GP-GPU and High Performances Computing

Lecture 4
Programming on GP-GPU (2)

https://christophe.picard.pages.ensimag.fr/courses/course/gp-gpu/Lectures/cgpu-fall23-lecture4.pdf

Review of GP-GPU hierarchy

Review

- Thread block organization: a grid of blocks of threads
- Streaming multiprocessor (SM): set of cores, cache, schedulers
  - A block is assigned to and executed on a single SM
- Warp: A group of up to 32 threads within a block
  - Threads in a single warp can only run 1 set of instructions at once
  - Performing different tasks can cause warp divergence and affect performance
- Need to overlap warp computation with data loads

How to apply a filter on this image (1170x540) ?

Memory model

Latency and Throughput

Latency: delay caused by the physical speed of the hardware
Throughput: maximum rate of production/processing

- ➔ CPU = low latency, low throughput
  - ◆ CPU clock = 3 GHz (3 clocks/ns)
  - ◆ CPU main memory latency: ~100+ ns
  - ◆ CPU arithmetic instruction latency: ~1+ ns

- ➔ GPU = high latency, high throughput
  - ◆ GPU clock = 1.275 GHz (1 clock/ns)
  - ◆ GPU main memory latency: ~300+ ns
  - ◆ GPU arithmetic instruction latency: ~10+ ns

Latency and Throughput (Quadro RTX 6000P-8C)

- ➔ Compute throughput: 16.3 TFLOPS (single precision) = 16.3 x10^3 GLFOPS
- ➔ Global memory bandwidth: 672 GB/s (168 Gfloat/s)

Global memory is ~100 times too slow to feed the compute unit.

- ➔ GPGPU is IO limited. IO will be the throughput bottleneck. There is a strong requirement to be careful about how the IO are performed.
- ➔ To get beyond memory limitations, multiple FLOPs have to be performed per shared memory load.

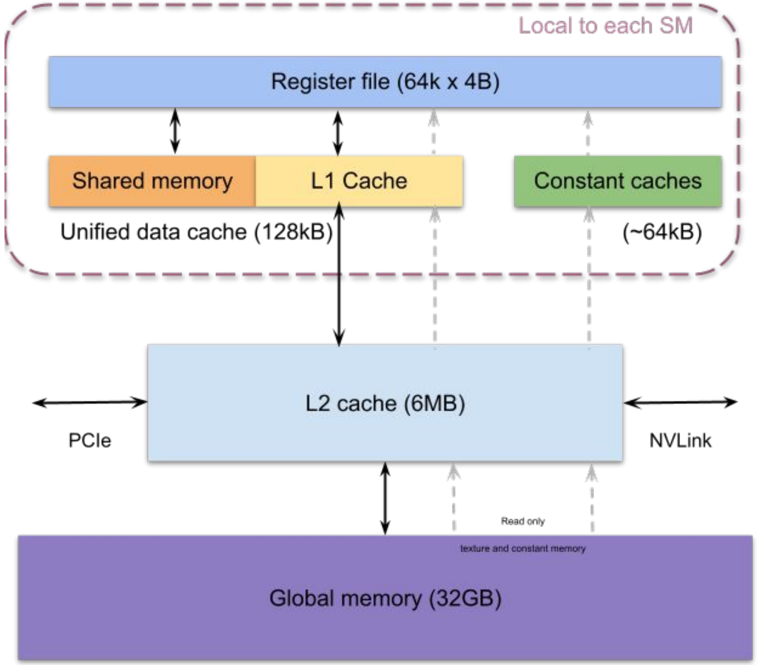
Cache

- A cache is a chunk of memory that sits in between a larger pool of memory and the processor
  - Often times implemented at hardware level
  - Has much faster access speed than the larger pool of memory
- When memory is requested, extra memory near the requested memory is read into a cache
  - Amount read is cache and memory pool specific
  - Regions of memory that will always be cached together are called cache lines ◦ This makes future accesses likely to be found in the cache
- Such accesses are called cache hits and allow much faster access
- If an access is not found in the cache, it's called a cache miss (and there is obviously no performance gain)

GPU Memory Breakdown

- ➔ Registers
- ➔ Local memory
- ➔ Global memory
- ➔ Shared memory
- ➔ L1/L2/L3 cache
- ➔ Constant memory
- ➔ Texture memory
- ➔ Read-only cache (CC 3.5+)

Memory organization



Memory hierarchy

Registers: The fastest form of memory on the multi-processor. Is only accessible by the thread. Has the lifetime of the thread.

