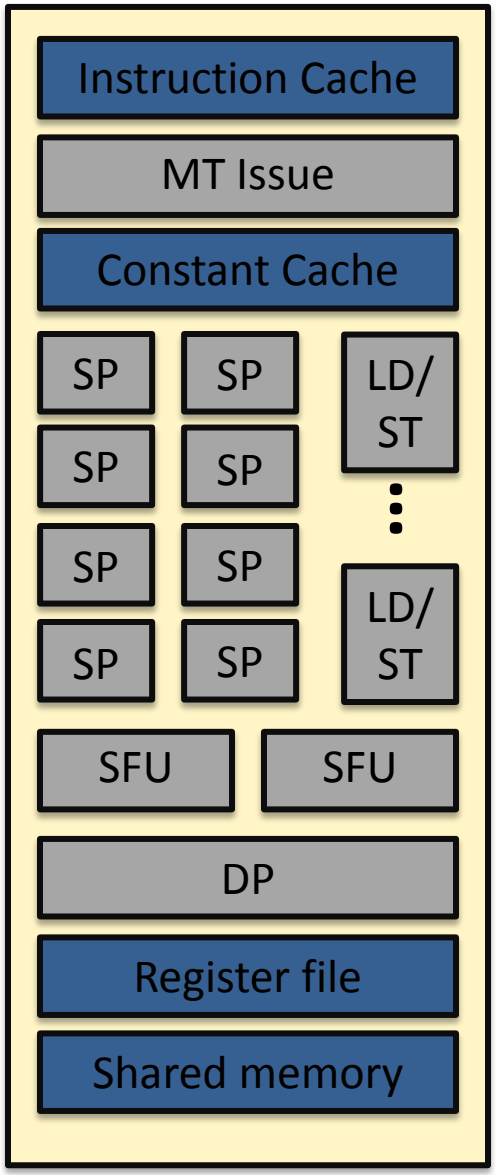
- Hardware multithreaded
  - ✓ Up to 768 concurrent threads
  - ✓ Each with own state and execution path
  - ✓ Lightweight scheduling and fast barrier synchronization
- **SIMT** (Single Instruction Multiple Thread) execution model
  - ✓ Threads managed and scheduled in groups of 32 (**warp**)
  - ✓ SIMT instruction broadcasted synchronously to a warp's active threads; Threads can be deactivated due to branching/predication

Tesla architecture



- **SIMT** (Single Instruction Multiple Thread) execution model
  - ✓ Fully efficient if all threads in warp take same execution path
  - ✓ If threads diverge, each path is taken serially
  - ✓ Different warps execute independently
- Compared to **SIMD**: Threads with branching capability vs. lanes
  - ✓ Branching handled at the hardware level, not exposed to software
  - ✓ SIMT can be (usually) ignored for correctness but is important for performance

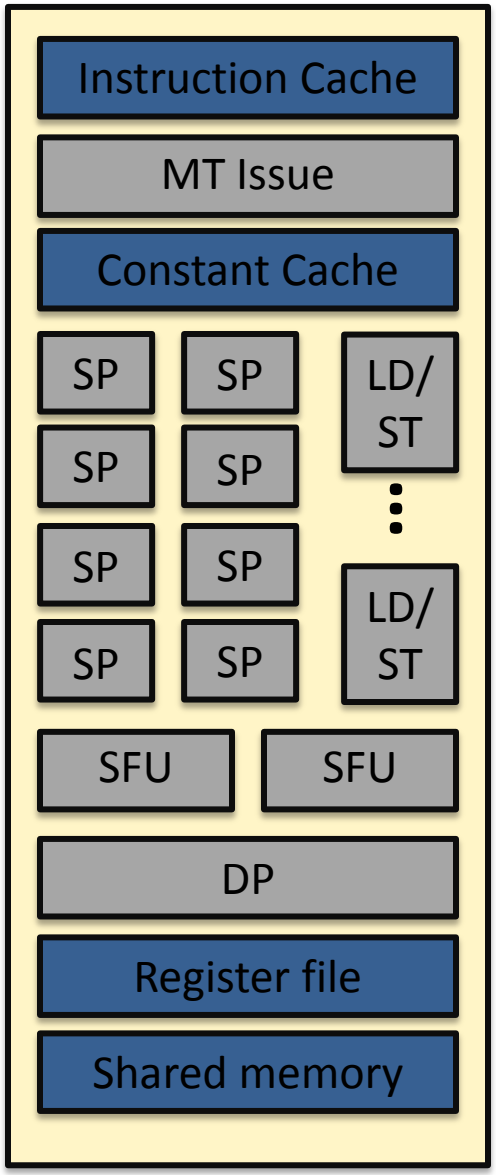
Tesla architecture



SIMT Warp Scheduling

- Warp scheduler operates at half the SP and SFU clock rate
- At each cycle one of the 24 warps is selected for execution of a SIMT warp instruction
- "An issued warp instruction executes as two sets of 16 threads over four processor cycles" *[Lindholm et al.]*
- SP and SFU execute independently and hence can be fed by the scheduler on alternating cycles

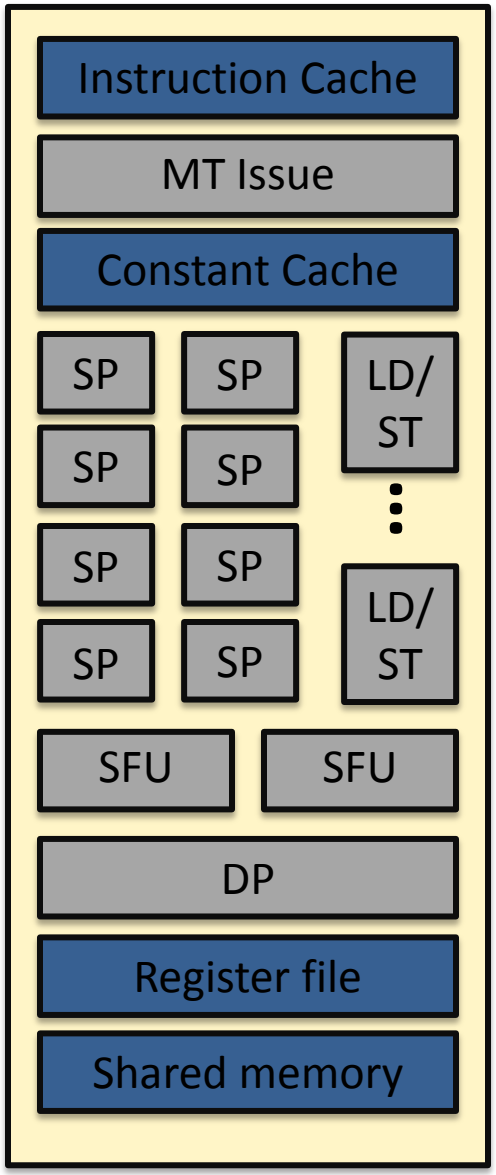
Tesla architecture



SIMT Warp Scheduling

- Throughput
  - ✓ 4 clocks per instruction for SPs
  - ✓ 16 clocks per instruction for SFUs
  - ✓ 32 clocks per instruction for DP
- MAD pipeline latency 24 clocks *[Volkov & Demmel]*
- If operand is in shared memory, MAD throughput is reduced by ~40% (6 clocks per instruction) *[Volkov & Demmel]*

Tesla architecture



Shared memory

- Connected to SPs via interconnected
- Organized into 16 interleaved banks
- Latency 36 – 38 cycles *[Wong]*

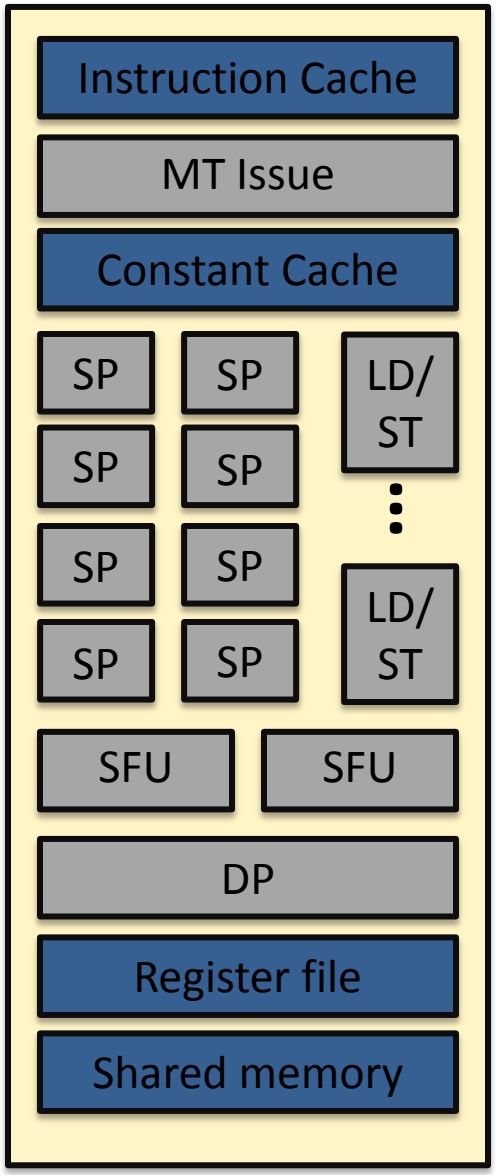
Constant memory

- Read-only memory in DRAM cached on the SM

Texture memory

- Read-only memory, cached, deeply pipelined (many misses and hits in flight)
- Designed to reduce DRAM bandwidth demand not latency

Tesla architecture



Global memory

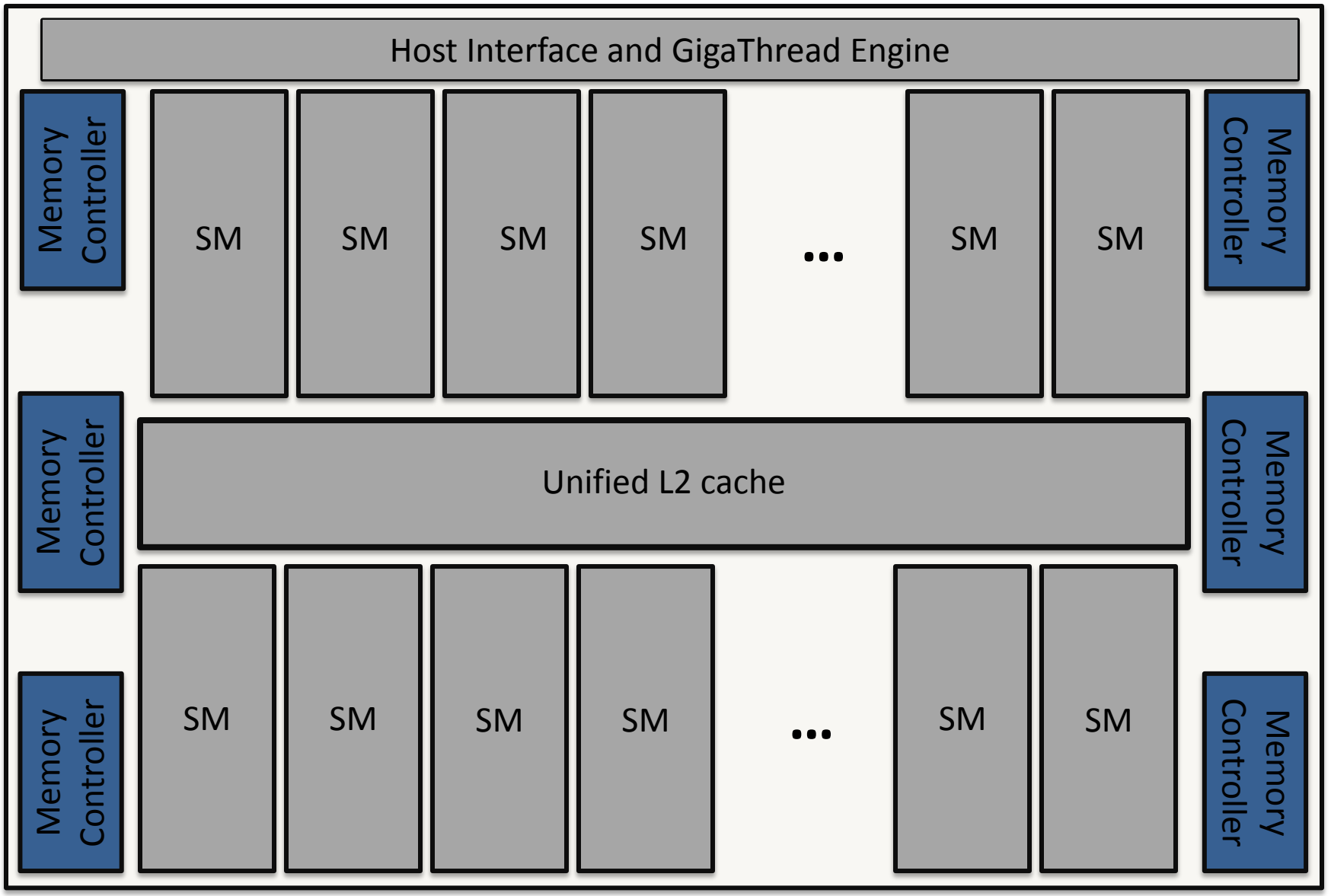
- 384 – 512 pins in 6 – 8 interleaved partitions of 64 pins each
  GDDR3 memory

