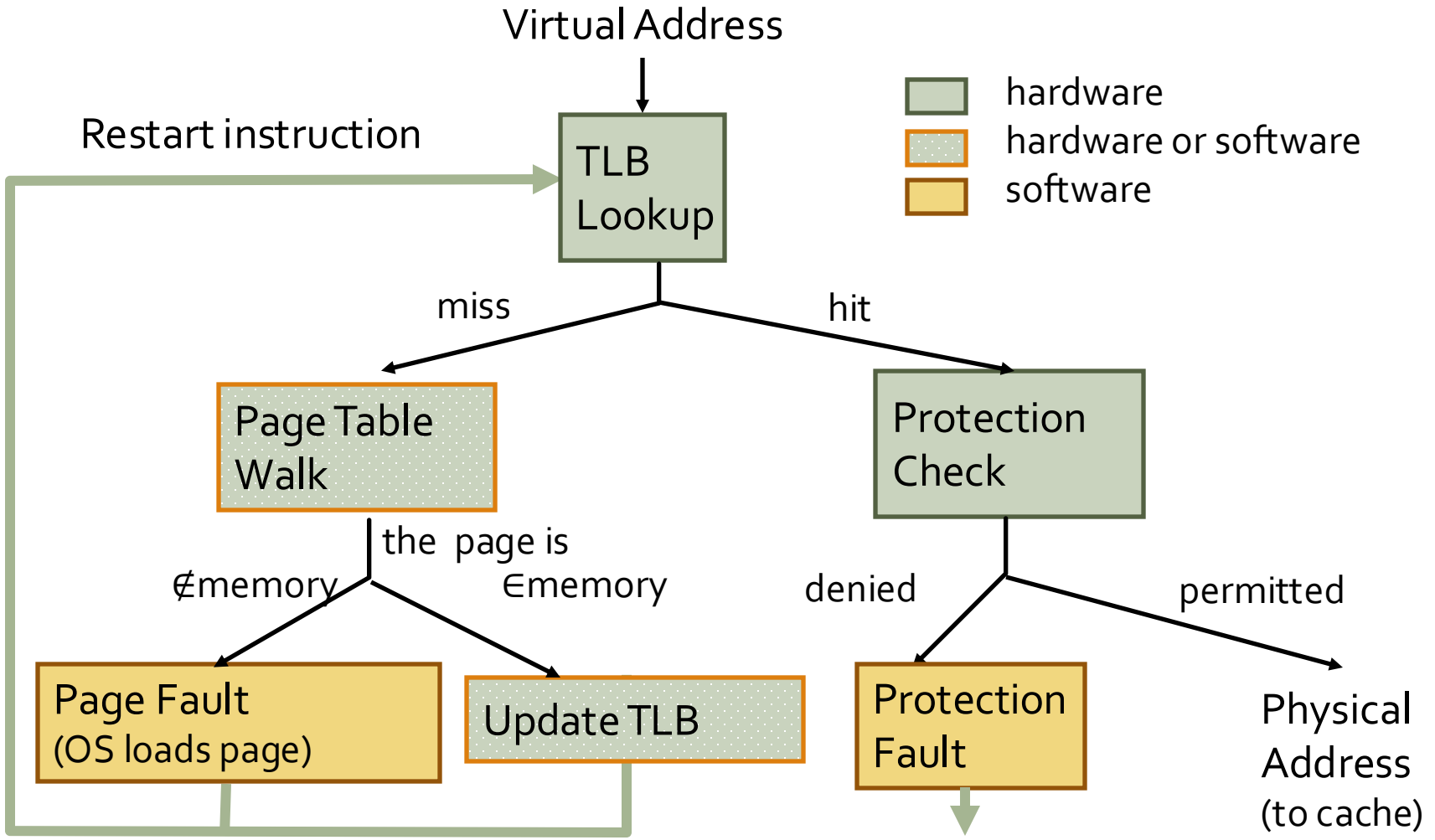
Computer Architecture

Vector Processors and GPUs

Ting-Jung Chang

NYCU CS

Recap: Memory Management



Data Parallelism

- Concurrency arises from performing the same operation on different pieces of data
  - Single instruction multiple data (SIMD)
  - E.g., dot product of two vectors

- SIMD exploits operation-level parallelism on different data
  - Same operation concurrently applied to different pieces of data
  - A form of ILP where instruction happens to be the same across data

Agenda

- Vector Processors
- Single Instruction Multiple Data (SIMD) Instruction Set Extensions
- Graphics Processing Units (GPU)

Supercomputers

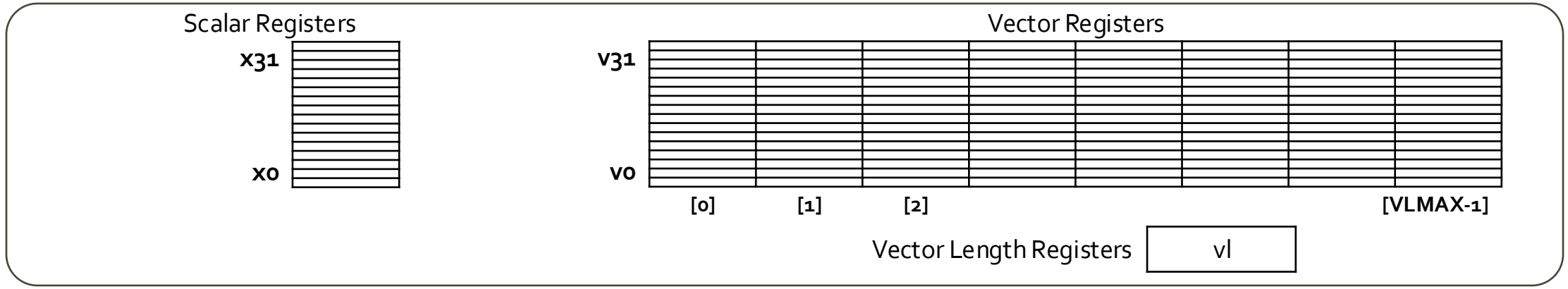
- Definition of a supercomputer:
- Fastest machine in the world at given task
- A device to turn a compute-bound problem into an I/O bound problem
- Any machine costing $30M+
- Any machine designed by Seymour Cray

CDC6600 (Cray, 1964) regarded as first supercomputer

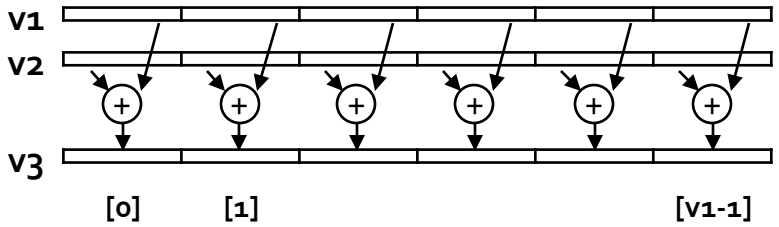
Supercomputer Applications

- Typical application areas
  - Military research (nuclear weapons, cryptography)
  - Scientific research
  - Weather forecasting
  - Oil exploration
  - Industrial design (car crash simulation)
  - Bioinformatics
  - Cryptography
- All involve huge computations on large data set
- Supercomputers: CDC6600, CDC7600, Cray-1, …
- In 70s-80s, Supercomputer ≡ Vector Machine

