Course Schedule (Updated)

Date	Topic
2021-08-31	Course Introduction
2021-09-02	Intoroduction to Parallel Computing
2021-09-07	GPU Architecture
2021-09-09	Fundamentals of CUDA1
2021-09-14	Fundamentals of CUDA2
2021-09-16	CUDA Threads 1
2021-09-21	CUDA Threads 2
2021-09-23	CUDA Memory Model 1
2021-09-28	CUDA Memory Model 2
2021-09-30	Performance Considerations
2021-10-05	Parallel Algorithm: Convolution
2021-10-07	Parallel Algorithm: Sparse Matrix Computation
2021-10-12	Parallel Algorithm: Graph Search
2021-10-14	Application case study: Deep Learning
2021-10-19	Midterm exam
2021-10-21	Midterm exam
2021-10-26	Multicore Architecture
2021-10-28	Multicore Architecture
2021-11-02	Parallel Programming Models
2021-11-04	Parallel Programming Basics
2021-11-09	Parallel Programming Basics
2021-11-11	Pthread
2021-11-16	OpenMP
2021-11-18	OpenMP
2021-11-23	Synchronization
2021-11-25	Cache Coherence
2021-11-30	Cache Coherence
2021-12-02	Memory Consistency
2021-12-07	Interconnection Networks
2021-12-09	Final exam

Phase1: GPU

Phase2: Multicore CPU

GPU Architecture

Prof. Seokin Hong

Agenda

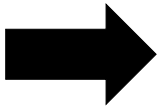
- Parallel Execution in Modern Processors
- Multi-threading for Hiding Memory Latency
- GPU Architectures

Parallel Execution in Modern Processors

Example Program

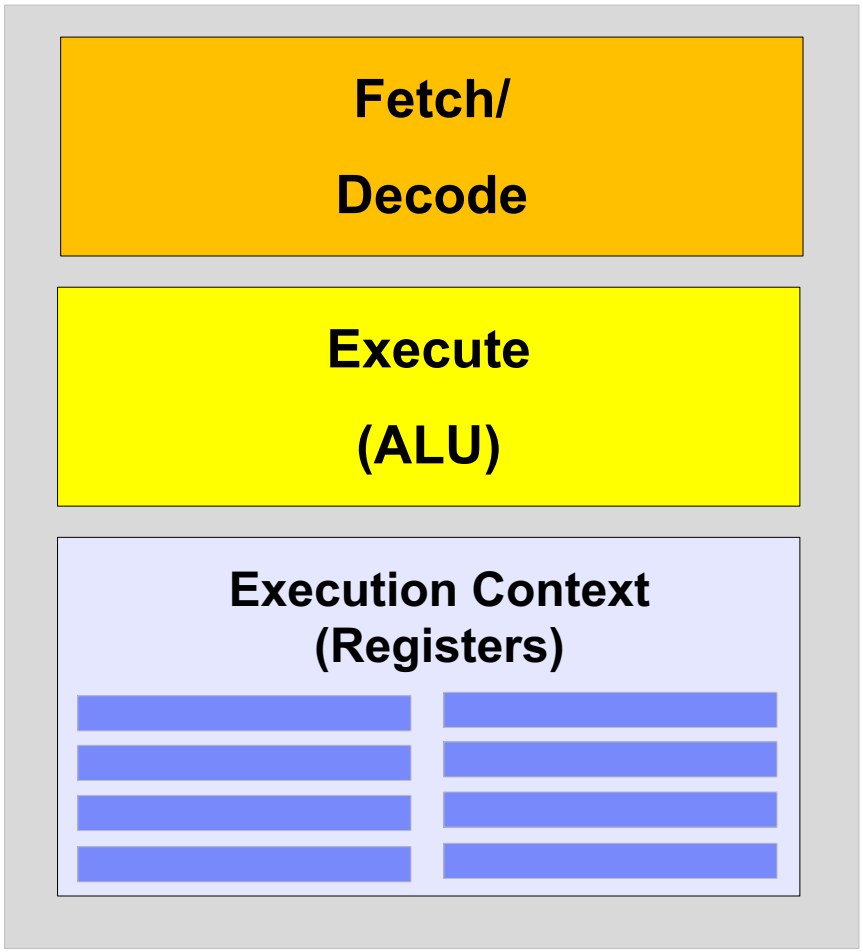
```
void mul(int N, float *x, float *result)
{
        for(int i=0; i<N; i++)
        {
                float value=0;
                for(int j=0; j<N; j++)
                    value += x[i] * j;
                result[i]= value;
         }
}
```

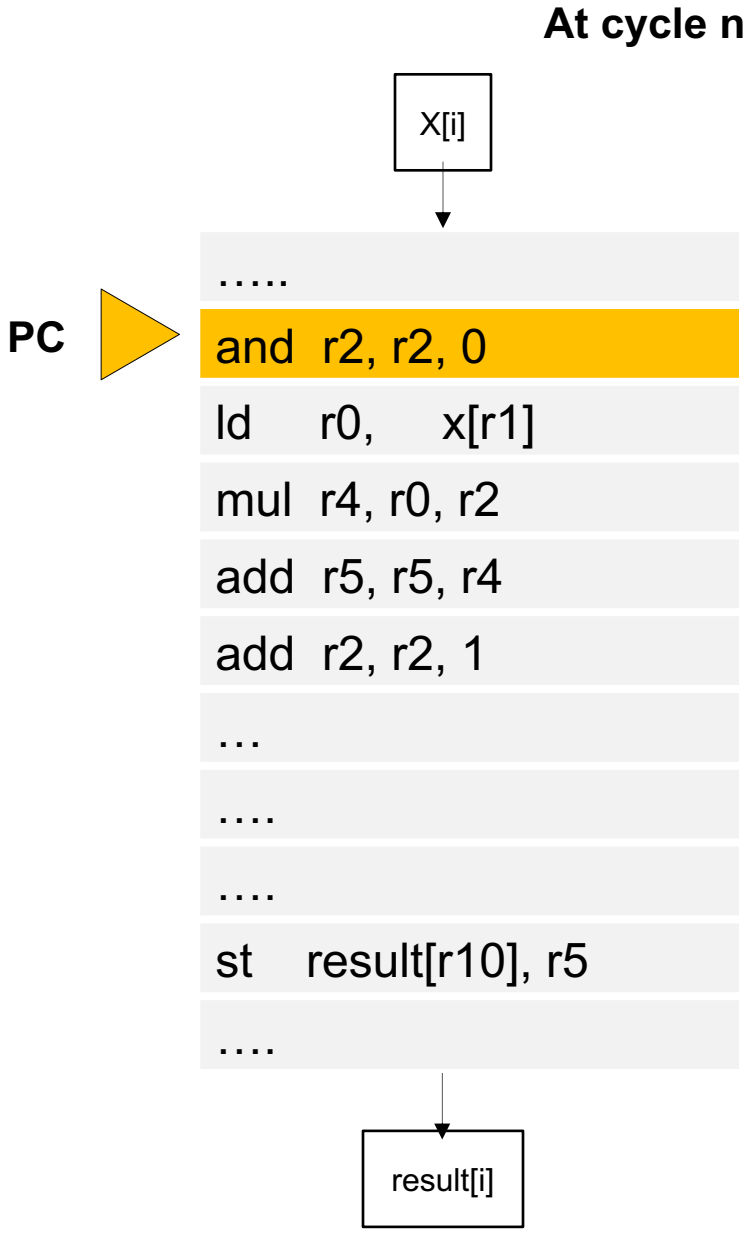
Compile



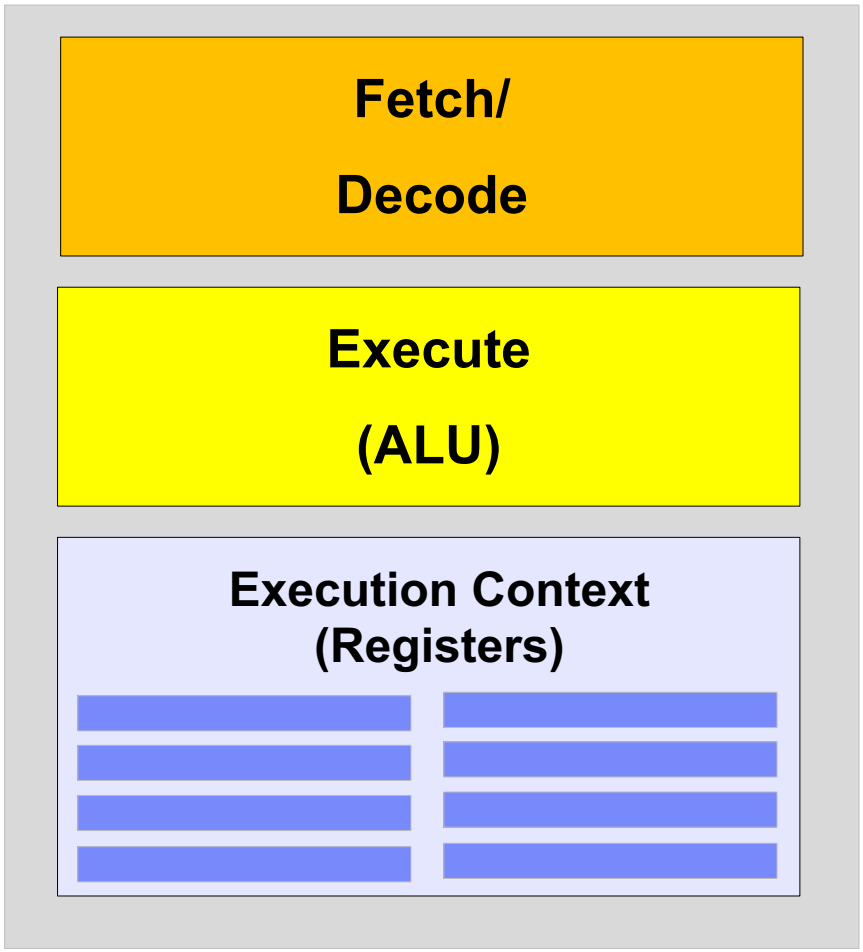
```
…..
and  r2, r2, 0
ld     r0,     x[r1]
mul  r4, r0, r2
add  r5, r5, r4
add  r2, r2, 1
…
….
….
st    result[r10], r5
….
```

