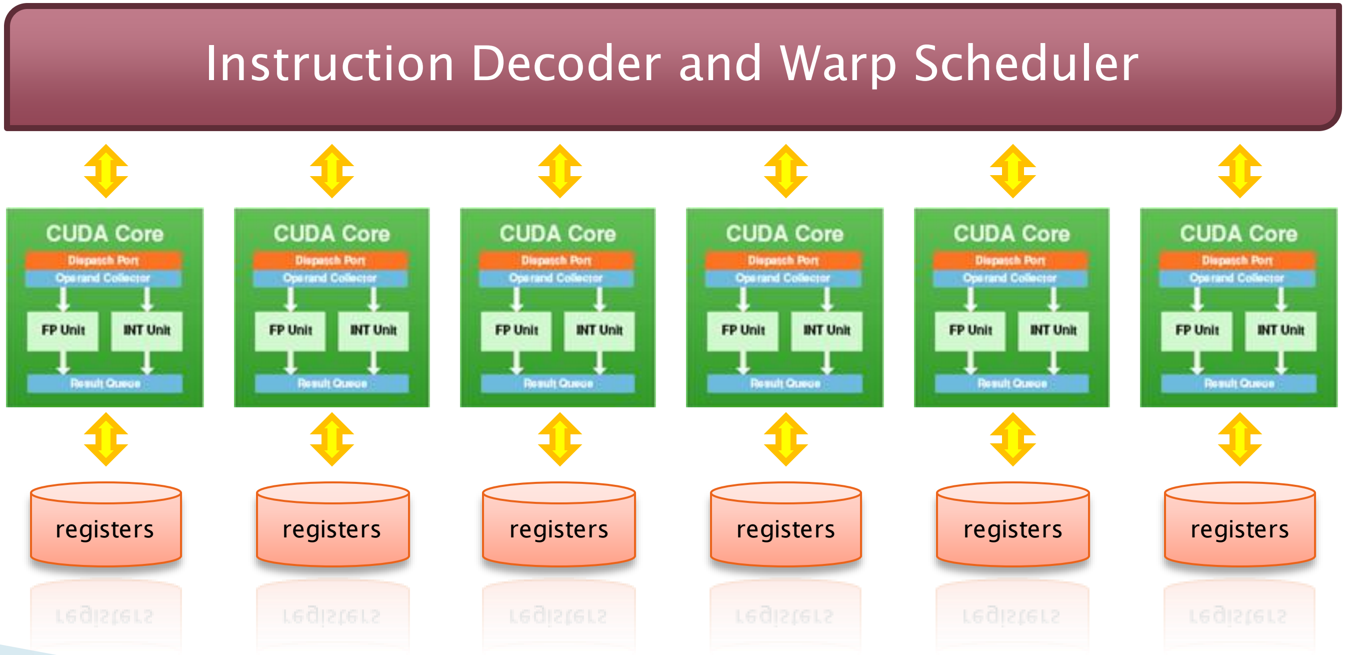
GPU Architectures and CUDA in More Detail

Martin Kruliš

SIMT Execution (Revision)

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

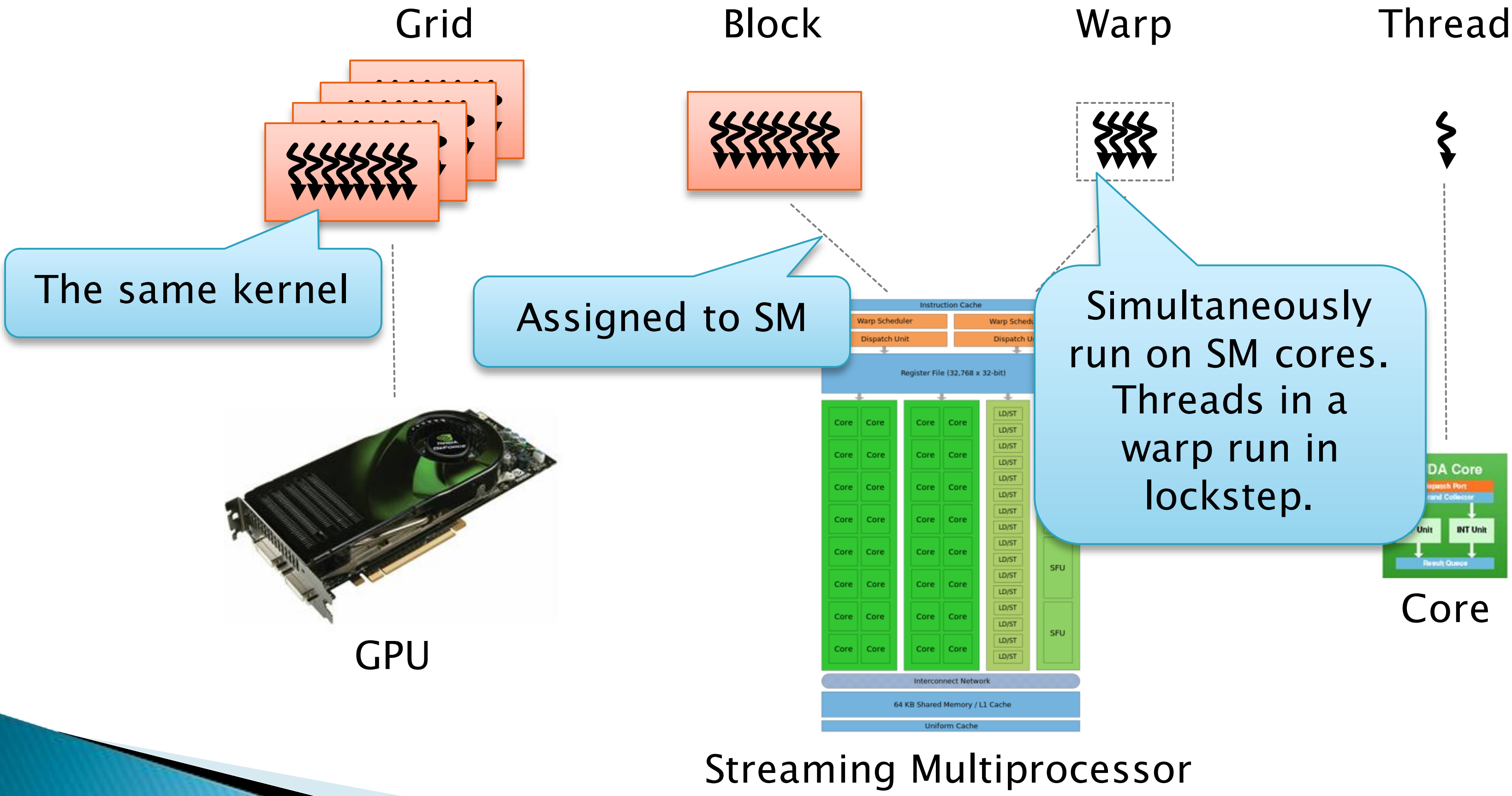


Instruction Decoder and Warp Scheduler

registers registers registers registers registers registers

Thread-Core Mapping (Revision)

- How are threads assigned to SMPs



Grid

Block

Warp

Thread

The same kernel

Assigned to SM

Simultaneously run on SM cores. Threads in a warp run in lockstep.