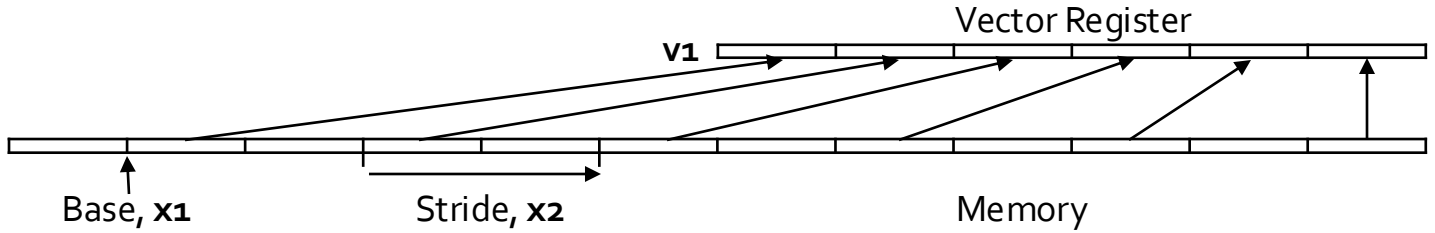
Vector Programming Model



Vector Arithmetic Instructions
vadd v3, v1, v2



Vector Load and Store Instructions
vls v1, (x1), x2



Vector Code Element-by-Element Multiplication

# C code	# Scalar Assembly Code	# Vector Assembly Code

`for (i=0; i<64; i++)`
`  C[i] = A[i] * B[i];`
	`  li x4, 64`
	`loop:`
	`  fld f1, 0(x1)`
	`  fld f2, 0(x2)`
	`  fmul.d f3,f1,f2`
	`  fsd f3, 0(x3)`
	`  addi x1, x1, 8`
	`  addi x2, x2, 8`
	`  addi x3, x3, 8`
	`  subi x4, x4, 1`
	`  bnez x4, loop`
		`  li x4, 64`
		`  vsetvl x4`
		`  vld v1, (x1)`
		`  vld v2, (x2)`
		`  vmul v3,v1,v2`
		`  vst v3, (x3)`

How many instructions?	1+9*64 instrs	5 instrs

Vector ISA Attributes

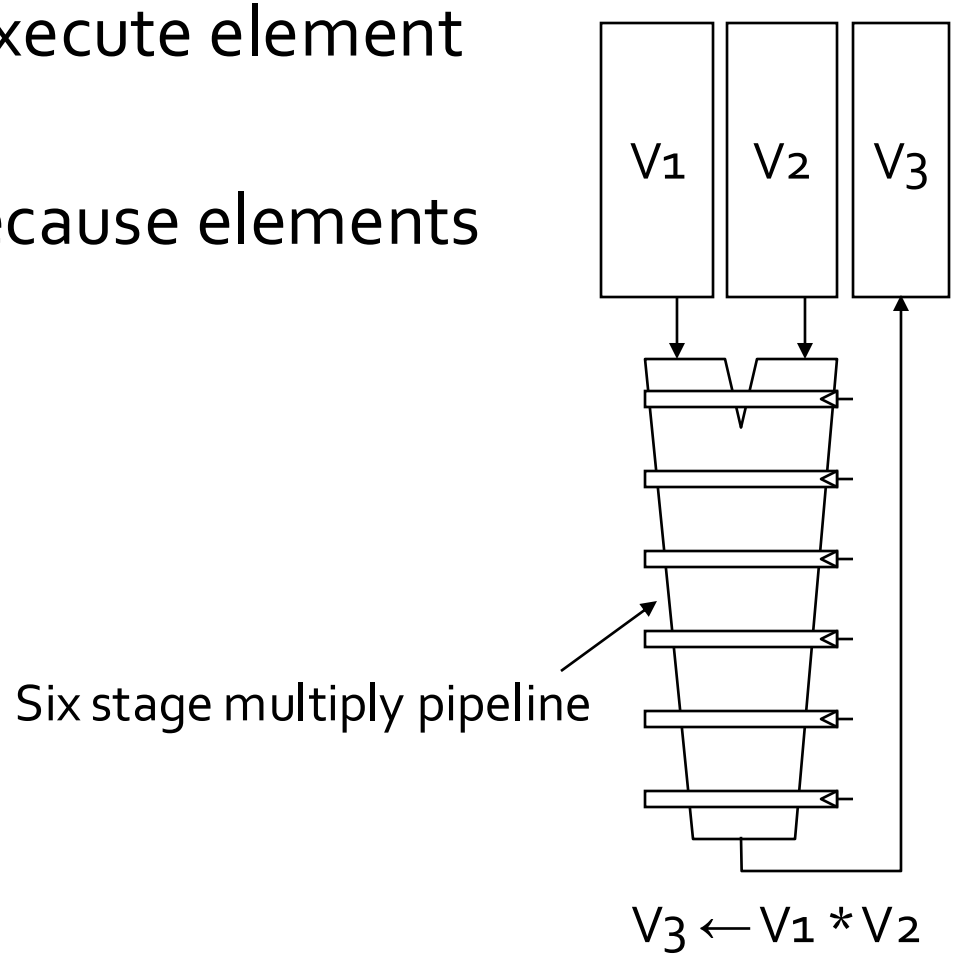
- Compact
  - one short instruction encodes N operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers (within one vector register)
  - access registers in same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)
- Scalable
  - can run same code on more parallel pipelines (lanes)

Vector ISA Hardware Implications

- Large amount of work per instruction
  - → Less instruction fetch bandwidth requirements
  - → Allows simplified instruction fetch design
- No data dependence within a vector
  - → Amenable to deeply pipelined/parallel designs
- Disjoint vector element accesses
  - → Banked rather than multi-ported register files
- Known regular memory access pattern
  - → Allows for banked memory for higher bandwidth

Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent
  - no data hazards!
  - no bypassing needed

