Computer Architecture
计 算 机 体 系 结 构

Lecture 10. Data-Level Parallelism and GPGPU
第十讲、数据级并行化与GPGPU

Chao Li, PhD.
李超 博士

SJTU-SE346, Spring 2019

Review

- Thread, Multithreading, SMT
- CMP and multicore
- Benefits of multicore
- Multicore system architecture
- Heterogeneous multicore system
- Heterogeneous-ISA CMP
- Multicore and manycore
- Design challenges

Outlines

- DLP Overview
- Vector Processor
- GPGPU

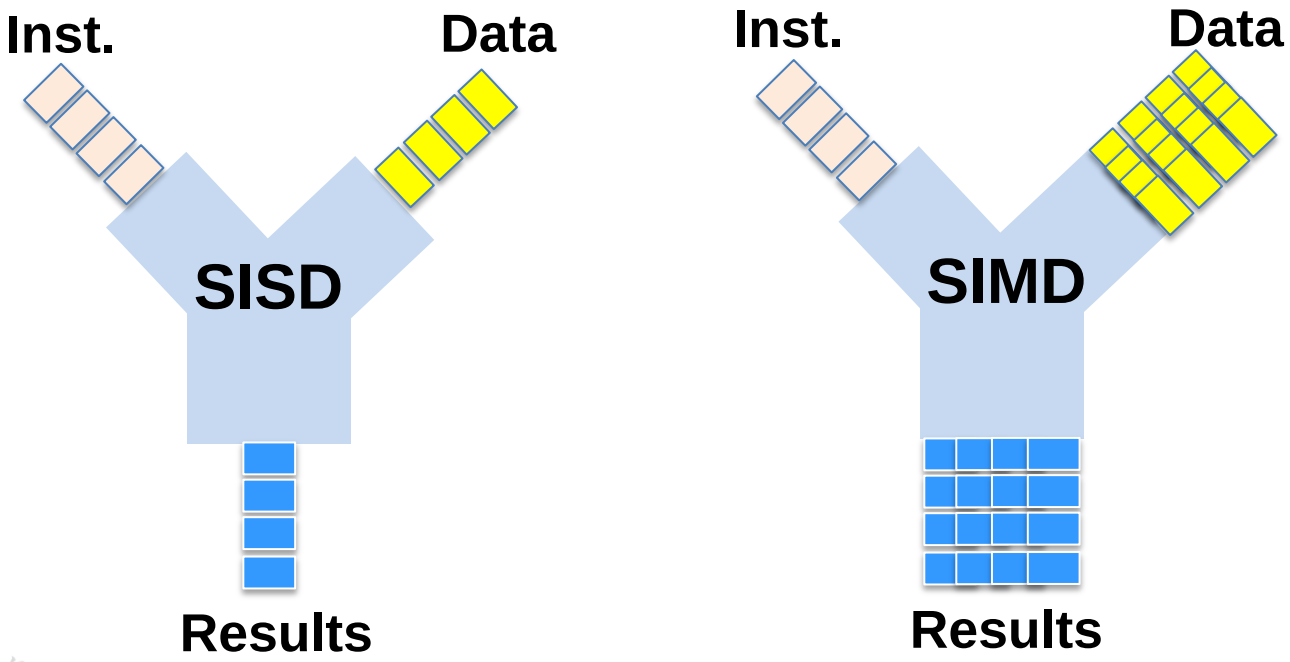
Throughput Workloads

- Interested in how many jobs can be completed
  - Finance, social media, gaming, etc.
- CPUs with multiple cores are considered suitable
- Recent years we observe a quick adoption of GPU

Data-Level Parallelism

- **DLP**: Parallelism comes from simultaneous operations across large data sets, rather than from multiple threads

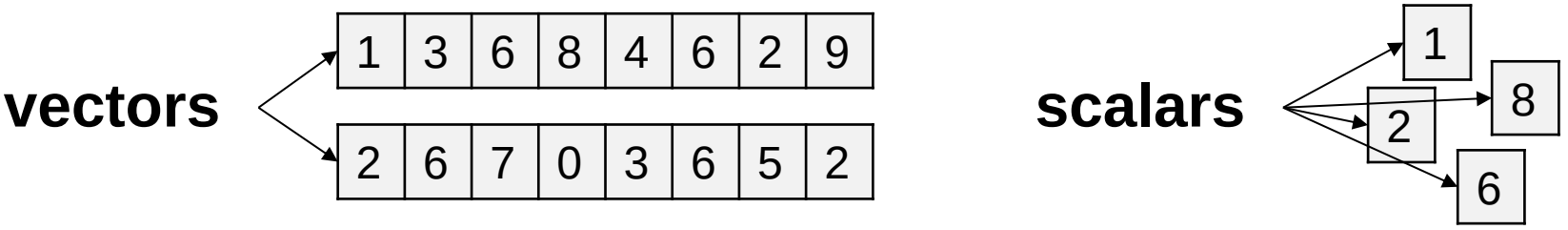
- Particularly useful for large scientific/engineering tasks



Outlines

- DLP Overview
- **Vector Processor**
- GPGPU

Vector Operation



A vector is just a linear array of numbers, or scalars

- Can we extend processors with **vector "data type"** ?
- The idea of vector operations:
  - Read a set of data elements
  - Operate on the element array

Vector Instructions

A vector instruction operates on a sequence of data items

A few key advantages:

- Reduced instruction fetch bandwidth
  - A single vector instruction specifies a great deal of work
- Less data hazard checking hardware
  - Independent operation between elements in the same vector
- Memory access latency is amortized
  - Vector instructions have a known memory access pattern

Vector Architecture

Two primary styles of vector architecture:

CDC STAR-100

- memory-memory vector processors
  - All vector operations are memory to memory
  - CDC STAR-100 (faster than the CDC 7600)

- vector-register processors
  - Vector equivalent of load-store architectures
  - Cray, Convex, Fujitsu, Hitachi, NEC in 1980s

Key Components

- A vector processor typically consists of vector units and the ordinary pipelined scalar units

- **Vector register**:
  - fixed-size bank holding a single vector
  - typically holding 64~128 FP elements
  - determines the maximum vector length (MVL)

- **Vector register file**:
  - 8~32 vector registers

Early Design: Cray-1

- Released in 1975
- 115 KW, 5.5 Tons
- 160 MFLOPS

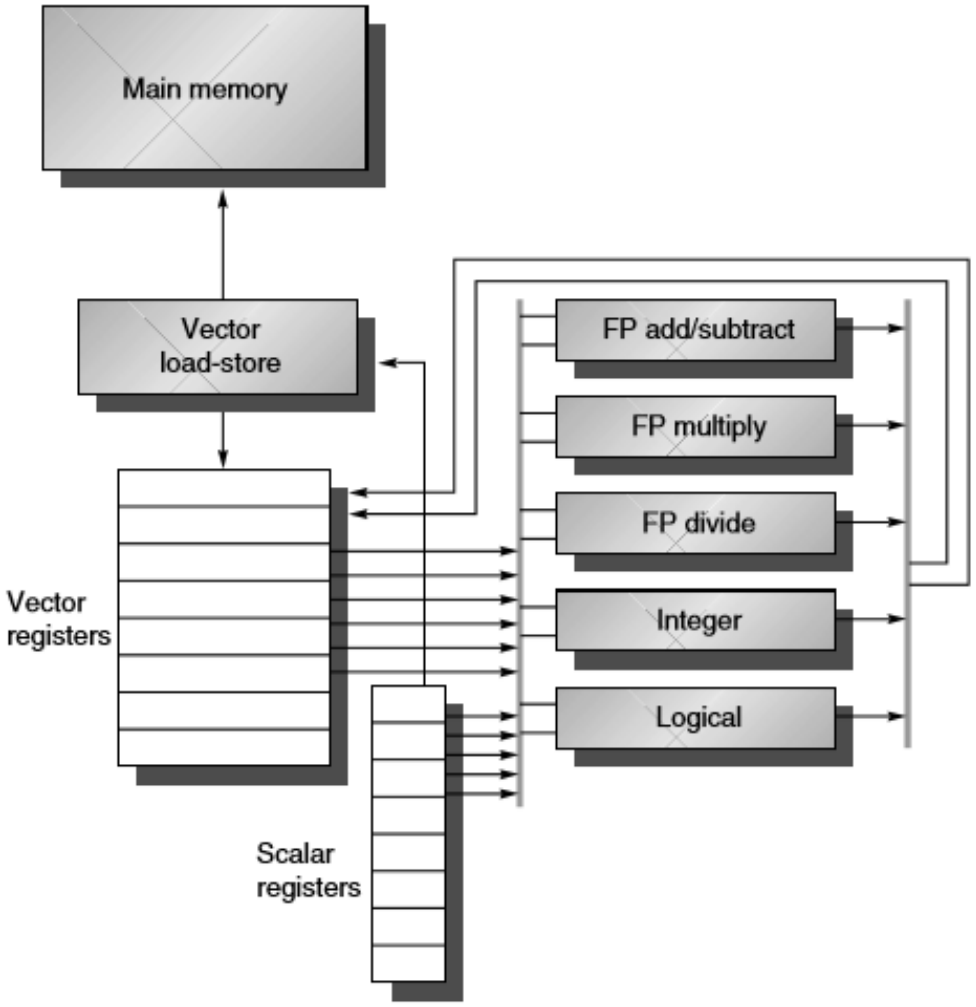
- Eight 64-entry vector registers
  - 64-bit register
  - 4096 bits in each register
  - 4KB for vector RF

Cray-1

- Special vector instructions:
  - Vector + Vector, Vector + Scalar
  - Each instruction specifies 64 operations

Case Study: VMIPS

- Vector registers
  - 64-element register
  - Register file has 16 read ports and 8 write ports
- Vector functional units
  - 5 fully pipelined FUs
  - Hazards detection
- Vector load-store unit
  - Loads/stores a vector
  - 1 word per clock cycle



Case Study: VMIPS

- **Add two vectors**

  ADDVV.D V1, V2, V3 // add elements of V2 and V3, then put each result in V1

- **Add vector to scalar**

  ADDVS.D V1, V2, F0 // add F0 to each elements of V2 , then put each result in V1

- **Load vector**

  LV V1, R1 // Load vector register V1 from memory starting at address R1

- **Store vector**

  SV V1, R1 // Store vector register V1 from memory starting at address R1

Vectorized Code

Consider a **DAXPY** loop: “Double-precision a × X plus Y” (**Y= a × X + Y**). Assume the starting addresses of X and Y are in Rx ad Ry, respectively