GPU

Streaming Multiprocessor

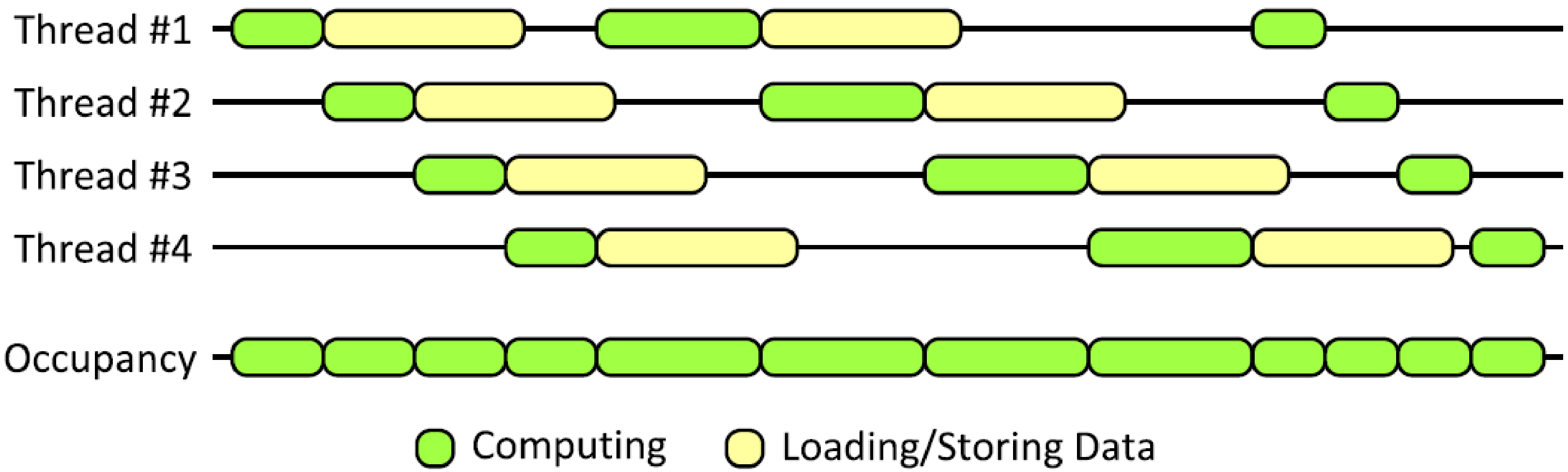
Core

Instruction Schedulers

- Decomposition
  - Each block assigned to the SMP is divided into warps and the warps are assigned to schedulers
- Schedulers
  - Select warp that is ready at every instruction cycle
  - The SMP instruction throughput depends on CC:
    - 1.x – 1 instruction per 4 cycles, 1 scheduler
    - 2.0 – 1 instruction per 2 cycles, 2 schedulers
    - 2.1 – 2 instructions per 2 cycles, 2 schedulers
    - 3.x and 5.x – 2 instructions per cycle, 4 schedulers

Hiding Latency

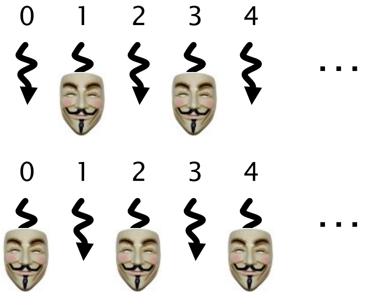
- Fast Context Switch
  - When a warp gets stalled
    - E.g., by data load/store
  - Scheduler switch to next active warp



SIMT and Branches

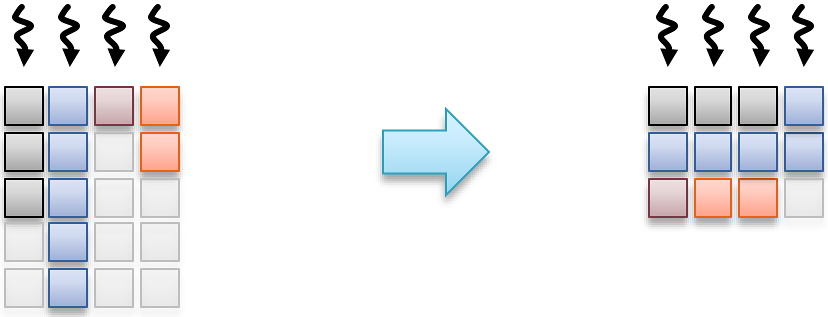
- Masking Instructions
  - In case of data-driven branches
    - if-else conditions, while loops, …
  - All branches are traversed, threads mask their execution in invalid branches

