Computer Architecture

Jens Teubner, TU Dortmund
jens.teubner@cs.tu-dortmund.de

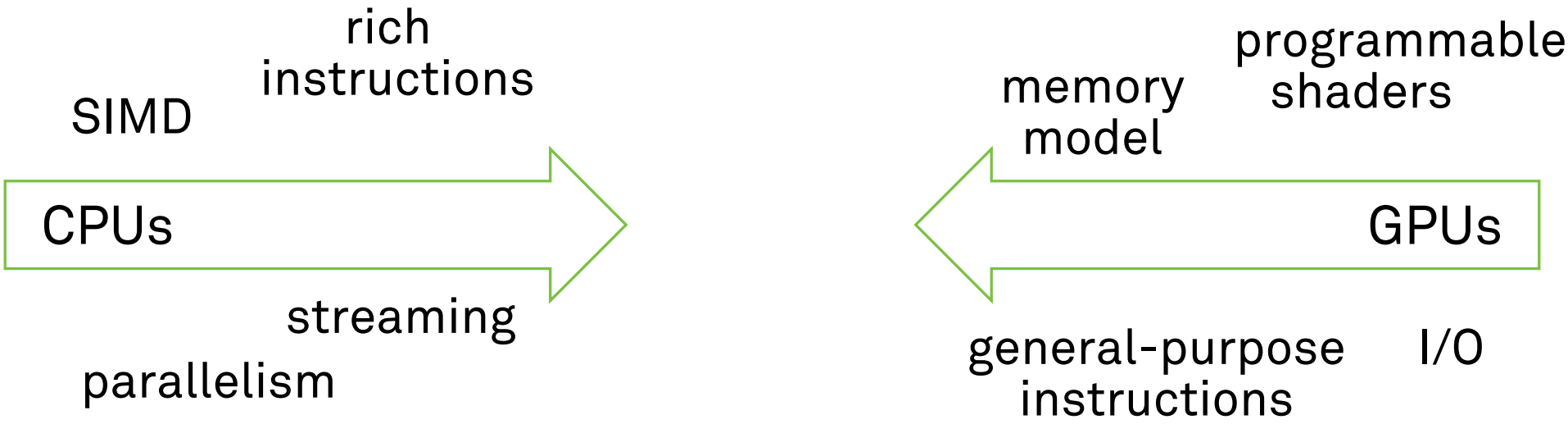
Summer 2016

Part II

Graphics Processing Units (GPUs)

Processor Technology

While **general-purpose CPUs** increasingly feature “multi-media” functionality,



graphics processors become increasingly **general-purpose.**

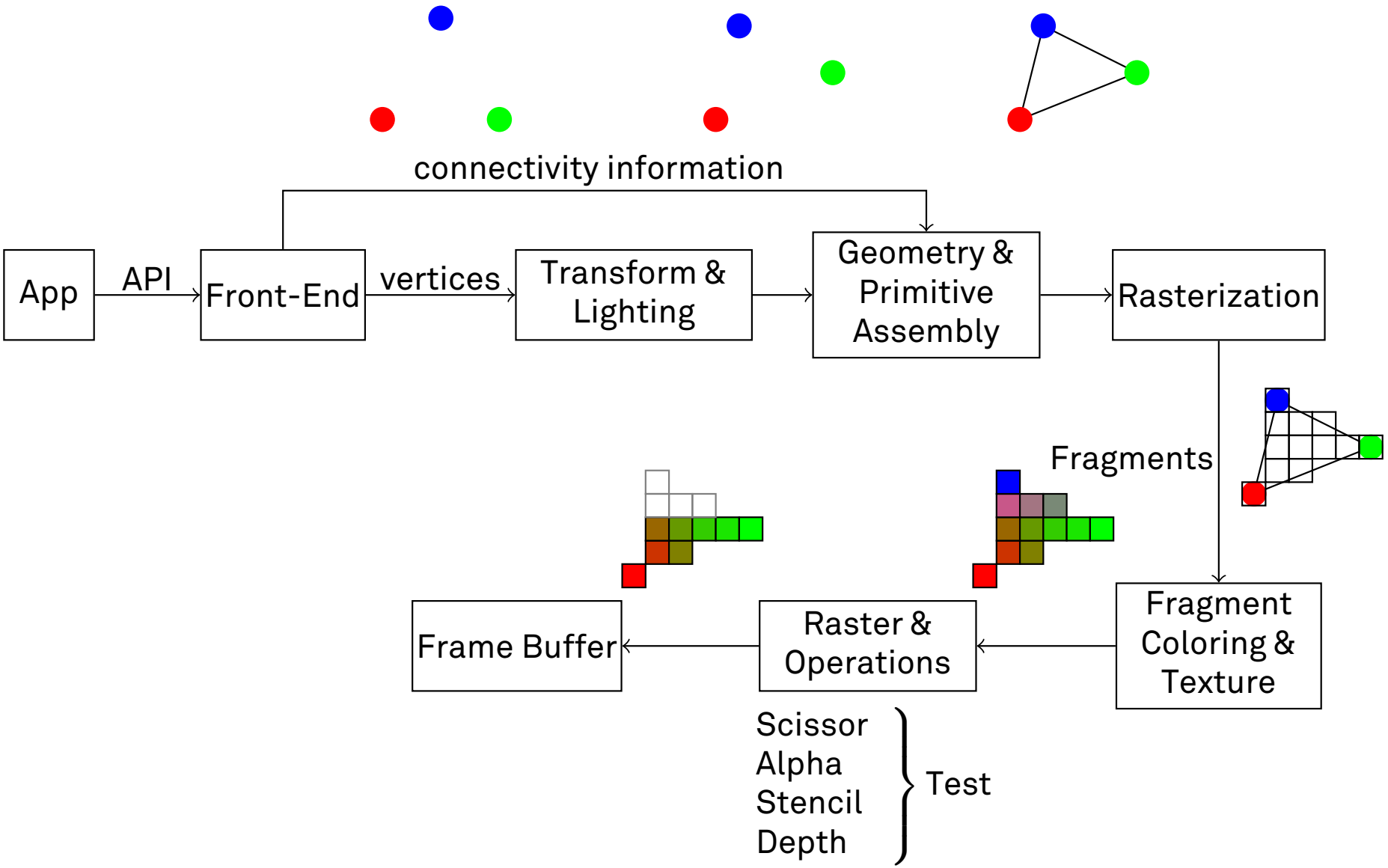
Graphics Processors

Graphics processors come from a world where...

- many problems are embarrassingly parallel,
- for each data point, there are few and fairly simple operations,
- caches, coherency, etc. are less of an issue.

Graphics processors evolved from dedicated hardware to accelerate typical **graphics pipelines**.

Graphics Pipeline



Toward Programmable GPUs

The programmability of GPUs has improved dramatically.

hard-coded **fix-function pipeline**

customization through **parameters**

programmable **shaders**

- vertex shader
- geometry shader
- fragment shader (fragment: pixel)

"general-purpose" GPUs (GPGPUs)

Today: C-like languages (*e.g.*, CUDA, OpenCL)

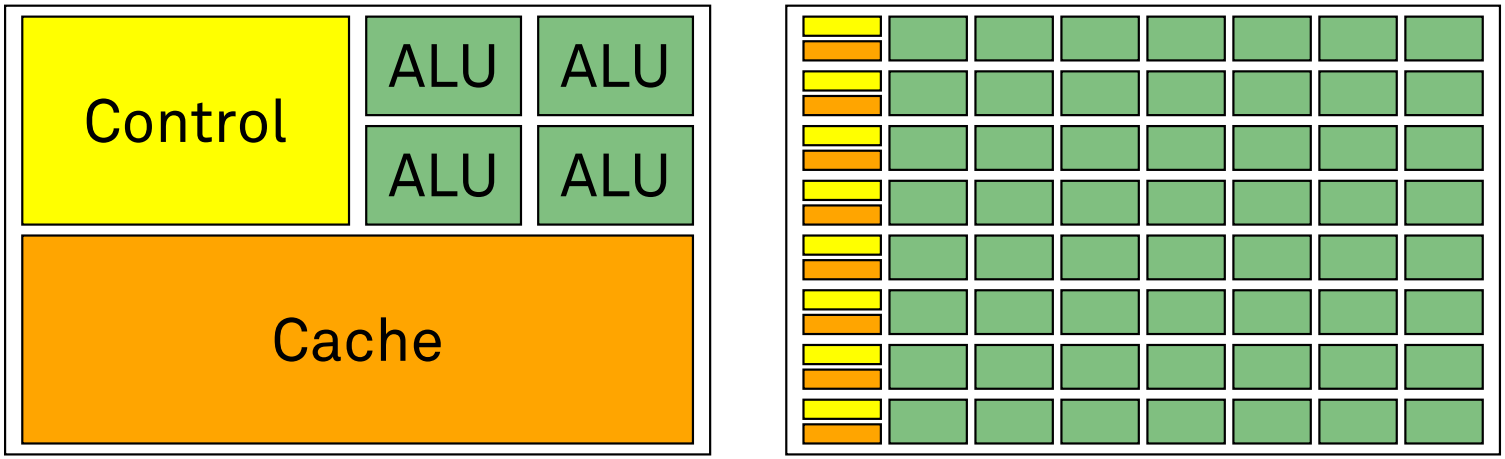
General-Purpose GPUs (GPGPUs)

Original GPU design based on graphics pipeline not flexible enough.

→ geometry shaders idle for pixel-heavy workloads and vice versa

→ **unified model** with general-purpose cores

Thus: Design inspired by CPUs, but different



Rationale: Optimize for **throughput**, not for **latency**.