MIPS code for DAXPY：

```
        L.D       F0,a          ;load scalar a
        DADDIU    R4,Rx,#512    ;last address to load
Loop:   L.D       F2,0(Rx)      ;load X[i]
        MUL.D     F2,F2,F0      ;a × X[i]
        L.D       F4,0(Ry)      ;load Y[i]
        ADD.D     F4,F4,F2      ;a × X[i] + Y[i]
        S.D       F4,9(Ry)      ;store into Y[i]
        DADDIU    Rx,Rx,#8      ;increment index to X
        DADDIU    Ry,Ry,#8      ;increment index to Y
        DSUBU     R20,R4,Rx     ;compute bound
        BNEZ      R20,Loop      ;check if done
```

~ 600 instructions;

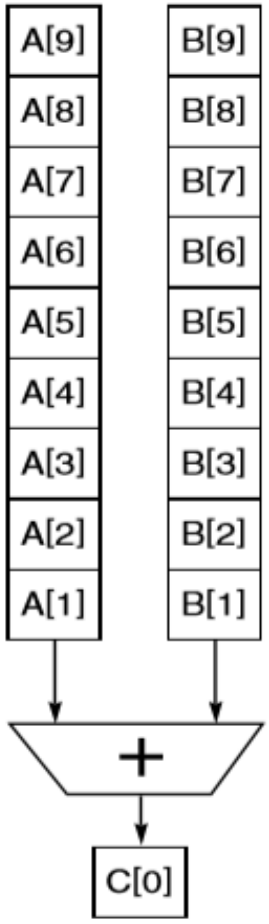
VMIPS code for DAXPY：

```
        L.D       F0,a          ;load scalar a
        LV        V1,Rx         ;load vector X
        MULVS.D   V2,V1,F0      ;vector-scalar multiply
        LV        V3,Ry         ;load vector Y
        ADDVV.D   V4,V2,V3      ;add
        SV        V4,Ry         ;store the result
```

6 instructions;

(the vector operation work on 64 elements)

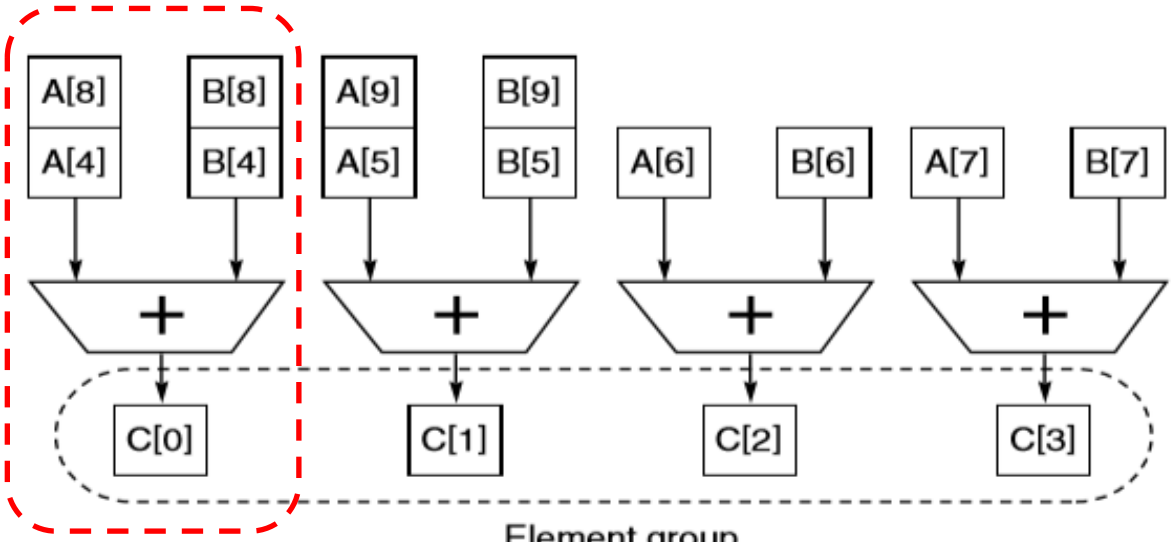
Multiple Lanes



(a)

ADDV C, A, B

- Element n of vector register A is "hardwired" to element n of vector register B
- Using multiple parallel pipelines (or lanes) to produce more than one results per cycle



lanes

(b)

Vector Instruction Execution

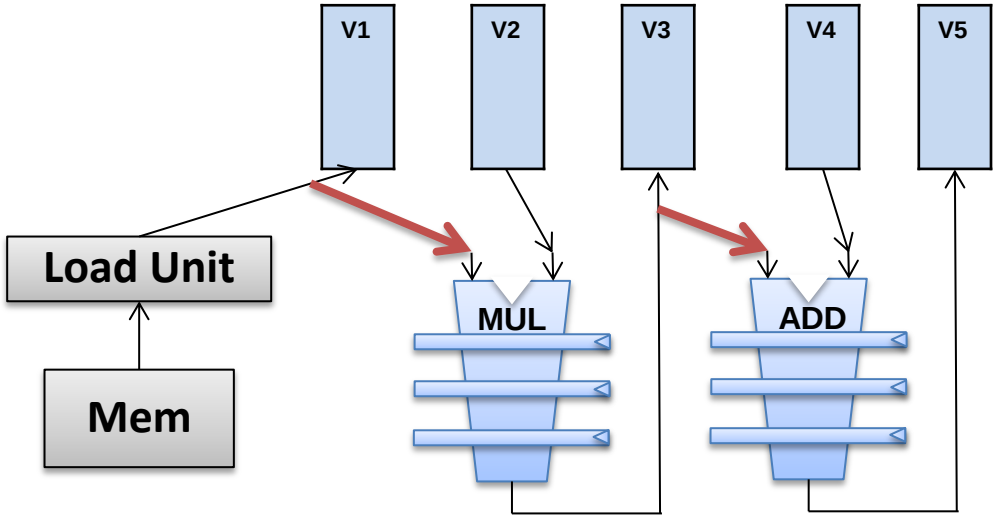
- Vector execution time mainly depends on:
  - The length of the operand vectors
  - Structural hazards
  - Data dependences
- Start-up time: pipeline latency (depth of the FU pipeline)
- VMIPS's FU consumes one element per cycle
  - Execution time is about the vector length
- Pipelined multi-lane vector execution

$T_{vector} = T_{scalar} + (\text{vector length} / \text{lane number}) - 1$

Vector Chaining

- **Chaining** is the vector version of register bypassing
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available

```
LV      v1
MULV    v3, v1, v2
ADDV    v5, v3, v4
```



- **Convoy**: Vector instructions that may execute together
  - No structural hazards in a convoy
  - No data hazards (or managed via chaining) in a convoy

Vector Length Register

What to do when vector length is not exactly 64?

vector size depends on *n* :

```
For ( i=0; i<n; i++)
    Y[i] = a * X[i] + Y[i]
```

- Vector length register (VLR)
  - Controls the length of any vector operation
  - The value is no greater than the maximum vector length (MVL)

- Use strip mining for vectors over maximum length
  - The first loop do short segment (n mod MVL);
  - All the subsequent segments use VL = MVL

Vector Mask Register

The IF statement cannot normally be vectorized

IF statement in the loop：

```
For ( i=0; i<64; i++)
    if (X[i] != 0)
        X[i] = X[i] + Y[i]
```

- Vector–mask control
  - Provide conditional execution of each element
  - Based on a Boolean vector (vector mask register)
  - Instruction operates only on vector elements whose corresponding entries in the mask register are 1

Outlines

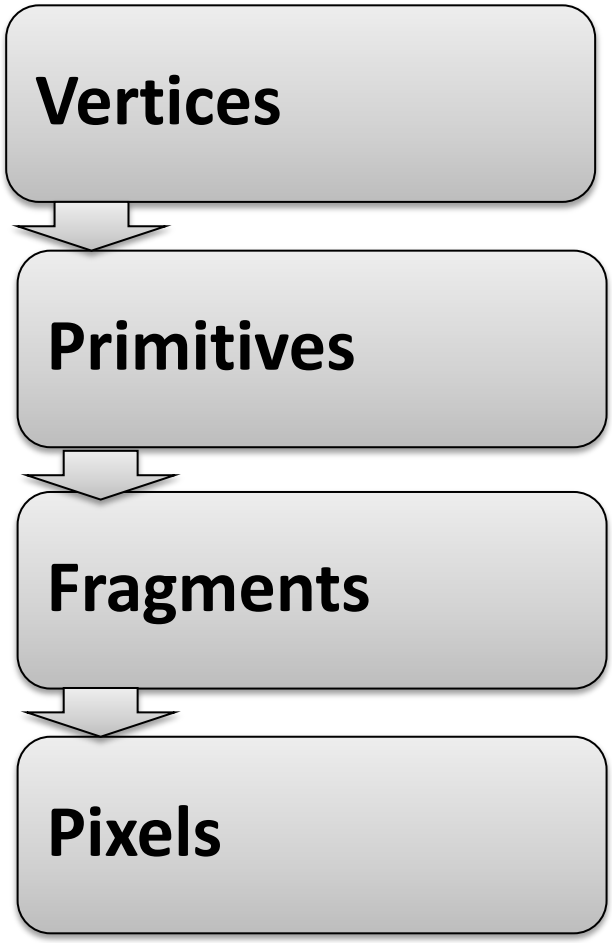
- DLP Overview
- Vector Processor
- **GPGPU**
  - **Graph Workloads**
  - **GPU Memory**
  - **Scheduling**

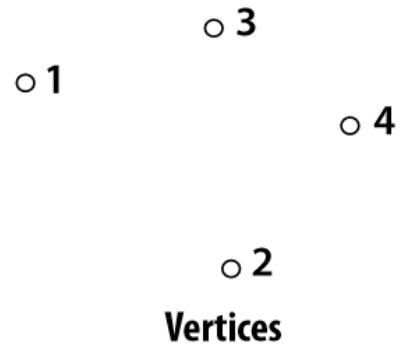
General Purpose GPU

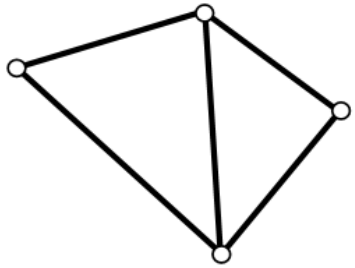
- Graphics processing unit (GPU) emerged in 1999
- GPU is basically a massively parallel architecture
- GPGPU: general-purpose computation on GPU
  - Leverage available GPU execution units for acceleration
  - Became practical and popular after about 2001
  - CPU codes runs on the host, GPU code runs on the device
- GPGPU exposes the massively multi-threaded hardware via a high level programming interface
  - NVIDIA's CUDA and AMD's CTM