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



Reducing Thread Divergence

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



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

Block-wise Synchronization

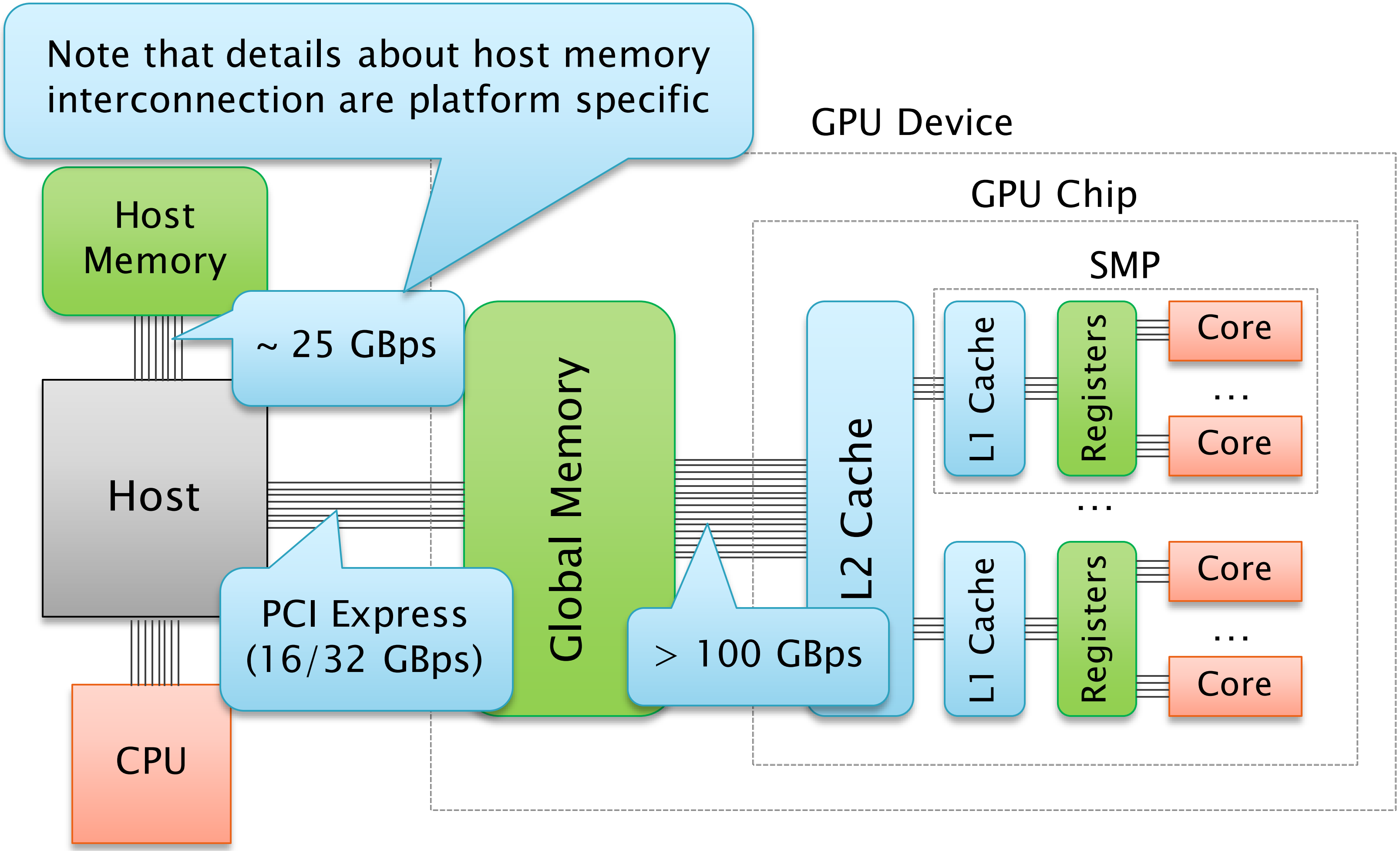
- Memory Fences

  ```
  __threadfence();
  __threadfence_block()
  __threadfence_system();
  ```

- Barrier
  - Synchronization between warps in block

  ```
  __syncthreads();
  __syncthreads_count(predicate);
  __syncthreads_and(predicate);
  __syncthreads_or(predicate);
  ```

GPU Memory (Revision)



Note that details about host memory interconnection are platform specific

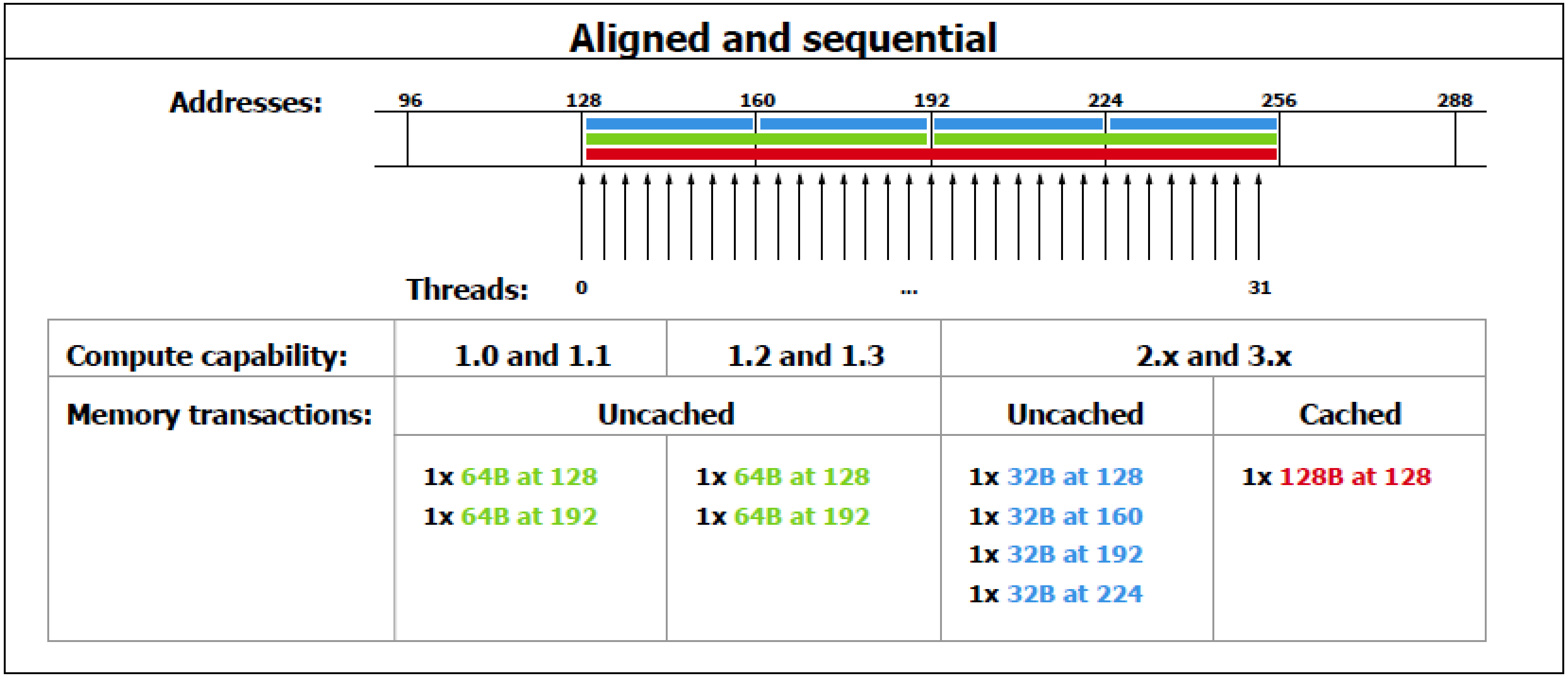
~ 25 GBps

PCI Express (16/32 GBps)