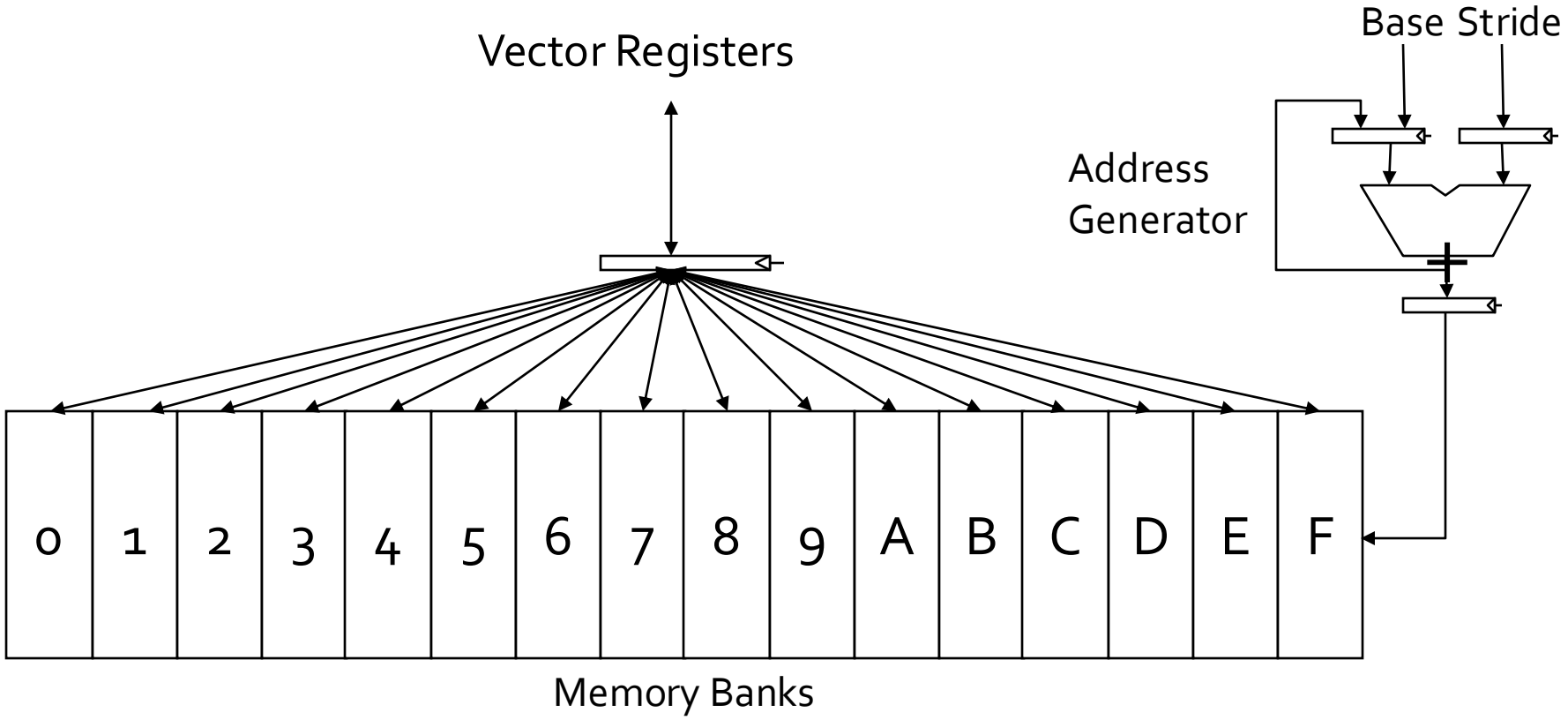


Given 64-element registers, how long does it take to compute V3?

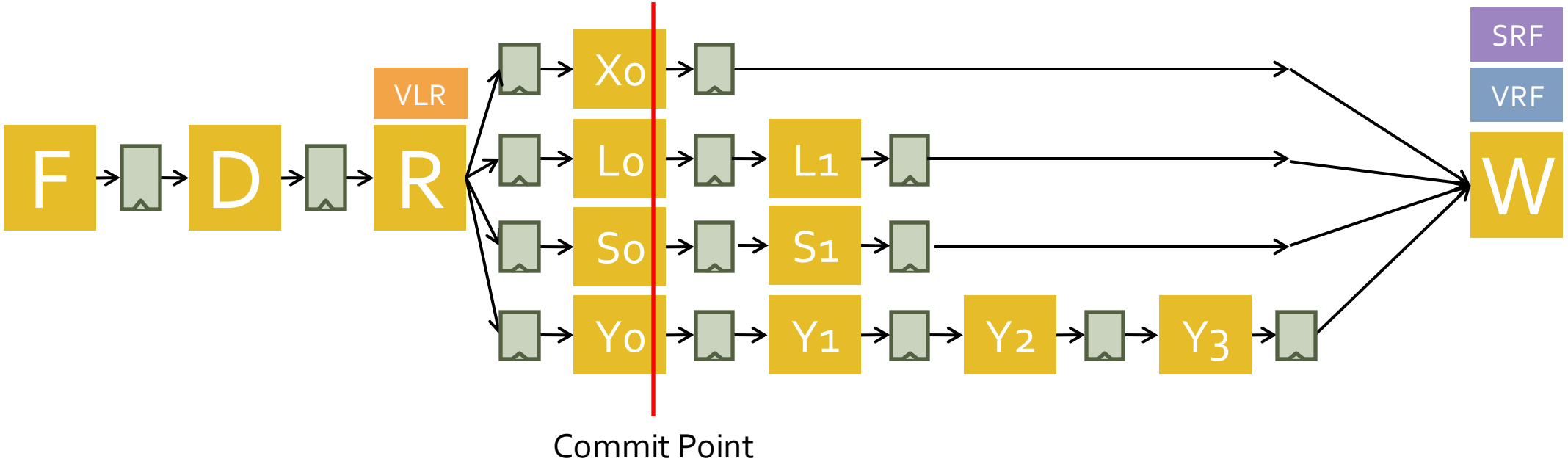
6+63 cycles

Interleaved Vector Memory System

- Bank busy time: Time before bank ready to accept next request
- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency