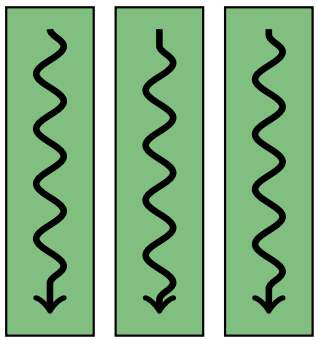
CPUs vs. GPUs

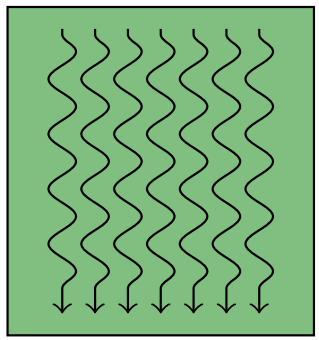
CPU: task parallelism

- relatively heavyweight threads
- 10s of threads on 10s of cores
- each thread managed explicitly
- threads run different code

GPU: data parallelism

- lightweight threads
- 10,000s of threads on 100s of cores
- threads scheduled in batches
- all threads run same code
  - → SPMD, single program, multiple data



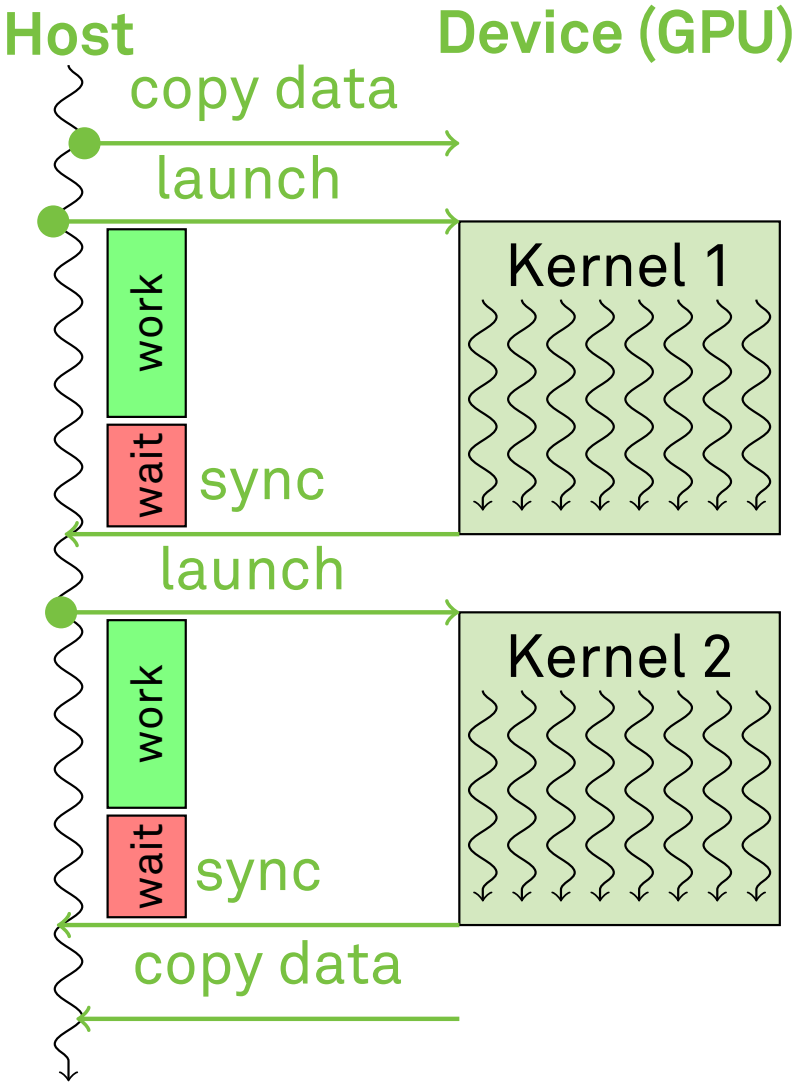


Threads on a GPU

To handle 10,000s of threads efficiently, keep things simple.

- Don't try to **reduce** latency, but **hide** it.
  - → **Large thread pool** rather than caches
    (This idea is similar to SMT in commodity CPUs.)
- Assume **data parallelism** and restrict **synchronization**.
  - → Threads and small **groups** of threads use local memories.
  - → Synchronization only within those groups (more later).
- Hardware **thread scheduling** (simple, in-order).
  - → Schedule threads in **batches** ($\rightsquigarrow$ "warps").

OpenCL Computation Model



- Host system and **co-processor** (GPU is only one possible co-processor.)
- Host triggers
  - data copying host ↔ co-processor,
  - invocations of **compute kernels.**
- Host interface: **command queue.**

Processing Model: (Massive) Data Parallelism

A **traditional loop**

```
for (i = 0; i < nitems; i++)
    do_something(i);
```

becomes a **data parallel kernel invocation** in OpenCL ($\rightsquigarrow$ `map`):

```
status = clEnqueueNDRangeKernel (
           commandQueue,
           do_something_kernel, …, &nitems, …);
```

```
__kernel void do_something_kernel(…) {
    int i = get_global_id(0);
    …;
}
```

Kernel Invocation

Idea: Invoke kernel for each point in a problem domain

- *e.g.*, 1024 $\times$ 1024 image, one kernel invocation per pixel;
  $\rightarrow$ 1,048,576 kernel invocations ("work items").
- Don't worry (too much) about task $\rightarrow$ core assignment or number of threads created; **runtime** does it for you.
- Problem domain can be 1-, 2-, or 3-dimensional.

- Can pass global parameters to all work item executions.
- Kernel must figure out work item by calling `get_global_id()`.

Compute Kernels

OpenCL defines a **C99-like** language for compute kernels.

- Compiled **at runtime** to particular core type.
- Additional set of built-in functions:
  - Context (*e.g.*, `get_global_id()`); synchronization.
  - Fast implementations for special math routines.

```
__kernel void square(__global float *in,
                     __global float *out)
{
    int i=get_global_id(0);
    out[i]=in[i]*in[i];
}
```

⇝ OpenMP: `omp_get_thread_num()`, etc.

OpenCL ↔ OpenMP

Observe how OpenCL's processing model resembles the **OpenMP join/fork model.**

CPU:

- Single thread[4]
- Controls execution ⇝ master thread

GPU:

- **Many** parallel threads that all execute the **same code.**
- **Work Sharing:**
  - → Each data item processed by exactly one GPU thread.
  - → Conceptually: One thread per item.

However:

- **Asynchronous calls**: CPU and GPU may execute in **parallel.**

[4] Of course, multi-threading could be used for the host code, too.

Work Items and Work Groups

Work items may be grouped into **work groups**.

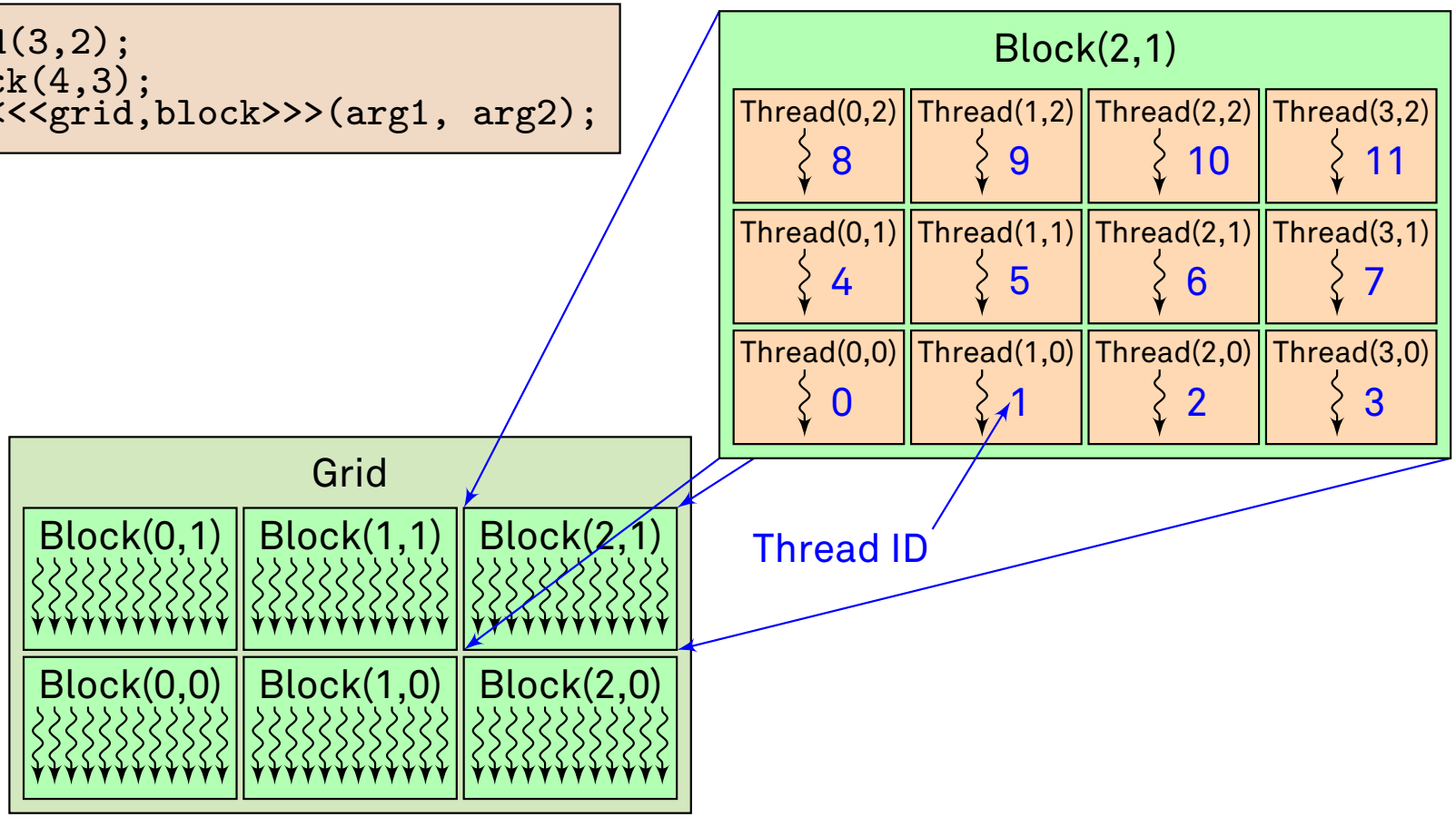
- Work groups $\leftrightsquigarrow$ scheduling batches.
- Synchronization between work items **only** within work groups.
- There is a device-dependent limit on the number of work items per work group (can be determined via `clGetDeviceInfo()`).
- Specify items per group when queuing the kernel invocation.
- All work groups must have same size (within one invocation).

