

18-447

Computer Architecture
Lecture 15: GPUs, VLIW, DAE

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015, 2/20/2015

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Issues in OoO Execution: Load-Store Handling, ...
- Alternative Approaches to Instruction Level Parallelism

Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- SIMD Processing (Vector and array processors, GPUs)
- VLIW
- Decoupled Access Execute
- Systolic Arrays

Homework 3.1: Feedback Form

- Due Monday Feb 23
- I would like your feedback on the course
- Easy to fill in
- Can submit anonymously, if you wish
- Worth 0.25% of your grade (extra credit)
- Need to get checked off after submitting to get your grade points
 - Can email
 - If anonymous, show that you are turning in and have a TA check you off

A Couple of Things

- Midterm I Date
 - March 4?
 - March 18?
- Collaboration on Labs
 - All labs individual – no collaboration permitted
- Collaboration on homeworks
 - You can collaborate
 - But need to submit individual writeups on your own

Readings for Today

- Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.
- Fatahalian and Houston, "A Closer Look at GPUs," CACM 2008.

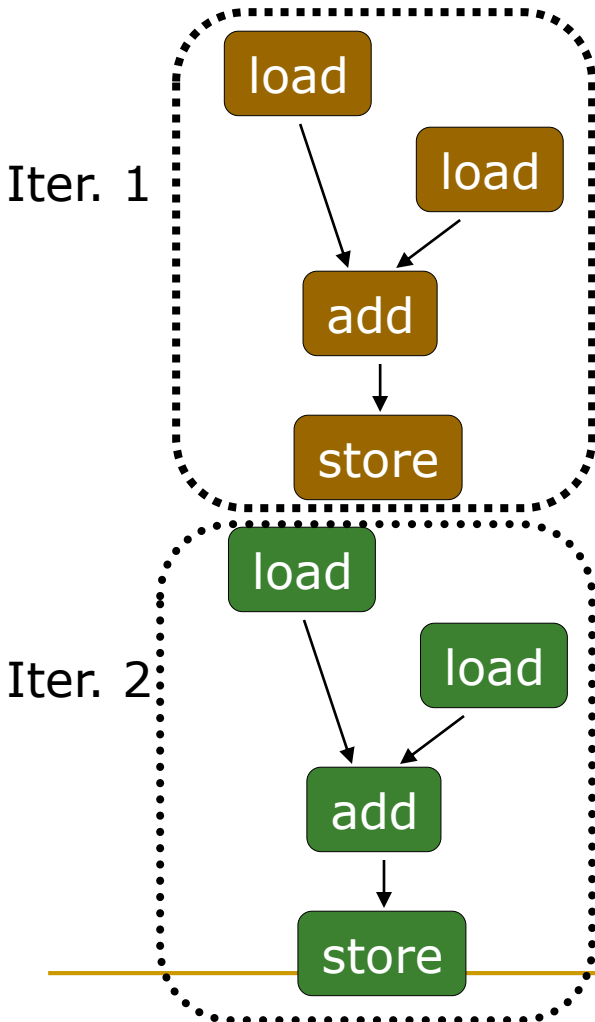
Recap of Last Lecture

- SIMD Processing
- Flynn's taxonomy: SISD, SIMD, MISD, MIMD
- VLIW vs. SIMD
- Array vs. Vector Processors
- Vector Processors in Depth
 - Vector Registers, Stride, Masks, Length
 - Memory Banking
 - Vectorizable Code
 - Scalar vs. Vector Code Execution
 - Vector Chaining
 - Vector Stripmining
 - Gather/Scatter Operations
 - Minimizing Bank Conflicts
 - Automatic Code Vectorization
- SIMD Operations in Modern ISAs: Example from MMX

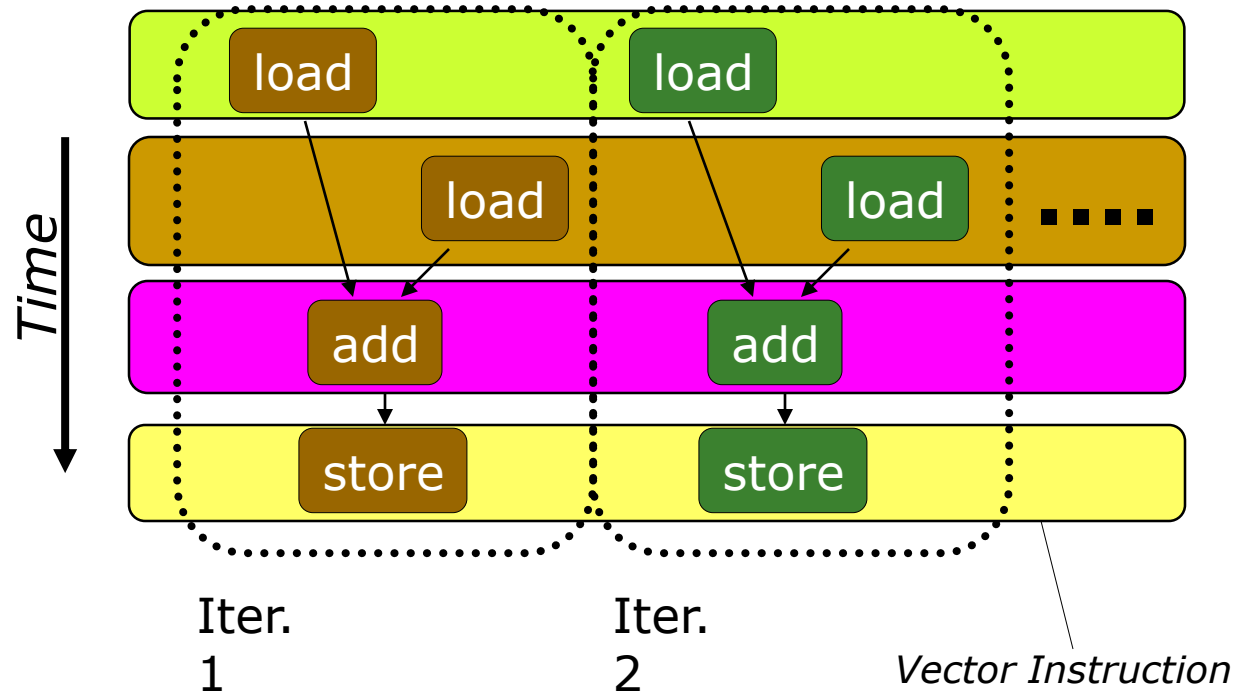
Review: Code Parallelization/Vectorization

```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```

Scalar Sequential Code



Vectorized Code



Vectorization is a compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

Recap: Vector/SIMD Processing Summary

- Vector/SIMD machines are good at exploiting **regular data-level parallelism**
 - ❑ Same operation performed on many data elements
 - ❑ Improve performance, simplify design (no intra-vector dependencies)
- **Performance improvement limited by vectorizability** of code
 - ❑ Scalar operations limit vector machine performance
 - ❑ Remember Amdahl's Law
 - ❑ CRAY-1 was the fastest SCALAR machine at its time!
- Many existing ISAs include SIMD operations
 - ❑ Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

GPUs are SIMD Engines Underneath

- The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)
- However, the programming is done using threads, NOT SIMD instructions
- To understand this, let's go back to our parallelizable code example
- But, before that, let's distinguish between
 - Programming Model (Software)
 - vs.
 - Execution Model (Hardware)

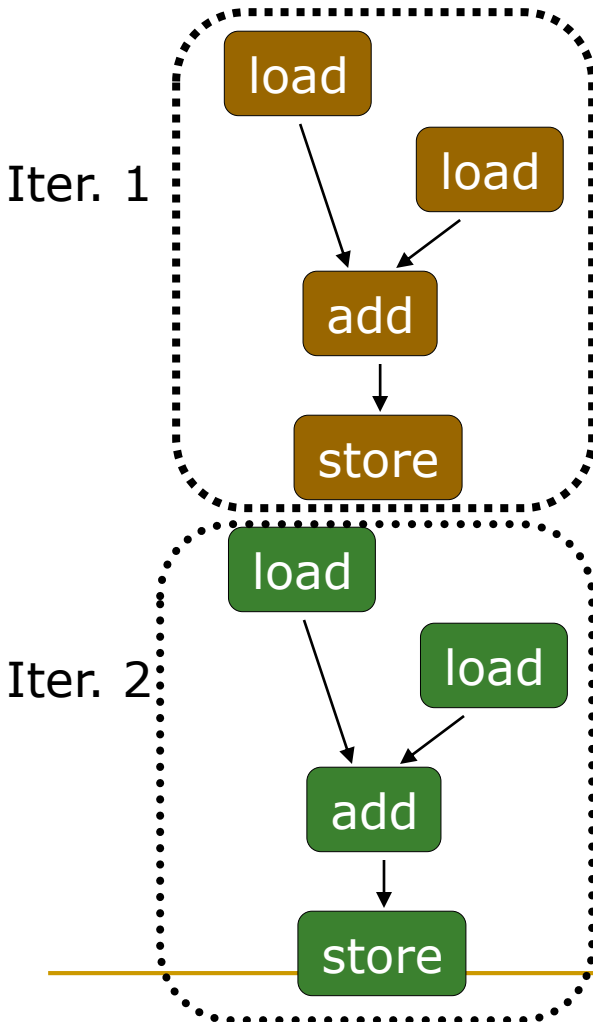
Programming Model vs. Hardware Execution Model

- Programming Model refers to **how the programmer expresses the code**
 - E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), ...
- Execution Model refers to **how the hardware executes the code underneath**
 - E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, ...
- **Execution Model can be very different from the Programming Model**
 - E.g., von Neumann model implemented by an OoO processor
 - E.g., SPMD model implemented by a SIMD processor (a GPU)

How Can You Exploit Parallelism Here?

```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```

Scalar Sequential Code



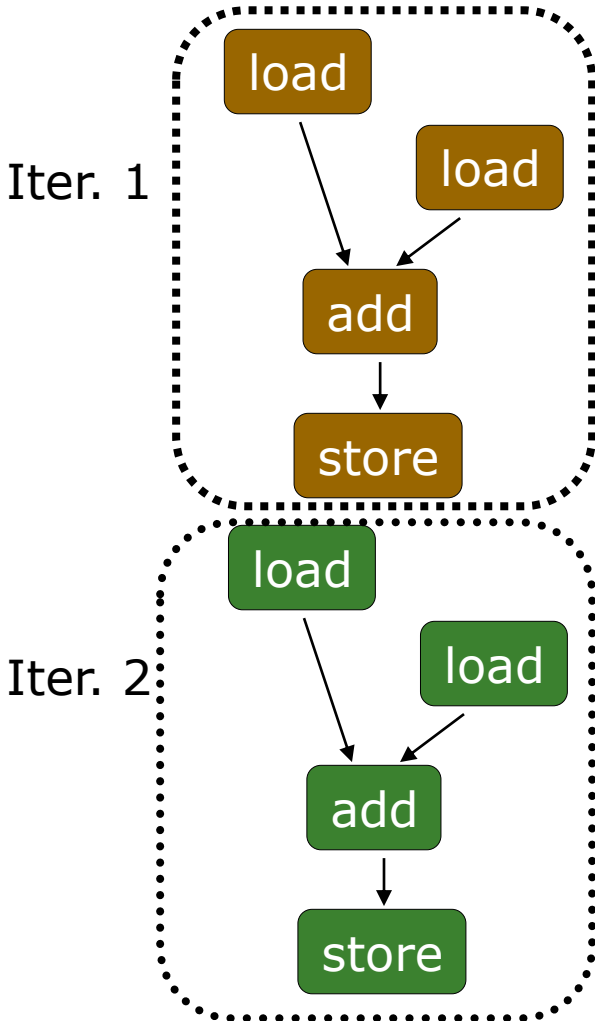
Let's examine three programming options to exploit instruction-level parallelism present in this sequential code:

