课程编号180086085404P2008H   2023-2024学年秋季学期

计 算 机 体 系 结 构

第 13 讲 向量处理器 和 GPU 体系结构 (Vector Processor and GPU)

主讲教师：刘珂

2024年12月3日

University of Chinese Academy of Sciences

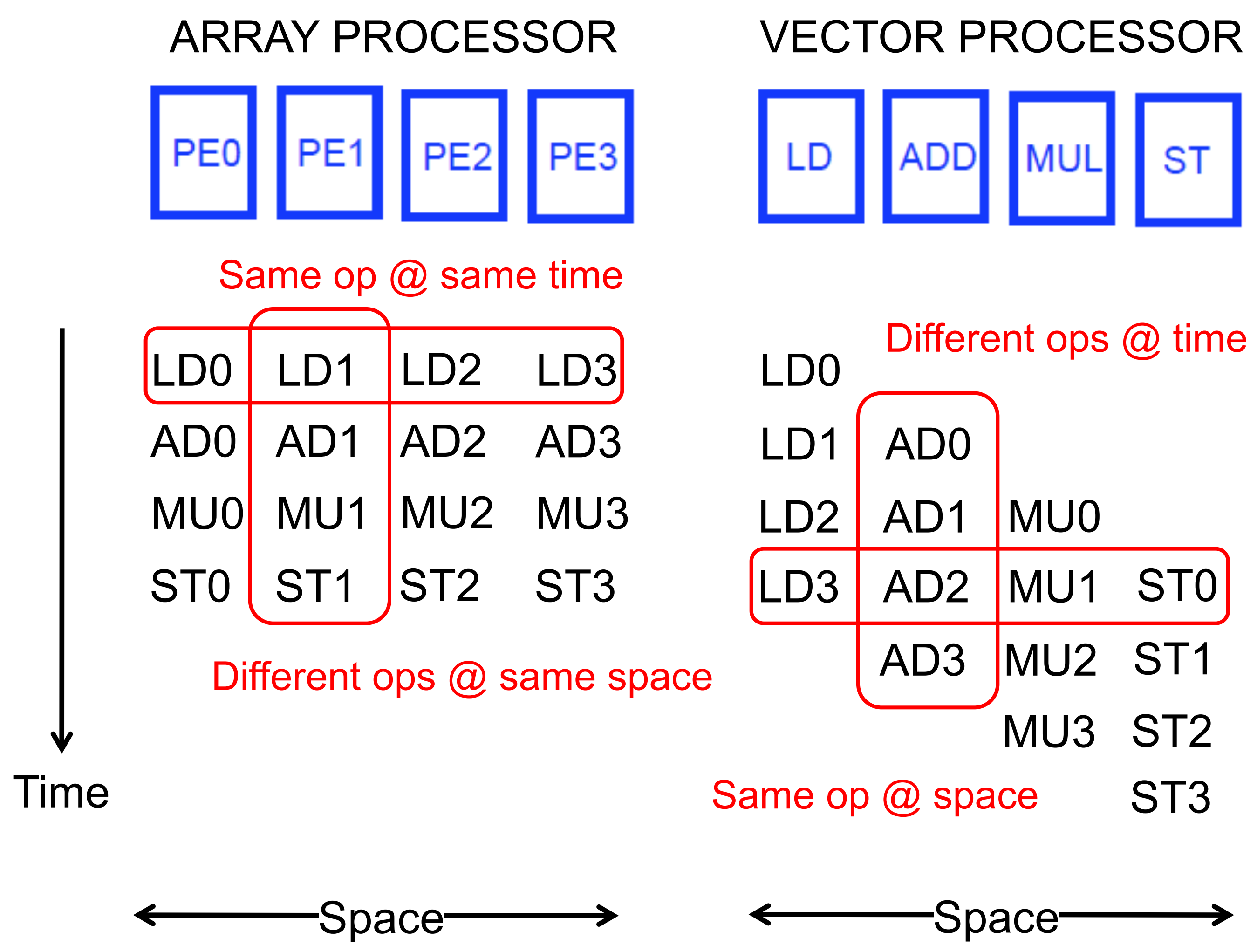
INSTITUTE OF COMPUTING TECHNOLOGY,CAS

Vector Processing in More Depth

Recall: Array vs. Vector Processors

Instruction Stream

LD	VR ← A[3:0]
ADD	VR ← VR, 1
MUL	VR ← VR, 2
ST	A[3:0] ← VR



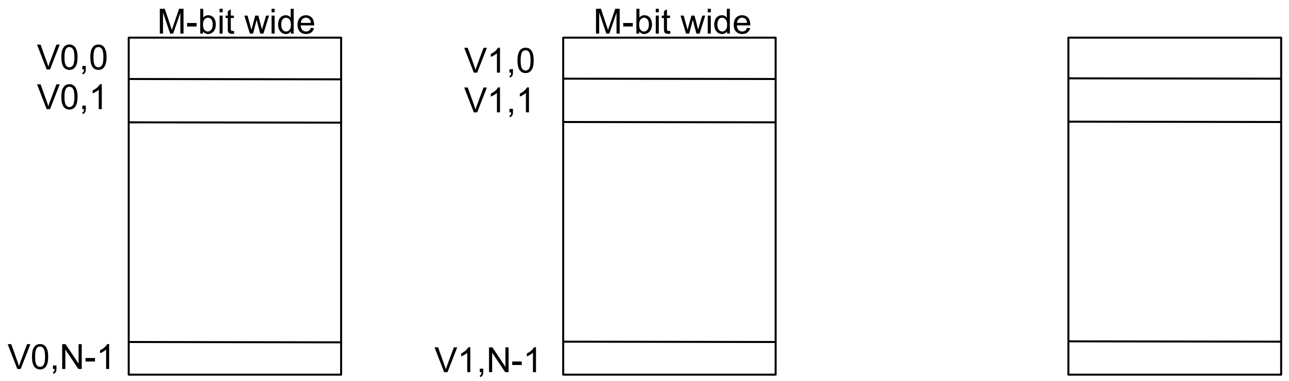
Recall: Array vs. Vector Processors

- Array vs. vector processor distinction
  - Time parallelism vs. Space parallelism

- Most "modern" SIMD processors are a combination of both
  - They exploit data parallelism in both time and space
  - GPUs are a prime example we will cover in more detail

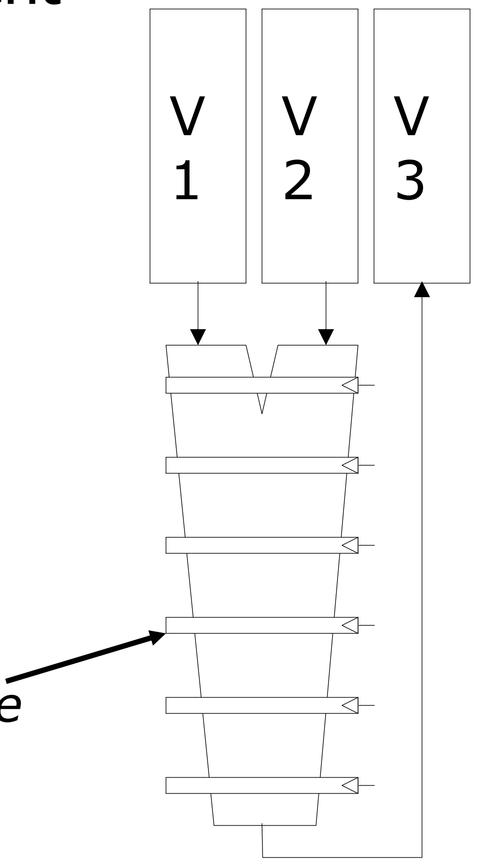
Vector Registers

- Each vector data register holds N M-bit values
- Vector control registers: VLEN, VSTR, VMASK
- Maximum VLEN can be N
  - Maximum number of elements stored in a vector register
- Vector Mask Register (VMASK)
  - Indicates which elements of vector to operate on
  - Set by vector test instructions
    - e.g., VMASK[i] = (Vk[i] == 0)



Vector Functional Units

- Use a deep pipeline to execute element operations
  - → fast clock cycle
- Control of deep pipeline is simple because elements in vector are independent



V1 * V2 → V3

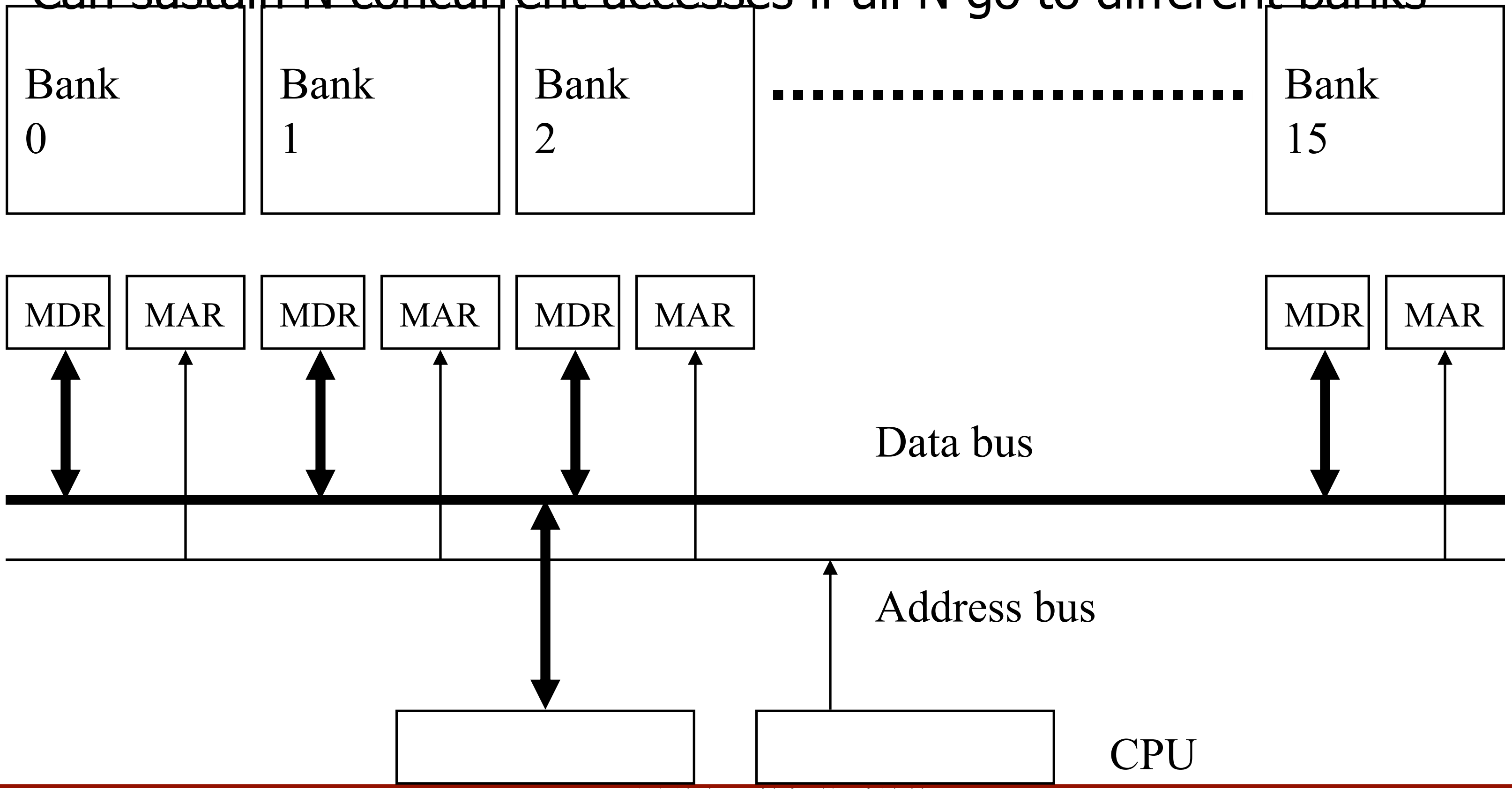
Loading/Storing Vectors from/to Memory

- Requires loading/storing multiple elements

- Elements separated from each other by a constant distance (stride)
  - Assume stride = 1 for now
- Elements can be loaded in consecutive cycles if we can start the load of one element per cycle
  - Can sustain a throughput of one element per cycle
- Question: How do we achieve this with a memory that takes more than 1 cycle to access?
- Answer: Bank the memory; interleave the elements across banks

Memory Banking

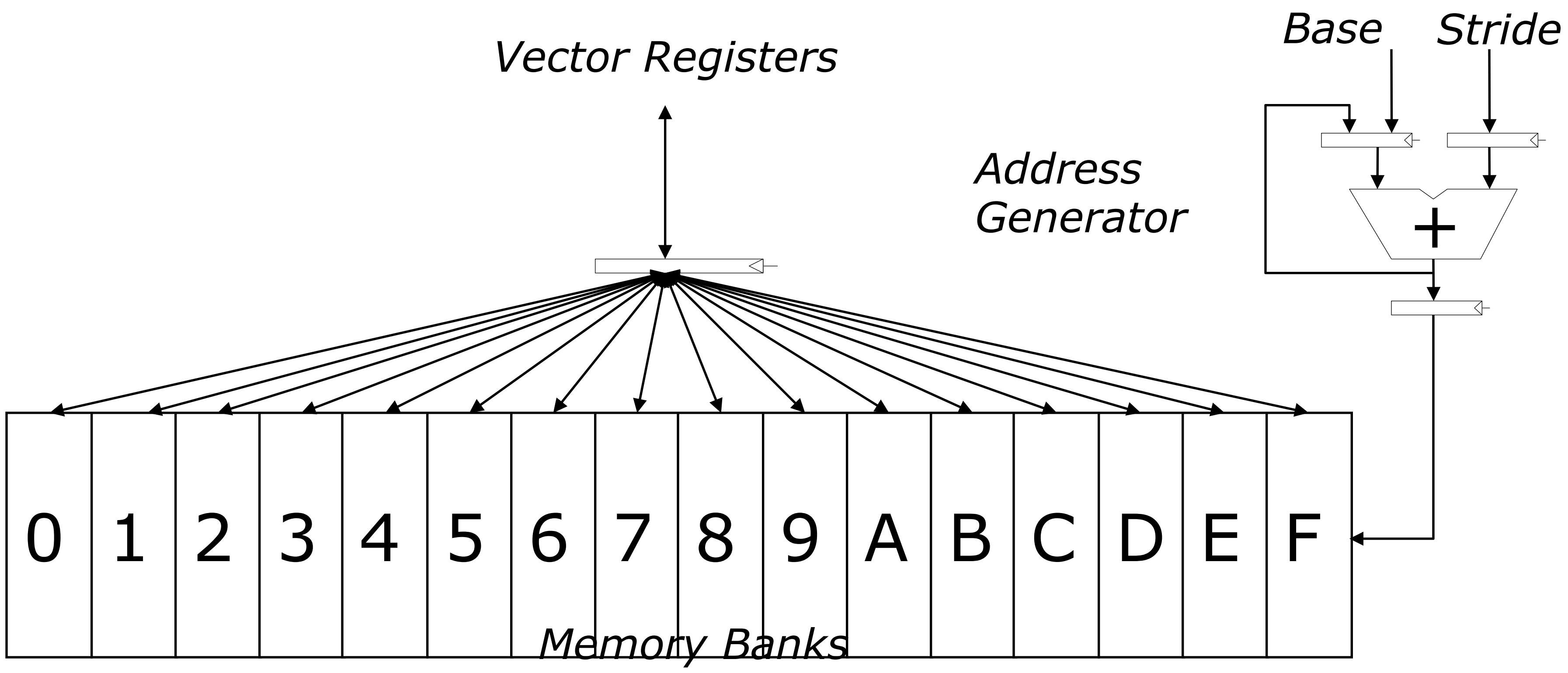
- Memory is divided into banks (存储体) that can be accessed independently; banks share address and data buses (to reduce memory chip pins)
- Can start and complete one bank access per cycle
- Can sustain N concurrent accesses if all N go to different banks



Picture credit: Derek Chiou

Vector Memory System

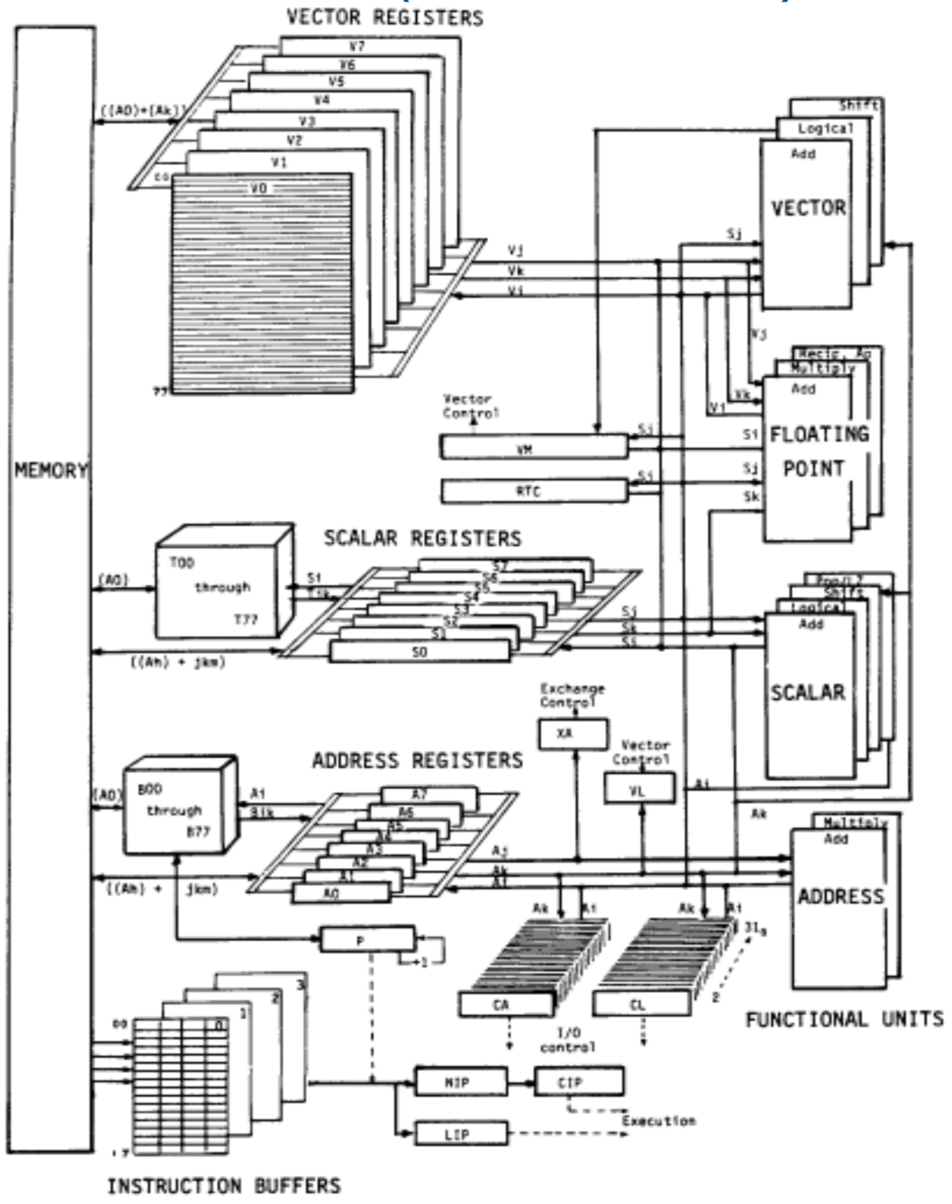
- Next address = Previous address + Stride
- If (stride == 1) && (consecutive elements interleaved across banks) && (number of banks >= bank latency), then
  - we can sustain 1 element/cycle throughput



Picture credit: Krste Asanovic

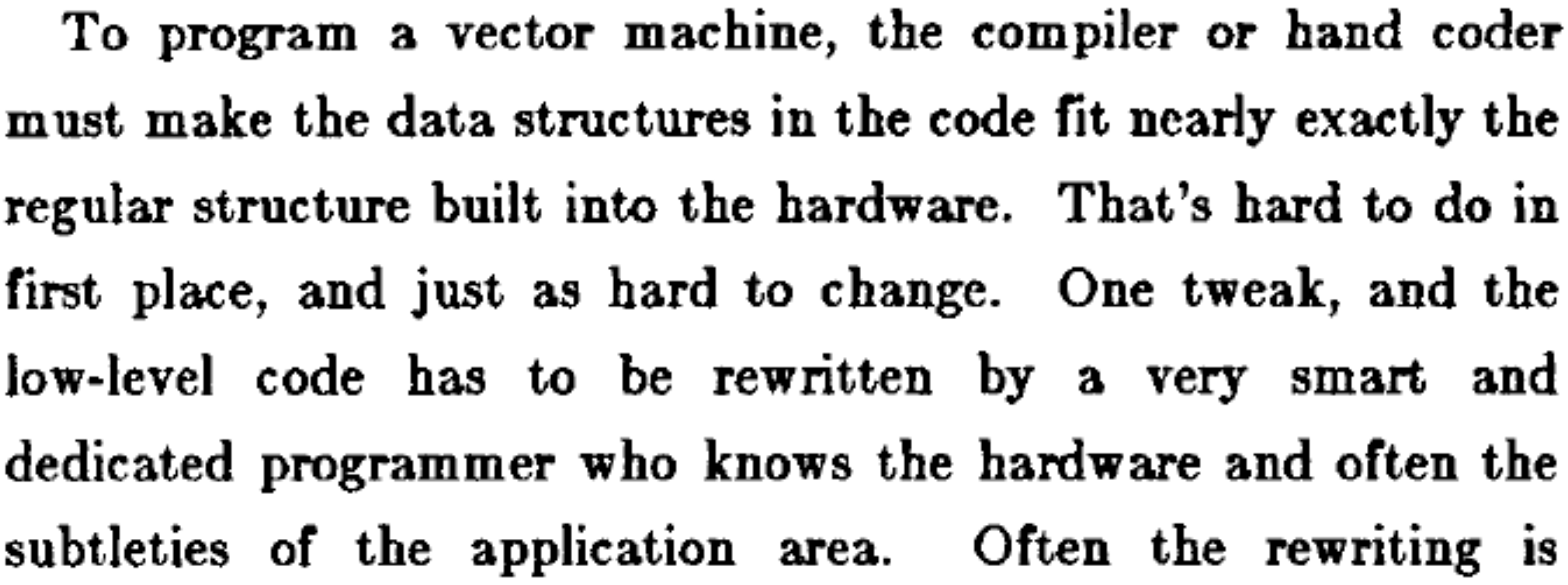
Vector Machine Organization (CRAY-1)

- CRAY-1
- Russell, "The CRAY-1 computer system," CACM 1978.
  - Scalar and vector modes
  - 8 64-element vector registers
  - 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers



Recall: Vector Processor Disadvantages

- -- Works (only) if parallelism is regular (data/SIMD parallelism)
  - ++ Vector operations
  - -- Very inefficient if parallelism is irregular
    - How about searching for a key in a linked list?



To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.

Scalar Code Example: Element-Wise Avg.