E.g., Problem space: 800×600 items (2-dimensional problem).

$\rightarrow$ Could choose 40×6, 2×300, 80×5, ... work groups.

CUDA Thread Hierarchy

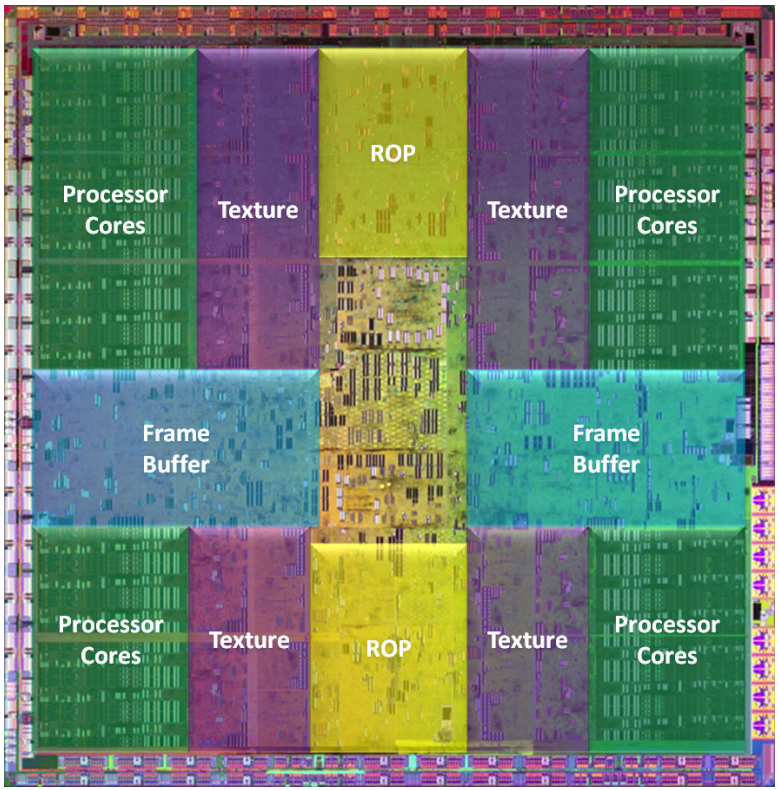
```
dim3 grid(3,2);
dim3 block(4,3);
myKernel<<<grid,block>>>(arg1, arg2);
```



- Block Index: 1- or 2-dimensional
- Thread Index: 1-, 2-, or 3-dimensional (x, y, z)
- Thread ID = $x + yD_x + zD_xD_y$ (block dimension: D_x, D_y, D_z)

Example: NVIDIA GPUs

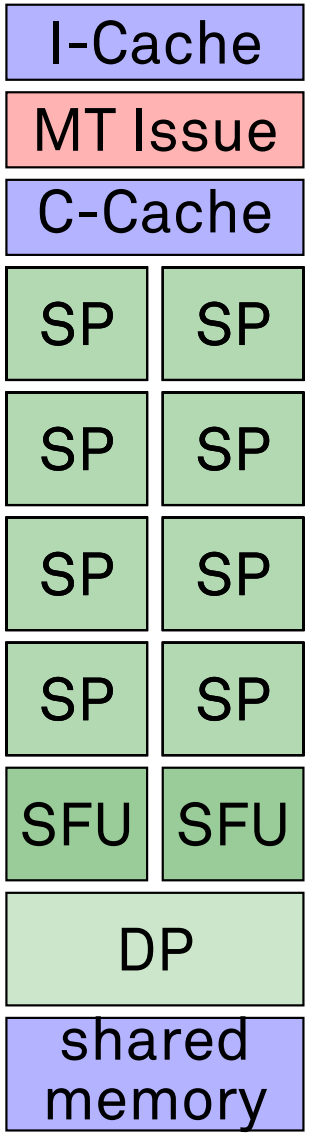
NIVIDA GTX 280



source: www.hardwaresecrets.com

- 10 Thread Processing Clusters
- 10×3 Streaming Multiprocessors
- $10 \times 3 \times 8$ Scalar Processor Cores
  - $\rightarrow$ More like ALUs (↗ slide 39)
- Each Multiprocessor:
  - 16k 32-bit registers
  - 16 kB shared memory
  - up to 1024 threads (may be limited by registers and/or memory)

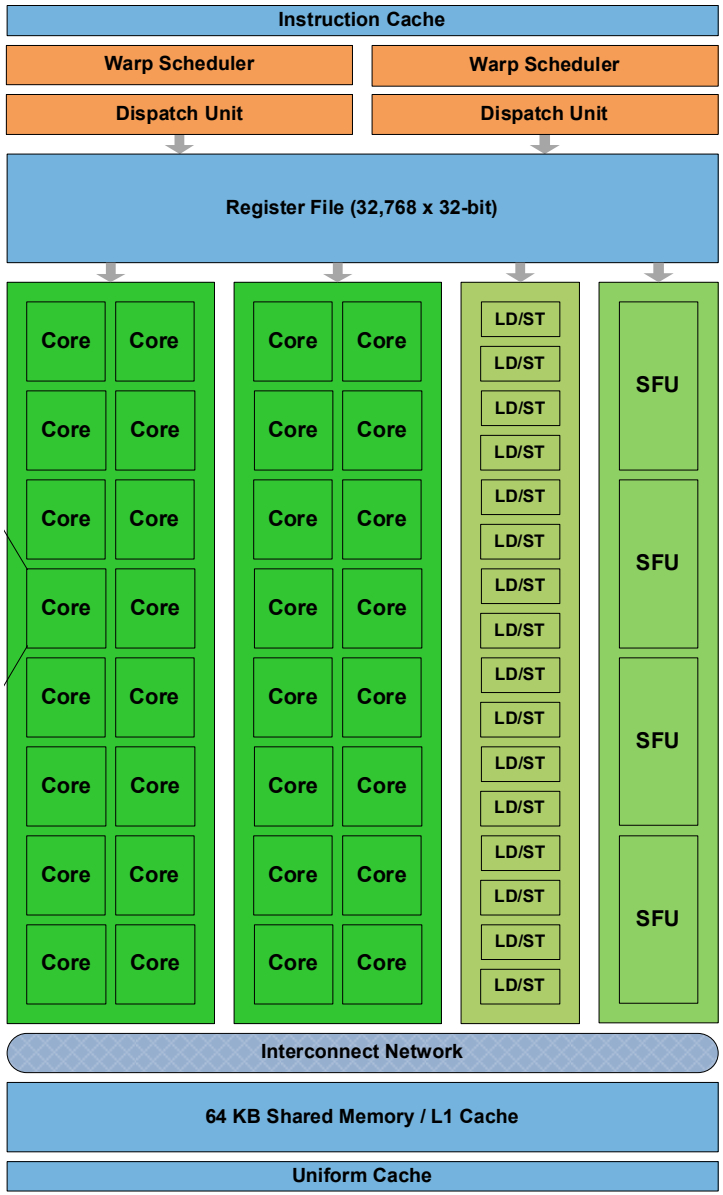
Inside a Streaming Multiprocessor



- 8 Scalar Processors (Thread Processors)
  - single-precision floating point
  - 32-bit and 64-bit integer
- 2 Special Function Units
  - sin, cos, log, exp
- Double Precision unit
- 16 kB Shared Memory

- Each Streaming Multiprocessor: up to 1,024 threads.
- GTX 280: 30 Streaming Multiprocessors
  → 30,720 concurrent threads (!)