1. Sequential (SISD)
2. Data-Parallel (SIMD)
3. Multithreaded (MIMD/SPMD)

Prog. Model 1: Sequential (SISD)

```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```

Scalar Sequential Code



- Can be executed on a:
 - Pipelined processor
 - Out-of-order execution processor
 - ❑ Independent instructions executed when ready
 - ❑ Different iterations are present in the instruction window and can execute in parallel in multiple functional units
 - ❑ In other words, the loop is dynamically unrolled by the hardware
- Superscalar or VLIW processor
 - ❑ Can fetch and execute multiple instructions per cycle

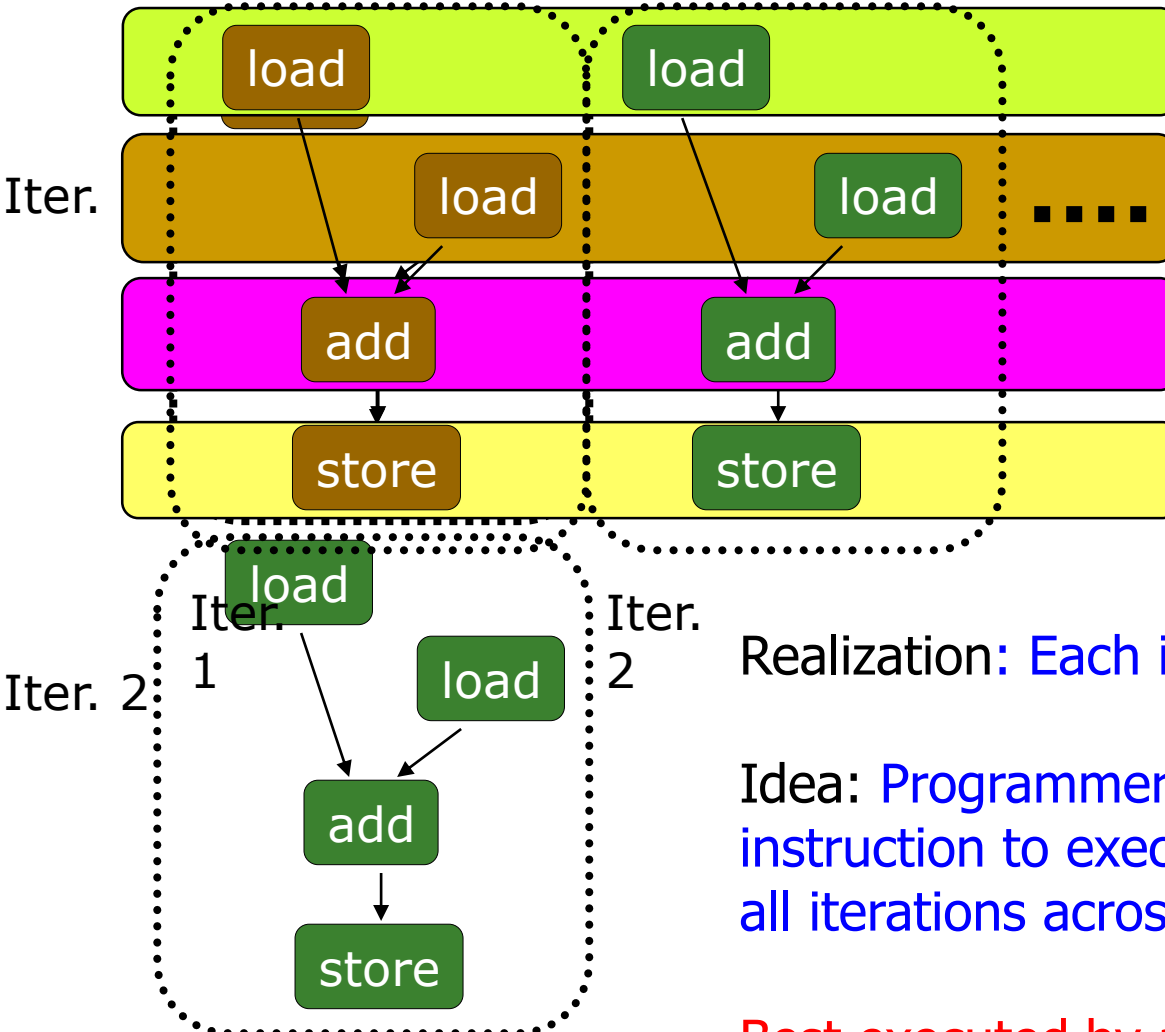
Prog. Model 2: Data Parallel (SIMD)

```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```

Scalar Sequential Code

Vector Instruction

Vectorized Code



Realization: Each iteration is independent

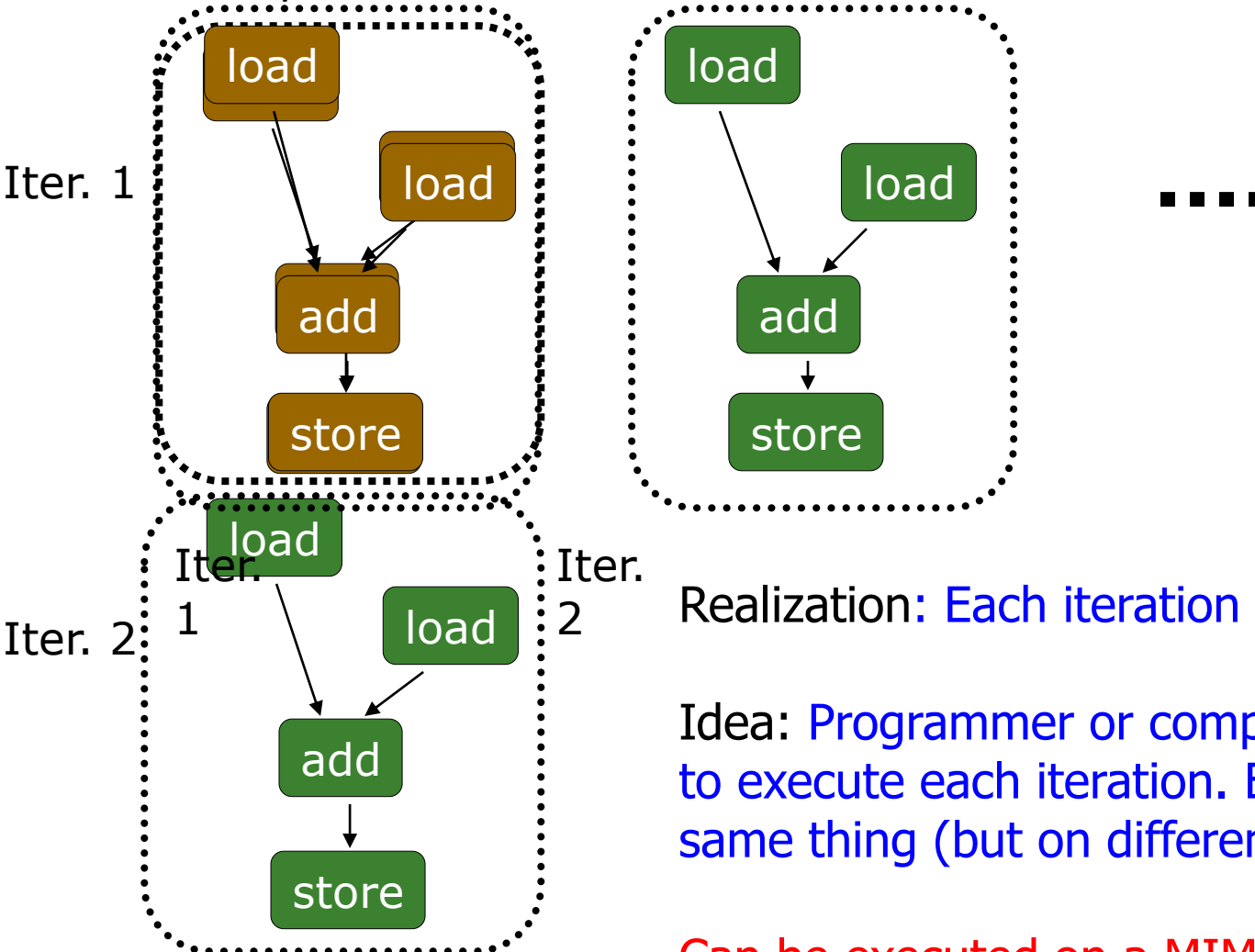
Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Best executed by a SIMD processor (vector, array)

Prog. Model 3: Multithreaded

```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```

Scalar Sequential Code



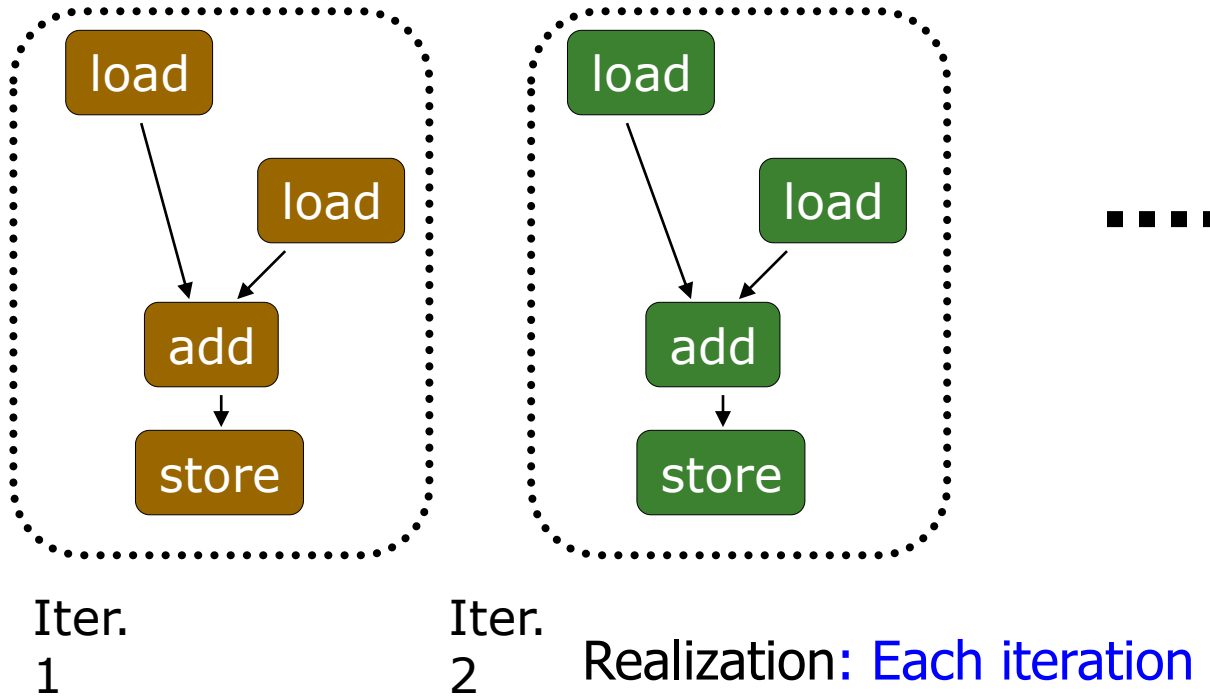
Realization: Each iteration is independent

Idea: Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

Can be executed on a MIMD machine

Prog. Model 3: Multithreaded

```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```



This particular model is also called:

SPMD: Single Program Multiple Data

Can be executed on a SIMT machine

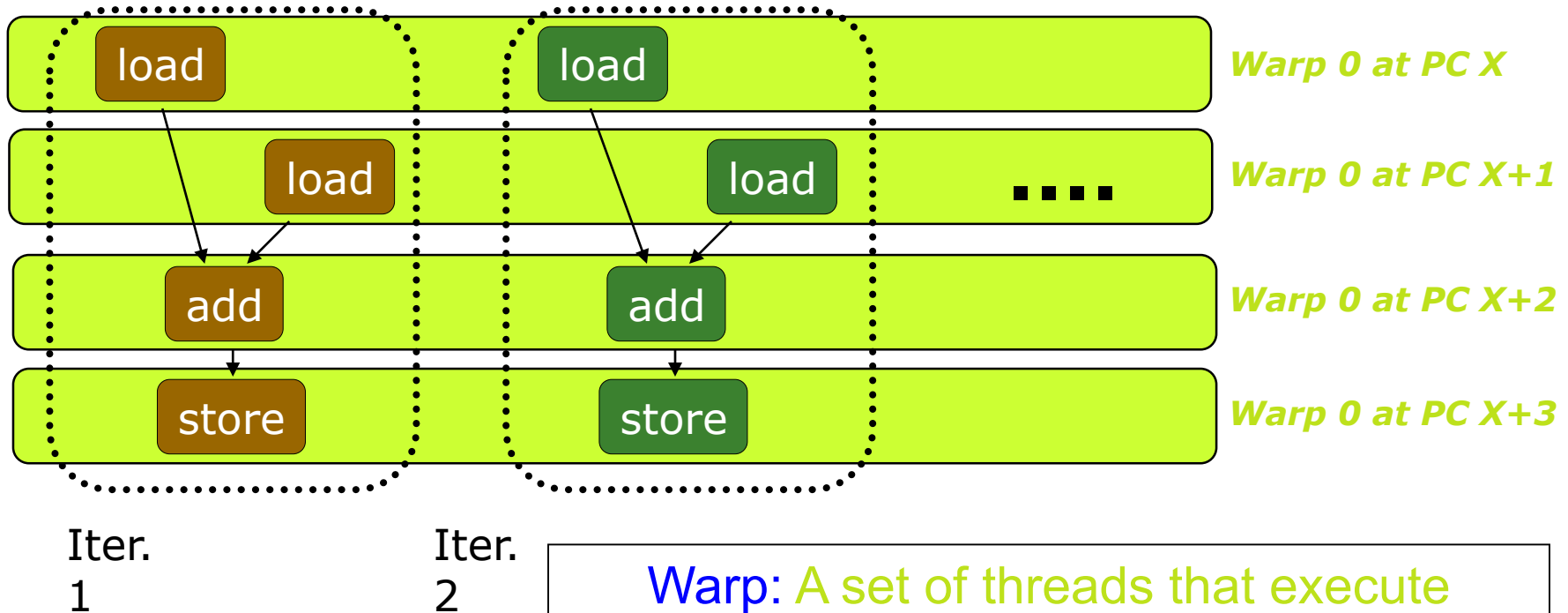
Single Instruction Multiple Thread

A GPU is a SIMD (SIMT) Machine

- Except it is not programmed using SIMD instructions
- It is programmed using threads (SPMD programming model)
 - Each thread executes the same code but operates a different piece of data
 - Each thread has its own context (i.e., can be treated/restarted/executed independently)
- A set of threads executing the same instruction are dynamically grouped into a **warp (wavefront)** by the hardware
 - A warp is essentially a SIMD operation formed by hardware!