  E.g. C1060: 512 pins, 8 partitions, GDDR3 @ 800 Mhz

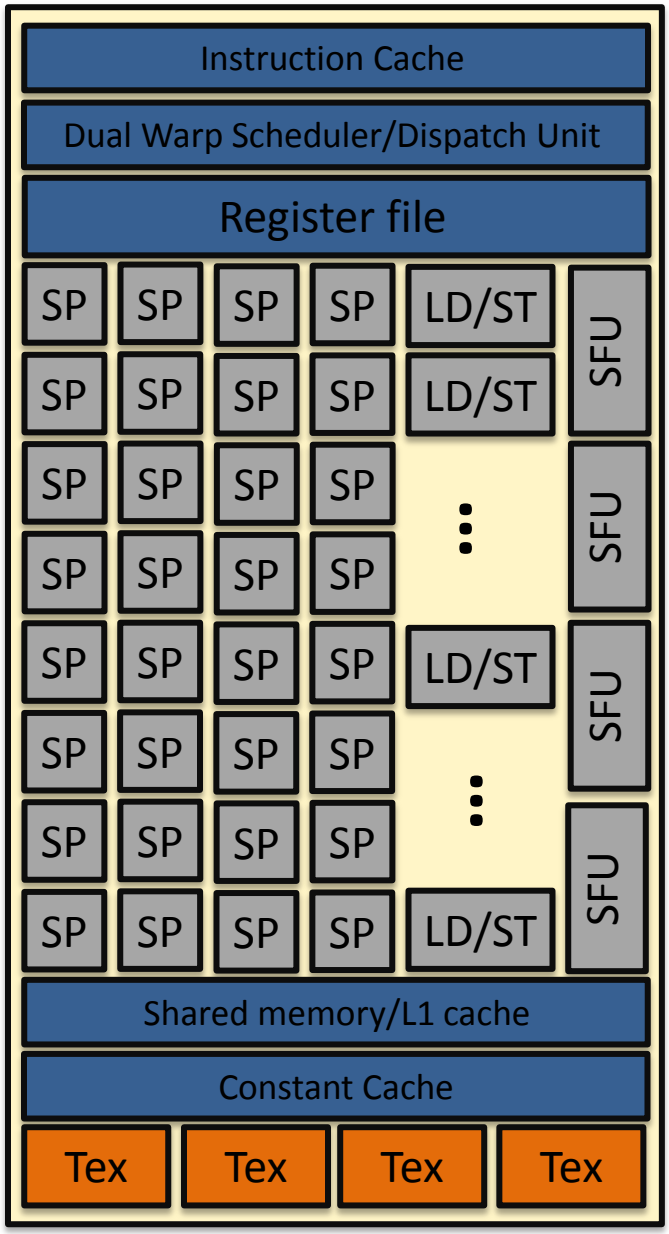
⇨ 0.8 × 2 × 512 / 8 = 102.4 Gbytes/sec

- Memory request coalescing important to “order” seemingly uncorrelated requests for DRAM
- Virtual memory addressing (32-bit), TLB

Fermi architecture

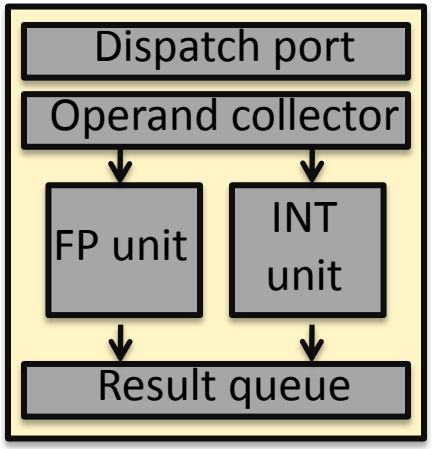


Fermi architecture



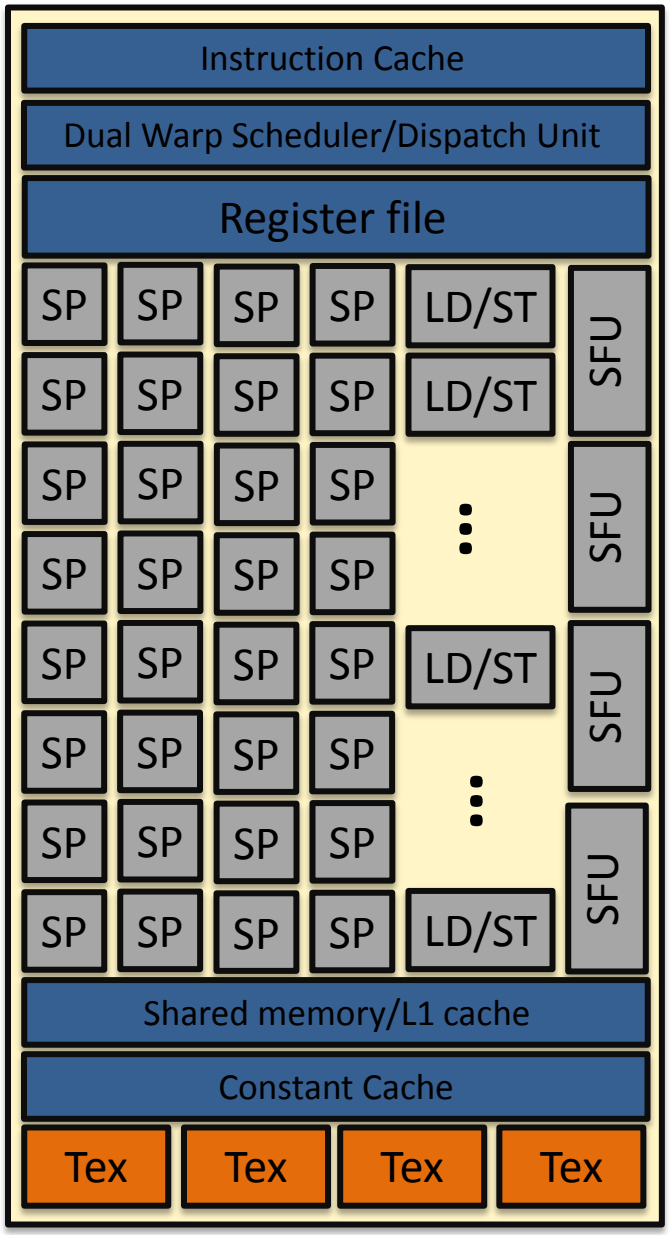
Stream Multiprocessor (SM)

- **32 SPs** (4× over Tesla)
  - ✓ Enhanced design



  - ✓ 32 bit integer ALU
  - ✓ Fused multiply-add
  - ✓ INT and FP unit execute in parallel
  - ✓ Full double precision support with 8 clocks per instruction for FMAD

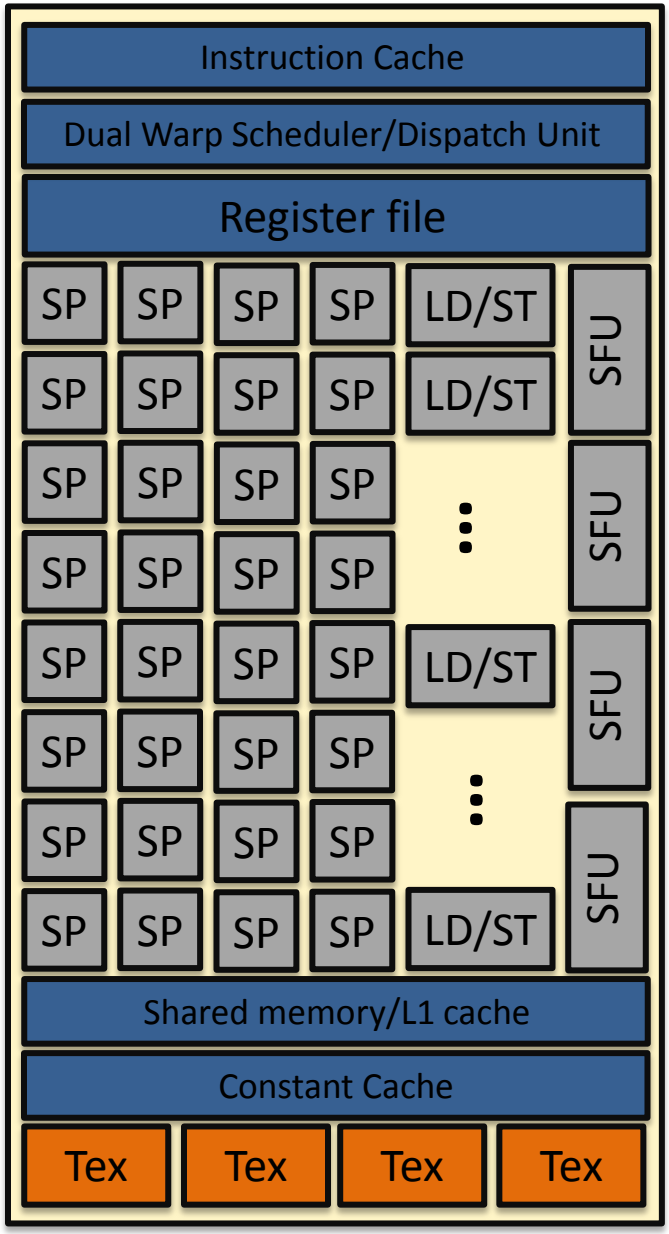
Fermi architecture



Stream Multiprocessor (SM)

- **16 LSUs** (Load Store Unit)
  - ✓ 2 clocks per instruction for computation of source and destination addresses
- Dual warp issue
  - ✓ Increased IPC for branchy code or code mixing integer ops and flops
- 4 texture units
- Configurable shared memory or L1 cache (16/48 Kbytes)