Inside a Streaming Multiprocessor: nVidia Fermi



Source: nVidia Fermi White Paper

- 32 “cores” (thread processors) per streaming multiprocessor (SM)
- but fewer SMs per GPU: 16 (vs. 30 in GT200 architecture)
- 512 “cores” total
- “cores” now double-precision-capable

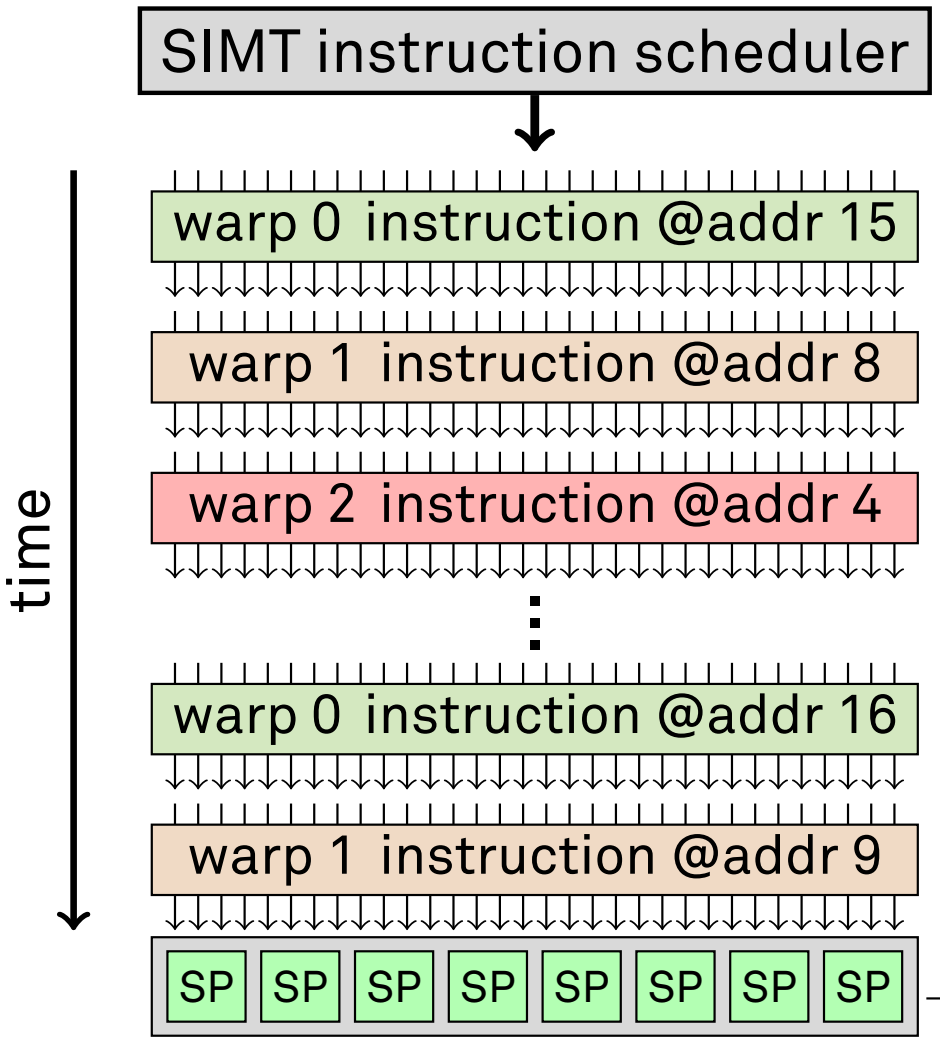
Scheduling in Batches

- In SM threads are scheduled in units of 32, called **warps**.
- **Warp**: Set of 32 parallel threads that start together at the same program address.

warp (dt. Kett- oder Längsfaden)

- For memory access warps are split into **half-warps** consisting of 16 threads
- Warps are scheduled with zero-overhead
- Scoreboard is used to track which warps are ready to execute
- GTX 280: 32 warps per multiprocessor (1024 threads)
- newer cards: 48 warps per multiprocessor (1536 threads)

SPMD / SIMT Processing

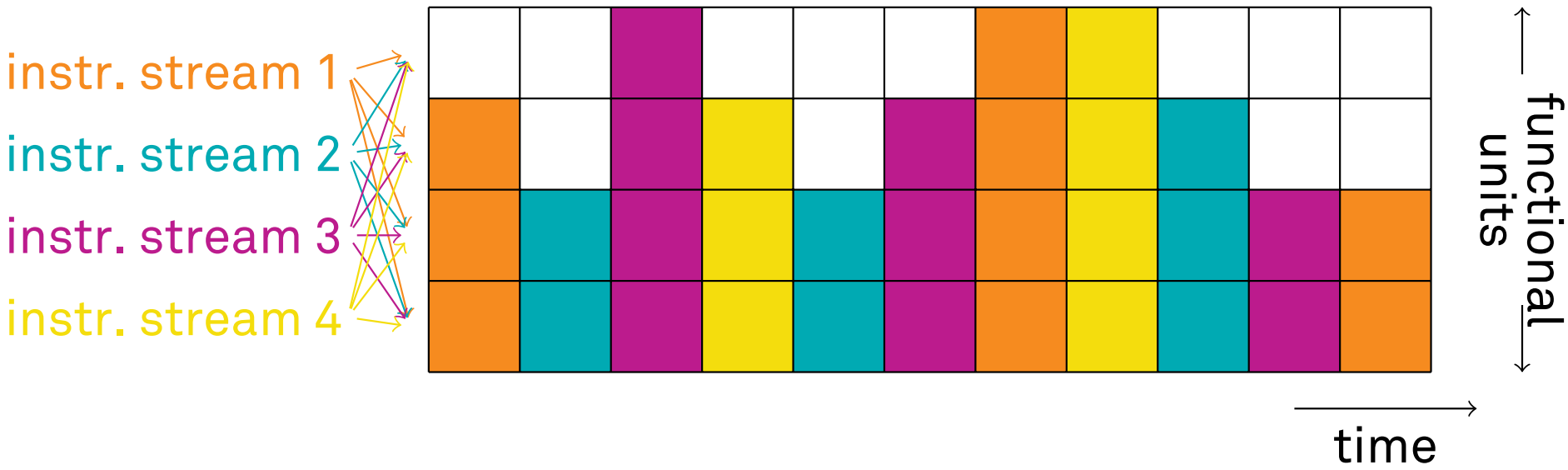


- **SIMT**: Single Instruction, Multiple Threads
- All threads execute the same instruction.
- Threads are split into warps by increasing thread IDs (warp 0 contains thread 0).
- At each time step scheduler selects warp ready to execute (*i.e.*, all its data are available)
- nVidia Fermi: dual issue; issue two warps at once[a]

[a] no dual issue for double-precision instr.

GPU Scheduling: Fine-Grained Multithreading

GPUs implement **fine-grained multithreading** (↗ "Multithreading" chapter)



But:

- Scheduling decisions here affect **entire warps.**
- GPUs have **more functional units** ("scalar processors").
- Functional units cannot be scheduled arbitrarily

  The above illustration is somewhat misleading in that regard.

Warps and Latency Hiding

Some runtime characteristics:

- Issuing a warp instruction takes **4 cycles** (8 scalar processors).
- Register write-read latency: **24 cycles.**
- Global (off-chip) memory access: $\approx$ **400 cycles.**

Threads are executed **in-order.**

$\rightarrow$ **Hide latencies** by executing other warps when one is paused.

$\rightarrow$ Need **enough warps** to fully hide latency.

E.g.,

- Need $^{24}/_{4} = 6$ warps to hide register dependency latency.
- Need $^{400}/_{4} = 100$ instructions to hide memory access latency. If every 8th instruction is a memory access, $^{100}/_{8} \approx 13$ warps would be enough.

Resource Limits

Ideally: 32 warps per multiprocessor (1024 threads)

But: Various **resource limits**

- limited number of 32-bit **registers** per multiprocessor
  E.g.: 11 registers per thread, 256 threads/items per work group.
  CUDA compute capability 1.1: 8,192 registers per multiprocessor.
  $\rightarrow$ max. 2 work groups per multiprocessor ($3 \times 256 \times 11 > 8192$)
- 48 kB **shared memory** per multiprocessor (compute cap. 2.0)
  E.g.: 12 kB per work group
  $\rightarrow$ max. 4 work groups per multiprocessor
- 8 **work groups** per multiprocessor; max. 512 work items per work group
- Additional constraints: **branch divergence**, **memory coalescing.**

Occupancy calculation (and choice of work group size) is complicated!

Work Groups (NVIDIA: “Blocks”)

Work Groups (on NVIDIA GTX 280):

- Work group can contain up to 512 threads
- A work group is scheduled to exactly one SM
  - Central round-robin distribution
  - Remember: Synchronization and collaboration through shared memory only within work group
- Each SM can execute up to 8 work groups
  - Actual number depends on register and shared memory usage
  - Combined shared memory usage of all work groups $\leq$ 16 kB

Characteristics of **one** particular piece of hardware, not part of the OpenCL specification!