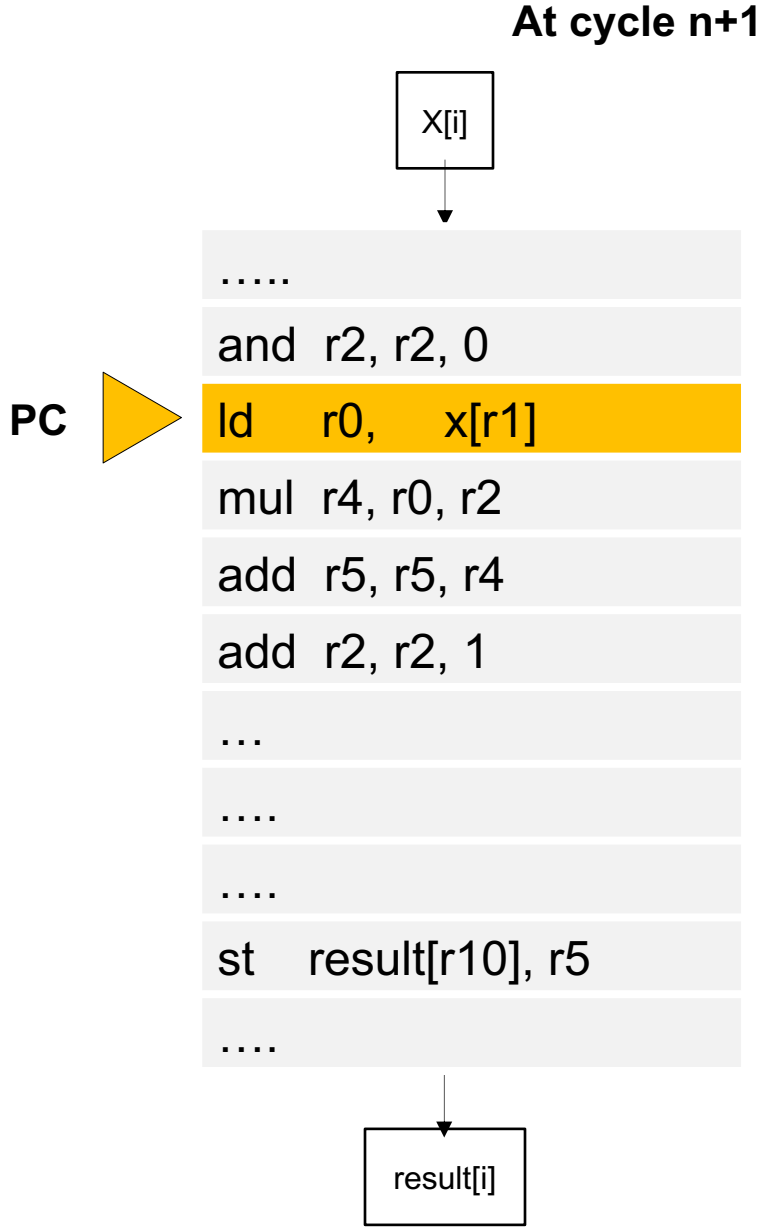
Execute Program on a simple processor



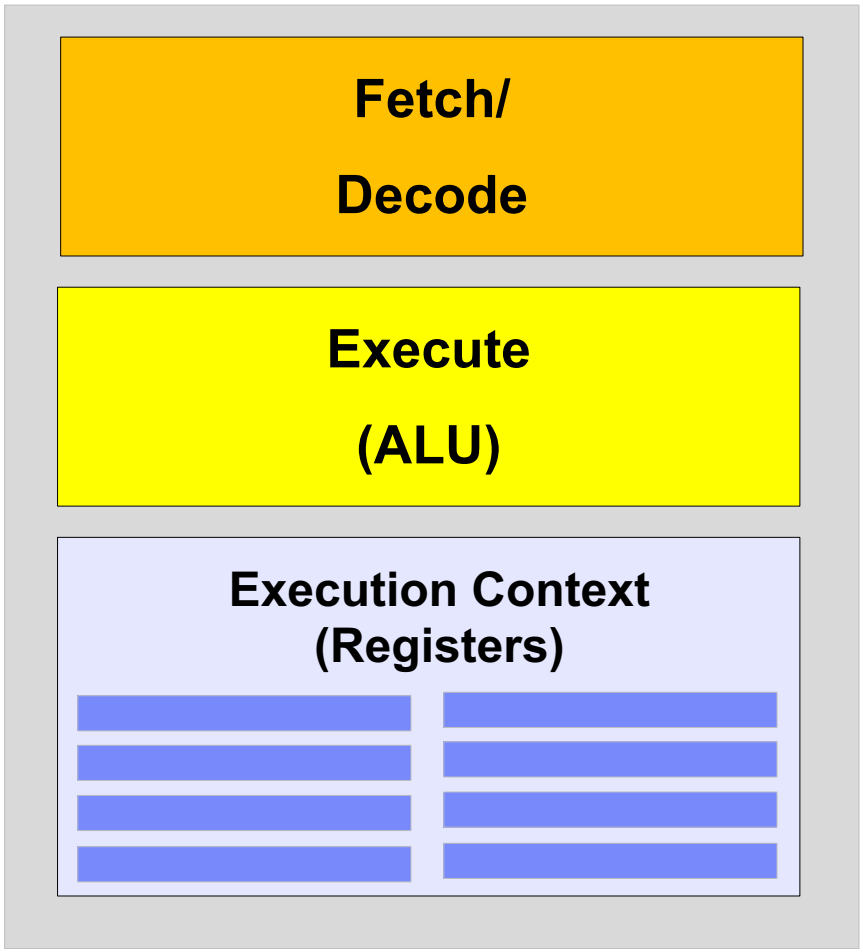


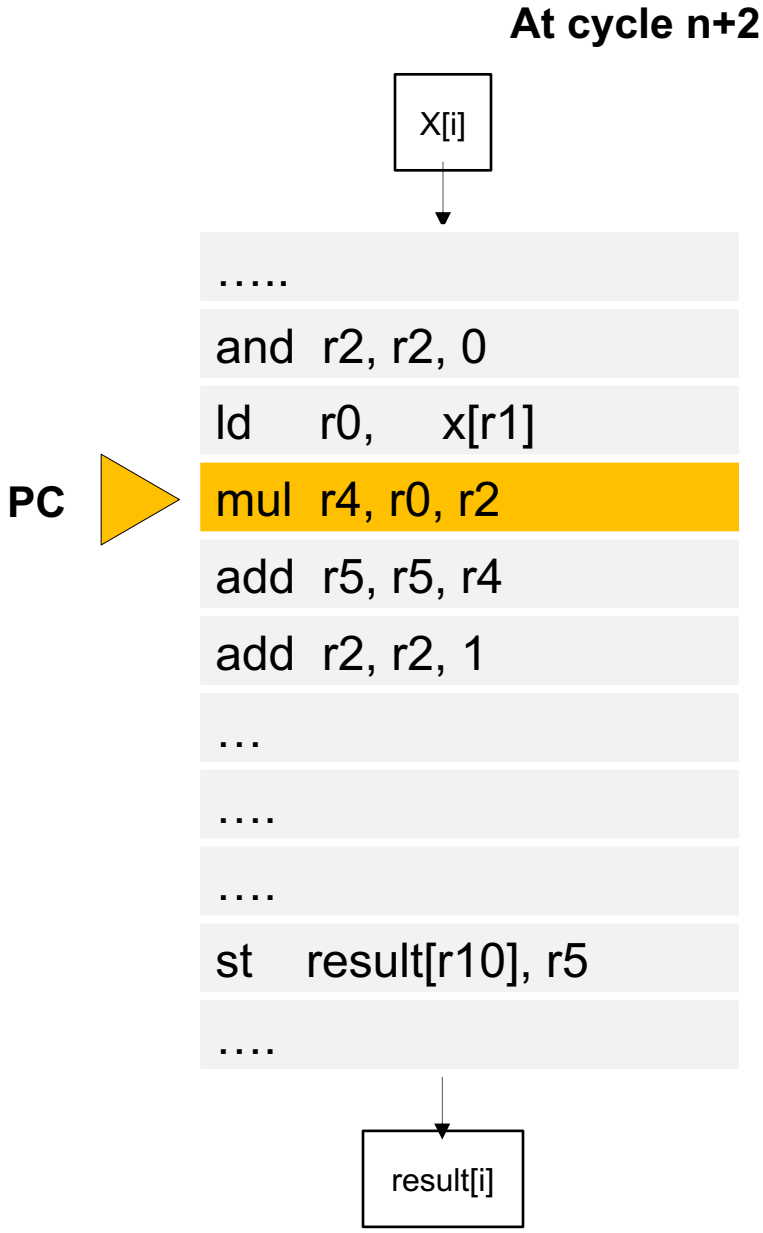
Execute Program on a simple processor





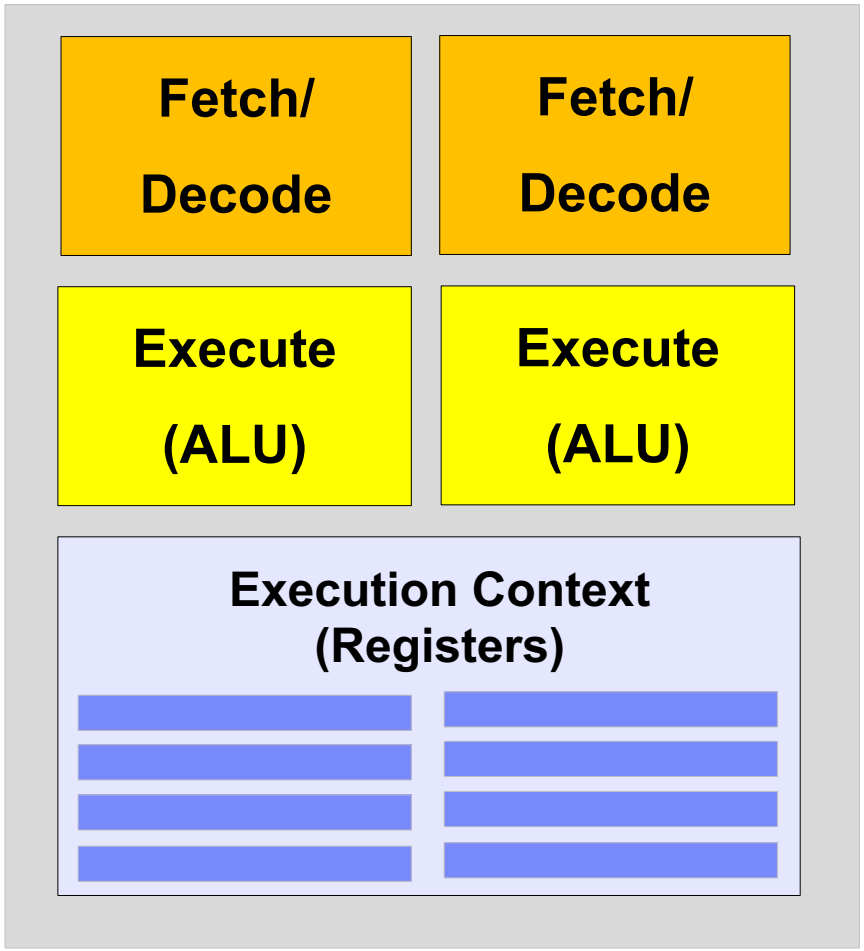
Execute Program on a simple processor

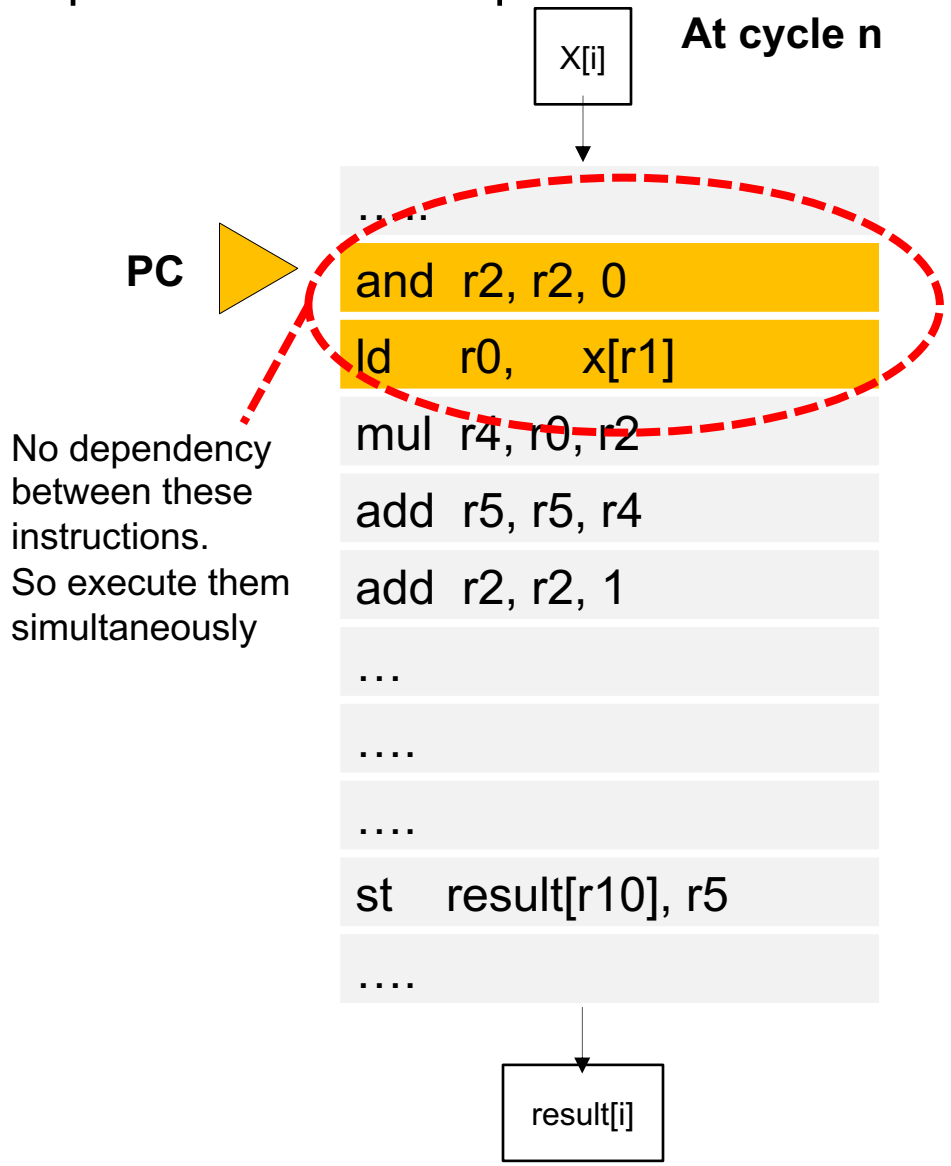




Execute Program on a Superscalar processor

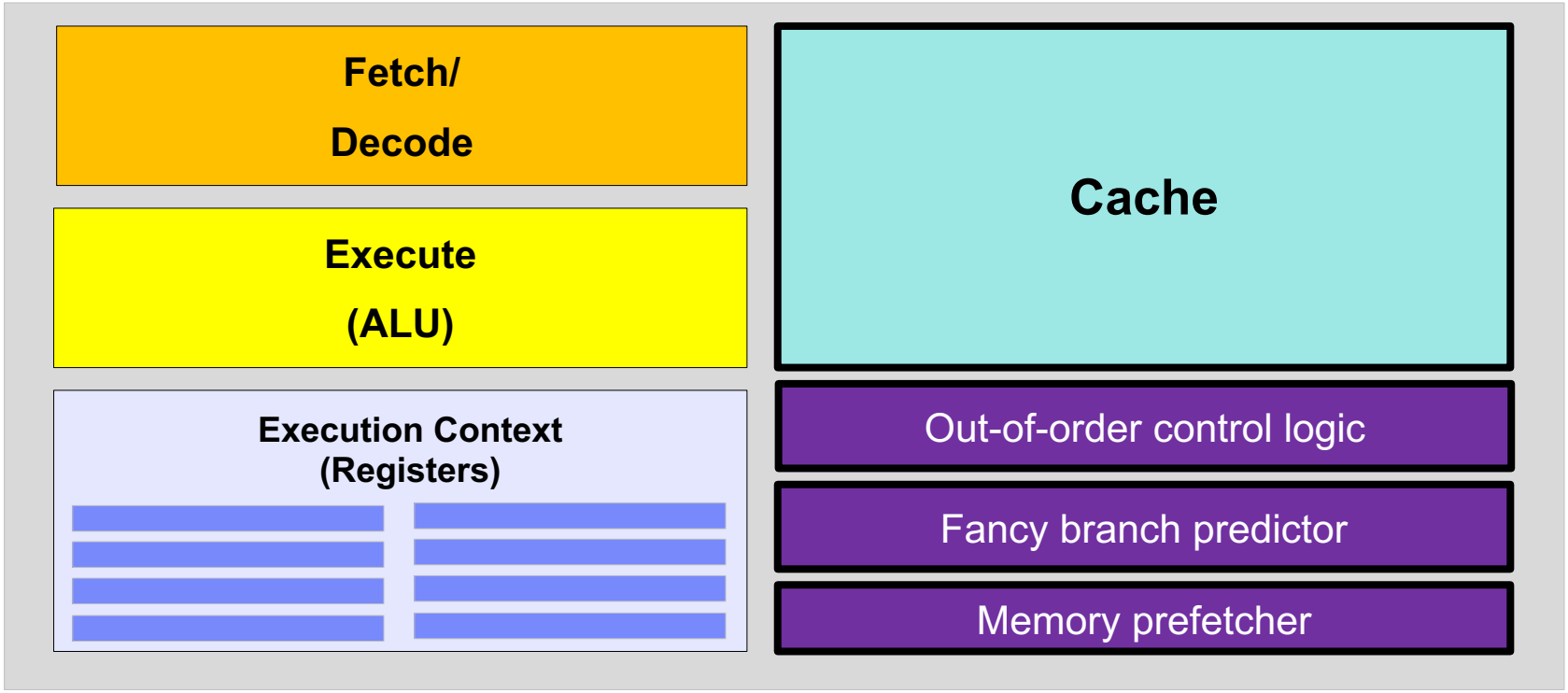
- **Exploit ILP**: Decode and execute multiple instrucitons in parallel