Fermi architecture



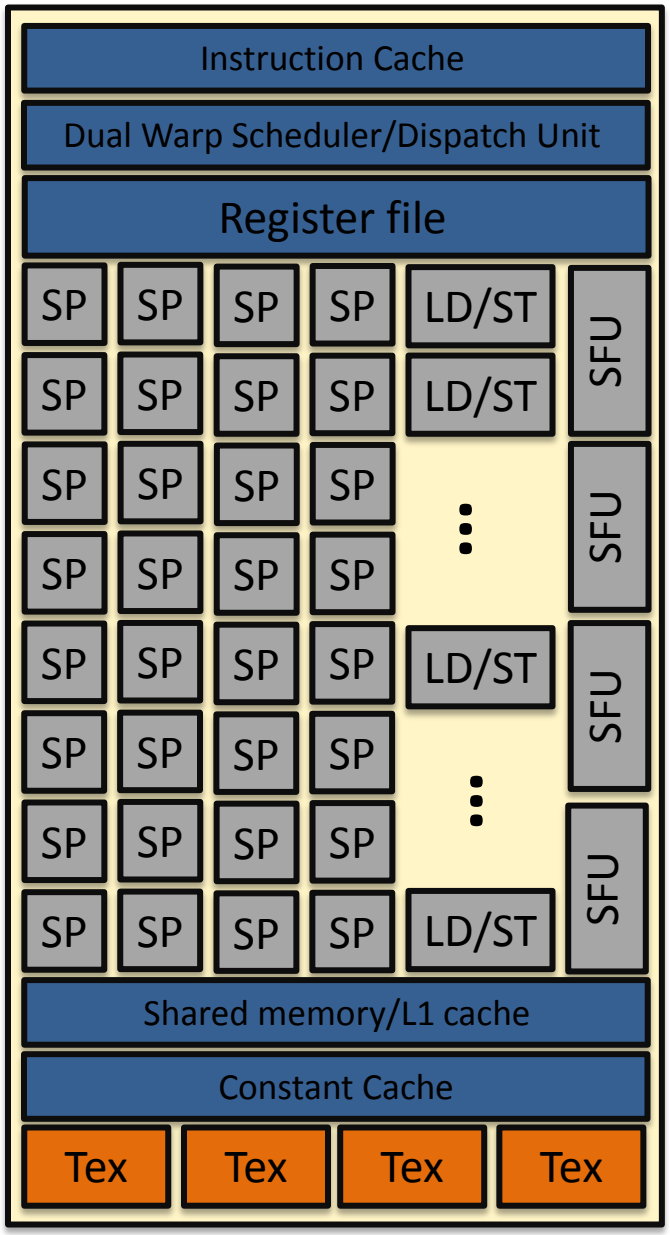
Stream Multiprocessor (SM)

- Shared Memory
  - ✓ 32 banks (2× over Tesla)
  - ✓ Improved handling of reads from same bank (multiple words are broadcasted)

Caches

- L1 cache per SM
  - ✓ useful for register spills, stack operations and caching of DRAM loads
  - ✓ Configurable size (16 Kbytes or 48 Kbytes)
- Unified read/write L2 cache
  - ✓ 768 Kbytes
  - ✓ 128 bytes lines

Fermi architecture



Other Features

- ECC support
- Single shared address space (instead of separate **local**, **shared** and **global**)
- Concurrent kernel execution
- Faster (and more) atomic integer operations for global synchronization
- 40-bit address space
- GDDR5 support

E.g. C2070: GDDR5 @ 1.5 Ghz, 384 pins

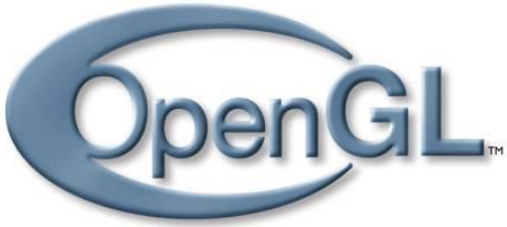
⇨ 1.5 × 2 × 384 / 8 = 144 Gbytes/sec

Kepler architecture

Kepler GK110 Full chip block diagram

OpenCL vs. CUDA

History



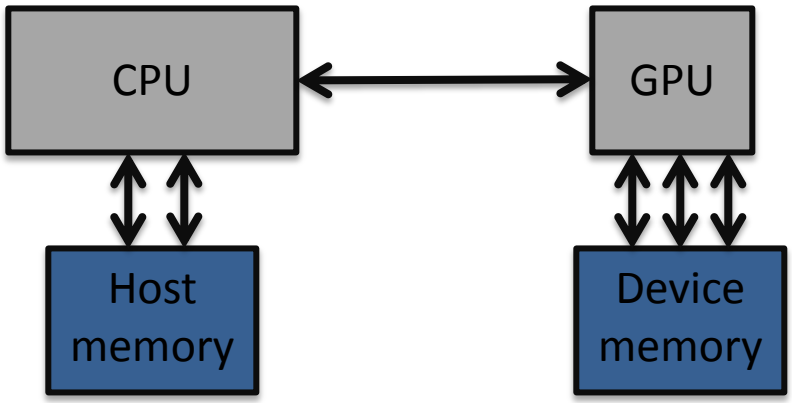
- Encoding generic data into textures
- Storing 32bit pointers into RGBA values
- Using graphic shaders as kernels

- Proper and clean API for accessing GPU HW

- Generalization of the idea

OpenCL/CUDA

- Initialize context and device(s)
- Allocate memory on device
- Copy data from host to device
- Execute kernel on device
- Copy data back to host
- Free memory on device
- Release context



Ecosystems

OpenCL

NVIDIA AMD intel IBM

NVIDIA CUDA

NVIDIA

Find & Replace…

OpenCL	CUDA
Device	GPU
Compute Unit	Multiprocessor
Processing element	Scalar core
Global memory	Global memory
Local memory	Shared (per-block) memory
Private memory	Local memory (automatic, or local)
Program	Kernel
Work-group	Block
Work item	Thread
cl* API	cu* API

(Matt Harvey, "Experiences porting from CUDA to OpenCL")

Comparison

	CUDA	OpenCL
What it is	HW architecture, ISA, programming language, API, SDK and tools	Open API and language specification
Proprietary or open technology	Proprietary	Open and royalty-free
When introduced	Q4 2006	Q4 2008
SDK vendor	Nvidia	Implementation vendors
Free SDK	Yes	Depends on vendor
Multiple implementation vendors	No, just Nvidia	Yes: Apple, Nvidia, AMD, IBM
Multiple OS support	Yes: Windows, Linux, Mac OS X; 32 and 64-bit	Depends on vendor
Heterogeneous device support	No, just Nvidia GPUs	Yes
Embedded profile available	No	Yes

(Sami Rosendahl, "CUDA and OpenCL API comparison")

Interoperability

- CUDA/OpenCL can share buffers with OpenCL/DirectX
- Good for real-time/interactive visualization of results without passing through external applications
- OpenGL introduced *compute shaders* to embed a kind of miniature-OpenCL within its API

THE NEED FOR SPEED…

FFT example

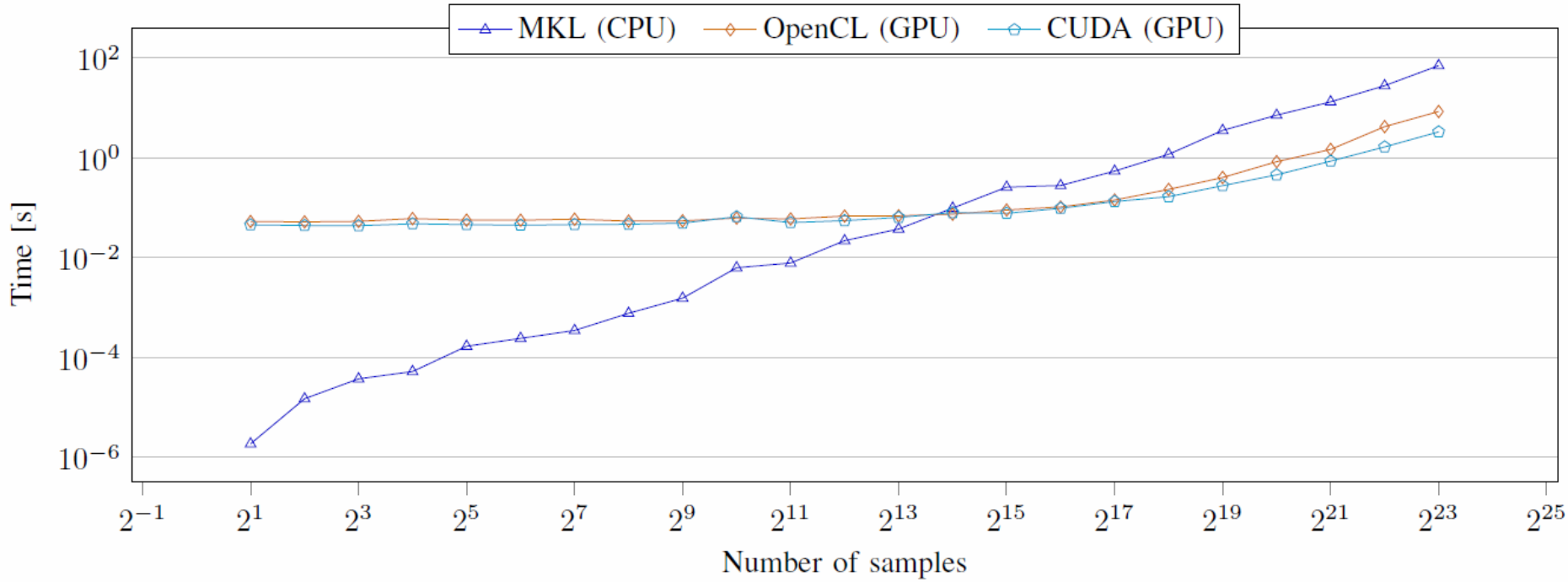


Fig. 1. FFT scalability test on CPU and GPU using an increasing number of samples (N) and three different implementations. Logarithmic scale is used on both axes.

CPU: Intel i7-930 2.8 GHz 4 cores
GPU: Nvidia Tesla C2050 1.15 GHz

(Peternier et al., APSAR 2013, to appear)

FFT example

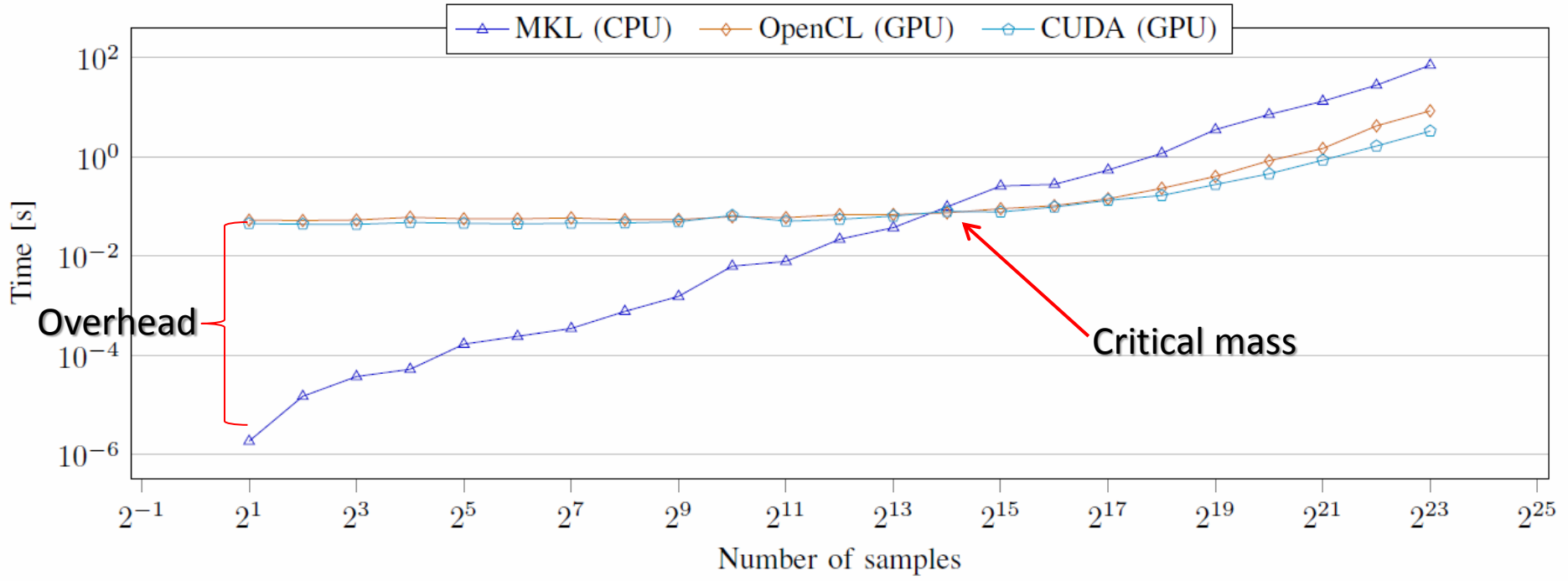


Fig. 1. FFT scalability test on CPU and GPU using an increasing number of samples (N) and three different implementations. Logarithmic scale is used on both axes.

