GPGPU with SYCL

NPRG042: Programming in Parallel Environment

Martin Kruliš

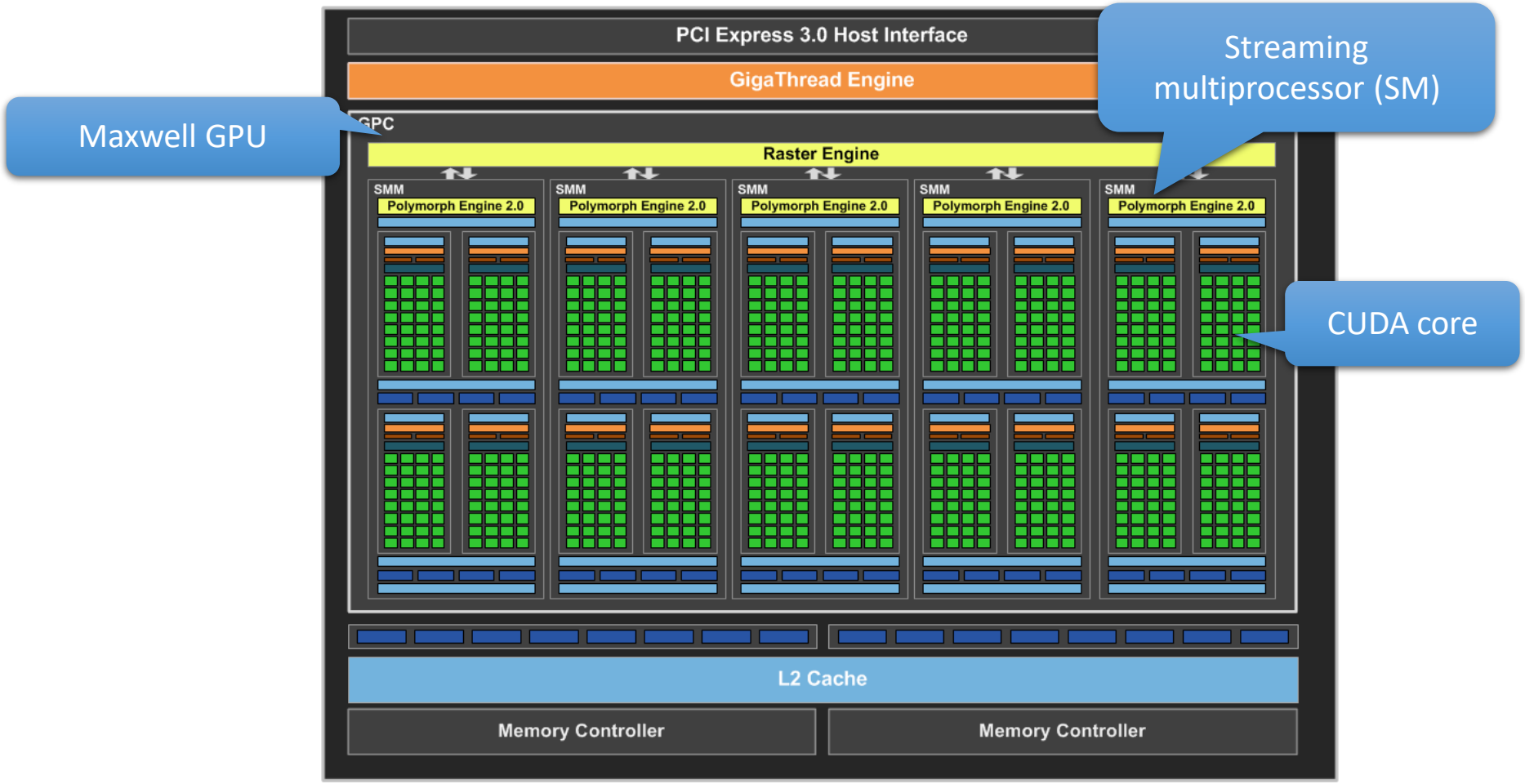
GPU vs. CPU

- CPU
  - Few cores per chip
  - General purpose cores
  - Processing different threads
  - Huge caches to reduce memory latency
    - Locality of reference problem

- GPU
  - Many cores per chip
  - Cores specialized for numeric computations
  - SIMT thread processing
  - Huge amount of threads and fast context switch
    - Results in more complex memory transfers