- For I = 0 to 49
  - C[i] = (A[i] + B[i]) / 2
- Scalar code (instruction and its latency in clock cycles)

```
       MOVI R0 = 50             1
       MOVA R1 = A              1
       MOVA R2 = B              1
       MOVA R3 = C              1
X:     LD R4 = MEM[R1++]        11  ;autoincrement addressing
       LD R5 = MEM[R2++]        11
       ADD R6 = R4 + R5         4
       SHFR R7 = R6 >> 1        1
       ST MEM[R3++] = R7        11
       DECBNZ R0, X             2   ;decrement and branch if NZ
```

304 dynamic instructions

Scalar Code Execution Time (In Order)

- Scalar execution time on an in-order processor with 1 bank
  - First two loads in the loop cannot be pipelined: 2*11 cycles
  - 4 + 50*40 = 2004 cycles

- Scalar execution time on an in-order processor with 1 bank with 2 memory ports (two different memory accesses can be serviced concurrently) OR 2 banks (where arrays A and B are stored in different banks)
  - First two loads in the loop can be pipelined: 1 + 11 cycles
  - 4 + 50*30 = 1504 cycles

Vectorizable Loops

- A loop is vectorizable if each iteration is independent of any other

- for I = 0 to 49

  C[i] = (A[i] + B[i]) / 2

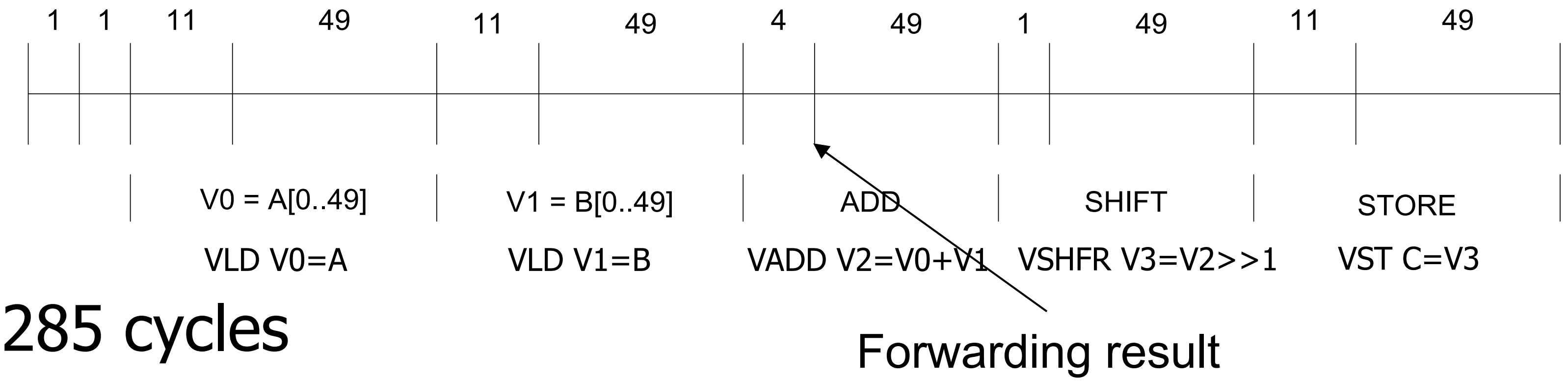
7 dynamic instructions

- Vectorized loop (each instruction and its latency):

MOVI VLEN = 50	1
MOVI VSTR = 1	1
VLD V0 = A	11 + VLEN – 1
VLD V1 = B	11 + VLEN – 1
VADD V2 = V0 + V1	4 + VLEN – 1
VSHFR V3 = V2 >> 1	1 + VLEN – 1
VST C = V3	11 + VLEN – 1

Basic Vector Code Performance

- Assume no chaining (no vector data forwarding)
  - i.e., output of a vector functional unit cannot be used as the direct input of another
  - The entire vector register needs to be ready before any element of it can be used as part of another operation
- 1 memory port (one address generator) per bank
- 16 memory banks (word-interleaved: consecutive elements of an array are stored in consecutive banks)



- 285 cycles

Basic Vector Code Performance (II)

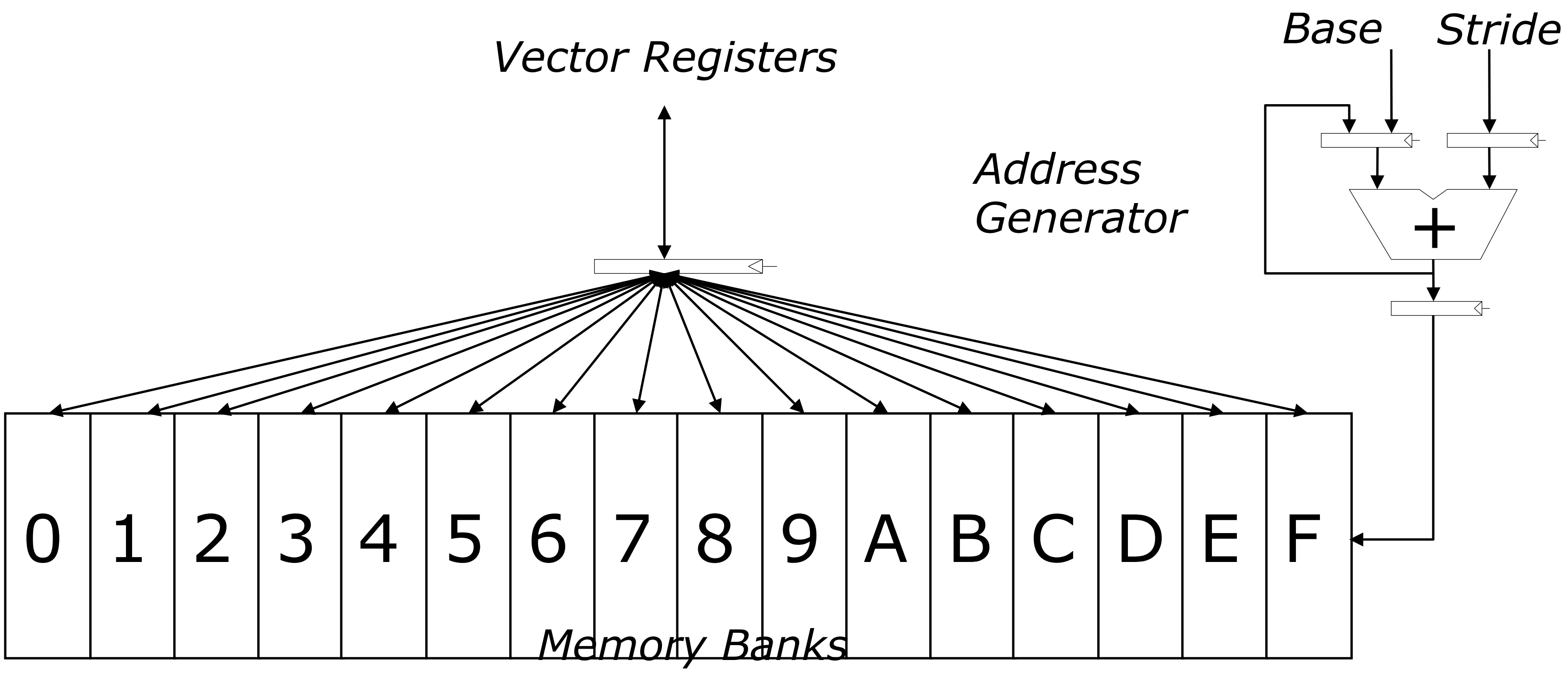
- Why 16 banks?
  - 11-cycle memory access latency
  - Having 16 (>11) banks ensures there are enough banks to overlap enough memory operations to cover memory latency

- The above assumes a unit stride (i.e., stride = 1)
  - Correct for our example program

- What if stride > 1?
  - How do you ensure we can access 1 element per cycle when memory latency is 11 cycles?

Recall: Vector Memory System

- Next address = Previous address + Stride
- If (stride == 1) && (consecutive elements interleaved across banks) && (number of banks >= bank latency), then
  - we can sustain 1 element/cycle throughput

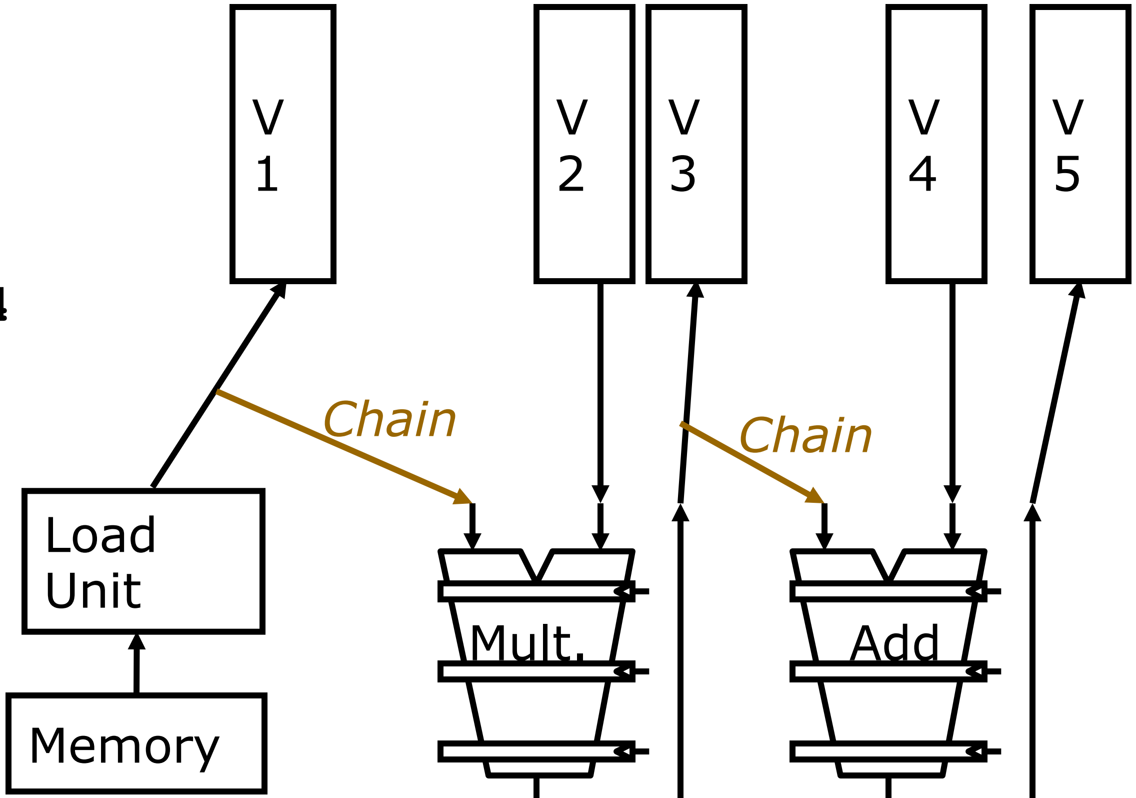


Picture credit: Krste Asanovic

Vector Chaining

- Vector chaining: Data forwarding from one vector functional unit to another

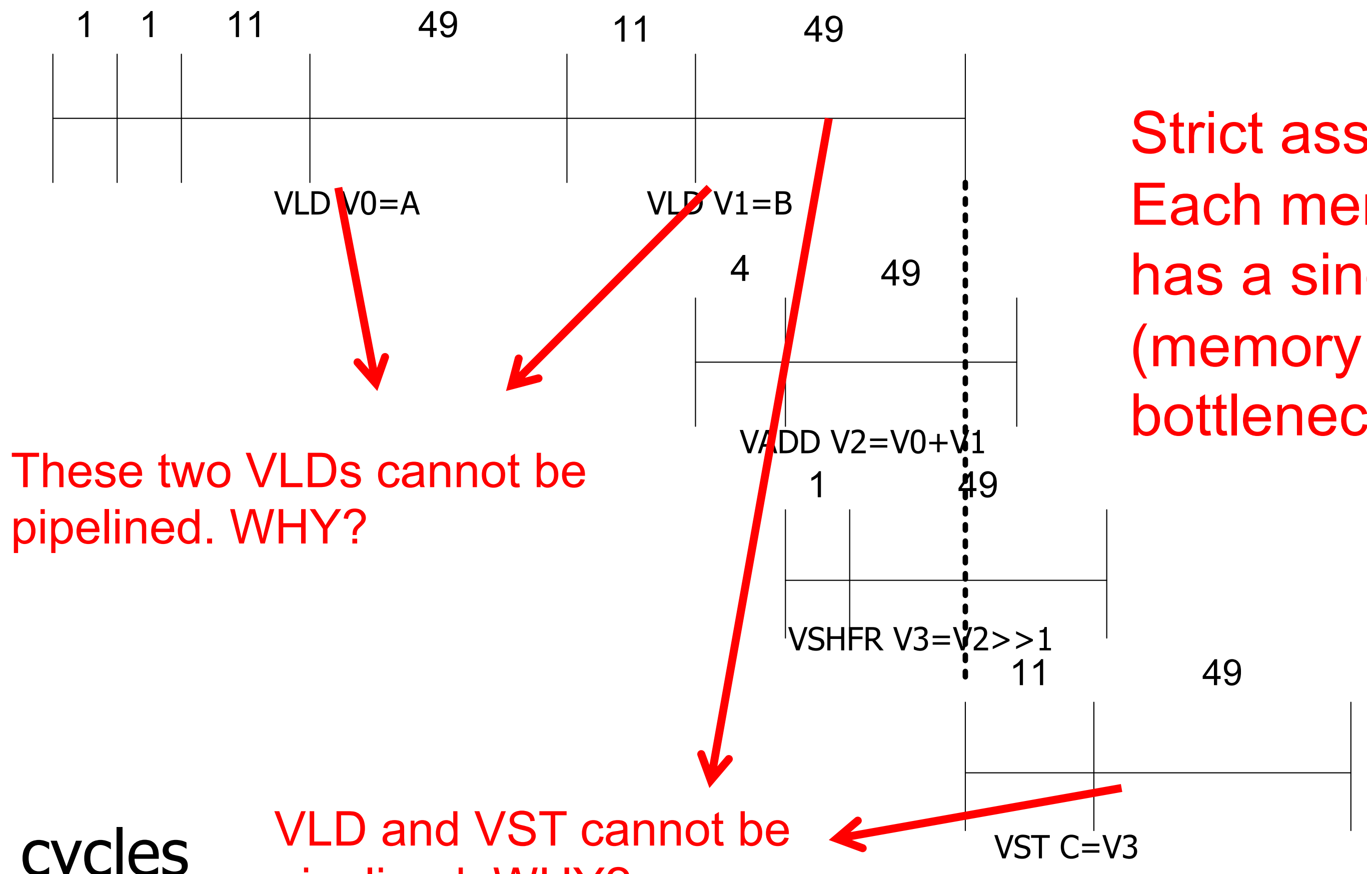
```
LV    v1
MULV  v3,v1,v2
ADDV  v5, v3, v4
```



Slide credit: Krste Asanovic

Vector Code Performance - Chaining

- Vector chaining: Data forwarding from one vector functional unit to another

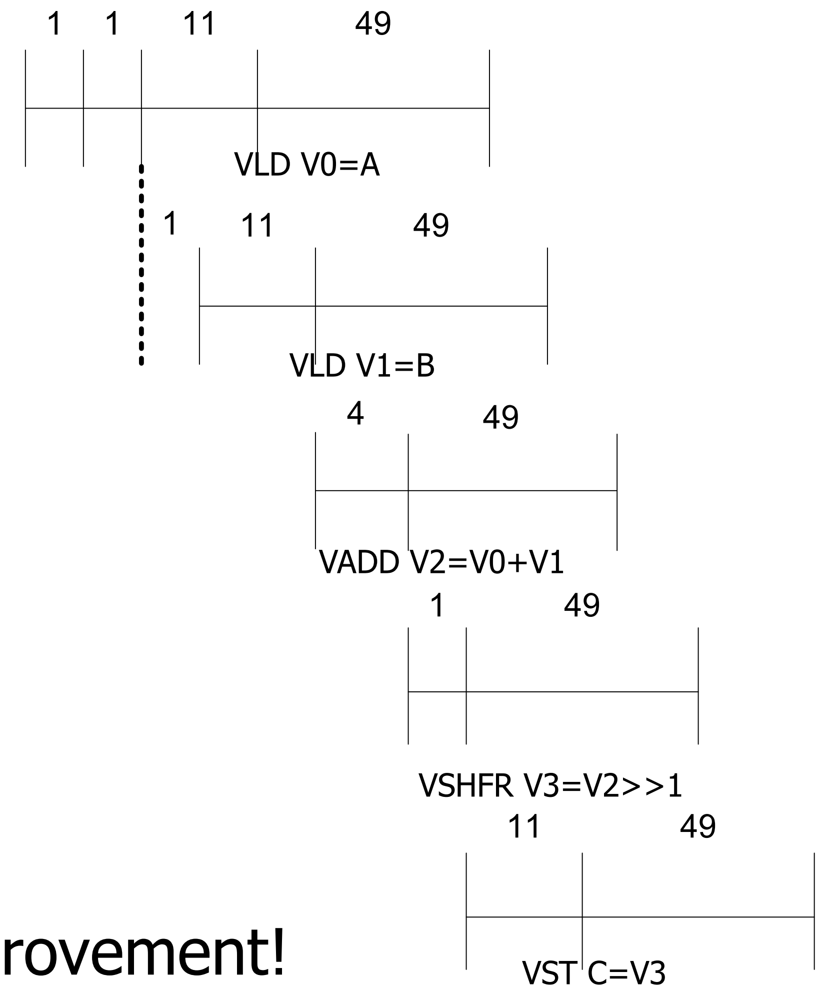

1
1
11
49
11
49
VLD V0=A
VLD V1=B
4
49
VADD V2=V0+V1
1
49
VSHFR V3=V2>>1
11
49
VST C=V3
These two VLDs cannot be pipelined. WHY?
VLD and VST cannot be pipelined. WHY?

Strict assumption: Each memory bank has a single port (memory bandwidth bottleneck)

- 182 cycles

Vector Code Performance – Multiple Memory Ports

- Chaining and 2 load ports, 1 store port in each bank



- 79 cycles
- 19X perf. improvement!

Questions (I)

- What if # data elements > # elements in a vector register?
  - Idea: Break loops so that each iteration operates on # elements in a vector register
    - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)
  - Called vector stripmining

Questions (II)

- What if vector data is not stored in a strided fashion in memory? (irregular memory access to a vector)
  - Idea: Use indirection to combine/pack elements into vector registers
  - Called scatter/gather operations
  - Doing so also helps with avoiding useless computation on sparse vectors (i.e., vectors where many elements are 0)

Gather/Scatter Operations

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (*Gather*)

```
LV vD, rD          # Load indices in D vector
LVI vC, rC, vD     # Load indirect from rC base
LV vB, rB          # Load B vector
ADDV.D vA,vB,vC    # Do add
SV vA, rA          # Store result
```

Gather/Scatter Operations

- Gather/scatter operations often implemented in hardware to handle sparse vectors (matrices) or indirect indexing
- Vector loads and stores use an index vector which is added to the base register to generate the addresses

- Scatter example

Index Vector	Data Vector (to Store)	Stored Vector (in Memory)	
0	3.14	Base+0	3.14
2	6.5	Base+1	X
6	71.2	Base+2	6.5
7	2.71	Base+3	X
		Base+4	X
		Base+5	X
		Base+6	71.2
		Base+7	2.71

Question (III): Conditional Operations in a Loop

- What if some operations should not be executed on a vector (based on a dynamically-determined condition)?

```
loop:       for (i=0; i<N; i++)
                      if (a[i] != 0) then b[i]=a[i]*b[i]
```

- Idea: Masked operations
  - VMASK register is a bit mask determining which data element should not be acted upon
  - VLD V0 = A
  - VLD V1 = B
  - VMASK = (V0 != 0)
  - VMUL V1 = V0 * V1
  - VST B = V1
    - This is predicated execution. Execution is predicated on mask bit.

Another Example with Masking

```
for (i = 0; i < 64; ++i)
    if (a[i] >= b[i])
        c[i] = a[i]
    else
        c[i] = b[i]
```

A	B	VMASK
1	2	0
2	2	1
3	2	1
4	10	0
-5	-4	0
0	-3	1
6	5	1
-7	-8	1

Steps to execute the loop in SIMD code

1. Compare A, B to get VMASK

2. Masked store of A into C

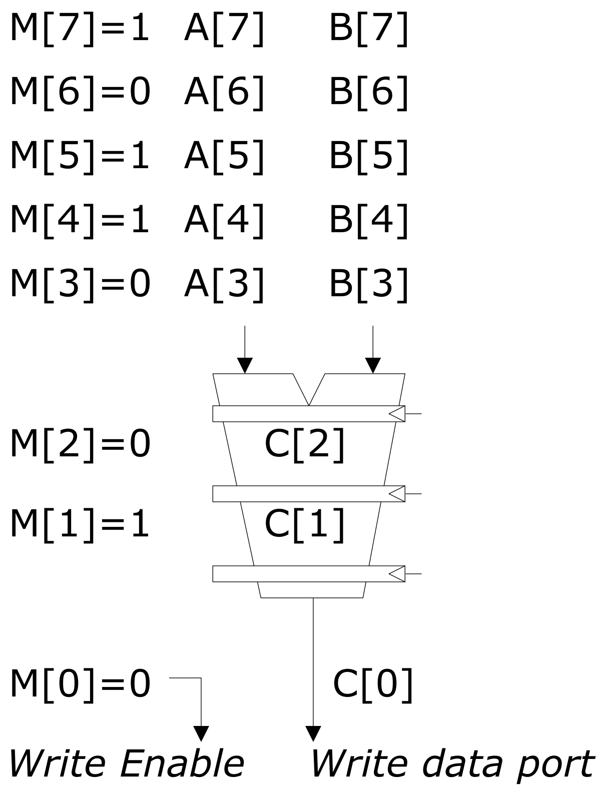
3. Complement VMASK

4. Masked store of B into C

Masked Vector Instructions

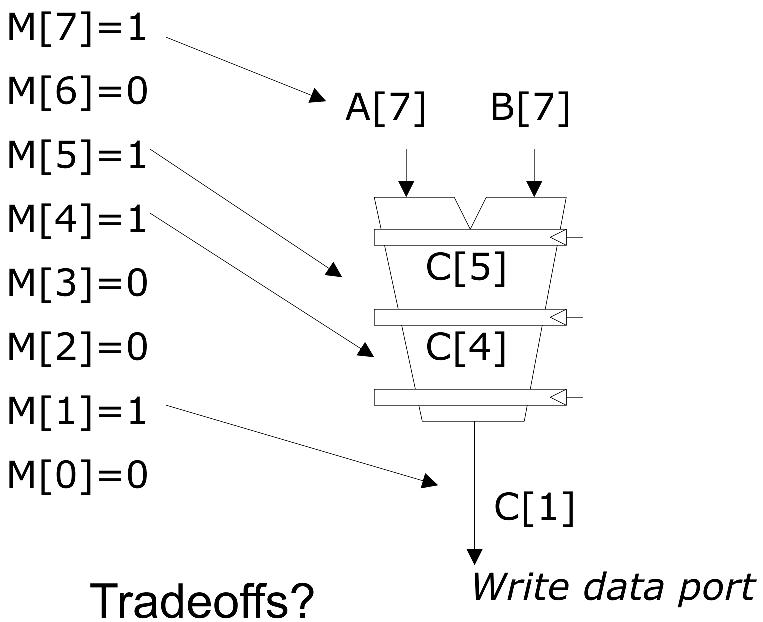
Simple Implementation

– execute all N operations, turn off result writeback according to mask



Density-Time Implementation

– scan mask vector and only execute elements with non-zero masks



Tradeoffs?