CPU: Intel i7-930 2.8 GHz 4 cores
GPU: Nvidia Tesla C2050 1.15 GHz

(Peternier et al., APSAR 2013, to appear)

PS example

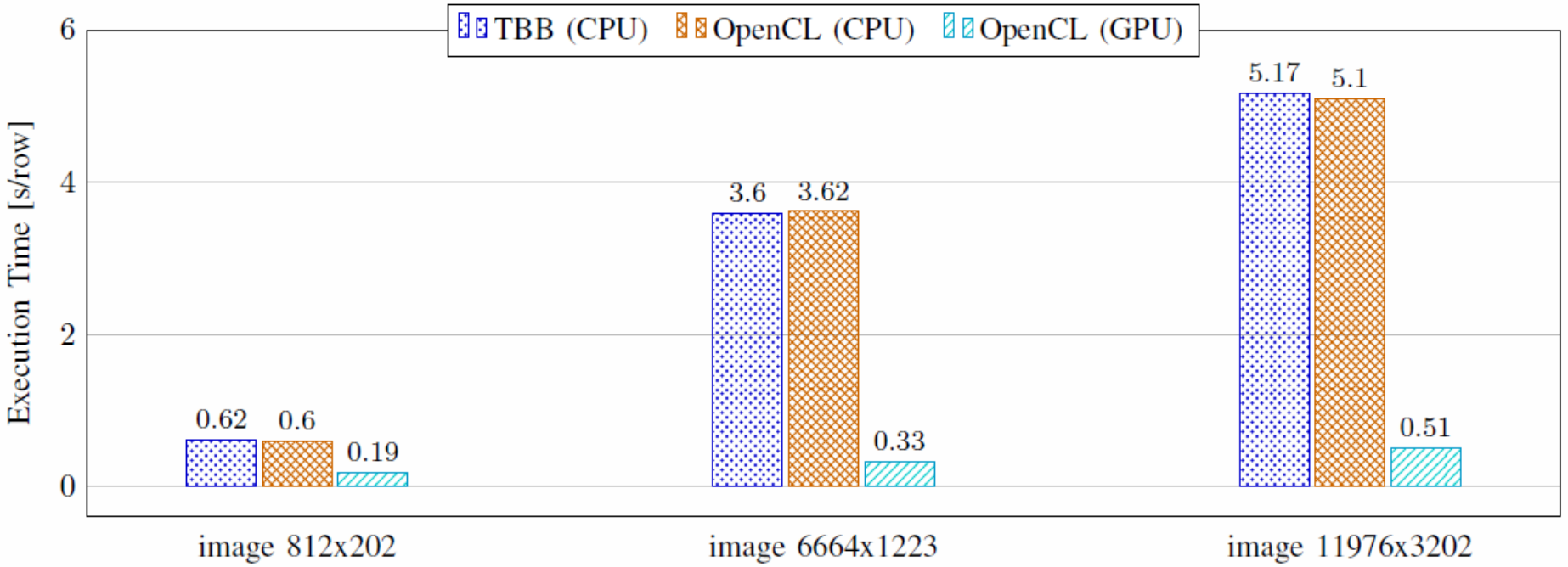


Fig. 4. Persistent scatterers performance comparison experiment using different image sizes and three different configurations.

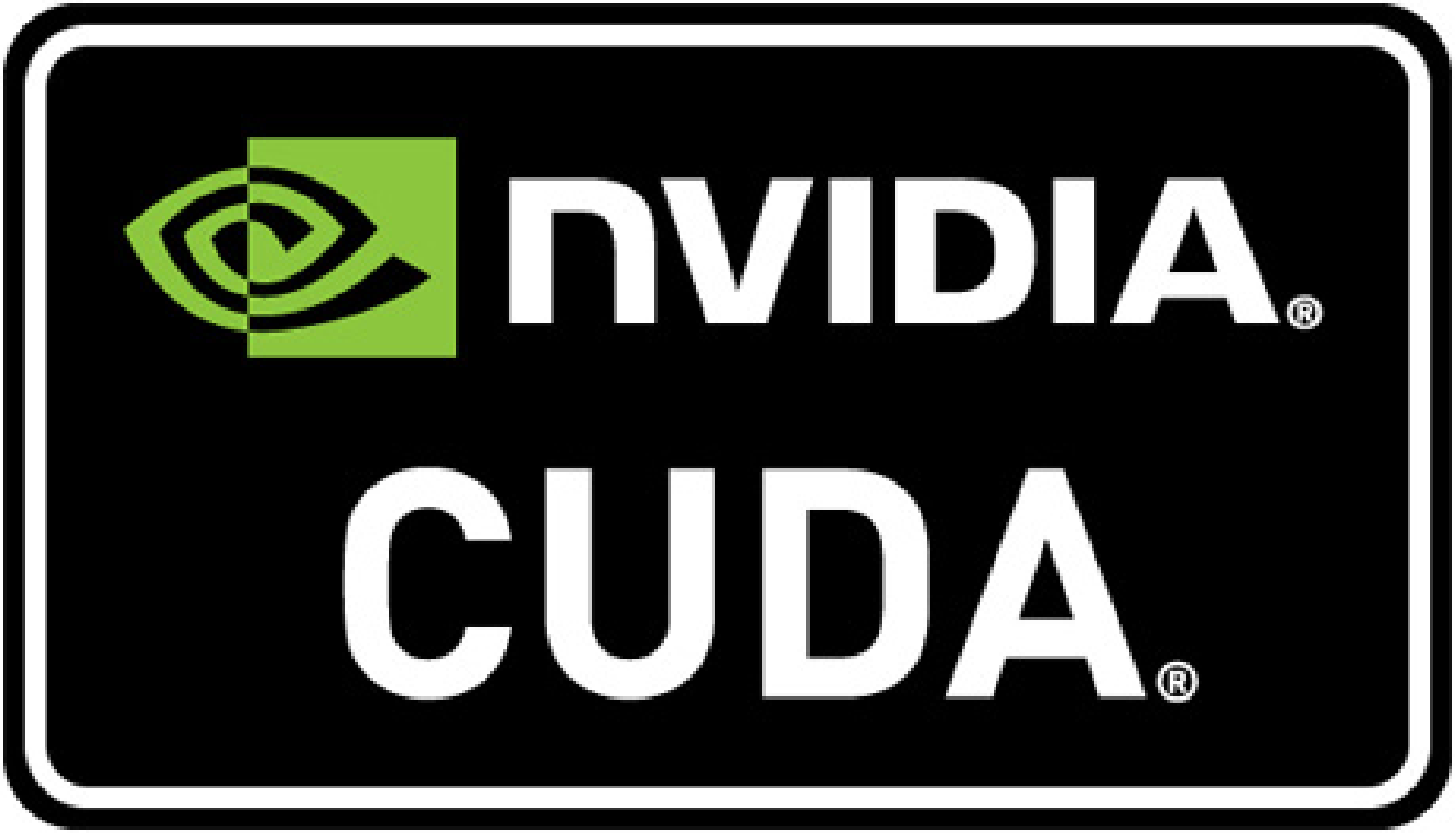
(Peternier et al., APSAR 2013, to appear)

Propag5

- Hearth electric activity simulator
  - Pure C+OpenMP vs OpenCL (Intel CPU only)
    - OpenCL 50% faster!
      - App wall-time, including not-optimized memory copies (no pinned/mapped memory used)
    - Better vectorization/SIMD?
  - Pure C+OpenMP vs OpenCL (K20 GPU)
    - OpenCL kernel 8x faster than C+OpenMP kernels…
      - …but wall-time only 60-70% faster than baseline
    - Lot of time wasted in copying memory to/from device

OpenCL/CUDA dilemma

- A deep understanding of the underlying HW is required for producing high-performance GPU code
- Both APIs are rather simple: the problem is the HW know-how

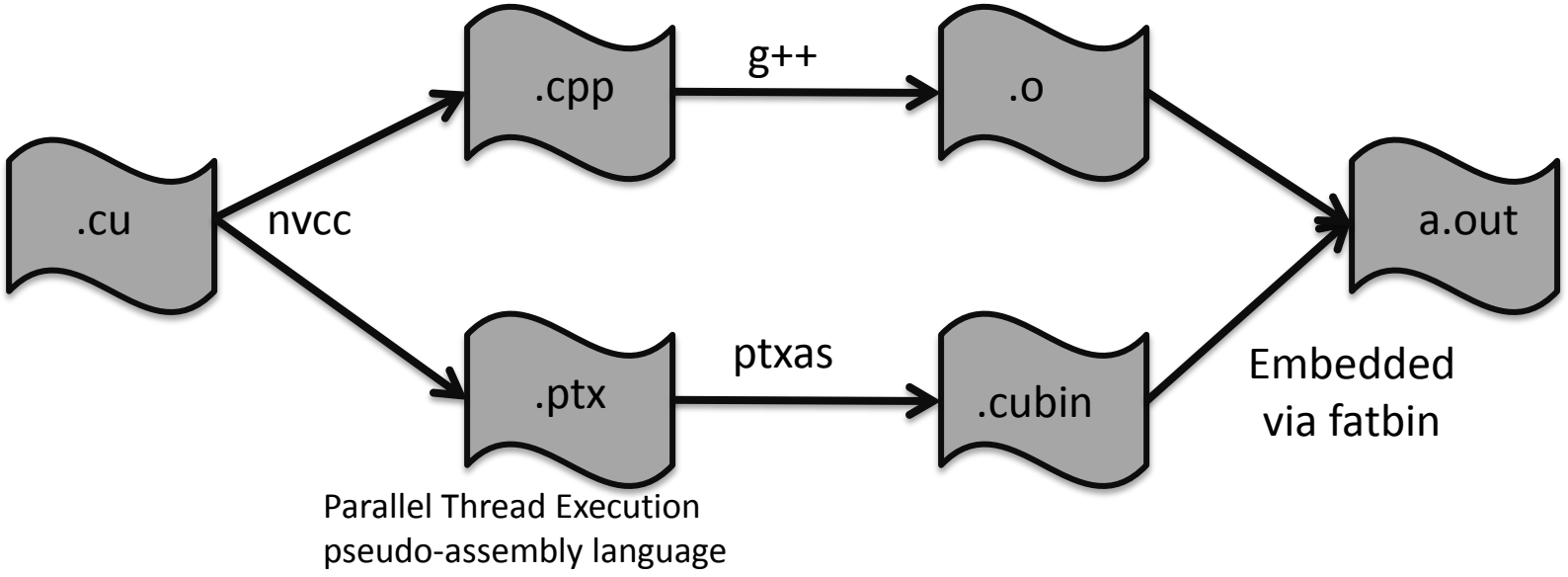

NVIDIA®
CUDA®

CUDA

- Compute Unified Device Architecture
- Introduced in 2006
- Created, maintained and owned by NVidia
- Current version: 5.5