Architecture Convergence

GPU Architecture

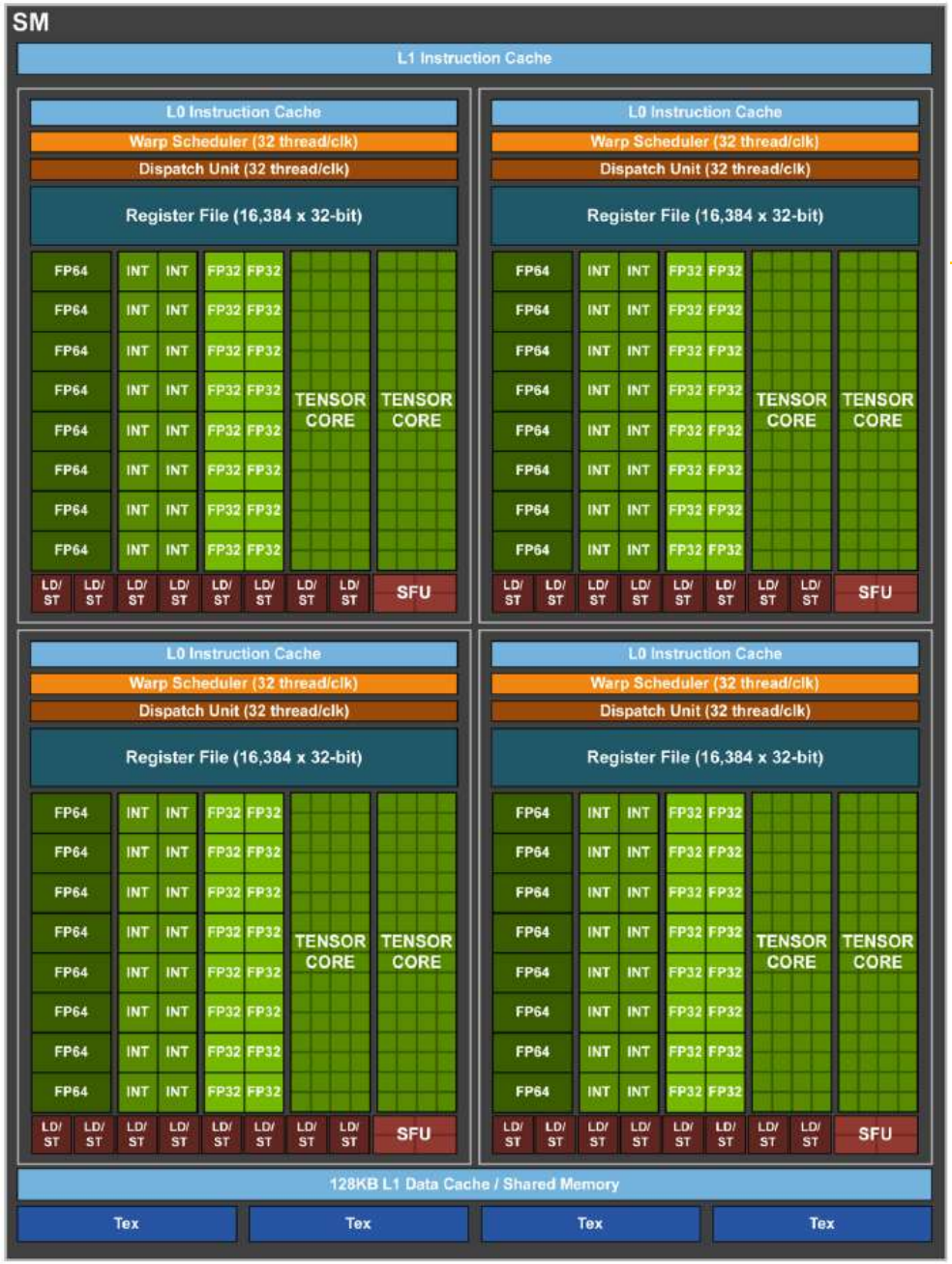


GPU Architecture

- Maxwell Architecture
  - 4 identical parts
    - 32 cores
    - 64 kB shared memory
    - 2 instruction schedulers
  - CC 5.0
  - SMM
    (Streaming Multiprocessor - Maxwell)

GPU Architecture

- Volta SM
  - 7.x CC
  - 8x tensor core
  - (64 FMA/clock each)



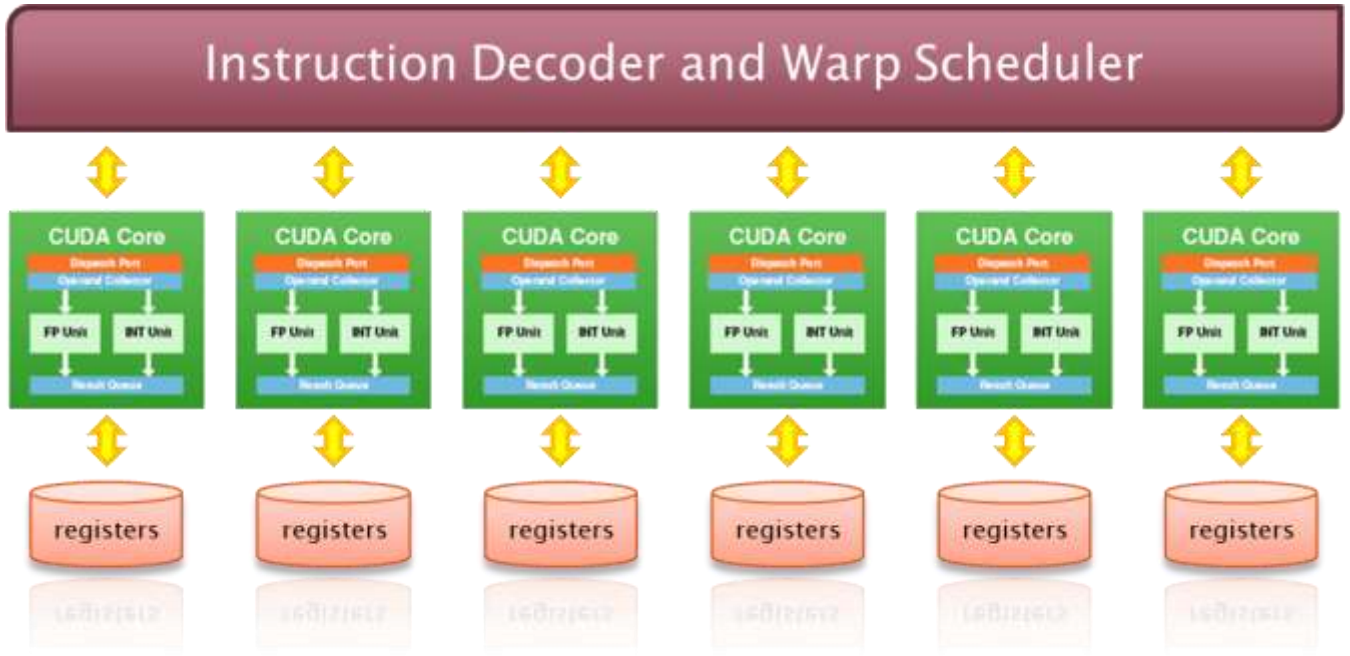
Execution Model

- Data Parallelism
  - Many data elements are processed concurrently by the same routine
  - GPUs are designed under this particular paradigm
  - Also have limited ways to express task parallelism