- **缺陷**
  - 简单实现时，条件不满足时向量指令仍然需要花费时间
  - 有些向量处理器针对带条件的向量执行时，仅控制向目标寄存器的写操作，可能会有除法错误

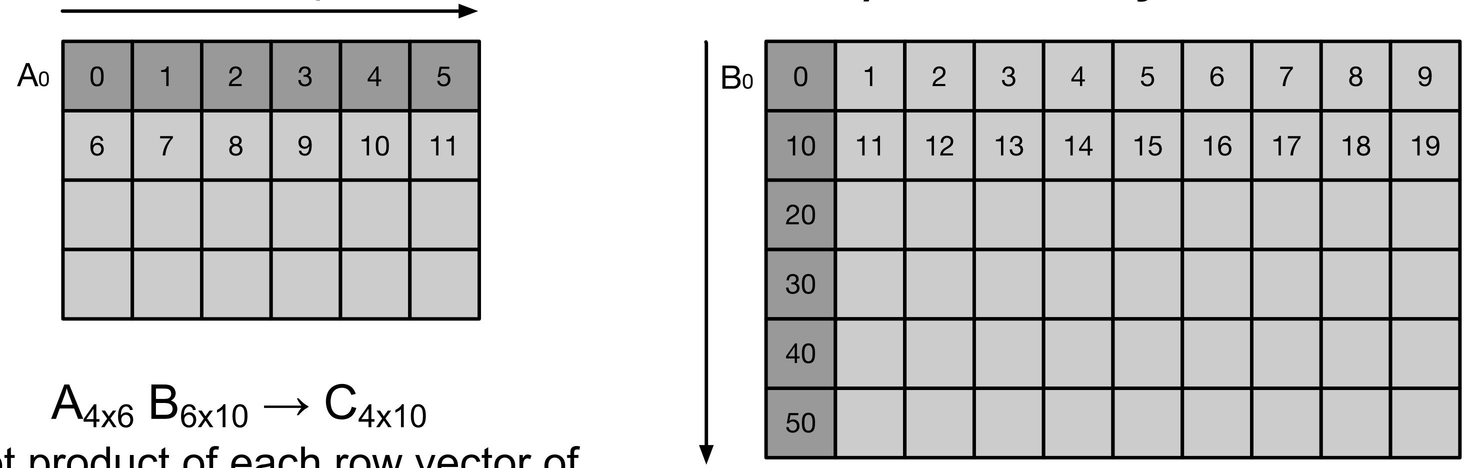
Some Issues

- Stride and bank count
  - As long as stride and bank count are relatively *prime* to each other and there are enough banks to cover bank access latency, we can sustain 1 element/cycle throughput

- Storage format of a matrix
  - Row major: Consecutive elements in a row are laid out consecutively in memory
  - Column major: Consecutive elements in a column are laid out consecutively in memory
  - You need to change the stride when accessing a row versus column

Bank Conflicts in Matrix Multiplication

- A and B matrices, both stored in memory in row-major order



$A_{4x6}\ B_{6x10} \rightarrow C_{4x10}$

Dot product of each row vector of A with each column vector of B

- Load A's row 0 into vector register V1
  - Each time, increment address by 1 to access the next column
  - Accesses have a stride of 1
- Load B's column 0 into vector register V2
  - Each time, increment address by 10
  - Accesses have a stride of 10

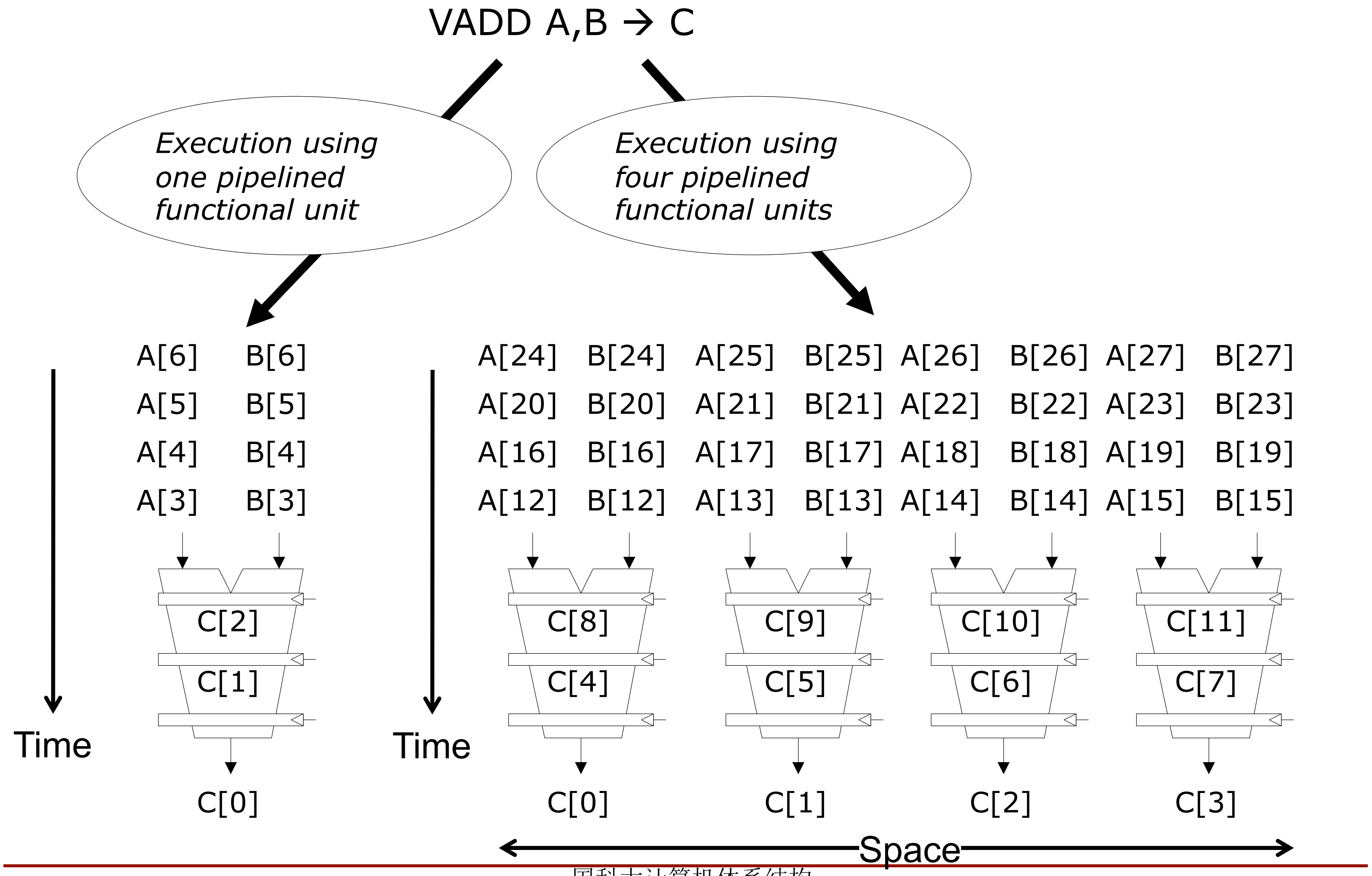
Different strides can lead to bank conflicts

How do we minimize them?

Minimizing Bank Conflicts

- More banks
- More ports in each bank
- Better data layout to match the access pattern
  - Is this always possible?
- Better mapping of address to bank
  - E.g., randomized mapping
  - Rau, "Pseudo-randomly interleaved memory," ISCA 1991.

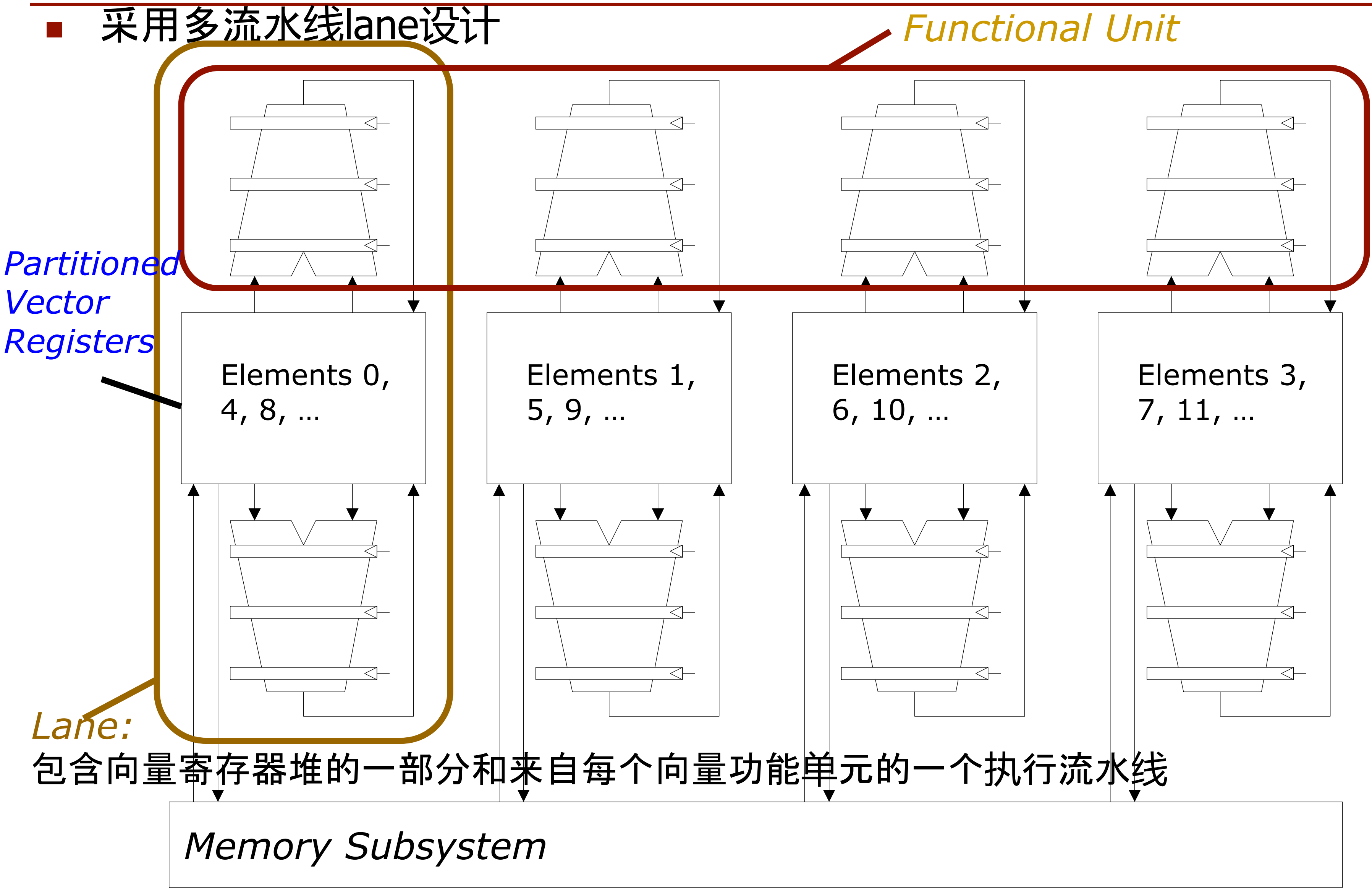
Vector Instruction Execution



VADD A,B → C
Execution using one pipelined functional unit
Execution using four pipelined functional units
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]
Time
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]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]
Time
Space

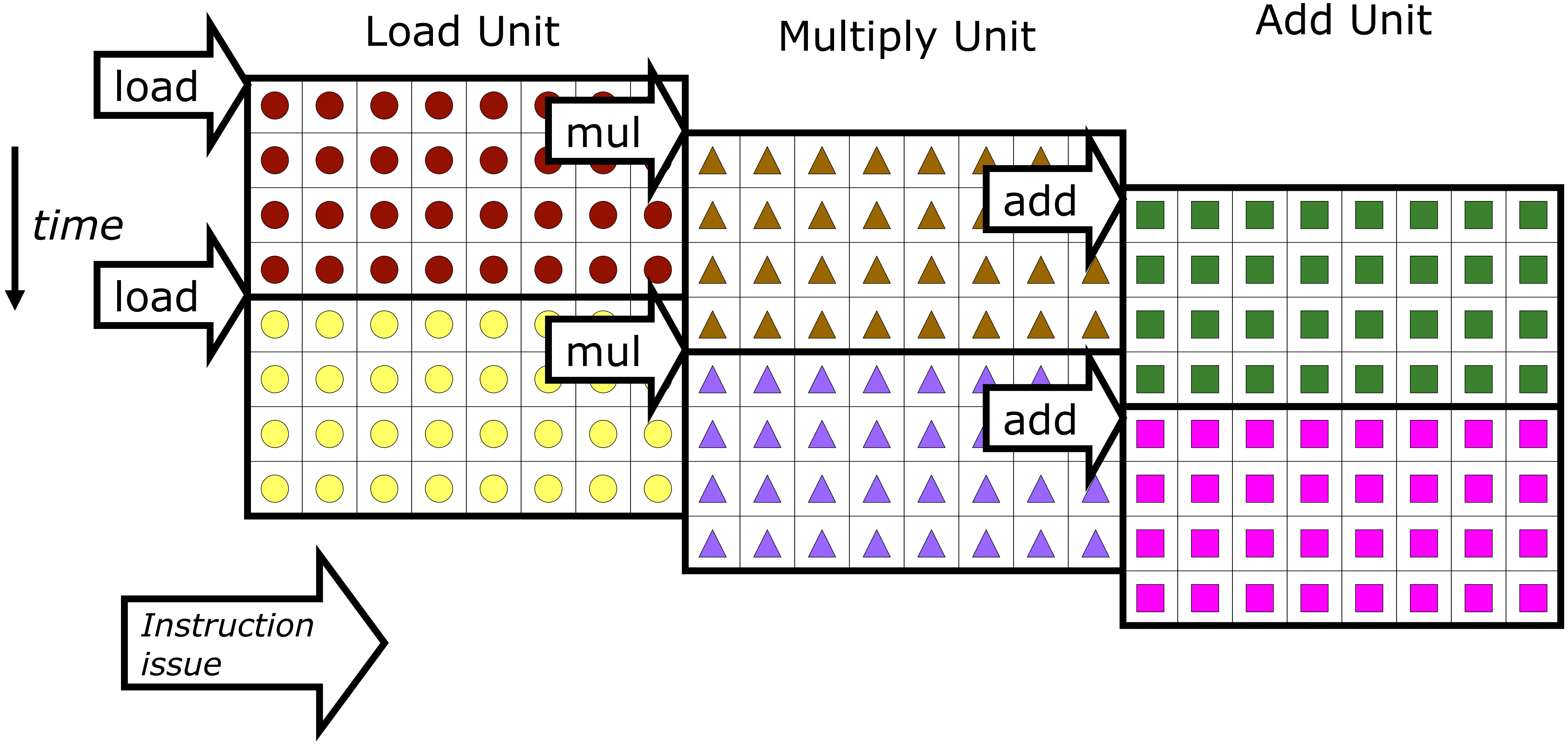
Vector Unit Structure

- 采用多流水线lane设计



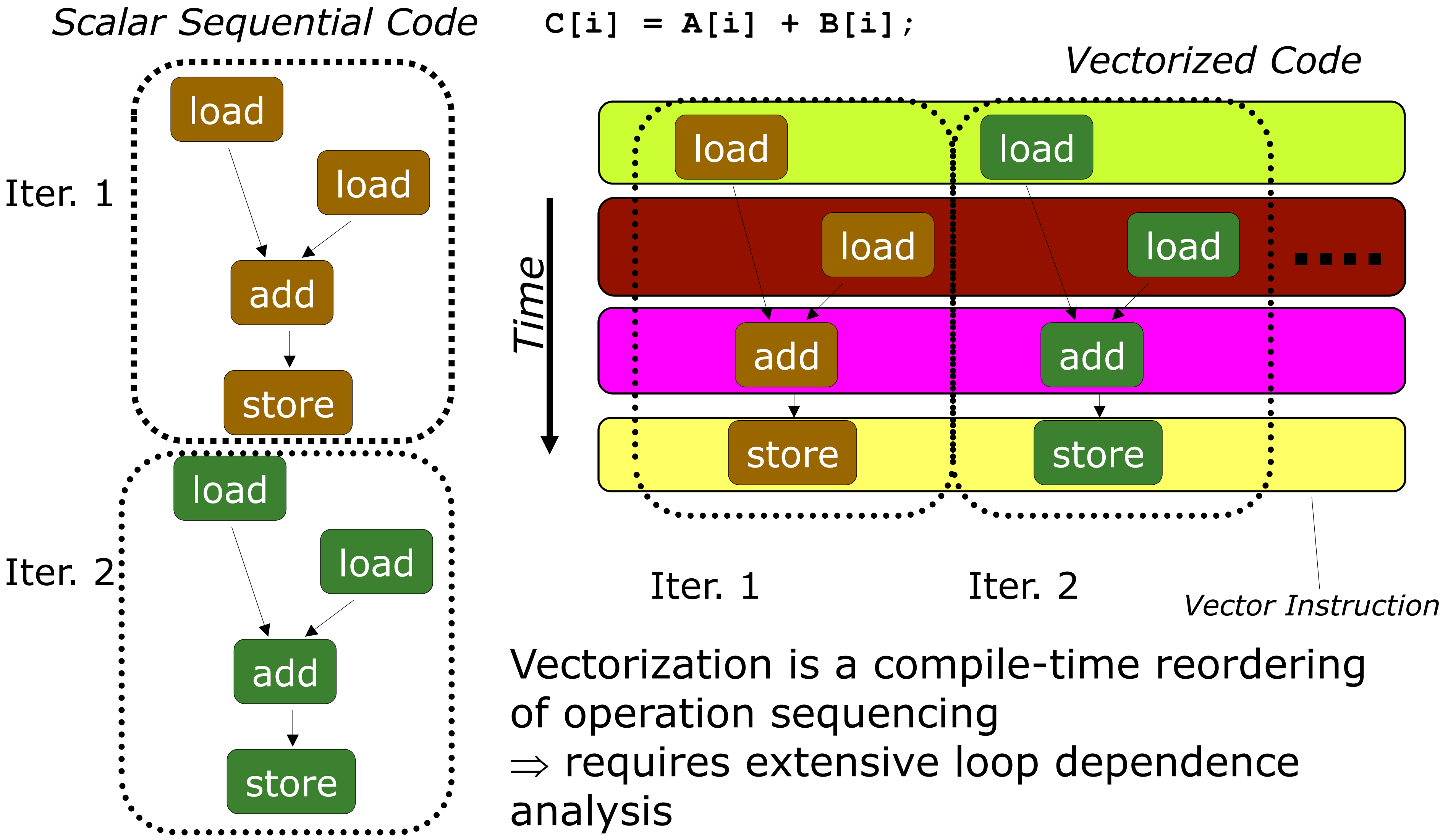
Vector Instruction Level Parallelism

- Can overlap execution of multiple vector instructions
  - Example machine has 32 elements per vector register and 8 lanes
  - Example with 24 operations/cycle (steady state) while issuing 1 vector instruction/cycle



Automatic Code Vectorization

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



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

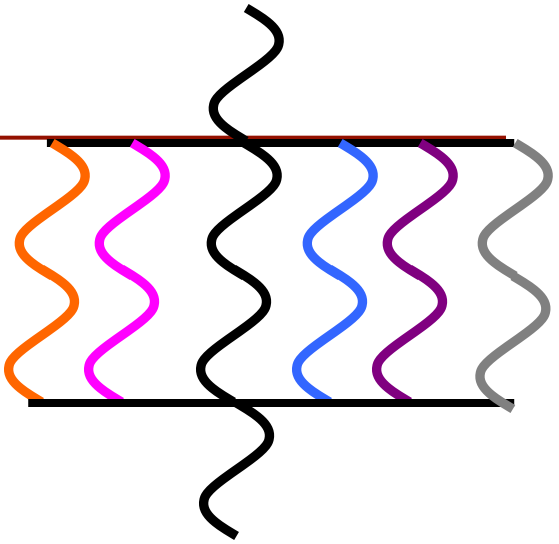
Vector/SIMD Processing Summary

- 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 (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

Recall: Amdahl's Law



- Amdahl's Law
  - f: Parallelizable fraction of a program
  - N: Number of processors

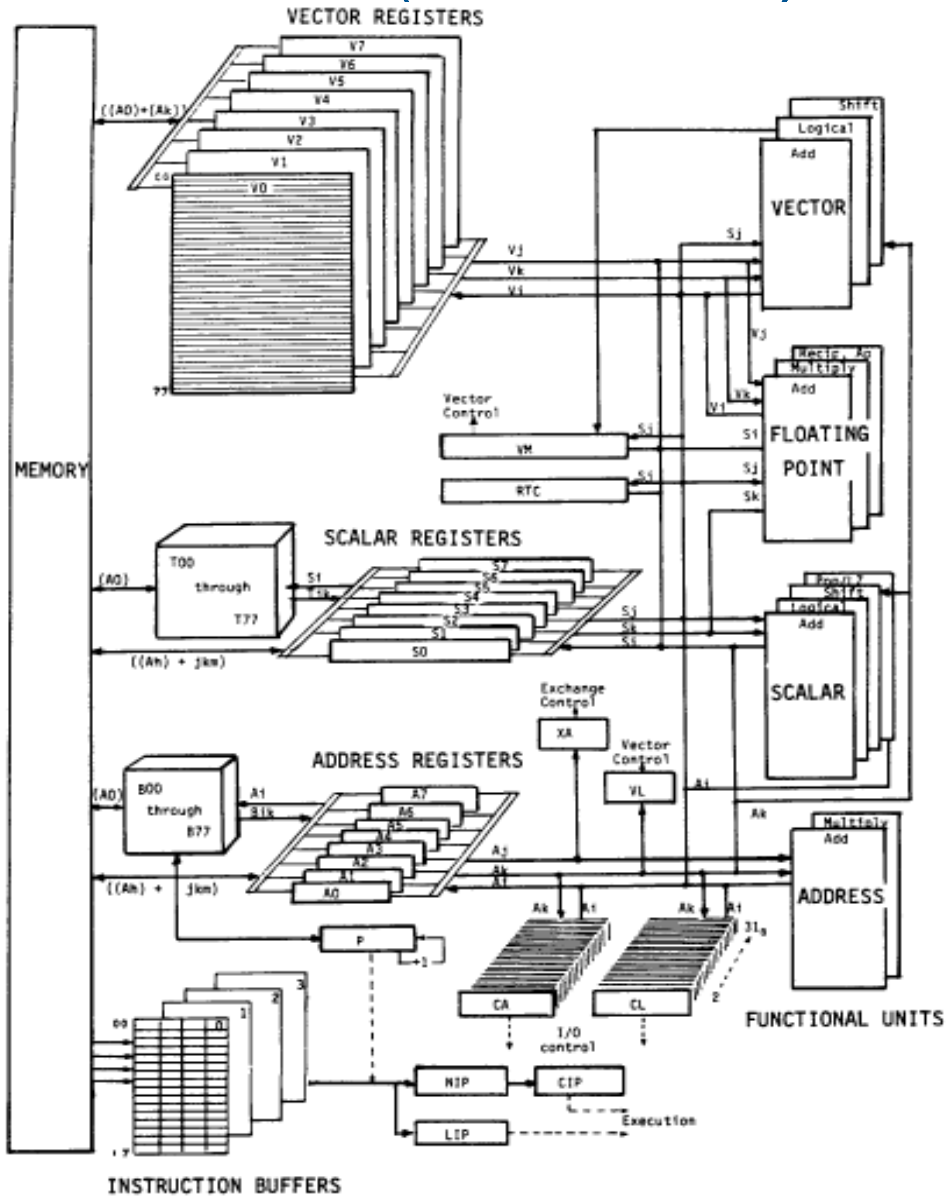
$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

  - Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

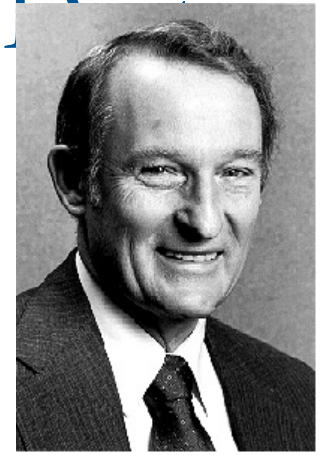
- Maximum speedup limited by serial portion: Serial bottleneck
- All parallel machines "suffer from" the serial bottleneck

Vector Machine Organization (CRAY-1)

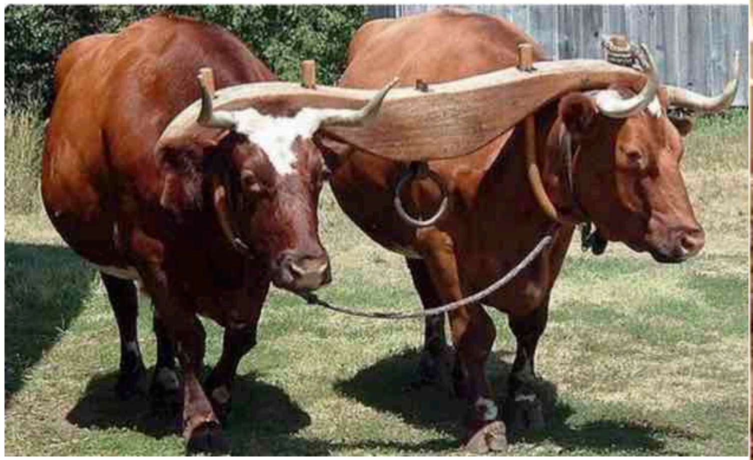
- CRAY-1
- Russell, "The CRAY-1 computer system," CACM 1978.
  - Scalar and vector modes
  - 8 64-element vector registers
  - 64 bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers



Seymour Cray, Leader in Supercomputer



"If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"



© amityrebecca / Pinterest. https://www.pinterest.ch/pin/473018767088408061/

© Scott Sinklier / Corbis. http://america.aljazeera.com/articles/2015/2/20/the-short-brutal-life-of-male-chickens.html

https://en.wikipedia.org/wiki/Seymour_Cray

GPUs (Graphics Processing Units)