SPMD on SIMT Machine

```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```



Warp: A set of threads that execute the same instruction (i.e., at the same PC)

This particular model is also called:

SPMD: Single Program Multiple Data

A GPU executes it using the SIMT model:
Single Instruction Multiple Thread

Graphics Processing Units

SIMD not Exposed to Programmer (SIMT)

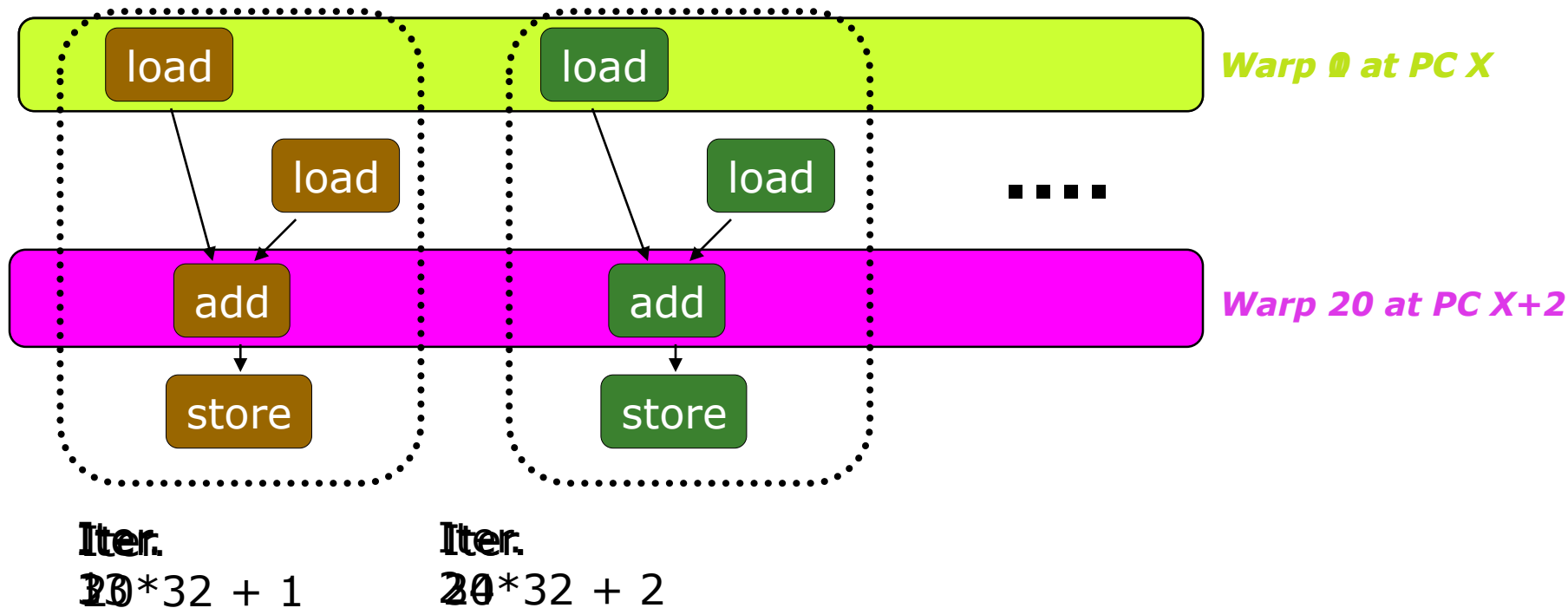
SIMD vs. SIMT Execution Model

- SIMD: A single **sequential instruction stream** of **SIMD instructions** → each instruction specifies multiple data inputs
 - [VLD, VLD, VADD, VST], VLEN
- SIMT: **Multiple instruction streams** of **scalar instructions** → threads grouped dynamically into warps
 - [LD, LD, ADD, ST], NumThreads
- Two Major SIMT Advantages:
 - **Can treat each thread separately** → i.e., can execute each thread independently (on any type of scalar pipeline) → MIMD processing
 - **Can group threads into warps flexibly** → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

Multithreading of Warps

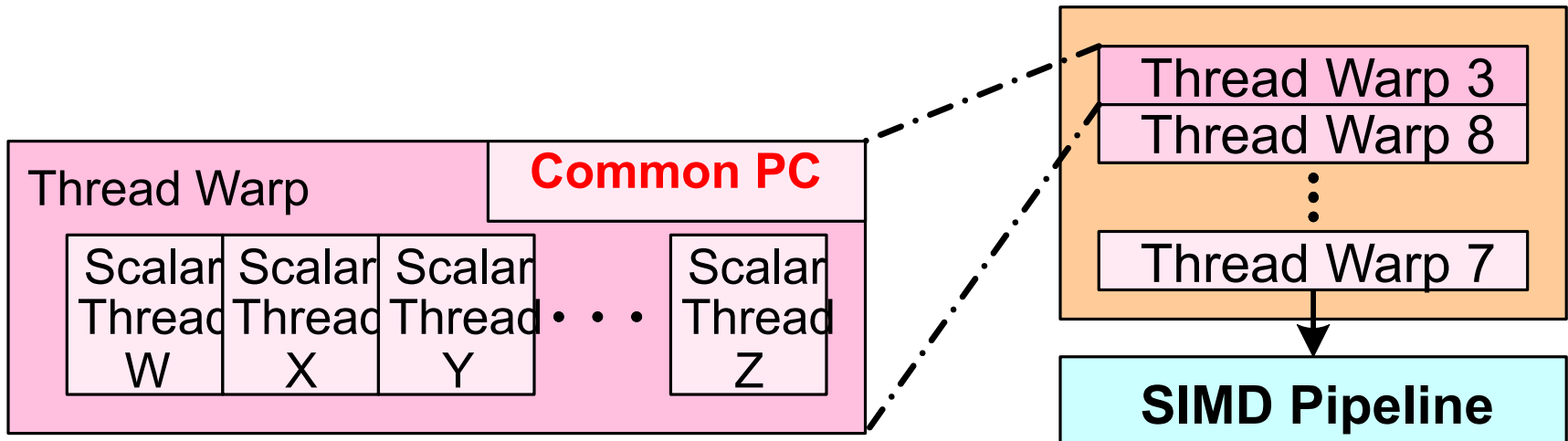
```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```

- Assume a warp consists of 32 threads
- If you have 32K iterations \rightarrow 1K warps
- Warps can be interleaved on the same pipeline \rightarrow Fine grained multithreading of warps

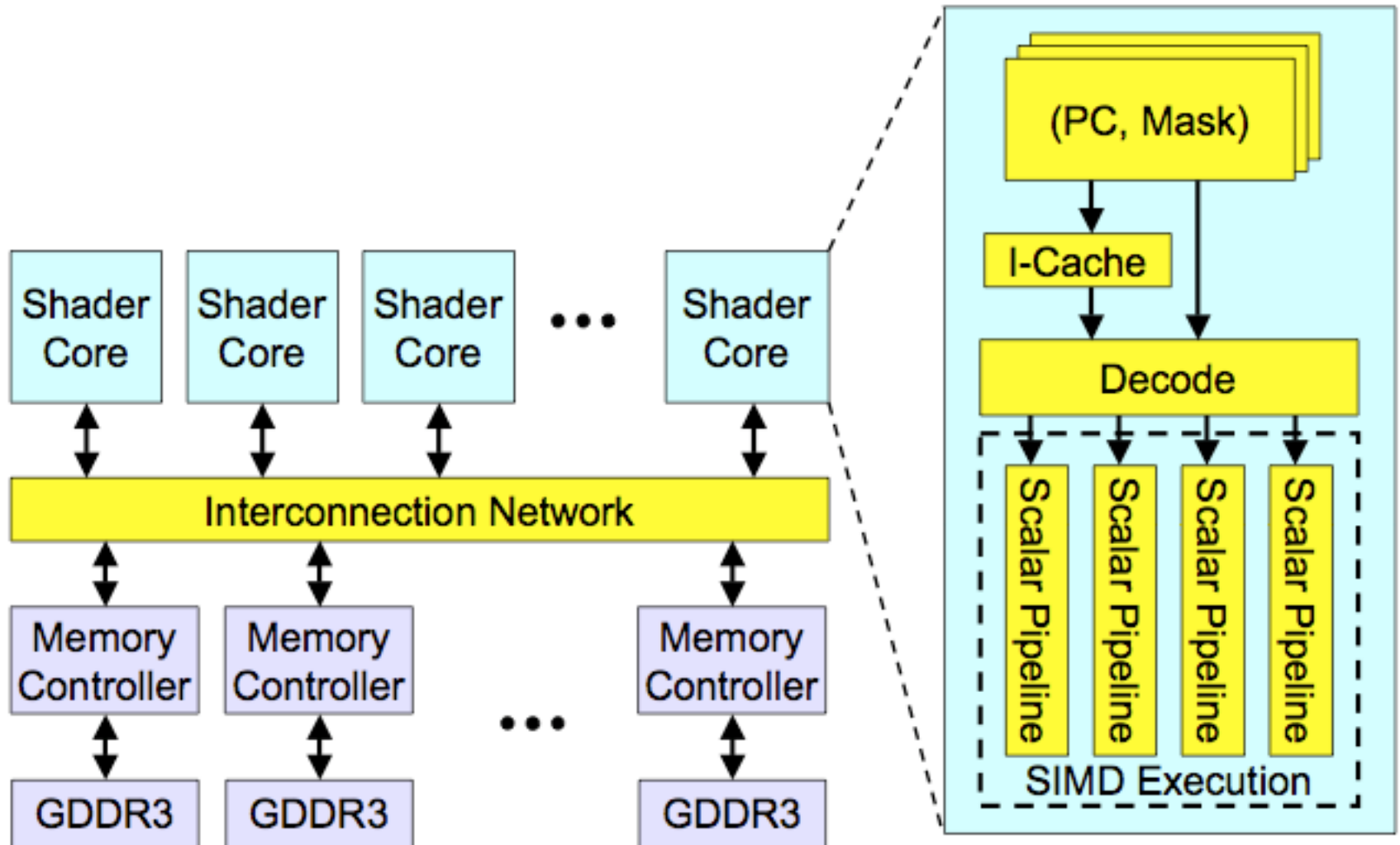


Warps and Warp-Level FGMT

- Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)
- All threads run the same code
- Warp: The threads that run lengthwise in a woven fabric ...

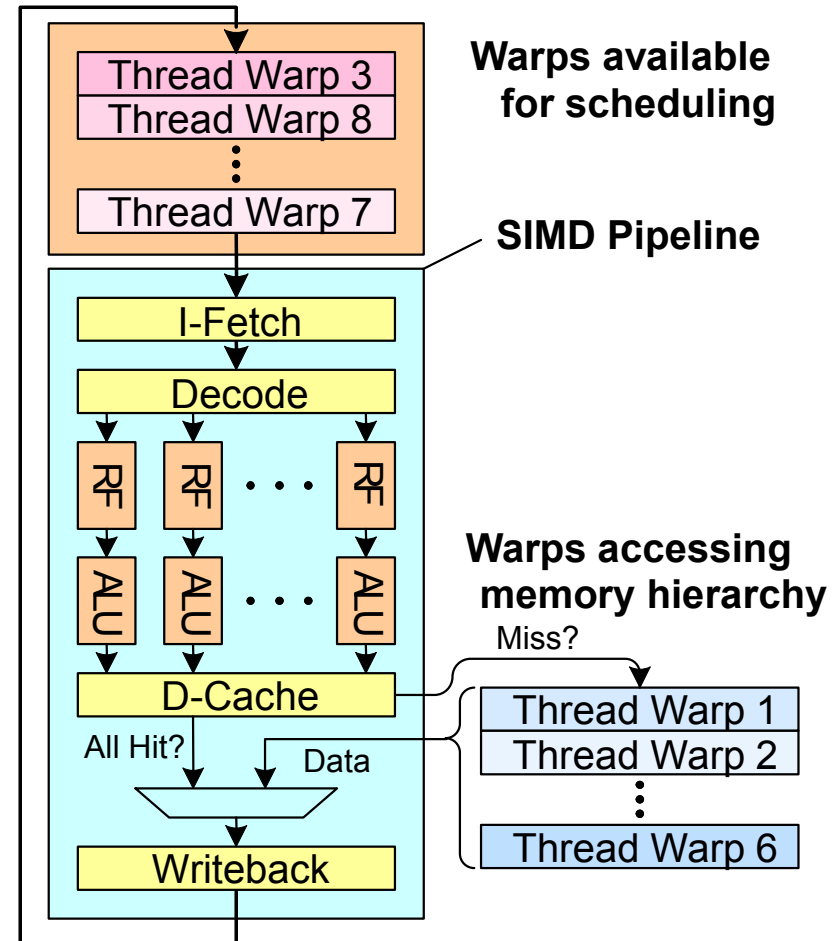


High-Level View of a GPU



Latency Hiding via Warp-Level FGMT

- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
 - One instruction per thread in pipeline at a time (No interlocking)
 - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables long latency tolerance
 - Millions of pixels

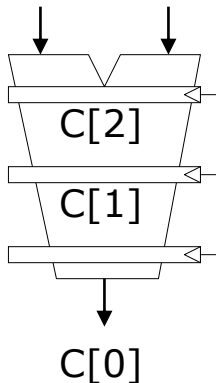


Warp Execution (Recall the Slide)

32-thread warp executing $\text{ADD } A[\text{tid}], B[\text{tid}] \rightarrow C[\text{tid}]$

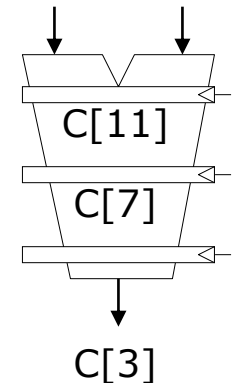
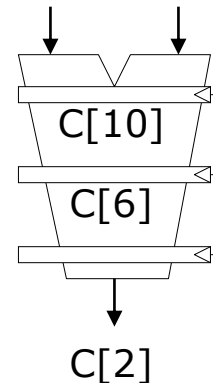
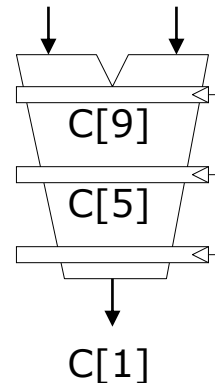
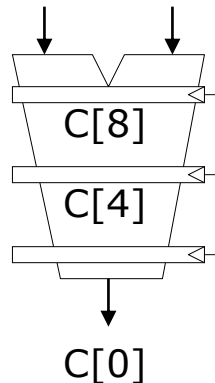
*Execution using
one pipelined
functional unit*

A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]