- Threading Execution Model
  - One function (kernel) is executed in many threads (work items)
  - Much more lightweight than the CPU threads
  - Threads are grouped into blocks (work groups) of the same size

Single Instruction Multiple Threads

- Single Instruction Multiple Threads
  - All cores are executing the same instruction
  - Each core has its own set of registers

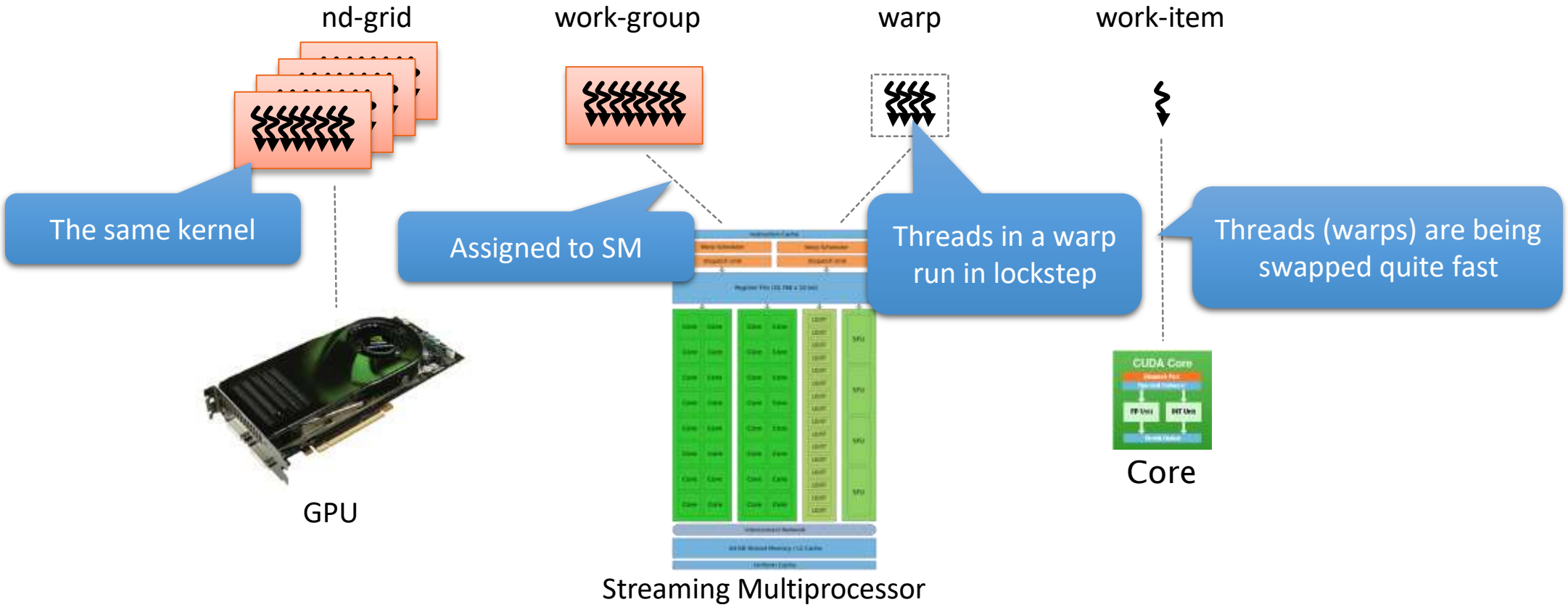


SIMT vs. SIMD

- Single Instruction Multiple Threads
  - Width-independent programming model
  - Serial-like code
  - Achieved by hardware with a little help from the compiler
  - Allows code divergence
- Single Instruction Multiple Data
  - Explicitly expose the width of the SIMD vector
  - Special instructions
  - Generated by the compiler or directly written by a programmer
  - Code divergence is usually not supported or tedious