Local memory: Resides in global memory and can be 150x slower than register or shared memory. Is only accessible by the thread. Has the lifetime of the thread.

Shared Memory: Can be as fast as a register when there are no bank conflicts or when reading from the same address. Accessible by any thread of the block from which it was created. Has the lifetime of the block.

Global memory: Potentially 150x slower than register or shared memory -- watch out for uncoalesced reads and writes. Accessible from either the host or device. Has the lifetime of the application—that is, it persistent between kernel launches.

Global memory

Global memory is separate hardware from the GPU core (containing SM’s, caches, etc).

- ➔ The vast majority of memory on a GPU is global memory
- ➔ If data doesn’t fit into global memory, you are going to have process it in chunks that do fit in global memory.
- ➔ GPUs have .5 - 24GB of global memory, with most now having ~2GB.
- ➔ Global memory latency is ~300ns on Kepler and ~600ns on Fermi

Accessing global memory efficiently

Global memory IO is the slowest form of IO on GPU

➔ except for accessing host memory

Because of this, we want to access global memory as little as possible

Access patterns that play nicely with GPU hardware are called coalesced memory accesses.

Memory Coalescing

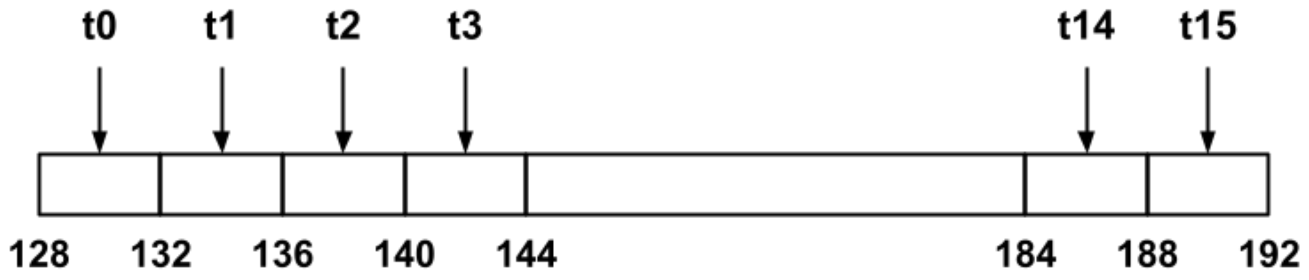
Memory accesses are done in large groups setup as Memory Transactions

- ➔ Done per warp
- ➔ Fully utilizes the way IO is setup at the hardware level

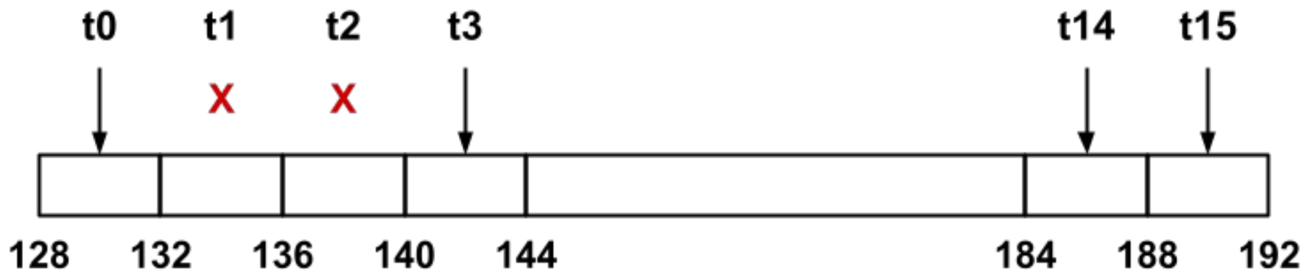
Coalesced memory accesses minimize the number of cache lines read in through these memory transactions