> 100 GBps

Global Memory

- Access Patterns
  - Perfectly aligned sequential access

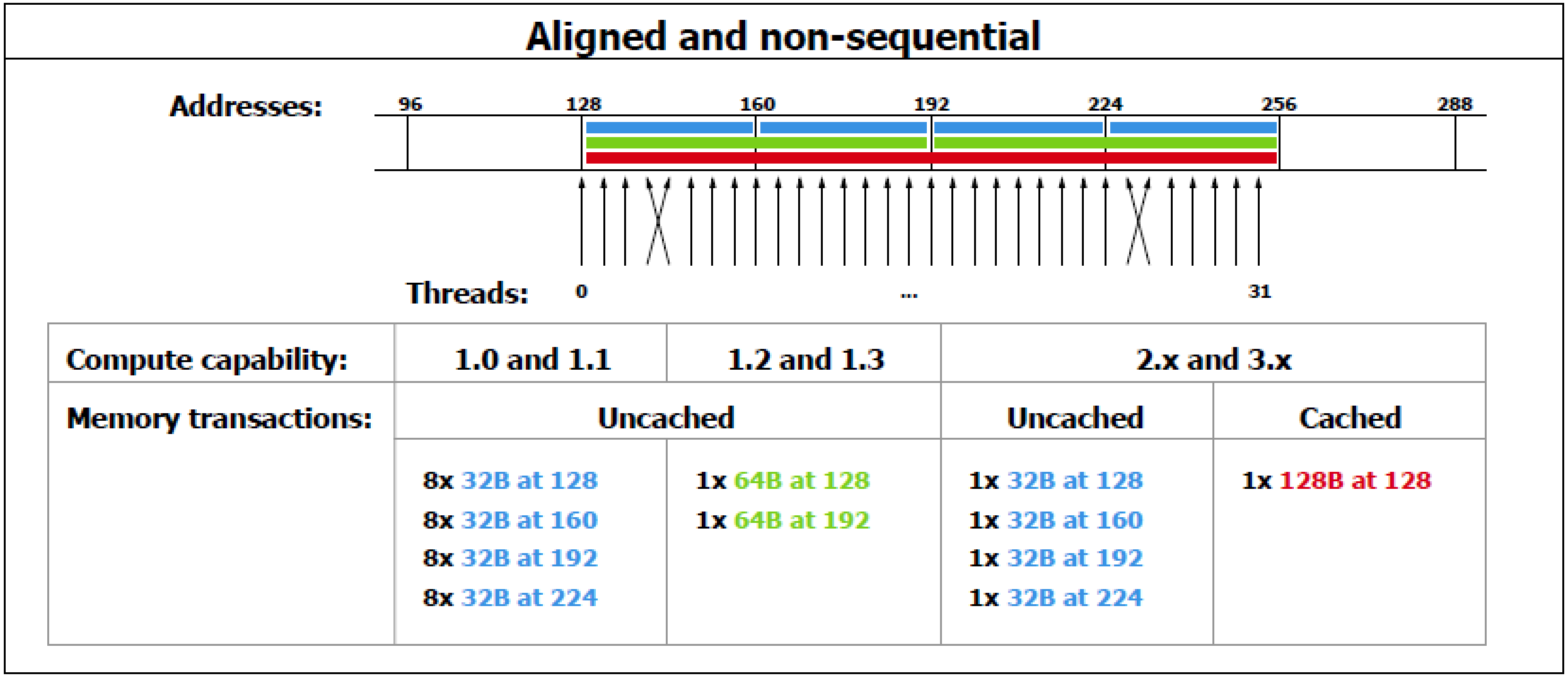


Aligned and sequential

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

Global Memory

- Access Patterns
  - Perfectly aligned with permutation

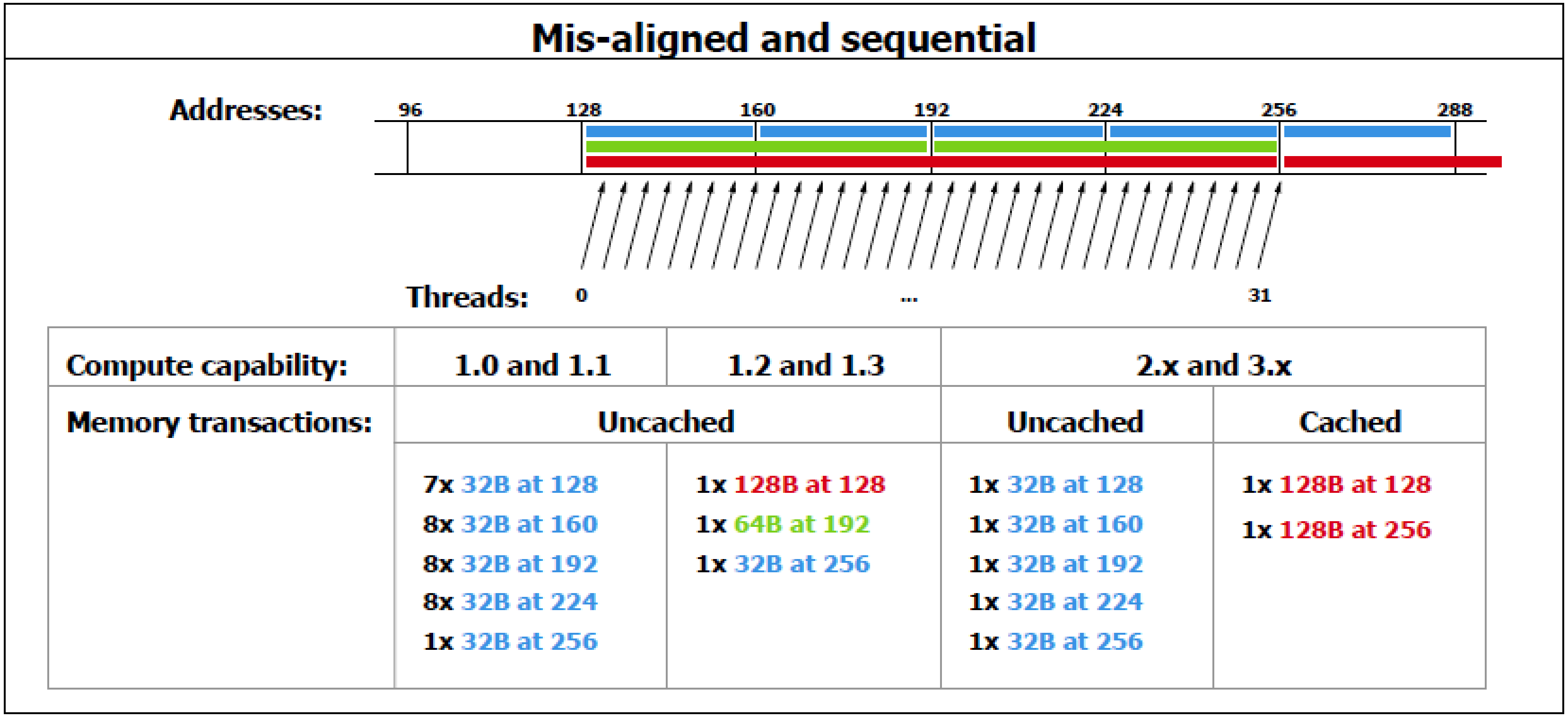


Aligned and non-sequential

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

Global Memory

- Access Patterns
  - Continuous sequential, but misaligned

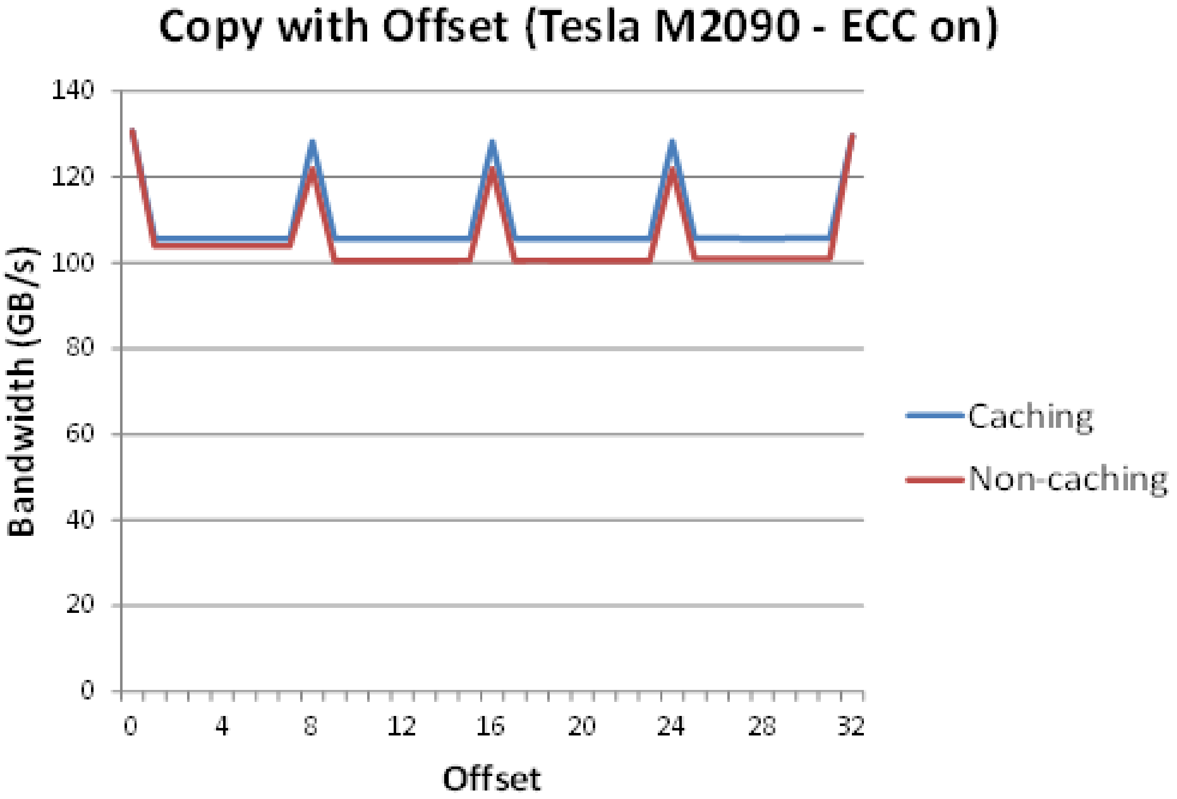


Mis-aligned and sequential