Executing a Warp Instruction

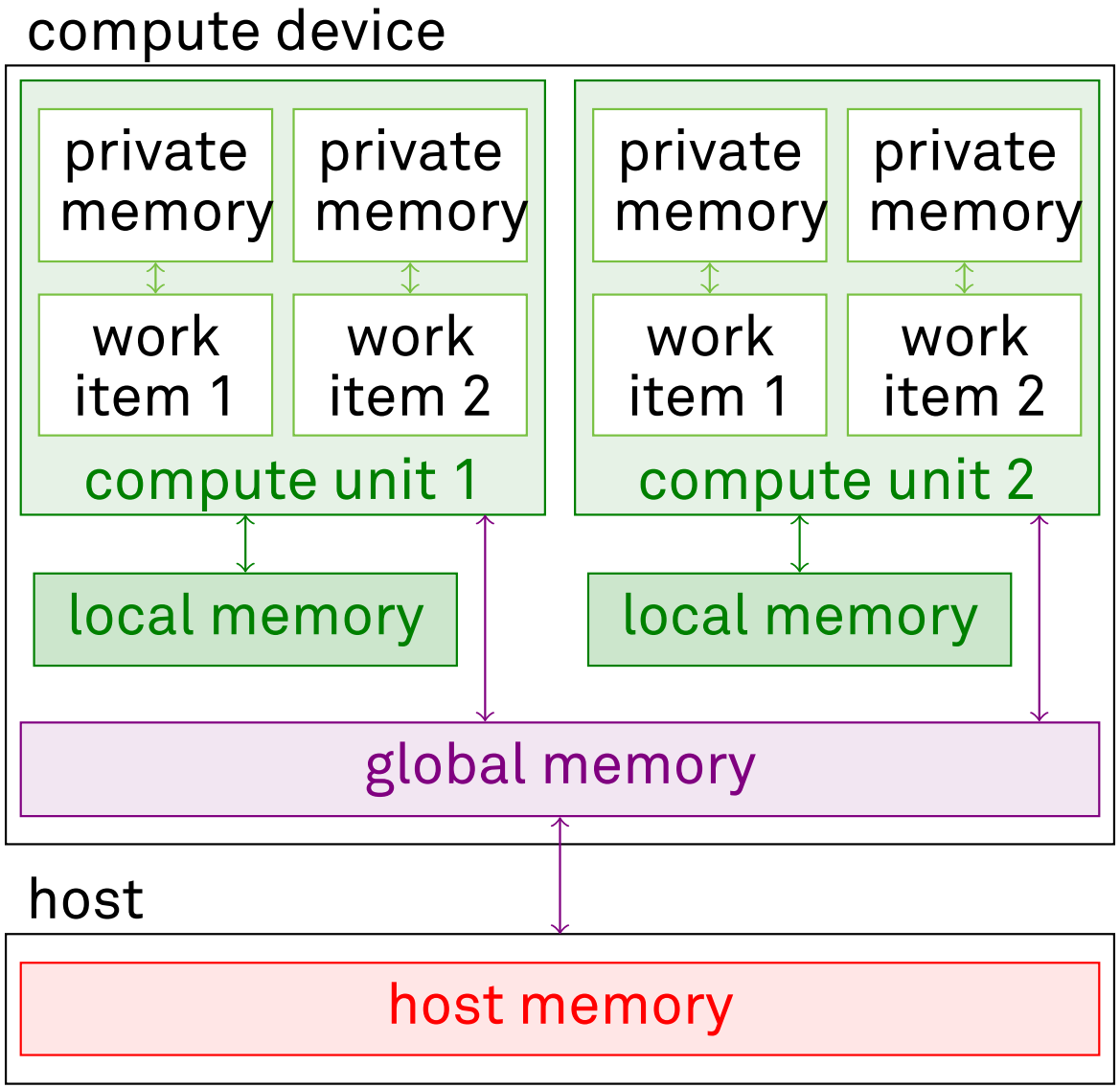
Within a warp, **all threads** execute **same instructions.**

→ What if the code contains **branches**?

```
if (i < 42)
    then_branch();
else
    else_branch();
```

- If **one** thread enters the branch, **all** threads have to execute it.
  - → Effect of branch execution discarded if necessary.
  - ⇝ Predicated execution
- This effect is called **branch divergence.**
- **Worst case:** all 32 threads take a different code path.
  - → Threads are effectively executed **sequentially.**

OpenCL Memory Model



OpenCL ↔ Cuda

NVIDIA/Cuda uses a slightly different terminology:

OpenCL	Cuda	
private memory	registers	on-chip
local memory	shared memory	on-chip
global memory	global memory	off-chip

On-chip memory is **significantly** faster than off-chip memory.

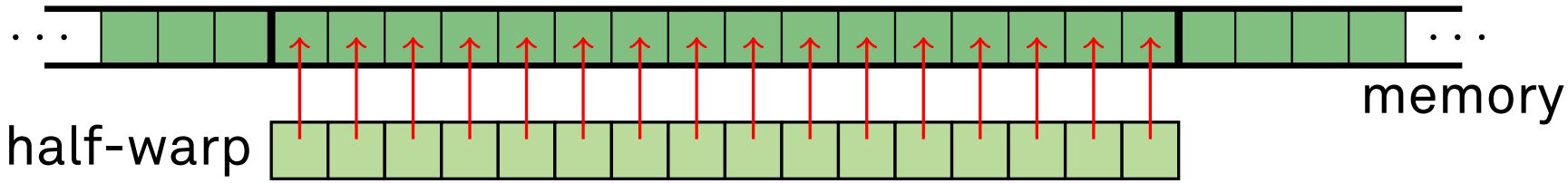
Memory Access Cost (Global Memory; NVIDIA)

Like in CPU-based systems, GPUs access **global memory** in chunks (32-bit, 64-bit, or 128-bit **segments**).

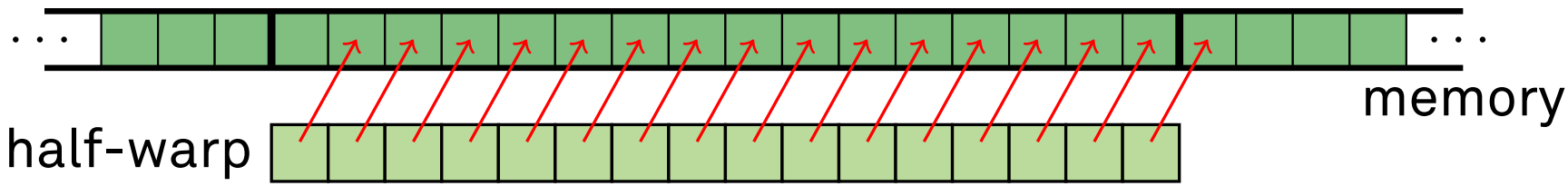
→ Most efficient if accesses by threads in a half-warp **coalesce.**

E.g., NVIDIA cards with compute capability 1.0 and 1.1:

- Coalesced access → 1 memory transaction



- Misaligned → 16 memory transactions (2 if comp. capability ≥ 1.2)



Coalescing Example

Example to demonstrate coalescing effect:

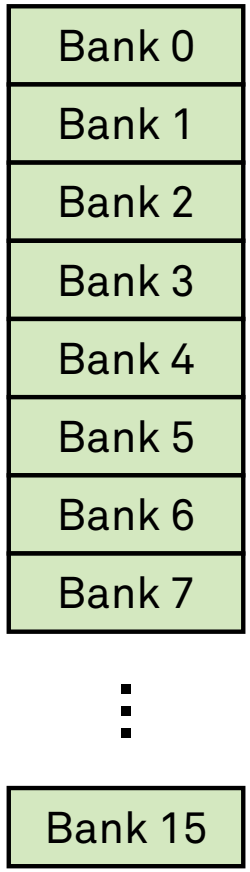
```
__kernel void
copy(__global unsigned int *din,
     __global unsigned int *dout,
     const unsigned int offset)
{
    int i=get_global_id(0);
    dout[i]=din[i+offset];
}
```

Strided access causes similar problems!

Shared Memory (NVIDIA)

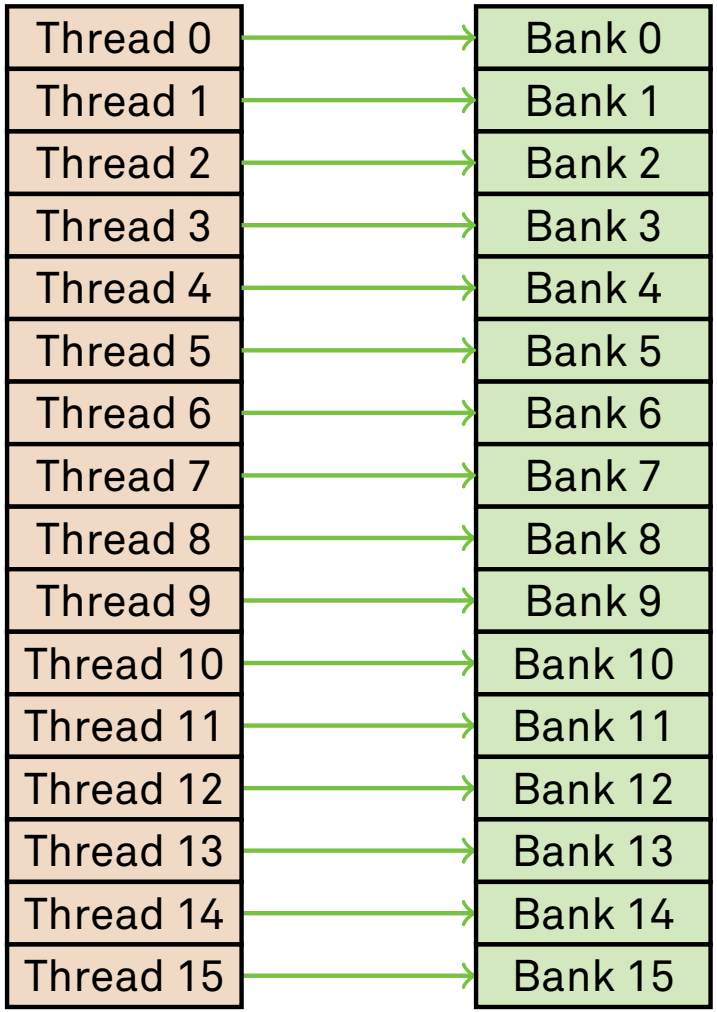
Shared memory (OpenCL: “local memory”):

- **fast** on-chip memory (few cycles latency)
- throughput: **38–44 GB/s per multiprocessor**(!)
- partitioned into **16 banks**
  - → 16 threads (1 **half-warp**) can access shared memory simultaneously **if and only if** they all access a different bank.
  - → Otherwise a **banking conflict** will occur.
- Conflicting accesses are **serialized**
  - → (potentially significant) **performance impact**