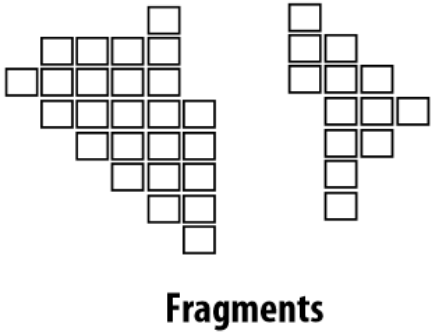
Graphics Pipeline

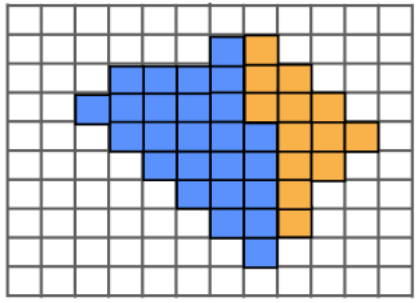
Modern GPUs structure their graphics computation in a similar organization called the graphics pipeline





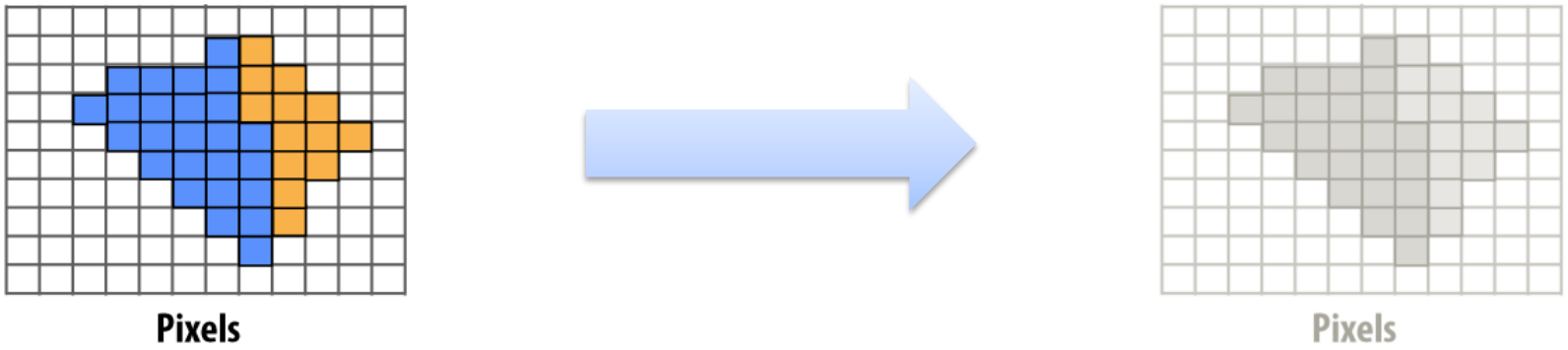






Kayvon Fatahalian, CMU 2011

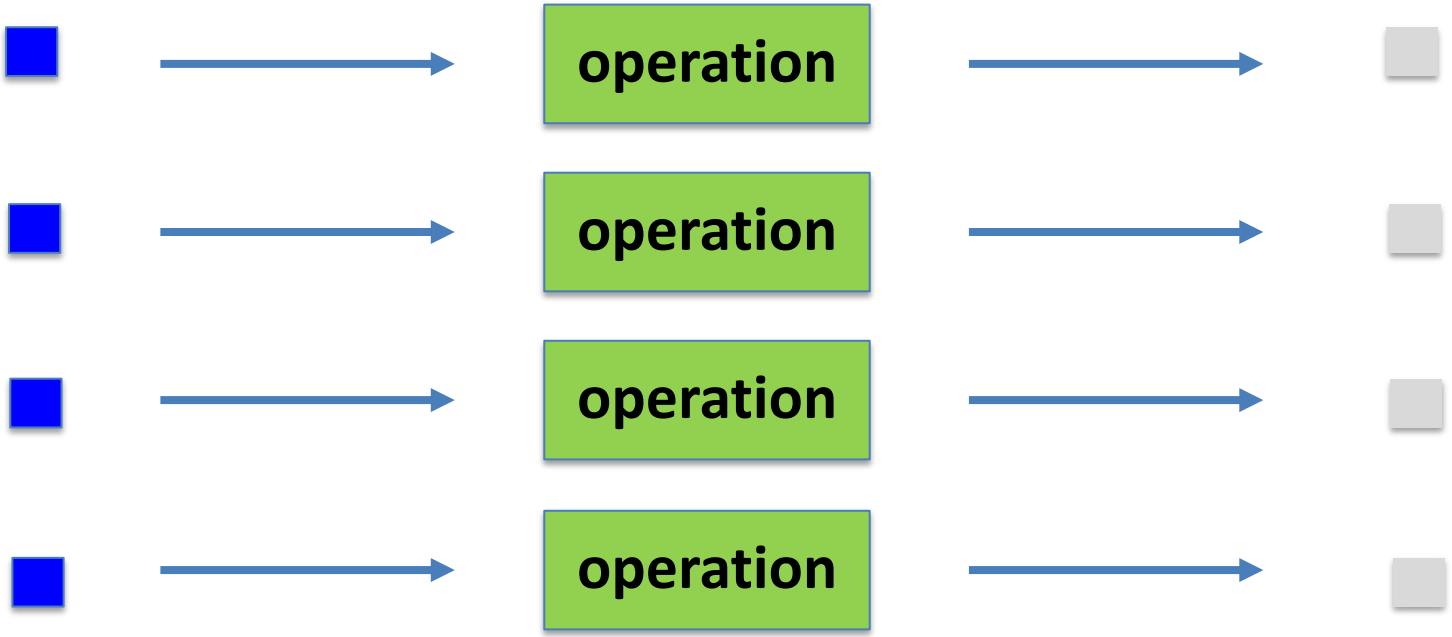
Graphics Workloads



Identical, independent, streaming computation on pixels

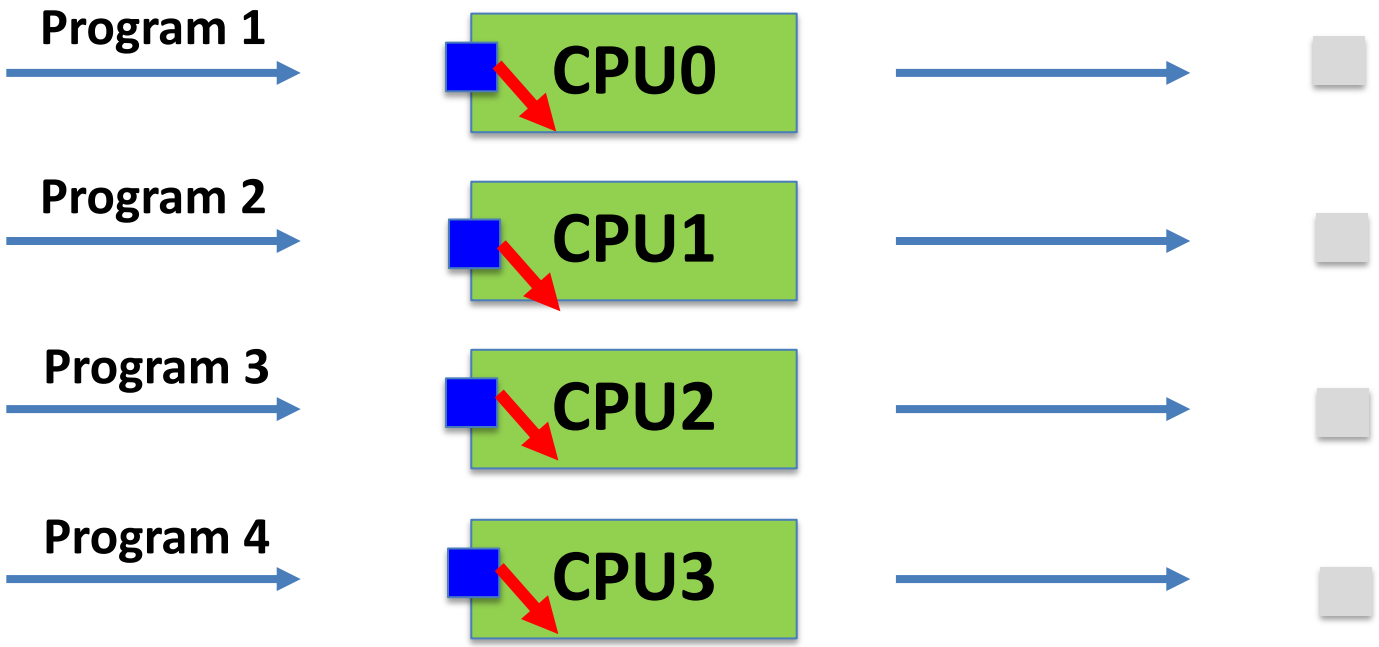
Data Parallel Workloads

Identical, independent computation on multiple data inputs



Processing Data Parallel Workloads

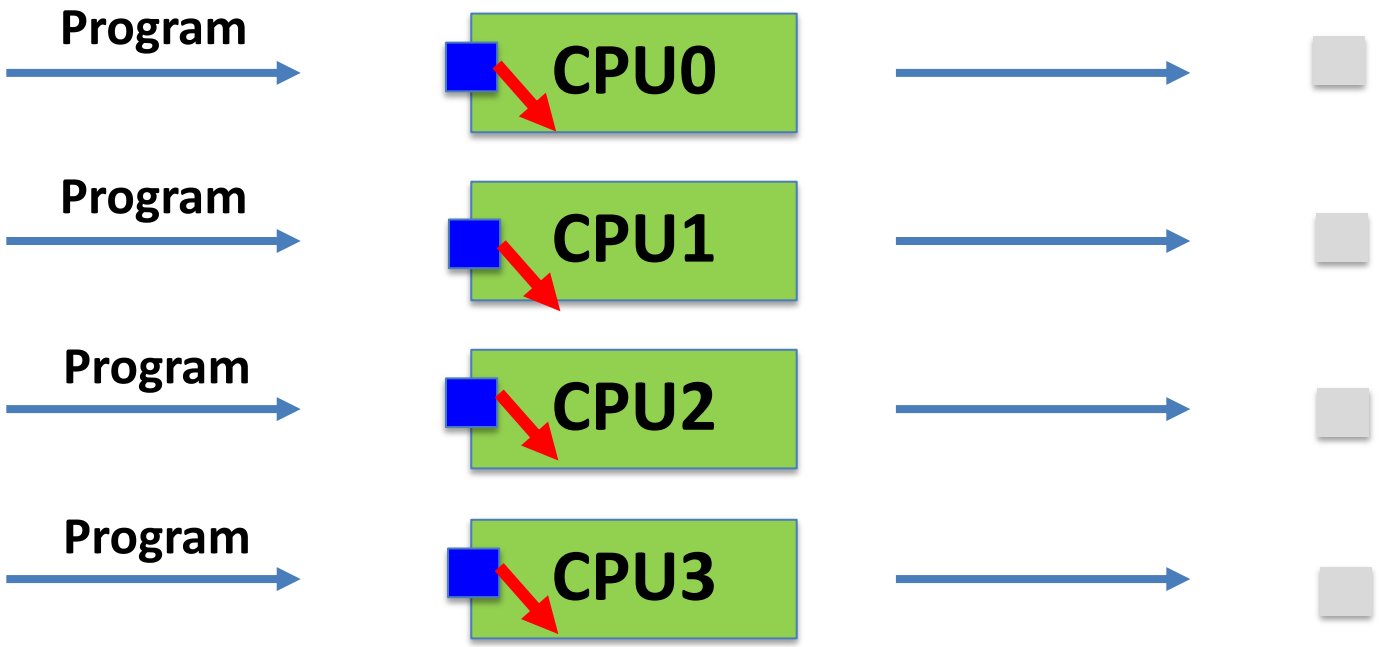
Multiprocessor Approach (MIMD)