Example Vector Microarchitecture



Basic Vector Execution

 # C code

```
for (i=0; i<4; i++)
  C[i] = A[i] * B[i];
```

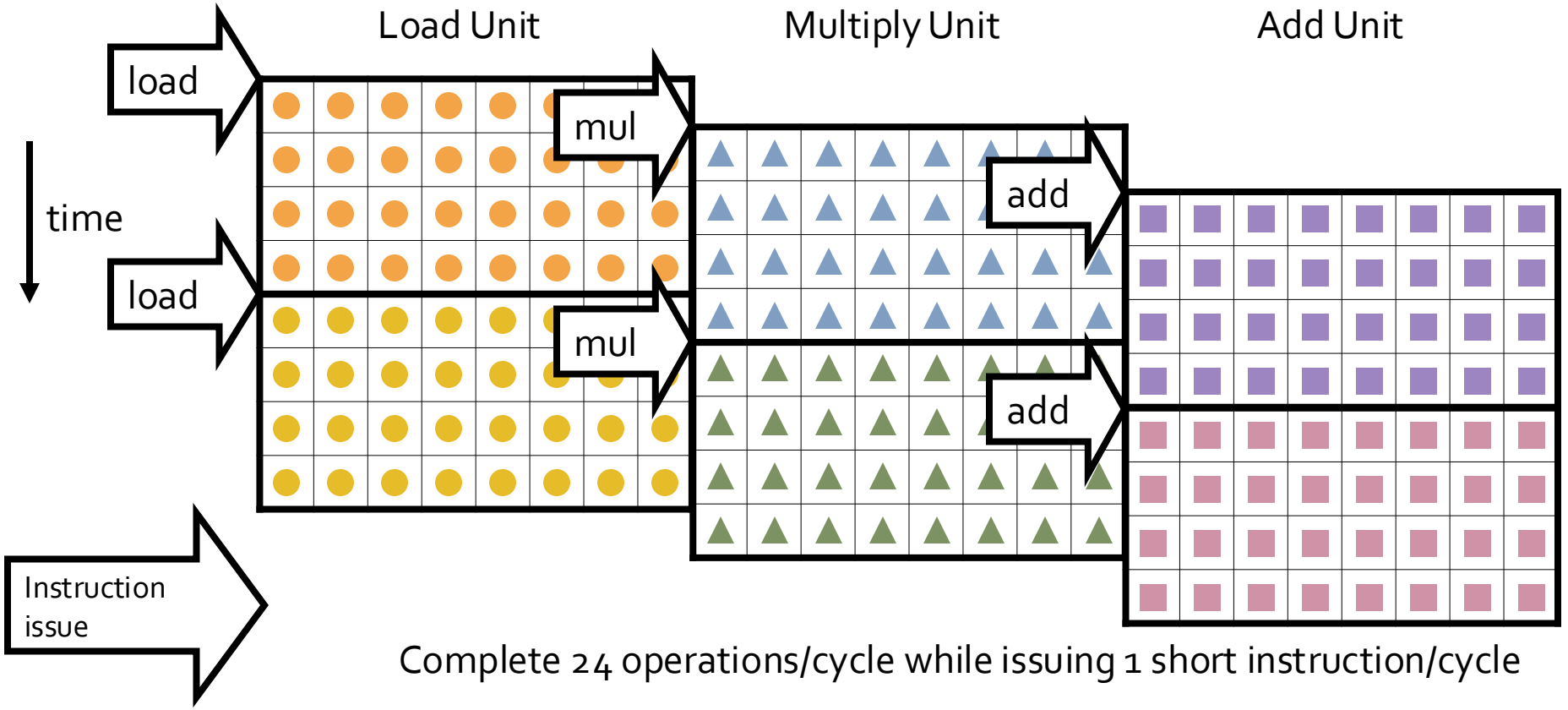
 # Vector Assembly Code

```
  li x4, 4
  vsetvl x4
  vld v1, (x1)
  vld v2, (x2)
  vmul v3,v1,v2
  vst v3, (x3)
```

```
v1= 4
vld     v2, (x2)      F  D  R  L0 L1 W
                         R  L0 L1 W
                            R  L0 L1 W
                               R  L0 L1 W
vmul    v3,v1,v2         F  D  D  D  D  D  D  D  R  Y0 Y1 Y2 Y3 W
                                                  R  Y0 Y1 Y2 Y3 W
                                                     R  Y0 Y1 Y2 Y3 W
                                                        R  Y0 Y1 Y2 Y3 W
vst     v3, (x3)            F  F  F  F  F  F  F  D  D  D  D  D  D  D  D  D  R  S0 S1 W
                                                                           R  S0 S1 W
                                                                              R  S0 S1 W
                                                                                 R  S0 S1 W
```

Vector Instruction Parallelism

- Can overlap execution of multiple vector instructions
  - example machine has 32 elements per vector register and 8 lanes

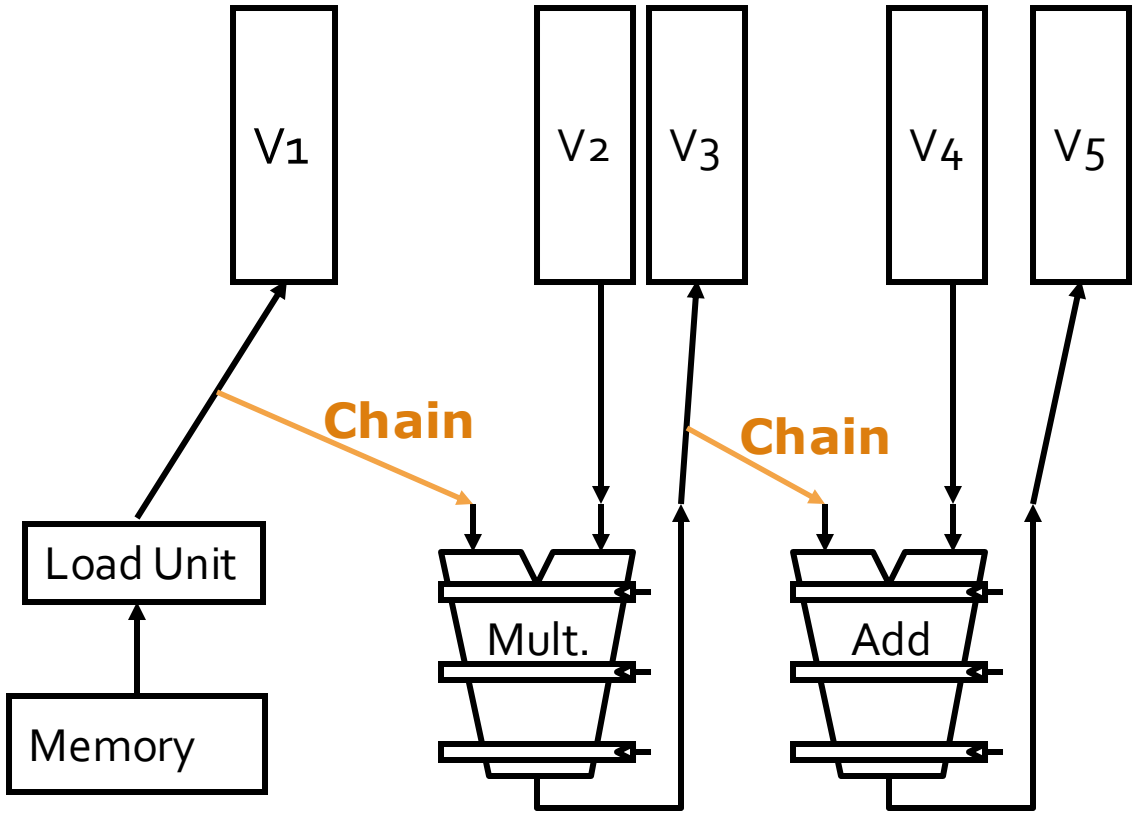


Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

- Vector version of register bypassing
  - introduced with Cray-1

```
vld    v1
vmul   v3,v1,v2
vadd   v5,v3,v4
```