CUDA

- Parallel programming model
- Serial host code spawns parallel device code (**kernel**)
- Compiler **nvcc** based on open64
- Two APIs: **runtime** (higher-level, C++ only) and **driver** (low-level)
- C for CUDA is a *variant* of C++ (extension plus restrictions for kernel code)



CUDA

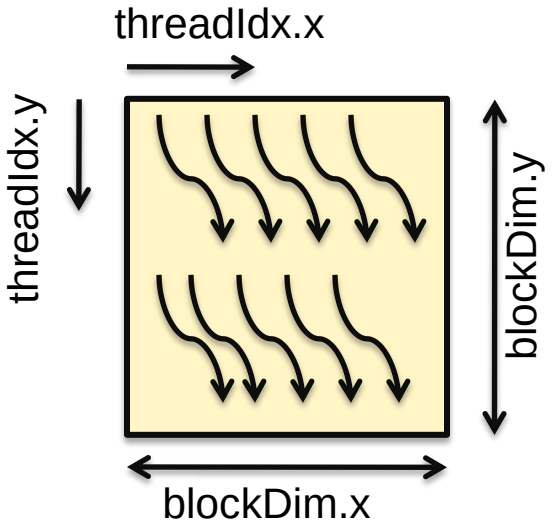
- Extended version of the C programming language:
  - Host/device type qualifier for functions
    - __global__ __device__ __host__
  - Variable type qualifier
    - __device__ __shared__
  - A few built-in variable for specifying grid and block dimensions

CUDA memory

Memory	Location	Description	Lifetime	Scope
register	on-chip	Zero extra cycles per instruction. To avoid bank conflicts the thread block size should be a multiple of 64. **GT200:** 64 Kbytes per SM **GF100:** 128 Kbytes per SM	thread	thread
shared	on-chip	As fast as register if no bank conflict occurs. **GT200:** 16 Kbytes per SM **GF100:** 16 Kbytes or 48 Kbytes per SM	block	thread in block
global	off-chip	Can be accessed by device and host. Device accesses should be coalesced for maximum performance. **GF200:** L2 cache and L1 cache with 128 bytes cache lines	All threads, host	application
local	off-chip	Used for spilled automatic variables. Should be avoided on G80/GT200 (not cached)	thread	thread
constant	off-chip	Stored in DRAM but cached on each SM	All threads, host	application
texture	off-chip	cached, can enhance random access performance	All threads, host	application

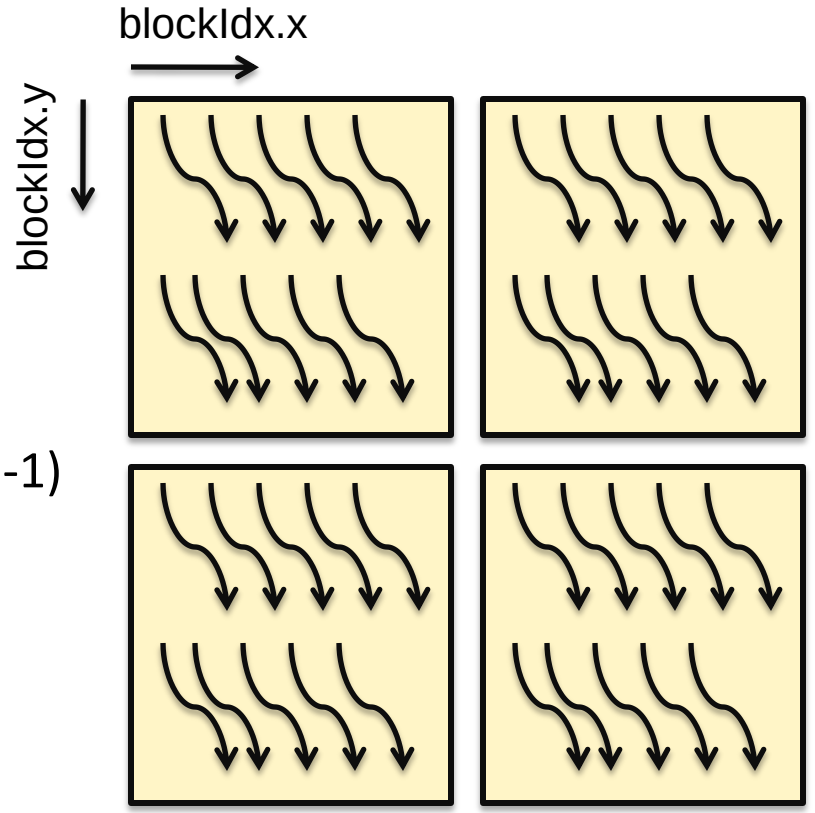
CUDA

- **Thread block**
  - ✓ Array of concurrent threads
  - ✓ 1 – 512 threads (Tesla), 1 – 1024 threads (Fermi, Kepler)
  - ✓ Threads have unique Ids
  - ✓ 1d, 2d or 3d shape (get rid of some index calculations)
  - ✓ Executed by one SM (not migrated)
  - ✓ Thread block size = vector length, strip mined into warps *[Volkov & Demmel]*
  - ✓ Threads in block share access to shared memory
  - ✓ Fast synchronization



CUDA

- **Grid**
  - ✓ Array of thread blocks
  - ✓ 1/2/3d shapes
  - ✓ Maximum size dictated by hardware (C1060/C2070: Max dimension is 65535, Kepler 2^32-1)
  - ✓ Provide coarse-grained parallelization
  - ✓ No language support for synchronization between blocks in grids
  - ✓ Sequentially dependent application steps must be mapped to seq. dependent grids



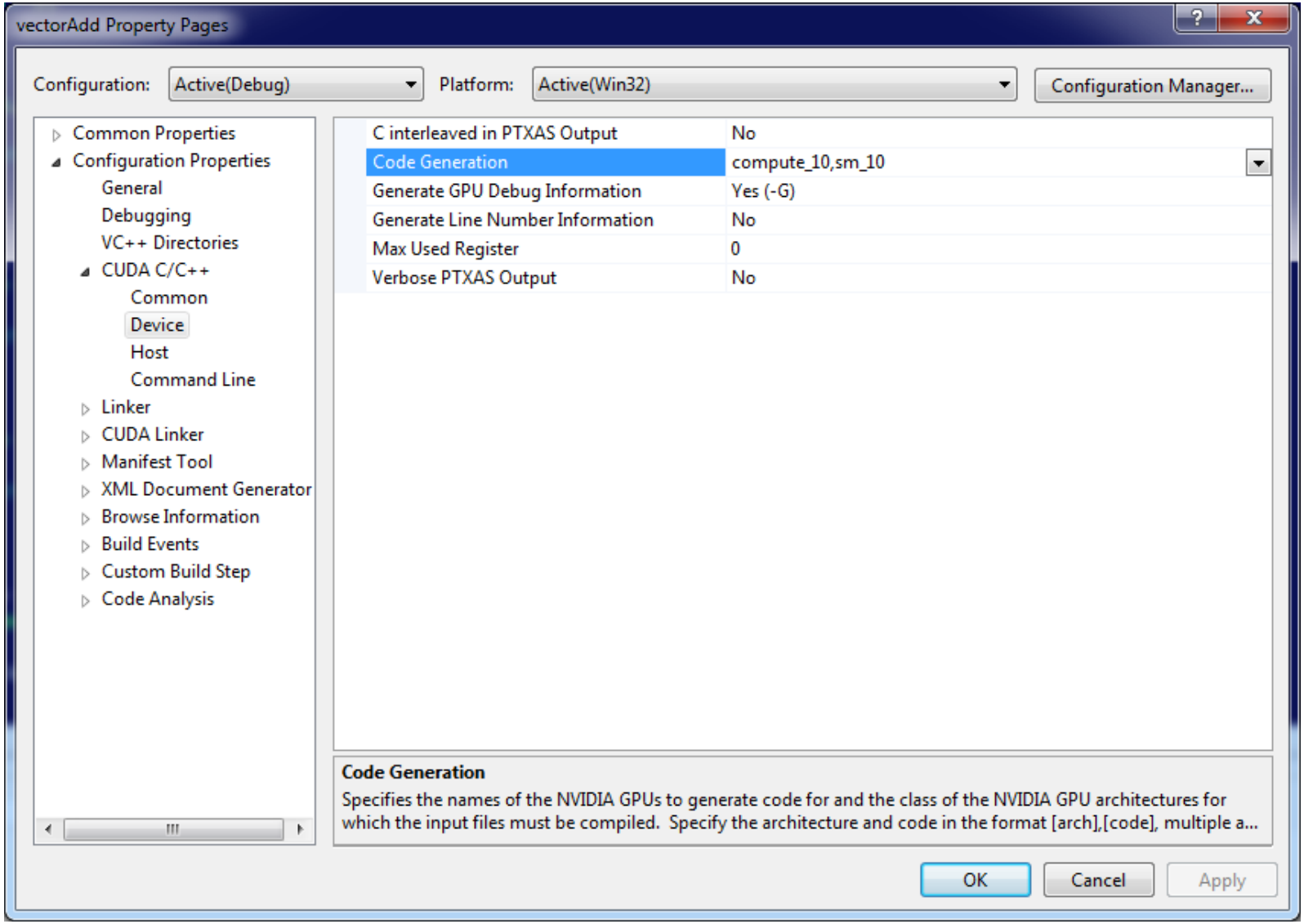
Compute capability

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

(Kepler GK110 White Paper, NVidia Corp.)

Compute capability



- nvcc –arch compute_13

CUDA example


```
__device__ __host__ void saxpy1(float a, float* x, float* y)
{
   *y= *y + a*(*x);
}

__global__ void saxpy(int N, float a, float* x, float* y)
{
    const int i = blockIdx.x*blockDim.x + threadIdx.x
    if (i < N) {
        saxpy1(a, x+i, y+i);
    }
}

int main()
{
    …
    saxpy<<<N/128,128>>>(N, 0.5, x, y);
    …
}
```

N items processed by a grid of N/128 thread blocks of 128 threads each

CUDA API

- cudaMalloc, cudaFree: Device memory management.
- cudaHostAlloc, cudaHostFree: Allocate pinned (page-locked) buffer in host memory (for improved device host bandwidth and asynchronous copies).
- cudaMemcpy: Host-to-device, Device-to-host, Device-to-device and Host-to-host copies.
- cudaMemcpyAsync: Asynchronous copy. Can be overlapped with kernels using multiple **streams**.
- cudaMemset: Initialize device memory.
- cudaGetLastError, cudaGetErrorString: Query error codes and their meaning.
  E.g. Unspecified launch failure ≈ Segmentation fault on host

Coalesced memory

(less important with modern GPUs and CUDA versions)

- "Global memory bandwidth is used most efficiently when simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction" CUDA Programming Guide

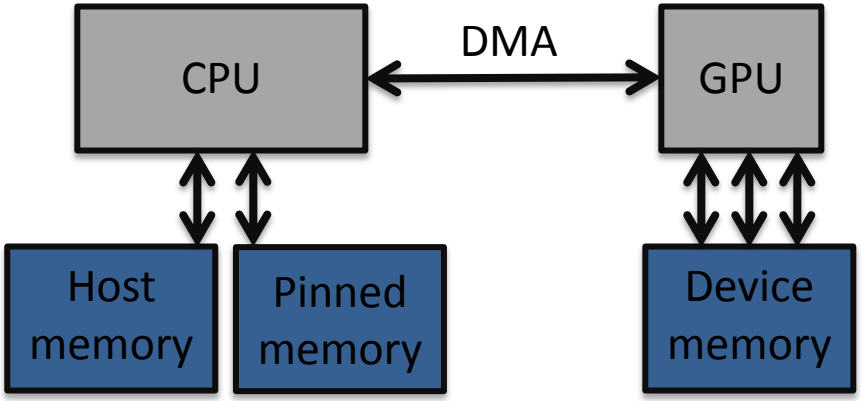
- Whether or not accesses can be coalesced can be coalesced depends on the access pattern, alignment and compute capability
  - ✓ Device with higher compute capability have less stringent requirements
  - ✓ Caches in GF100 help but optimizations are still advantageous

- **Rule of Thumb:** Threads with contiguous `threadIdx` should access contiguous words. The first word should be aligned on a 64 or 128 byte boundary (depends on datatype).
- **Note:** Optimum stride-1 access across threads (half-warp) instead of stride-1 access by a single thread (as on commodity CPUs).