Workload Distribution

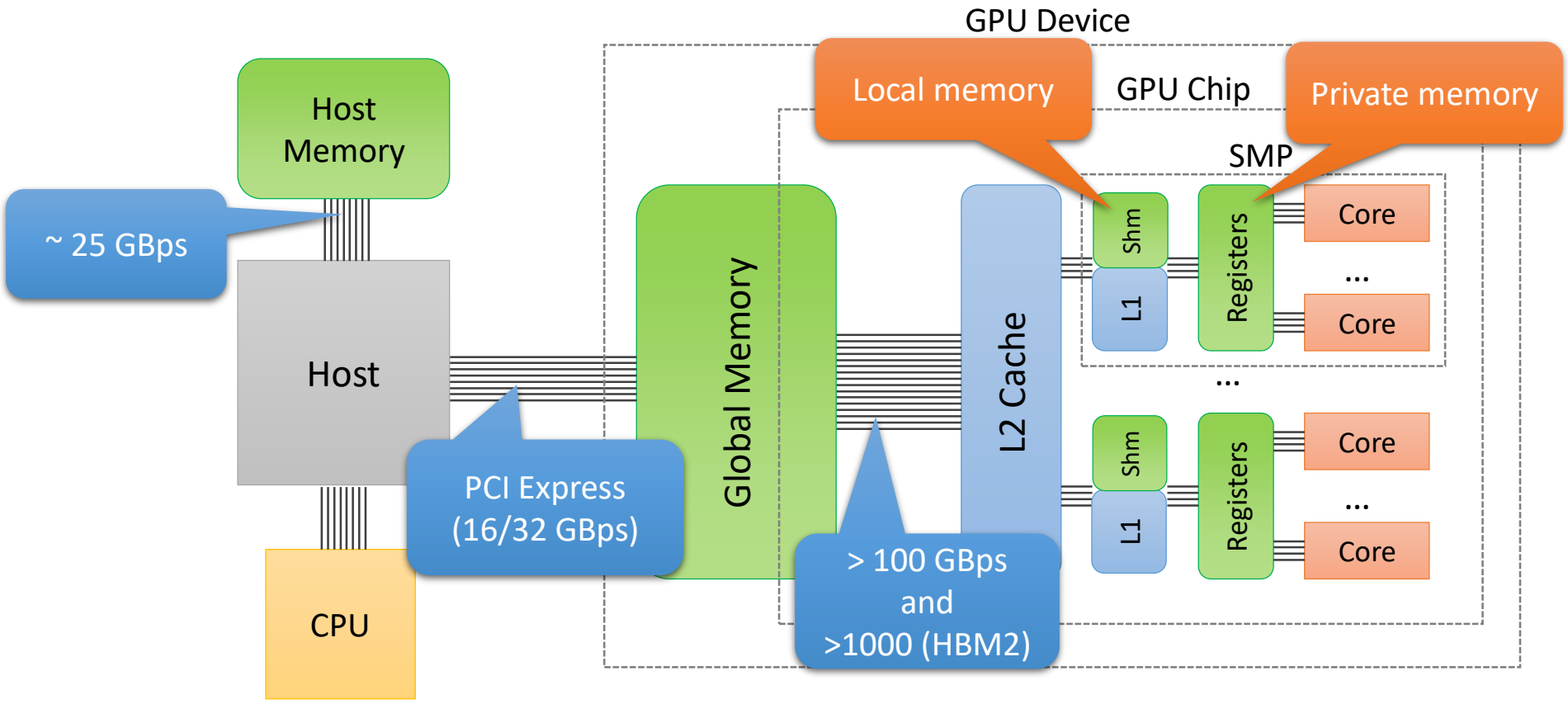
- Nd-grid and work elements assignment



Lockstep Execution

- Lockstep
  - Multiple threads (warp) execute the same instruction simultaneously
  - Simplifies the design of the SMs
  - The programmer needs to know
    - When optimizing the code
    - Not expressed explicitly in SYCL
    - Cannot rely on that (e.g., for sync.)
  - Divergence is possible
    - But not very efficient
  - Volta significantly loosens lockstep
    - Threads may diverge in hardware

Memory


GPU Device
Local memory
GPU Chip
Private memory
Host Memory
SMP
~ 25 GBps
Shm
Registers
Core
L1
...
Core
Host
Global Memory
L2 Cache
PCI Express (16/32 GBps)
Shm
Registers
Core
L1
...
Core
> 100 GBps and >1000 (HBM2)
CPU

Memory

- Host-device transfers
  - Implicit (declared by accessors, handled by the scheduler) – no control
  - Explicit handler.copy(src, dest);
- Registers
  - All local variables of the kernel
  - Limited size, registry spilling (cached in L1)
- Local (shared) memory
  - Special memory shared among work-group
  - Very fast, used for (manual) caching or data exchange (items in group)

```
auto shm = sycl::accessor<int, 1, sycl::access::target::local>(…);
```

Code Example

```
static auto exh =
  [](sycl::exception_list l)
{
  for (std::exception_ptr const &e : l){
    try {
      std::rethrow_exception(e);
    }
    catch (std::exception const &e) {
      std::terminate();
    }
  }
};

std::vector<int> A, B, C;
auto selector = sycl::gpu_selector_v;

try {
  sycl::buffer a_buf(A);
  sycl::buffer b_buf(B);
  sycl::range<1> rng{ A.size() };
  sycl::buffer c_buf(C.data(), rng);

  sycl::queue q(selector, exh);
  q.submit([&](sycl::handler &h) {
    sycl::accessor a(a_buf, h, sycl::read_only);
    sycl::accessor b(b_buf, h, sycl::read_only);
    sycl::accessor c(c_buf, h, sycl::write_only,
      sycl::no_init);

    h.parallel_for(rng, [=](auto i) {
      c[i] = a[i] + b[i];
    });
  }
  q.wait_and_throw();

} catch (std::exception const &e) {
  // print e and terminate
}
```

Synchronization (Reminder)

- Host side
  - `queue.wait();`
  - Events
    - Returned by queue operations
      ```
      auto event =
                  queue.submit(…);
      ```
    - `event.wait();`
    - Events can be passed as dependencies to some operations
      `queue.submit(…, event, …)`

Queue can be configured as in-order

Kernel execution overhead is in the range of µs

- In kernel
  - Barriers
    - `nd_item::barrier()`
    - **Only within the work group!**
  - Atomic operations
    - Similar to C++

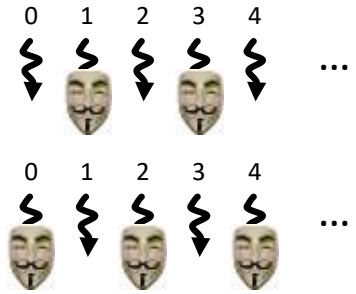
```
auto v =
sycl::ext::oneapi::atomic_ref
<int,order,scope,space>(a[0]);
v += something;
```

Available both for local and global memory, quite efficient (if no collisions)

Lockstep Implementation

- Masking Instructions
  - In case of data-driven branches
    - if-else conditions, while loops, …
  - All branches are traversed, threads mask their execution in invalid branches