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

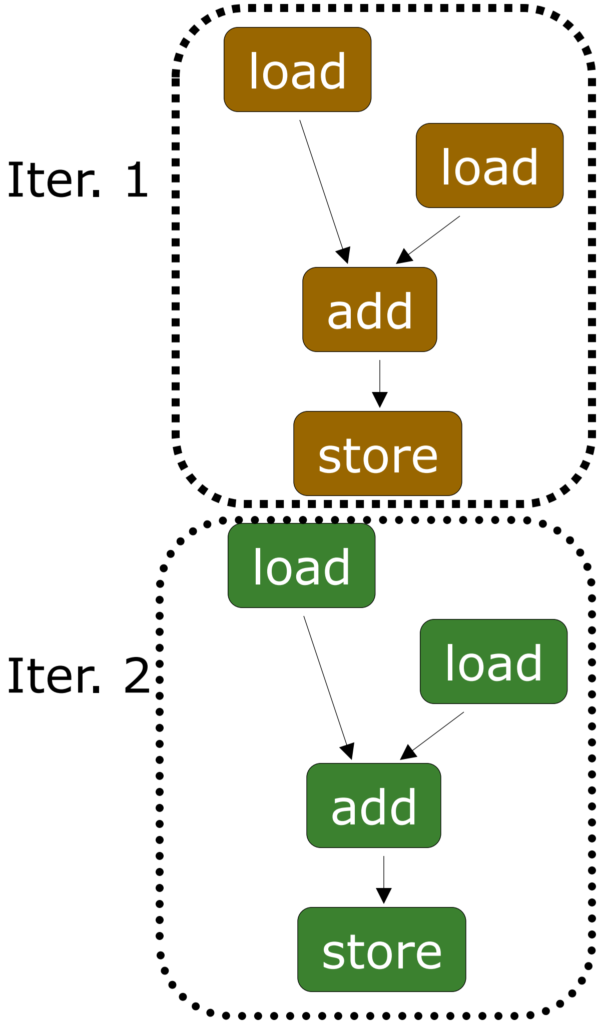
- Programming Model refers to how the programmer expresses the code
  - E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), …
- Execution Model refers to how the hardware executes the code underneath
  - E.g., Out-of-order execution, Vector processor, Array processor, 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)

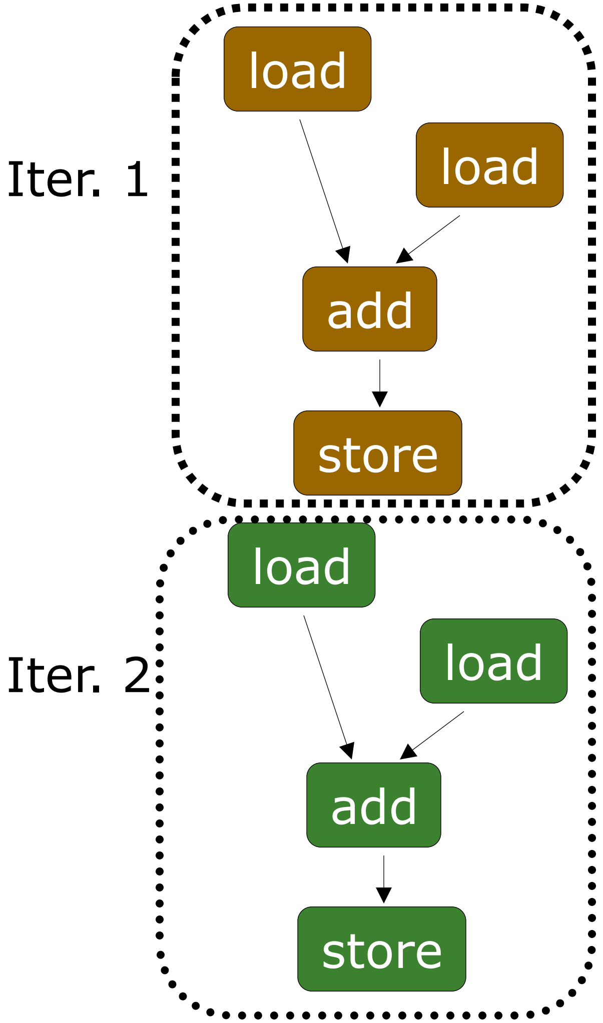
2. Data-Parallel (SIMD)

3. Multithreaded (MIMD/SPMD)

Prog. Model 1: Sequential (SISD)

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

Scalar Sequential Code

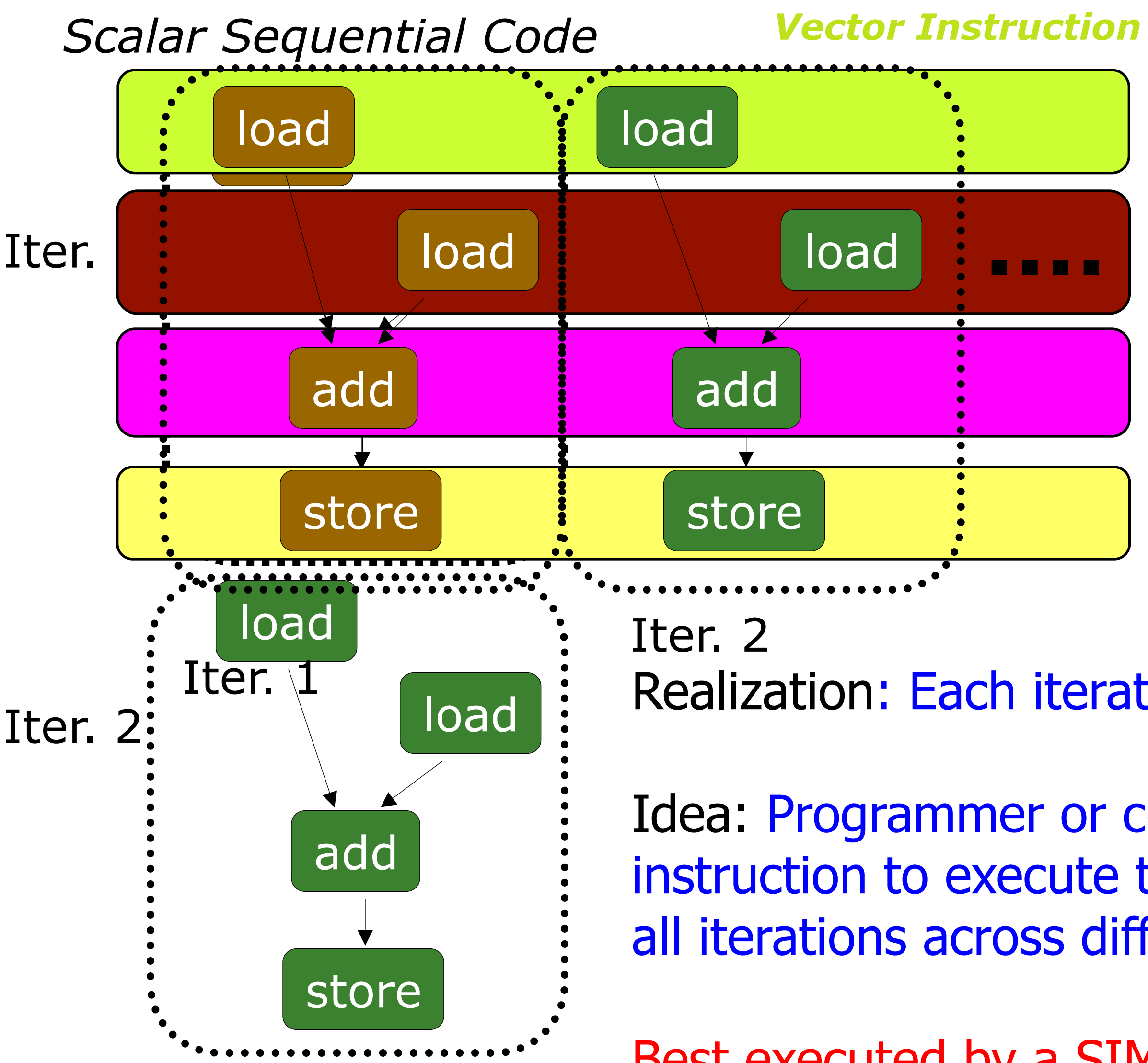


- Can be executed on a:

- 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];
```



Vectorized Code

VLD	A → V1
VLD	B → V2
VADD	V1 + V2 → V3
VST	V3 → C

Realization: Each iteration is independent

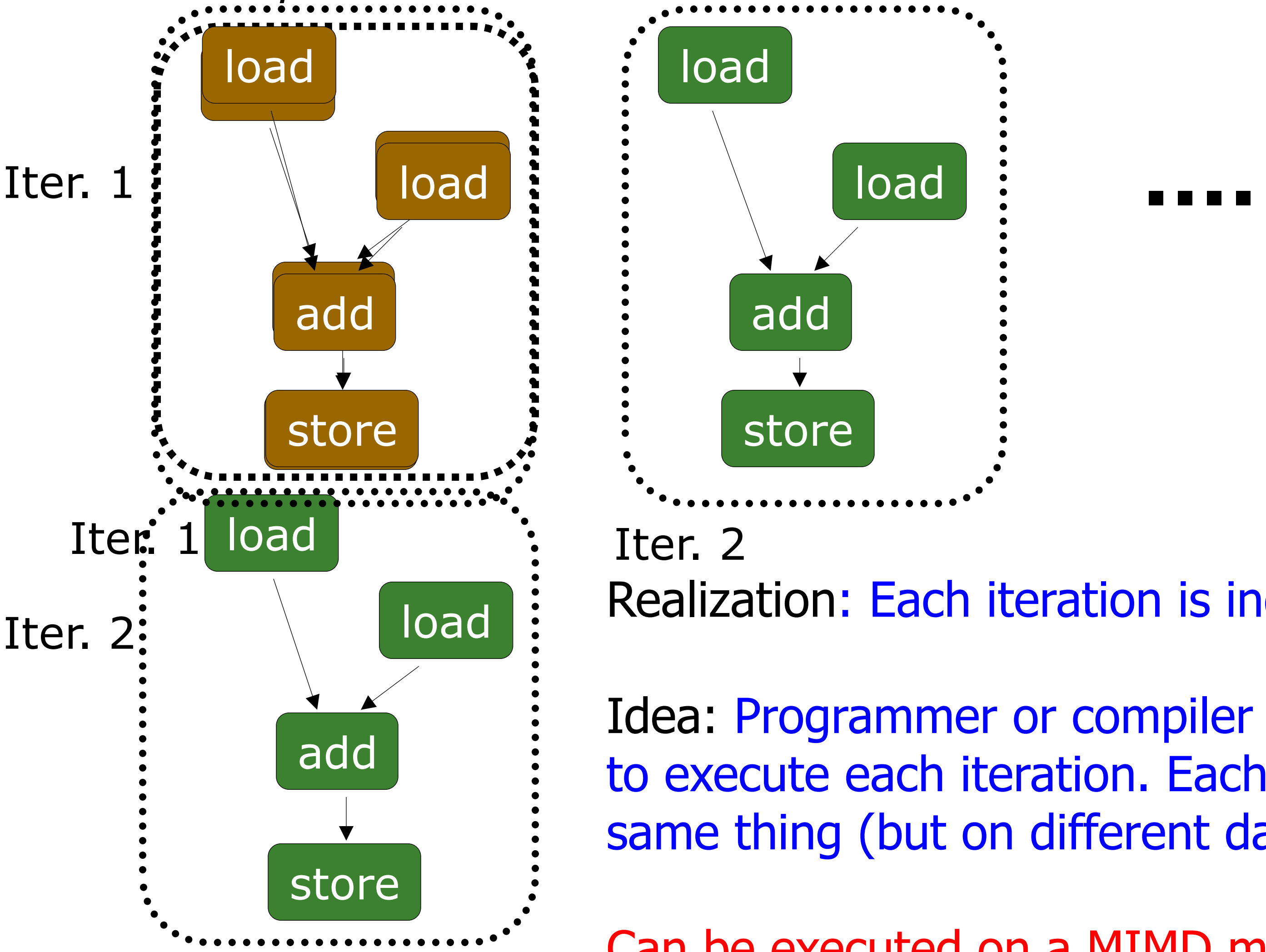
Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

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



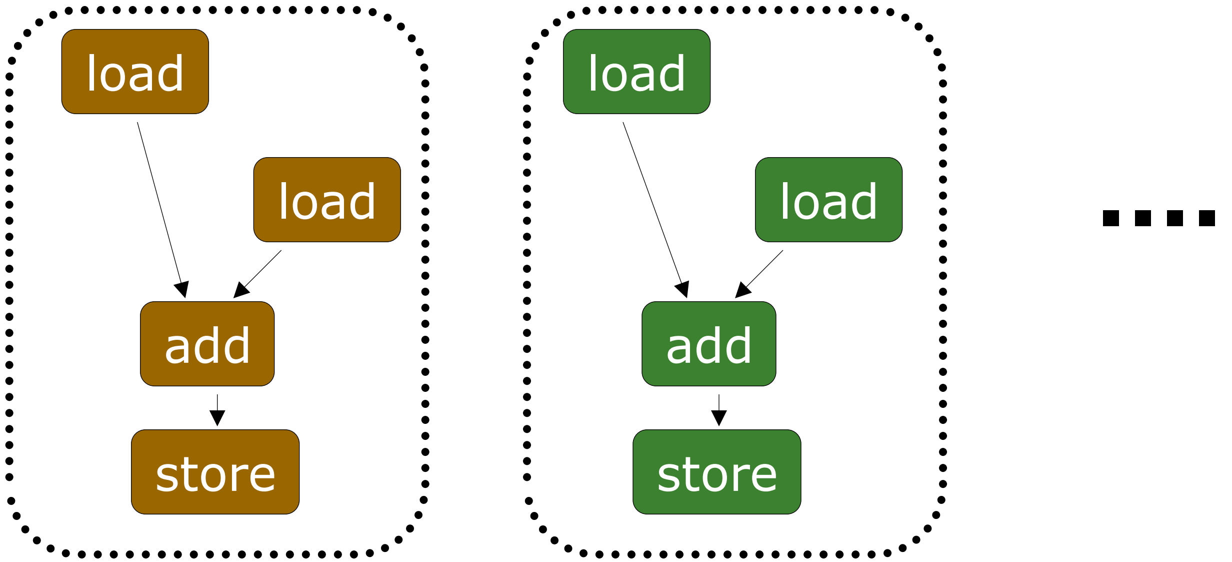
Realization: Each iteration is independent

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];
```



Iter. 1

Iter. 2

Realization: Each iteration is independent

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

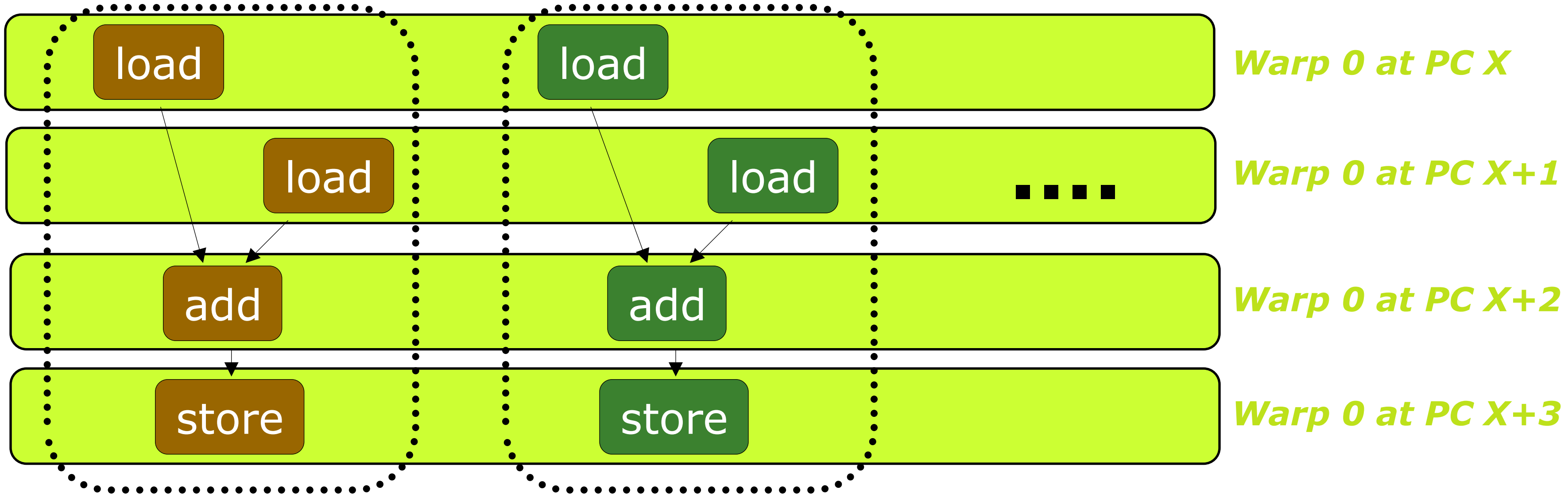
- Except it is not programmed using SIMD instructions