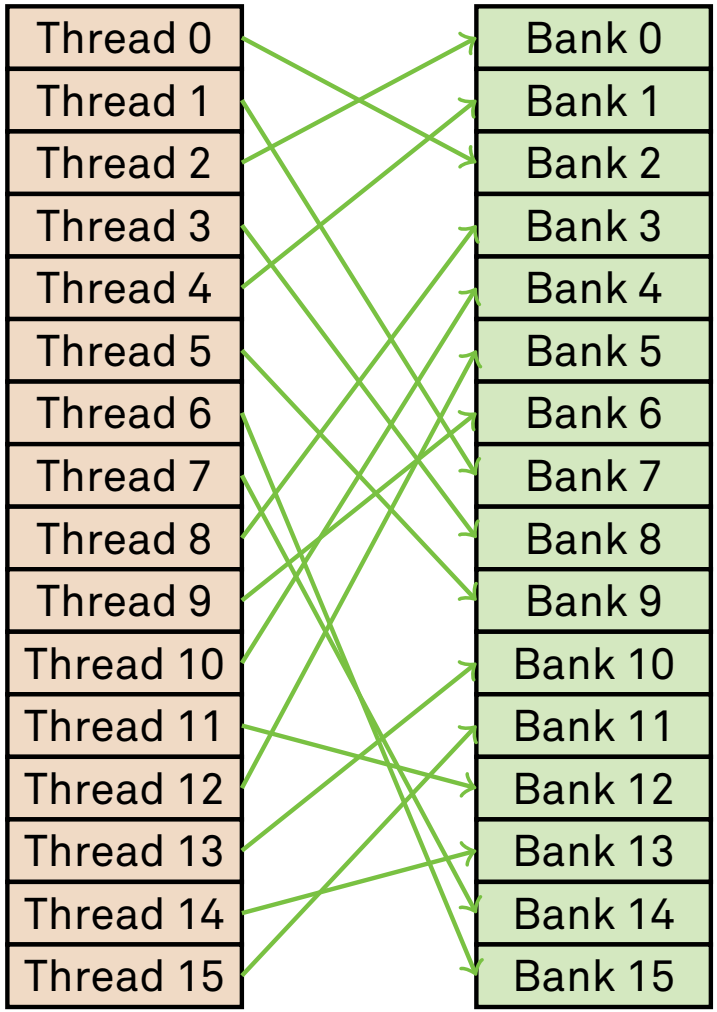


Bank Conflicts to Shared Memory

stride width: 1 word



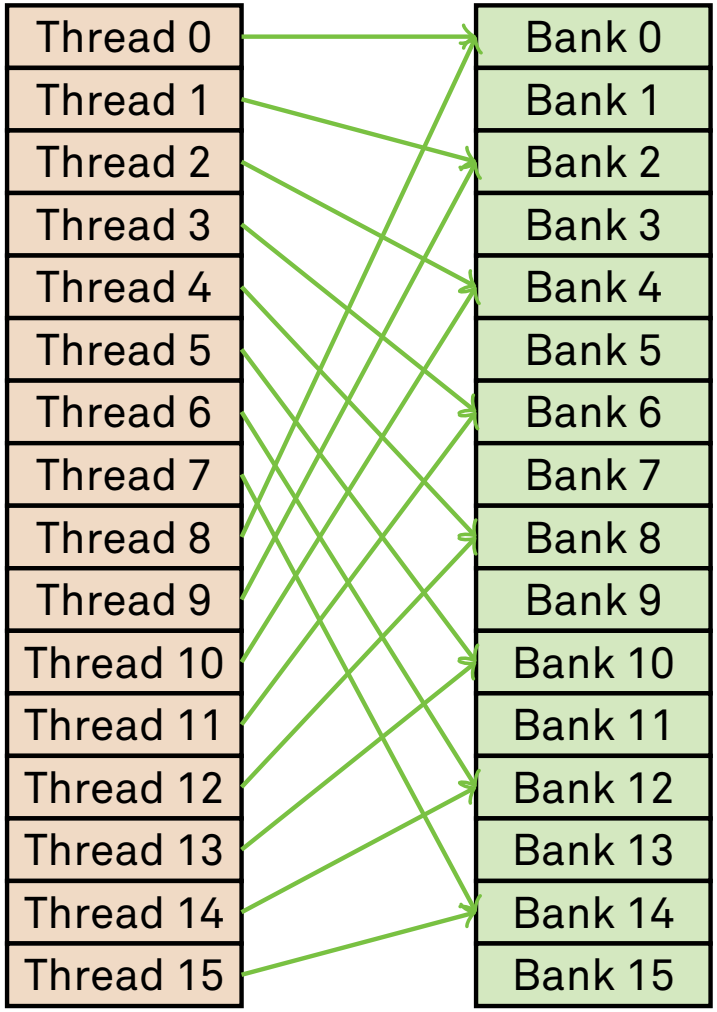
→ **no bank conflicts**



→ **no bank conflicts**

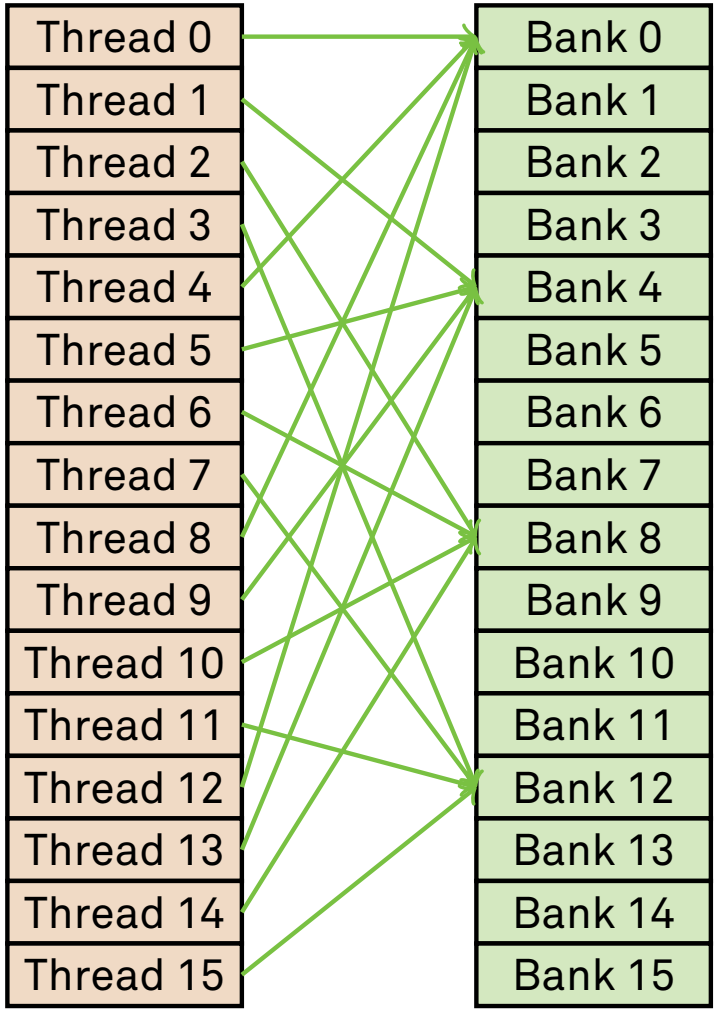
Bank Conflicts to Shared Memory (cont.)

stride width: 2 words



→ **2-way bank conflicts**

stride width: 4 words

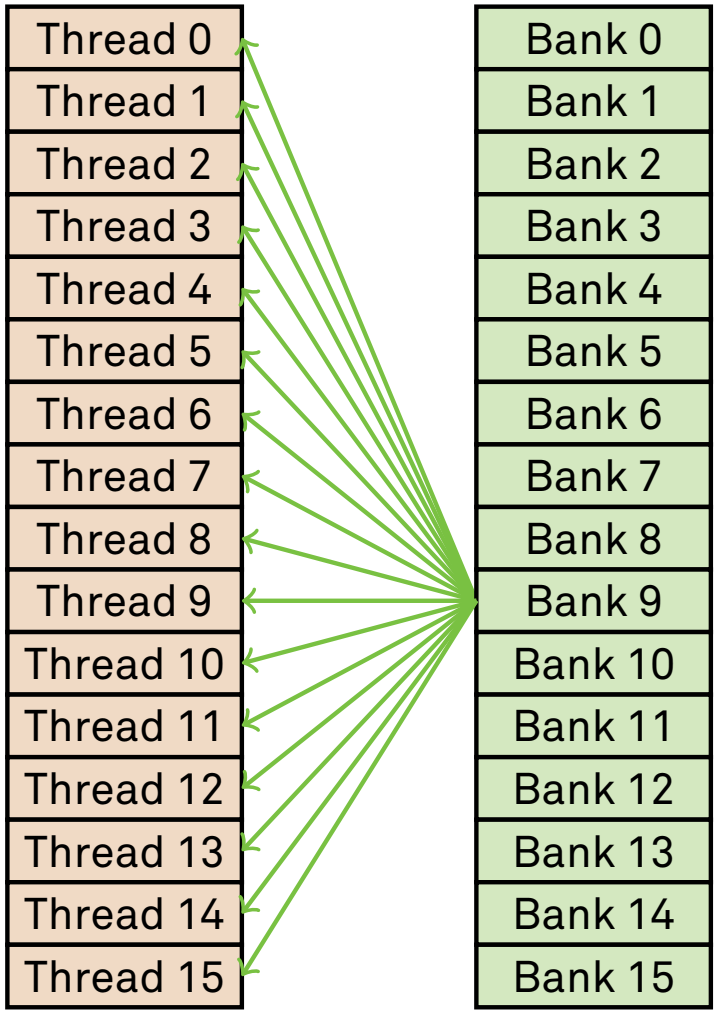


→ **4-way bank conflicts**

Exception: Broadcast Reads

Broadcast reads do **not** lead to a bank conflict.

- All threads must read the **same** word.



Thread Synchronization

Threads may use built-in functions to synchronize **within** work groups.