Vector Chaining Advantage

- Without chaining, must wait for last element of result to be written before starting dependent instruction

- With chaining, can start dependent instruction as soon as first result appears

Chaining (Register File) Vector Execution

 # C code

```
for (i=0; i<4; i++)
  C[i] = A[i] * B[i];
```

 # Vector Assembly Code

```
li x4, 4
vsetvl x4
vld v1, (x1)
vld v2, (x2)
vmul v3,v1,v2
vst v3, (x3)
```

```
vl = 4
vld     v2, (x2)      F  D  R  L0 L1 W
                         R  L0 L1 W
                            R  L0 L1 W
                               R  L0 L1 W
vmul    v3,v1,v2         F  D  D  D  D  R  Y0 Y1 Y2 Y3 W
                                        R  Y0 Y1 Y2 Y3 W
                                           R  Y0 Y1 Y2 Y3 W
                                              R  Y0 Y1 Y2 Y3 W
vst     v3, (x3)            F  F  F  F  D  D  D  D  D  D  R  S0 S1 W
                                                          R  S0 S1 W
                                                             R  S0 S1 W
                                                                R  S0 S1 W
```

Chaining (Bypass Network) Vector Execution

 # C code

```
for (i=0; i<4; i++)
  C[i] = A[i] * B[i];
```

 # Vector Assembly Code

```
li x4, 4
vsetvl x4
vld v1, (x1)
vld v2, (x2)
vmul v3,v1,v2
vst v3, (x3)
```

```
vl = 4
vld     v2, (x2)      F  D  R  L0 L1 W
                         R  L0 L1 W
                            R  L0 L1 W
                               R  L0 L1 W
vmul    v3,v1,v2         F  D  D  D  D  R  Y0 Y1 Y2 Y3 W
                                        R  Y0 Y1 Y2 Y3 W
                                           R  Y0 Y1 Y2 Y3 W
                                              R  Y0 Y1 Y2 Y3 W
vst     v3, (x3)            F  F  F  F  D  D  D  D  D  D  R  S0 S1 W
                                                          R  S0 S1 W
                                                             R  S0 S1 W
                                                                R  S0 S1 W
```

Chaining (Bypass Network) Vector Execution and More RF Ports

 # C code

```
for (i=0; i<4; i++)
  C[i] = A[i] * B[i];
```

```
vl = 4
vld     v2, (x2)      F  D  R  L0 L1 W
                         R  L0 L1 W
                            R  L0 L1 W
                               R  L0 L1 W
vmul    v3,v1,v2         F  D  D  R  Y0 Y1 Y2 Y3 W
                               R  Y0 Y1 Y2 Y3 W
                                  R  Y0 Y1 Y2 Y3 W
                                     R  Y0 Y1 Y2 Y3 W
vst     v3, (x3)            F  F  D  D  D  D  R  S0 S1 W
                                              R  S0 S1 W
                                                 R  S0 S1 W
                                                    R  S0 S1 W
```

 # Vector Assembly Code

```
li x4, 4
vsetvl x4
vld v1, (x1)
vld v2, (x2)
vmul v3,v1,v2
vst v3, (x3)
```

Chaining (Bypass Network) Vector Execution and More RF Ports

```
v1= 8
vld     v2, (x2)     F  D  R  L0 L1 W
                        R  L0 L1 W
                           R  L0 L1 W
                              R  L0 L1 W
                                 R  L0 L1 W
                                    R  L0 L1 W
                                       R  L0 L1 W
                                          R  L0 L1 W
vmul    v3,v1,v2        F  D  D  R  Y0 Y1 Y2 Y3 W
                              R  Y0 Y1 Y2 Y3 W
                                 R  Y0 Y1 Y2 Y3 W
                                    R  Y0 Y1 Y2 Y3 W
                                       R  Y0 Y1 Y2 Y3 W
                                          R  Y0 Y1 Y2 Y3 W
                                             R  Y0 Y1 Y2 Y3 W
                                                R  Y0 Y1 Y2 Y3 W
```

Chaining (Bypass Network) Vector Execution and More RF Ports (cont.)

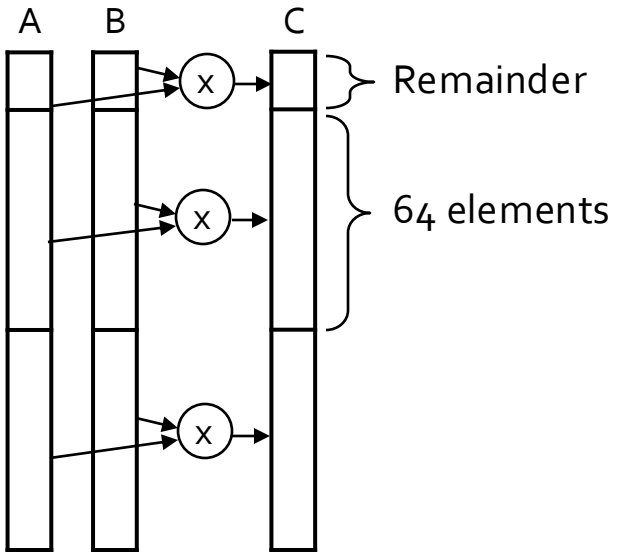
```
vmul    v3,v1,v2        F  D  D  R  Y0 Y1 Y2 Y3 W
                                   R  Y0 Y1 Y2 Y3 W
                                      R  Y0 Y1 Y2 Y3 W
                                         R  Y0 Y1 Y2 Y3 W
                                            R  Y0 Y1 Y2 Y3 W
                                               R  Y0 Y1 Y2 Y3 W
                                                  R  Y0 Y1 Y2 Y3 W
                                                     R  Y0 Y1 Y2 Y3 W
vst     v3, (x3)           F  F  D  D  D  D  R  S0 S1 W
                                               R  S0 S1 W
                                                  R  S0 S1 W
                                                     R  S0 S1 W
                                                        R  S0 S1 W
                                                           R  S0 S1 W
                                                              R  S0 S1 W
                                                                 R  S0 S1 W
```

Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit in registers, "**Stripmining**"

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




```
 andi    x1, xN, 63  # N mod 64
 vsetvl  x1          # Do remainder
loop:
  vld    v1, (xA)
  vld    v2, (xB)
  vmul   v3, v1, v2
  vst    v3, (xC)
  slli   x2, x1, 3   # Multiply by 8
  add    xA, xA, x2  # Bump pointer
  add    xB, xB, x2
  add    xC, xC, x2
  sub    xN, xN, x1  # Subtract elements
  li     x1, 64
  vsetvl x1           # Reset full length
  bgtz   xN, loop    # Any more to do?
```