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

Global Memory

- Coalesced Loads Impact



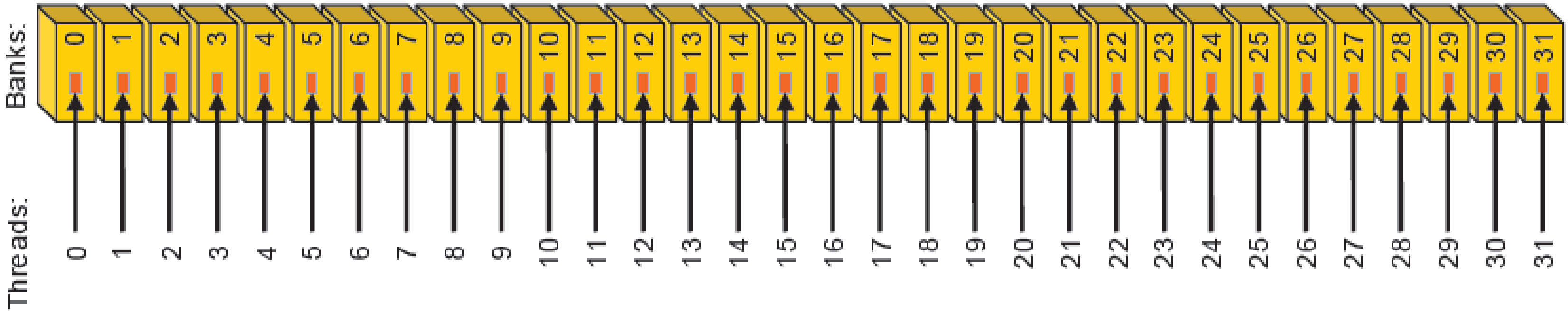
Shared Memory

- Memory Shared by SM
  - Divided into banks
    - Each bank can be accessed independently
    - Consecutive 32-bit words are in consecutive banks
      - Optionally, 64-bit words division is used (CC 3.x)
  - Bank conflicts are serialized
    - Except for reading the same address (broadcast)

Compute capability	Mem. size	# of banks	latency
1.x	16 kB	16	32 bits / 2 cycles
2.x	48 kB	32	32 bits / 2 cycles
3.x	48 kB	32	64 bits / 1 cycle