- `barrier ( flags )` Block until all threads in the group have reached the barrier. Also enforces memory ordering.
- `mem_fence ( flags )` Enforce memory ordering: all memory operations are committed before thread continues.

```
for (unsigned int i = 0; i < n; i++)
{
    do_something();
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

If barrier occurs in a **branch**, same branch must be taken by **all threads** in the group (danger: deadlocks or unpredictable results).

Synchronization Across Work Groups

To synchronize **across** work groups,

- use **in-order** command queue and queue multiple kernel invocations from the host side
  - → Can also queue **markers** and **barriers** to the command queue.

or

- use OpenCL **event mechanism.**
  - → Can also synchronize host ↔ device and kernel executions in **multiple command queues.**

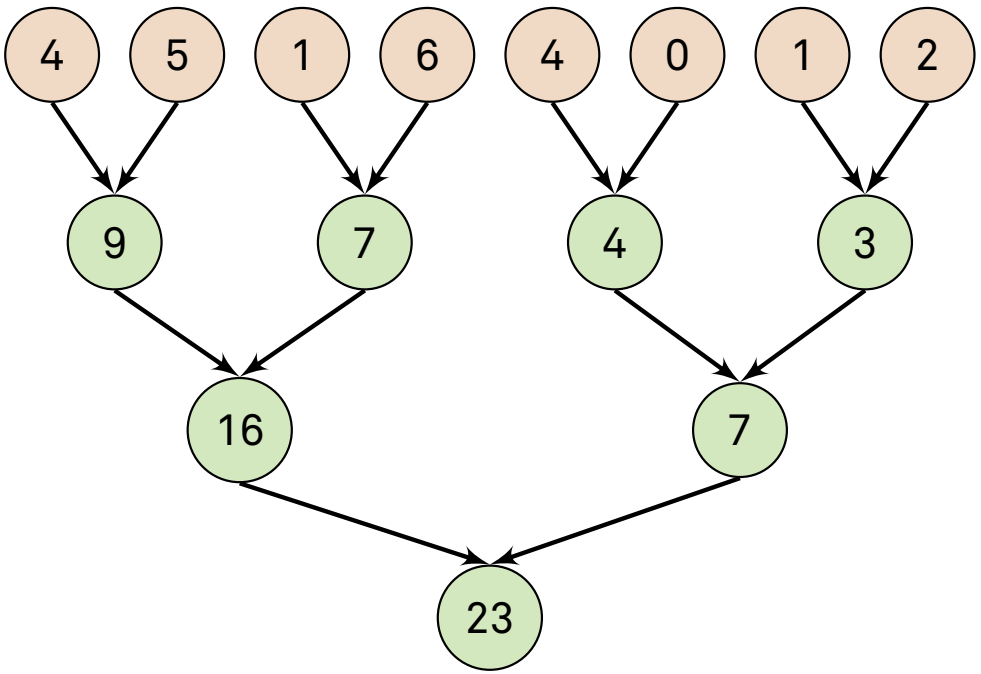
To wait on host side until all queued commands have been completed, use `clFinish` (*command queue*).

Parallel Data Reduction

Example: Parallel Reduction ($x_1 + x_2 + \cdots + x_n$)

→ Based on Mark Harris' sample code in the NVIDIA CUDA SDK.

→ We'll use CUDA syntax in the following.

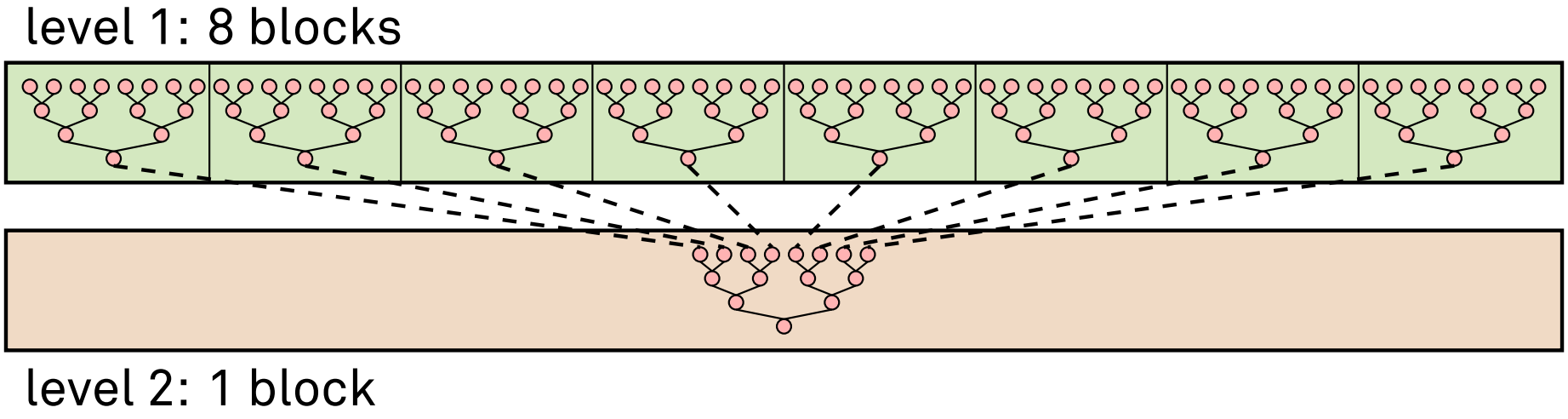


- **Goal:** compute sum of array elements in parallel
- Tree-based approach with concurrent operations
  - Level 1: 4 sums in parallel
  - Level 2: 2 sums in parallel
  - Level 3: 1 sum

Synchronization

- Large array $\rightarrow$ many work groups to keep multiprocessors busy
- How to aggregate sum between threads? $\rightarrow$ **synchronization** required
- There is no **global** synchronization
  - There is, through atomic instruction but this would result in a bottleneck.
  - And how to synchronize between levels?
  - Threads synchronize within group
- How many work groups with how many threads?
- Do “global” synchronization implicitly by invoking multiple kernels, e.g., one per level.

Kernel Decomposition



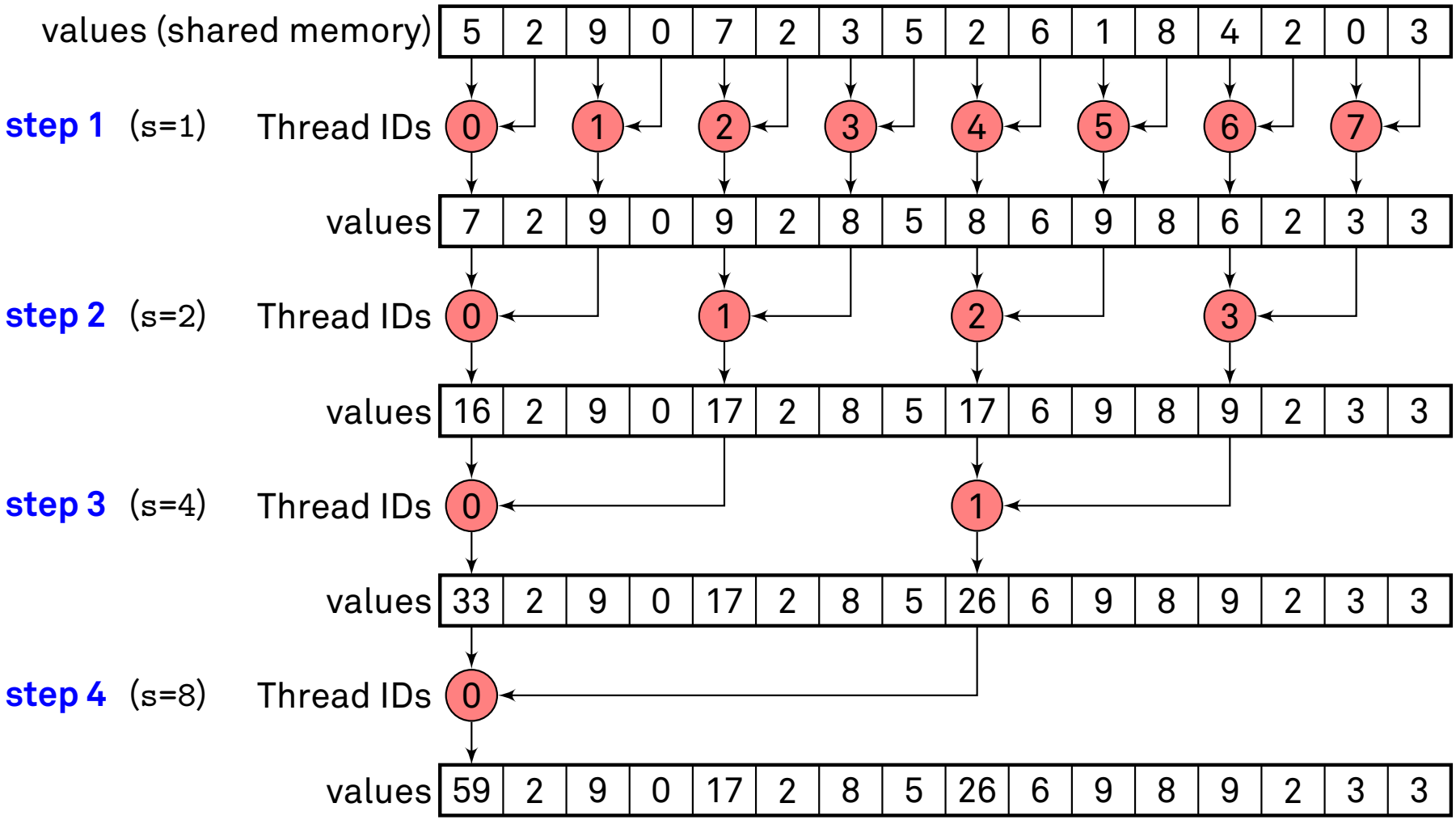
- Use multiple kernel invocations to synchronize computation
- Kernel code is the same for all invocations
- Simplified illustration: use at least 32 threads per block

Reduction Kernel — Implementation #1