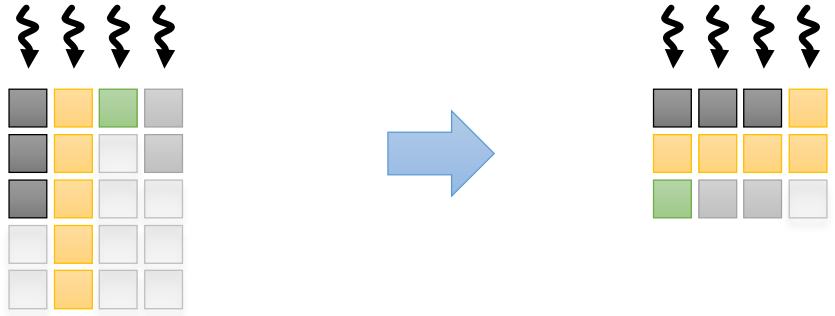
```
if (threadIdx.x % 2 == 0) {
    ... even threads code ...
} else {
    ... odd threads code ...
}
```



Reducing Thread Divergence

- Work Reorganization
  - In case the workload is imbalanced
  - “Cheap” load balancing can lead to better occupancy

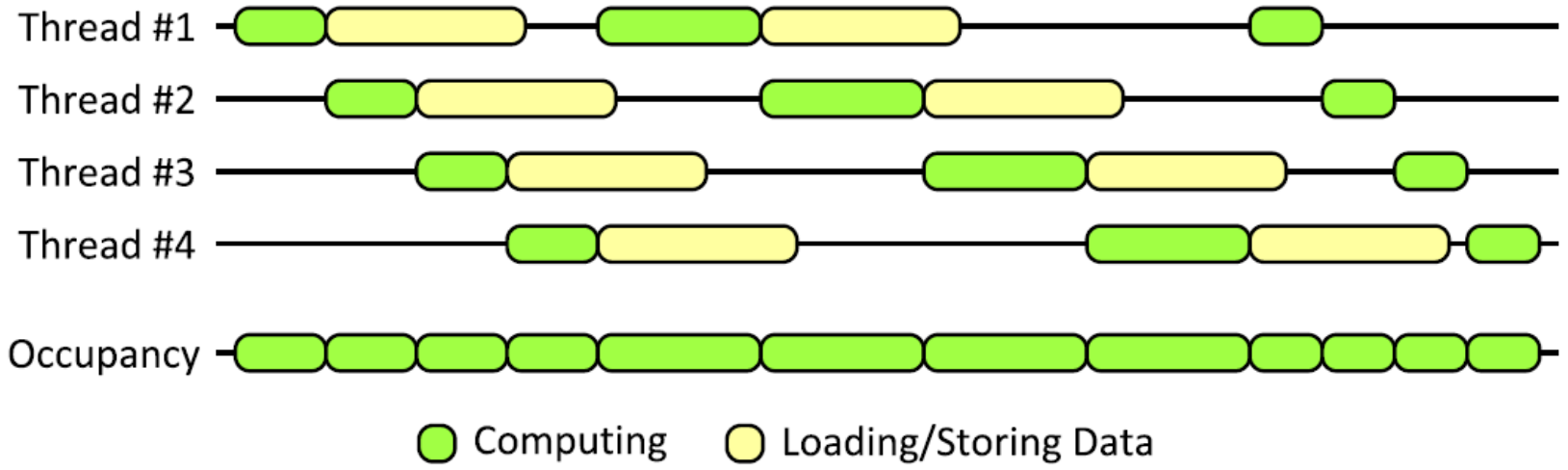
The overhead must not eliminate the gain



  - Example
    - Matrix with dimensions not divisible by warp size
    - Item `(i,j)` has linear index `i*width + j`

Optimizations: Hiding Latency

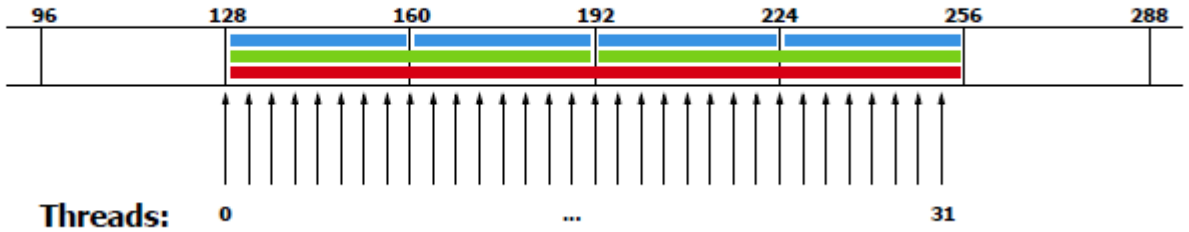
- Fast Context Switch
  - When a warp gets stalled
    - E.g., by data load/store
  - Scheduler switch to next active warp



Optimizations: Global Memory

- Access Patterns
  - Perfectly aligned sequential access

Aligned and sequential

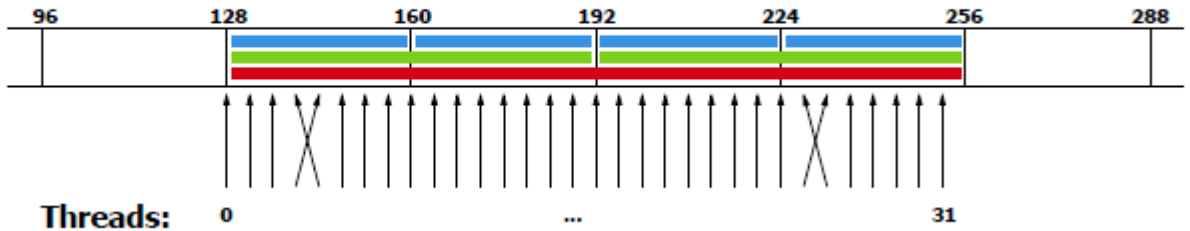


Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	1x 64B at 128 1x 64B at 192	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Optimizations: Global Memory