Pre multi-core era

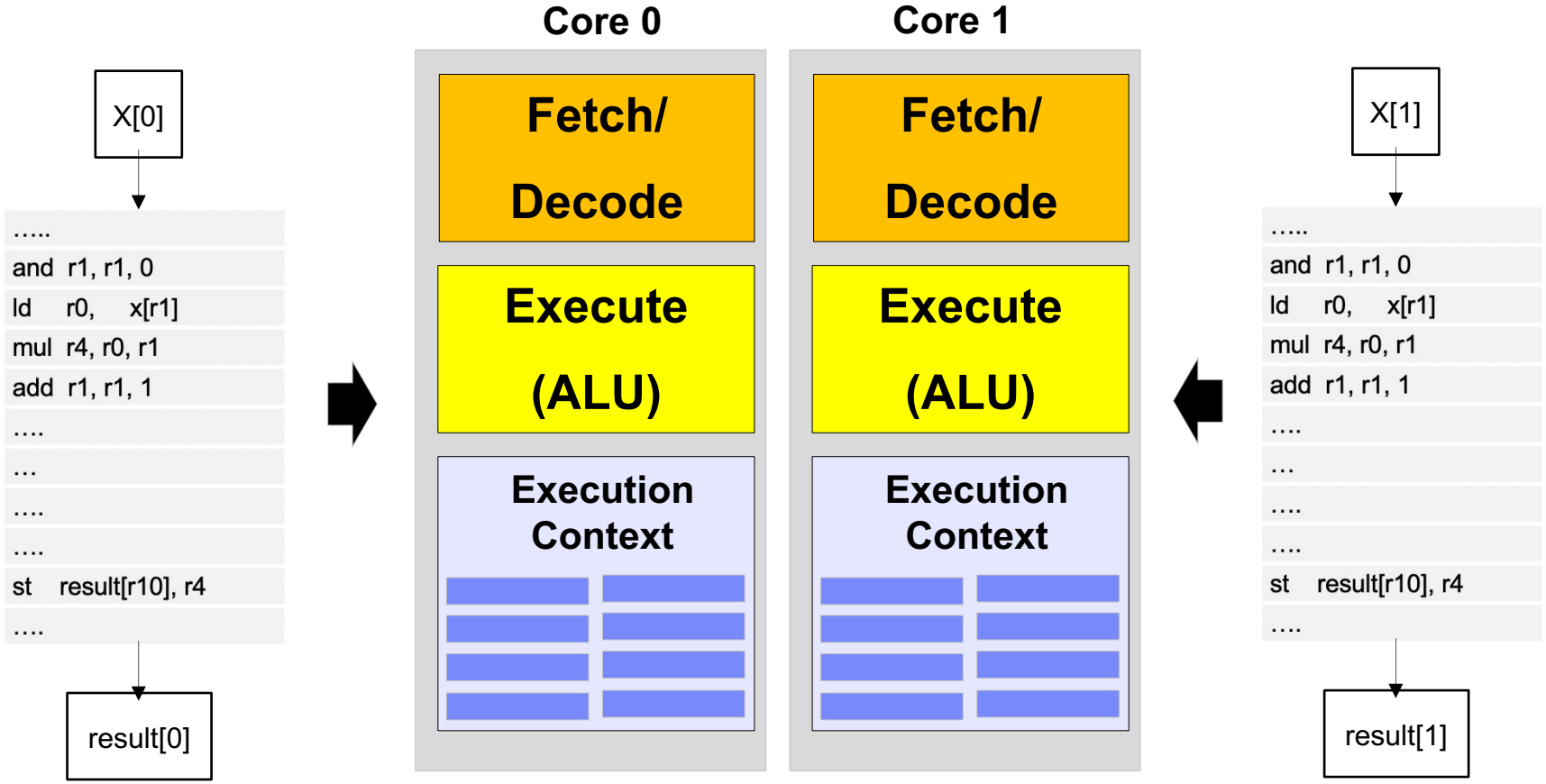
- Majority of transistors are used to perform operations that **help a single instruction stream run fast**



- More transistors → Larger cache, smarter out-of-order logic → smarter branch predictor, etc…
- **This approach has fundemental limitations:** Power wall, Diminishing gain with ILP

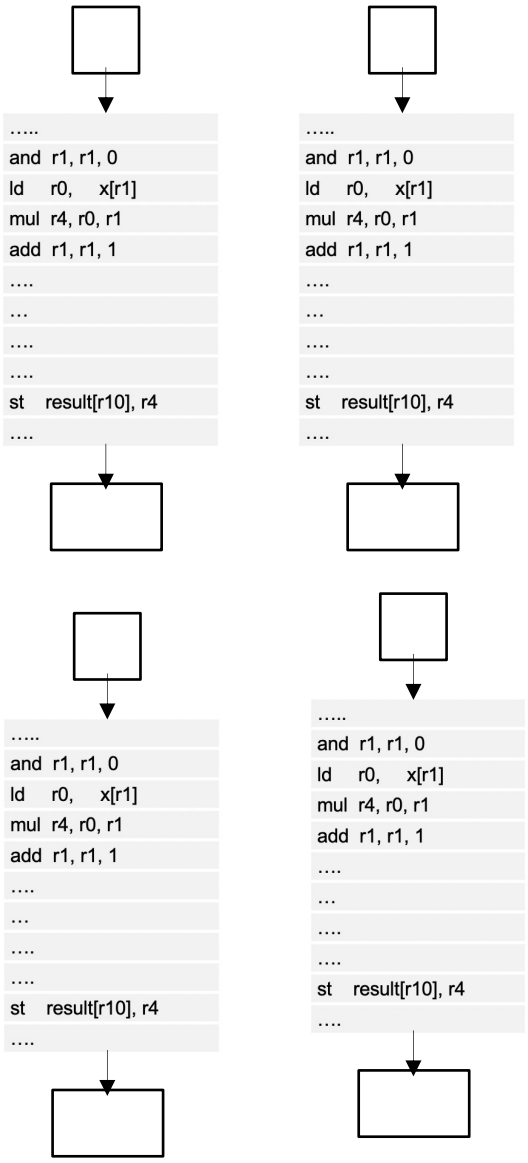
Multi-core

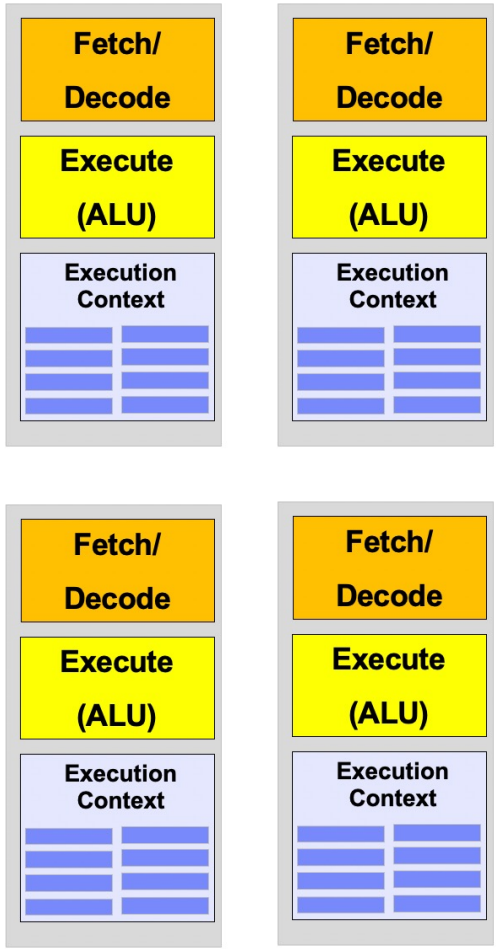
- Use increasing transistor count to **add more cores** to processor rather than use transistors to accelerates a single instruction stream



- Each core can be slower than a high-performance core (e.g., 0.75 times as fast)
- But, overall performance of two cores will be higher (e.g., 0.75 x 2 =1.5)

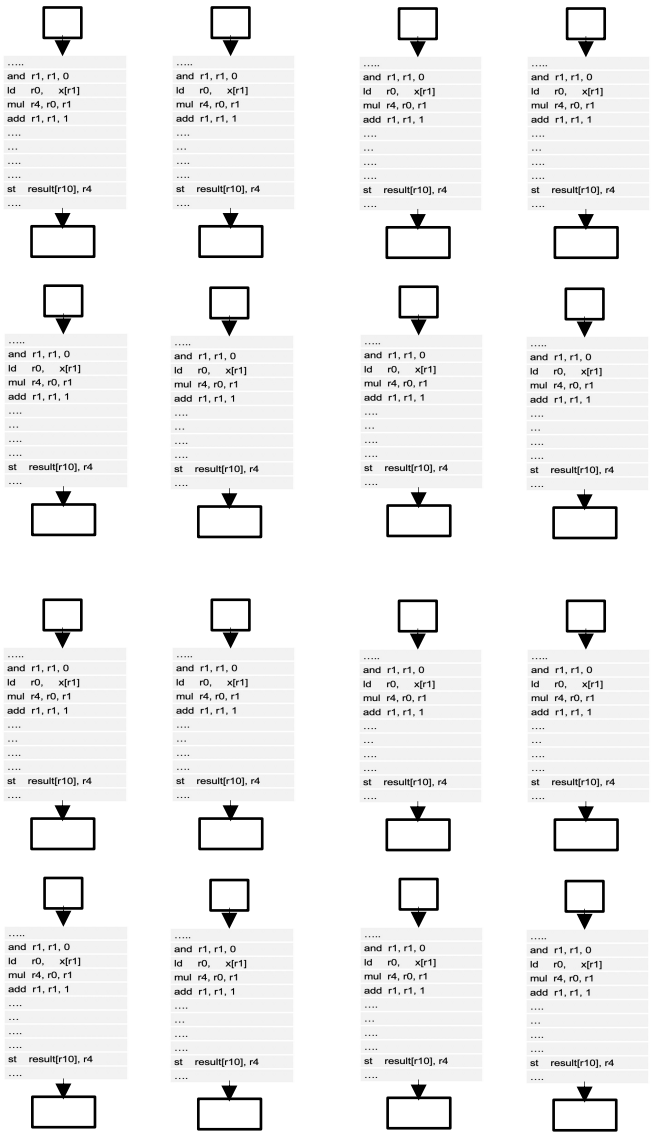
Four cores: compute 4 elements in parallel





Four cores run four simultaneous instruction streams

Sixteen cores: compute 16 elements in parallel



Sixteen cores run sixteen simultaneous instruction streams

128 cores?

128 cores → 128 simultaneous instruction streams

But, how do you feed all these cores? → **Data-level Parallelism**

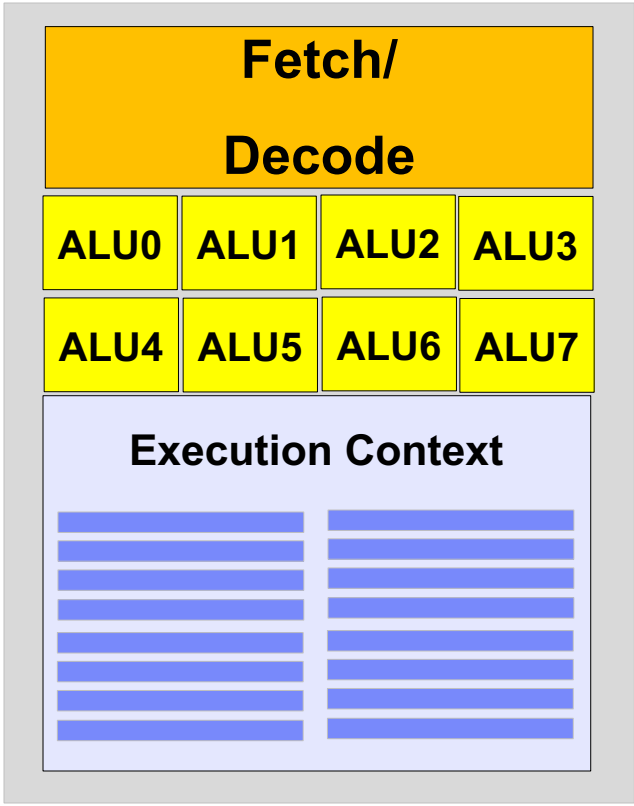
Interesting property of Example Program

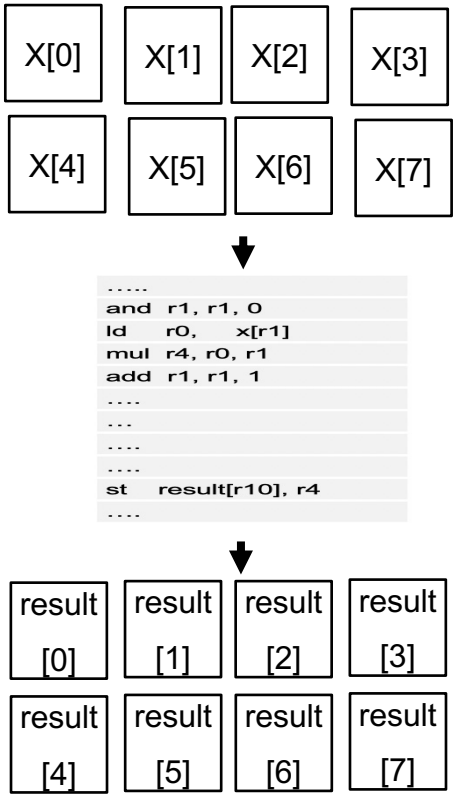
- Parallelism is across iterations of the loop
- All the iterations of the loop **do the same thing**

```
void mul(int N, float *x, float *result)
{
    for(int i=0; i<N; i++)
    {
        float value=0;
        for(int j=0; j<N; j++)
            value += x[i] * j;
        result[i]= value;
    }
}
```

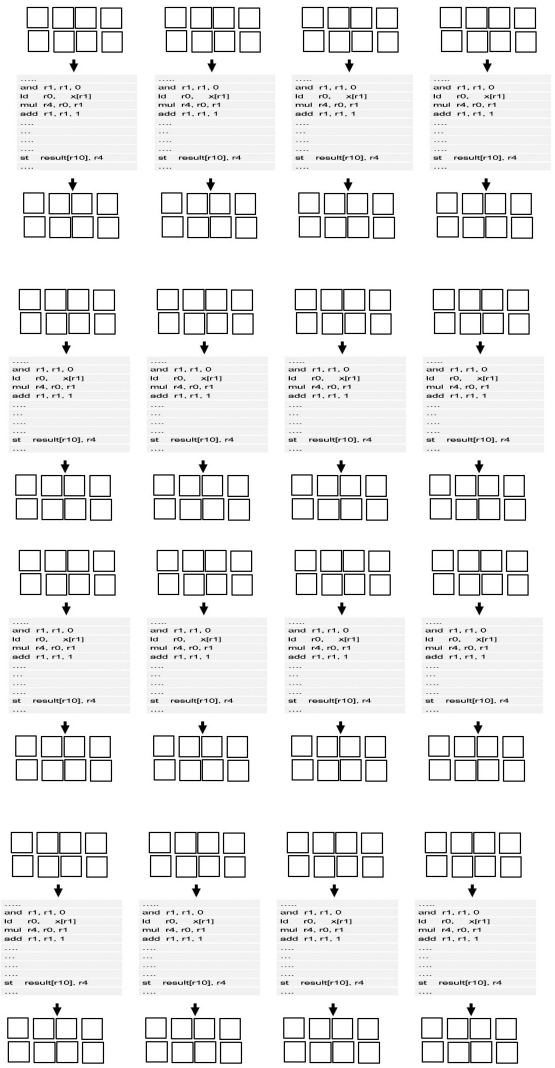
Add ALUs to increase compute capability

- **SIMD (Single Instruction Multiple Data)** Processing
  - Share cost of fetch / decode across many ALUs
  - Add ALUs and execute the **same instruction** on them **with different data**