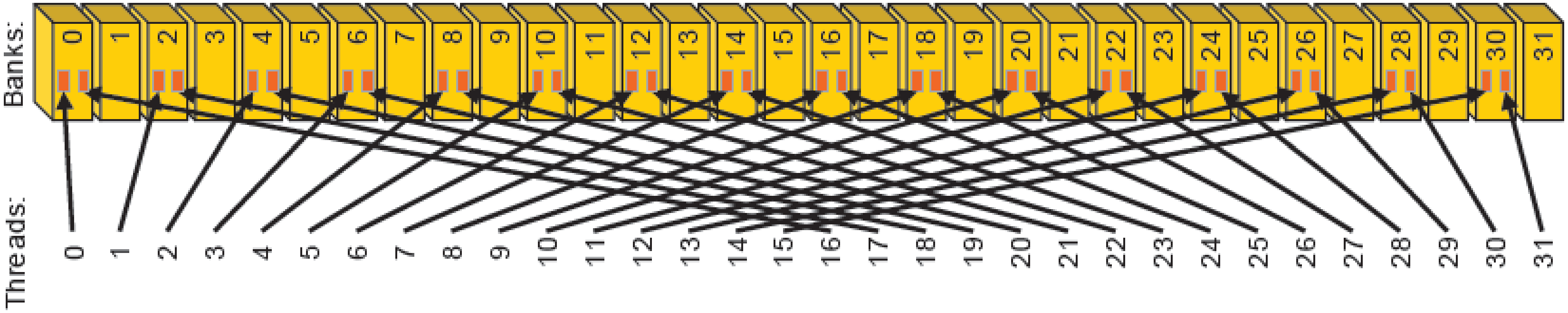
Shared Memory

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



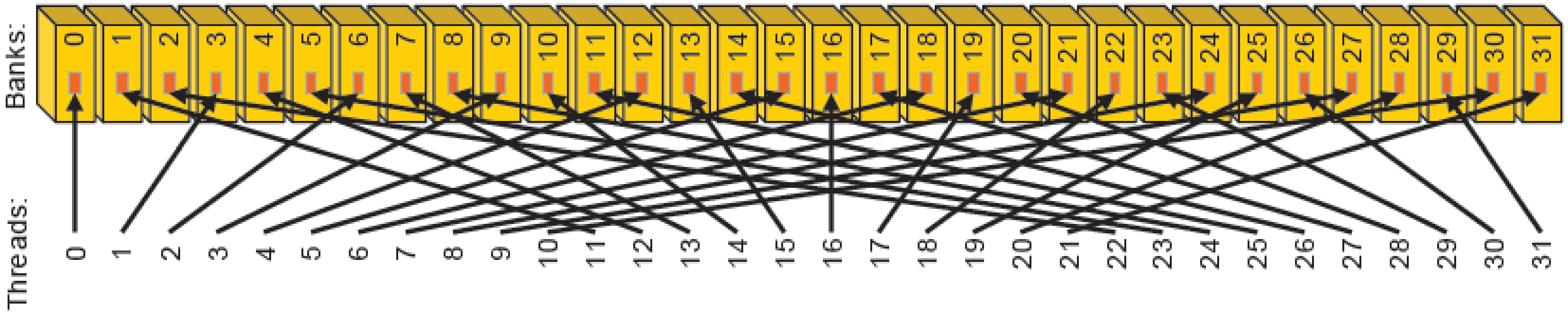
Shared Memory

- 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



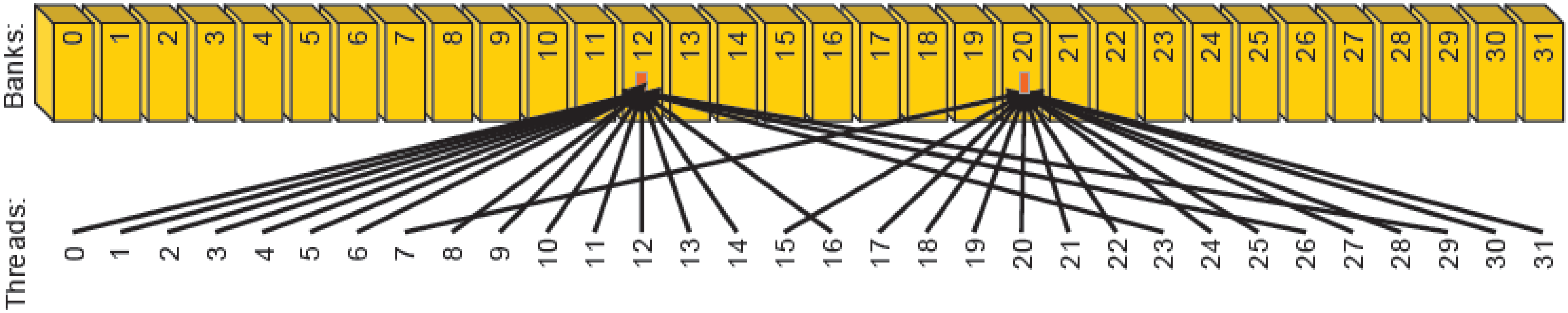
Shared Memory

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



Shared Memory

- Broadcast
  - One set of threads access value in bank #12 and the remaining threads access value in bank #20
  - Broadcasts are served independently on CC 1.x
    - I.e., sample bellow causes 2-way conflict
  - CC 2.x and newer serve broadcasts simultaneously



Registers

- Registers
  - One register pool per multiprocessor
    - 8-64k of 32-bit registers (depending on CC)
    - Register allocation is defined by compiler
  - All allocated blocks share the registry pool
    - Register pressure (heavy utilization) may limit number of blocks running simultaneously
      - It may also cause registry spilling
  - As fast as the cores (no extra clock cycles)
  - Read-after-write dependency
    - 24 clock cycles
    - Can be hidden if there are enough active warps

Local Memory

- Per–thread Global Memory
  - Allocated automatically by compiler
    - Compiler may report the amount of allocated local memory (use **`--ptxas-options=-v`**)
  - Large local structures and arrays are places here
    - Instead of the registers
  - Register Pressure
    - The registers are spilled into the local memory

Memory Allocation

- Global Memory
  - **`cudaMalloc()`**, **`cudaFree()`**
  - Dynamic kernel allocation
    - **`malloc()`** and **`free()`** called from kernel
    - **`cudaDeviceSetLimit(cudaLimitMallocHeapSize,`** *`size`*)
- Shared Memory
  - Statically (e.g., **`__shared__ int foo[16];`**)
  - Dynamically (by 3rd kernel launch parameter)

```
extern __shared__ float bar[];
float *bar1 = &(bar[0]);
float *bar2 = &(bar[size_of_bar1]);
```

Page-locked Memory

- Page-locked (Pinned) Host Memory
  - Host memory that is prevented from swapping
  - Created/dismissed by

```
cudaHostAlloc(), cudaFreeHost()
cudaHostRegister(), cudaHostUnregister()
```