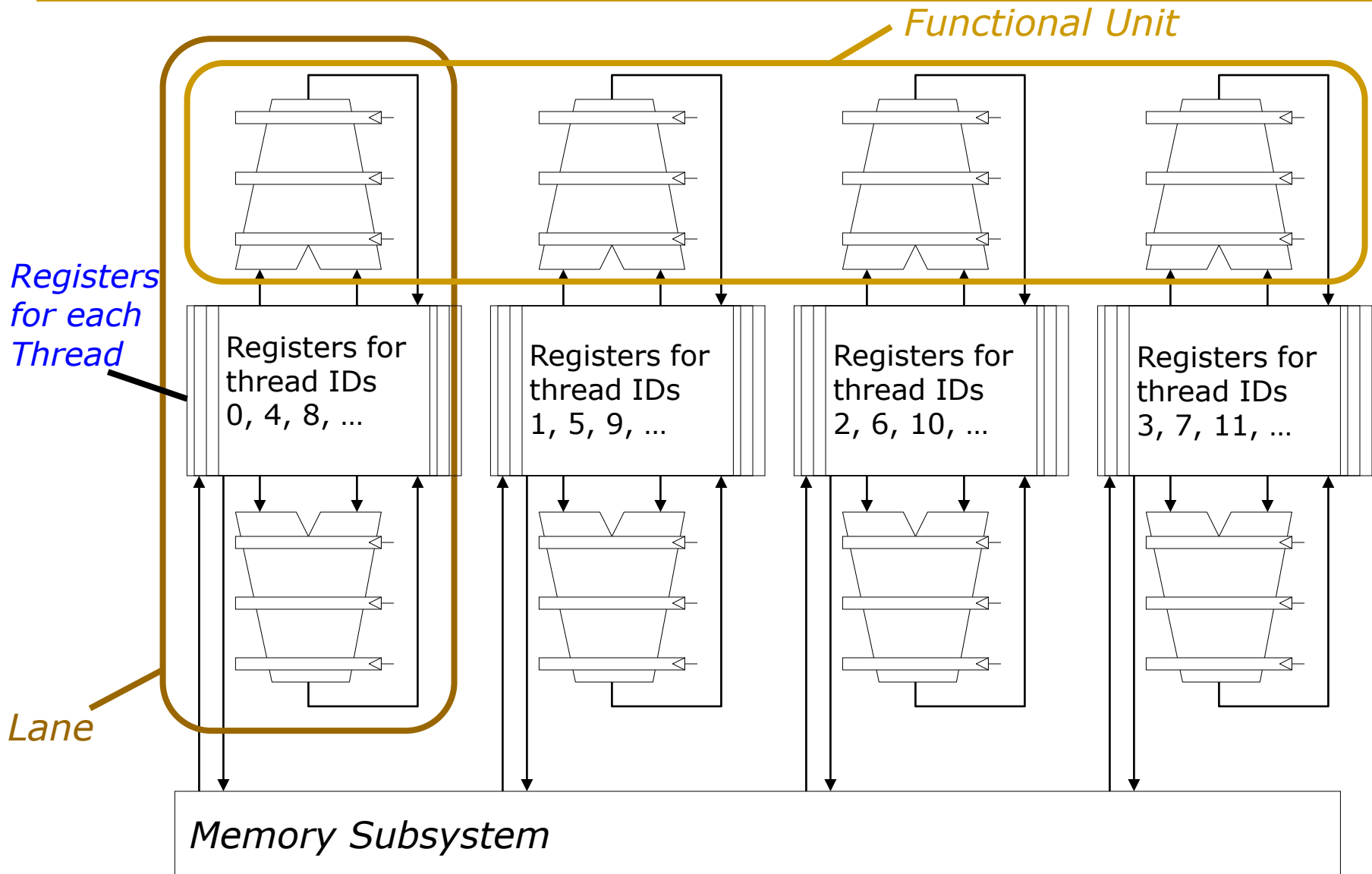


*Execution using
four pipelined
functional units*

A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]



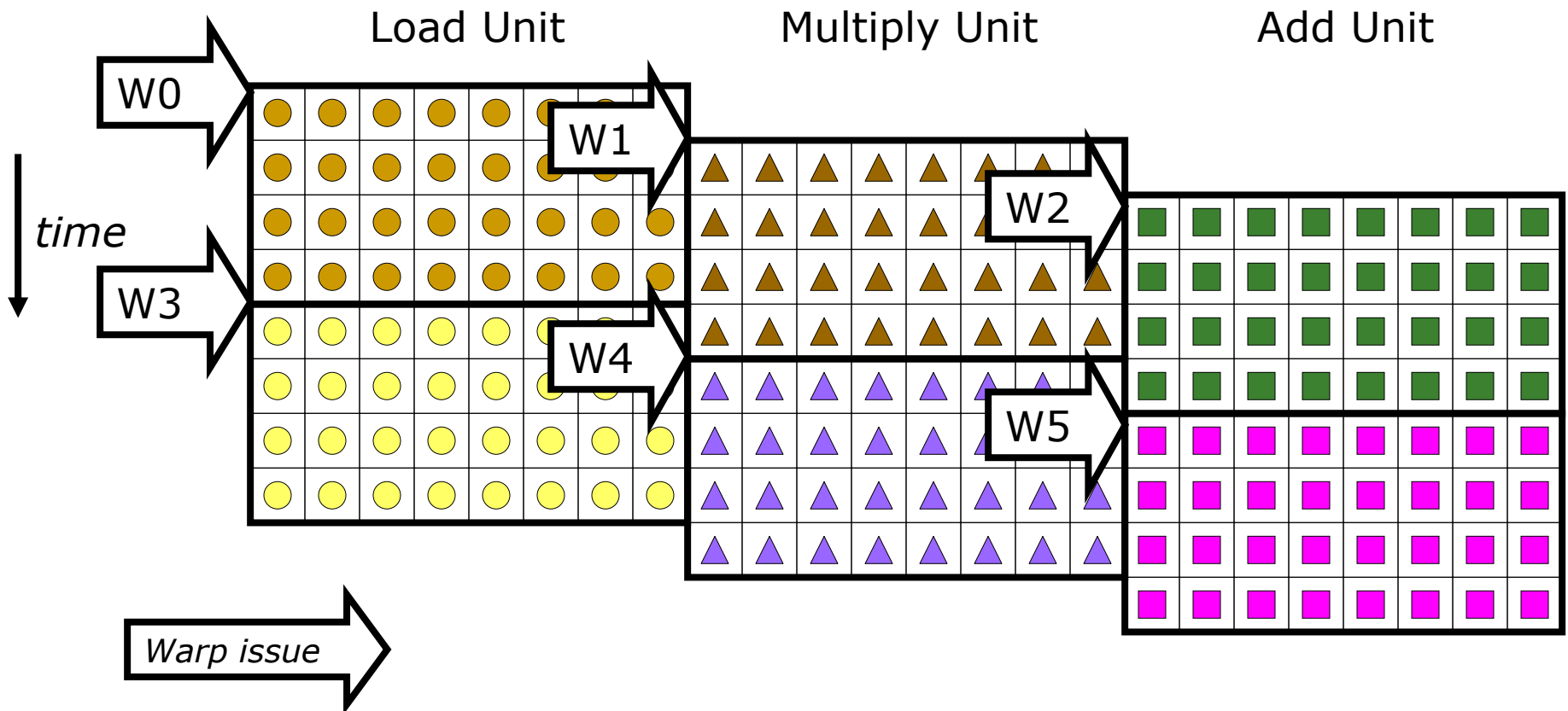
SIMD Execution Unit Structure



Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

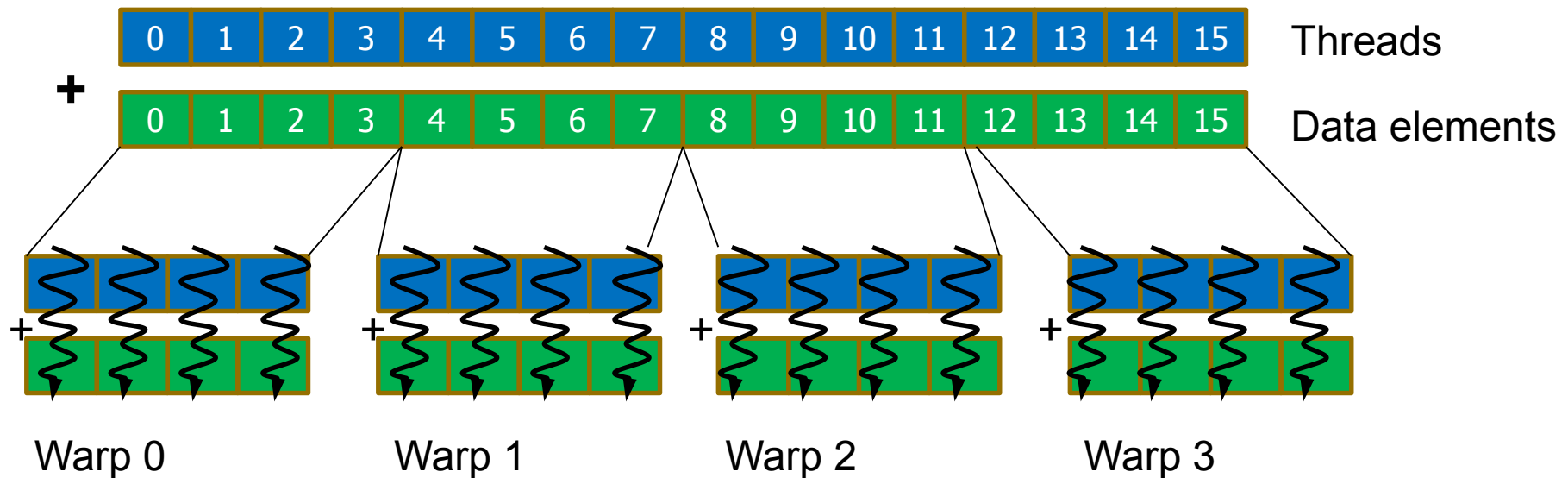
- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle



SIMT Memory Access

- Same instruction in different threads uses thread id to index and access different data elements

Let's assume $N=16$, 4 threads per warp \rightarrow 4 warps



Sample GPU SIMT Code (Simplified)

CPU code

```
for (ii = 0; ii < 100000; ++ii) {  
    C[ii] = A[ii] + B[ii];  
}
```



CUDA code

```
// there are 100000 threads  
__global__ void KernelFunction(...) {  
    int tid = blockDim.x * blockIdx.x + threadIdx.x;  
    int varA = aa[tid];  
    int varB = bb[tid];  
    C[tid] = varA + varB;  
}
```

Sample GPU Program (Less Simplified)

CPU Program

```
void add matrix
( float *a, float* b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
}

int main () {

    add matrix (a, b, c, N);
}
```

