Lecture 4 - GP-GPU and High Performance Computing

Programming on GP-GPU (2)

Review of GP-GPU Hierarchy

Thread Organization

- **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

Definitions

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

Characteristic	CPU (Ryzen 9)	GPU (Blackwell)
Latency Type	Low	High
Throughput Type	Low	High
Clock Speed	3.8 - 4.6 GHz (3.8 clocks/ns)	~2.0 – 2.4 GHz (2 clock/ns)
Arithmetic Latency	~1 ns	~4 ns
Main Memory Latency	~100 ns	~300 ns

Intel, AMD, and VIA instructions table

Balckwell with microbenchmarks

Latency and troughput (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.

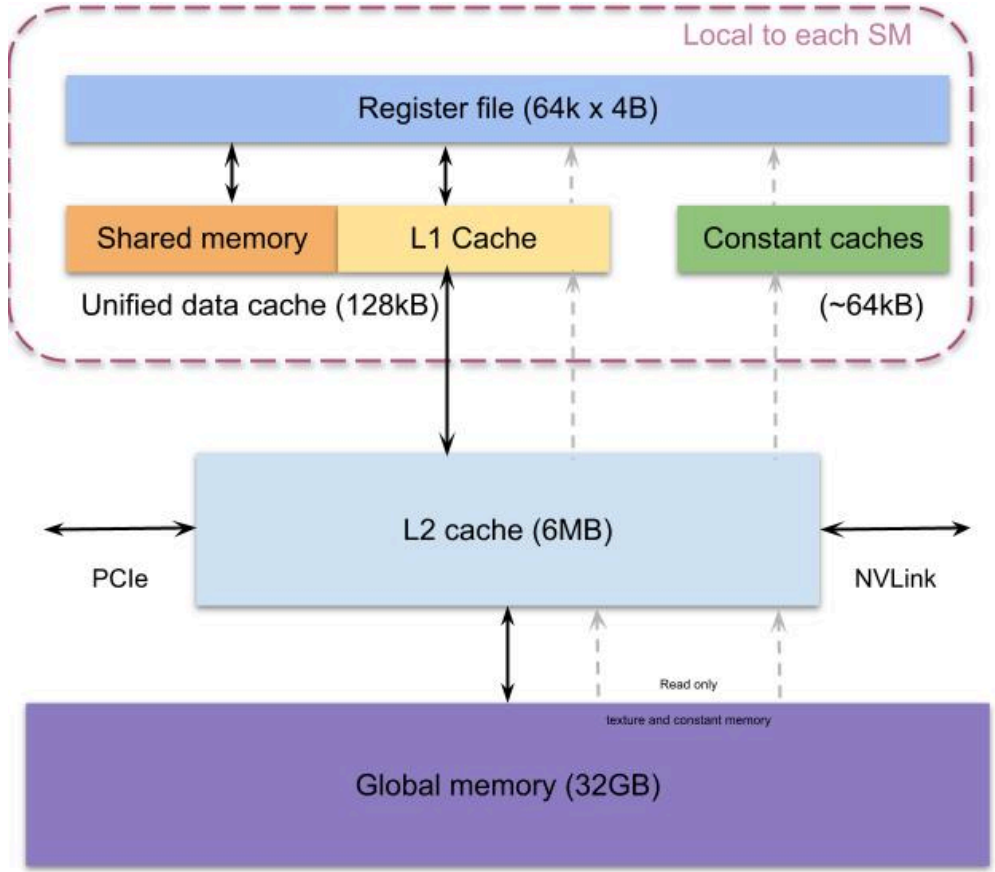
Cache Mechanics

- **Cache** is an area of memory between a larger memory pool and the processor
  - Often implemented at hardware level
  - 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
- **Cache Line**: Regions of memory that will always be cached together.
  - **Cache Hit**: Access found in the cache (fast access).
  - **Cache Miss**: Access not found in the cache (no performance gain).