- 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];
```



Iter. 1

Iter. 2

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

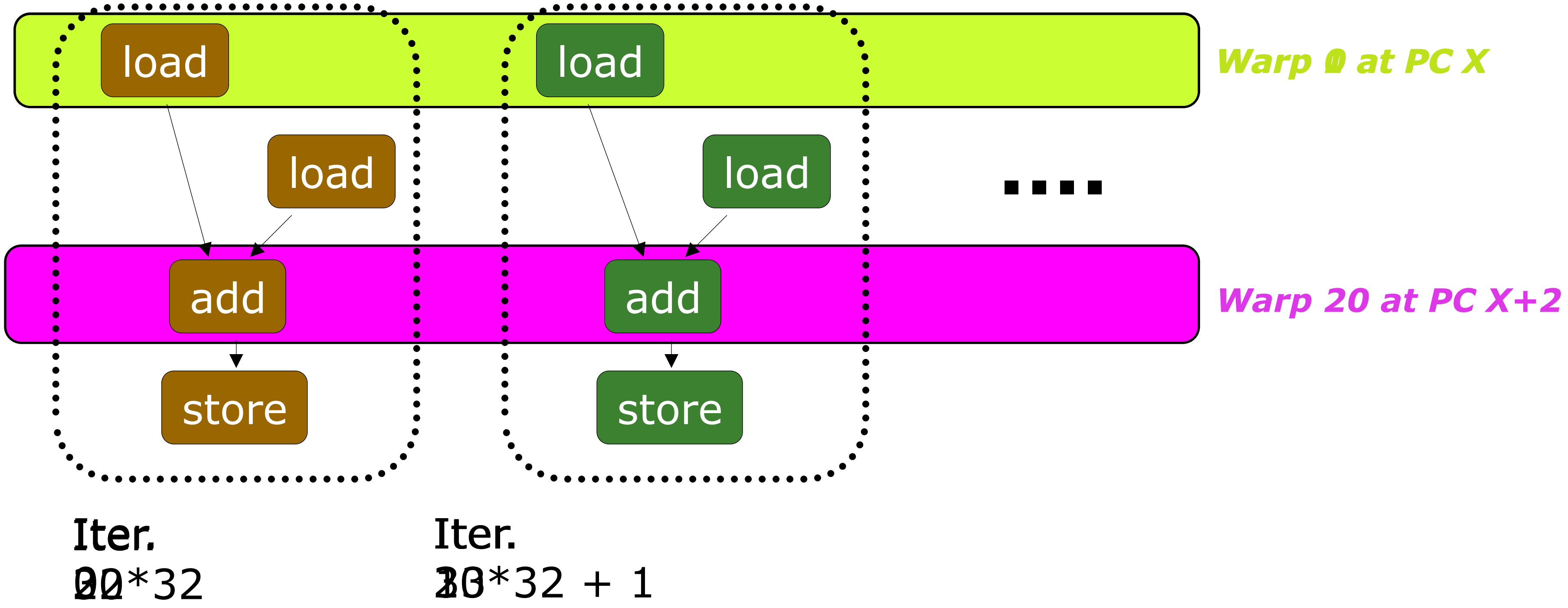
SIMD vs. SIMT Execution Model

- 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

Fine-Grained Multithreading of Warps

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

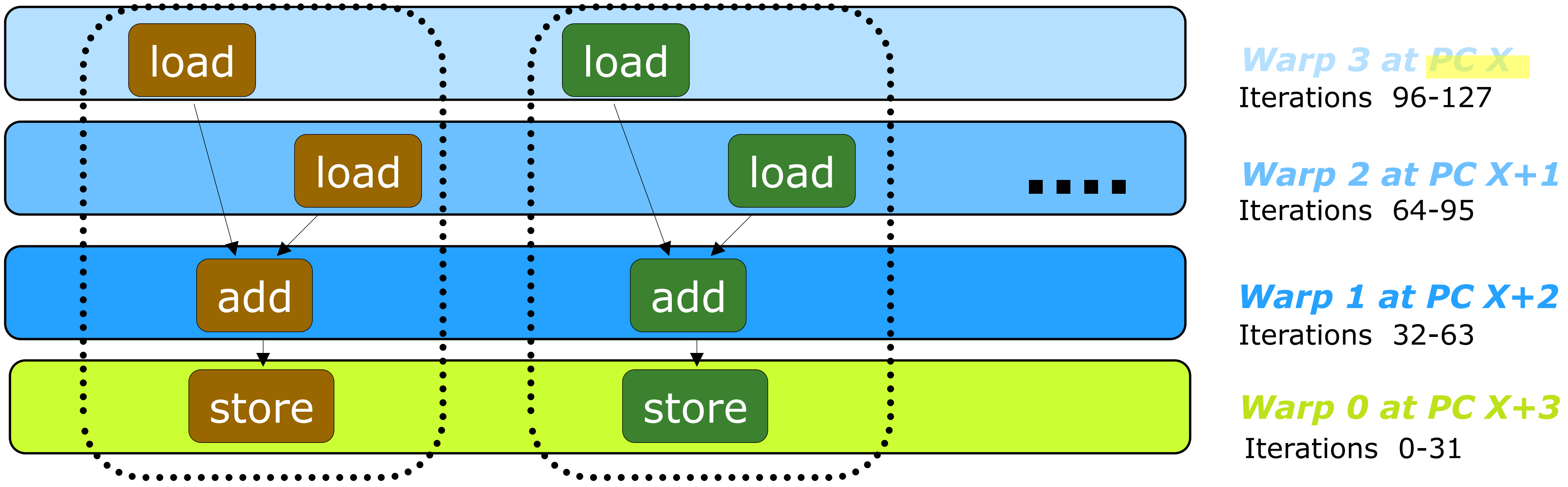
- Assume a warp consists of 32 threads
- If you have 32K iterations, and 1 iteration/thread → 1K warps
- Warps can be interleaved on the same pipeline → Fine grained multithreading of warps



Fine-Grained Multithreading of Warps

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

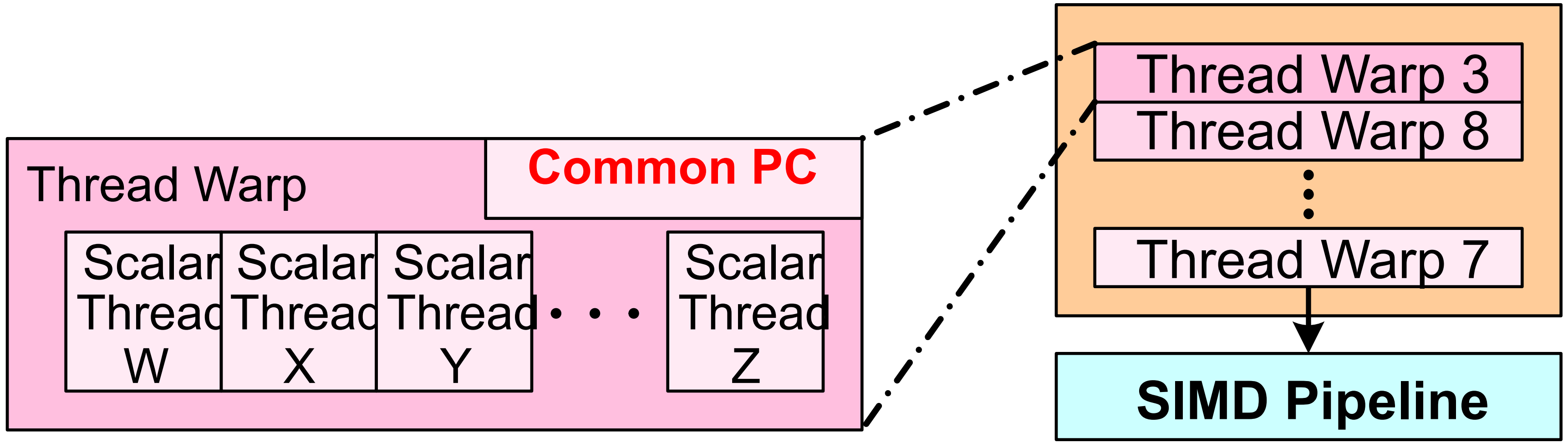
- Assume a warp consists of 32 threads
- If you have 32K iterations, and 1 iteration/thread → 1K warps
- Warps can be interleaved on the same pipeline → Fine grained multithreading of warps



All threads in a warp are independent of each other
→ They be executed seamlessly in a fine-grained multithreaded pipeline

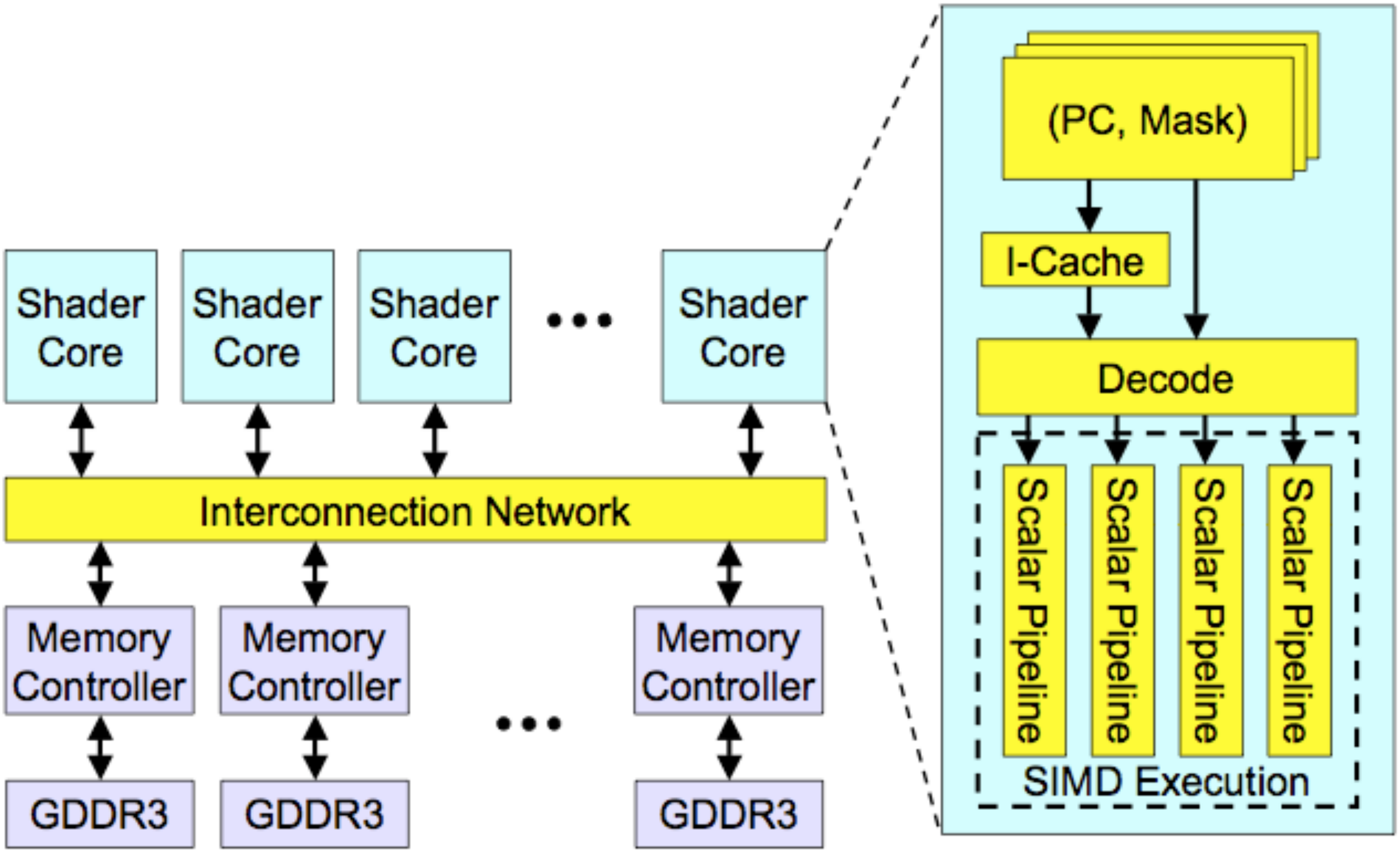
Warps and Warp-Level FGMT

- 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 …



Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

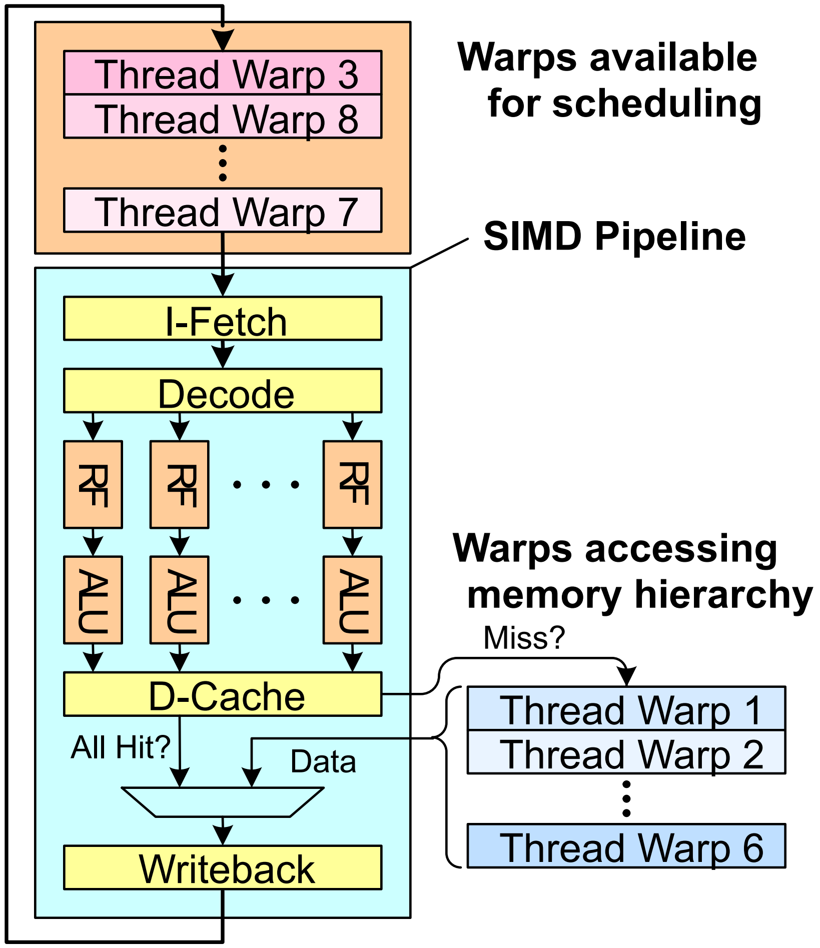
High-Level View of a GPU



Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

Latency Hiding via Warp-Level FGMT

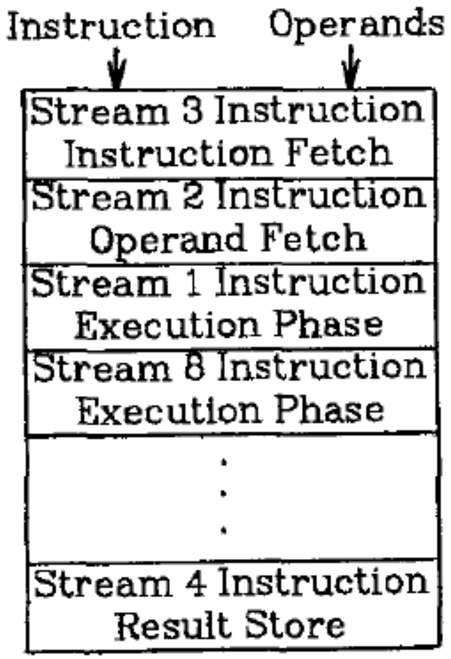
- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables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



Slide credit: Tor Aamodt

Multithreading

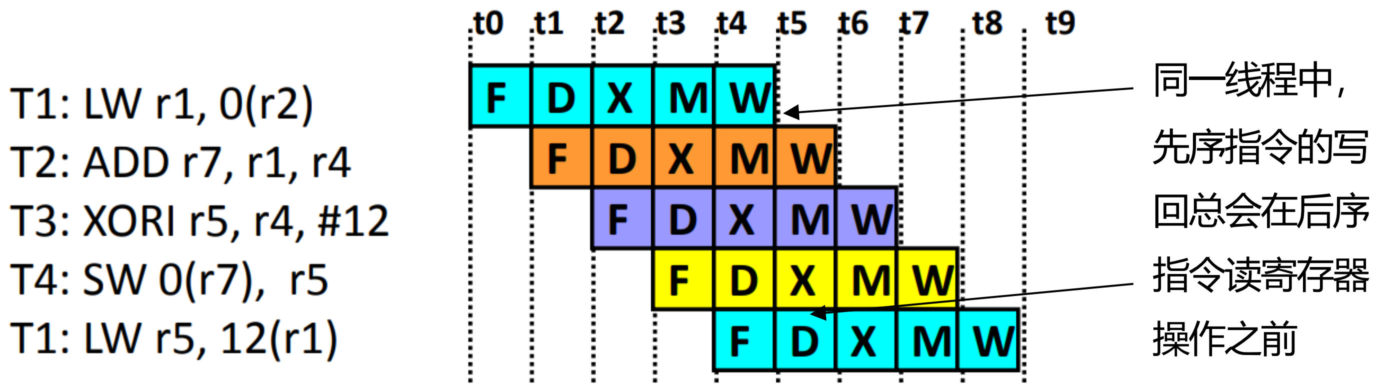
- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads' instructions
- + No logic needed for handling control and
- data dependences within a thread
- -- Single thread performance suffers
- -- Extra logic for keeping thread contexts
- -- Does not overlap latency if not enough
- threads to cover the whole pipeline


Instruction
Operands
Stream 3 Instruction Instruction Fetch
Stream 2 Instruction Operand Fetch
Stream 1 Instruction Execution Phase
Stream 8 Instruction Execution Phase
Stream 4 Instruction Result Store

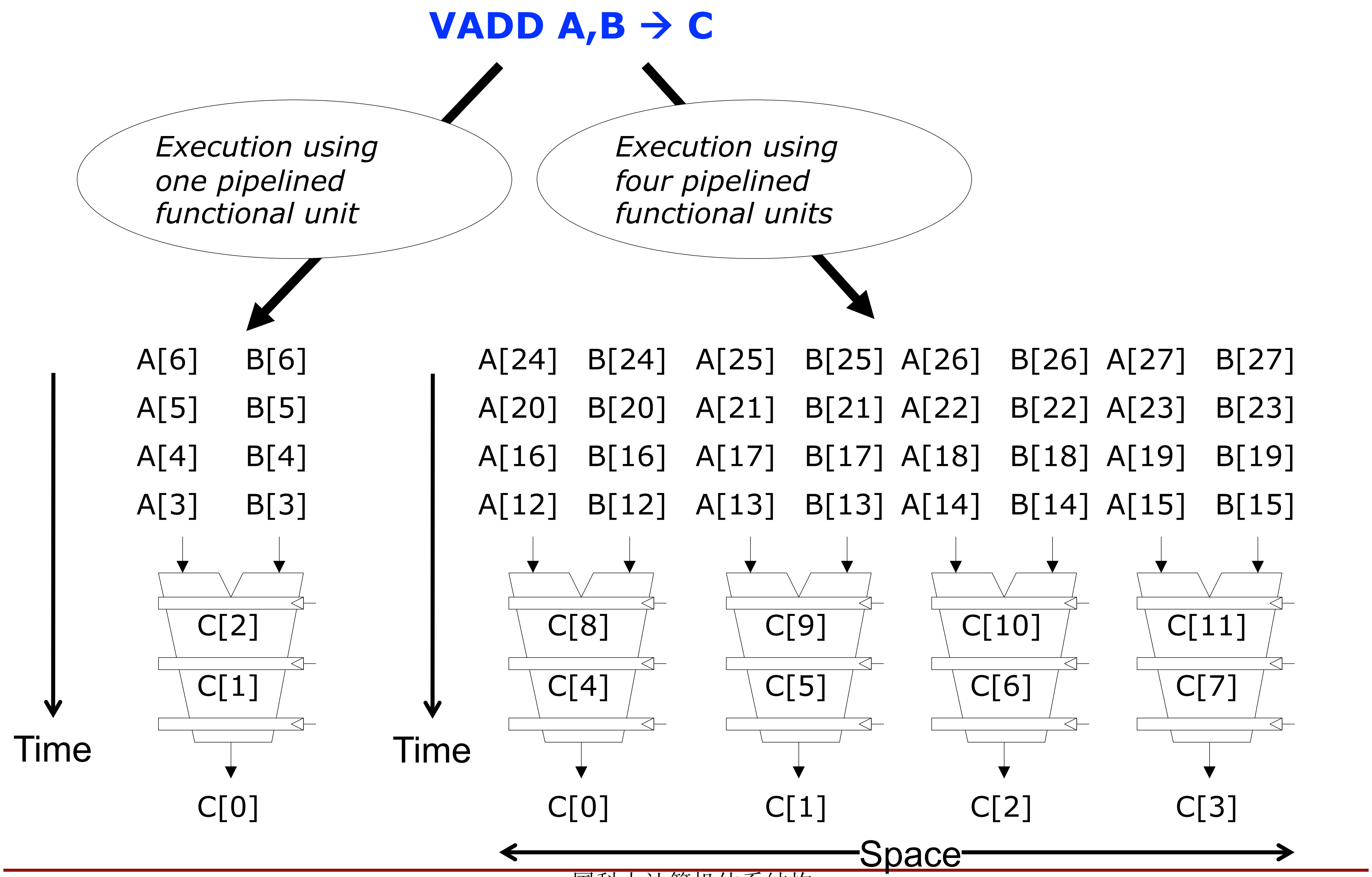
多线程策略

- 如何保证一条流水线上的指令之间不存在数据依赖关系？
  - 一种办法：在相同的流水线中交叉执行来自不同线程的指令

T_1-T_4 4个线程交叉执行，无旁路5阶段流水线



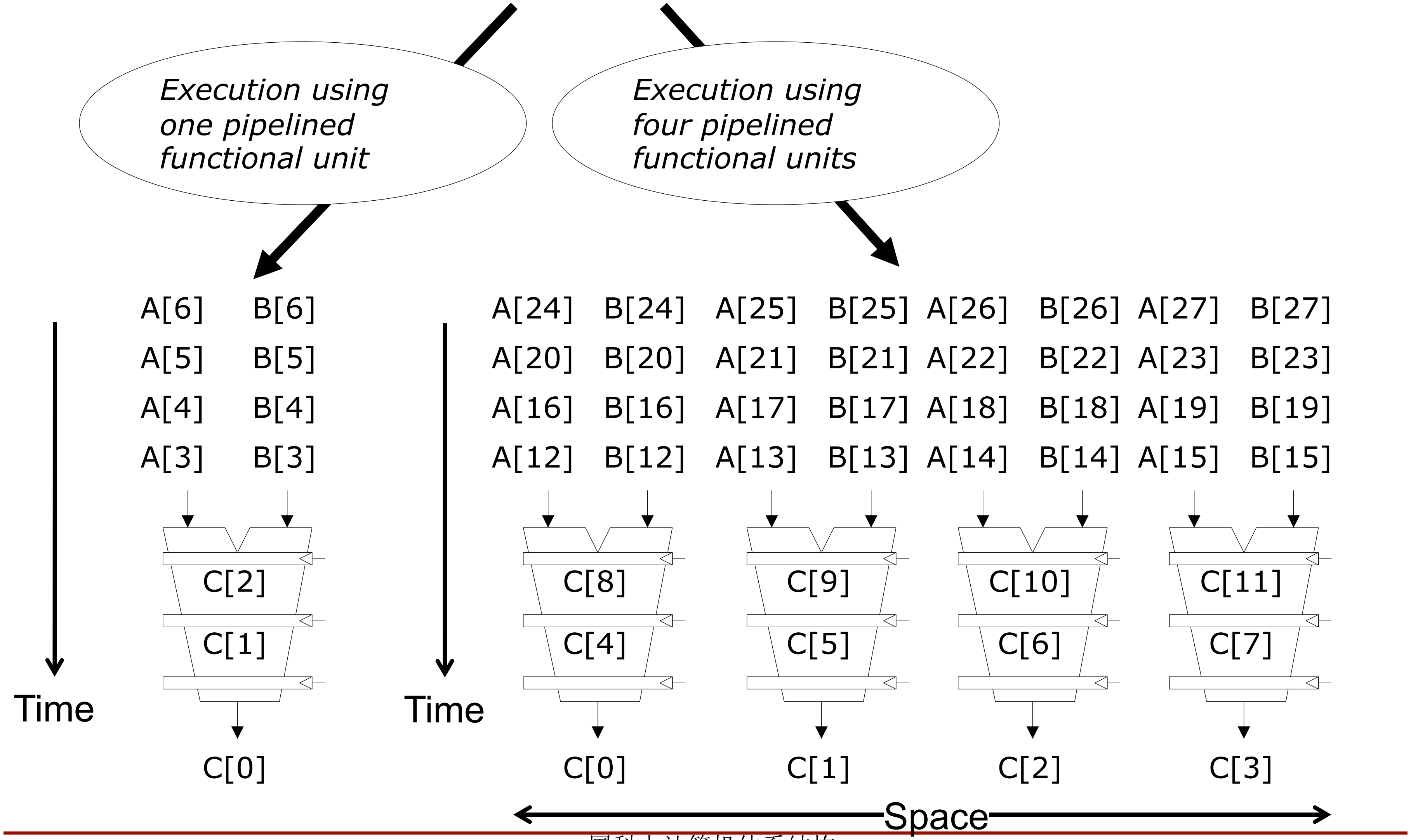
Recall: Vector Instruction Execution



VADD A,B → C
Execution using one pipelined functional unit
Execution using four pipelined functional units
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]
Time
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]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]
Time
Space

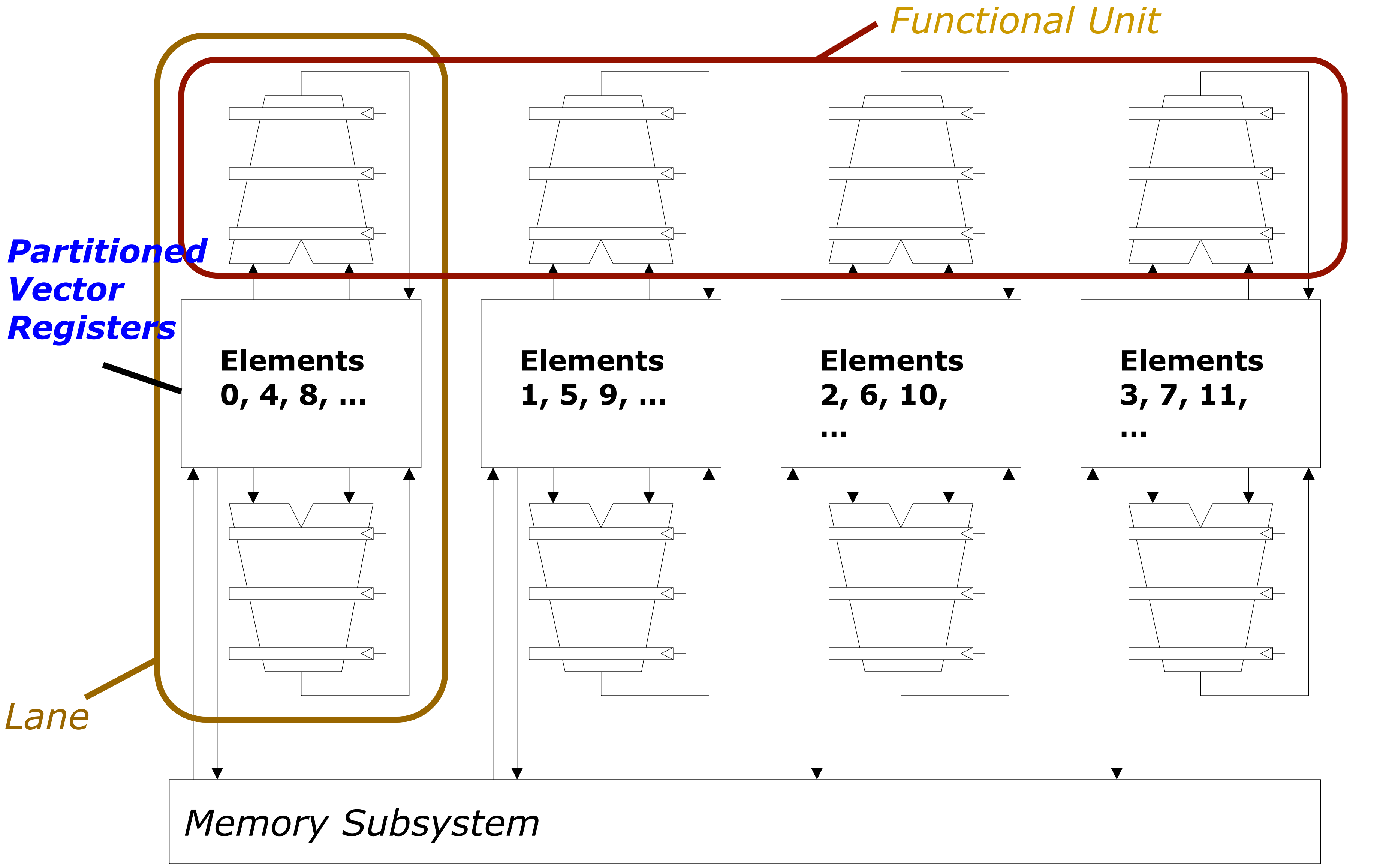
Warp Execution

32-thread warp executing **ADD A[tid],B[tid] → C[tid]**

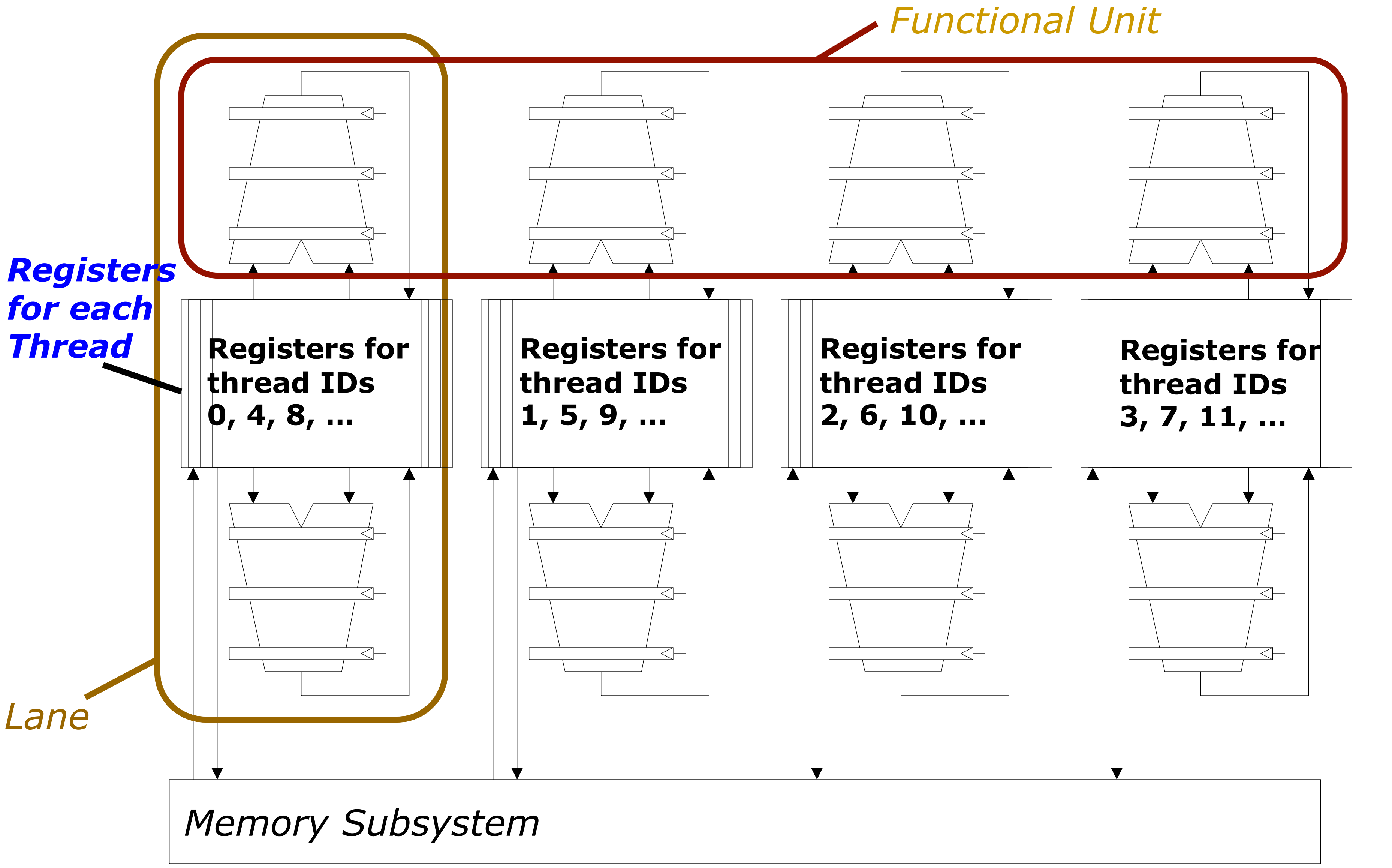


Execution using one pipelined functional unit
Execution using four pipelined functional units
A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]
Time
A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]
Time
Space

Recall: Vector Unit Structure



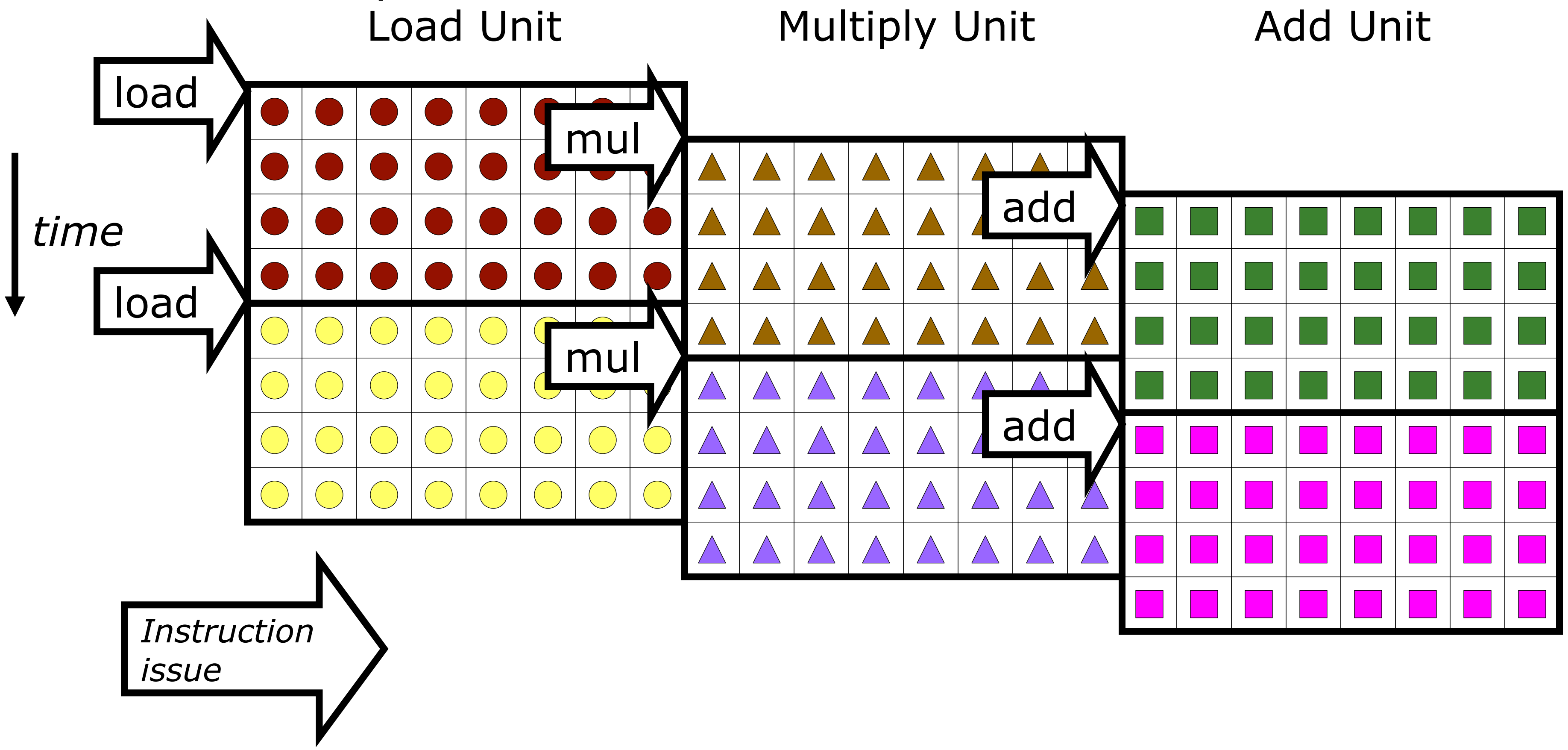
GPU SIMD Execution Unit Structure


Functional Unit
Registers for each Thread
Registers for thread IDs 0, 4, 8, …
Registers for thread IDs 1, 5, 9, …
Registers for thread IDs 2, 6, 10, …
Registers for thread IDs 3, 7, 11, …
Lane
Memory Subsystem

Slide credit: Krste Asanovic

Recall: Vector Instruction Level Parallelism

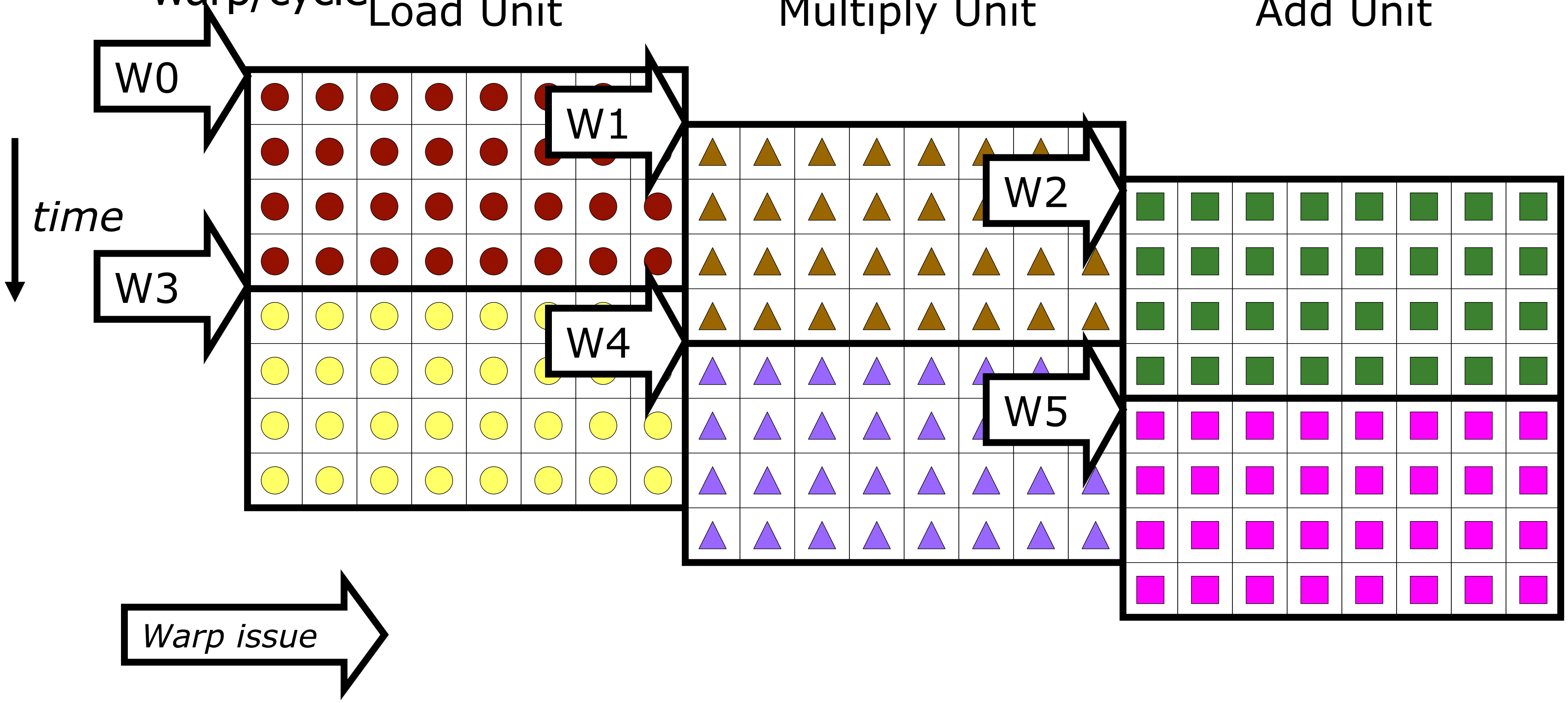
- Can overlap execution of multiple vector instructions
  - Example machine has 32 elements per vector register and 8 lanes
  - Example with 24 operations/cycle (steady state) while issuing 1 vector instruction/cycle



Slide credit: Krste Asanovic

Warp Instruction Level Parallelism

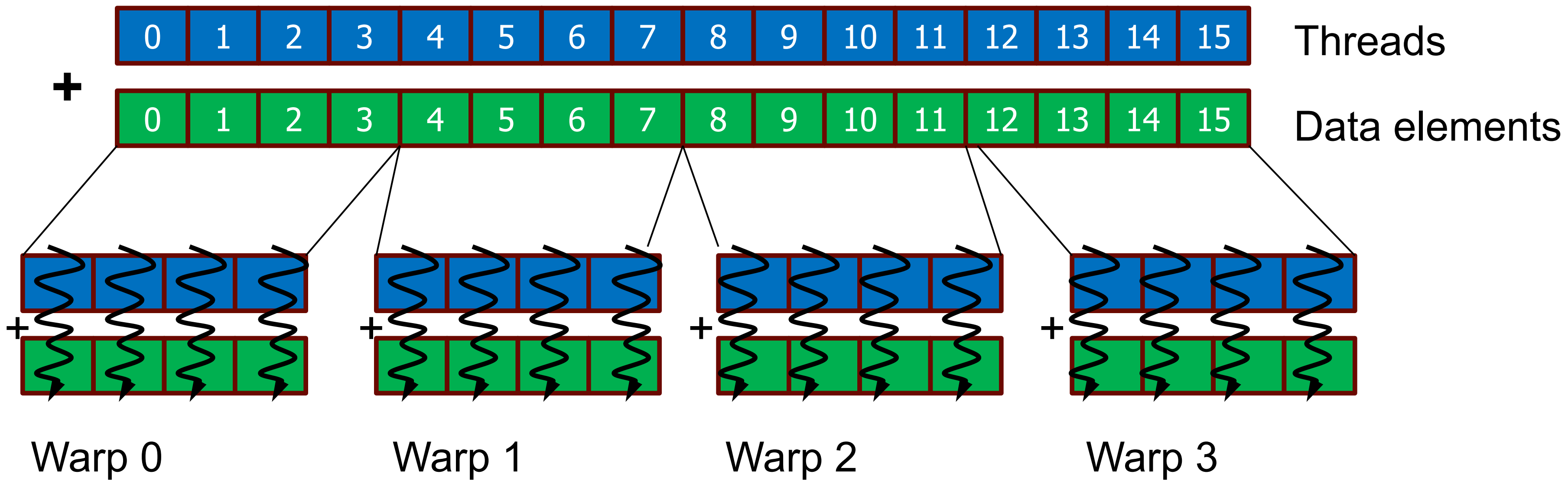
- Can overlap execution of multiple instructions
  - Example machine has 32 threads per warp and 8 lanes
  - Completes 24 operations/cycle (steady state) while issuing 1 warp/cycle



SIMT Memory Access (Loads and Stores)

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

Let's assume N=16, 4 threads per warp → 4 warps



For maximum performance, memory should provide enough bandwidth
(i.e., elements per cycle throughput to match computation unit throughput)

Slide credit: Hyesoon Kim

Warps not Exposed to GPU Programmers

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU: Blocks of threads

Serial Code (host)

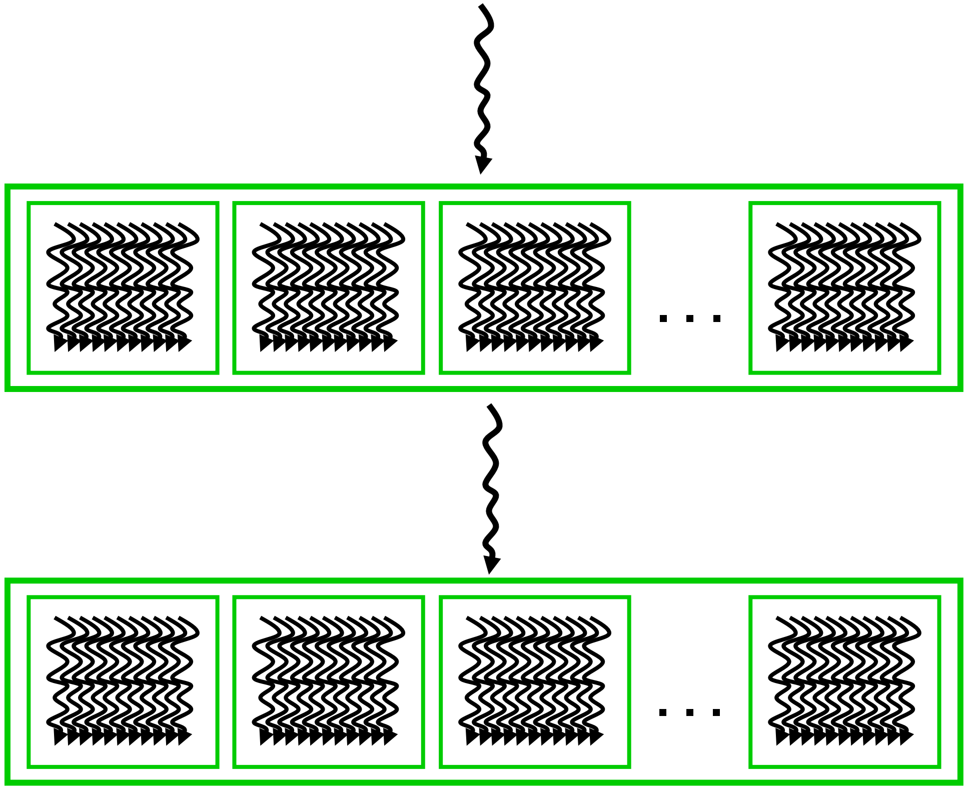