Split independent tasks over multiple CPUs

Processing Data Parallel Workloads

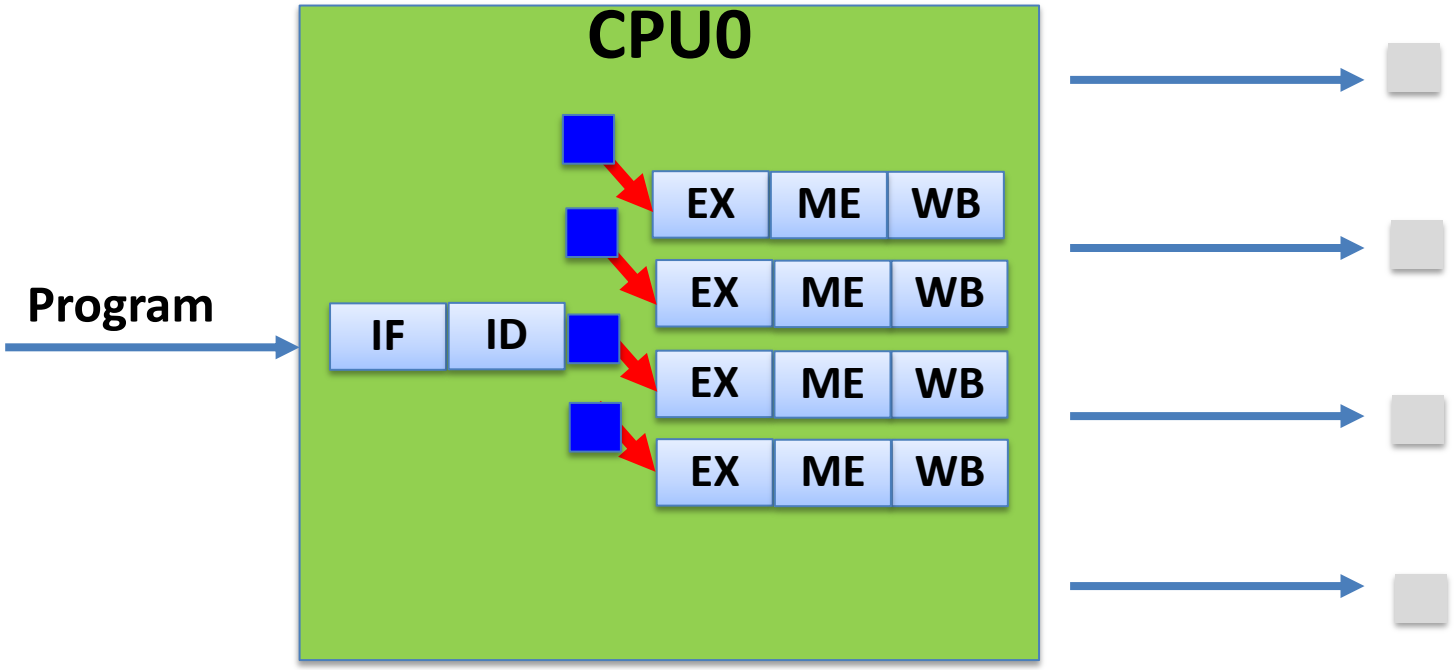
SPMD Approach (Single Program Multiple Data)



Split identical, independent task over multiple CPUs

Processing Data Parallel Workloads

SIMD/Vector Approach – eliminate redundant units

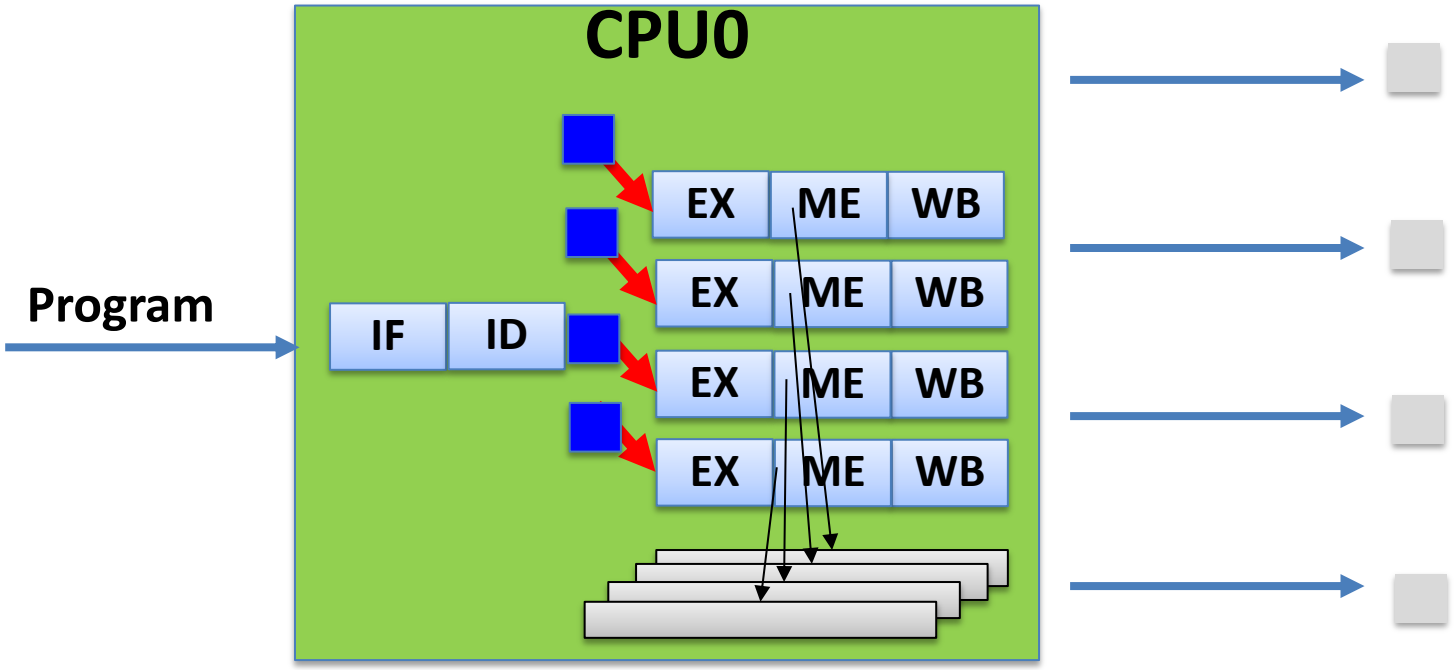


Split identical, independent task over multiple execution units

Processing Data Parallel Workloads

SIMD/Vector Approach: 1 heavy thread + data parallel ops

(single PC, single register file)

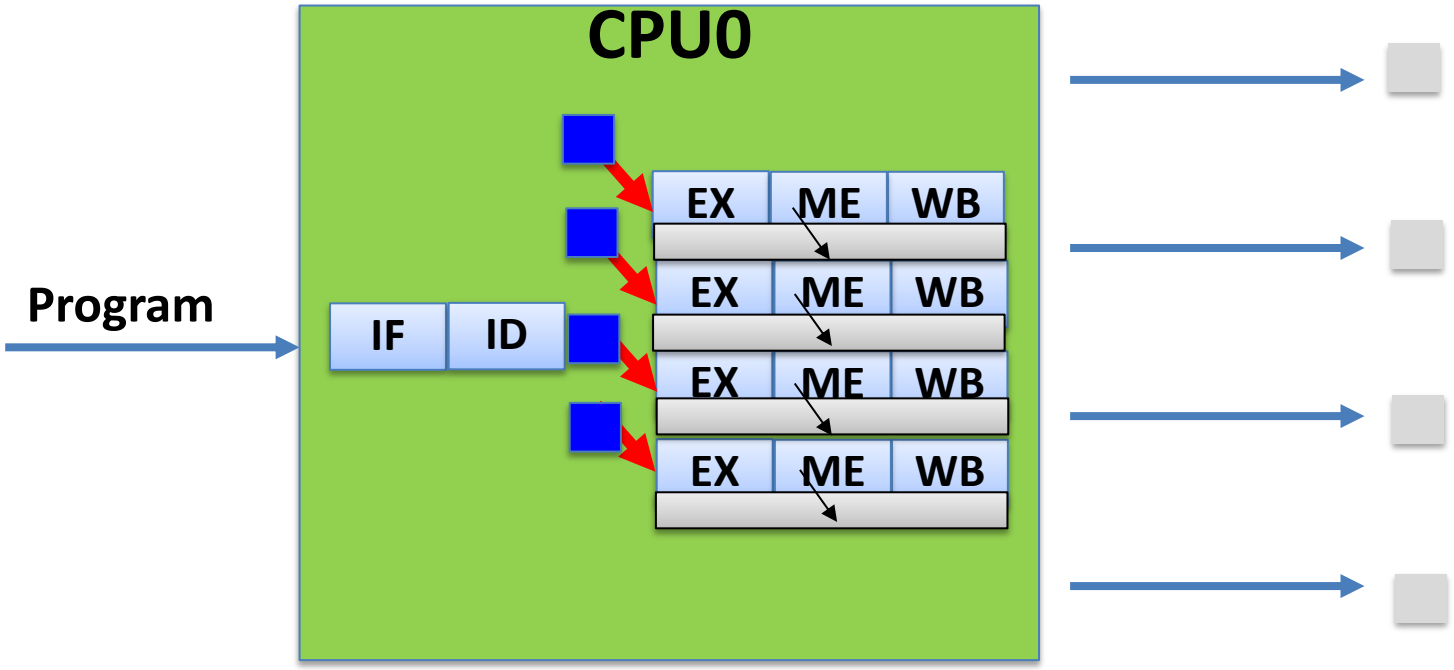


Split identical, independent task over multiple execution units

Processing Data Parallel Workloads

SIMT Approach:

multiple threads + scalar ops (single PC, multiple register file)