Pinned memory

- Pageable memory (malloc, new)
  - Host -> pinned copy -> DMA -> device
  - You don't want memory to be moved during operations…
- With pinned memory:
  - Host (already pinned) -> DMA -> device

You skip a buffer copy!
(can be up to 2x faster)



Tools

- CUDA toolkit SDK
- Debuggers
- Profilers (best way to optimize your code)
- IDEs
- Books, tutorials, examples, etc.

Advantages

- Short learning curve
- Less optimizations required
- Good support
- Good tools and derived libs (e.g., cuBLAS, cuFFT, …)

Drawbacks

- NVidia-only, not an open standard
- An NVidia GPU is required
  - ARM support recently announced
  - Will GPUs remain the mainstream for HPC?
    - FPGA, ARM, MIC, etc.

Exercise #2

- Download and compile the CUDA vectorAdd example
  - Measure time spent in copying memory and running the kernel
  - How can you optimize grid and block sizes?
  - Use asynchronous memory copy for input buffers
  - Use pinned memory: how faster/slower it is?
  - Refactor the code into a quadratic equation solver:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

OpenCL

OpenCL

- Open Computing Language
- First spec released in December 2008
- Maintained by the non-profit Khronos Group
  - OpenGL, COLLADA, WebGL, etc.
- Writing parallel algorithms is still difficult, but the API and its portability is now much easier
- Current version 1.2
  - Version 2.0 (draft) available

OpenCL support

	NVIDIA	AMD	Intel
OpenCL	1.1 (but 1.2 reported on new devices)	1.2	1.2

Advantages

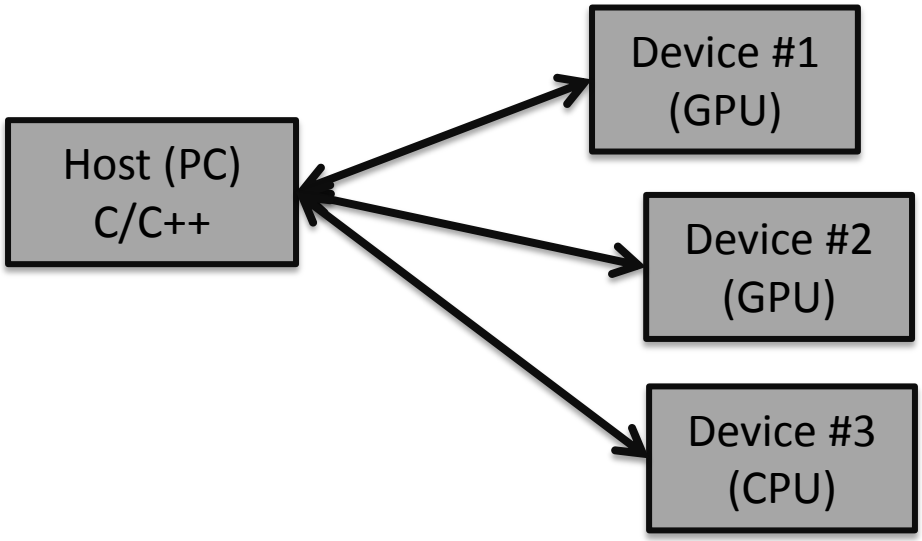
- Portable
- Cross-device
- It runs on CPUs as well

Drawbacks

- Longer learning curve than CUDA
  - API somehow similar to OpenGL
- Code is portable, performance is not

OpenCL

- One host application connected to one or more devices



OpenCL context

- First select a **cl_platform_id** to use
  - Usually platform == device vendor (e.g., NVidia, AMD, Intel, etc.)
- One **cl_platform_id** platform has at least one device
- A **cl_device_id** identifies a specific device within the selected platform
- A **cl_context** enables OpenCL on the selected devices being part of a specific (and same) platform
  - Contexts across multiple platforms are not supported (→that's the reason way platforms do exist)

OpenCL context

- Example (Desktop PC with 2 GPUs):
  - Platform 0: NVidia,
    - Device 0: Geforce GTX 560
      - Type: GPU
    - Device 1: Tesla C2050
      - Type: GPU
  - Platform 1: Intel
    - Device 0: Intel Xeon
      - Type: CPU
  - Platform 2: AMD
    - Device 0: AMD compatible
      - Type: CPU

OpenCL context

- Example (dual-CPU server with 4 GPUs):
  - Platform 0: NVidia,
    - Device 0: Tesla K20
      - Type: GPU
    - Device 1: Tesla K20
      - Type: GPU
    - Device 2: Tesla K20
      - Type: GPU
    - Device 3: Tesla K20
      - Type: GPU
  - Platform 1: Intel
    - Device 0: Intel Xeon
      - Type: CPU (12 cores)

OpenCL

- Compiled like any other application
  - No specific compiler required
  - Just cl.h and opencl.lib

hwinfo example

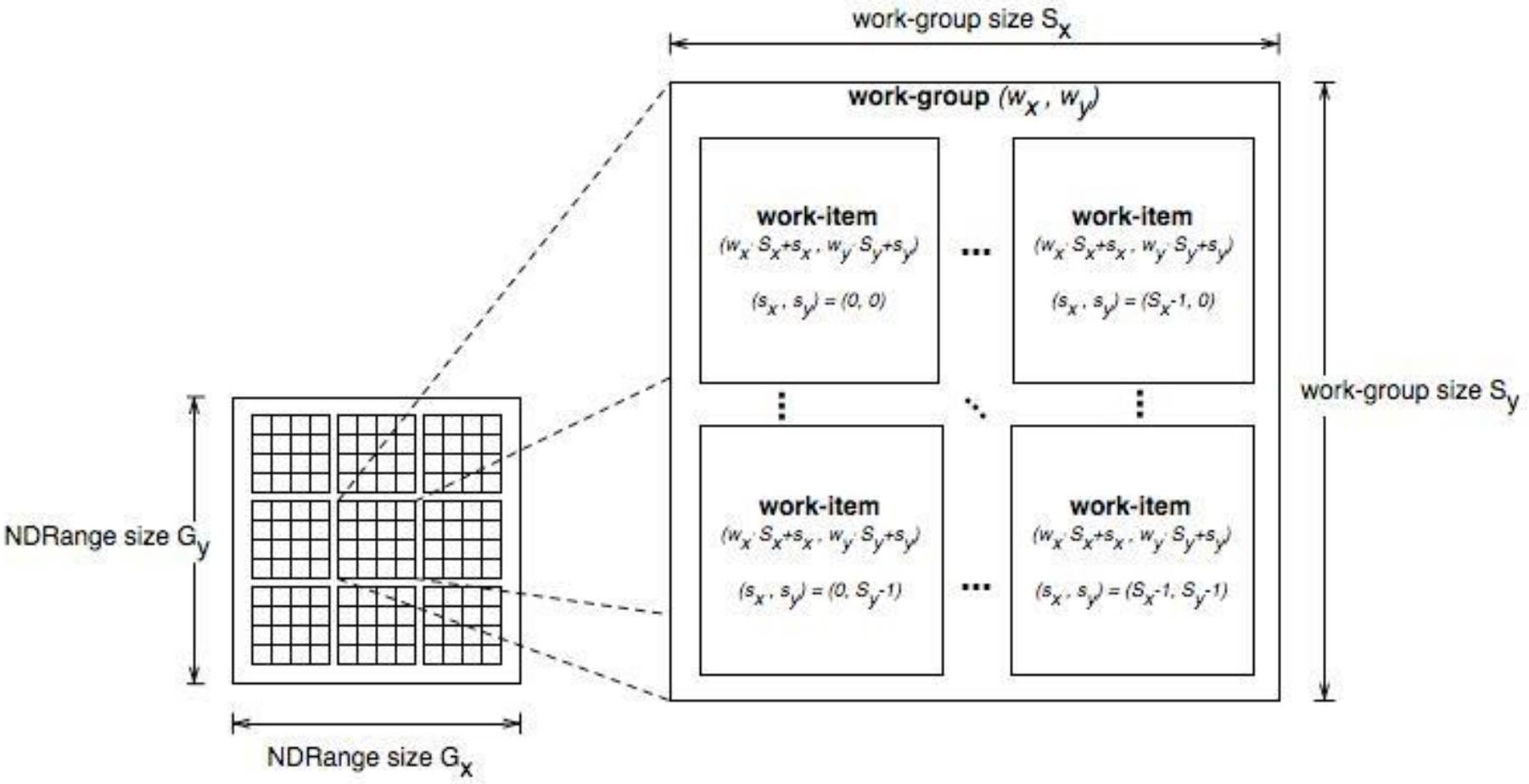
- Download and compile the hwinfo example
  - It also reports on host-side specs
    - Uncomment the code
    - Uses the hardware locality lib (hwloc, not installed on todi)

```
------------------
Hardware topology:
------------------

 Available RAM . . . :  5507 MB
 Nr. of systems  . . :  1
 Nr. of NUMA nodes . :  1
 Nr. of CPUs . . . . :  1
 Nr. of cores  . . . :  2
 Nr. of PUs  . . . . :  4


-----------------
OpenCL platforms:
-----------------

 Platforms found  :  1
  Platform 0  . . :  Intel(R) OpenCL
   GPUs . . . . . :  0
   CPUs . . . . . :  1
    Device 0  . . :  Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz (defau
    Device version:  OpenCL 1.1 (Build 37149.37214)
    Driver version:  1.1
```

OpenCL objects

- Are created within the current context
- **cl_program**: the kernel source code
  - Can contain multiple functions
- **cl_kernel**: a specific functions/entry point that can be invoked from the host
- **cl_command_queue**: a series of operations that are assigned to a specific device
  - Memory copy, kernel execution, …
- **cl_mem**: memory buffers

OpenCL enqueueNDRange



OpenCL API ref. card

http://www.khronos.org/opencl/

OpenCL API 1.1 Quick Reference Card - Page 1

OpenCL™ (Open Computing Language) is a multi-vendor open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs and other processors. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance compute servers, desktop computer systems and handheld devices.

[n.n.n] refers to the section in the API Specification available at www.khronos.org/opencl.

The OpenCL Platform Layer

The OpenCL platform layer implements platform-specific features that allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices.