- ➔ GPU cache lines are 128 bytes and are aligned
- ➔ Memory coalescing is much more complicated in reality

Coalesced access

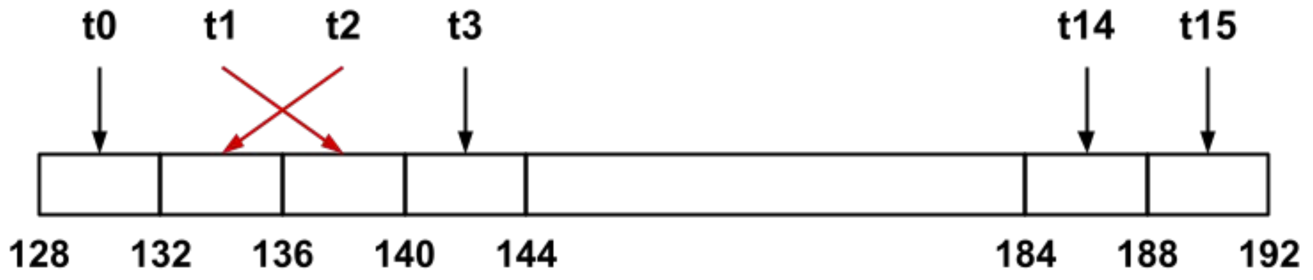


All threads participate

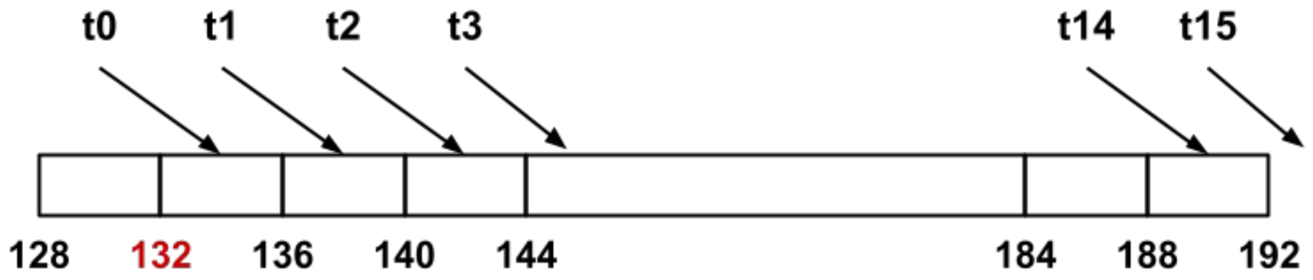


Some threads do not participate

Uncoalesced access



Permuted access by threads



Misaligned starting address (not starting at a multiple of 64)

Shared Memory

- ➔ Very fast memory located in the SM
- ➔ Same hardware as L1 cache
- ➔ ~5ns of latency
- ➔ Maximum size of ~48KB (varies per GPU) Scope of shared memory is the block

- ➔ SM = streaming multiprocessor
- ➔ SM ≠ shared memory

Shared memory syntax

Can allocate shared memory statically (size known at compile time) or dynamically (size not known until runtime)

➔ Static allocation syntax:
```
__shared__ float data[1024];
```

➔ Declared in the kernel, nothing in host code
  - Host:
    ```
    kernel<<<grid_dim, block_dim, numBytesShMem>>>(args);
    ```
  - Device (in kernel):
    ```
    extern __shared__ floats[];
    ```

Shared Memory Allocation

Task: Compute byte frequency counts

Input: array of bytes of length n

Output: 256 element array of integers containing number of occurrences of each byte

Naive: build output in global memory, n global stores

Smart: build output in shared memory, copy to global memory at end, 256 global stores

Computational Intensity

Computational intensity is a representation of how many operations must be done on a single data point (FLOPs / IO)

- ➔ Vaguely similar to the big O notation in concept and usage
  - ◆ Matrix multiplication: n^3 / n^2 = n
  - ◆ n-body simulation: n^2 / n = n

If computational intensity is > 1, then same data used in more than 1 computation

- ➔ Do as few global loads and as many shared loads as possible

A common pattern in kernels

(1) copy from global memory to shared memory

(2) **`__syncthreads()`**

(3) perform computation, incrementally storing output in shared memory,

(4) **`__syncthreads()`**

(5) copy output from shared memory to output array in global memory

Bank conflicts

Shared memory is setup as 32 banks

➔ If you divide the shared memory into 4 byte-long elements, element i lies in bank i % 32.