GPU Program

```
__global__ add_matrix
( float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i < N && j < N)
        c[index] = a[index]+b[index];
}

int main() {
    dim3 dimBlock( blocksize, blocksize) ;
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```

Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
 - Lock step: a vector instruction needs to finish before another can start
 - Programming model is SIMD (no extra threads) → SW needs to know vector length
 - ISA contains vector/SIMD instructions

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
 - Does not have to be lock step
 - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
 - SW does not need to know vector length
 - Enables multithreading and flexible dynamic grouping of threads
 - ISA is scalar → vector instructions can be formed dynamically
 - Essentially, it is SPMD programming model implemented on SIMD hardware

SPMD

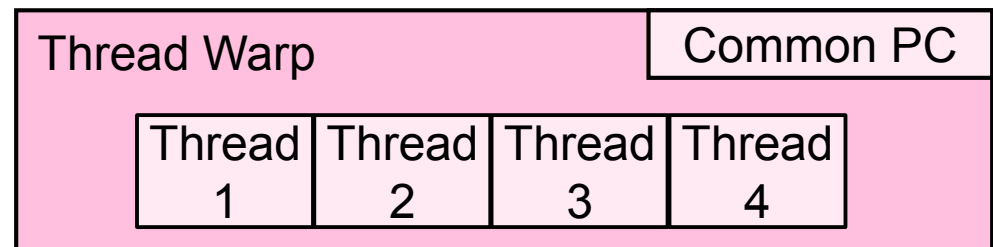
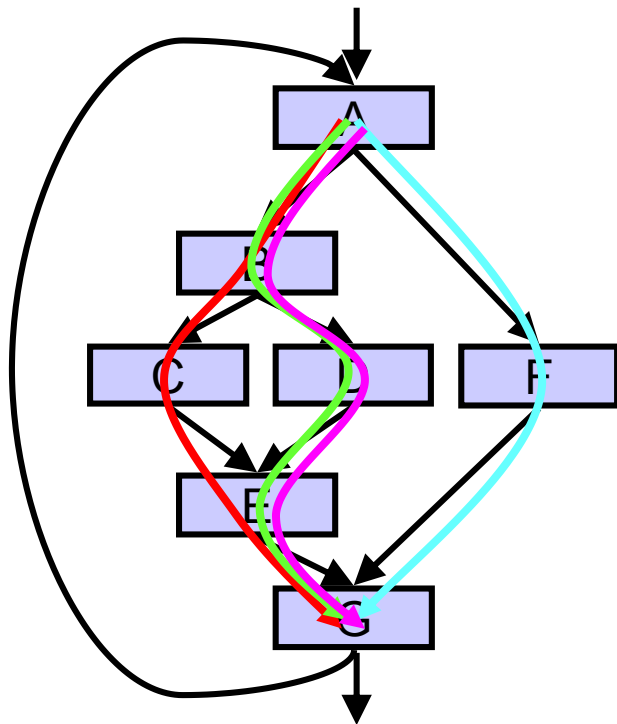
- Single procedure/program, multiple data
 - This is a programming model rather than computer organization
- Each processing element executes the same procedure, except on different data elements
 - Procedures can synchronize at certain points in program, e.g. barriers
- Essentially, multiple instruction streams execute the same program
 - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
 - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
 - Modern GPUs programmed in a similar way on a SIMD hardware

SIMD vs. SIMT Execution Model

- SIMD: A single **sequential instruction stream** of **SIMD instructions** → each instruction specifies multiple data inputs
 - [VLD, VLD, VADD, VST], VLEN
- SIMT: **Multiple instruction streams** of **scalar instructions** → threads grouped dynamically into warps
 - [LD, LD, ADD, ST], NumThreads
- Two Major SIMT Advantages:
 - **Can treat each thread separately** → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
 - **Can group threads into warps flexibly** → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

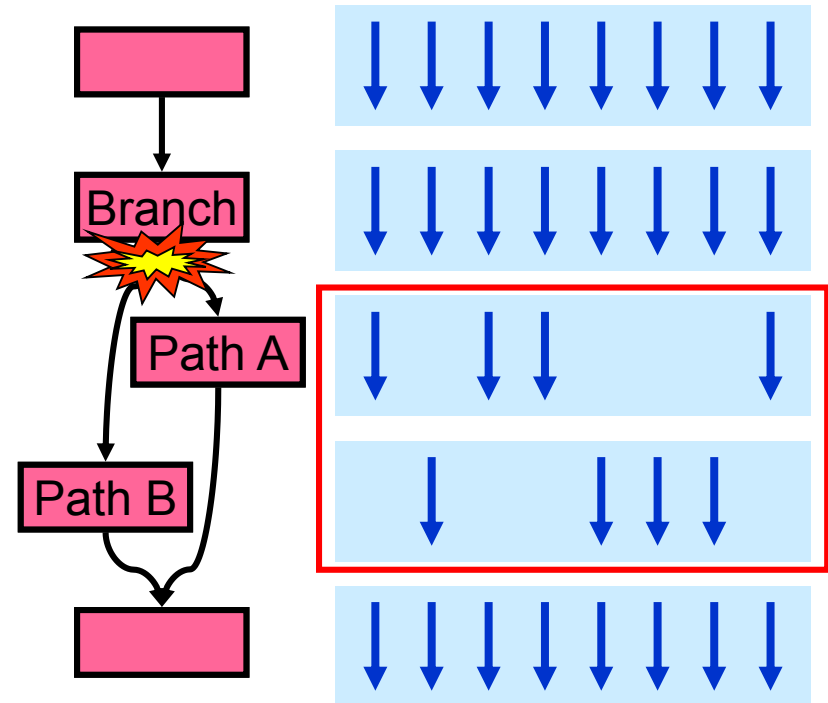
Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have conditional control flow instructions
- Threads can execute different control flow paths



Control Flow Problem in GPUs/SIMT

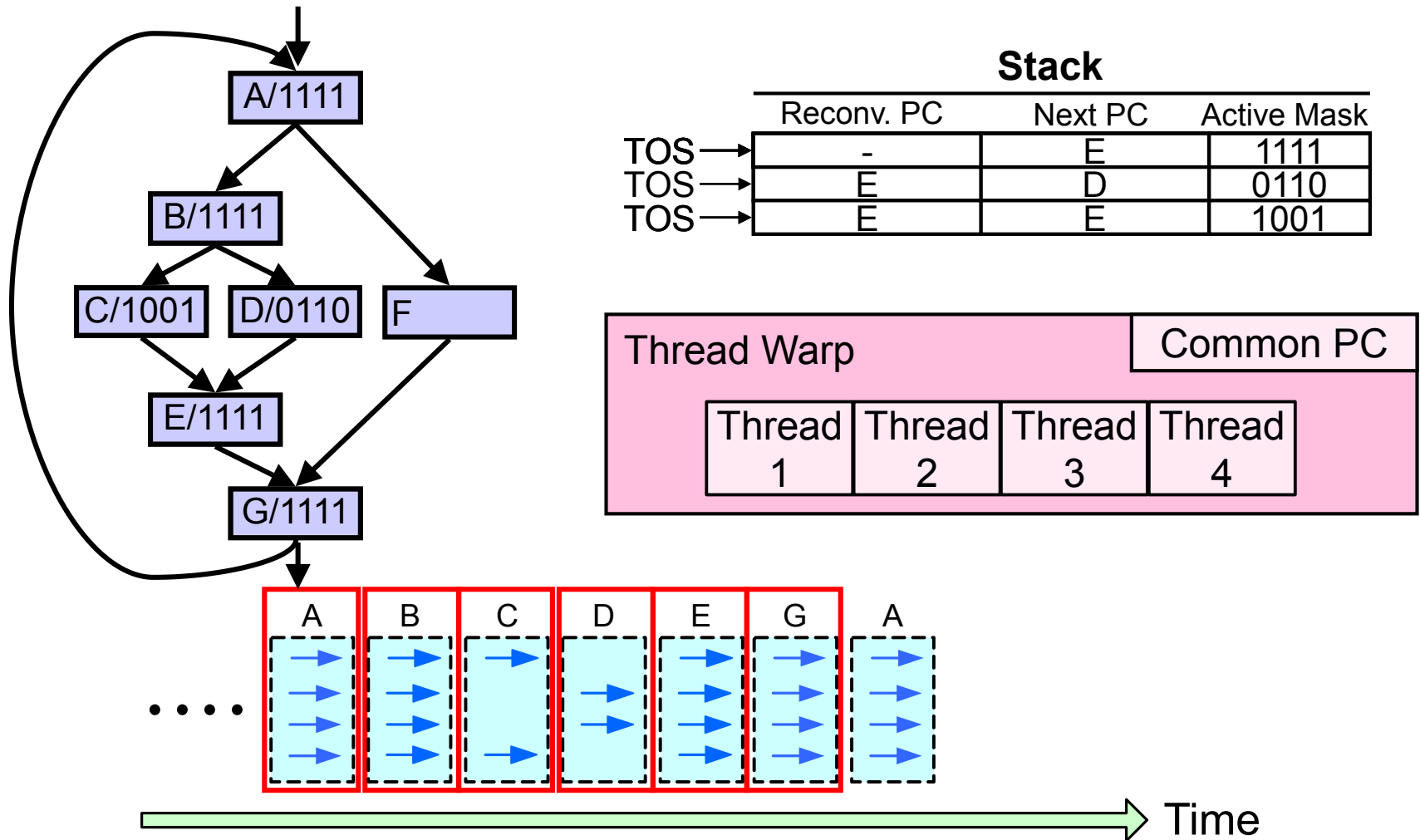
- A GPU uses a SIMD pipeline to save area on control logic.
 - Groups scalar threads into warps
- **Branch divergence** occurs when threads inside warps branch to different execution paths.



**This is the same as conditional execution.
Recall the Vector Mask and Masked Vector Operations?**

Branch Divergence Handling (I)

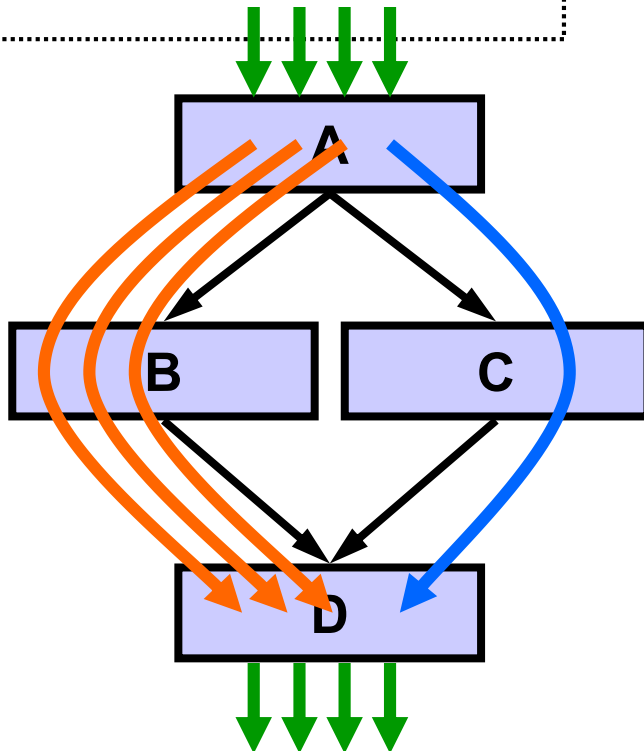
- Idea: Dynamic predicated (conditional) execution



Branch Divergence Handling (II)

```

A;
if (some condition) {
    B;
} else {
    C;
}
D;
    