GPU Memory Hierarchy - Types

Memory Type	Location	Accessibility / Scope	Relative Speed
Registers	Multi-processor (SM)	Accessible only by the **thread**	**Fastest**
Shared Memory	SM (Same hardware as L1 cache)	Accessible by any thread within the **block**	Very fast (~5ns)
Local Memory	Resides in **Global Memory**	Accessible only by the **thread**	150x slower
Global Memory	Separate hardware from GPU core	Accessible by **Host or Device**	150x slower
Constant Memory	Global Memory with special cache	Accessible by **Host or Device (Read only)**	Optimized for **broadcast** to threads in a warp

Memory organization



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, it should be process 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

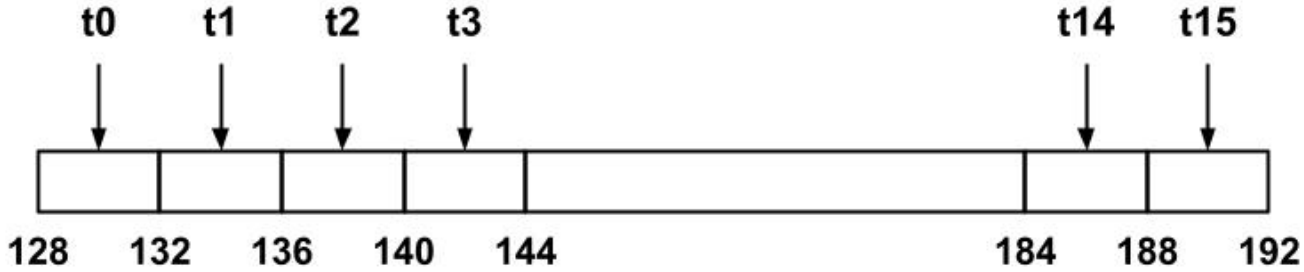
Accessing global memory efficiently

- 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.

Coalesced Memory Accesses

- Access patterns that align well with GPU hardware.
- Memory accesses are performed in large groups, called **Memory Transactions**, per **warp**.
- Coalesced accesses **minimize the number of cache lines read** during these transactions.
- GPU cache lines are **128 bytes** and are aligned.
- Memory coalescing is much more complicated in reality.

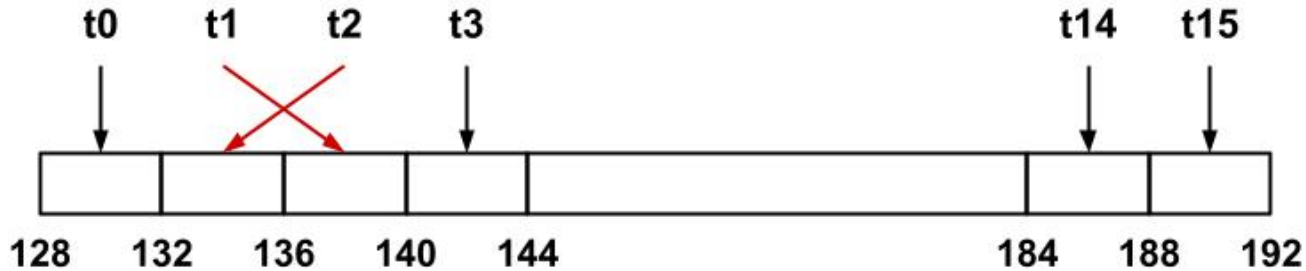
Coalesced access



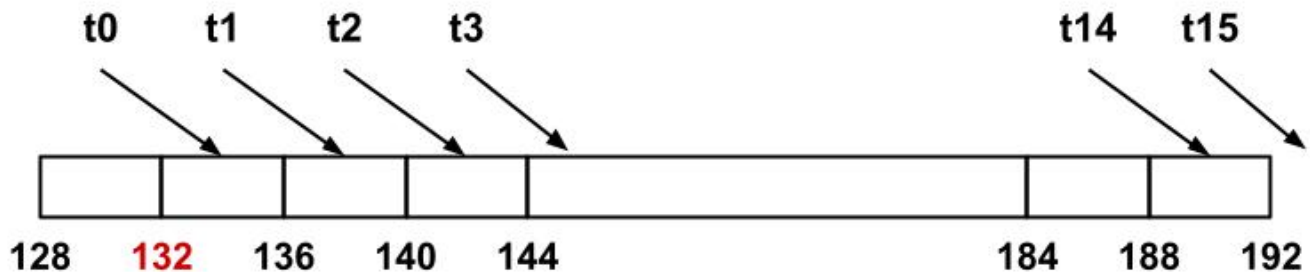
All threads participates

Some threads do not participate

Uncoalesced access



Permuted access by threads



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

Coalesced vs. Non-Coalesced Access

Coalesced Access

- Accesses are contiguous and aligned across the warp.
- Minimizes memory transactions.