A bank conflict occurs when 2 threads in a warp access different elements in the same bank.

➔ Bank conflicts cause serial memory accesses rather than parallel
➔ Serial anything in GPU programming = bad for performance

Bank conflicts and strides

Stride is the distance from thread i access to thread i + 1 access

Stride 1 ⇒ 32 x 1-way “bank conflicts” (so conflict-free) Stride 2 ⇒ 16 x 2-way bank conflicts

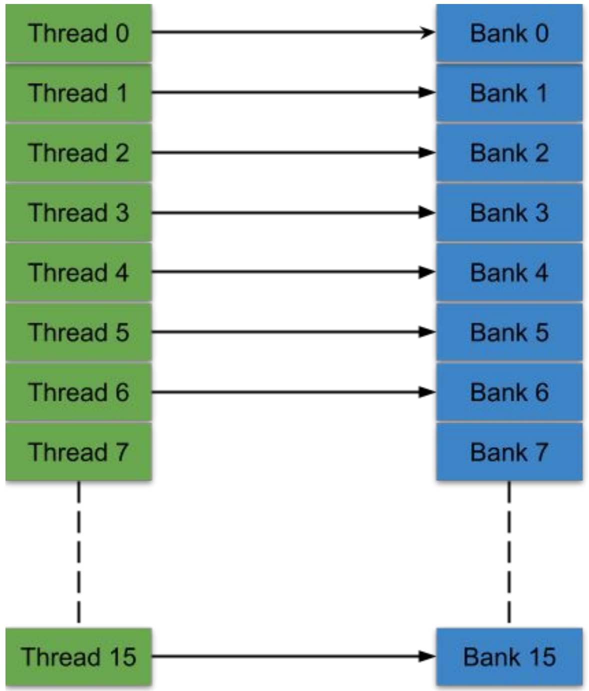
Stride 3 ⇒ 32 x 1-way “bank conflicts” (so conflict-free) Stride 4 ⇒ 8 x 4-way bank conflicts

...