```
__global__ void reduce(int *input, int *output) {
  extern __shared__ int sdata[];
  unsigned int tid = threadIdx.x; /* thread ID */

  /* index into data array for this thread */
  unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

  /* load data from global memory into shared memory */
  sdata[tid] = input[idx];
  __syncthreads(); /* wait for everybody else in the block */

  /* do the block-internal recursion */
  for (unsigned int s=1; s<blockDim.x; s *= 2) {
      int i = 2*s*tid;
      if (i < blockDim.x) sdata[i] += sdata[i+s];
      __syncthreads(); /* wait for everybody else in the block */
  }
  /* thread 0 writes result of sum into global memory */
  if (tid == 0) output[blockIdx.x] = sdata[0];
}
```

Implementation #1

For illustration `dimBlock.x=16`:



⇒ Strides 2, 4, and 8 result in 2-, 4- and 8-way bank conflicts.

Sequential Addressing — Implementation #2

Get rid of strided access

```
for (unsigned int s=1; s<blockDim.x; s *= 2) {
  int i = 2*s*tid;
  if (i < blockDim.x) {
      sdata[i] += sdata[i+s];
  }   __syncthreads(); /* wait for everybody else in the block */
}
```

by replacing it as:

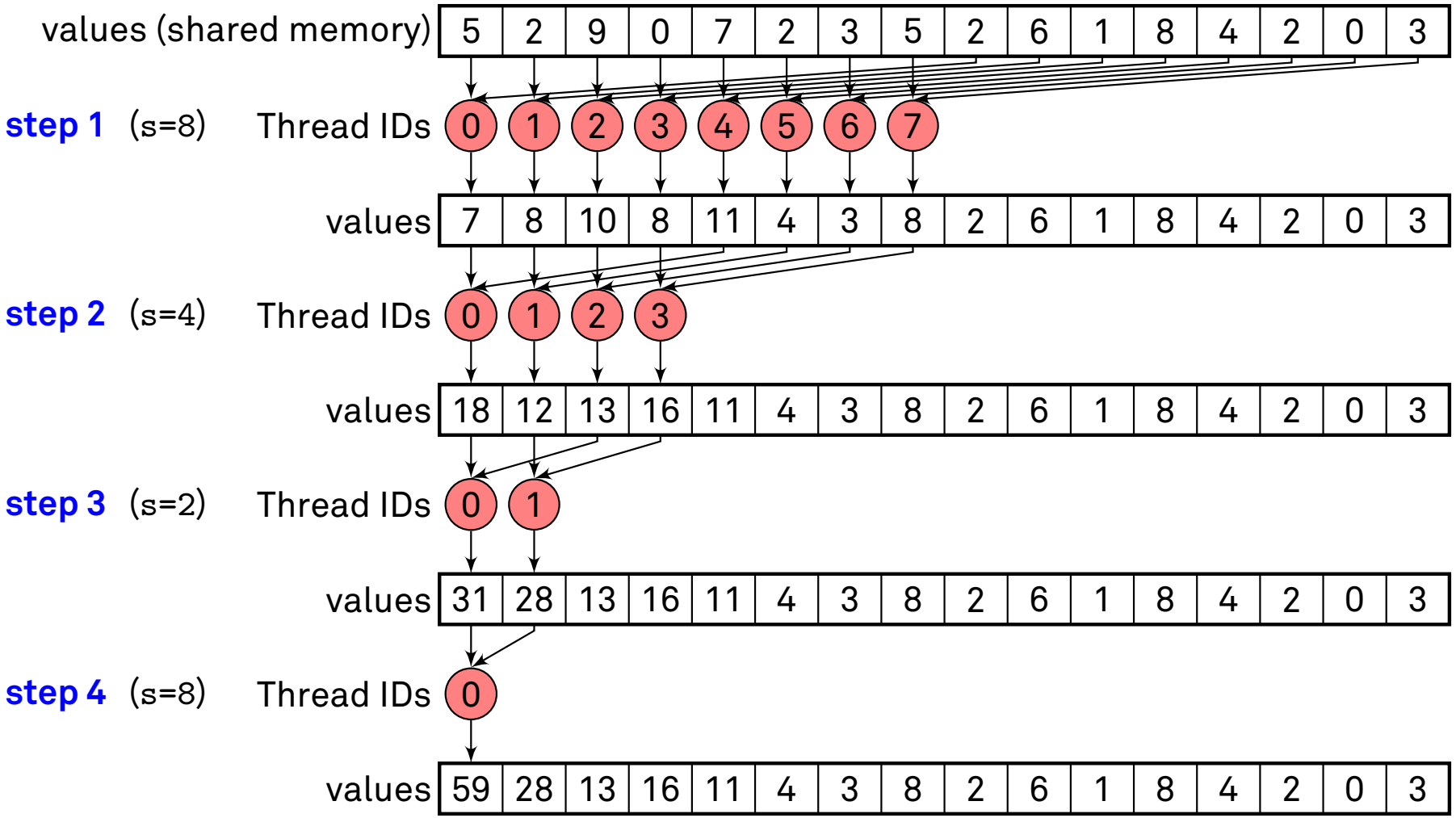
```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if (tid < s) {
      sdata[tid] += sdata[tid+s];
  }
  __syncthreads(); /* wait for everybody else in the block */
}
```

Note: `__syncthreads()` *is outside of "if". Why?*

Reduces kernel time (GPU) from 6832.03 μs down to 3691.97 μs
$\rightarrow$ Speedup: 1.85$\times$ (for 8 MiB elements & 256 threads/group)

Implementation #2

For illustration `dimBlock.x=16`:



⇒ Sequential addressing is conflict free.

Increasing the Number of Active Threads

- 50% of the threads do not even survive the first iteration!

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
  if (tid < s) {
      sdata[tid] += sdata[tid+s];
  }
  __syncthreads(); /* wait for everybody else in the block */
}
```

- Half the number of work groups
- Load two elements and add them right away

Load and Add — Implementation #3

Half the number of work groups and replace single load

```
unsigned int tid = threadIdx.x; /* thread ID */
/* index into data array for this thread */
unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

/* load data from global memory into shared memory */
sdata[tid] = input[idx];
__syncthreads();
```

with two loads and an add:

```
unsigned int tid = threadIdx.x; /* thread ID */
/* index into data array for this thread */
unsigned int idx = blockIdx.x*(blockDim.x*2) + threadIdx.x;

/* load data from global memory into shared memory */
sdata[tid] = input[idx] + input[idx+blockDim.x];
__syncthreads();
```

- Reduces kernel time (GPU) from 3691.97μs down to 1984.45μs
  Speedup: 1.86$\times$ (for 8 MiB elements & 256 threads/group)

Load and Add — Implementation #3

Half the number of work groups and replace single load

```
unsigned int tid = threadIdx.x; /* thread ID */
/* index into data array for this thread */
unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;

/* load data from global memory into shared memory */
sdata[tid] = input[idx];
__syncthreads();
```

with two loads and an add:

```
unsigned int tid = threadIdx.x; /* thread ID */
/* index into data array for this thread */
unsigned int idx = blockIdx.x*(blockDim.x*2) + threadIdx.x;

/* load data from global memory into shared memory */
sdata[tid] = input[idx] + input[idx+blockDim.x];
__syncthreads();
```

- Reduces kernel time (GPU) from 3691.97μs down to 1984.45μs
  Speedup: 1.86$\times$ (for 8 MiB elements & 256 threads/group)

Loop Unrolling

- As `s` is reduced by half in every step, the number of active threads decreases
  - Active = threads that do work
- When $s \leq 32$ there is only one warp left
- Instructions within a warp behave like SIMD instructions
  - (only if warps are not serialized, i.e., no bank conflicts and no branching)
- For $s \leq 32$
  - No `__syncthreads()` needed
  - `if (tid < s)` $\rightarrow$ `if (tid < 32)`
- Unroll last 6 iterations

Unrolling Last Warp — Implementation #4

```
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
  if (tid < s) {
      sdata[tid] += sdata[tid+s];
  }
  __syncthreads(); /* wait for everybody else in the block */
}
if (tid<32) {
  sdata[tid] += sdata[tid+32];
  sdata[tid] += sdata[tid+16];
  sdata[tid] += sdata[tid+ 8];
  sdata[tid] += sdata[tid+ 4];
  sdata[tid] += sdata[tid+ 2];
  sdata[tid] += sdata[tid+ 1];
}
```

- Reduces kernel time (GPU) from 1984.45μs down to 1276.8μs
  Speedup: 1.55$\times$ (for 8 MiB elements & 256 threads/group)

Additional Optimizations

- Overall Speedup through implementations #1 $\rightarrow$ #4: 5.35$\times$
  - Implementation #1: 6832.02μs
  - Implementation #4: 1276.80μs
- Complete unrolling for work group sizes of power of 2
- Each thread loads and sums multiple elements into shared memory
- *For details see documentation for NVIDIA CUDA SDK sample code,* `reduction` *example.*

GPUs

To summarize,

- GPUs provide **high degrees of parallelism** that can be programmed using a **high-level language.**
- **Idea:** ~~latency~~ → throughput

But:

- GPUs are not simply “multi-core processors.”
- Unleashing their performance requires efforts and care for details.

Also note that

- GPUs provide lots of **Giga-FLOPS**.
  → Useful (only) if your code actually needs FLOPS.