- Access Patterns
  - Perfectly aligned with permutation

Aligned and non-sequential

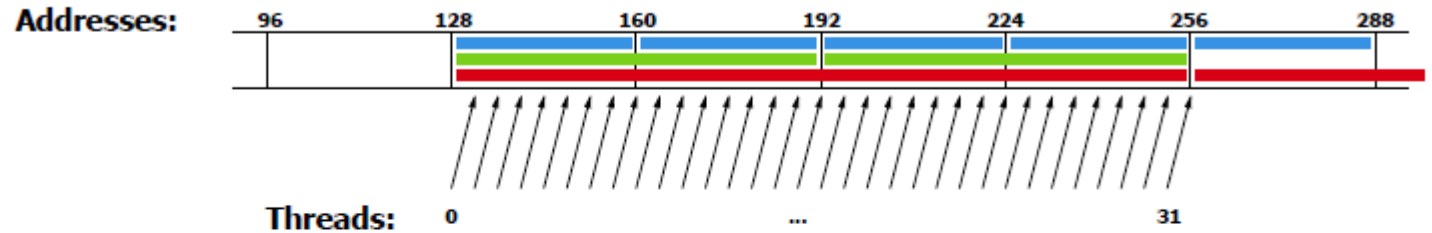


Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	8x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Optimizations: Global Memory

- Access Patterns
  - Continuous sequential, but misaligned

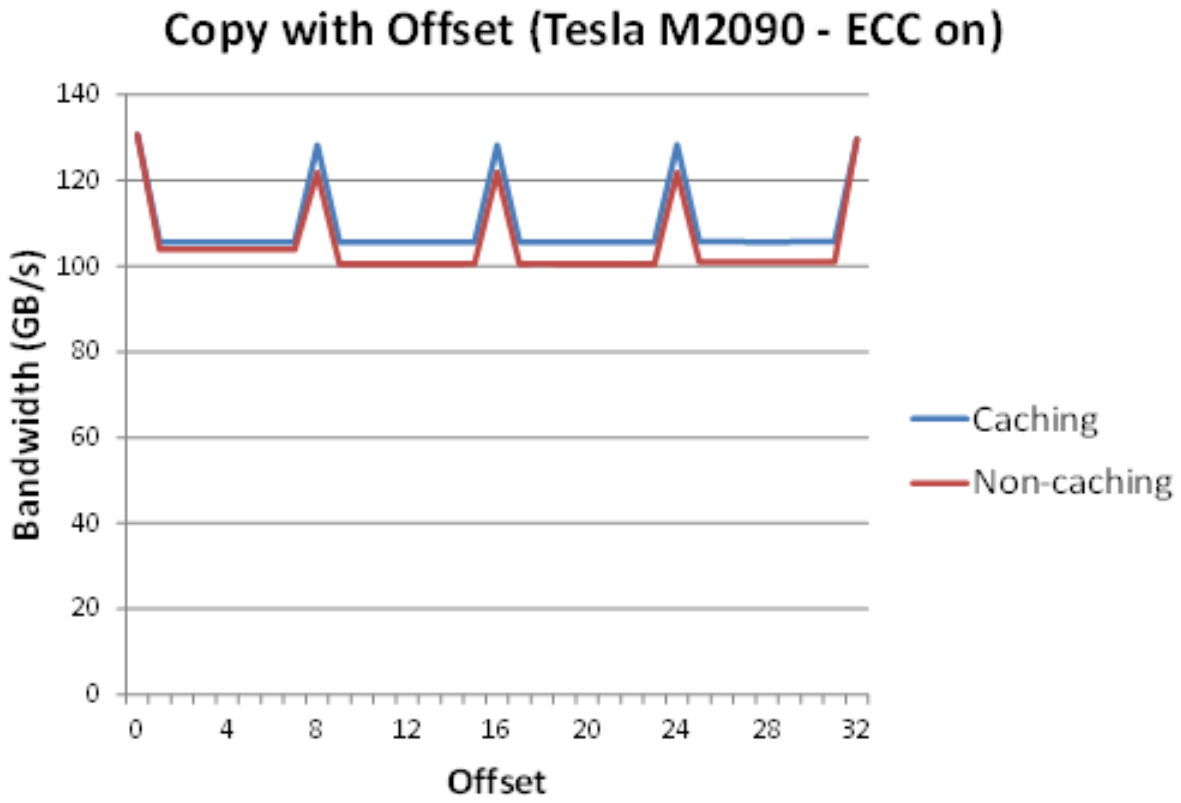
Mis-aligned and sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	7x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224 1x 32B at 256	1x 128B at 128 1x 64B at 192 1x 32B at 256	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