Split identical, independent task over multiple lockstep threads

Execution Model

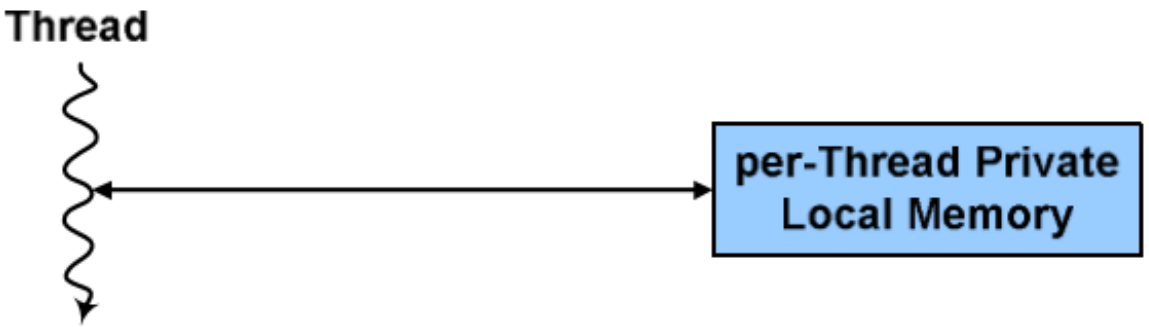
- Programmer provides a serial task ("shader program") that defines pipeline logic for certain stages

  - Vertex processors
    - Running **vertex shader programs**
    - Operates on point, line, and triangle primitives
  - Fragment processors
    - Running **pixel shader programs**
    - Operates on rasterizer output to fill up the primitives

- Program GPGPUs with CUDA or OpenCL
  - A minimal extension of C/C++
  - Executes shader programs across many data elements

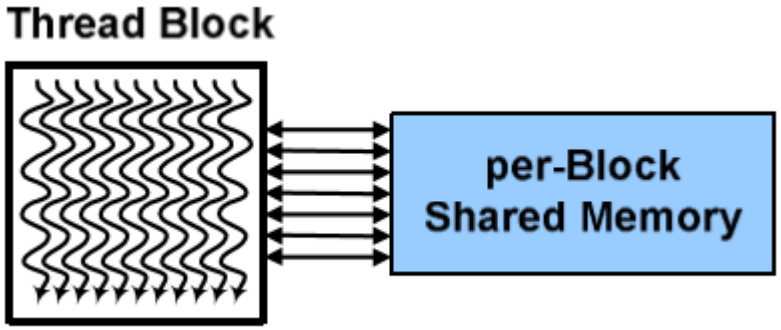
CUDA Parallel Computing Model

- Stands for Compute Unified Device Architecture (CUDA)
- CUDA is the HW/SW architecture that enables NVIDIA GPUs to execute programs written with C, C++, etc.
- The GPU instantiates a kernel program which executes in parallel by a large number of CUDA threads

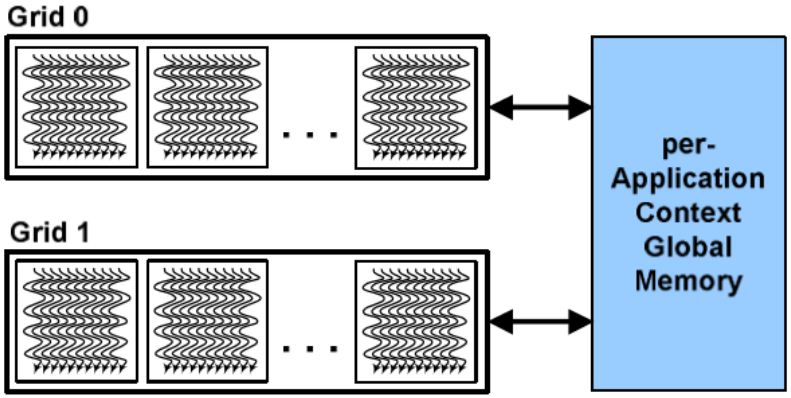


CUDA threads are different from POSIX threads

CUDA Parallel Computing Model



- A thread block is an array of cooperative thread
  - 1~512 concurrent threads
  - Each thread has a unique ID



- Grid: an array of thread blocks that execute the same kernel
  - Read/write from global memory
  - Sync between dependent calls

Hardware Execution

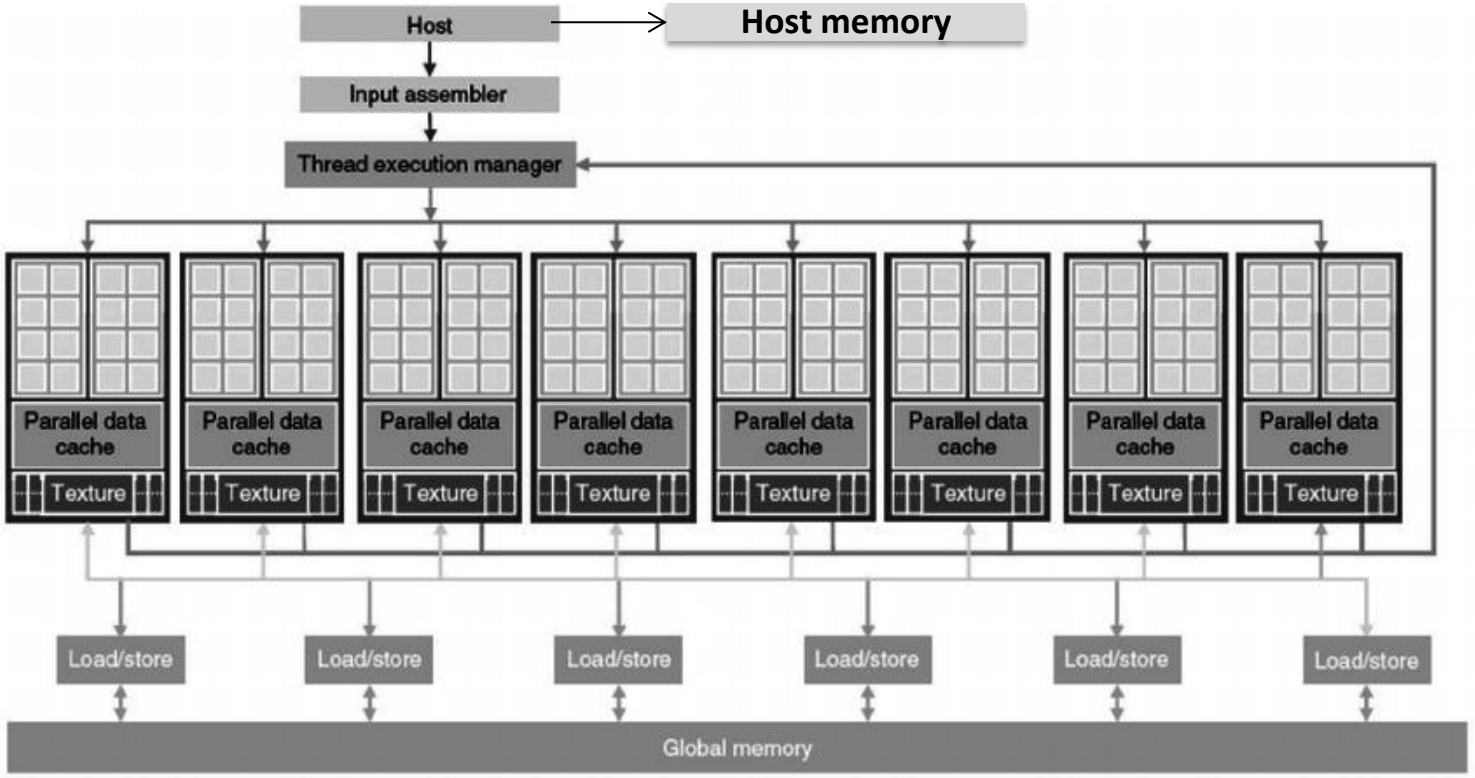
- The GPU hardware contains a collection of multithreaded SIMD processors the executes a **grid of thread blocks**

  - Thread Block Scheduler
    - Assigns thread blocks to multithreaded SIMD processors
  - SIMD Thread Scheduler
    - Schedules when threads of SIMD instructions should run

- The SIMD processor must have parallel function units
  - **SIMD Lanes** (# of lanes per SIMD varies across GPUs)

Case Study: Tesla GPU Architecture

- The primary design objective for Tesla was to execute vertex and pixel-fragment shader programs on the same unified processor architecture

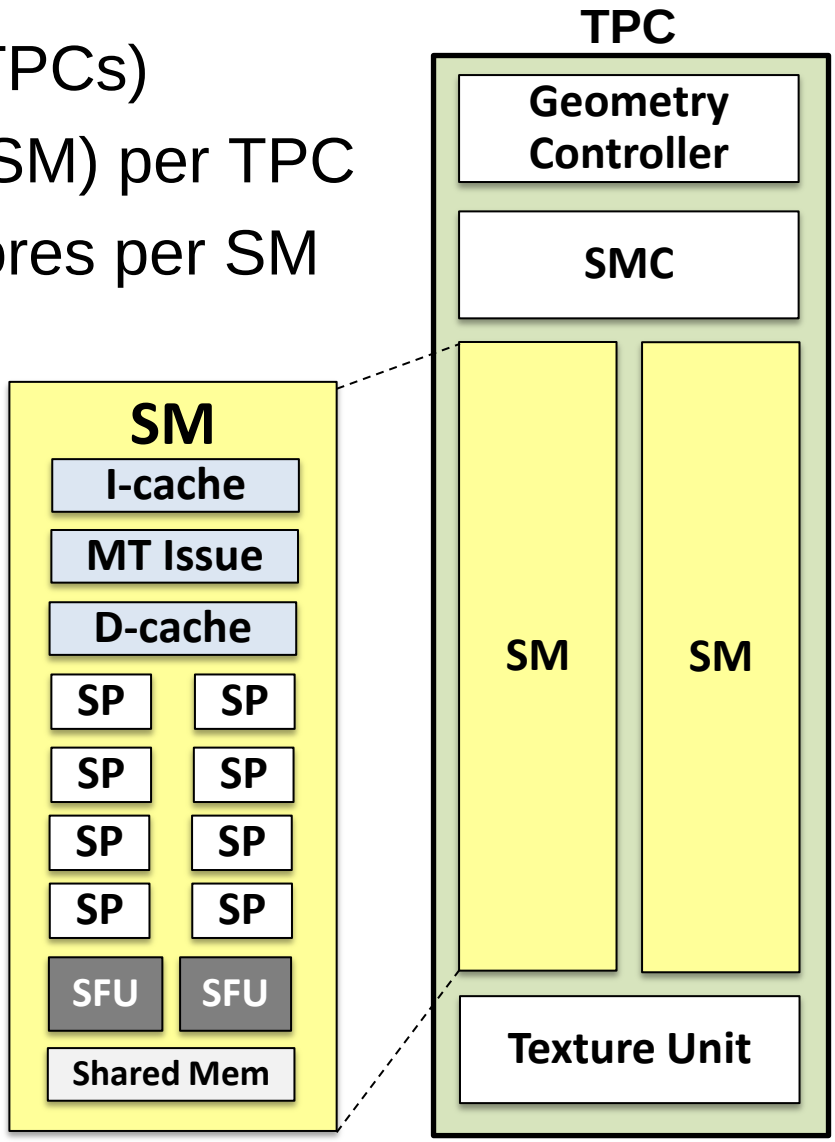


Case Study: Tesla GPU Architecture

- 8 texture/processor clusters (TPCs)
- 2 streaming multiprocessors (SM) per TPC
- 8 streaming processor (SP) cores per SM