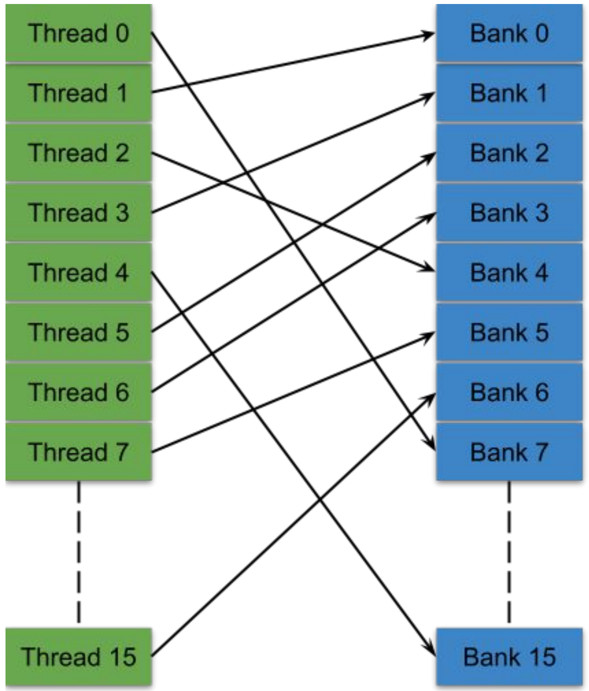
Stride 32 ⇒ 1 x 32-way bank conflict

No bank conflict

Linear addressing stride == 1

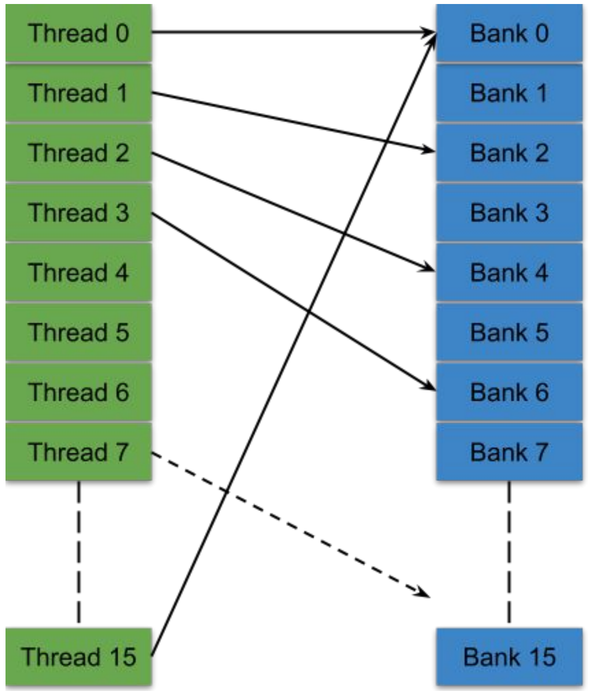


Random 1:1 Permutation

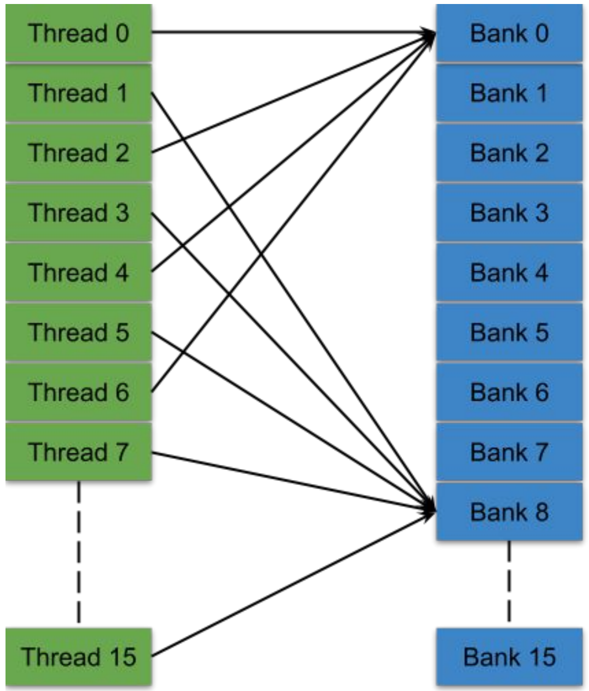


Bank conflict

Linear addressing stride == 2



Linear addressing stride == 8



Padding to avoid bank conflicts

To fix the stride 32 case, waste a byte on **padding** and make the stride 33

Don’t store any data in slots 32, 65, 98, ...

Now we have

- ➔ thread 0 ⇒ index 0 (bank 0)
- ➔ thread 1 ⇒ index 33 (bank 1)
- ➔ thread i ⇒ index 33 * i (bank i)

Shared memory bank conflicts

Shared memory is as fast as registers if there are no bank conflicts

- The fast case
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)

- The slow case
  - Bank Conflict: multiple threads in the same half-warp access the same bank

- Must serialize the accesses
- Cost = max # of simultaneous accesses to a single bank
- Each bank has a bandwidth of 32 bits per clock cycle, 32 bits successive words are addressed to successives banks (32 on modern GPU)

Registers

A Register is a piece of memory used directly by the processor

- ➔ Fastest “memory” possible, about 10x faster than shared memory ◦ There are tens of thousands of registers in each SM
- ➔ Generally works out to a maximum of 32 or 64 32-bit registers per thread
- ➔ Most stack variables declared in kernels are stored in registers
  - ◆ example: `float x;`

Statically indexed arrays stored on the stack are sometimes put in registers.

Local memory

- ➔ Local memory is everything on the stack that can’t fit in registers
- ➔ The scope of local memory is just the thread.
- ➔ Local memory is stored in global memory much slower than registers

Constant Memory

Constant memory is global memory with a special cache

- ➔ Used for constants that cannot be compiled into program
- ➔ Constants must be set from host before running kernel.