  - Optionally with flags

```
cudaHostAllocWriteCombined
cudaHostAllocMapped
cudaHostAllocPortable
```

Optimized for writing, not cached on CPU

  - Copies between pinned host memory and device are automatically performed asynchronously
  - Pinned memory is a scarce resource

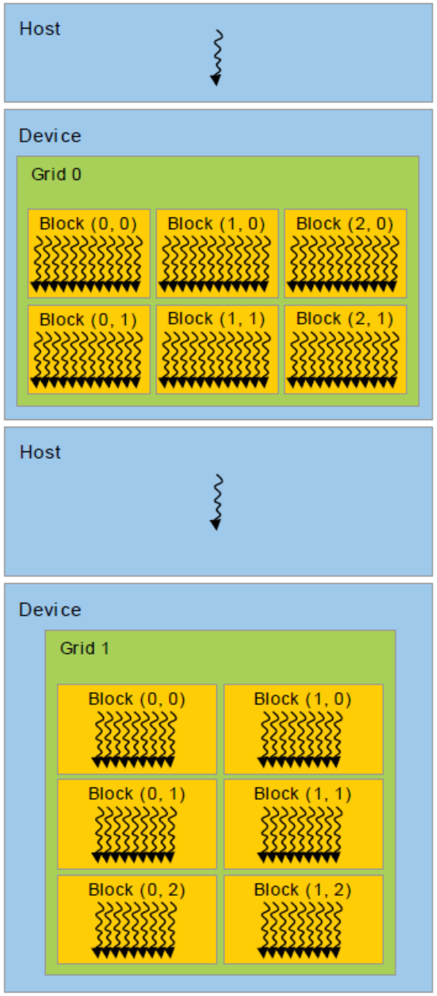
Memory Mapping

- Device Memory Mapping
  - Allowing GPU to access portions of host memory directly (i.e., without explicit copy operations)
    - For both reading and writing
  - The memory must be allocated/registered with flag **`cudaHostAllocMapped`**
  - The context must have **`cudaDeviceMapHost`** flag (set by **`cudaSetDeviceFlags()`**)
  - Function **`cudaHostGetDevicePointer()`** gets host pointer and returns corresponding device pointer

Heterogeneous Programming

- GPU
  - "Independent" device
  - Controlled by host
  - Used for "offloading"

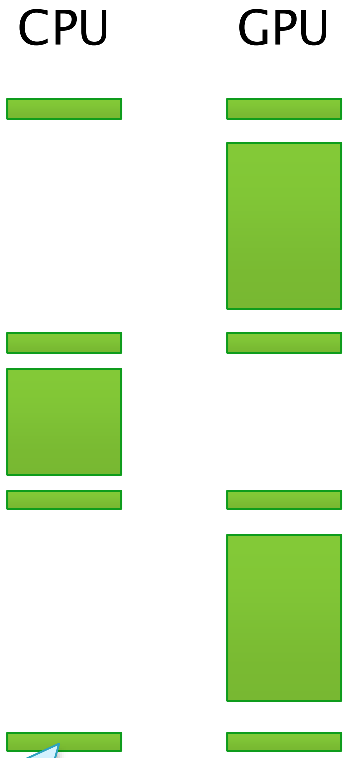
- Host Code
  - Needs to be designed in a way that
    - Utilizes GPU(s) efficiently
    - Utilize CPU while GPU is working
    - CPU and GPU do not wait for each other



Heterogeneous Programming

- Bad Example

```
cudaMemcpy(..., HostToDevice);
Kernel1<<<...>>>(...);
cudaDeviceSynchronize();
cudaMemcpy(..., DeviceToHost);
...
cudaMemcpy(..., HostToDevice);
Kernel2<<<...>>>(...);
cudaDeviceSynchronize();
cudaMemcpy(..., DeviceToHost);
...
```



Device is doing something useful

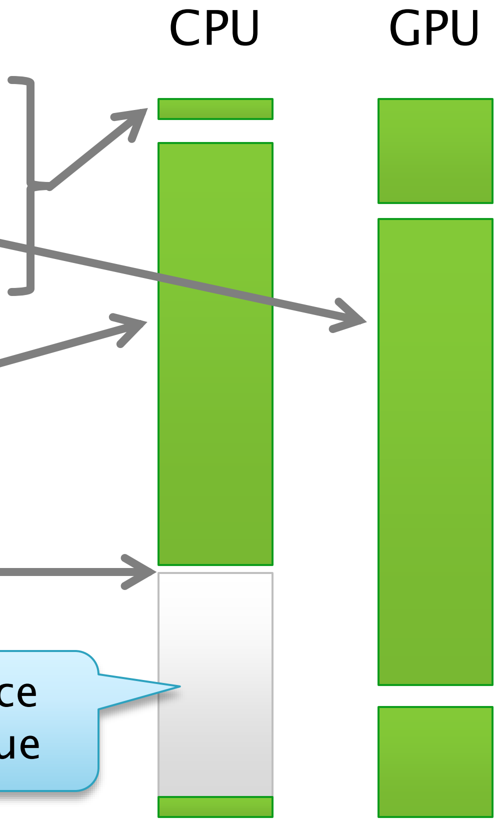
Overlapping Work

- Overlapping CPU and GPU work
  - Kernels
    - Started asynchronously
    - Can be waited for (**`cudaDeviceSynchronize()`**)
    - A little more can be done with streams
  - Memory transfers
    - **`cudaMemcpy()`** is synchronous and blocking
    - Alternatively **`cudaMemcpyAsync()`** starts the transfer and returns immediately
    - Can be synchronized the same way as the kernel

Overlapping Work

- Using Asynchronous Transfers

```
cudaMemcpyAsync(HostToDevice);
Kernel1<<<...>>>(...);
cudaMemcpyAsync(DeviceToHost);
...
do_something_on_cpu();
...
cudaDeviceSynchronize();
```



Workload balance becomes an issue

Streams

- Stream
  - In-order GPU command queue
    - Asynchronous GPU operations are registered in queue
      - Kernel execution
      - Memory data transfers
    - Commands in different streams may overlap
    - Provide means for explicit and implicit synchronization
  - Default stream (stream 0)
    - Always present, does not have to be created
    - Global synchronization capabilities