Parallel Kernel (device)

`KernelA<<<nBlk, nThr>>>(args);`

Serial Code (host)

Parallel Kernel (device)

`KernelB<<<nBlk, nThr>>>(args);`

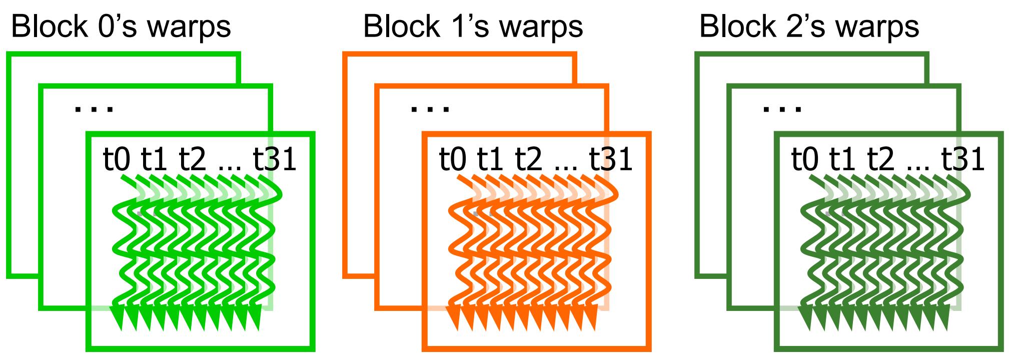


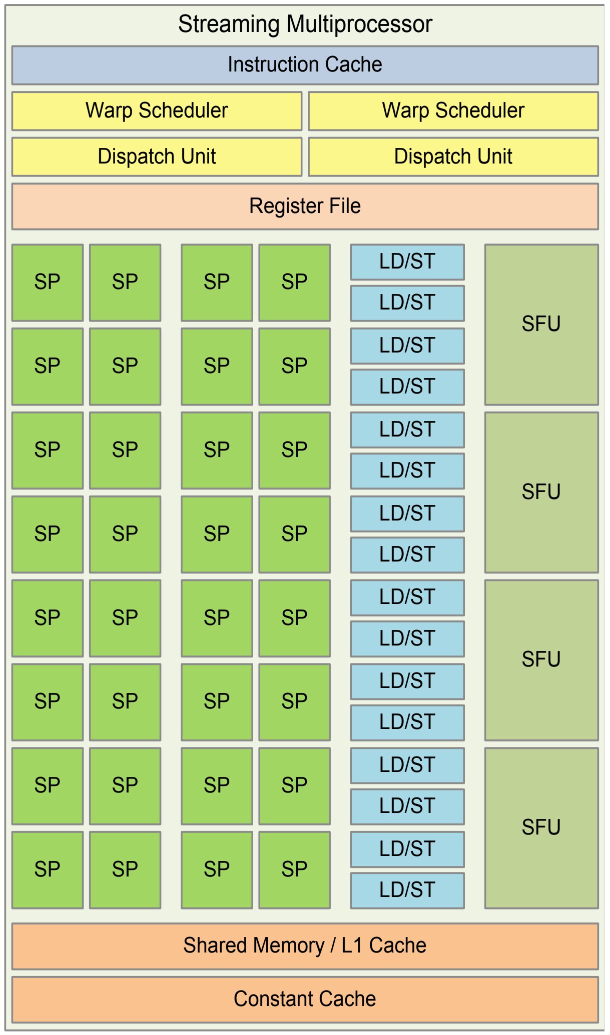
Slide credit: Hwu & Kirk

From Blocks to Warps

- GPU core: A SIMD pipeline
  - Streaming Processor (SP)
  - Many such SIMD Processors
    - Streaming Multiprocessor (SM)

- Blocks are divided into warps
  - SIMD/SIMT unit (32 threads)





NVIDIA Fermi architecture

Threads and Blocks

- Thread：一个线程（Thread）对应一个数据元素
- Block：大量的线程组织成很多线程块（Thread Block）
- Grid：许多线程块组成一个网格
- GPU 由硬件对线程进行管理
  - Thread Block Scheduler
  - SIMD Thread Scheduler
  - Warp
    - SIMD 线程
    - 线程调度的基本单位

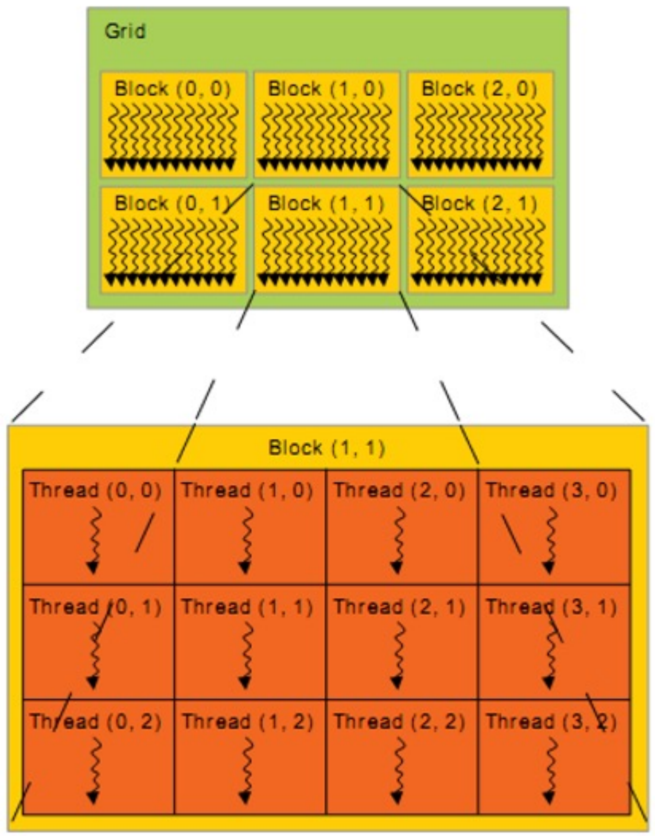
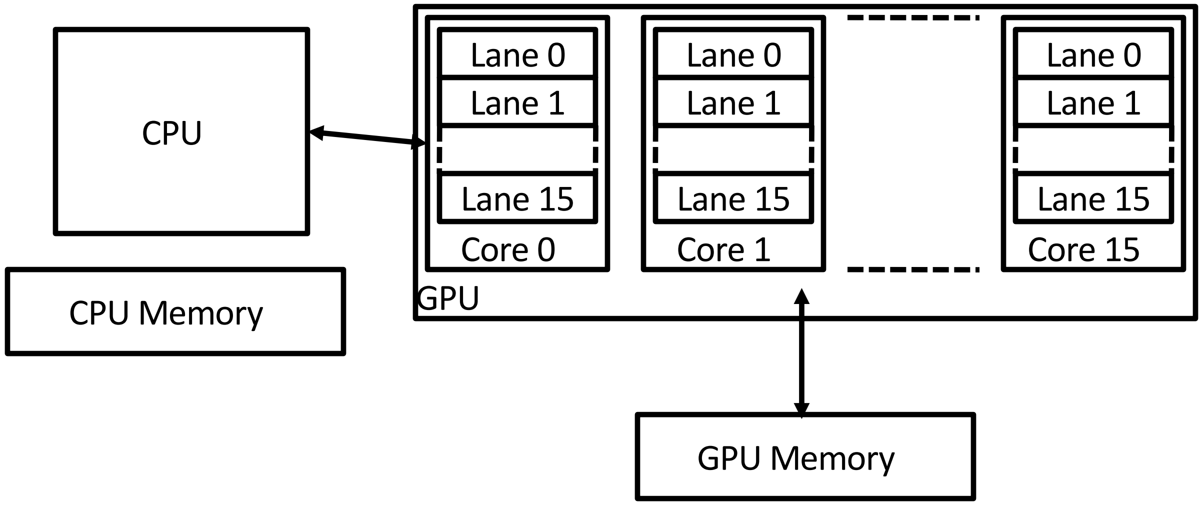


Figure 6 Grid of Thread Blocks

Hardware Execution Model

- A GPU has multiple GPU cores
  - 每个核是一个多线程SIMD 处理器（包含多个车道（Lanes））
- CPU 发送整个Grid到GPU，GPU调度这些Thread blocks (线程块) 分发到多个核上（每个Thread block (线程块) 在一个核上运行）
  - GPU上核的数量对程序员而言是透明的



Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Sequential instruction execution; lock-step operations in a SIMD instruction
  - 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 → SIMD operations can be formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware

SPMD

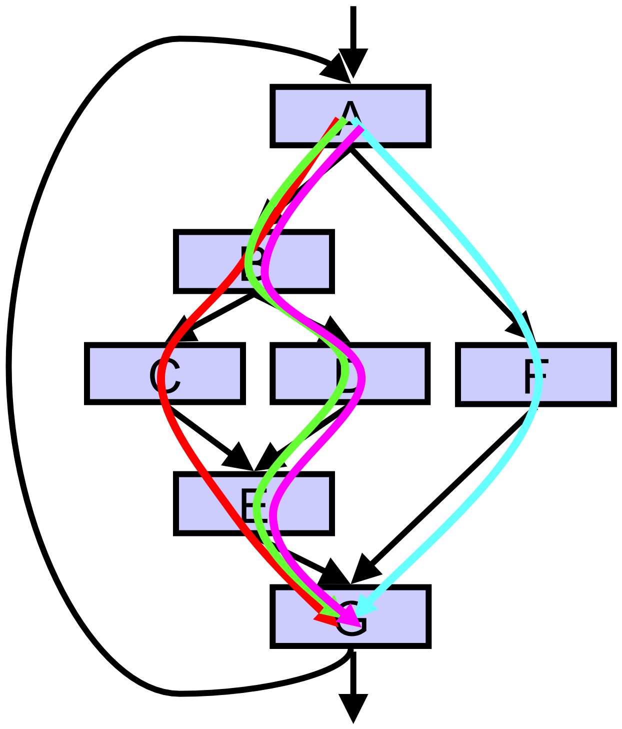
- Single procedure/program, multiple data
  - This is a programming model rather than computer organization
- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers
- Essentially, multiple instruction 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

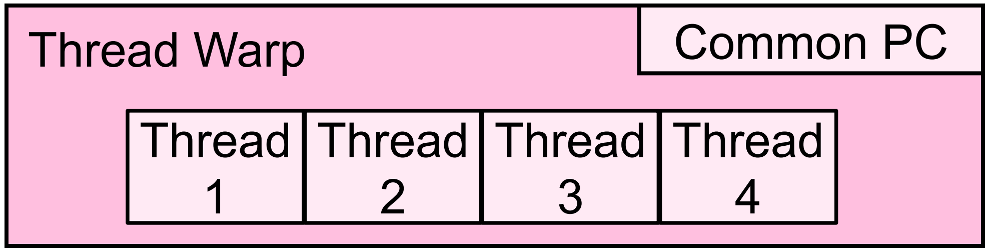
SIMD vs. SIMT Execution Model

- 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

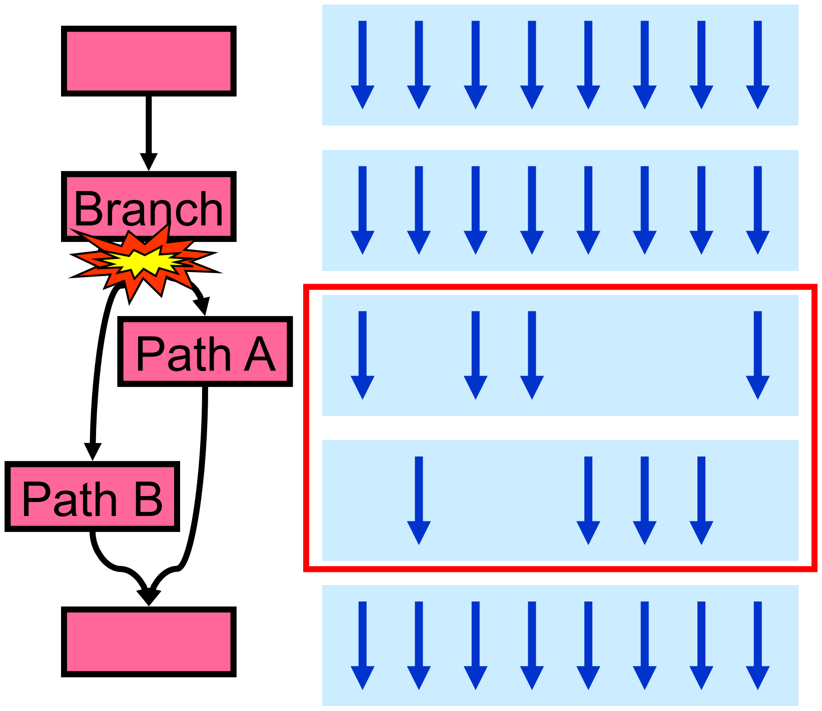
- Each thread can have conditional control flow instructions
- Threads can execute different control flow paths





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



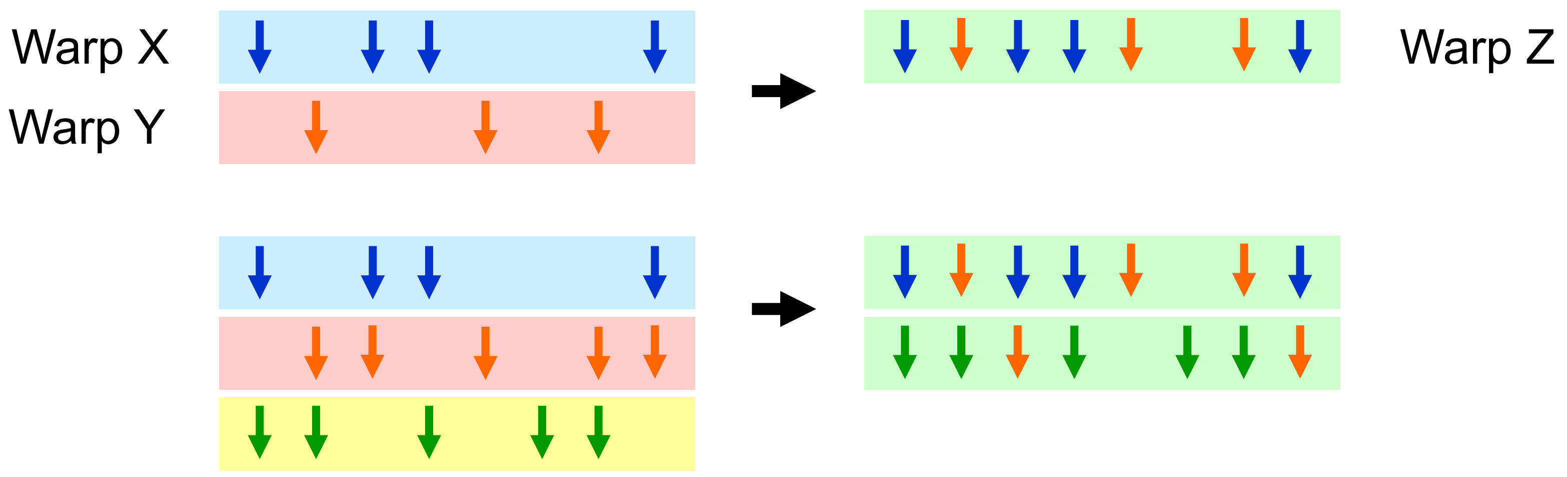
This is the same as conditional/predicated/masked execution. Recall the Vector Mask and Masked Vector Operations

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)

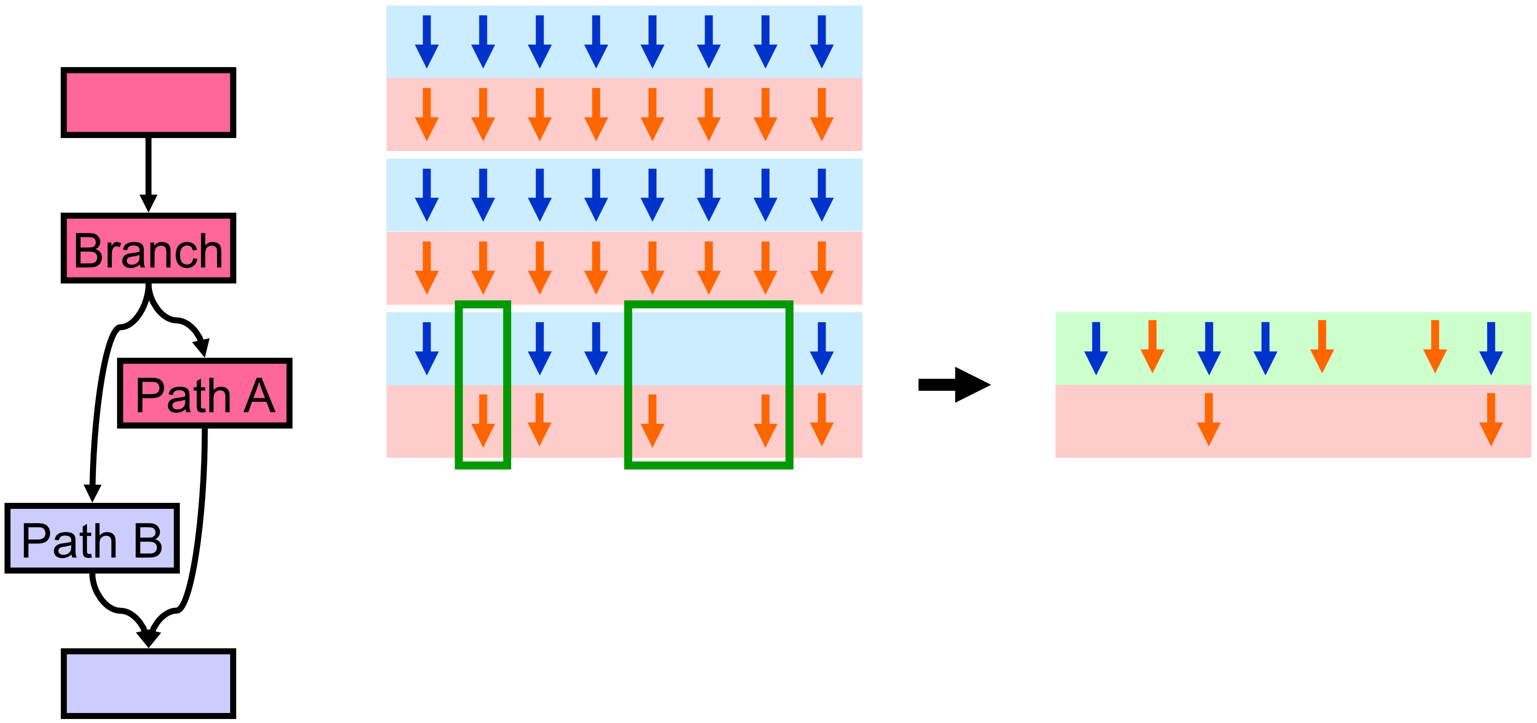
Dynamic Warp Formation/Merging

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



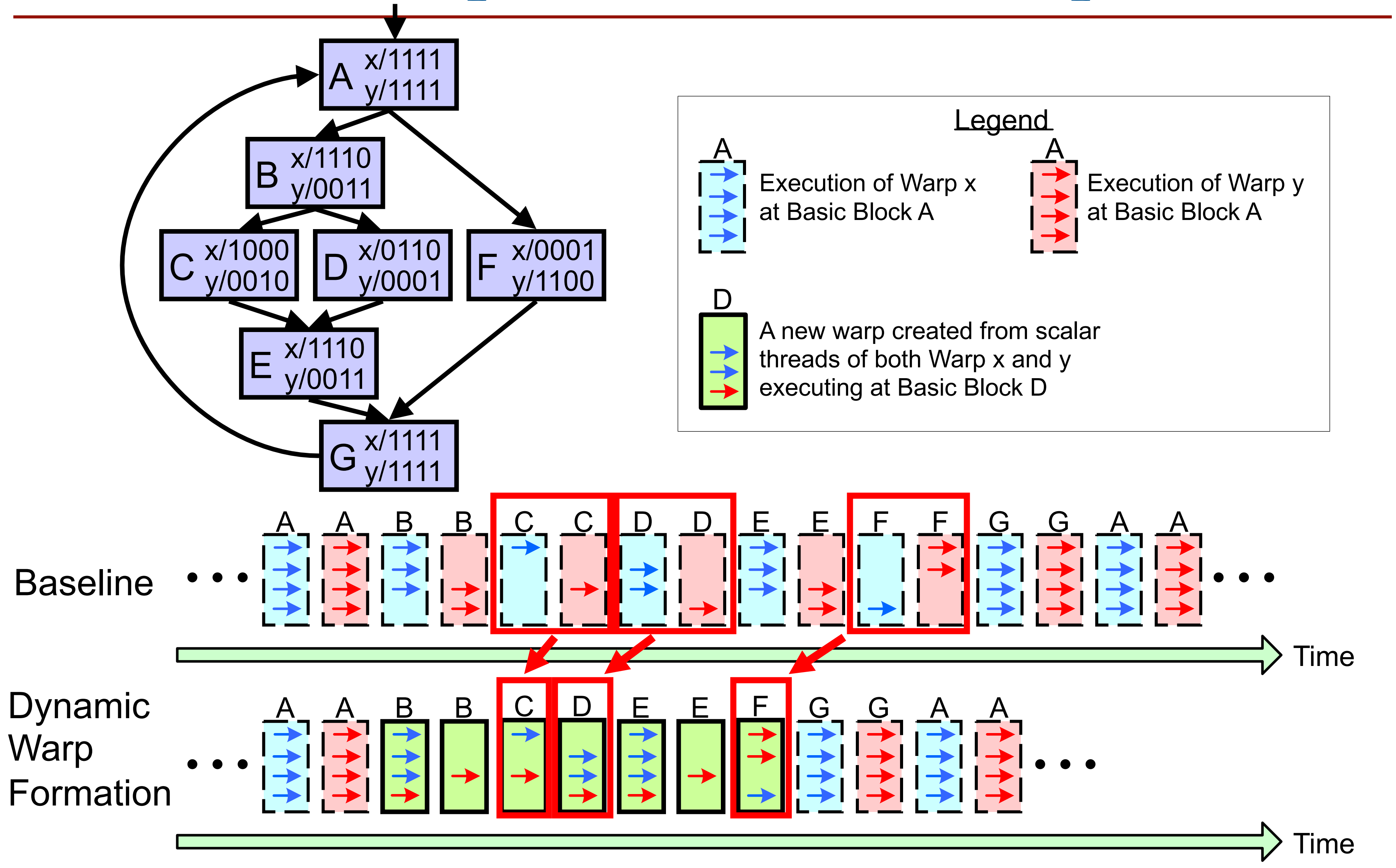
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction, i.e., at the same PC (after branch divergence)

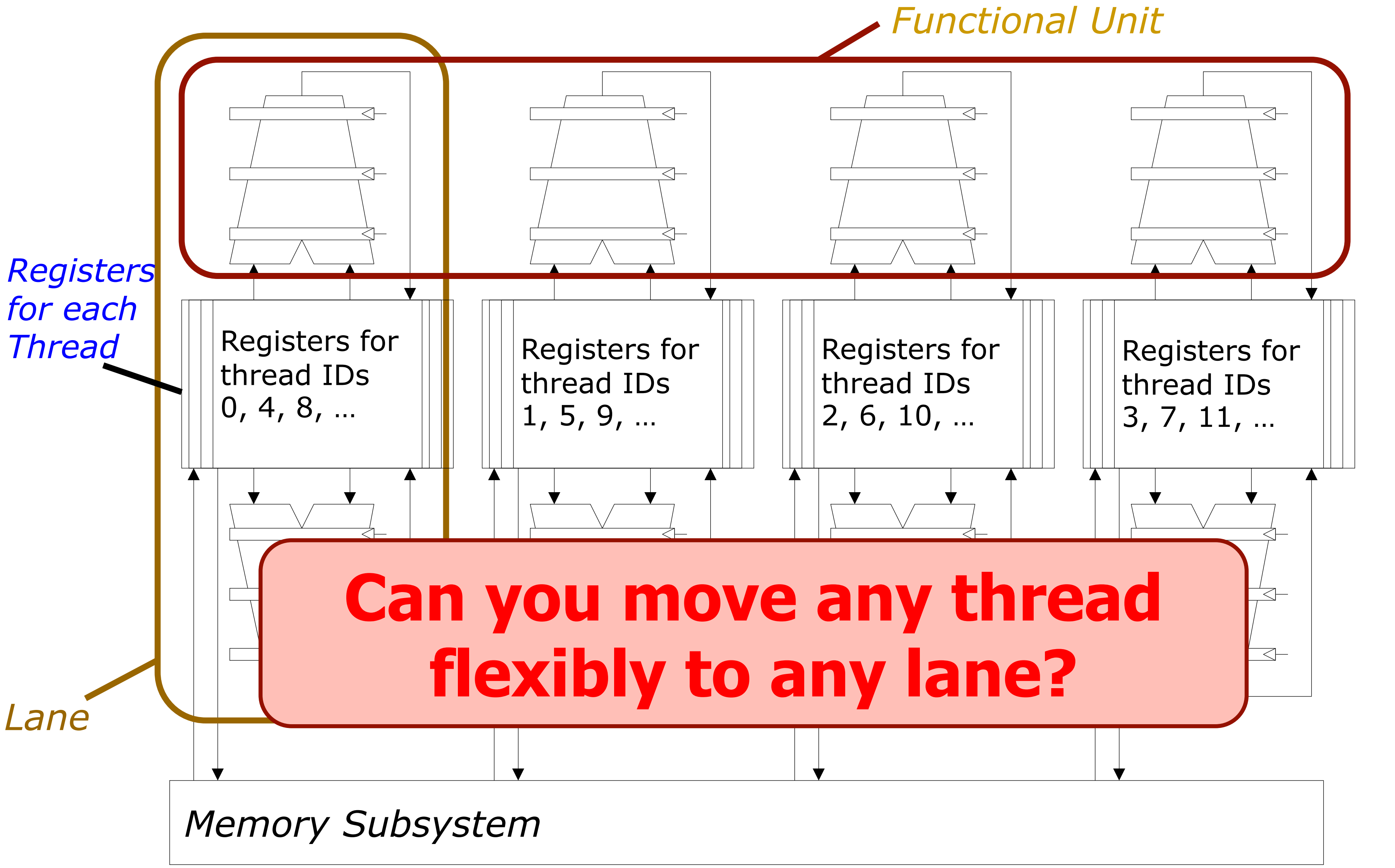


  - Fung et al., “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” MICRO 2007.

Dynamic Warp Formation Example


A x/1111 y/1111
B x/1110 y/0011
C x/1000 y/0010
D x/0110 y/0001
F x/0001 y/1100
E x/1110 y/0011
G x/1111 y/1111
Legend
Execution of Warp x at Basic Block A
Execution of Warp y at Basic Block A
A new warp created from scalar threads of both Warp x and y executing at Basic Block D
Baseline
A A B B C C D D E E F F G G A A
Time
Dynamic Warp Formation
A A B B C D E E F G G A A
Time

Hardware Constraints Limit Flexibility of Warp Grouping


Functional Unit
Registers for each Thread
Registers for thread IDs 0, 4, 8, …
Registers for thread IDs 1, 5, 9, …
Registers for thread IDs 2, 6, 10, …
Registers for thread IDs 3, 7, 11, …
Lane
Memory Subsystem

Can you move any thread flexibly to any lane?

Slide credit: Krste Asanovic

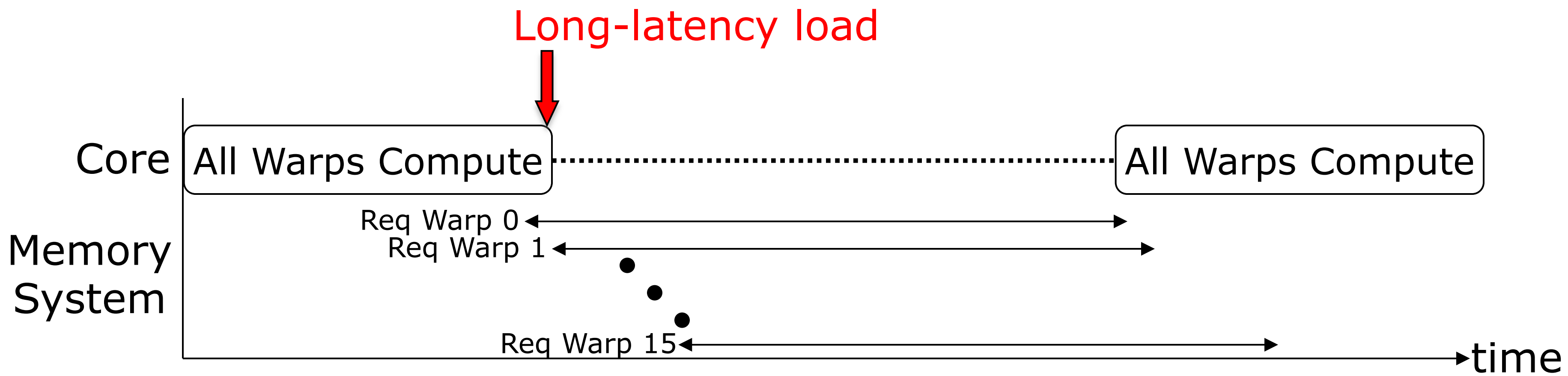
Large Warps and Two-Level Warp Scheduling (II)

- Two main reasons for GPU resources be underutilized
  - Branch divergence
  - Long latency operations

Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.

Large Warps and Two-Level Warp Scheduling

- Two main reasons for GPU resources be underutilized
  - Branch divergence
  - Long latency operations

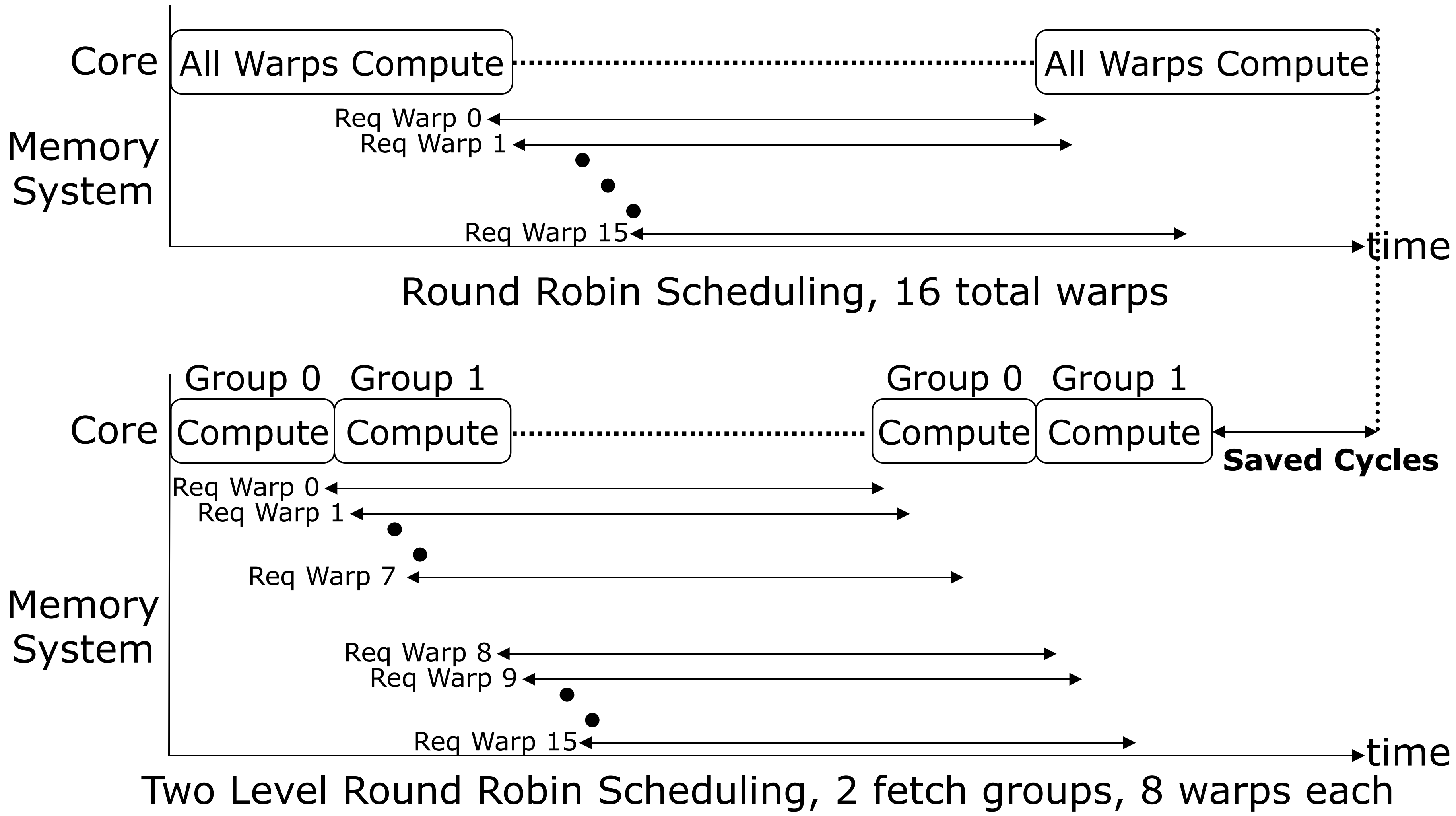


Round Robin Scheduling, 16 total warps

Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.

Two-Level Scheduling of Warps

- Scheduling smaller warp groups reduces stalls due to long latency operations



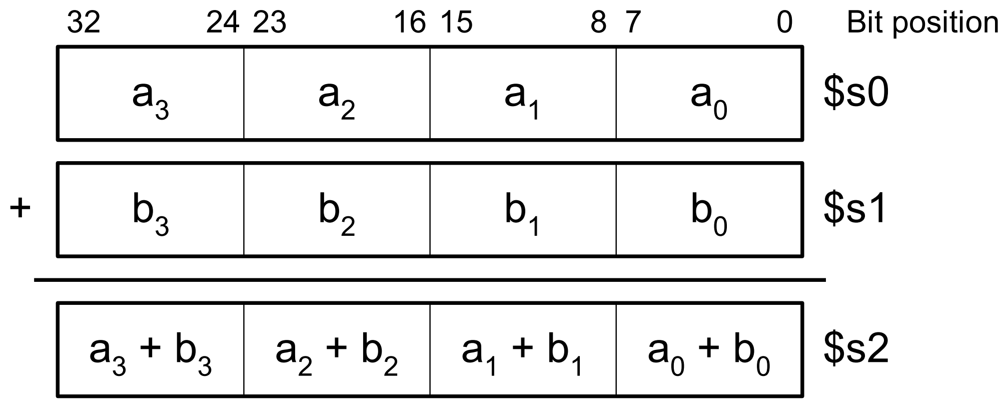
Narasiman et al., "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," MICRO 2011.

SIMD Operations in Modern ISAs

SIMD ISA Extensions

- Single Instruction Multiple Data (SIMD) extension instructions
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics, multimedia, image processing
  - Perform short arithmetic operations (also called packed arithmetic)
- For example: add four 8-bit numbers
- Must modify ALU to eliminate carries between 8-bit values