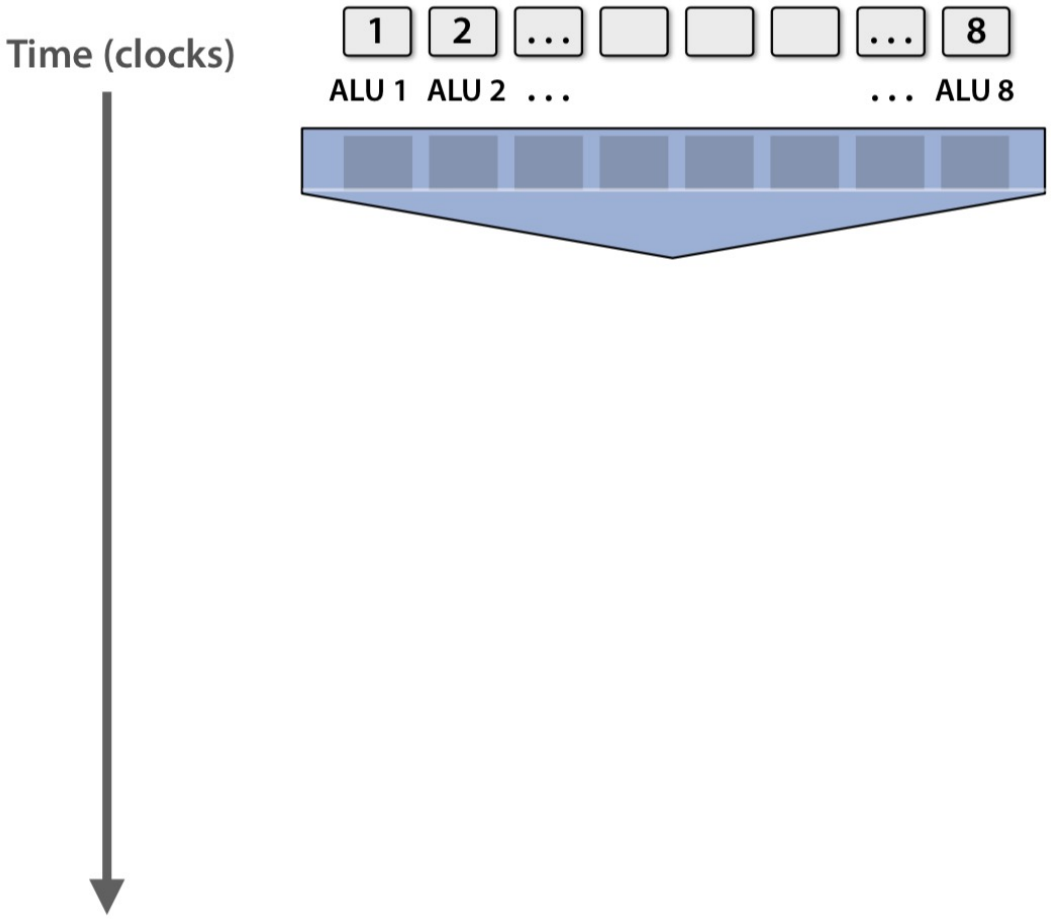




16 SIMD cores: 128 elements in parallel

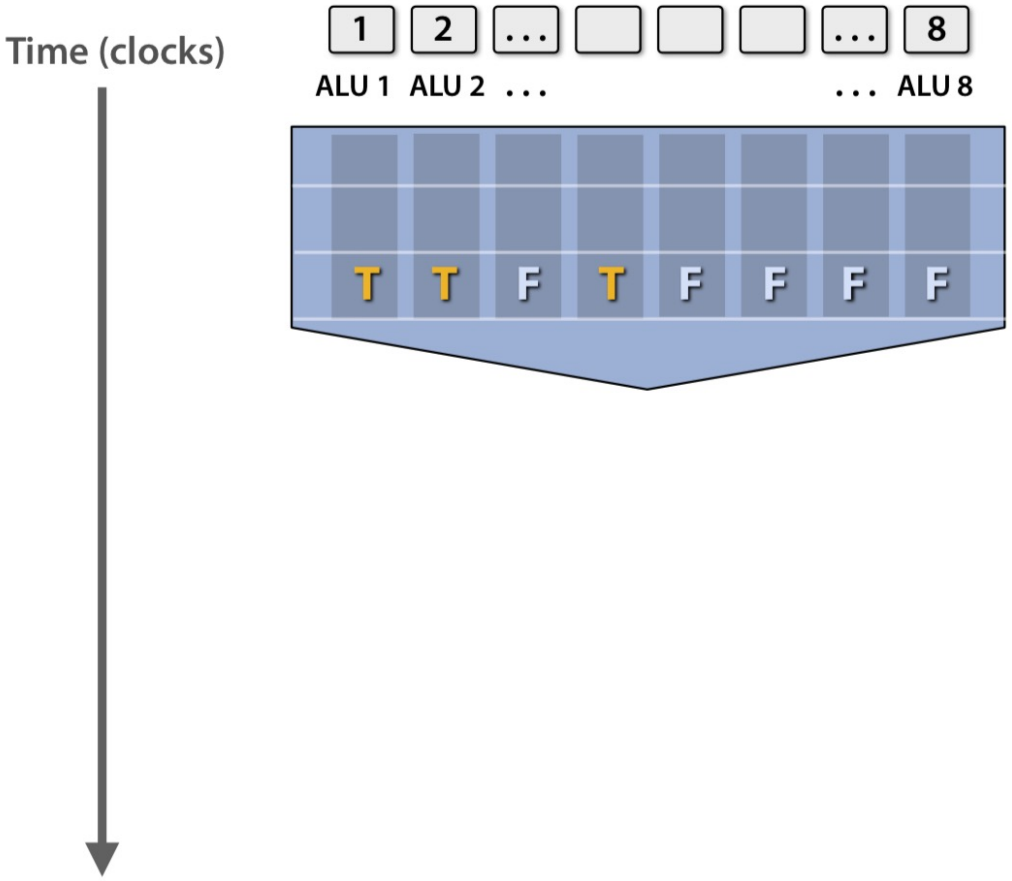


What about conditional execution in SIMD?




```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);

   tmp *= kMyConst1;

   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;

   x = 2.f * tmp;
}

<resume unconditional code>


result[i] = x;
```

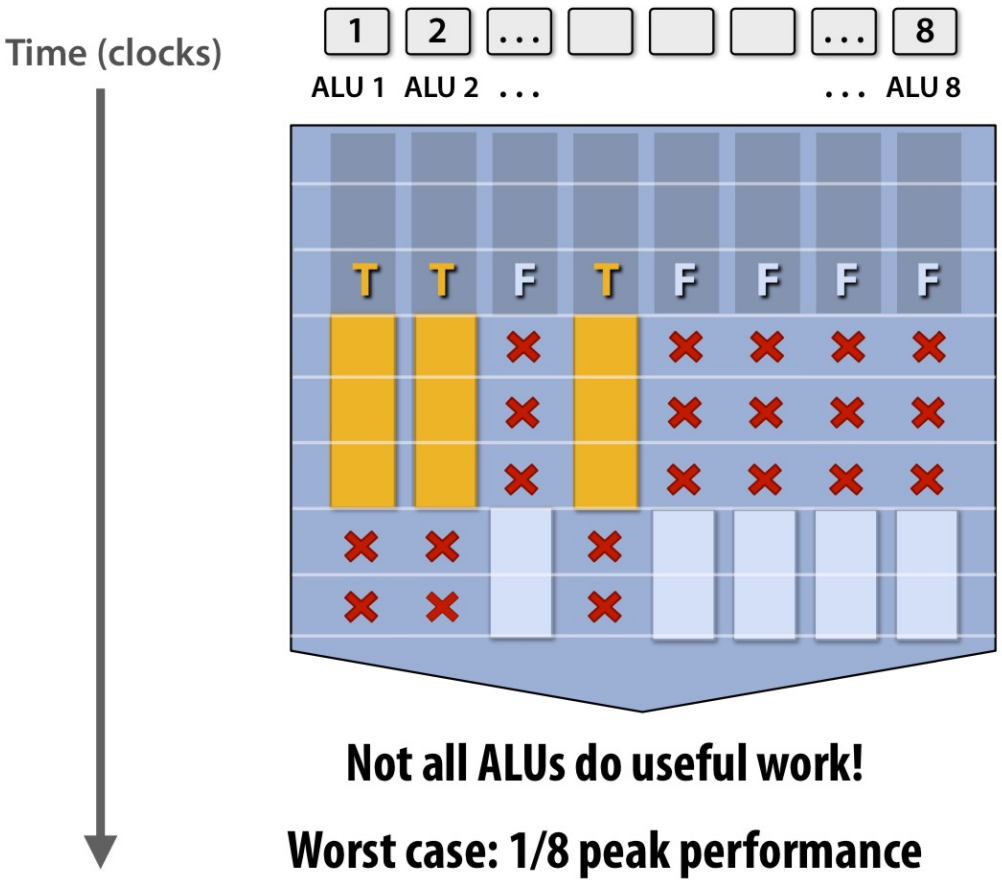
What about conditional execution in SIMD?




```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);

   tmp *= kMyConst1;

   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;

   x = 2.f * tmp;
}

<resume unconditional code>


result[i] = x;
```

What about conditional execution in SIMD?