```

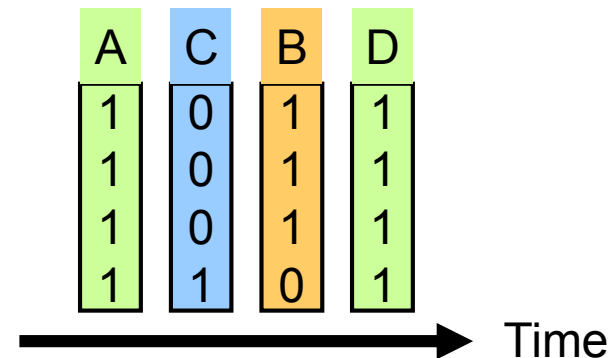


One per warp

Control Flow Stack

	Next PC	Recv PC	Active Mask
TOS →	D	--	1111
	B	D	1110
	D	D	0001

Execution Sequence

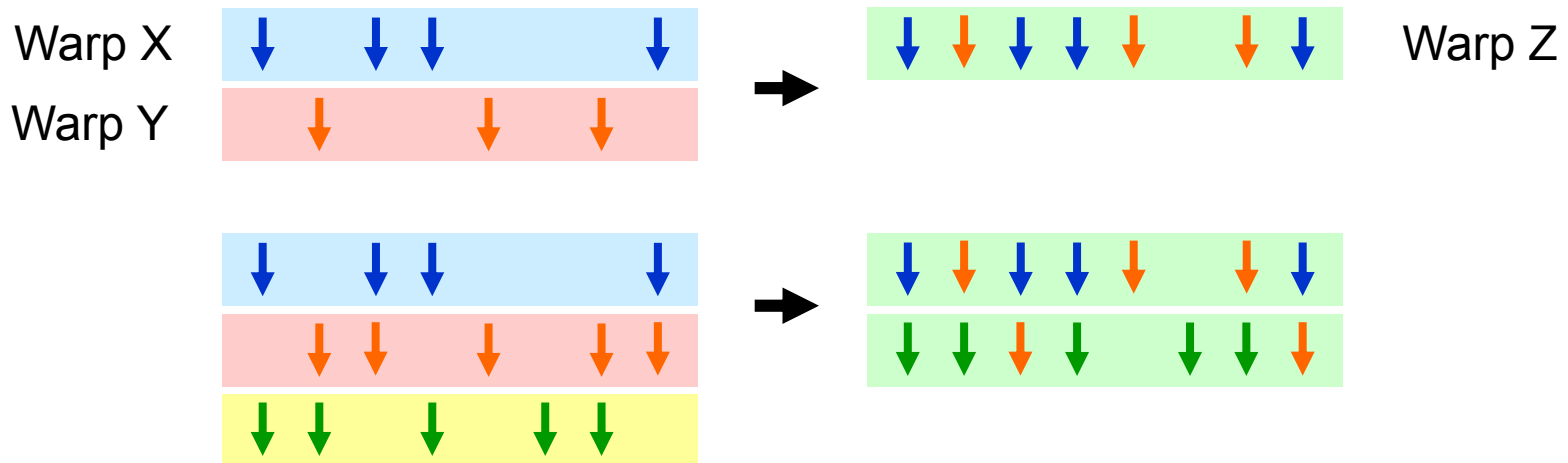


Remember: Each Thread Is Independent

- Two Major SIMT Advantages:
 - Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
 - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing
- If we have many threads
- We can find individual threads that are at the same PC
- And, group them together into a single warp dynamically
- This reduces “divergence” → improves SIMD utilization
 - SIMD utilization: fraction of SIMD lanes executing a useful operation (i.e., executing an active thread)

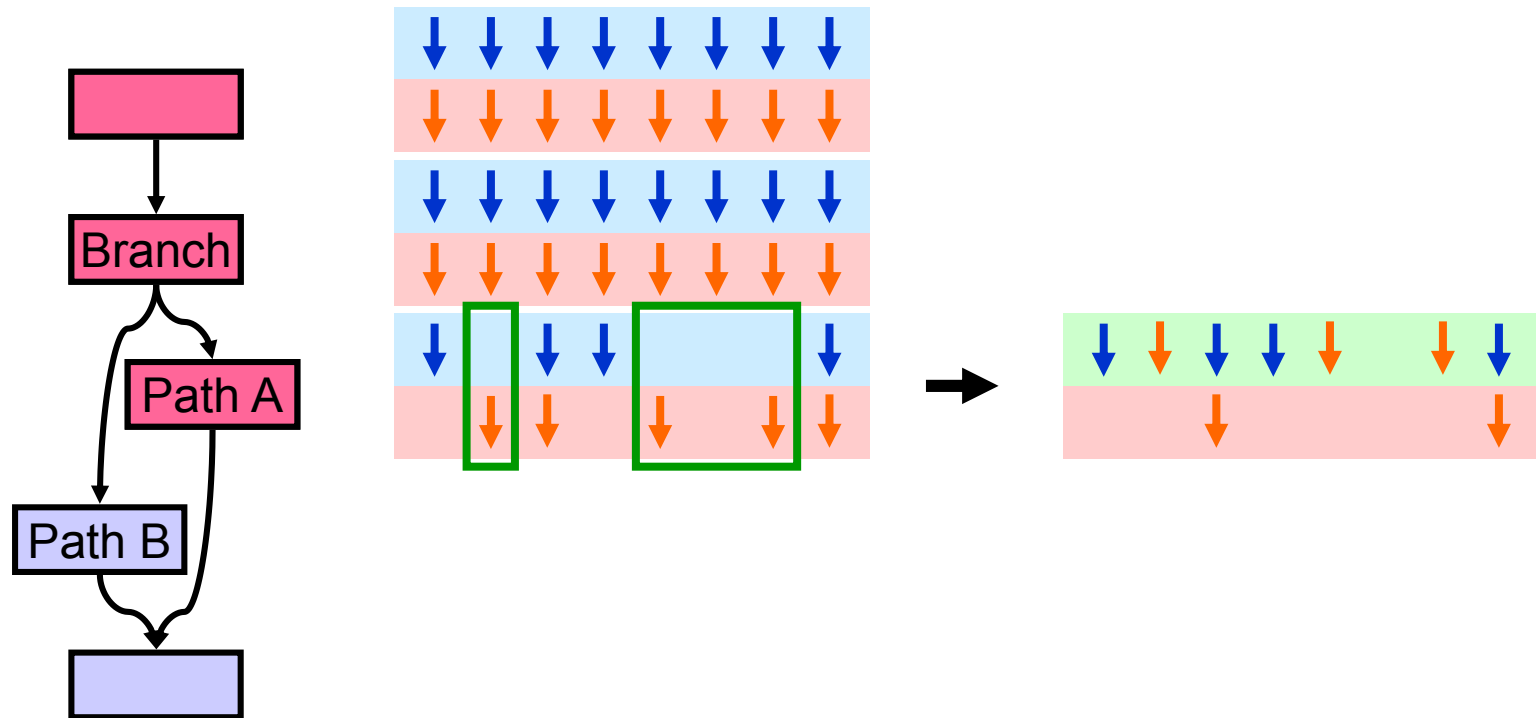
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
- Form new warps from warps that are waiting
 - Enough threads branching to each path enables the creation of full new warps



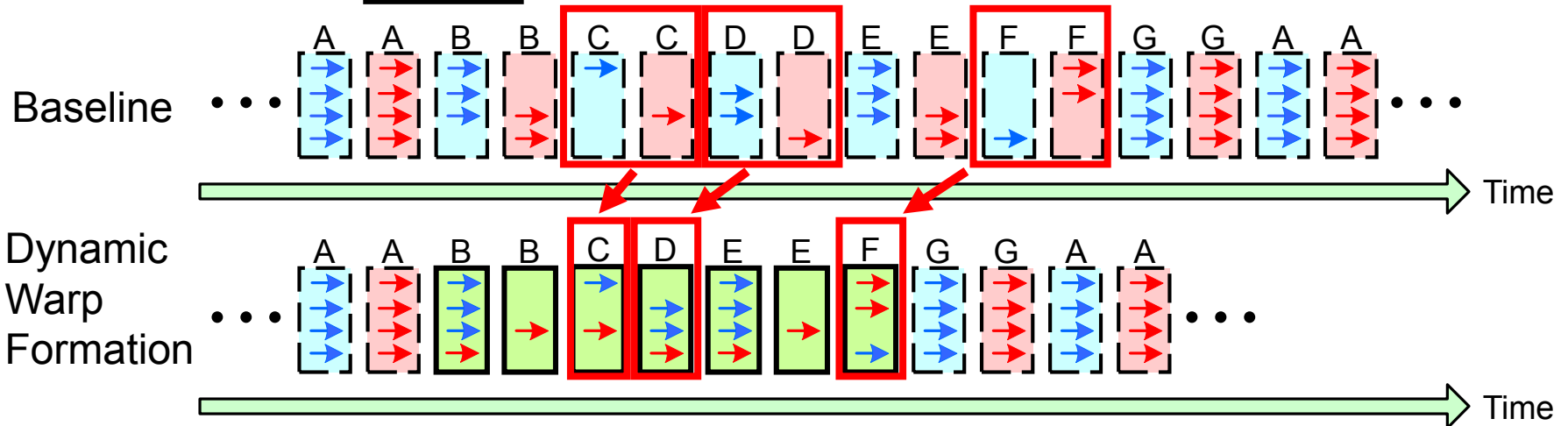
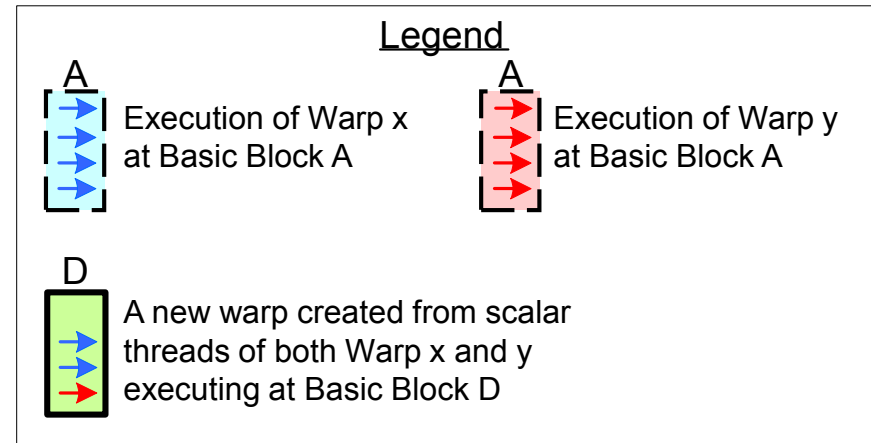
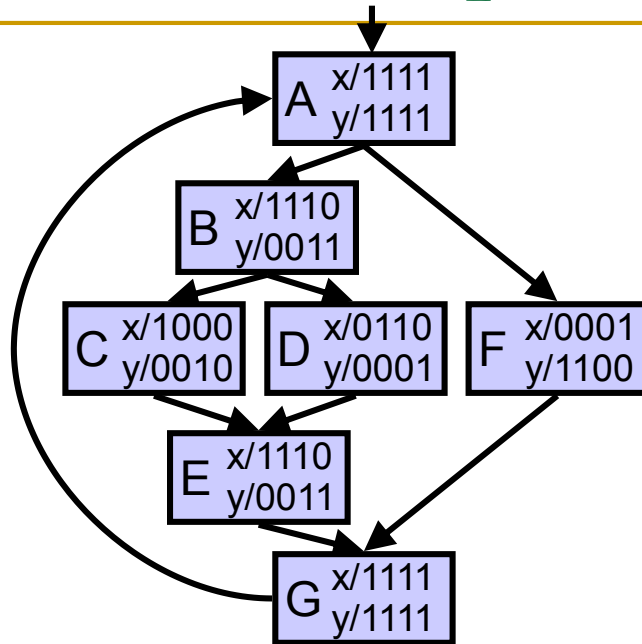
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)

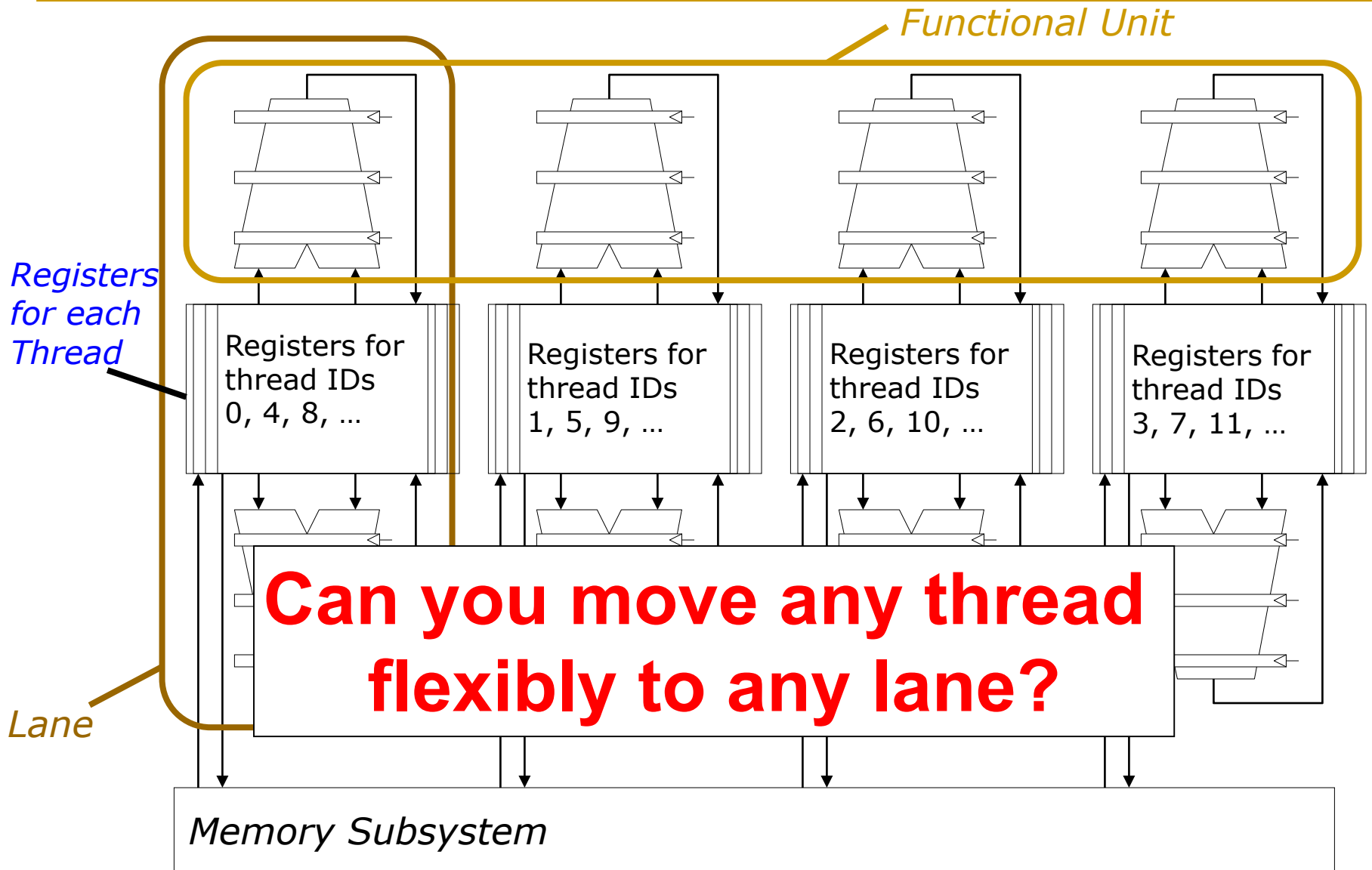


- Fung et al., “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” MICRO 2007.

Dynamic Warp Formation Example



Hardware Constraints Limit Flexibility of Warp Grouping



When You Group Threads Dynamically ...

- What happens to memory accesses?
- Simple, strided (predictable) memory access patterns within a warp can become complex, randomized (unpredictable) with dynamic regrouping of threads
 - Can reduce locality in memory
 - Can lead to inefficient bandwidth utilization

What About Memory Divergence?

- Modern GPUs have caches
 - To minimize accesses to main memory (save bandwidth)
- Ideally: Want all threads in the warp to hit (without conflicting with each other)
- Problem: Some threads in the warp may hit others may miss
- Problem: One thread in a warp can stall the entire warp if it misses in the cache.

- Need techniques to
 - Tolerate memory divergence
 - Integrate solutions to branch and memory divergence

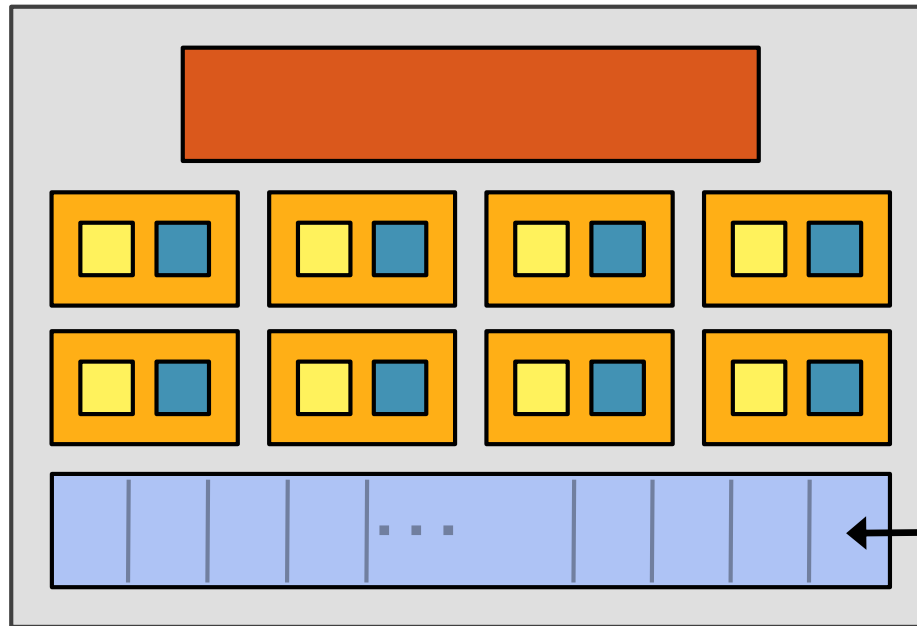
An Example GPU

NVIDIA GeForce GTX 285

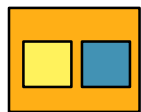
- NVIDIA-speak:
 - ❑ 240 stream processors
 - ❑ “SIMT execution”
- Generic speak:
 - ❑ 30 cores
 - ❑ 8 SIMD functional units per core



NVIDIA GeForce GTX 285 “core”



64 KB of storage
for thread contexts
(registers)



= SIMD functional unit, control
shared across 8 units

■ = multiply-add
■ = multiply

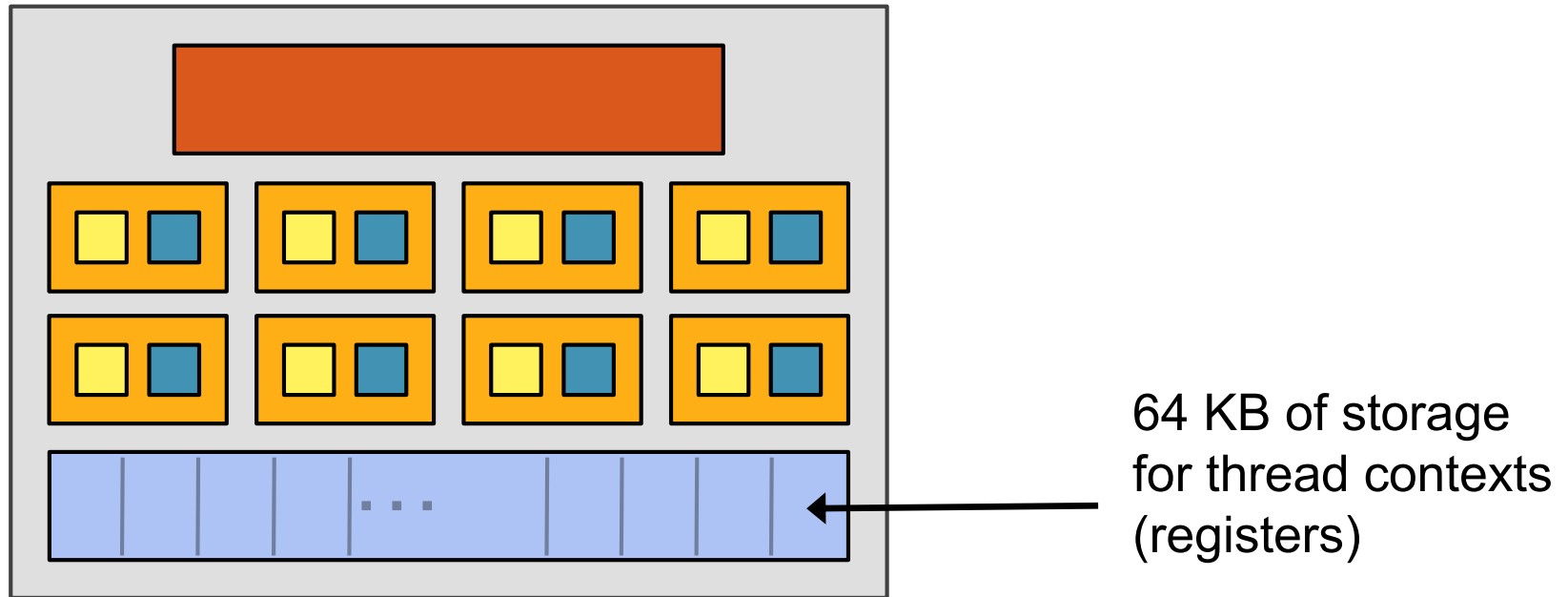


= instruction stream decode



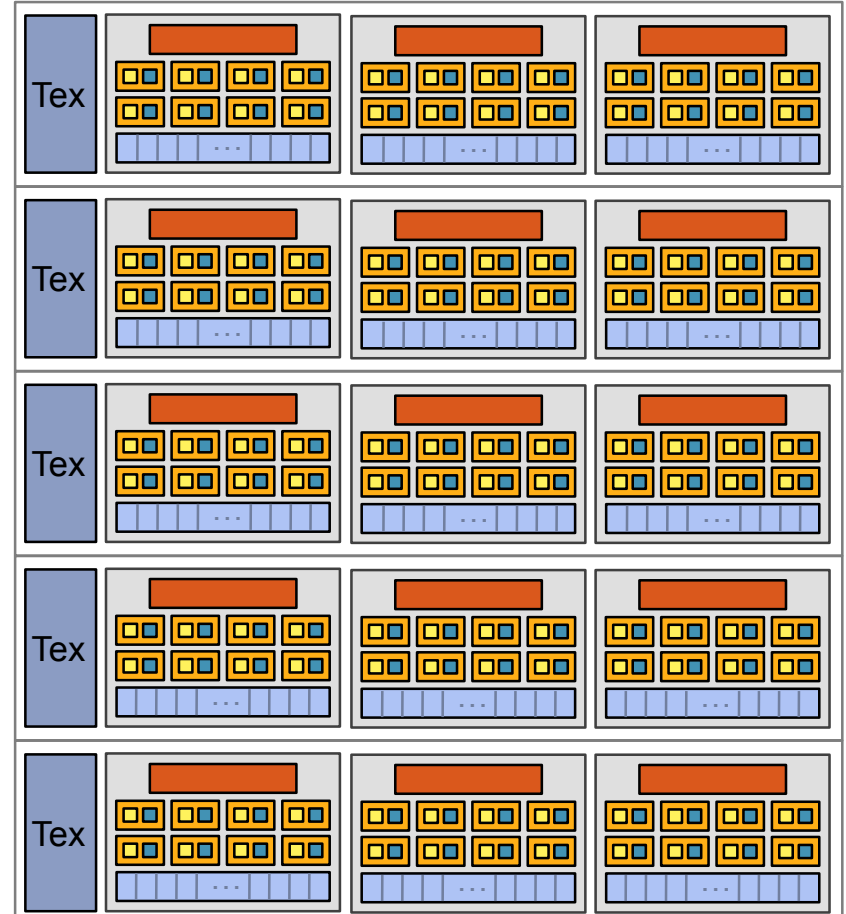
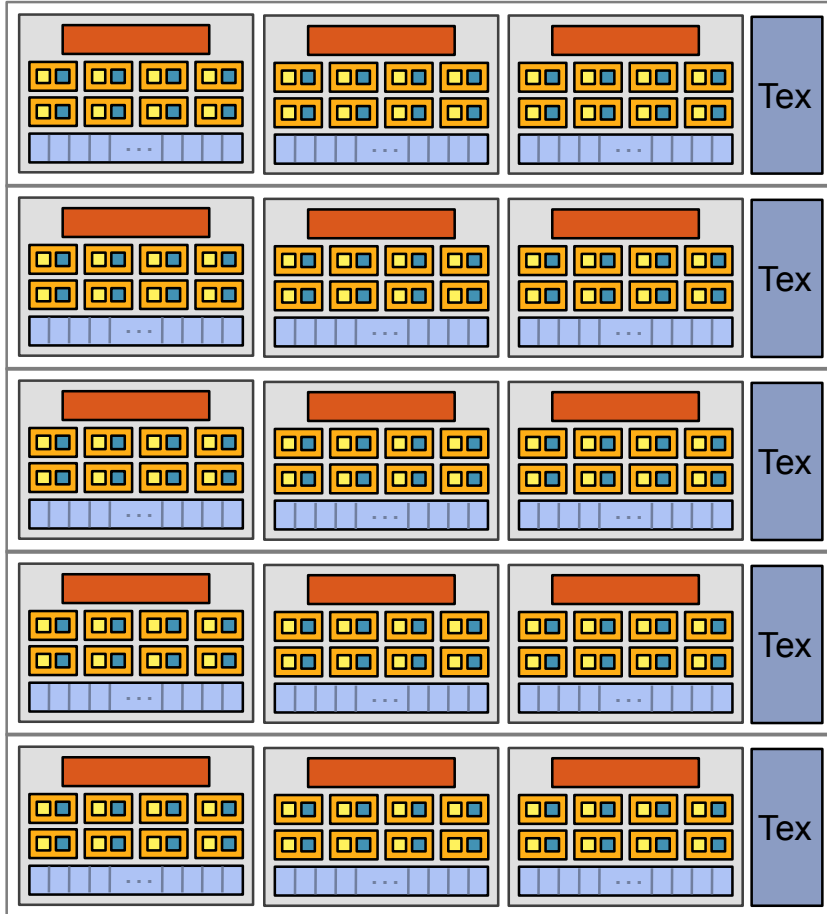
= execution context storage

NVIDIA GeForce GTX 285 “core”



- Groups of 32 **threads** share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

NVIDIA GeForce GTX 285



30 cores on the GTX 285: 30,720 threads

GPU Readings

■ Required

- Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.
- Fatahalian and Houston, "A Closer Look at GPUs," CACM 2008.

■ Recommended

- Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.
- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.
- Jog et al., "Orchestrated Scheduling and Prefetching for GPGPUs," ISCA 2013.

VLIW and DAE

Remember: SIMD/MIMD Classification of Computers

- Mike Flynn, “[Very High Speed Computing Systems](#),” Proc. of the IEEE, 1966
- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
 - Array processor
 - Vector processor
- **MISD**? Multiple instructions operate on single data element
 - Closest form: systolic array processor?
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
 - Multiprocessor
 - Multithreaded processor

SISD Parallelism Extraction Techniques