~64KB for user, ~64KB for compiler

- ➔ kernel arguments are passed through constant memory
- ➔ 8KB cache on each SM specially designed to broadcast a single memory address to all threads in a warp (called static indexing)
  - ◆ Can also load any statically indexed data through constant cache using “load uniform” (LDU) instruction

Constant memory syntax

- ➔ In global scope (outside of kernel, at top level of program):

```
__constant__ int foo[1024];
```

- ➔ In host code:

```
cudaMemcpyToSymbol(foo, h_src, sizeof(int) * 1024);
```

Texture Memory

Complicated and only marginally useful for general purpose computation
Useful characteristics:

- ➔ 2D or 3D data locality for caching purposes through “CUDA arrays”. Goes into special texture cache.
- ➔ fast interpolation on 1D, 2D, or 3D array
- ➔ converting integers to “unitized” floating point numbers

Use cases:

(1) Read input data through texture cache and CUDA array to take advantage of spatial caching. This is the most common use case.
(2) Take advantage of numerical texture capabilities.
(3) Interaction with OpenGL and general computer graphics

Texture Memory

And that’s all we’re going to say on texture memory.
It’s a complex topic, you can learn everything you want to know about it from the textbook

Synchronization

Ideal case for parallelism:

- ➔ no resources shared between threads
- ➔ no communication needed between threads

However, many algorithms that require shared resources can still be accelerated by massive parallelism of the GPU.

- ➔ Need to avoid Deadlock! Processes can depend on each other and get stuck!
- ➔ Each member of a group can wait for another member, including itself, to take action.

Synchronization

- ➔ Synchronization is a process by which multiple threads must indirectly communicate with each other in order to make sure they do not clash with each other

Example of synchronization issue:

```
int x = 1;
Thread 1: x += 1;
Thread 2: x += 1;
```

- ◆ Thread 1 reads in the value of x (which is 1) into a register
- ◆ Thread 2 reads in the value of x (which is still 1) into a register
- ◆ Both threads increment the values they read in but they both think the final value is 2
- ◆ They write x back out and the final result is 2

CUDA Synchronization

- Usually use the **`__syncthreads()` function** to sync threads **within a block**
  - Only works at the block level
  - SMs are separate from each other so can't do “better” than this
  - Similar to `barrier()` function in C/C++
  - This `__synchthreads()` call is very useful for kernels using **shared memory**.

Atomic Operations

- ➔ **Atomic Operations** are operations that ONLY happen **in sequence, independent of other processes**
- ➔ For example, adding up to N numbers by adding the numbers to a variable that starts in 0, you must add one number at a time

Don't do this though. Only use when you have no other options

- ➔ "Atomic operations" in concurrent programming are program operations that run completely independently of any other processes.

Atomic Operations

- ➔ CUDA provides built in atomic operations
  - ◆ Use the functions: **`atomic<op>(float *address, float val);`**
  - ◆ Replace <op> with one of: Add, Sub, Exch, Min, Max, Inc, Dec, And, Or, Xor

e.g. `atomicAdd(float *address, float val)` for atomic addition

- ➔ These functions are all implemented using a function called `atomicCAS(int *address, int compare, int val)`
  - ◆ CAS stands for compare and swap.
  - ◆ The function compares *address to compare and swaps the value to val if the values are different
  - ◆ Double precision more accurate, but can be much slower!

Instruction Dependencies

An **Instruction Dependency** is a requirement relationship between instructions that force a sequential execution

- ➔ In the example on the right, each summation call must happen in sequence because the value of acc depends on the previous summation as well

Can be caused by direct dependencies or requirements set by the execution order of code

- ➔ I.e. You can't start an instruction until all previous operations have been completed in a single thread

```
acc += x[0];
acc += x[1];
acc += x[2];
acc += x[3];
...
```

Instruction Level Parallelism (ILP)

- **Instruction Level Parallelism** is when you avoid performances losses caused by instruction dependencies
  - Do not to wait until instruction *n* has finished to start instruction *n + 1*
  - In CUDA, also removes performances losses caused by how certain operations are handled by the hardware