```
padd8 $s2, $s0, $s1
```



Intel Pentium MMX Operations

- Idea: One instruction operates on multiple data elements simultaneously
  - array processing (yet much more limited)
  - Designed with multimedia (graphics) operations in mind

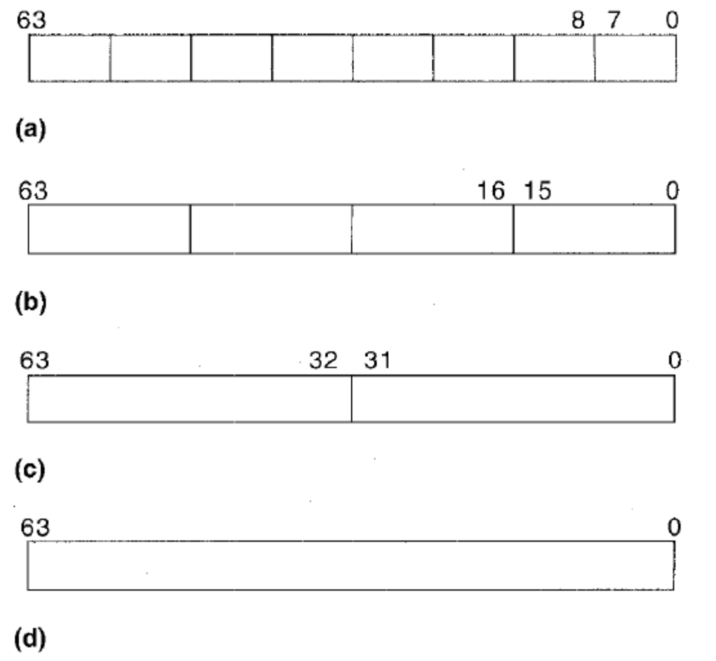


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

No VLEN register
Opcode determines data type:
8 8-bit bytes
4 16-bit words
2 32-bit doublewords
1 64-bit quadword

Stride is always equal to 1.

Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, 1996.

Intel Pentium MMX Operations (II)

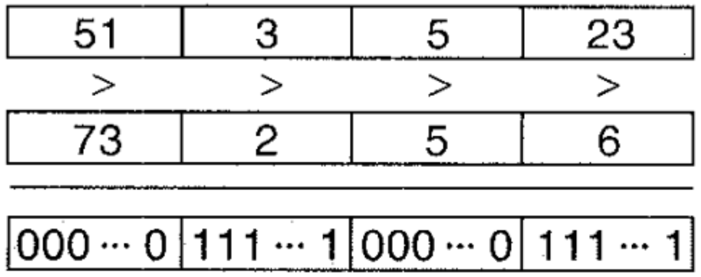


Figure 3. Packed compare greater-than word.

PCMPEQ(b,w,d), PCMPGT(b,w,d)	Equal or greater than	1	Compares packed 8 bytes, four 16-bit words, or two 32-bit elements in parallel. Result is mask of 1s if true or 0s if false.

Intel Pentium MMX Operations (II)

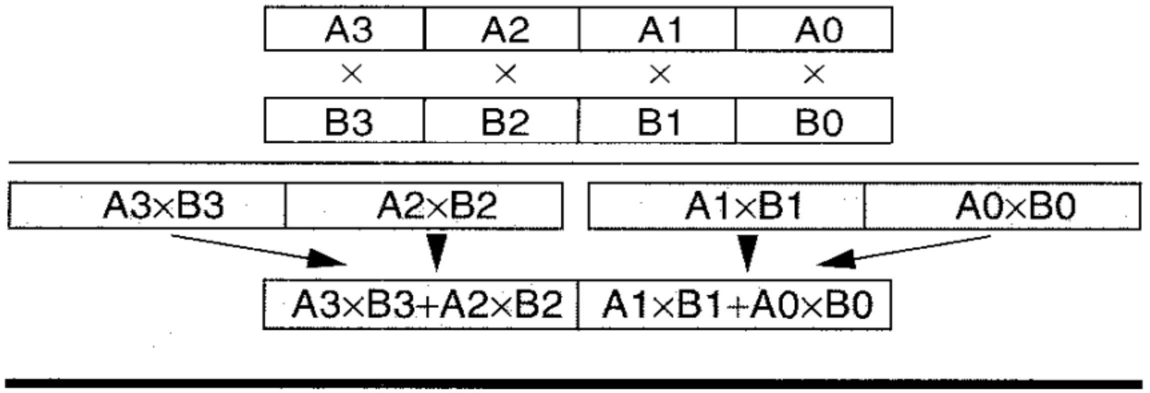


Figure 2. Packed multiply-add word to doubleword.

PMADDWD	Word to doubleword conversion	Latency: 3; throughput: 1	Multiplies four packed, signed 16-bit words and adds together adjacent pairs of 32-bit results in parallel. Result is a doubleword.

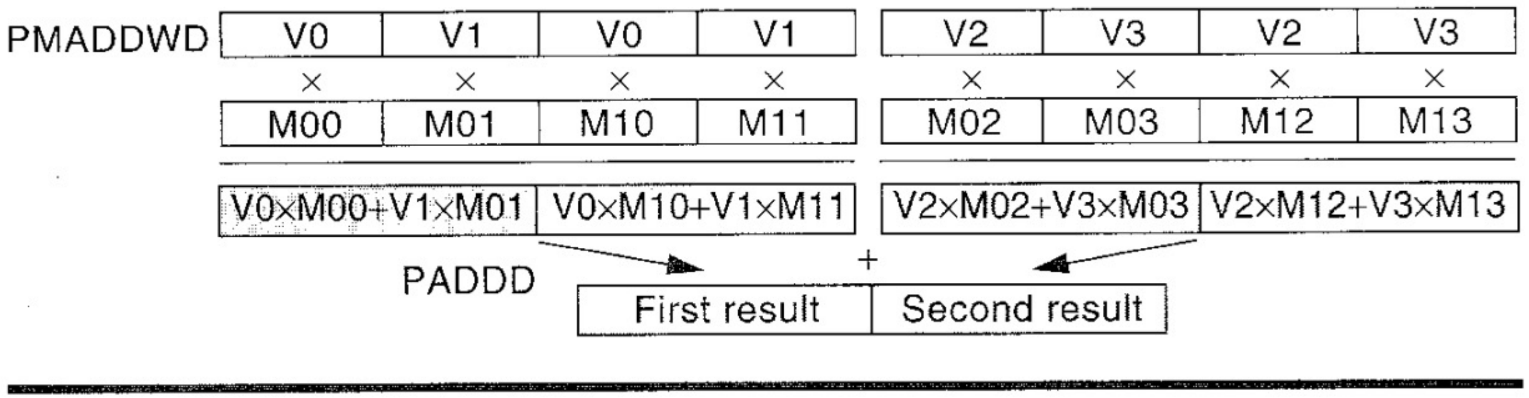


Figure 7. Flow diagram of matrix-vector multiply.

MMX Example: Image Overlaying (I)

- Goal: Overlay the human in image x on top of the background in image y

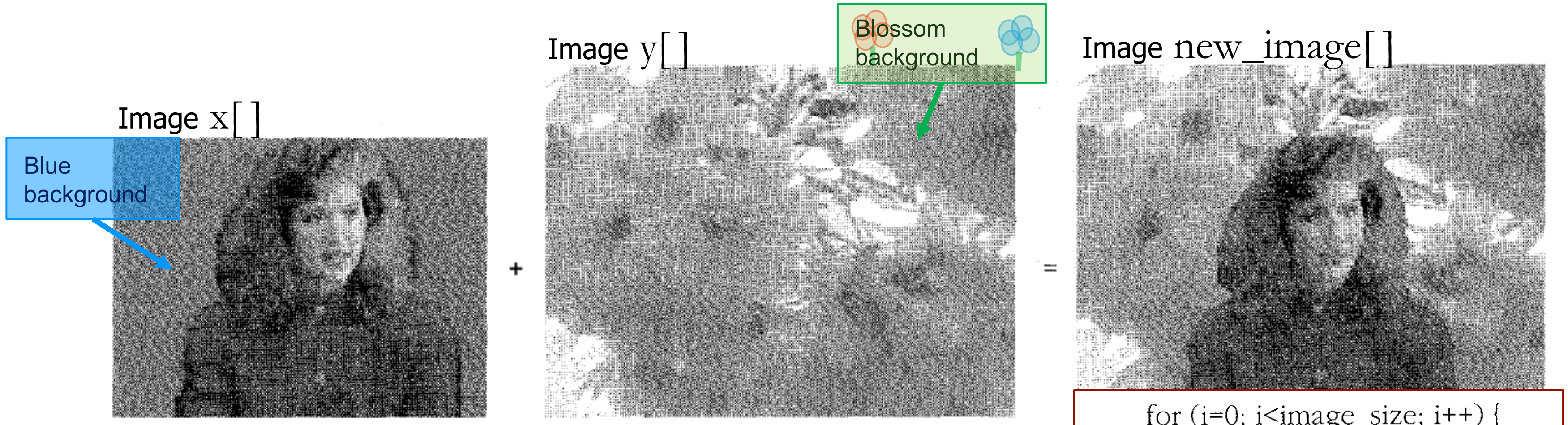
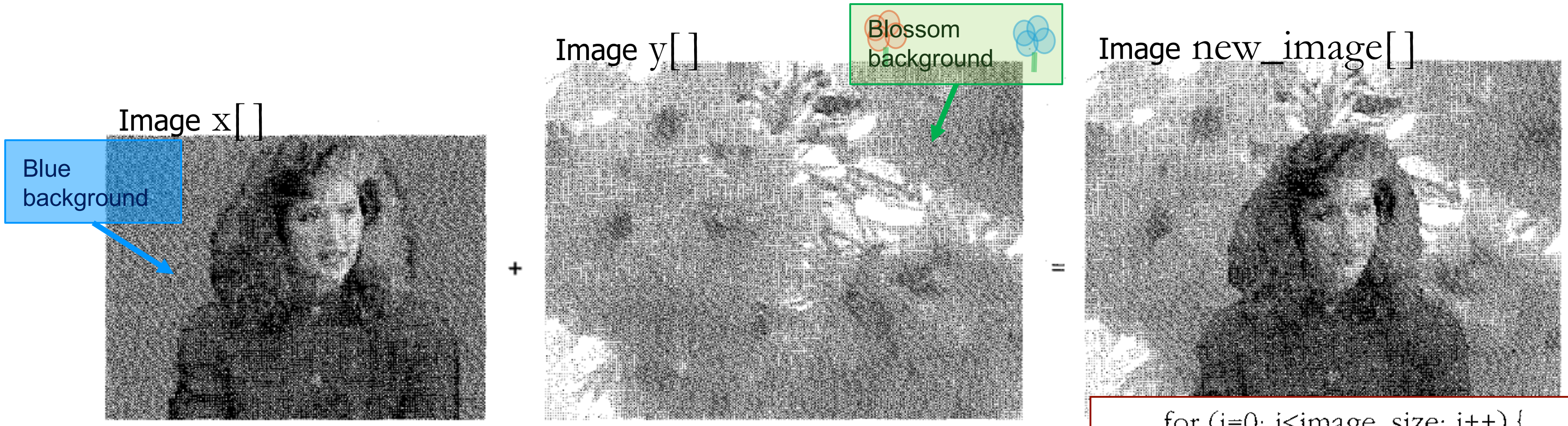


Figure 8. Chroma keying: image overlay using a background color.