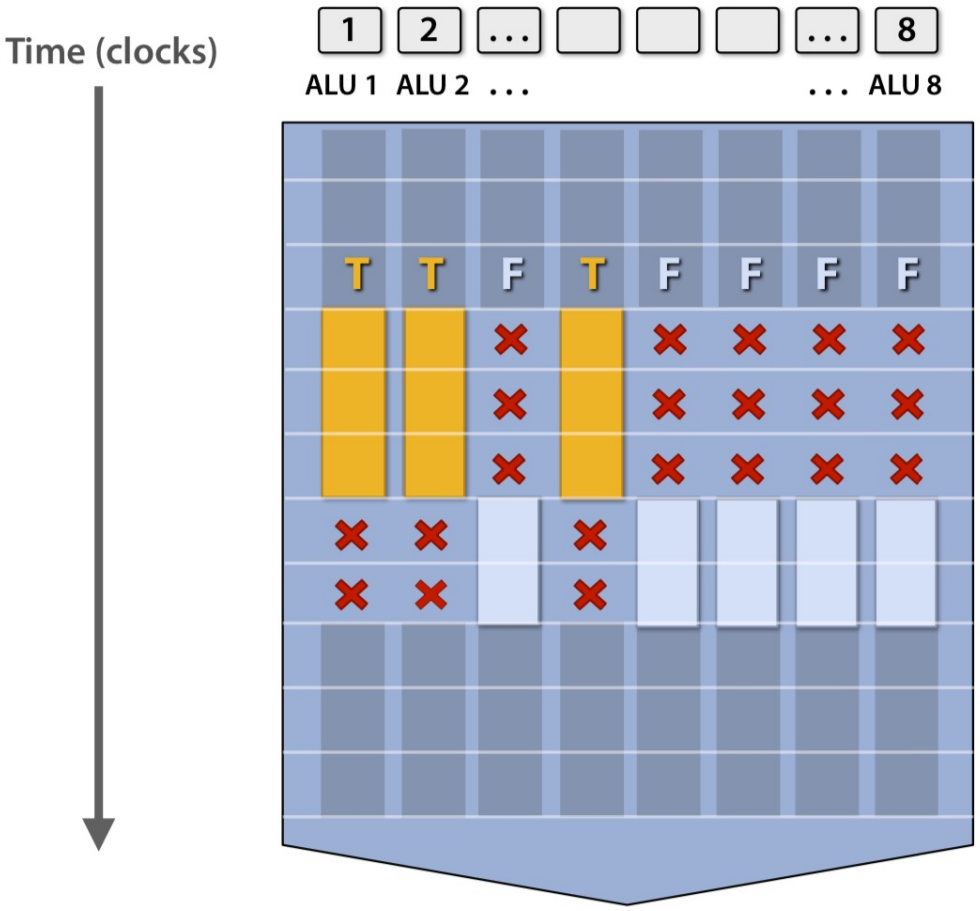
- Mask (discard) output of ALU




```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

What about conditional execution in SIMD?

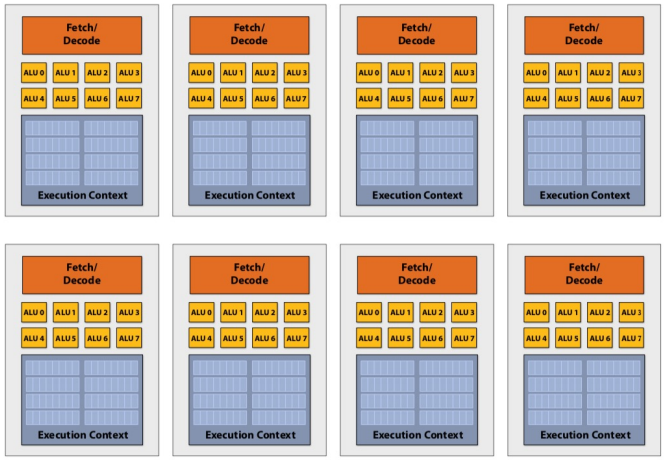
- After branch: continue at full performance



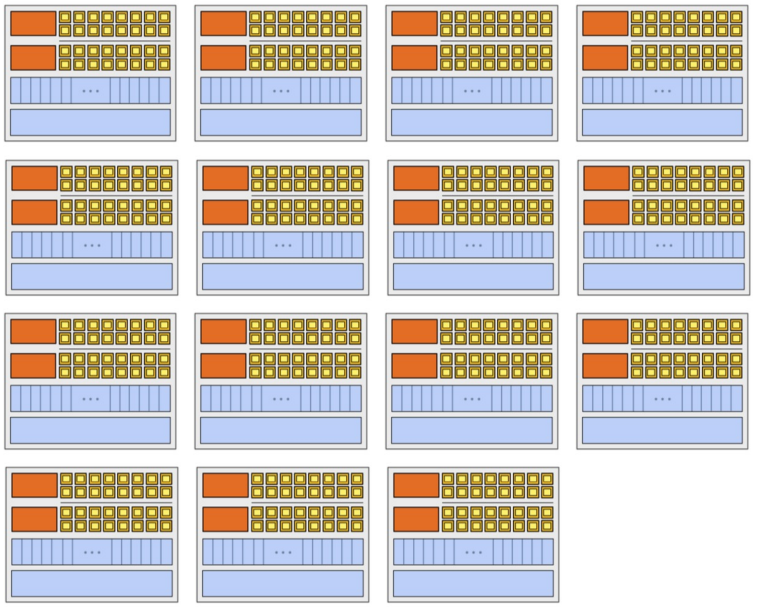

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

Examples:

- **Intel Core i9 (Coffee Lake)**
  - 8 cores
  - 8 SIMD ALUs per core



- **NVIDIA GTX480**
  - 15 cores
  - 32 SIMD ALUs per core
  - 1.3 TFLOPS



Summary

- Several forms of parallel execution in modern processors
  - **Multi-core**: use **multiple processing cores**
    - Provides thread-level parallelism: simultaneously execute a completely different instruction stream on each core
    - Software decides when to create threads (e.g., via pthread API)
  - **SIMD**: use **multiple ALUs** controlled by same instruction stream (within a core)
    - Efficient design for data-parallel workloads by exploting DLP **(Data-level Parallelism)**
    - Vectorization can be done by compiler or at runtime by hardware
  - **Superscalar**: exploit **ILP (Instruction-level Parallelism)** within an instruction stream
    - Process different instructions from the same instruction stream in parallel (within a core)
    - Parallelism dynamically discovered by hardware during execution

Multi-threading for hiding memory latency

Terminology

- **Memory Latency**
  - The amount of time for a memory request (from., load, store) to be serviced by the memory system
  - Example: 100 cycles, 100nsec

- **Memory Bandwidth**
  - The rate at which the memory system can provide data to a processor
  - Example : 20 GB/s

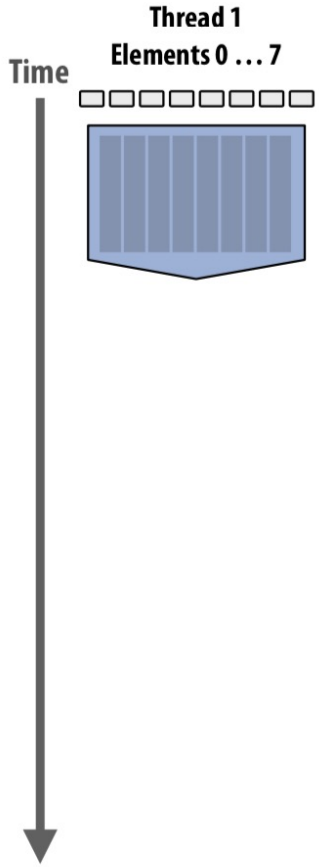
- **Stall**
  - A Processor “stalls” when it cannot run the next instruction in an instruciton stream because of a dependency on a previous instruction
  - Accessing memory is a major source of stalls