Vector Stripmining

```
vl= 4
vld     v1, (xA)     F  D  R  L0 L1 W
                        R  L0 L1 W
                           R  L0 L1 W
                              R  L0 L1 W
vld     v1, (xB)              F  D  R  L0 L1 W
                                    R  L0 L1 W
                                       R  L0 L1 W
                                          R  L0 L1 W
vmul    v3,v1,v2                 F  D  D  R  Y0 Y1 Y2 Y3 W
                                          R  Y0 Y1 Y2 Y3 W
                                             R  Y0 Y1 Y2 Y3 W
                                                R  Y0 Y1 Y2 Y3 W
vst     v3, (x3)                    F  F  D  D  D  D  R  S0 S1 W
                                                      R  S0 S1 W
                                                         R  S0 S1 W
                                                            R  S0 S1 W
```

```
  andi   x1, xN, 63 # N mod 64
  vsetvl x1         # Do remainder
loop:
  vld    v1, (xA)
  vld    v2, (xB)
  vmul   v3, v1, v2
  vst    v3, (xC)
  slli   x2, x1, 3  # Multiply by 8
  add    xA, xA, x2 # Bump pointer
  add    xB, xB, x2
  add    xC, xC, x2
  sub    xN, xN, x1 # Subtract elements
  li     x1, 64
  vsetvl x1         # Reset full length
  bgtz   xN, loop   # Any more to do?
```

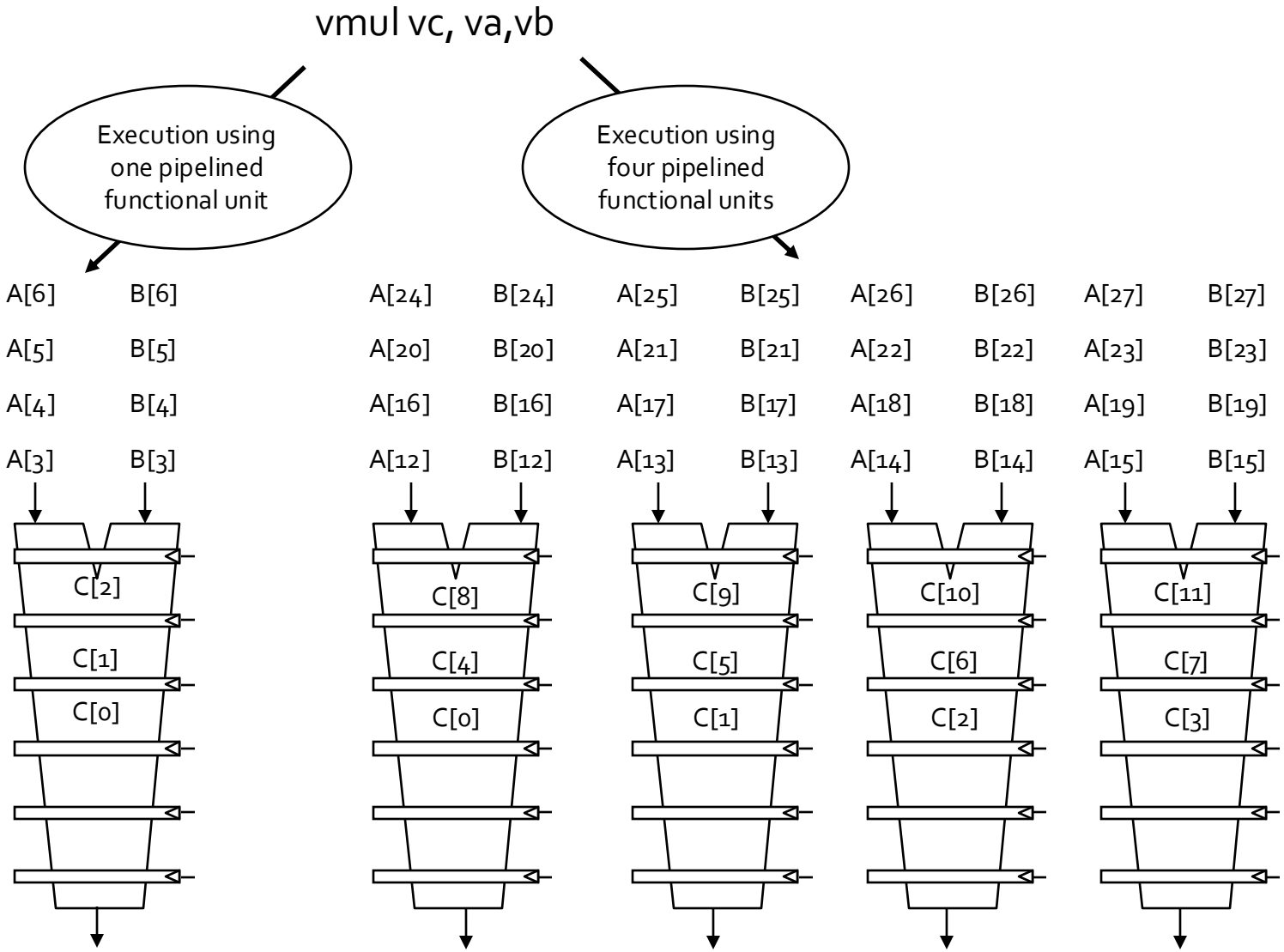
Vector Stripmining (cont.)

```
vst     v3, (x3)
slli    x2, x1, 3
add     xA, xA, x2
add     xB, xB, x2
add     xC, xC, x2
sub     xN, xN, x1
li      x1, 64
vsetvl  x1
bgtz    xN, loop
```