Streams

- Stream Creation

```
cudaStream_t stream;
cudaStreamCreate(&stream);
```

- Stream Usage

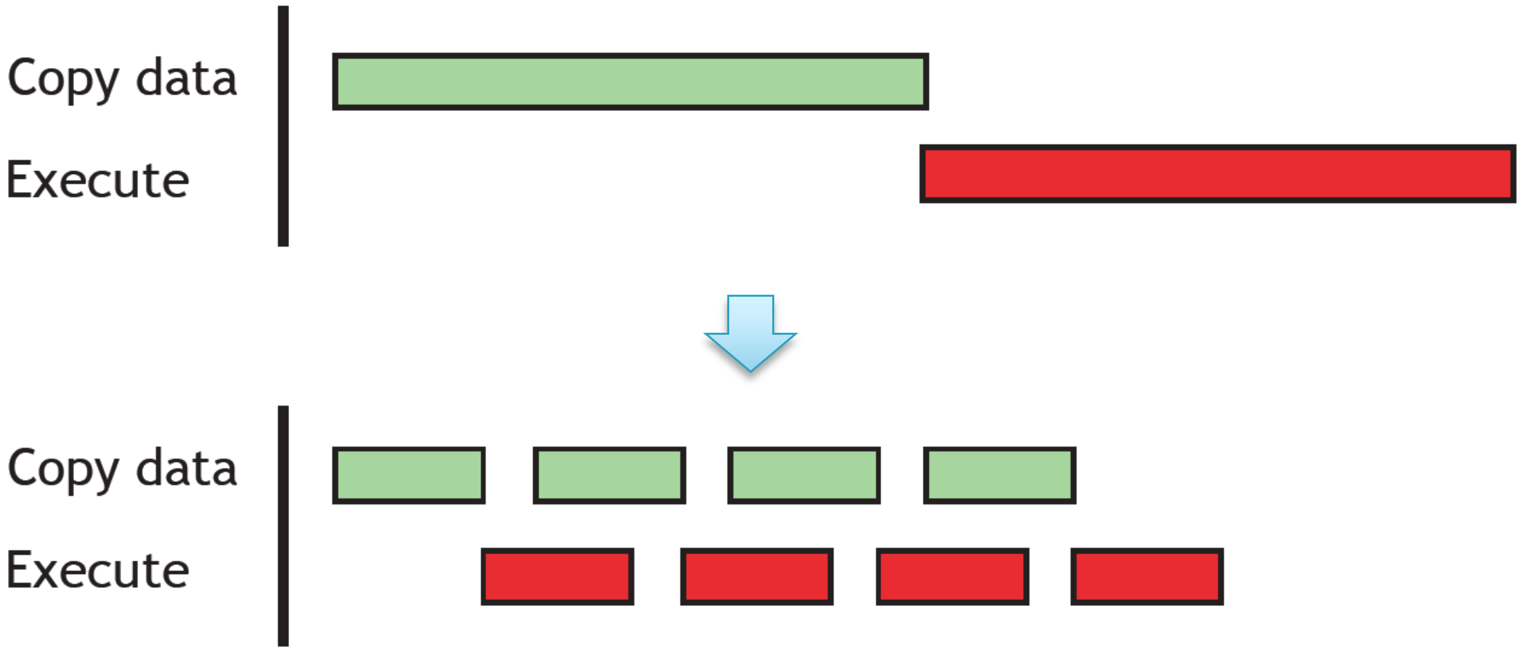
```
cudaMemcpyAsync(dst, src, size, kind, stream);
kernel<<<grid, block, sharedMem, stream>>>(...);
```

- Stream Destruction

```
cudaStreamDestroy(stream);
```

Pipelining

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



Instruction Set

- GPU Instruction Set
  - Oriented for rendering and geometry calculations
  - Rich set of mathematical functions
    - Many of those are implemented as instructions
    - Separate functions for doubles and floats
      - e.g., **sqrtf(***float***)** and **sqrt(***double***)**
  - Instruction behavior depends on compiler options
    - **-use_fast_math** – fast but lower precision
    - **-ftz=***bool* – flush denormals to zero
    - **-prec-div=***bool* – precise float divisions
    - **-prec-sqrt=***bool* – precise float sqrts
    - **-fmad=***bool* – use mul-add instructions (e.g., FFMA)

Single precision floats only

Atomic Instructions

- Atomic Instructions
  - Perform read-modify-write operation of one 32bit or 64bit word in global or shared memory
  - Require CC 1.1 or higher (1.2 for 64bit global atomics and 2.0 for 64bit shared mem. atomics)
  - Operate on integers, except for **`atomicExch()`** and **`atomicAdd()`** which also work on 32bit floats
  - Atomic operations on mapped memory are atomic only from the perspective of the device
    - Since they are usually performed on L2 cache

Atomic Instructions

- Atomic Instructions Overview
  - **atomicAdd(&*p*,*v*)**, **atomicSub(&*p*,*v*)** – atomically adds or subtracts **v** to/from **p** and return the old value
  - **atomicInc(&*p*,*v*)**, **atomicDec(&*p*,*v*)** – atomic increment/decrement computed modulo **v**
  - **atomicMin(&*p*,*v*)**, **atomicMax(&*p*,*v*)**
  - **atomicExch(&*p*,*v*)** – atomically swaps a value
  - **atomicCAS(&*p*,*v*)** – classical compare-and-set
  - **atomicAnd(&*p*,*v*)**, **atomicOr(&*p*,*v*)**, **atomicXor(&*p*,*v*)** – atomic bitwise operations

Warp Functions

- Voting Instructions
  - Intrinsics that allows the whole warp to perform reduction and broadcast in one step
    - Only active threads are participating
  - **`__all(`***`predicate`***`)`**
    - All active threads evaluate predicate
    - Returns non-zero if ALL predicates returned non-zero
  - **`__any(`***`predicate`***`)`**
    - Like **`__all`**, but the results are combined by logical OR
  - **`__ballot(`***`predicate`***`)`**
    - Return bitmask, where each bit represents the predicate result of the corresponding thread

Warp Functions

- Warp Shuffle Instructions
  - Fast variable exchange within the warp
  - Available for architectures with CC 3.0 or newer
  - Intrinsics
    - **`__shfl_sync()`** – direct copy from given lane
    - **`__shfl_up_sync()`** – copy with lower relative ID
    - **`__shfl_down_sync()`** – copy with higher relative ID
    - **`__shfl_xor_sync()`** – copy from a lane, which ID is computed as XOR of caller ID
    - All functions have optional width parameter, that allows to divide warp into smaller segments

Summary

- Right amount of threads
  - Saturate SMPs, but avoid registry spilling
- SIMT
  - Avoid warp divergence
  - Synchronization within a block is cheap
- Memory
  - Host-device transfer overlapping
  - Global memory coalesced transactions
  - Shared memory banking

Discussion