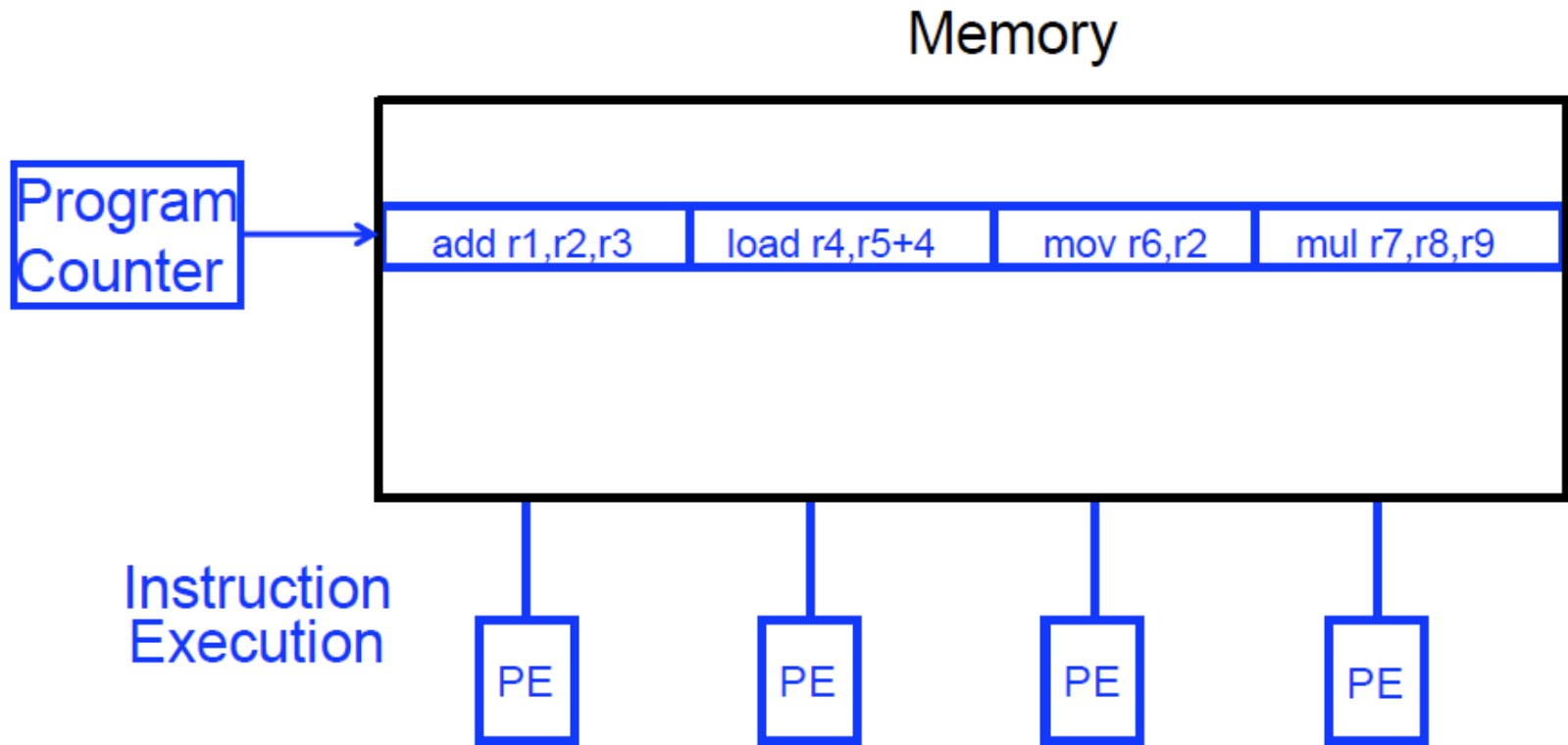
- We have already seen
 - Superscalar execution
 - Out-of-order execution
- Are there simpler ways of extracting SISD parallelism?
 - VLIW (Very Long Instruction Word)
 - Decoupled Access/Execute

VLIW

VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
 - Packed instructions can be logically unrelated (contrast with SIMD)
- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction
- Traditional Characteristics
 - Multiple functional units
 - Each instruction in a bundle executed in lock step
 - Instructions in a bundle statically aligned to be directly fed into the functional units

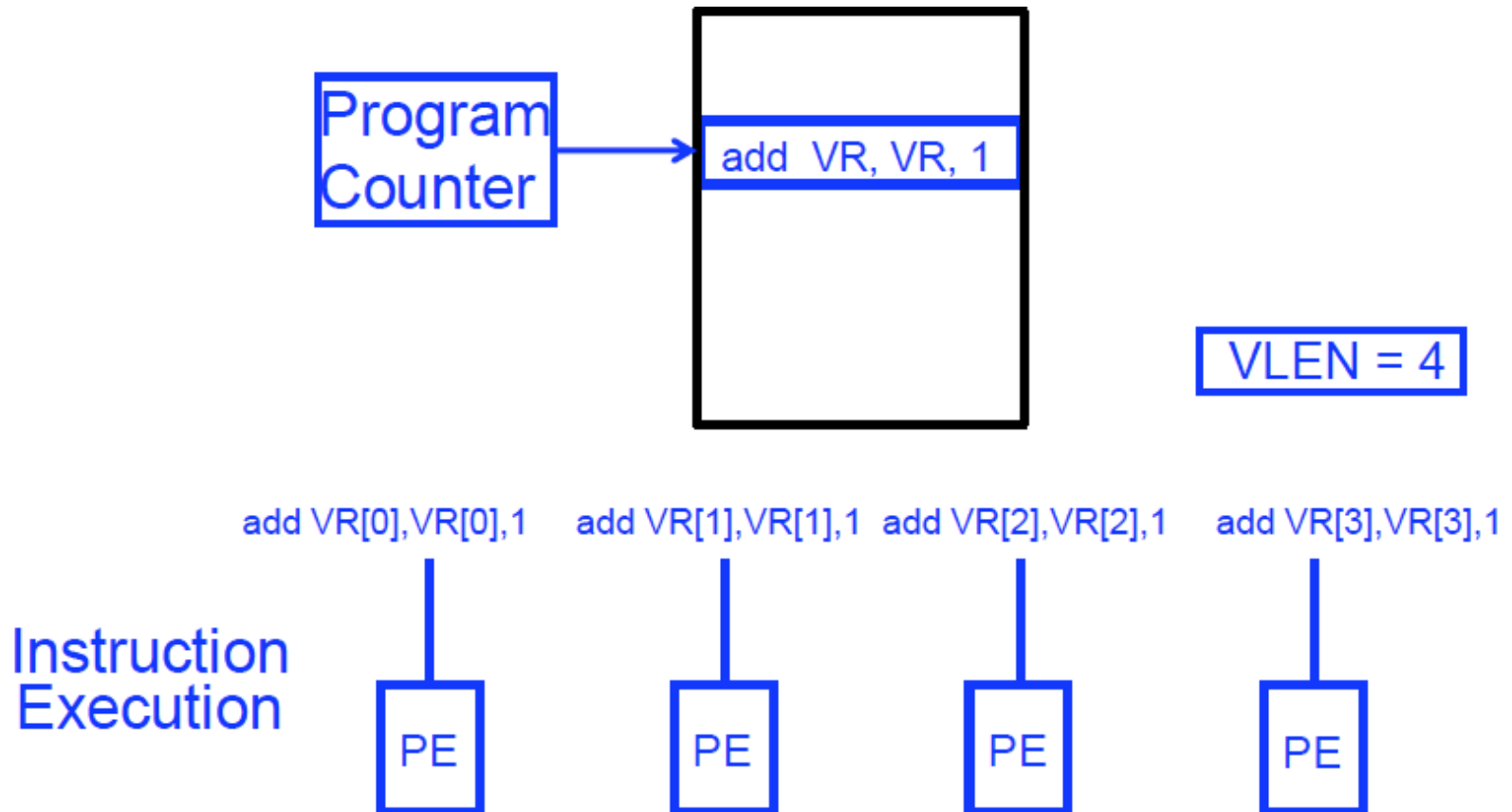
VLIW Concept



- Fisher, “**Very Long Instruction Word architectures and the ELI-512**,” ISCA 1983.
 - ELI: Enormously longword instructions (512 bits)

SIMD Array Processing vs. VLIW

- Array processor



VLIW Philosophy

- Philosophy similar to RISC (simple instructions and hardware)
 - Except multiple instructions in parallel

- RISC (John Cocke, 1970s, IBM 801 minicomputer)
 - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
 - And, to reorder simple instructions for high performance
 - Hardware does little translation/decoding → very simple

- VLIW (Fisher, ISCA 1983)
 - Compiler does the hard work to find instruction level parallelism
 - Hardware stays as simple and streamlined as possible
 - Executes each instruction in a bundle in lock step
 - Simple → higher frequency, easier to design

VLIW Philosophy and Properties

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.

Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors)
 - Most successful commercially
- Intel IA-64
 - Not fully VLIW, but based on VLIW principles
 - EPIC (Explicitly Parallel Instruction Computing)
 - Instruction bundles can have dependent instructions
 - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones

VLIW Tradeoffs

■ Advantages

- + No need for dynamic scheduling hardware → simple hardware
- + No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming
- + No need for instruction alignment/distribution after fetch to different functional units → simple hardware

■ Disadvantages

- Compiler needs to find N independent operations per cycle
 - If it cannot, inserts NOPs in a VLIW instruction
 - Parallelism loss AND code size increase
- Recompile required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
- Lockstep execution causes independent operations to stall
 - No instruction can progress until the longest-latency instruction completes

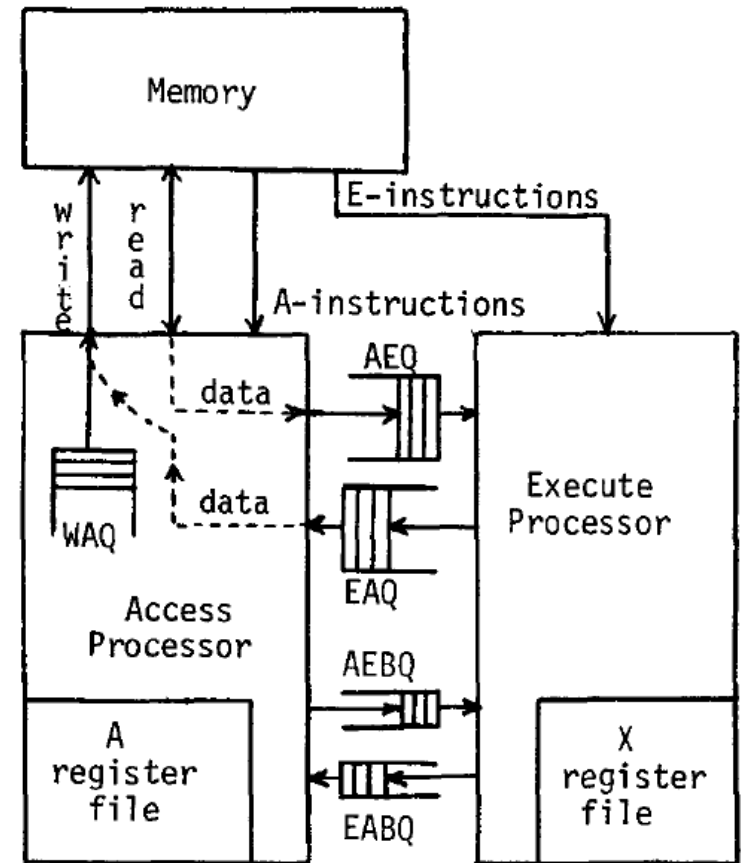
VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
 - Solely-compiler approach of VLIW has several downsides that reduce performance
 - Too many NOPs (not enough parallelism discovered)
 - Static schedule intimately tied to microarchitecture
 - Code optimized for one generation performs poorly for next
 - No tolerance for variable or long-latency operations (lock step)
- ++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
- Enable code optimizations
- ++ VLIW successful when parallelism is easier to find by the compiler (traditionally embedded markets, DSPs)

Decoupled Access/Execute (DAE)

Decoupled Access/Execute (DAE)

- Motivation: Tomasulo's algorithm too complex to implement
 - 1980s before Pentium Pro
- Idea: Decouple operand access and execution via **two separate instruction streams that communicate via ISA-visible queues**.
- Smith, “**Decoupled Access/Execute Computer Architectures**,” ISCA 1982, ACM TOCS 1984.



Decoupled Access/Execute (II)

- Compiler generates two instruction streams (A and E)
 - Synchronizes the two upon control flow instructions (using branch queues)