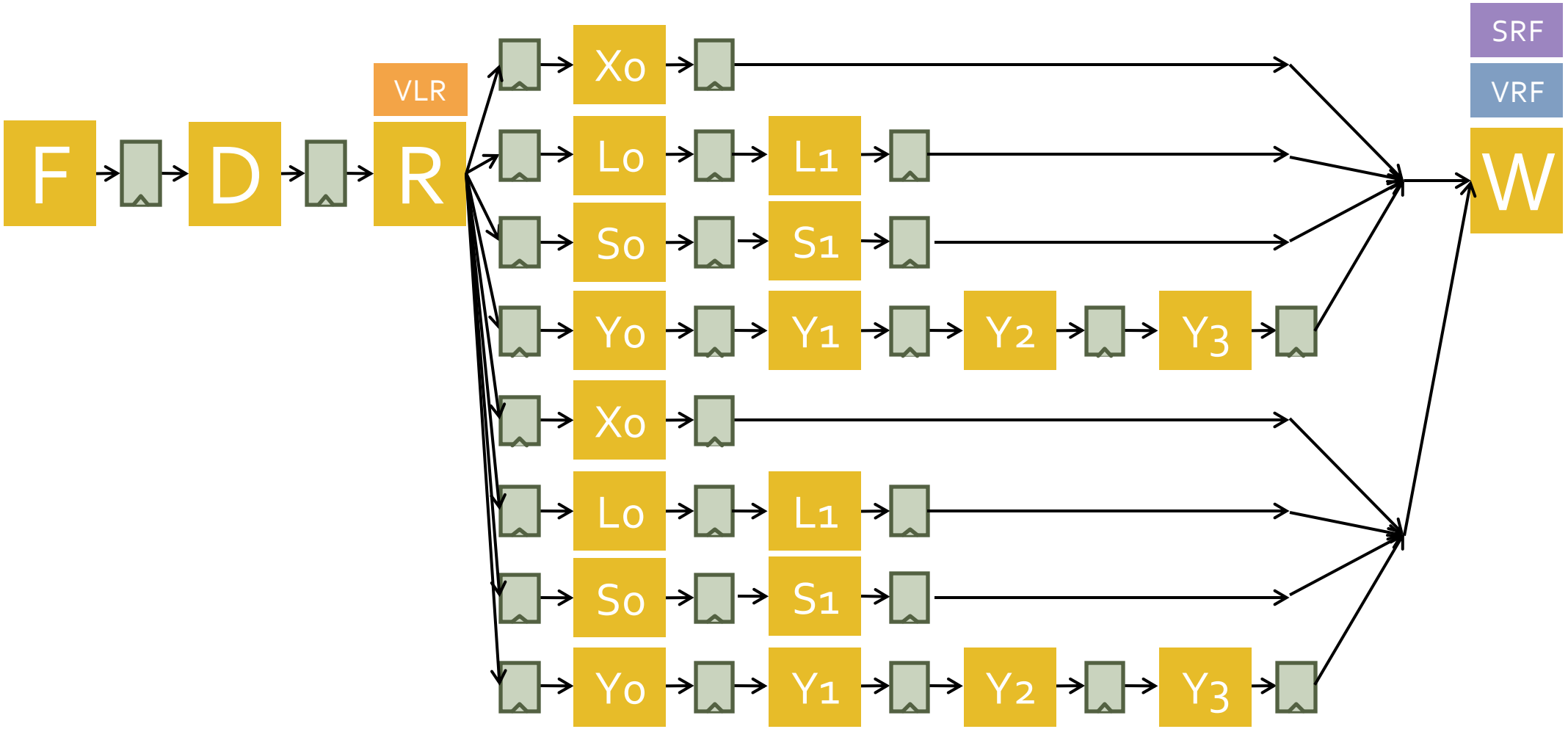
```
F  F  D  D  D  D  R  S0 S1 W
                  R  S0 S1 W
                     R  S0 S1 W
                        R  S0 S1 W
      F  F  F  F  D  R  X  W
                  F  D  R  X  W
                     F  D  R  X  W
                        F  D  R  X  W
                           F  D  R  X  W
                              F  D  R  X  W
                                 F  D  R  X  W
```

```
  andi   x1, xN, 63 # N mod 64
  vsetvl x1         # Do remainder
loop:
  vld    v1, (xA)
  vld    v2, (xB)
  vmul   v3, v1, v2
  vst    v3, (xC)
  slli   x2, x1, 3  # Multiply by 8
  add    xA, xA, x2 # Bump pointer
  add    xB, xB, x2
  add    xC, xC, x2
  sub    xN, xN, x1 # Subtract elements
  li     x1, 64
  vsetvl x1         # Reset full length
  bgtz   xN, loop   # Any more to do?
```

Vector Instruction Execution



Two Lane Vector Microarchitecture



Vector Stripmining 2-Lanes

```
vl= 4
vld     v1, (xA)     F  D  R  L0 L1 W
                        R  L0 L1 W
                           R  L0 L1 W
                           R  L0 L1 W
vld     v1, (xB)        F  D  D  R  L0 L1 W
                              R  L0 L1 W
                                 R  L0 L1 W
                                 R  L0 L1 W
vmul    v3,v1,v2           F  F  D  D  R  Y0 Y1 Y2 Y3 W
                                    R  Y0 Y1 Y2 Y3 W
                                       R  Y0 Y1 Y2 Y3 W
                                       R  Y0 Y1 Y2 Y3 W
vst     v3, (x3)                F  F  D  D  D  D  R  S0 S1 W
                                                  R  S0 S1 W
                                                     R  S0 S1 W
                                                     R  S0 S1 W
```

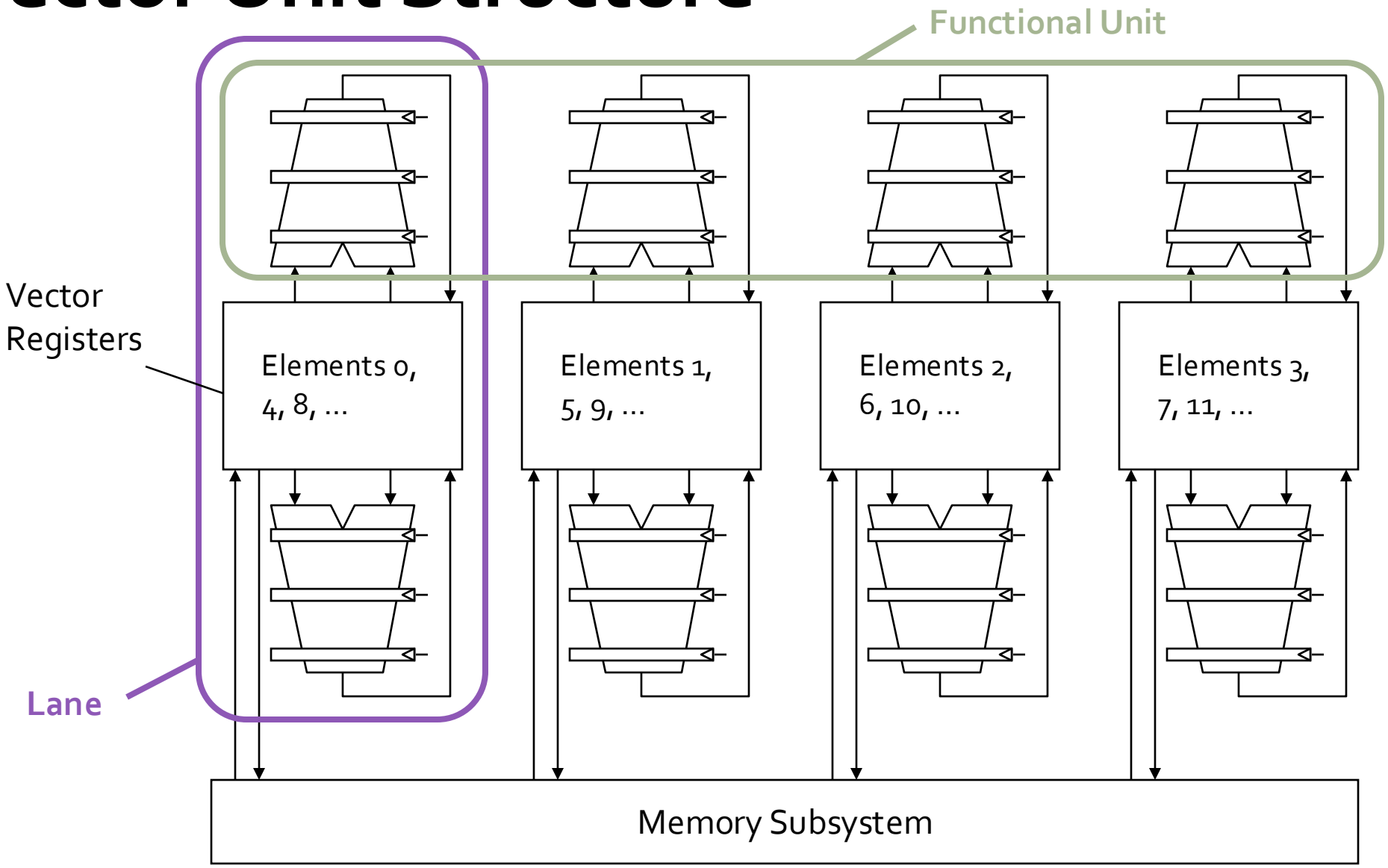
```
  andi   x1, xN, 63 # N mod 64
  vsetvl x1         # Do remainder
loop:
  vld    v1, (xA)
  vld    v2, (xB)
  vmul   v3, v1, v2
  vst    v3, (xC)
  slli   x2, x1, 3  # Multiply by 8
  add    xA, xA, x2 # Bump pointer
  add    xB, xB, x2
  add    xC, xC, x2
  sub    xN, xN, x1 # Subtract elements
  li     x1, 64
  vsetvl x1         # Reset full length
  bgtz   xN, loop   # Any more to do?
```