ILP Example

```
z0 = x[0] + y[0];
z1 = x[1] + y[1];
```

COMPILATION

```
x0 = x[0];
y0 = y[0];
z0 = x0 + y0;

x1 = x[1];
y1 = y[1];
z1 = x1 + y1;
```

- The second half of the code can't start execution until the first half completes

ILP Example

```
z0 = x[0] + y[0];
z1 = x[1] + y[1];
```

```
x0 = x[0];
y0 = y[0];
x1 = x[1];
y1 = y[1];
z0 = x0 + y0;
z1 = x1 + y1;
```

- ➔ Sequential nature of the code due to instruction dependency has been minimized.
- ➔ Additionally, this code minimizes the number of memory transactions required

Warp Schedulers

Warp schedulers find a warp that is ready to execute its next instruction and available execution cores and then start execution

- ➔ GPU has at least **one warp scheduler per SM**.
- ➔ Each scheduler has **2 dispatchers**.
- ➔ The scheduler picks an eligible warp and executes all threads in the warp .
- ➔ If any of the threads in the executing warp stalls (uncached memory read) the scheduler makes it inactive. If there are no eligible warps left then GPU **idles.**
- ➔ Context switch between warps is fast–About 1 or 2 cycles (1 nano-second on 1 GHz GPU)
    - ◆ The whole thread block has resources allocated on an SM by the compiler ahead of time

- ➔ Volta-Ampere:
    - ◆ 4 warp schedulers in each SM and 2 dispatchers in each scheduler
    - ◆ Can start instructions in up to 4 warps each clock and up to 2 **subsequent, independent** instructions in each warp. Up to 80 warp instructions to hide latency of warp add (10 cycles)

Occupancy

Idea: Need enough independent threads per SM to hide latencies

- ➔ Instruction latencies
- ➔ Memory access latencies

Occupancy: number of concurrent threads per SM

- ➔ Occupancy = active warps per SM / max warps per SM

Number of threads that fit per SM (max warps per SM) is determined by the hardware resources of the GPU.
Threads/block matters because (combined with the number of blocks) let's us know how many warps there are on the SM.

Occupancy

The number of active warps per SM is determined by the limiting resources

- ➔ Registers per thread
  - ◆ SM registers are partitioned among the threads
- ➔ Shared memory per thread block
  - ◆ SM shared memory is partitioned among the blocks
- ➔ Threads per thread block
  - ◆ Threads are allocated at thread block granularity

Needed occupancy depends on the code

- ➔ More independent work per thread -> less occupancy is needed
- ➔ Memory-bound codes tend to need more occupancy Higher latency than for arithmetic, need more work to hide it

Don’t need for 100% occupancy for maximum performance

Ampere

- ➔ max threads / SM = 2048 (64 warps)
- ➔ max threads / block = 1024 (32 warps)
- ➔ 32 bit registers / SM = 64k
- ➔ max shared memory / SM = 48KB

The number of blocks that run concurrently on a SM depends on the resource requirements of the block!

GK110 Occupancy

100% occupancy

- ➔ 2 blocks of 1024 threads
- ➔ 32 registers/thread
- ➔ 24KB of shared memory / block

50% occupancy

- ➔ 1 block of 1024 threads
- ➔ 64 registers/thread
- ➔ 48KB of shared memory / block

Conclusion

Block grid definition

Masking of GPU memory access times

➔ a GPU switches from one thread warp to another very quickly
➔ a GPU masks the latency of its memory accesses by multi-threading

Do not hesitate to create large numbers of small GPU threads

Ex.: to process an array of N elements, you can define:

➔ Threads dealing with ONE element each
➔ and a Grid of blocks of *N* threads in total

Or

➔ Threads dealing with *n* elements each
➔ and a Grid of blocks of *N/n* threads in total

Conclusions

Hierarchy of memory in a GPU

- ➔ Access to memory is the bottleneck.
- ➔ Access to memory should be taken into consideration when designing grids

Themes of this class

- ➔ Different level of memory
- ➔ Avoiding memory conflicts