Non-Coalesced Access

- Misaligned starting addresses.
- Accesses require more memory transactions.

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

Manipulating shared memory

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

- Dynamic allocation syntax

➜ 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

Common pattern in threads

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

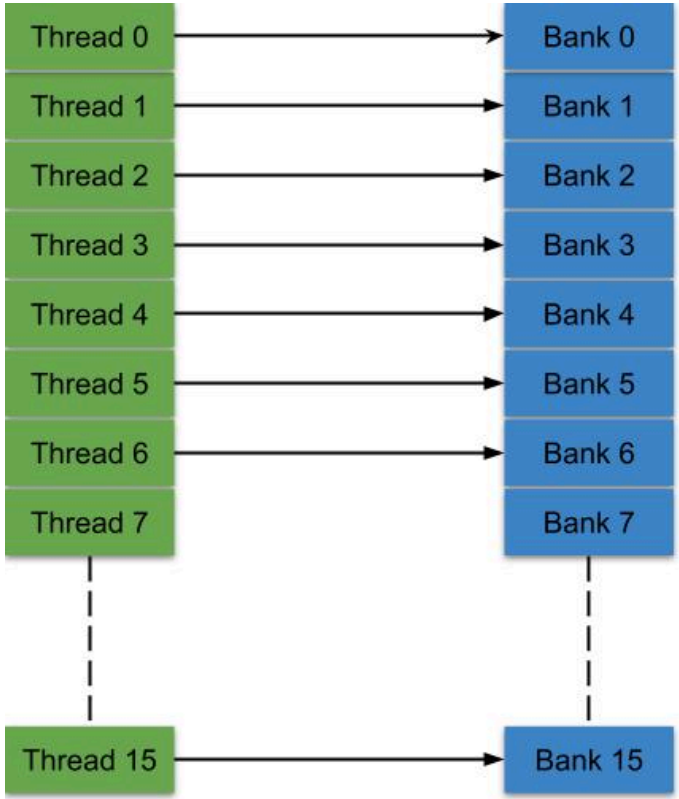
Bank Organization

- Shared memory is organized into **32 banks**.
- An element i (4 bytes) is in bank $i \pmod{32}$.
- A **bank conflict** occurs when 2 threads in a warp access different elements in the **same bank**.
- **Consequence**: Memory accesses are **serialized** instead of parallel (**very bad for performance**).

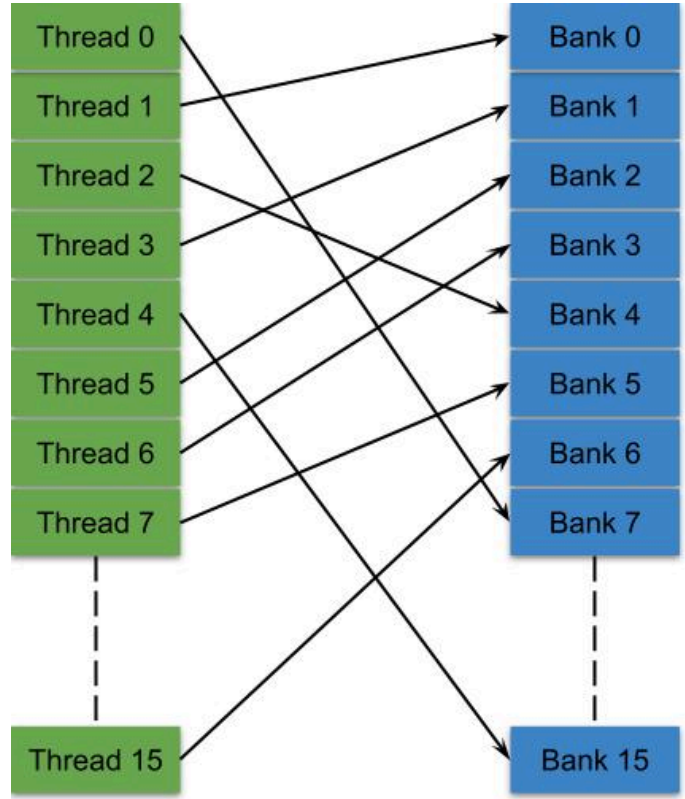
Stride and Conflicts

- **Stride**: Distance between the access of thread i and that of thread $i + 1$.
- **Stride 1**: Conflict-free (32 x 1-way).
- **Stride 2**: 16 x 2-way conflicts.
- **Stride 32**: 1 x 32-way conflict (worst case).