Contexts [4.3]

cl_context **clCreateContext** (
const cl_context_properties *properties, cl_uint num_devices,
const cl_device_id *devices, void (CL_CALLBACK*pfn_notify)
(const char *errinfo, const void *private_info,
size_t cb, void *user_data),
void *user_data, cl_int *errcode_ret)

properties: CL_CONTEXT_PLATFORM, CL_GL_CONTEXT_KHR, CL_CGL_SHAREGROUP_KHR, CL_{EGL, GLX}_DISPLAY_KHR, CL_WGL_HDC_KHR

cl_context **clCreateContextFromType** (
const cl_context_properties *properties,
cl_device_type device_type, void (CL_CALLBACK *pfn_notify)
(const char *errinfo, const void *private_info, size_t cb,
void *user_data),
void *user_data, cl_int *errcode_ret)

properties: See clCreateContext

cl_int **clRetainContext** (cl_context context)

cl_int **clReleaseContext** (cl_context context)

cl_int **clGetContextInfo** (cl_context context,
cl_context_info param_name, size_t param_value_size,
void *param_value, size_t *param_value_size_ret)

param_name: CL_CONTEXT_REFERENCE_COUNT, CL_CONTEXT_{DEVICES, PROPERTIES}, CL_CONTEXT_NUM_DEVICES

Querying Platform Info and Devices [4.1, 4.2]

cl_int **clGetPlatformIDs** (cl_uint num_entries,
cl_platform_id *platforms, cl_uint *num_platforms)

cl_int **clGetPlatformInfo** (cl_platform_id platform,
cl_platform_info param_name, size_t param_value_size,
void *param_value, size_t *param_value_size_ret)

param_name: CL_PLATFORM_{PROFILE, VERSION}, CL_PLATFORM_{NAME, VENDOR, EXTENSIONS}

cl_int **clGetDeviceIDs** (cl_platform_id platform,
cl_device_type device_type, cl_uint num_entries,
cl_device_id *devices, cl_uint *num_devices)

device_type: CL_DEVICE_TYPE_{CPU, GPU}, CL_DEVICE_TYPE_{ACCELERATOR, DEFAULT, ALL}

cl_int **clGetDeviceInfo** (cl_device_id device,
cl_device_info param_name, size_t param_value_size,
void *param_value, size_t *param_value_size_ret)

param_name: CL_DEVICE_TYPE, CL_DEVICE_{VENDOR_ID, MAX_COMPUTE_UNITS, MAX_WORK_ITEM_{DIMENSIONS, SIZES}, MAX_WORK_GROUP_SIZE, {NATIVE_PREFERRED}_VECTOR_WIDTH_{CHAR, SHORT, INT, LONG, FLOAT, DOUBLE, HALF}, MAX_CLOCK_FREQUENCY, ADDRESS_BITS, MAX_{MEM_ALLOC_SIZE, IMAGE_SUPPORT, READ_WRITE_IMAGE_ARGS, IMAGE{2D, 3D}_MAX_{WIDTH, HEIGHT, DEPTH}}, MAX_{SAMPLERS, PARAMETER_SIZE}, MEM_BASE_ADDR_ALIGN, {MIN_DATA_TYPE_ALIGN, SINGLE_FP_CONFIG}, GLOBAL_MEM_{CACHE_{TYPE, SIZE}, CACHELINE_SIZE, SIZE}, MAX_CONSTANT_{BUFFER_SIZE, ARGS}, LOCAL_MEM_{TYPE, SIZE}, ERROR_CORRECTION_SUPPORT, PROFILING_TIMER_RESOLUTION, ENDIAN_LITTLE, AVAILABLE, COMPILER_AVAILABLE, {EXECUTION, QUEUE_PROPERTIES}, {NAME, VENDOR, PROFILE, EXTENSIONS}, {HOST_UNIFIED_MEMORY, OPENCL_C_VERSION}, VERSION, DRIVER_VERSION, CL_DEVICE_PLATFORM

The OpenCL Runtime

Command Queues [5.1]

cl_command_queue **clCreateCommandQueue** (
cl_context context, cl_device_id device,
cl_command_queue_properties properties,
cl_int *errcode_ret)

properties: CL_QUEUE_{PROFILING_ENABLE, OUT_OF_ORDER_EXEC_MODE_ENABLE}

cl_int **clRetainCommandQueue** (
cl_command_queue command_queue)

cl_int **clReleaseCommandQueue** (
cl_command_queue command_queue)

cl_int **clGetCommandQueueInfo** (
cl_command_queue command_queue,
cl_command_queue_info param_name,
size_t param_value_size,
void *param_value,
size_t *param_value_size_ret)

param_name: CL_QUEUE_{CONTEXT, CL_QUEUE_DEVICE, CL_QUEUE_REFERENCE_COUNT, CL_QUEUE_PROPERTIES}

Buffer Objects

Elements of a buffer object can be a scalar or vector data type or a user-defined structure. Elements are stored sequentially and are accessed using a pointer by a kernel executing on a device. Data is stored in the same format as it is accessed by the kernel.

Create Buffer Objects [5.2.1]

cl_mem **clCreateBuffer** (cl_context context,
cl_mem_flags flags, size_t size, void *host_ptr,
cl_int *errcode_ret)

cl_mem **clCreateSubBuffer** (cl_mem buffer,
cl_mem_flags flags,
cl_buffer_create_type buffer_create_type,
const void *buffer_create_info, cl_int *errcode_ret)

flags for clCreateBuffer and clCreateSubBuffer: CL_MEM_{READ_WRITE, {WRITE, READ}_ONLY, {USE, ALLOC, COPY}_HOST_PTR}

Read, Write, Copy Buffer Objects [5.2.2]

cl_int **clEnqueueReadBuffer** (
cl_command_queue command_queue, cl_mem buffer,
cl_bool blocking_read, size_t offset, size_t cb,
void *ptr, cl_uint num_events_in_wait_list,
const cl_event *event_wait_list, cl_event *event)

cl_int **clEnqueueWriteBuffer** (
cl_command_queue command_queue, cl_mem buffer,
cl_bool blocking_write, size_t offset, size_t cb,
const void *ptr, cl_uint num_events_in_wait_list,
const cl_event *event_wait_list, cl_event *event)

cl_int **clEnqueueReadBufferRect** (
cl_command_queue command_queue, cl_mem buffer,
cl_bool blocking_read, const size_t buffer_origin[3],
const size_t host_origin[3], const size_t region[3],
size_t buffer_row_pitch, size_t buffer_slice_pitch,
size_t host_row_pitch, size_t host_slice_pitch,
void *ptr, cl_uint num_events_in_wait_list,
const cl_event *event_wait_list, cl_event *event)

cl_int **clEnqueueWriteBufferRect** (
cl_command_queue command_queue, cl_mem buffer,
cl_bool blocking_write, const size_t buffer_origin[3],
const size_t host_origin[3], const size_t region[3],
size_t buffer_row_pitch, size_t buffer_slice_pitch,
size_t host_row_pitch, size_t host_slice_pitch,
void *ptr, cl_uint num_events_in_wait_list,
const cl_event *event_wait_list, cl_event *event)

cl_int **clEnqueueCopyBuffer** (
cl_command_queue command_queue,
cl_mem src_buffer, cl_mem dst_buffer, size_t src_offset,
size_t dst_offset, size_t cb,
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list, cl_event *event)

cl_int **clEnqueueCopyBufferRect** (
cl_command_queue command_queue,
cl_mem src_buffer, cl_mem dst_buffer,
const size_t src_origin[3], const size_t dst_origin[3],
const size_t region[3], size_t src_row_pitch,
size_t src_slice_pitch, size_t dst_row_pitch,
size_t dst_slice_pitch, cl_uint num_events_in_wait_list,
const cl_event *event_wait_list, cl_event *event)

Map Buffer Objects [5.2.2]

void * **clEnqueueMapBuffer** (
cl_command_queue command_queue, cl_mem buffer,
cl_bool blocking_map, cl_map_flags map_flags,
size_t offset, size_t cb, cl_uint num_events_in_wait_list,
const cl_event *event_wait_list, cl_event *event,
cl_int *errcode_ret)

Map Buffer Objects [5.4.1.2]

cl_int **clRetainMemObject** (cl_mem memobj)

cl_int **clReleaseMemObject** (cl_mem memobj)

cl_int **clSetMemObjectDestructorCallback** (
cl_mem memobj, void (CL_CALLBACK *pfn_notify)
(cl_mem memobj, void *user_data),
void *user_data)

cl_int **clEnqueueUnmapMemObject** (
cl_command_queue command_queue, cl_mem memobj,
void *mapped_ptr, cl_uint num_events_in_wait_list,
const cl_event *event_wait_list, cl_event *event)

Query Buffer Object [5.4.3]

cl_int **clGetMemObjectInfo** (cl_mem memobj,
cl_mem_info param_name, size_t param_value_size,
void *param_value, size_t *param_value_size_ret)

param_name: CL_MEM_{TYPE, FLAGS, SIZE, HOST_PTR}, CL_MEM_{MAP_REFERENCE}_COUNT, CL_MEM_OFFSET, CL_MEM_CONTEXT, CL_MEM_ASSOCIATED_MEMOBJECT

Program Objects

Create Program Objects [5.6.1]

cl_program **clCreateProgramWithSource** (
cl_context context, cl_uint count, const char **strings,
const size_t *lengths, cl_int *errcode_ret)

cl_program **clCreateProgramWithBinary** (
cl_context context, cl_uint num_devices,
const cl_device_id *device_list, const size_t *lengths,
const unsigned char **binaries, cl_int *binary_status,
cl_int *errcode_ret)

cl_int **clRetainProgram** (cl_program program)

cl_int **clReleaseProgram** (cl_program program)

Build Program Executable [5.6.2]

cl_int **clBuildProgram** (cl_program program,
cl_uint num_devices, const cl_device_id *device_list,
const char *options, void (CL_CALLBACK*pfn_notify)
(cl_program program, void *user_data),
void *user_data)

Build Options [5.6.3]

Preprocessor: *(-D processed in order listed in clBuildProgram)*
-D name -D name=definition -I dir

Optimization options:
-cl-opt-disable
-cl-strict-aliasing
-cl-mad-enable
-cl-no-signed-zeros
-cl-finite-math-only
-cl-fast-relaxed-math
-cl-unsafe-math-optimizations

Math Intrinsics:
-cl-single-precision-constant -cl-denorms-are-zero

Warning request/suppress:
-w -Werror

Control OpenCL C language version:
-cl-std=CL1.1 // OpenCL 1.1 specification.