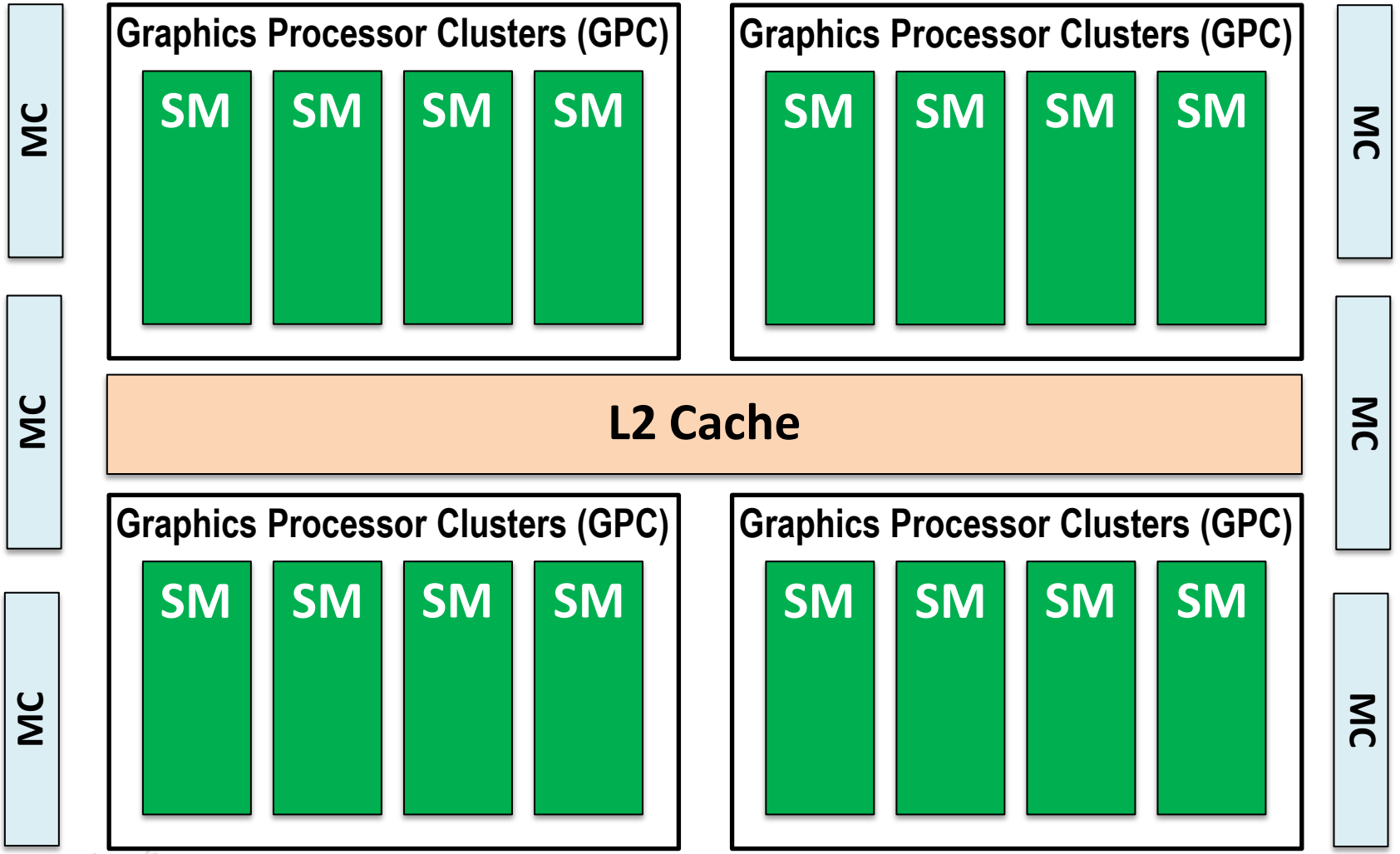
- SM is a unified graphics and computing engine
  - SM executes thread blocks

- Streaming processor (SP)
  - Scalar ALU for a single CUDA thread
  - SP core is a SIMD lane



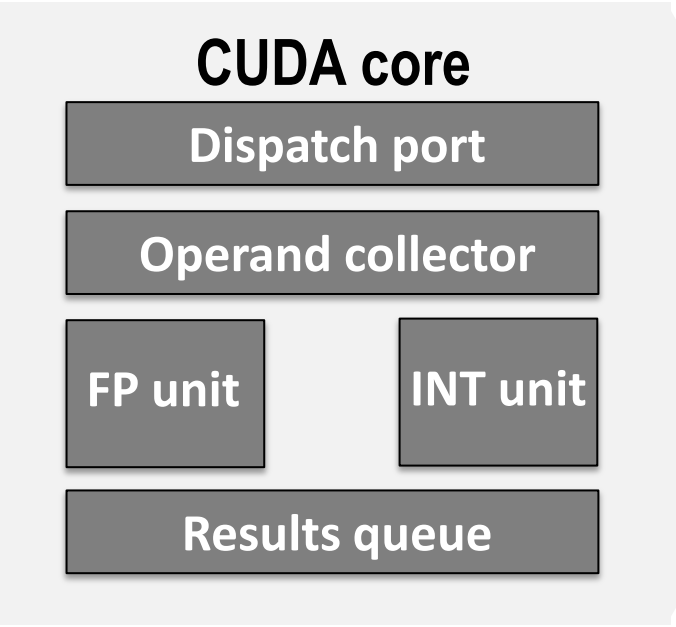
Case Study: Fermi GF100 Architectural Overview

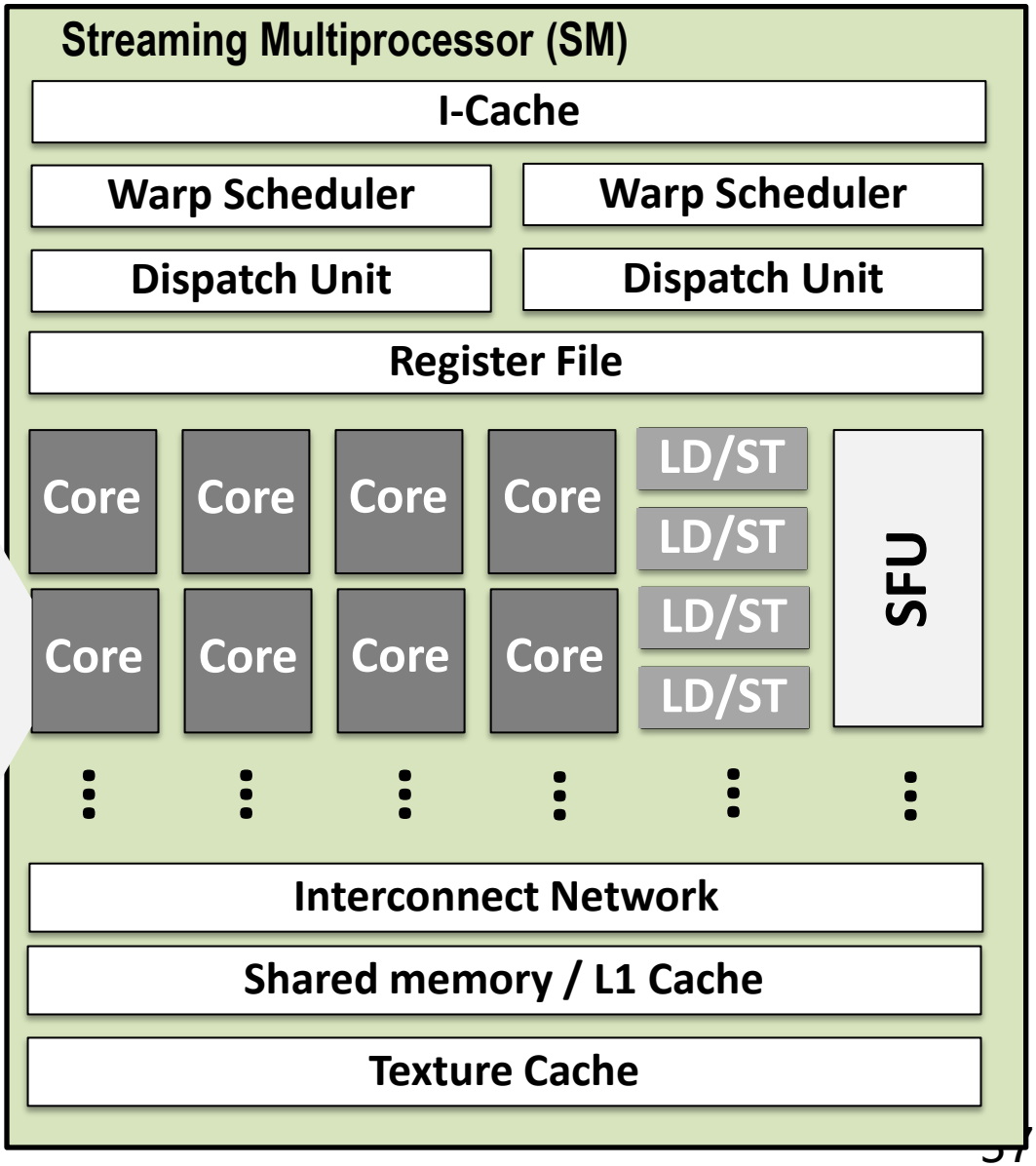
- Each GPC contains four streaming multiprocessors (SMs)