No bank conflict

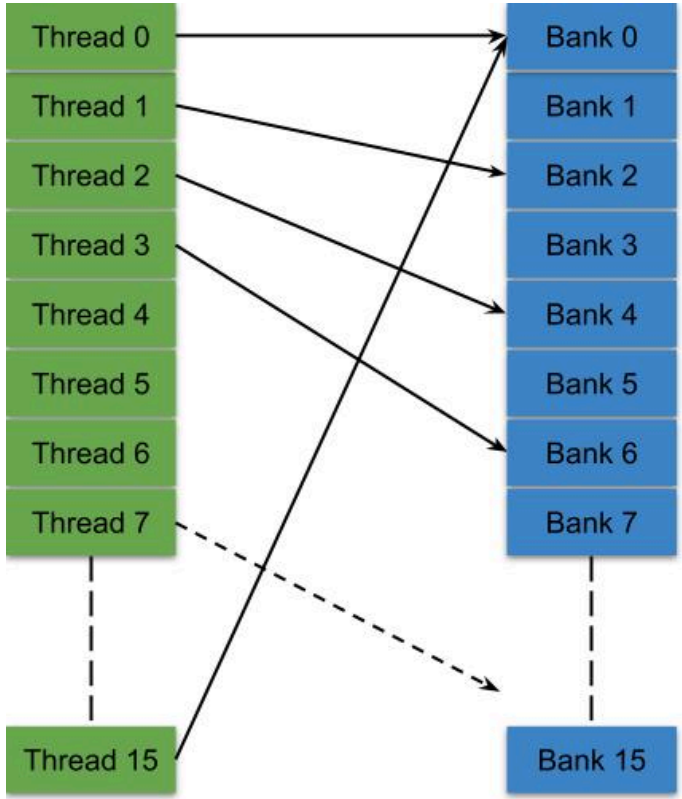


Linear addressing stride == 1

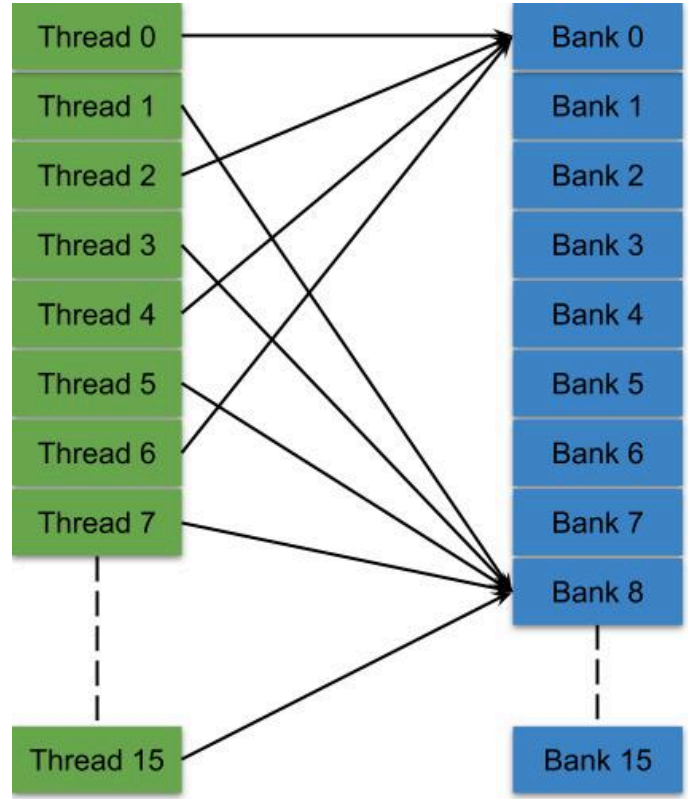


Random 1:1 Permutation

Bank conflict



Linear addressing stride == 2



Linear addressing stride == 8

5. Avoiding Bank Conflicts - Padding

- To solve the **stride 32** case, one can add one byte of **padding** to make the stride **33**.
- This ensures that thread i accesses index $33 \times i$, which is in bank $i \pmod{32}$ (conflict-free for $i < 32$).
- 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 **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.
- Data stored in local memory are 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

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

Dealing with memory

Synchronization

Ideal case for parallelism

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

Necessary when algorithms require **shared resources** and **communication**.