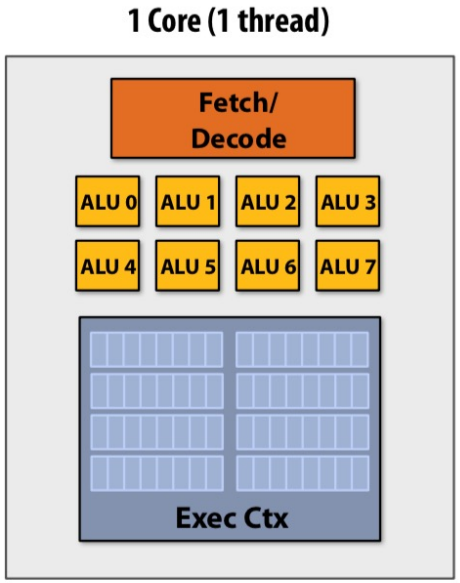
  - Memory latency : more than 100 cycles

Slide credit : CMU 15-418/15-618

Hiding stalls with multi-threading

- **Idea:** interleave processing of multiple threads on the same core to hide stalls

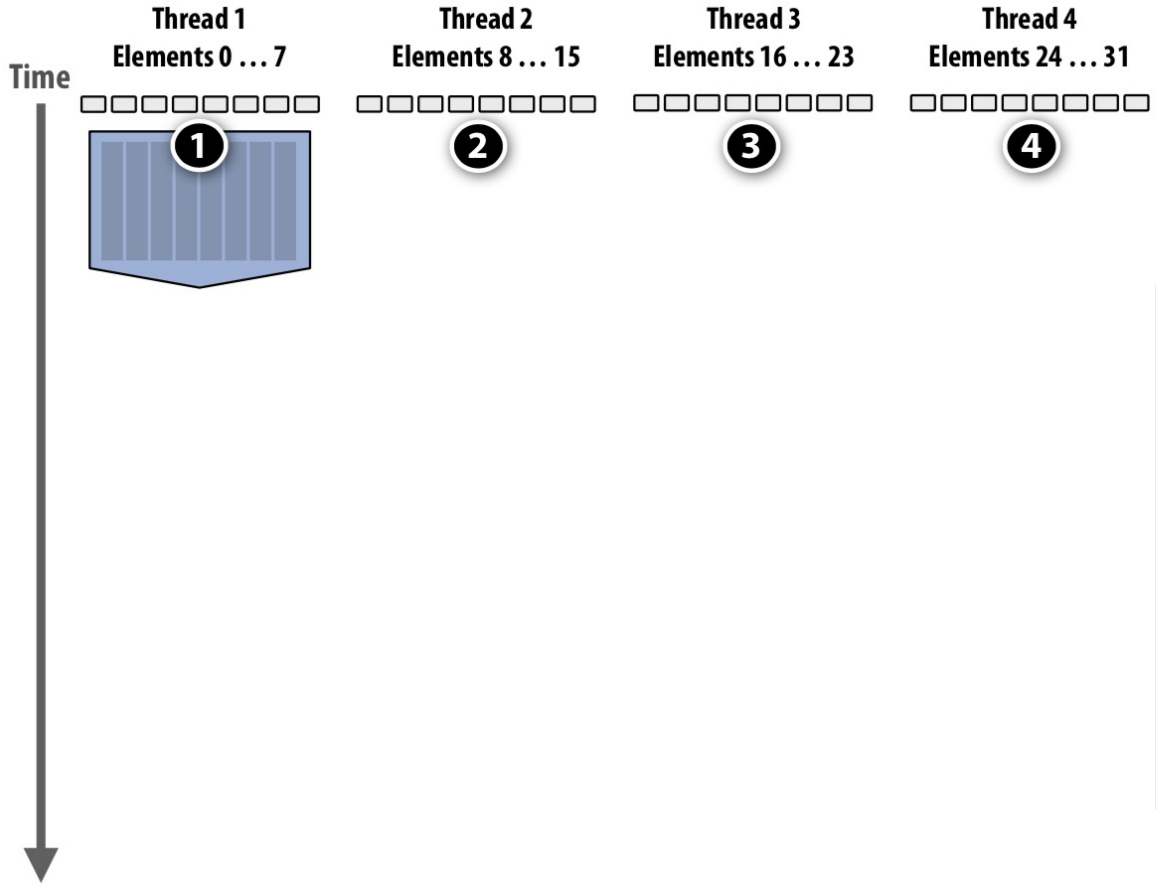


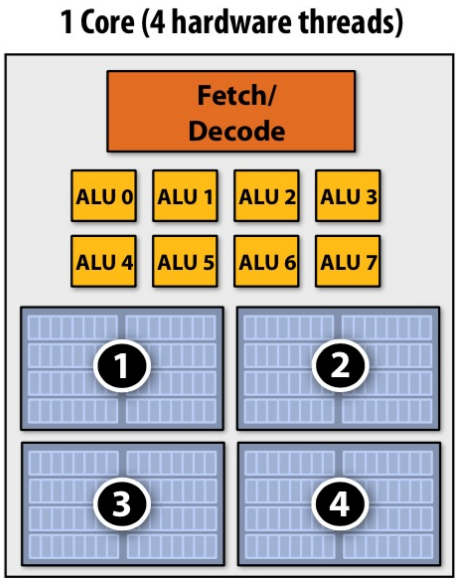


Slide credit : CMU 15-418/15-618

Hiding stalls with multi-threading

- **Idea:** interleave processing of multiple threads on the same core to hide stalls

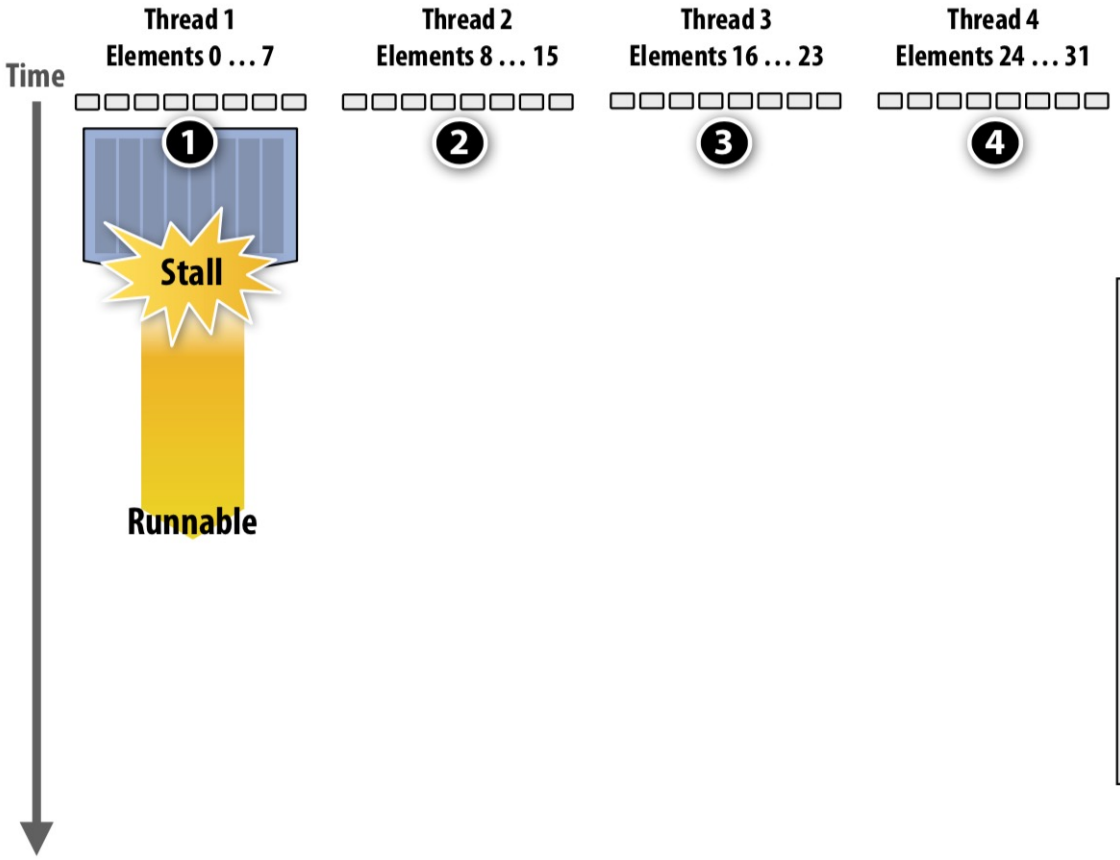


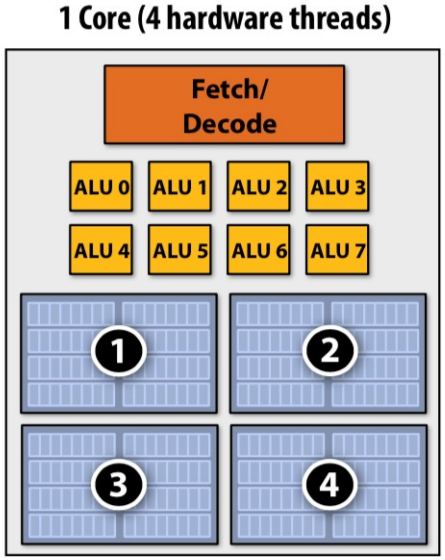


Slide credit : CMU 15-418/15-618

Hiding stalls with multi-threading

- **Idea:** interleave processing of multiple threads on the same core to hide stalls

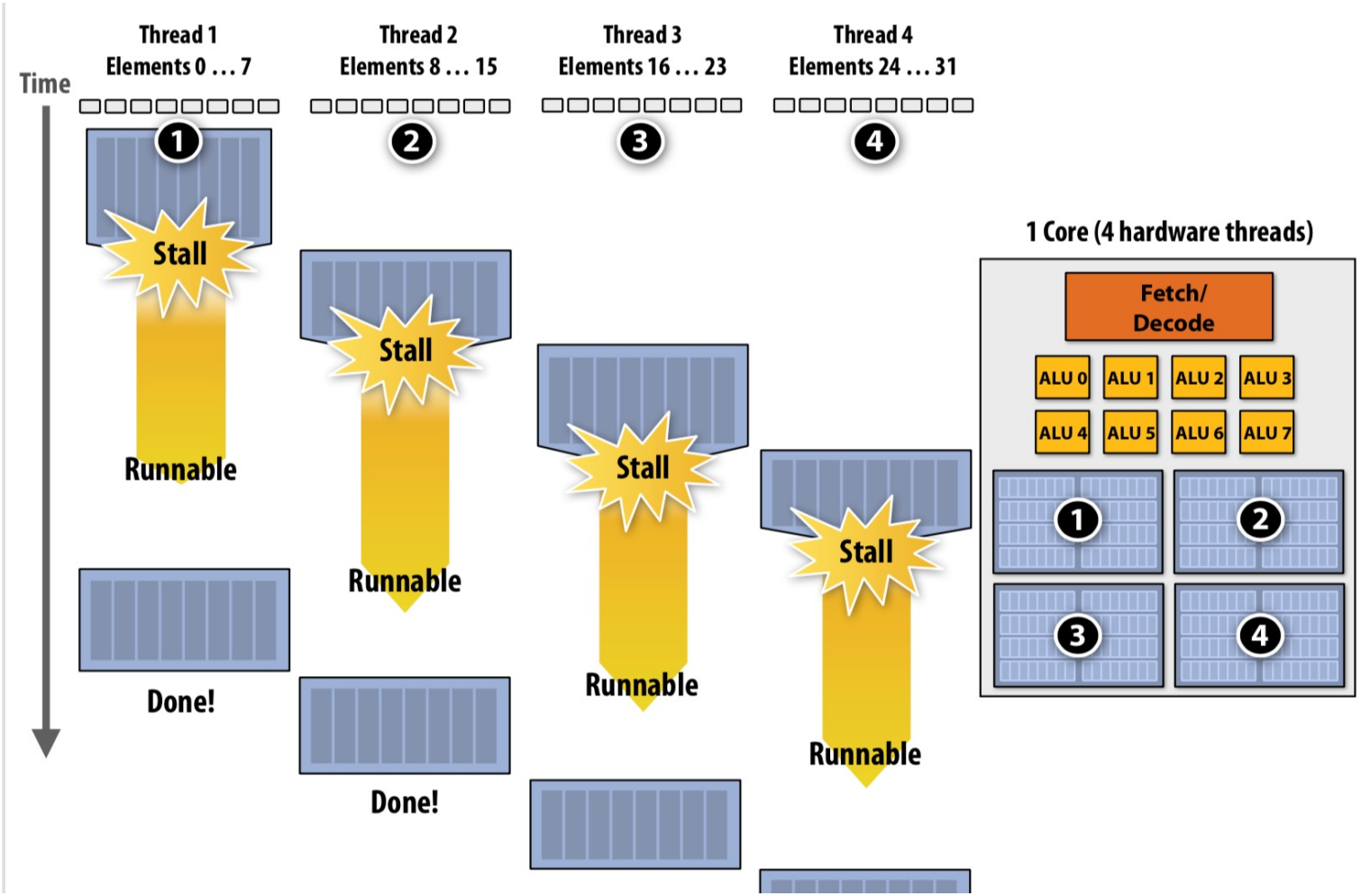




Slide credit : CMU 15-418/15-618

Hiding stalls with multi-threading

- **Idea:** interleave processing of multiple threads on the same core to hide stalls



Slide credit : CMU 15-418/15-618

Multi-threading summary

- **Benefits:** use a core's ALU resources more efficiently by hiding memory latency

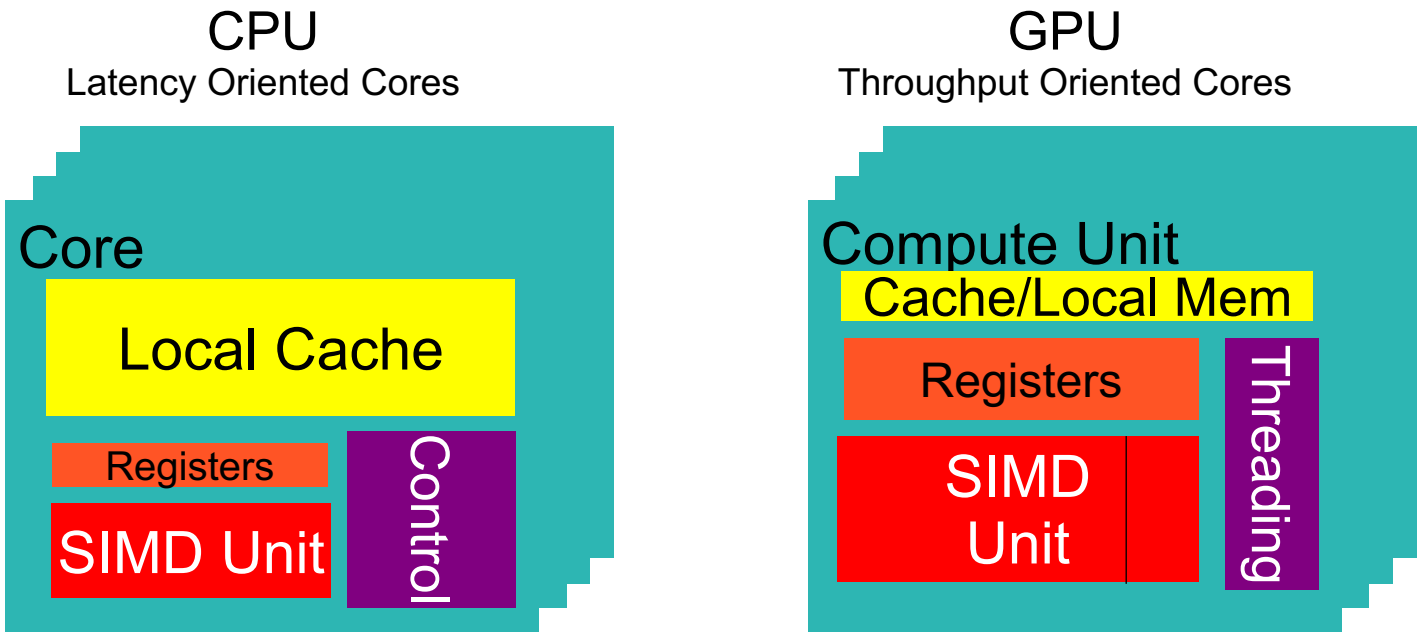
- **Costs**
  - Require additional storage for thread contexts
  - Relies heavily on memory bandwidth
    - More threads → Larger working set → less cache space per thread
    - May go to memory more often, but can hide the latency

Slide credit : CMU 15-418/15-618

GPU Architectures

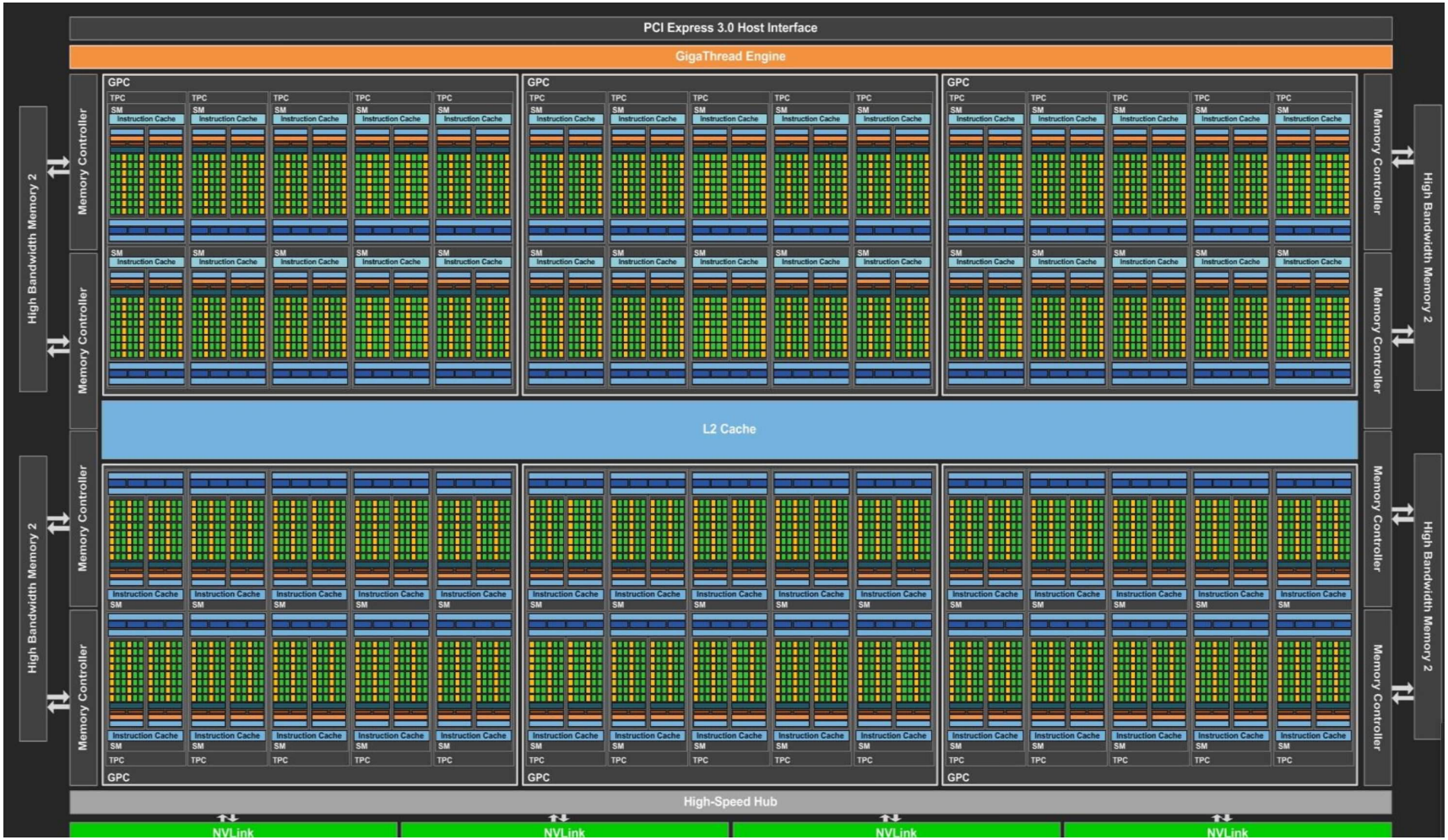
CPU and GPU are designed very differently

- CPU is designed to minimize the execution latency of a single thread
- GPU is designed to maximize the computation throughput
- GPU uses larger fraction of silicon for computation than CPU
- GPU consumes order of magnitude less energy per operation than CPU.
  - 2nJ/Operation at CPU, 200pJ/Operation at GPU

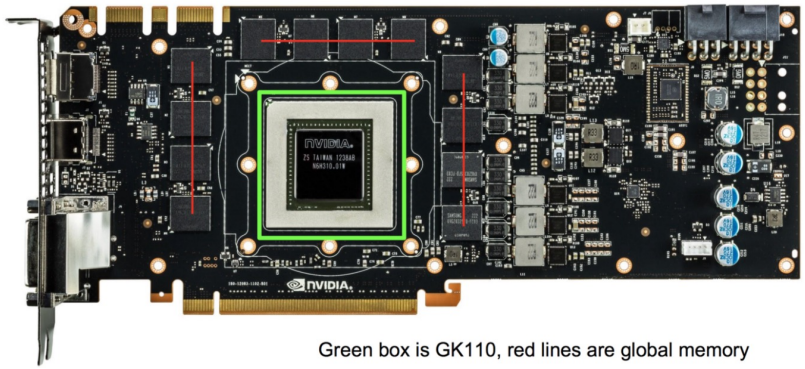


Modern GPU looks like

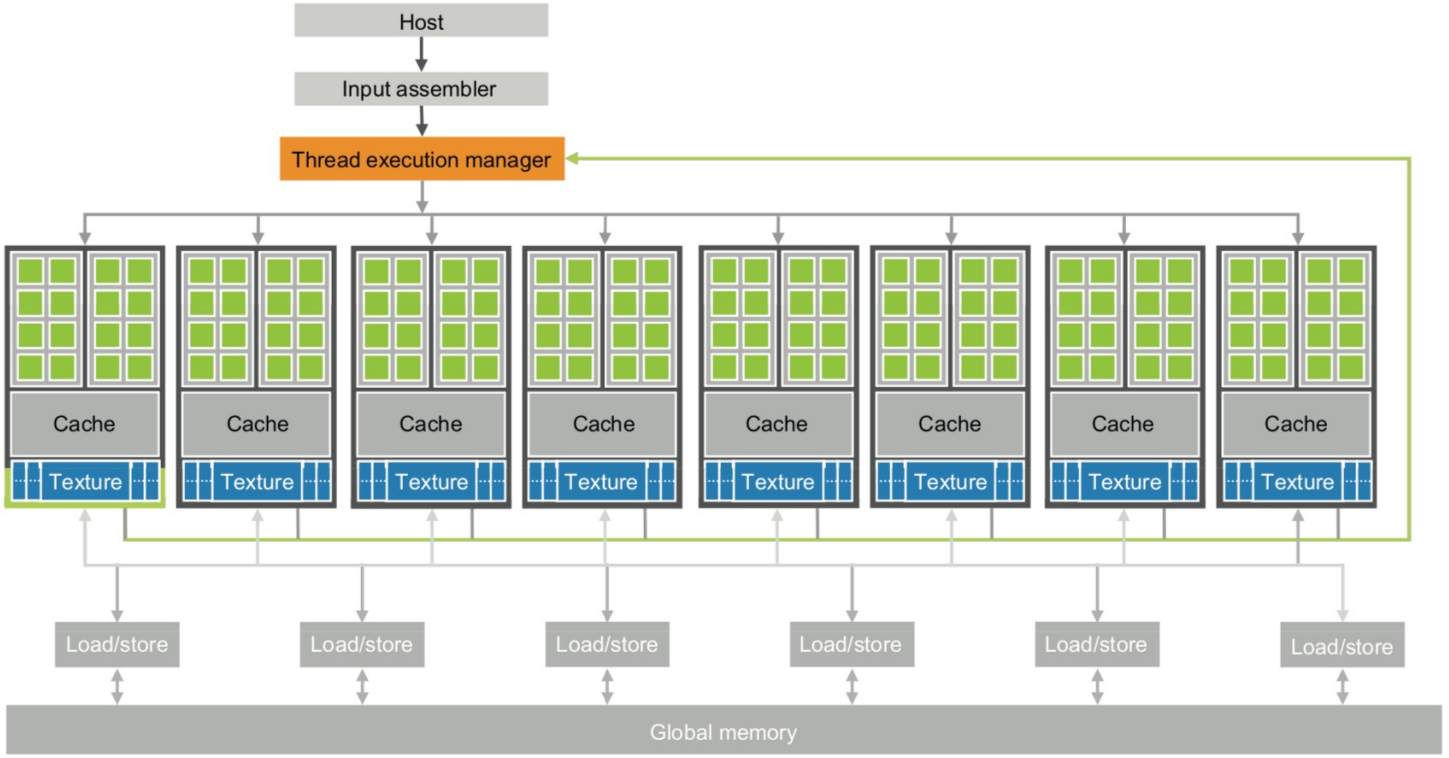
- NVIDIA Pascal



Inside a GPU

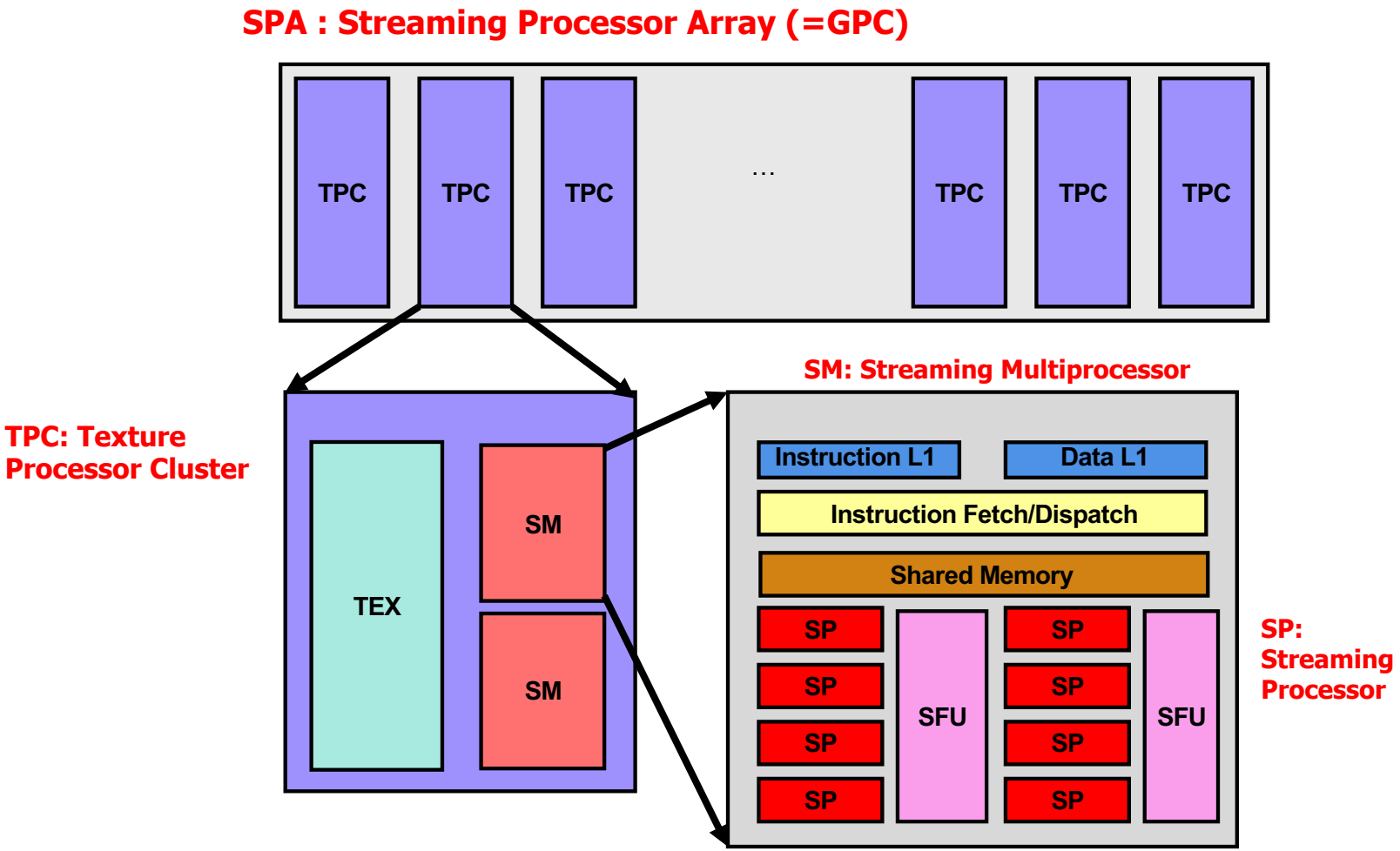