Case Study: Fermi GF100 Architectural Overview

- A single SM contains 32 CUDA cores
- Dual warp scheduler

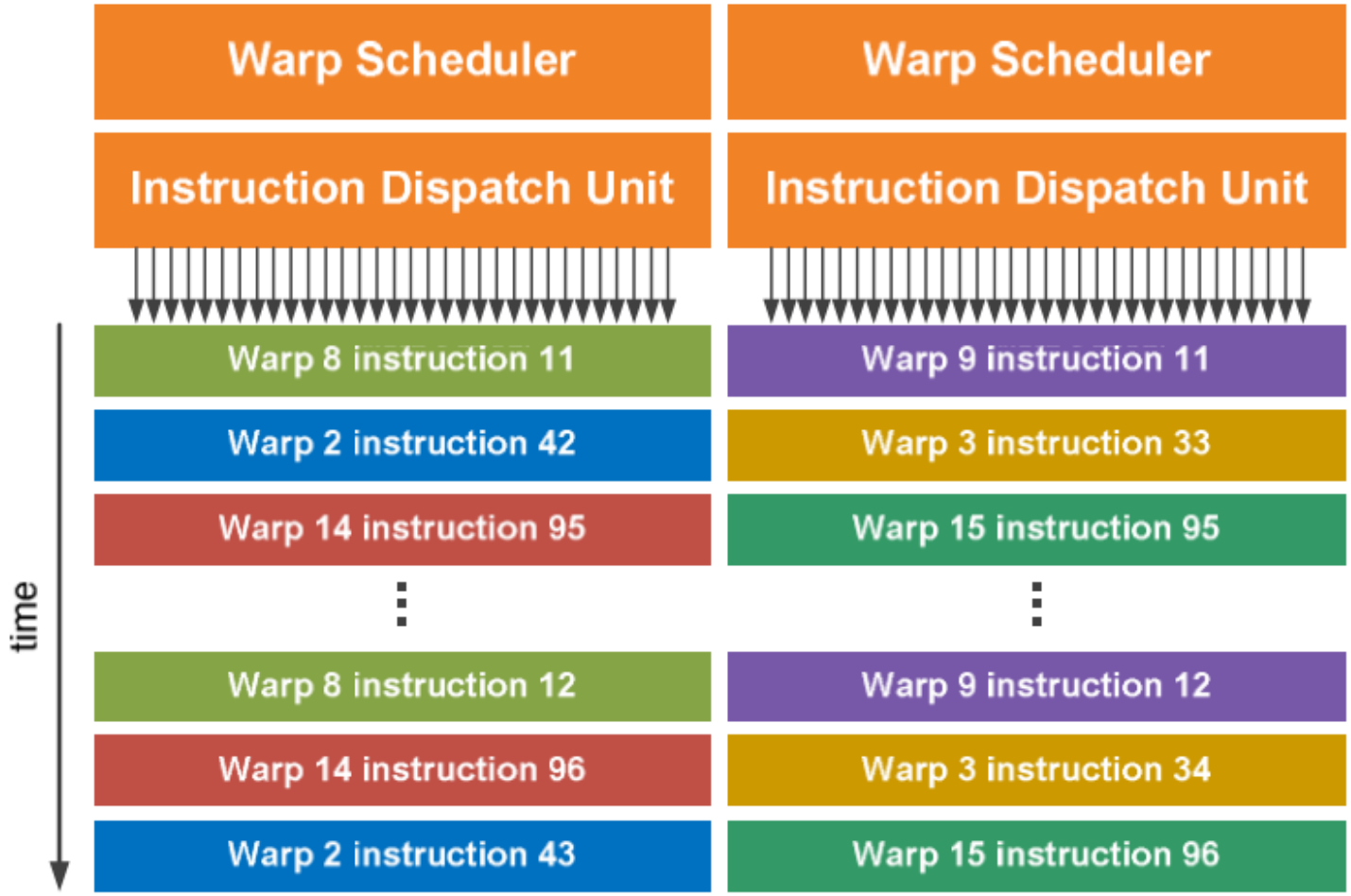




Warps：Schedule Threads in Batches

- GPUs rely on massive hardware multithreading to keep arithmetic units occupied
- SM executes a pool of threads organized into groups
  - It is called a “warp” by NVIDIA
  - It is called a “wavefront” by AMD
- Warp: 32 parallel CUDA threads of the same type
  - Start together at the same program address
  - All the threads in a warp use a common PC
  - Warps are scheduling units in SM

Fermi's Dual Warp Scheduler

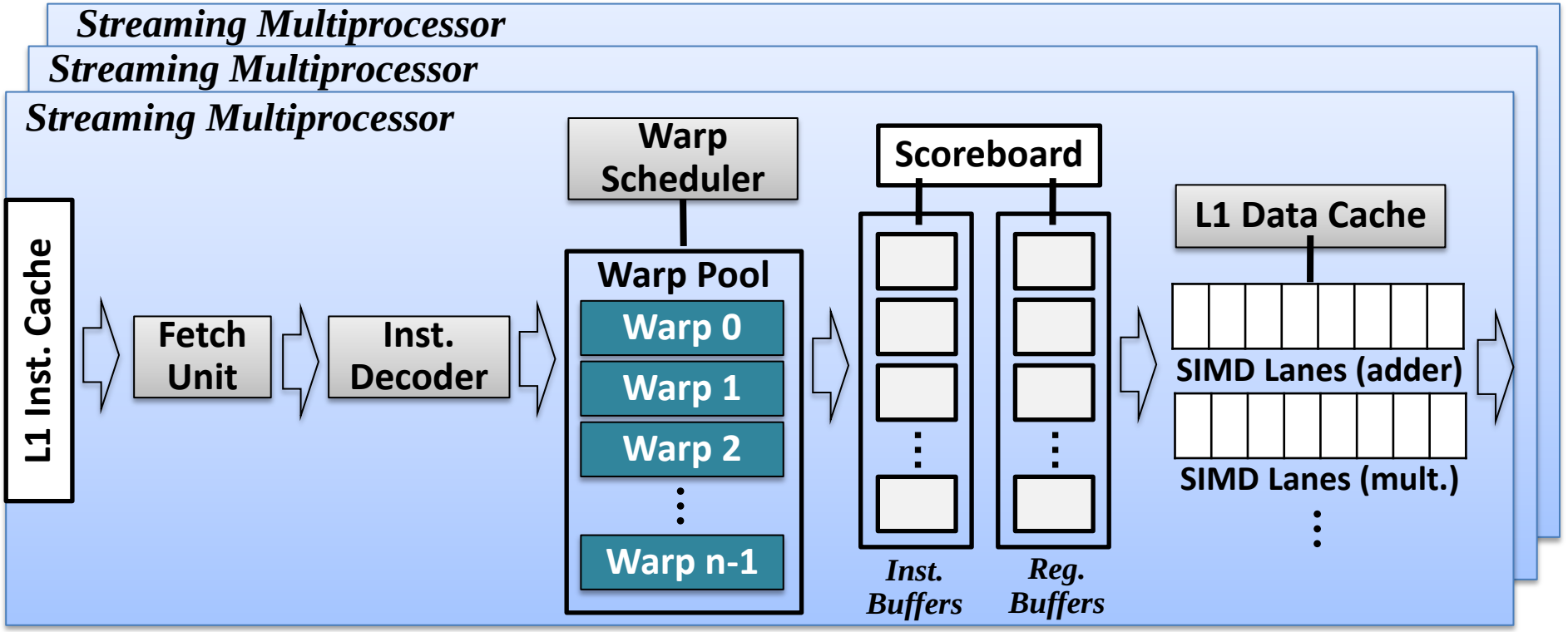


Simultaneously schedules and dispatches instructions from two independent warps

Warp Scheduling

- Thread scheduling happens at the granularity of warps
- Scheduler selects a warp from a pool for execution
  - Considers instruction type and fairness, etc.
- It is preferable that all the SIMD lanes are occupied
  - All 32 threads of a warp take the same execution path
- Latency-hiding capability of GPGPU
  - Fast context-switching
  - Large number of warps
  - Warps can be out-of-order

Warp Scheduling (Cont'd)



- Potential factors that can delay a warp's execution
  - Scheduling policies
  - Instruction/Data cache miss
  - Structural/Control/Data hazard
  - Synchronization primitives

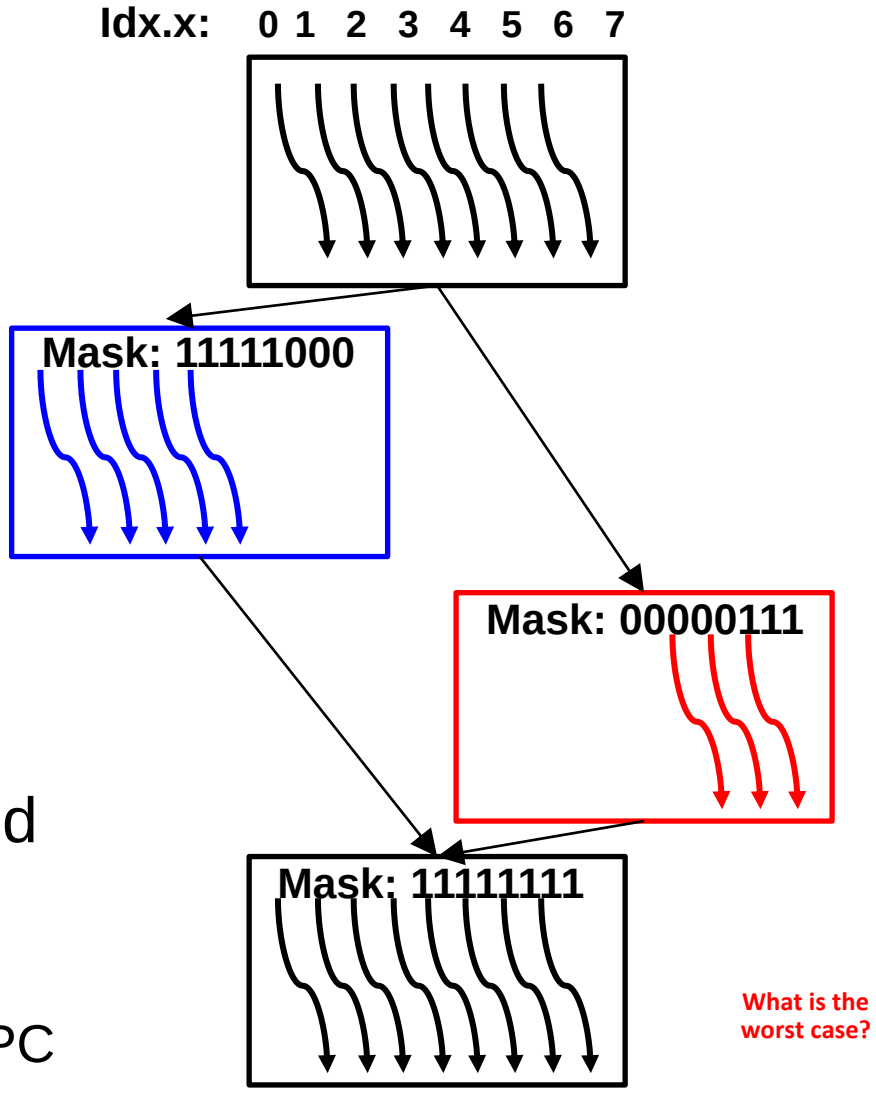
Branch Divergence

- Normally, all threads of a warp execute in the same way
- What if the threads of warp diverge due to control flow?
- Warp divergence: Branch instructions may cause some threads to jump, while others fall through in a warp
- Diverged warps possess inactive thread lanes
  - Diverging paths execute serially