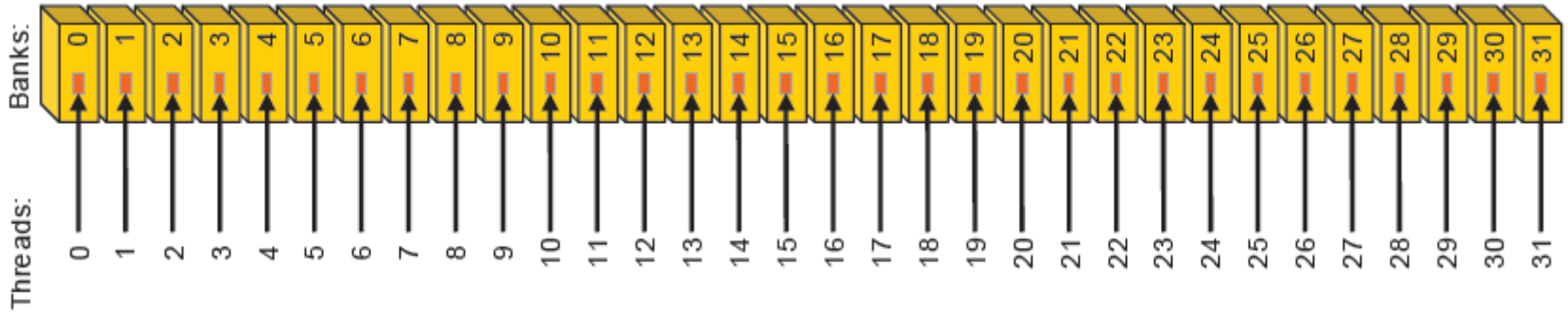
Optimizations: Global Memory

- Coalesced Loads Impact



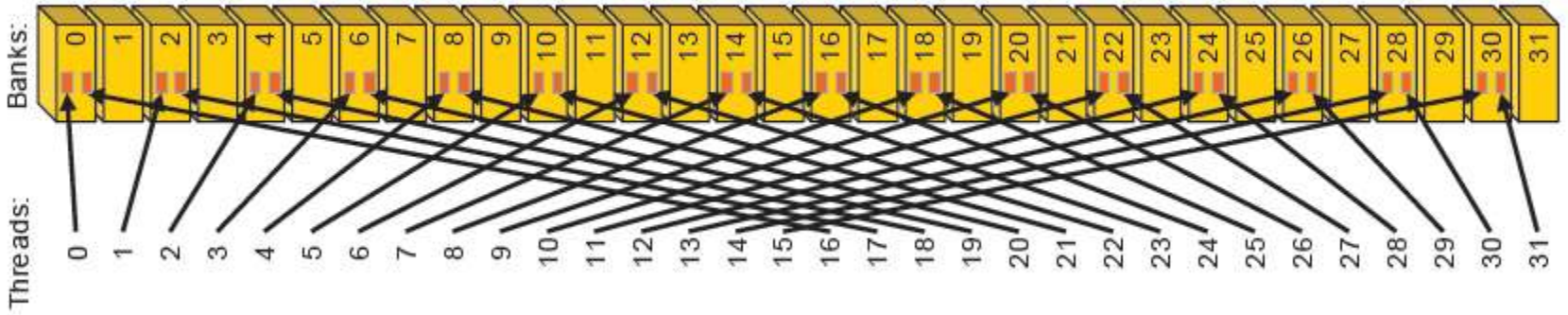
Optimizations: Local (Shared) Memory

- Linear Addressing
  - Each thread in warp access different memory bank
  - No collisions



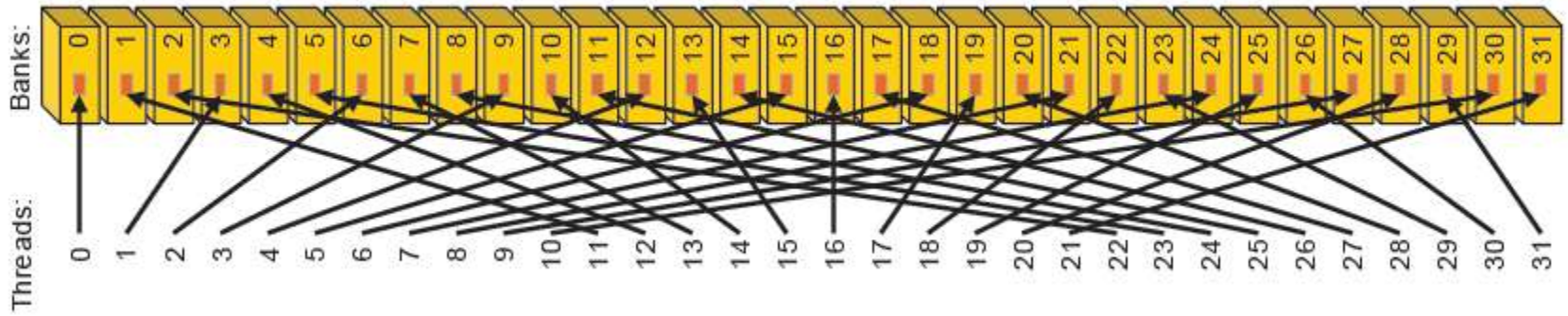
Optimizations: Local (Shared) Memory

- Linear Addressing with Stride
  - Each thread access **`2*i`**-th item
  - 2-way conflicts (2x slowdown) on CC < 3.0
  - No collisions on CC 3.x
    - Due to 64-bits per cycle throughput



Optimizations: Local (Shared) Memory

- Linear Addressing with Stride
  - Each thread access **`3*i`**-th item
  - No collisions, since the number of banks is not divisible by the stride