Green box is GK110, red lines are global memory



Inside a GPU

- Hierarchical Approach

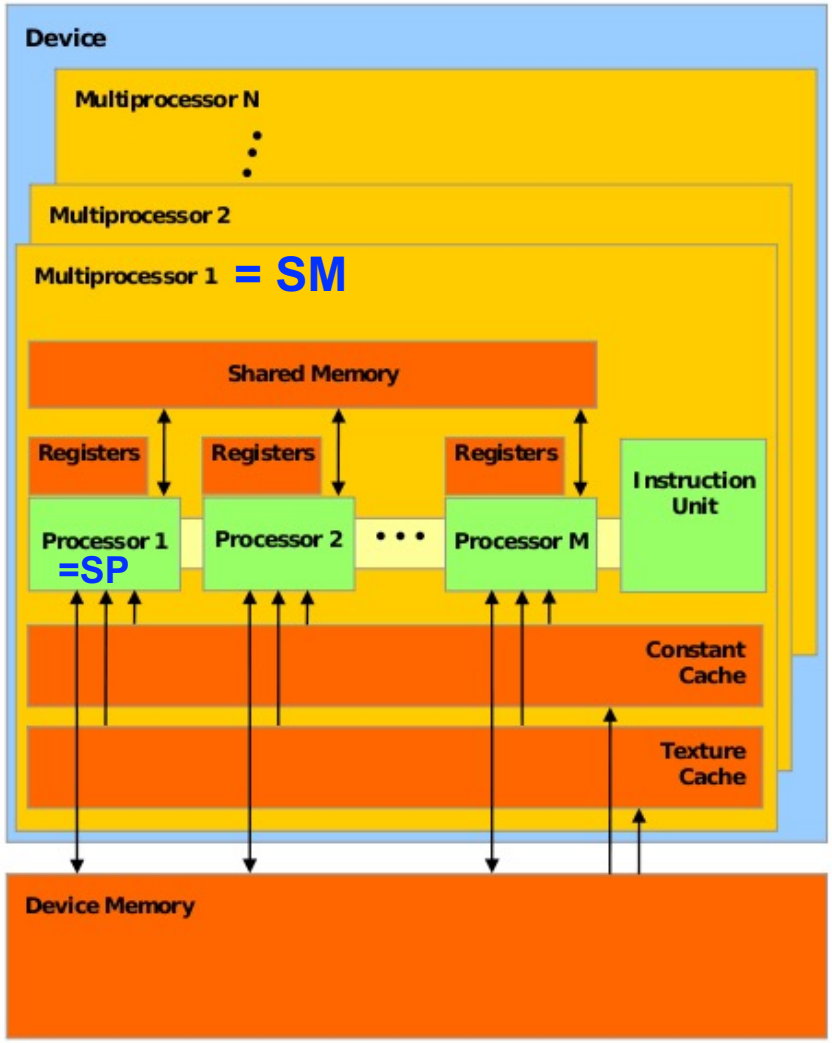


Inside a GPU

- **SPA :** Streaming Processor Array
- **TPC :** Texture Processor Cluster
  - Multiple SMs + TEX
  - TEX : texture processor for graphics purpose
- **SM :** Streaming Multiprocessor
  - Multiple Processors (SPs)
  - Multi-threaded processor core
  - Fundamental processing unit for thread block
- **SP (or CUDA core) :** Streaming Processor
  - ALU for a single thread
- **SFU:** special function unit
  - For complex math functions: sin, cos, square root, ...
  - Execute one special instruction per thread

Inside a GPU

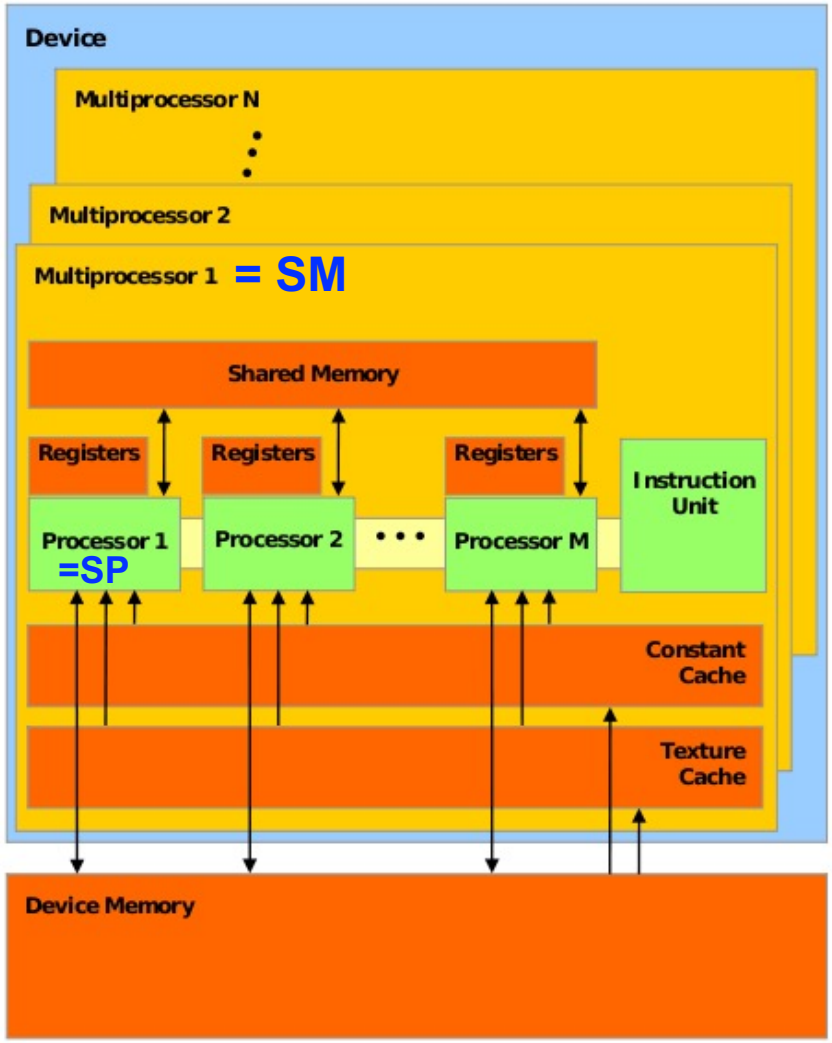
- GPUs have many **Streaming Multiprocessors (SMs)**
  - Each SM has multiple processors but only one instruction unit
    - All **SP** within a **SM** shares program counter
  - **Groups of processors must run the exact same set of instructions at any given time** within a single SM



- When a kernel (GPU code) is called, **the task is divided up into threads**
  - Each thread handles a small portion of the given task
  - All thread execute the same kernel code
- **The threads are divided into Blocks**

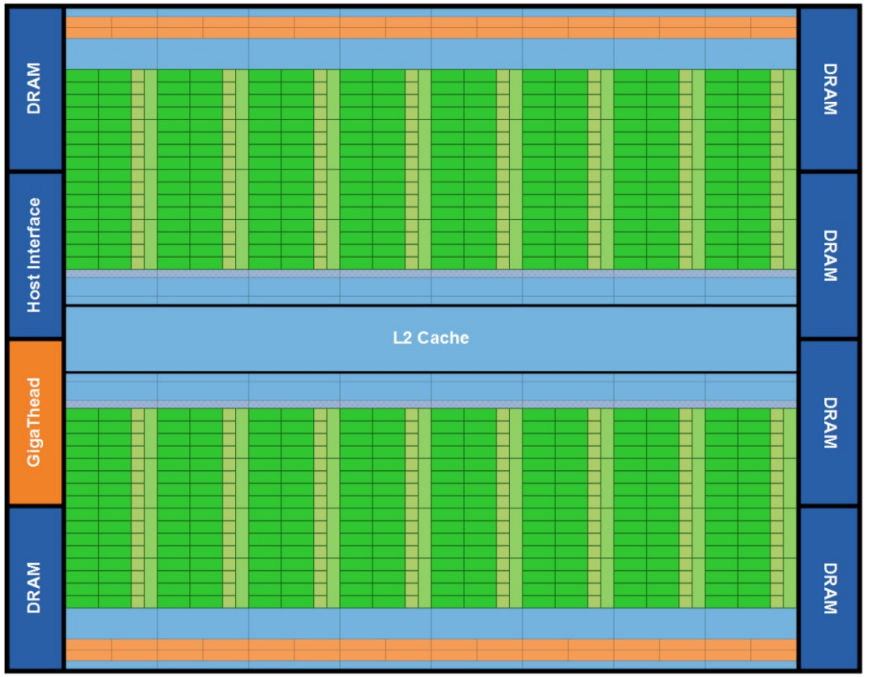
Inside a GPU

- **Each block is assigned to an SM**
- Inside the SM, the block is divided into **Warps** of threads
  - **Warps** consist of **32 threads**
  - All 32 threads MUST run the exact same set of instructions at the same time ➔ **SIMT**
    - Due to the fact that there is only one instruction unit
  - **Warps are run concurrently in an SM**



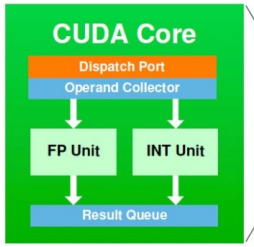
Example: NVIDIA Fermi Architecture

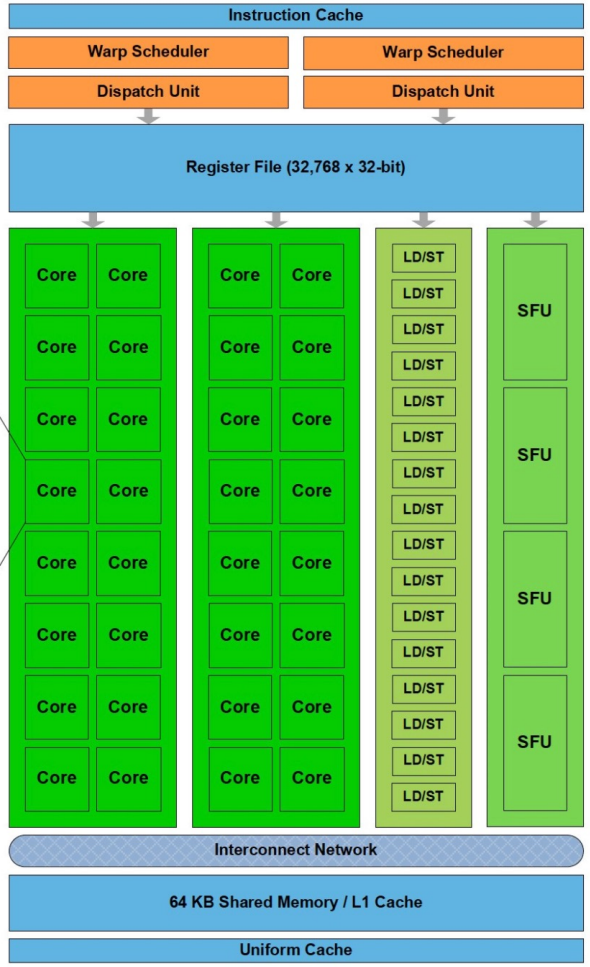
- **16 SMs**
- **Each with 32 cores**
  - 512 total cores
- **Each SM hosts up to**
  - 48 warps (=1,536 threads)
    - Warp : 32 threads
- **In flight, up to**
  - 24,576 threads