Stack Based Re-Convergence

- Divergent Branch

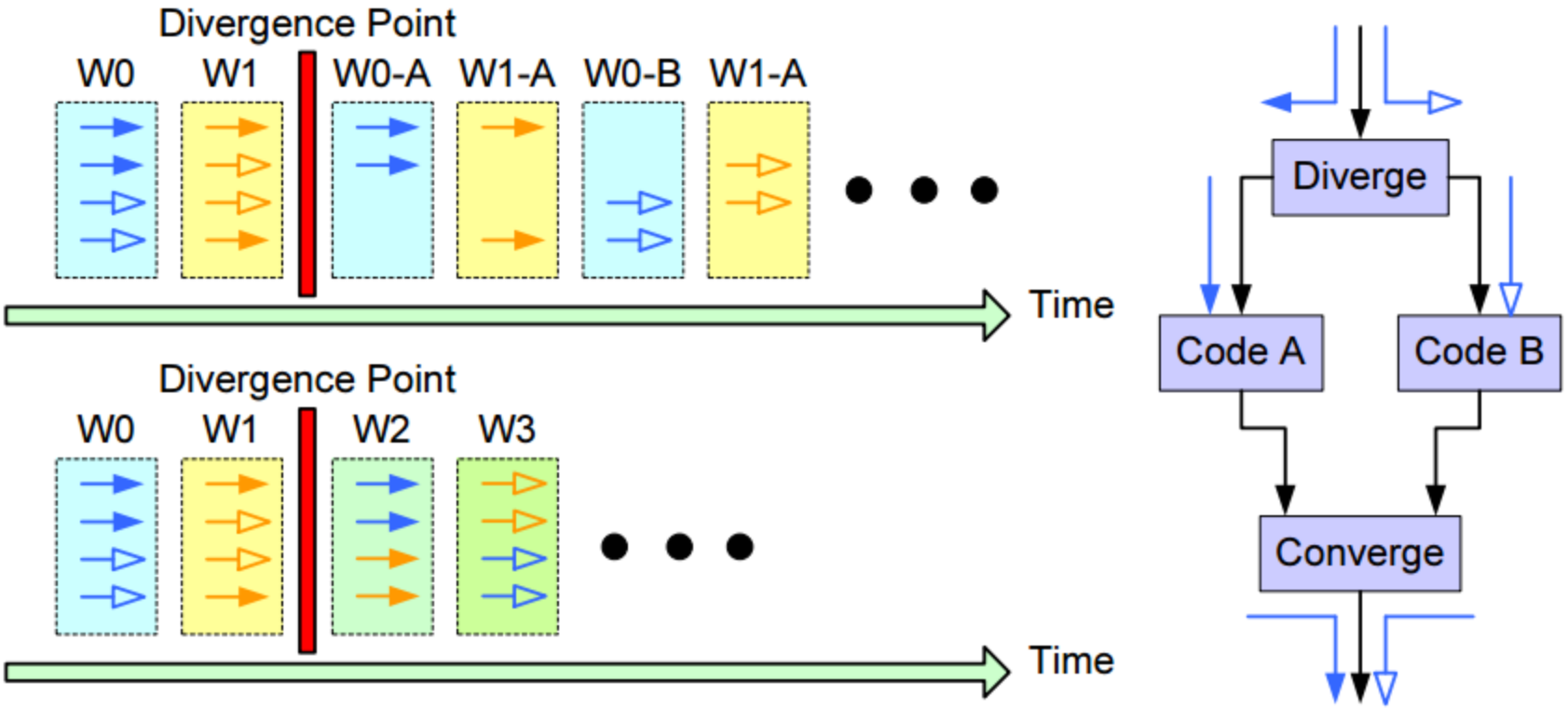
```
If (IDx.x < 5) {
my_data[IDx.x] += a
}
else {
my_data[IDx.x] *= b
}
```



- Re-convergence stack is used to merge back thread
  - Saves re-convergence PC, an active mask and an execution PC

Dynamic Warp Formation

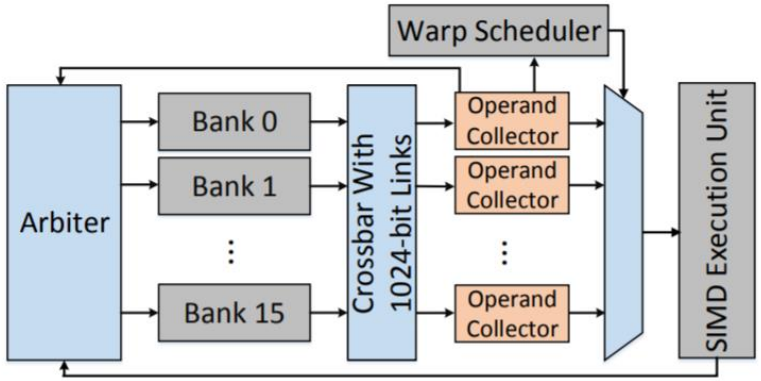
- Form new warps from a pool of ready threads by combining threads whose PC values are the same



Resource Limits

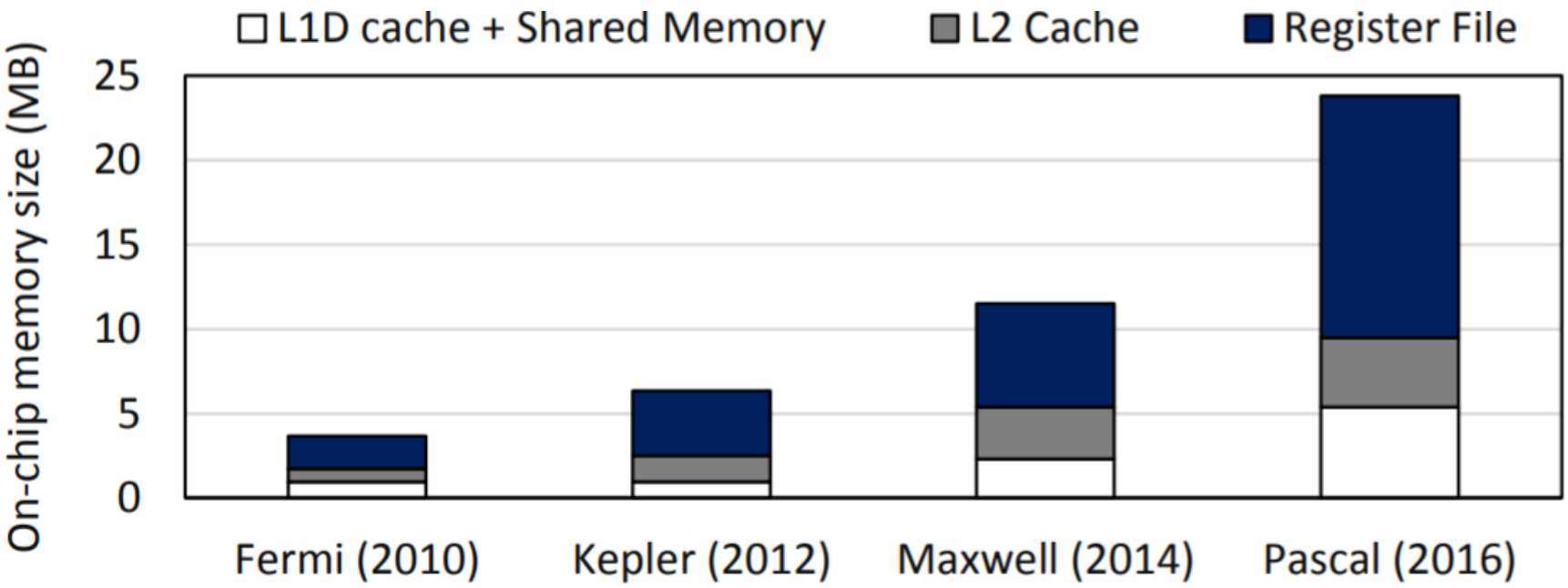
- The maximum parallelism in GPUs is often limited by the register file capacity
  - Applications with high TLP triggers more active warps

- Register file is a large SRAM structure
  - Fastest memory block available to the processor
  - Store intermediate results from units such as ALU
  - Power hungry structure



GPU Register File

- The size of RF is often greater than the L2/L1 $
- Its size keeps increasing



Design Considerations

- Large register file capacity
  - Pros:
    - Can accommodate more active threads
    - Enable frequent context switching with low overhead
    - Allows GPUs to fully utilize its memory bandwidth
  - Cons:
    - High power consumption, large silicon area, etc.
- Optimization approaches
  - Shrink register file size?
  - Use new memory technology?

Summary

- Throughput computing and data-level parallelism
- Vector processor and vector instruction
- VMPIS, vector registers, DAXPY, execution latency
- SIMD lanes, chaining, vector length register
- GPGPU, SIMT, CUDA programming model
- TPC, SM, SP, warp and warp scheduling
- Branch divergence
- GPU register file

References

- 课本内容：J. Hennessy, D. Patterson. Computer Architecture, Fifth Edition: A Quantitative Approach.
  - Chapters: 4.1, 4.2, 4.4

- 参考阅读：
  - [1] E. Lindholm et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro Magazine*, 2008.
  - [2] C. Wittenbrink et al. Fermi GF100 GPU Architecture. *IEEE Micro Magazine*, 2011.
  - [3] W. Fung et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow, *MICRO* 2007

Exercises

- Think about vectors vs superscalar vs. VLIW architecture
- What is the difference between SIMD and SIMT
- How do you compare CPU and GPU
- What are the advantages of dynamic warp formation?