- **Deadlock**: Processes can depend on each other and become stuck. Each member of a group can wait for another member, including itself, to take action.

Synchronization: example

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;
```

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

Synchronization in CUDA

Usually use the `__syncthreads()` function to sync threads **within a block**

- Only works at the **block level**.
- SMs are separate from each other so cannot there is no way of doing 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 Implementation

CUDA provides built in atomic operations

- `atomic<Op>(float *address, float val)` (e.g., `atomicAdd`).
- Replace `<op>` with one of: Add, Sub, Exch, Min, Max, Inc, Dec, And, Or, Xor

Example:

```
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 Level Parallelism (ILP)

Instruction Dependency

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

> 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

- **Goal**: Minimize the sequential nature of the code and the number of memory transactions.

ILP example: naive approach

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

Naive/sequential assembly (loads, compute, store for element 0 then element 1):

```
fld dword [x]       ; load x[0]
fadd dword [y]      ; add y[0]
fstp dword [z]      ; store z[0]

fld dword [x+4]     ; load x[1]
fadd dword [y+4]    ; add y[1]
fstp dword [z+4]    ; store z[1]
```

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

ILP example: interleaved approach

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

Interleaved assembly that exposes ILP (overlap loads with other work):

```
fld dword [x]       ; load x[0]
fld dword [x+4]     ; load x[1]
fadd dword [y]      ; add y[0] to st(0)
fadd dword [y+4]    ; add y[1] to st(1)
fstp dword [z]      ; store z[0]
fstp dword [z+4]    ; store z[1]
```

➜ the loads for different elements and the arithmetic can be overlapped. On GPUs this reduces time a single thread waits on memory and reduces pressure on warp scheduling when there are fewer ready warps.

Warp Schedulers

Warp schedulers select a warp that is ready to execute its next instruction and dispatch it to the execution units. Every SM has one or more warp schedulers.

- Each scheduler exposes dispatch units (often two dispatchers per scheduler on modern NVIDIA GPUs).
- When a scheduler issues an instruction it is executed by all threads in the chosen warp.
- If one or more threads in the issuing warp stall (for example on a long-latency global memory access), the warp is marked inactive and the scheduler switches to another eligible warp. If no eligible warps remain the SM may idle.
- Context switching between warps is very fast (on the order of 1–2 cycles, architecture dependent). This extremely cheap switch helps the GPU hide instruction and memory latencies by keeping many warps ready to run.
  - Note: when a thread block is scheduled on an SM, its resources (registers, shared memory, etc.) are allocated there for the block's lifetime.

Warp schedulers (Volta-Ampère)

- On Volta and Ampere-family GPUs an SM contains 4 warp schedulers, each with multiple dispatch units (e.g. 2). In aggregate the SM can issue instructions for several warps per cycle and may dual-issue independent instructions in a warp when possible.
- This high issue bandwidth means many warp-instructions can be in-flight simultaneously, which is what allows the hardware to hide multi-cycle latencies.
- Up to 80 warp instructions to hide latency of warp add (10 cycles).

Occupancy

Idea: have enough independent threads per SM to hide latencies:

- Instruction latencies
- Memory access latencies

Occupancy – fraction of the device's warp capacity that can be active for a kernel:

$$extOccupancy = \frac{\text{Active warps per SM}}{\text{Max warps per SM}}$$

How to estimate active warps per SM (practical):

- Determine the resources a block needs: registers per thread, shared memory per block, and threads per block.
- Calculate how many blocks can reside on the SM given each resource limit; the smallest of these limits determines blocks/resident_SM.
- Active warps per SM = (threads per block × blocks resident per SM) / warp_size

Occupancy (2)

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 100% occupancy for maximum performance

Ampere Architecture Limits

- 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 an SM depends on the **resource requirements of the block**!

GK110 Occupancy Examples

Occupancy	Blocks / Threads	Registers / Thread	Shared Memory / Block
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

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

Block Grid Definition

- **Do not hesitate to create large numbers of small GPU threads**.
  - **Ex. 1**: Define **Threads** dealing with **ONE** element each AND a Grid of blocks of N threads in total.
  - **Ex. 2**: Define **Threads** dealing with n elements each AND a Grid of blocks of N/n threads in total.

–

Final 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