Example: NVIDIA Fermi Architecture

- **Streaming Multiprocessor (SM)**
  - 32 Streaming Processors (SP) = CUDA Core
  - 16 Load/store units
  - 4 Special Function Units (SFU)
  - 64KB high speed on-chip memory (L1+shared memory)
  - Interface to the L2 cache
  - 32K of 32-bit registers
  - Two warp schedulers, two dispatch units

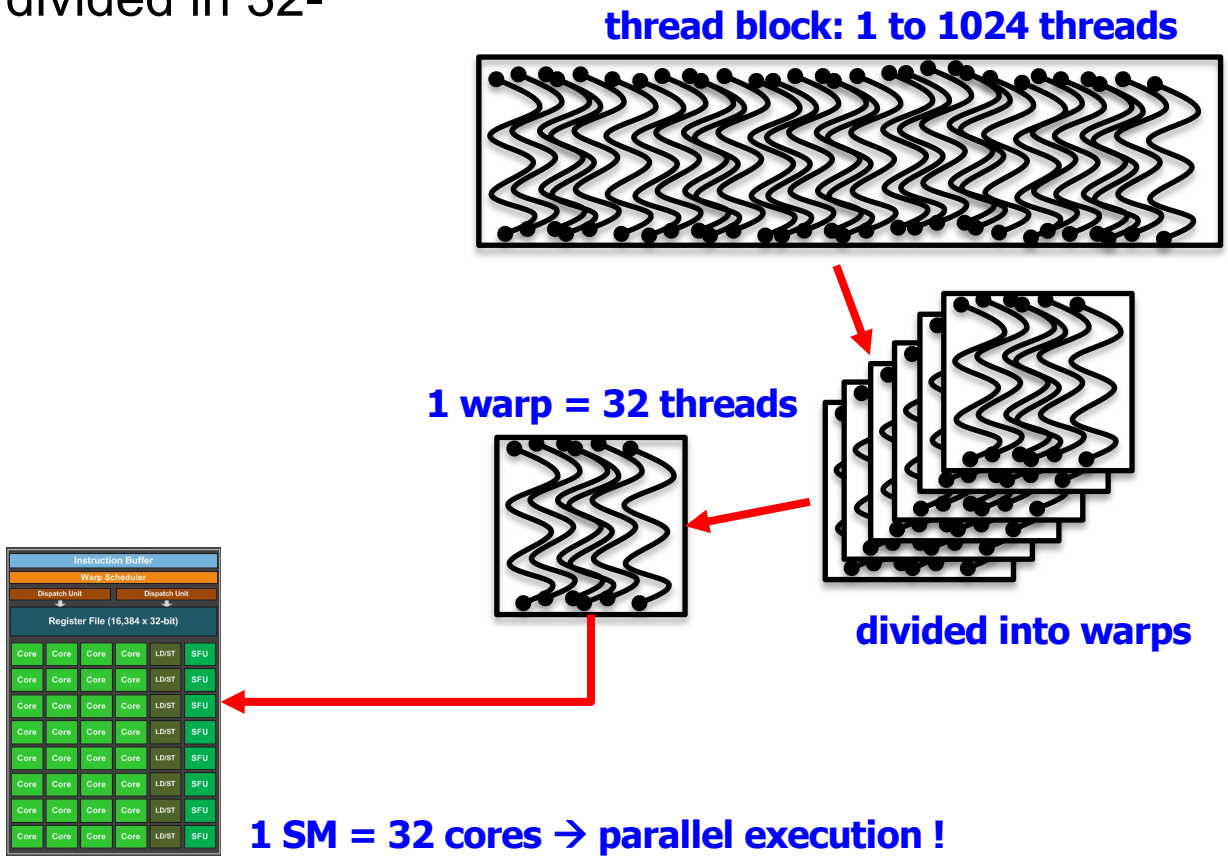




- **SP (CUDA core)**
  - execution unit for integer and floating-point numbers
  - 32-bit precision for all instructions

Thread Scheduling/Execution

- Threads run concurrently
  - SM assigns/maintains thread id #s
  - SM manages/schedules thread execution
- Each Thread Blocks is divided in 32-thread Warps



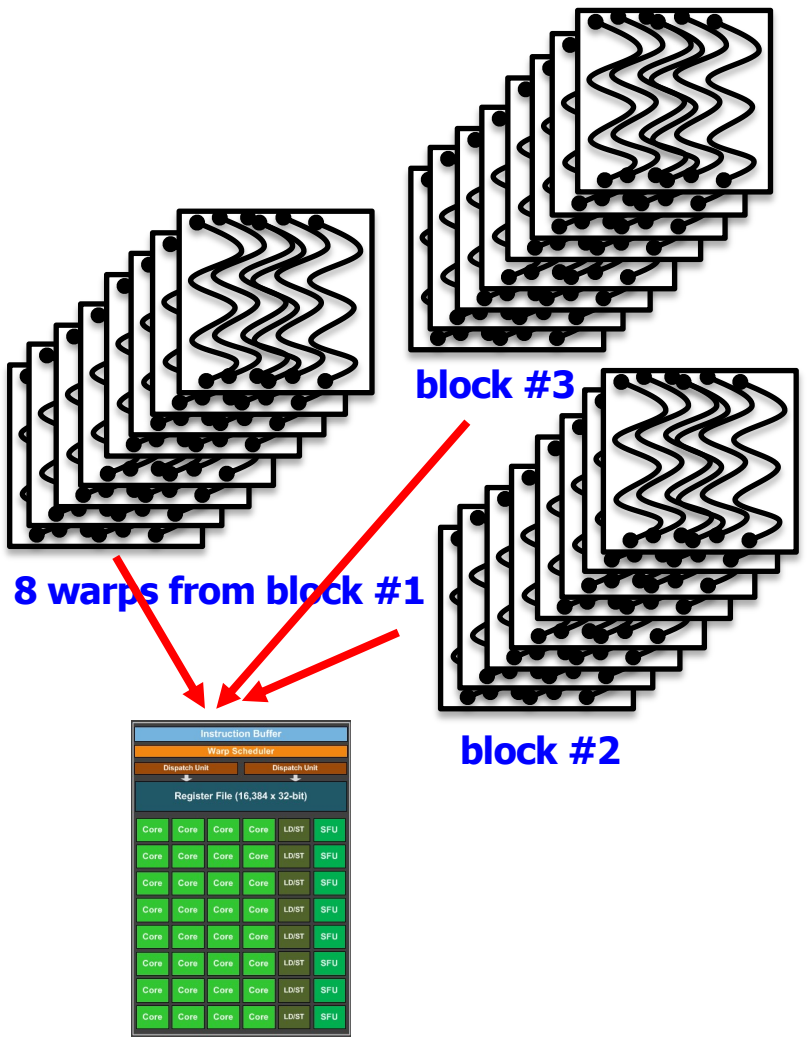
Slide credit : Prof. Baek

Thread Scheduling/Execution

- Warps are scheduling units in SM

- A scenario
  - 3 blocks to an SM
  - each block has 256 threads

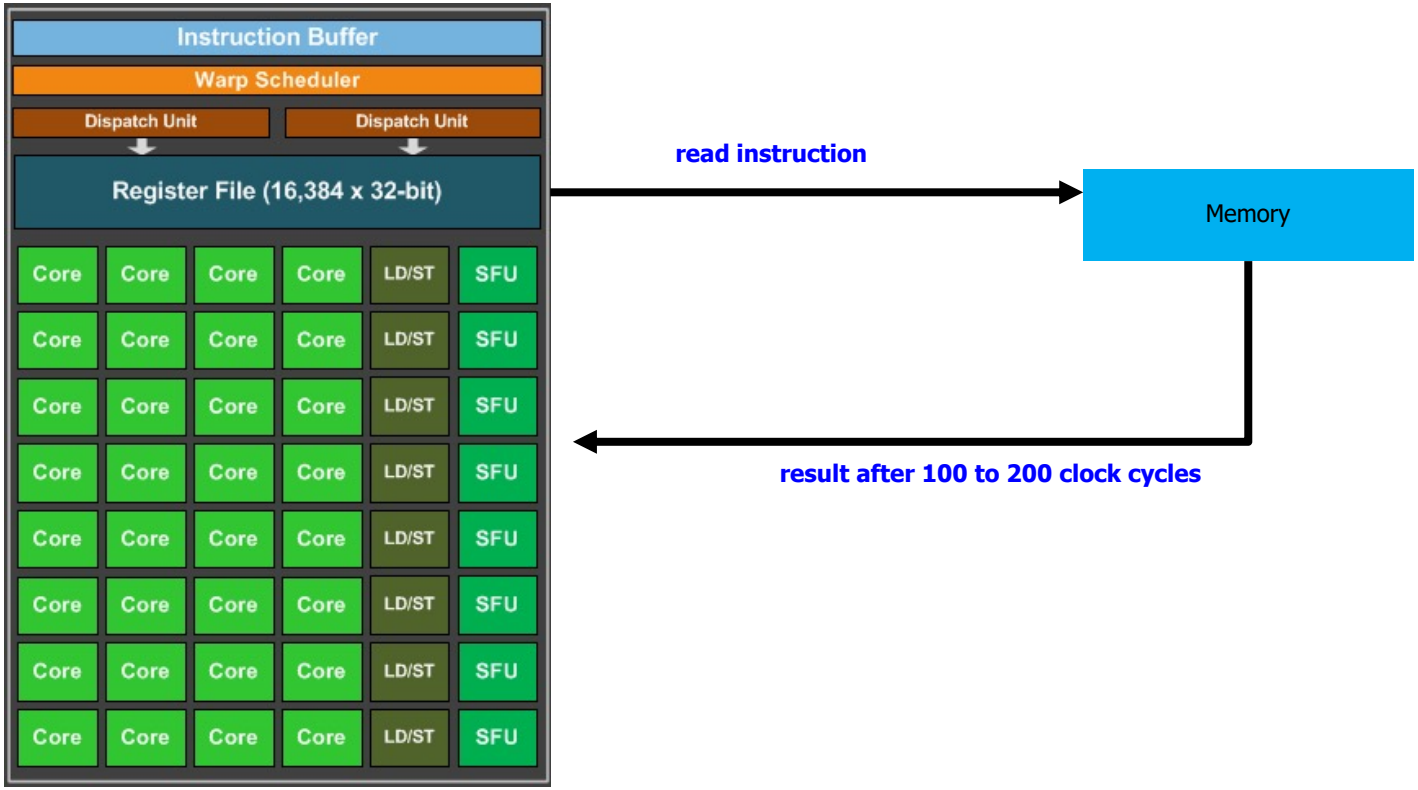
- how many warps?
  - each block has 256 / 32 = 8 warps
  - SM has 3 * 8 = 24 warps
  - At any point in time,
    only one of the 24 Warps will be selected for instruction fetch and execution.



Slide credit : Prof. Baek

SM Warp Scheduling

- All threads in a Warp execute the same instruction when selected
  - only one control logic for an SM
- **memory access** → latency problem → scheduling required !

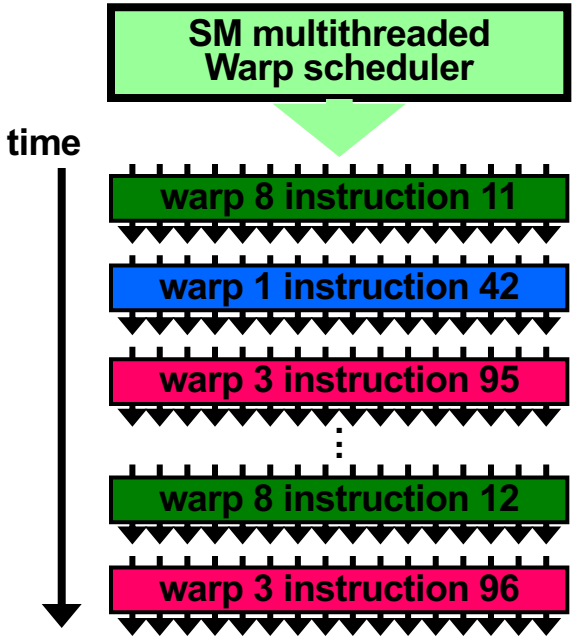


Slide credit : Prof. Baek

SM Warp Scheduling to Hide Memory Latency

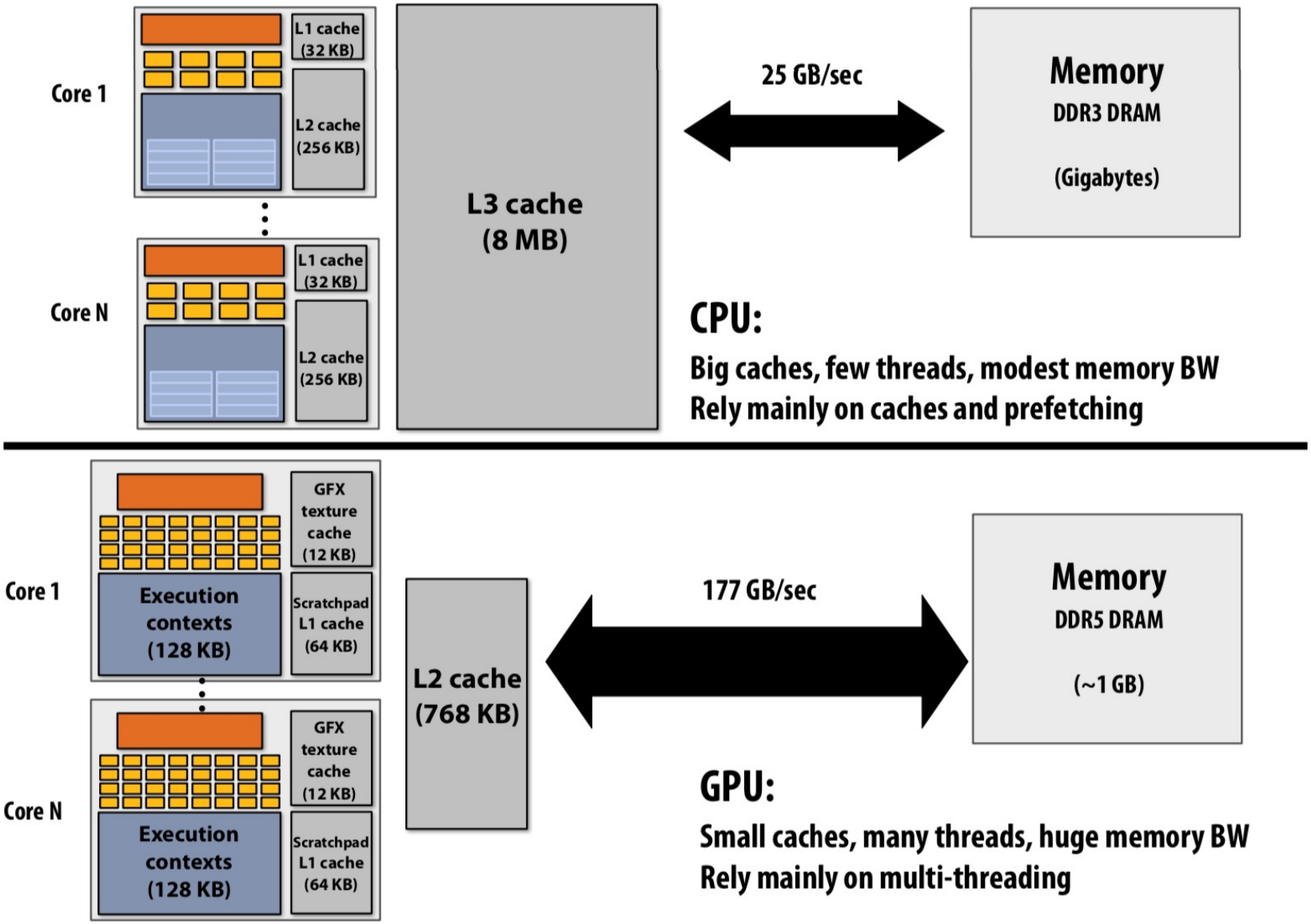
- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy

- **Example**
  - **Assumption**
    - **1 clock cycles** needed to dispatch the same instruction for all threads in a Warp
    - If **one global memory access** is needed **for every 4 instructions**
  - **A minimal of 26 Warps** are needed to **fully tolerate 100-cycle memory latency**



Slide credit : Prof. Baek

CPU vs GPU memory hierarchies



Slide credit : CMU 15-418/15-618

Next..

- Fundamentals of CUDA