```

q = 0.0
Do 1 k = 1, 400
1  x(k) = q + y(k) * (r * z(k+10) + t * z(k+11))
    
```

Fig. 2a. Lawrence Livermore Loop 1 (HYDRO EXCERPT)

	A7 ← -400	. negative loop count
	A2 ← 0	. initialize index
	A3 ← 1	. index increment
	X2 ← r	. load loop invariants
	X5 ← t	. into registers
loop:	X3 ← z + 10, A2	. load z(k+10)
	X7 ← z + 11, A2	. load z(k+11)
	X4 ← X2 *f X3	. r*z(k+10)-flt. mult.
	X3 ← X5 *f X7	. t * z(k+11)
	X7 ← y, A2	. load y(k)
	X6 ← X3 +f X4	. r*z(x+10)+t*z(k+11))
	X4 ← X7 *f X6	. y(k) * (above)
	A7 ← A7 + 1	. increment loop counter
	x, A2 ← X4	. store into x(k)
	A2 ← A2 + A3	. increment index
	JAM loop	. Branch if A7 < 0

Fig. 2b. Compilation onto CRAY-1-like architecture

<u>Access</u>	<u>Execute</u>
.	
.	
.	
AEQ ← z + 10, A2	X4 ← X2 *f AEQ
AEQ ← z + 11, A2	X3 ← X5 *f AEQ
AEQ ← y, A2	X6 ← X3 +f X4
A7 ← A7 + 1	EAQ ← AEQ *f X6
x, A2 ← EAQ	.
A2 ← A2 + A3	.
.	.
.	.
.	.

Fig. 2c. Access and execute programs for straight-line section of loop

Decoupled Access/Execute (III)

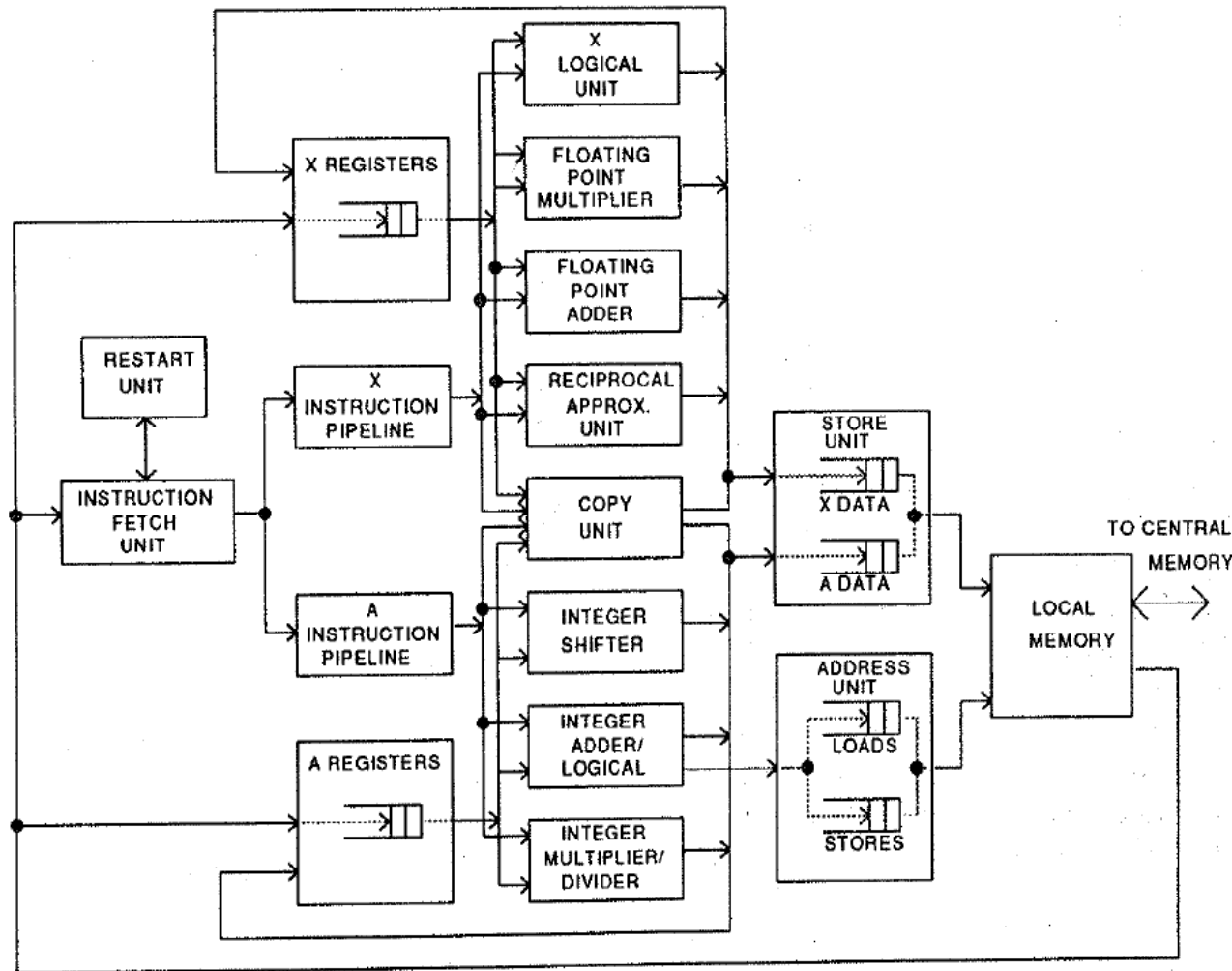
■ Advantages:

- + Execute stream can run ahead of the access stream and vice versa
 - + If A takes a cache miss, E can perform useful work
 - + If A hits in cache, it supplies data to lagging E
 - + Queues reduce the number of required registers
- + Limited out-of-order execution without wakeup/select complexity

■ Disadvantages:

- Compiler support to partition the program and manage queues
 - Determines the amount of decoupling
- Branch instructions require synchronization between A and E
- Multiple instruction streams (can be done with a single one, though)

Astronautics ZS-1



- Single stream steered into A and X pipelines
- Each pipeline in-order
- Smith et al., “The ZS-1 central processor,” ASPLOS 1987.
- Smith, “Dynamic Instruction Scheduling and the Astronautics ZS-1,” IEEE Computer 1989.

Astronautics ZS-1 Instruction Scheduling

■ Dynamic scheduling

- ❑ A and X streams are issued/executed independently
- ❑ Loads can bypass stores in the memory unit (if no conflict)
- ❑ Branches executed early in the pipeline
 - To reduce synchronization penalty of A/X streams
 - Works only if the register a branch sources is available

■ Static scheduling

- ❑ Move compare instructions as early as possible before a branch
 - So that branch source register is available when branch is decoded
- ❑ Reorder code to expose parallelism in each stream
- ❑ Loop unrolling:
 - Reduces branch count + exposes code reordering opportunities

Loop Unrolling

```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    i = i + 1
}
```

```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m

    a[i+1] = b[i+2] + (i+2)/m
    b[i+1] = a[i] - (i+1)/m
    i = i + 2
}
```

- Idea: Replicate loop body multiple times within an iteration
- + Reduces loop maintenance overhead
 - Induction variable increment or loop condition test
- + Enlarges basic block (and analysis scope)
 - Enables code optimization and scheduling opportunities
- What if iteration count not a multiple of unroll factor? (need extra code to detect this)
- Increases code size