```
for (i=0; i<image_size; i++) {
if (x[i] == Blue)  new_image[i] =y[i];
        else new_image[i] = x[i];
    }
```

MMX Example: Image Overlaying (I)

- Goal: Overlay the human in image x on top of the background in image y



```
for (i=0; i<image_size; i++) {
if (x[i] == Blue)  new_image[i] =y[i];
        else new_image[i] = x[i];
    }
```

Figure 8. Chroma keying: image overlay using a background color.

PCMPEQB MM1, MM3

	MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
Image x[ ]	MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
Bit mask	MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF

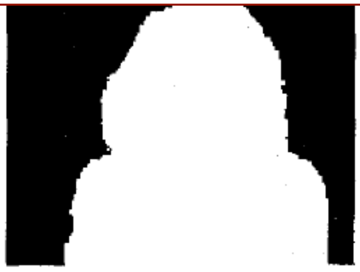

Bitmask

Figure 9. Generating the selection bit mask.

MMX Example: Image Overlaying (II)

- Goal: Overlay the human in image x on top of the background in image y

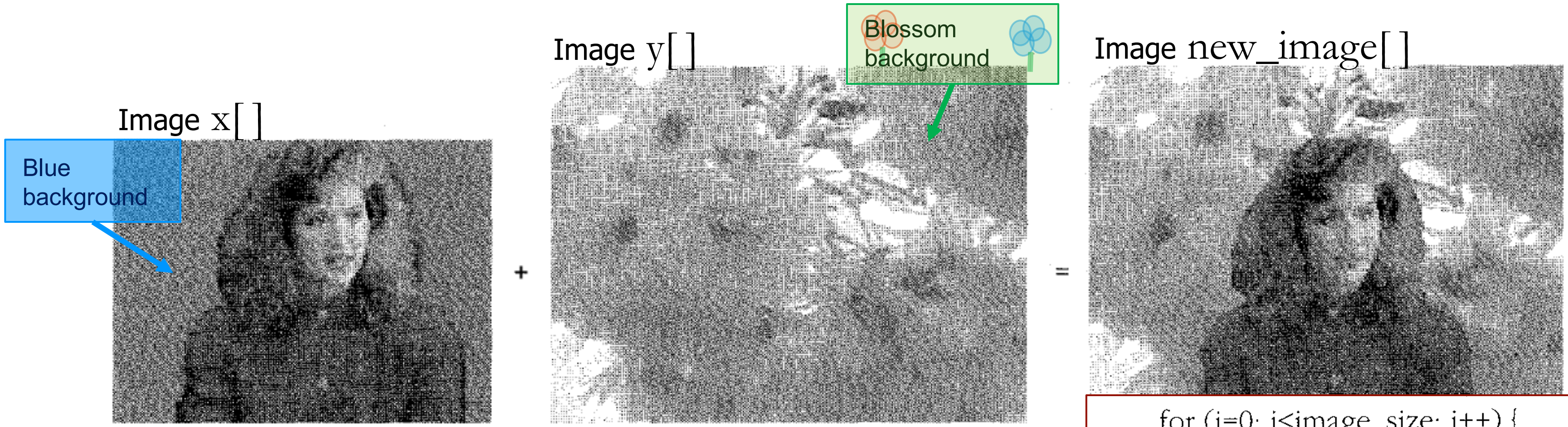
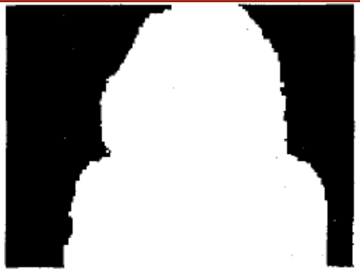


Figure 8. Chroma keying: image overlay using a background color.

```
for (i=0; i<image_size; i++) {
if (x[i] == Blue)   new_image[i] =y[i];
        else new_image[i] = x[i];
}
```

PCMPEQB MM1, MM3

	MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
Image x[ ]	MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
Bit mask	MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF

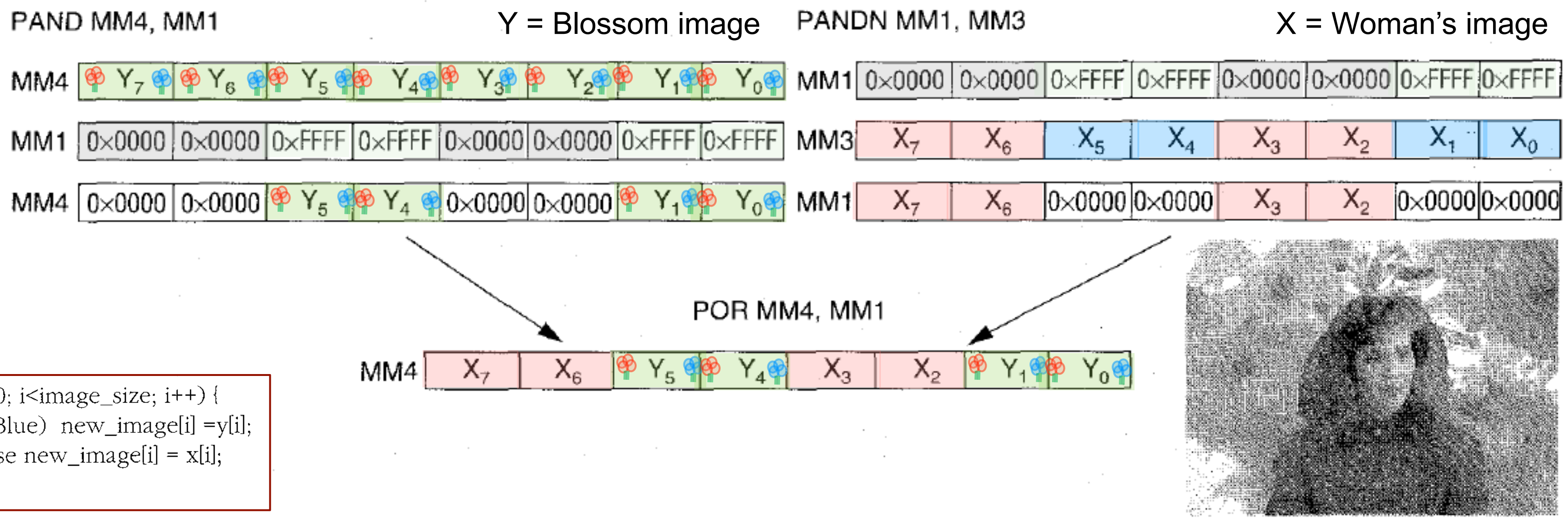


Bitmask

Figure 9. Generating the selection bit mask.

Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, 1996.

MMX Example: Image Overlaying (III)


PAND MM4, MM1
Y = Blossom image
PANDN MM1, MM3
X = Woman's image
MM4 | Y_7 | Y_6 | Y_5 | Y_4 | Y_3 | Y_2 | Y_1 | Y_0
MM1 | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF
MM4 | 0x0000 | 0x0000 | Y_5 | Y_4 | 0x0000 | 0x0000 | Y_1 | Y_0
MM1 | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF | 0x0000 | 0x0000 | 0xFFFF | 0xFFFF
MM3 | X_7 | X_6 | X_5 | X_4 | X_3 | X_2 | X_1 | X_0
MM1 | X_7 | X_6 | 0x0000 | 0x0000 | X_3 | X_2 | 0x0000 | 0x0000
POR MM4, MM1
MM4 | X_7 | X_6 | Y_5 | Y_4 | X_3 | X_2 | Y_1 | Y_0

```
for (i=0; i<image_size; i++) {
if (x[i] == Blue)  new_image[i] =y[i];
        else new_image[i] = x[i];
}
```

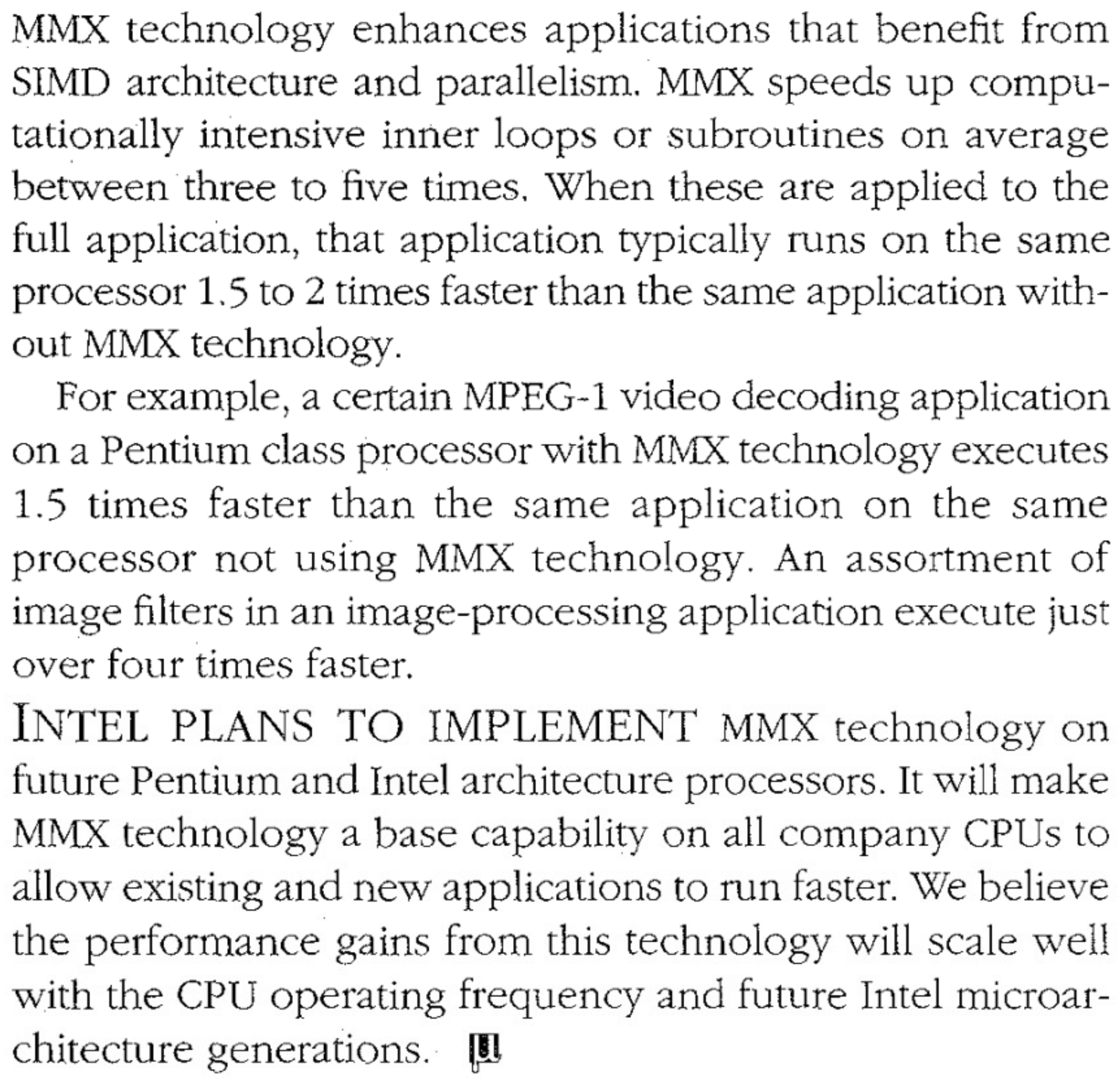
Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```
Movq      mm3, mem1    /* Load eight pixels from
                          woman's image
Movq      mm4, mem2    /* Load eight pixels from the
                          blossom image
Pcmpeqb   mm1, mm3
Pand      mm4, mm1
Pandn     mm1, mm3
Por       mm4, mm1
```

Figure 11. MMX code sequence for performing a conditional select.

Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, 1996.

Intel Pentium MMX Operations



MMX technology enhances applications that benefit from SIMD architecture and parallelism. MMX speeds up computationally intensive inner loops or subroutines on average between three to five times. When these are applied to the full application, that application typically runs on the same processor 1.5 to 2 times faster than the same application without MMX technology.

For example, a certain MPEG-1 video decoding application on a Pentium class processor with MMX technology executes 1.5 times faster than the same application on the same processor not using MMX technology. An assortment of image filters in an image-processing application execute just over four times faster.

INTEL PLANS TO IMPLEMENT MMX technology on future Pentium and Intel architecture processors. It will make MMX technology a base capability on all company CPUs to allow existing and new applications to run faster. We believe the performance gains from this technology will scale well with the CPU operating frequency and future Intel microarchitecture generations.

Acknowledgements

- EPFL, Onur Mutlu, Digital Design and Computer Architecture, 2020, 2023
- Utah University, Rajeev, CS/ECE 6810, Computer Architecture
- 计算机体系结构：量化研究方法（中文版第六版）
- UCSD CSE 240
- Washington University, CSE3 78
- 国科大，胡伟武，计算机体系结构
- CMU, CS 447 Computer Architecture
- UC Berkeley, CS 152 and CS 61C
- 国科大南京学院，安学军等，计算机体系结构
- 浙江大学，计算机体系结构
- 南京大学，计算机体系结构