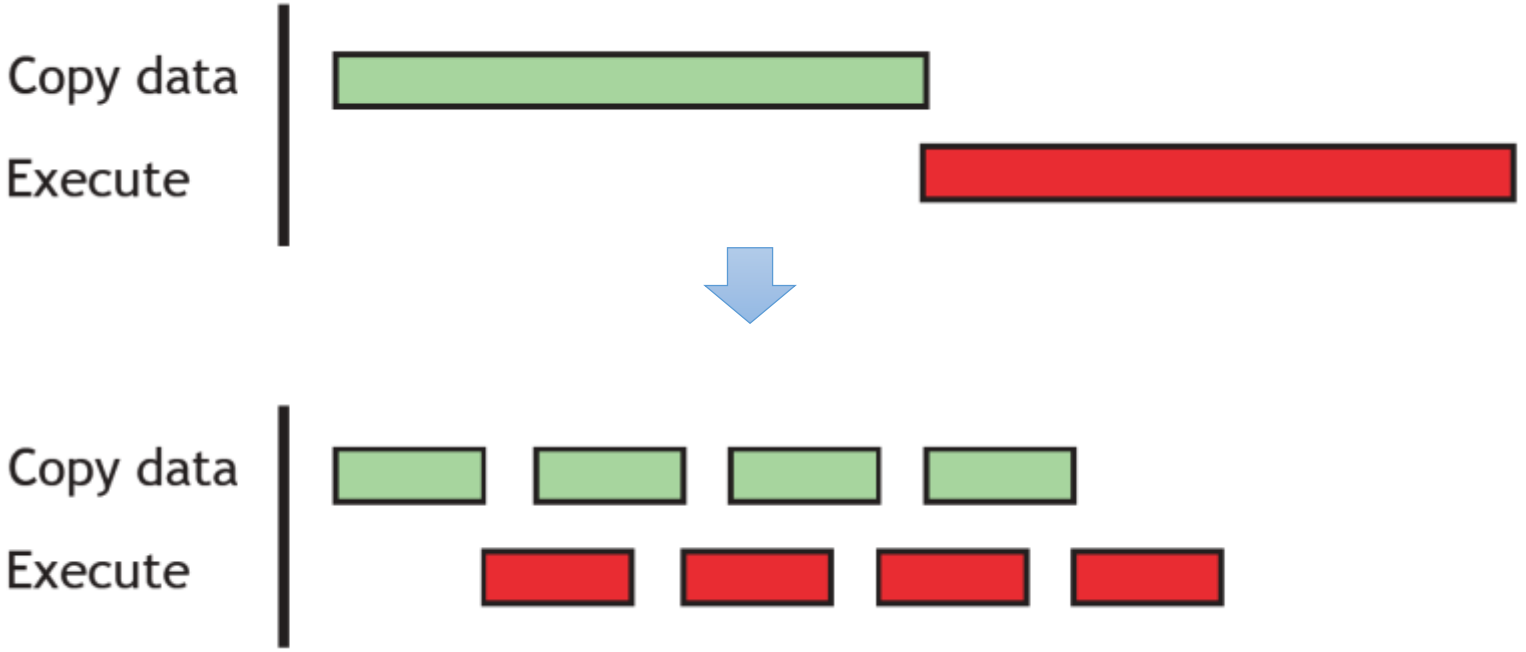


Optimizations: Local (Shared) Memory

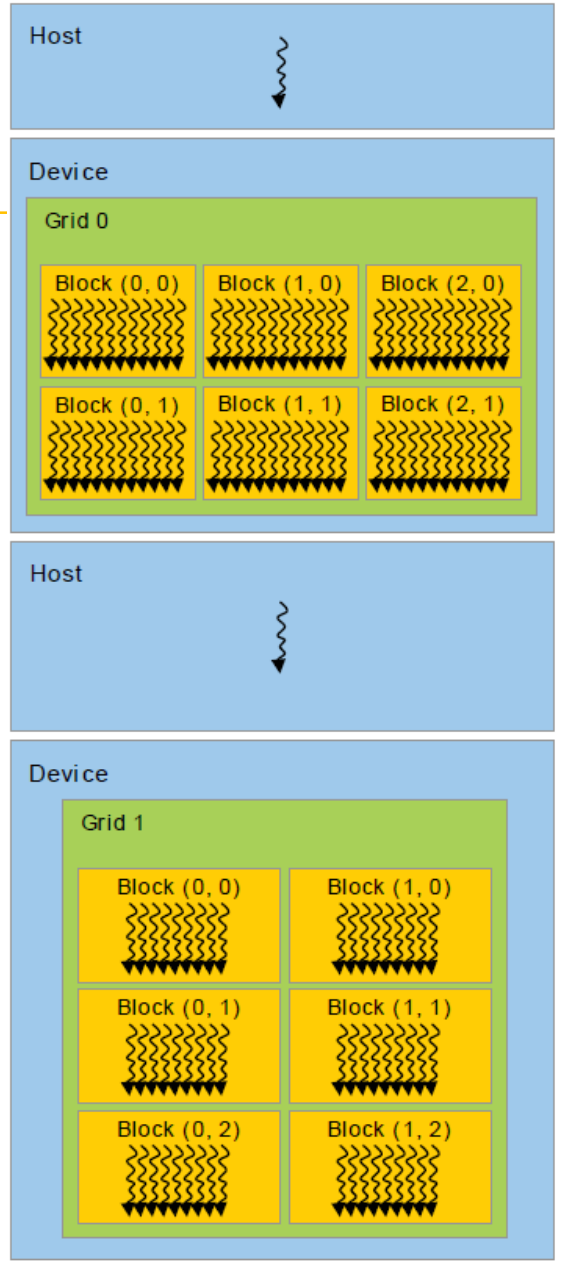
- Broadcast
  - One set of threads access value in bank #12 and the remaining threads access value in bank #20
  - Broadcasts simultaneously

Optimizations: Host-device Transfers

- Making a Good Use of Overlapping
  - Split the work into smaller fragments
  - Create a pipeline effect (load, process, store)



Heterogeneous Programming



- GPU
  - “Independent” device
  - Controlled by host
  - Used for “offloading”
- Host Code
  - Needs to be designed in a way that
    - Utilizes GPU(s) efficiently
    - Utilize CPU while GPU is working
    - CPU and GPU do not wait for each other

Discussion