Query Program Objects [5.6.5]

cl_int **clGetProgramInfo** (cl_program program,
cl_program_info param_name, size_t param_value_size,
void *param_value, size_t *param_value_size_ret)

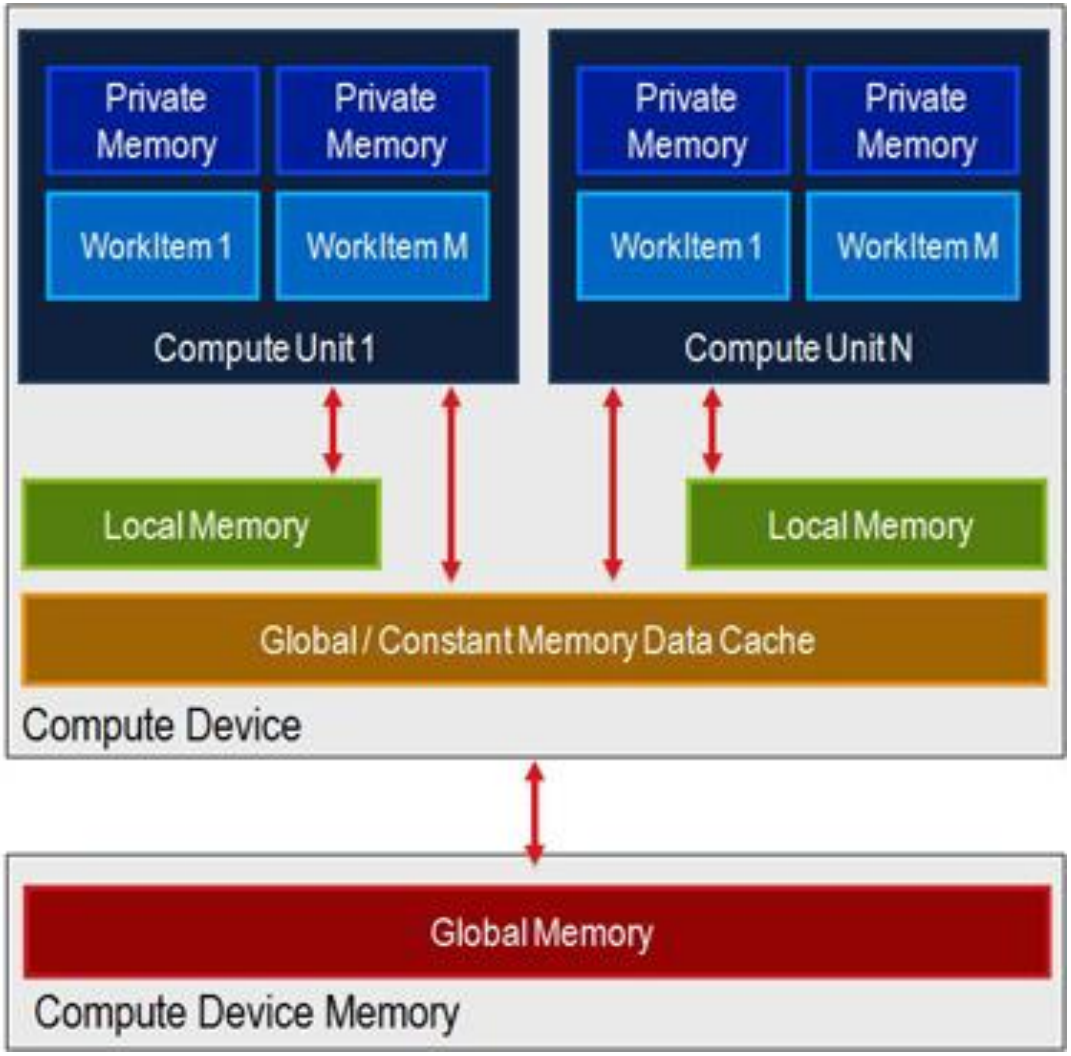
param_name: CL_PROGRAM_{REFERENCE_COUNT}, CL_PROGRAM_{CONTEXT, NUM_DEVICES, DEVICES}, CL_PROGRAM_{SOURCE, BINARY_SIZES, BINARIES}

(Program Objects Continue >)

OpenCL memory model

- Four different address spaces:
  - Global memory: device-wide data (DRAM)
    - clGetDeviceInfo(CL_DEVICE_GLOBAL_MEM_SIZE)
  - Constant memory: same as global, but read-only
    - Usually HW-optimized for broadcast
  - Local memory: shared by work-items within one same workgroup
    - Generally on-chip
  - Private memory: data stored by a single work-item
    - Registers. When full, spills to cache or global memory
- Different address spaces have different speed

OpenCL memory model



(conflicting) optimizations

- HW-specific guidelines
- Performance is not portable:

  "Choosing sensible thread block sizes, such as multiples of 16, facilitates memory accesses by half warps that are aligned to segments." (NVidia)

  "To benefit from the auto-vectorization optimization, the work-group size must be a multiple of 4 for Nehalem micro-architecture, and 8 for SandyBridge respectively. " (Intel)
- Tuning device A might reduce performance portability to device B and C
- NVidia Compute Capabilities vs. OpenCL clGetDeviceInfo() scanning

A nice trick…

- On-the-fly kernel code generation has some nice advantage:
  - Built-in constant, to reduce register usage
  - Dynamic code generation
    - HW-aware
    - Algorithm-ad hoc

2 min of hacking…

…on the AMD FFT

Tools

- OpenCL SDKs
  - Intel, AMD, NVidia (as part of the CUDA SDK)
- Debuggers
- Profilers
- IDEs
- Wrappers
  - C++

Exercise #3

- Matrix multiplication with OpenCL
  - Understand the code
  - Instrument the main operations
  - Improve grid sizes
  - Add built-in constants
  - Add custom platform selection

More examples

- A series of OpenCL examples written and tested on the CSCS systems
  - Written by Ugo Varetto, CSCS
- GIT https://github.com/ugovaretto/opencl-training
- Several topics covered, from device queries to convolution filtering, OpenGL interoperability, C++ wrappers for OpenCL, etc.

Conclusión

Conclusions

- Pick the solution that better suits your need
  - Today and tomorrow…
- OpenCL is particularly suitable for writing intermediate libraries, CUDA to get the stuff done ASAP
- If GPU==NVidia and NEED==MaxSpeed, then go for CUDA
  - Latest features (PCIe DMA, etc.)
- If portability <= 0, follow the guidelines specific for your HW;
  else keep the code implementation-agnostic and use queries to adapt your code to the underlying HW
- Learning CUDA or OpenCL is just a small part of the job: the difficult part is about HW-awareness

SarXEngine

SarXEngine

- OpenCL abstraction layer used in a Synthetic Aperture Radar (SAR) processor accelerated through GPUs
- A.K.A.: how to make OpenCL looking like CUDA ☺

Context initialization


```
#include <sarxengine.h>

int main(int argc, char *argv[])
{
  SxeConfig config;
  config.load("config.xml");

  SarxEngine::init(&config);
  …
```

```
<?xml version="1.0" encoding="UTF-8"?>
<sarxengine>
        <platformid>NVidia</platformid>
        <deviceid>1</deviceid>
</sarxengine>
```

config.xml

Building kernels


```
...
  SxeProgram *prog = new SxeProgram();
  prog->load("kernel_source.cl");

  SxeKernel *kernel = new SxeKernel();
  kernel->create(prog, "vectorAdd");

  SxeQueue *queue = new SxeQueue();
  queue->create();
...
```

```
__kernel void vectorAdd(__global float * v1,
                        __global float * v2,
                        __global float * v3)
{
  int i = get_global_id(0);
  v3[i] = v1[i] + v2[i];
}
```

kernel_source.cl

Allocating buffers


```
…
  SxeBuffer *vect1 = new SxeBuffer();
    vect1->create(N * sizeof(float), SxeBuffer::WRITE_ONLY);

  SxeBuffer *vect2 = new SxeBuffer();
    vect2->create(N * sizeof(float), SxeBuffer::WRITE_ONLY | SxeBuffer::PINNED);
    sxeByte *vect2Ptr = vect2->map(queue, SxeBuffer::WRITE_ONLY);

  SxeBuffer *vect3 = new SxeBuffer();
    vect3->create(N * sizeof(float), SxeBuffer::READ_AND_WRITE | SxeBuffer::PINNED);
    sxeByte *vect3Ptr = vect3->map(queue, SxeBuffer::READ_AND_WRITE);
…
```

Memory copy


```
…
  vect1->write(queue, inputBuffer, SxeBuffer::NON_BLOCKING);
  vect2->write(queue, vect2Ptr, SxeBuffer::NON_BLOCKING);
    queue->waitTermination();
…
```

Kernel args and run


```
…
  kernel->setArg(0, vect1);
  kernel->setArg(1, vect2);
  kernel->setArg(2, vect3);

  queue->push(kernel, N);
  queue->waitForTermination();
…
```

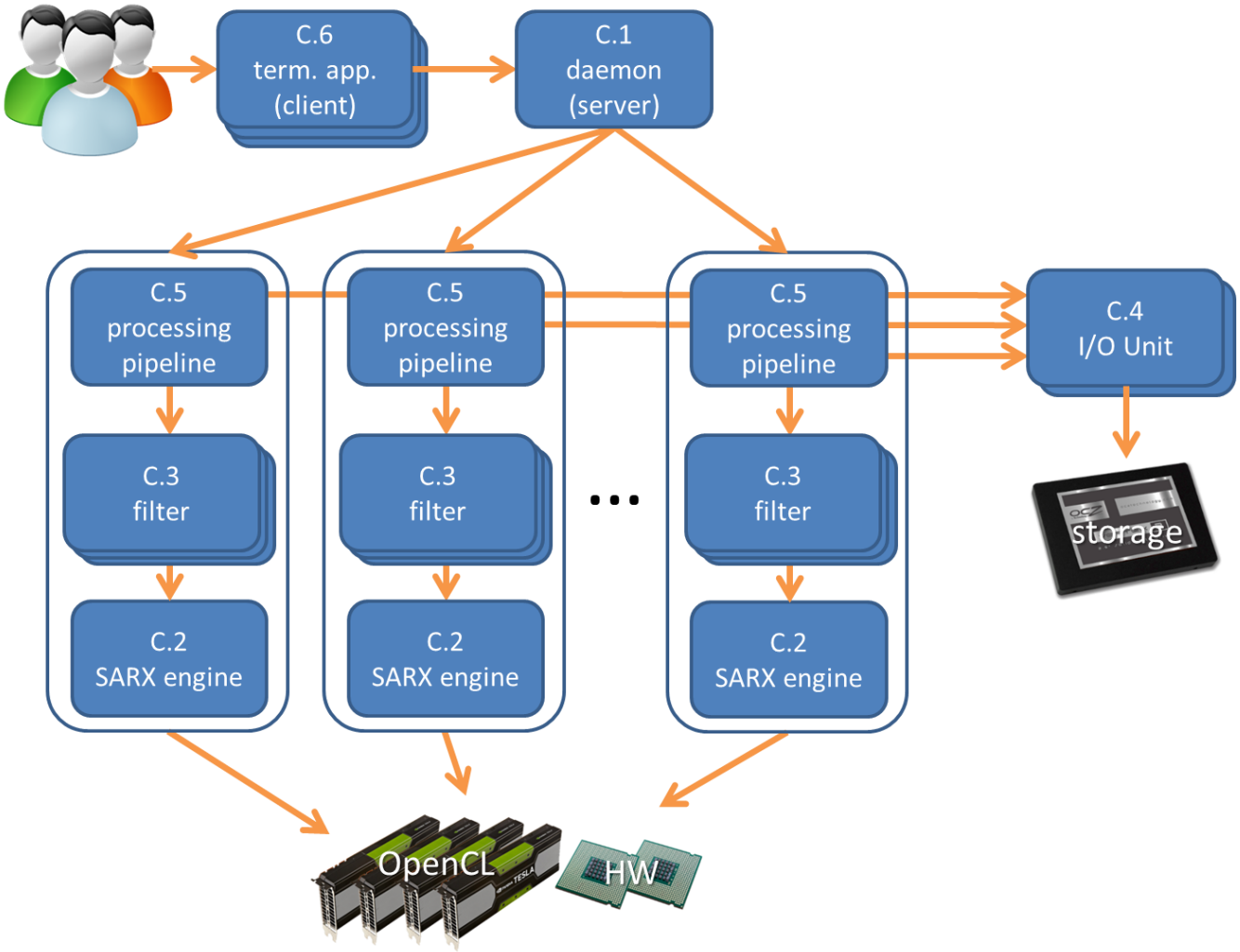
Termination


```
...
  SarxEngine::free()
}
```

Extra functionalities

- Timers (CscsTimer directly taken from…)
- HW scanner (for HW-aware auto-tuning)
- Improved kernel preprocessor (for typed-in constants)
- Inter-process facilities (via boost):
  - Shared message queues
  - Shared memory segments

The big picture




That's all Folks!

Exercise #4

- Improve exercise #3 speed by adopting some of the NVidia OpenCL best practices guidelines