Vector Stripmining 2-Lanes (cont.)

```
vl = 4
vst     v3, (x3)                F  F  D  D  D  D  R  S0 S1 W
                                                  R  S0 S1 W
                                                     R  S0 S1 W
                                                     R  S0 S1 W
slli    x2, x1, 3                  F  F  F  F  D  R  X  W
add     xA, xA, x2                                F  D  R  X  W
add     xB, xB, x2                                   F  D  R  X  W
add     xC, xC, x2                                      F  D  R  X  W
sub     xN, xN, x1                                         F  D  R  X  W
li      x1, 64                                                F  D  R  X  W
vsetvl  x1                                                       F  D  R  X  W
bgtz    xN, loop                                                    F  D  R  X  W
```

```
  andi   x1, xN, 63 # N mod 64
  vsetvl x1         # Do remainder
loop:
  vld    v1, (xA)
  vld    v2, (xB)
  vmul   v3, v1, v2
  vst    v3, (xC)
  slli   x2, x1, 3  # Multiply by 8
  add    xA, xA, x2 # Bump pointer
  add    xB, xB, x2
  add    xC, xC, x2
  sub    xN, xN, x1 # Subtract elements
  li     x1, 64
  vsetvl x1         # Reset full length
  bgtz   xN, loop   # Any more to do?
```

Vector Unit Structure



To Vector Microprocessor (UCB/ICSI, 1995)

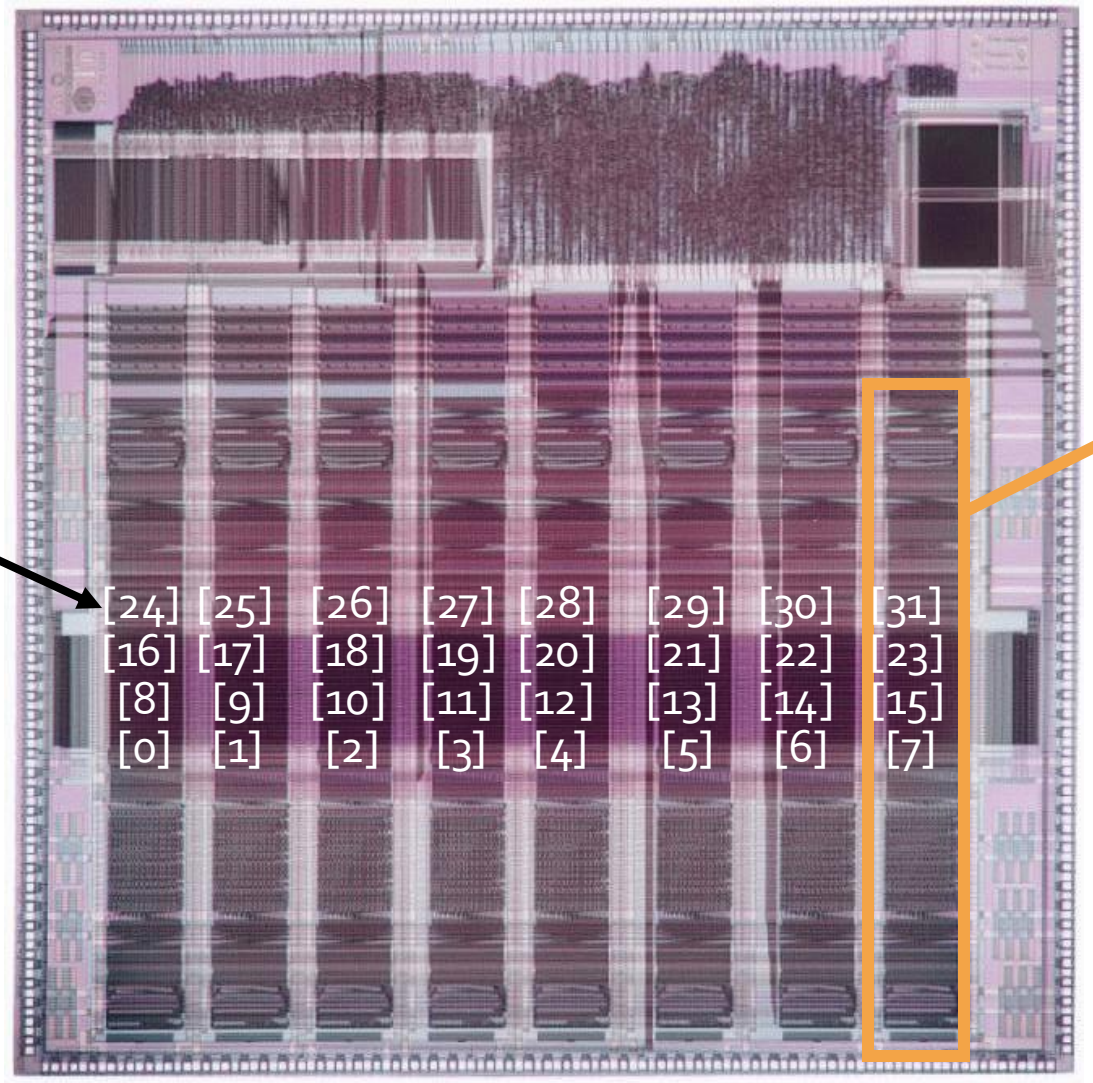
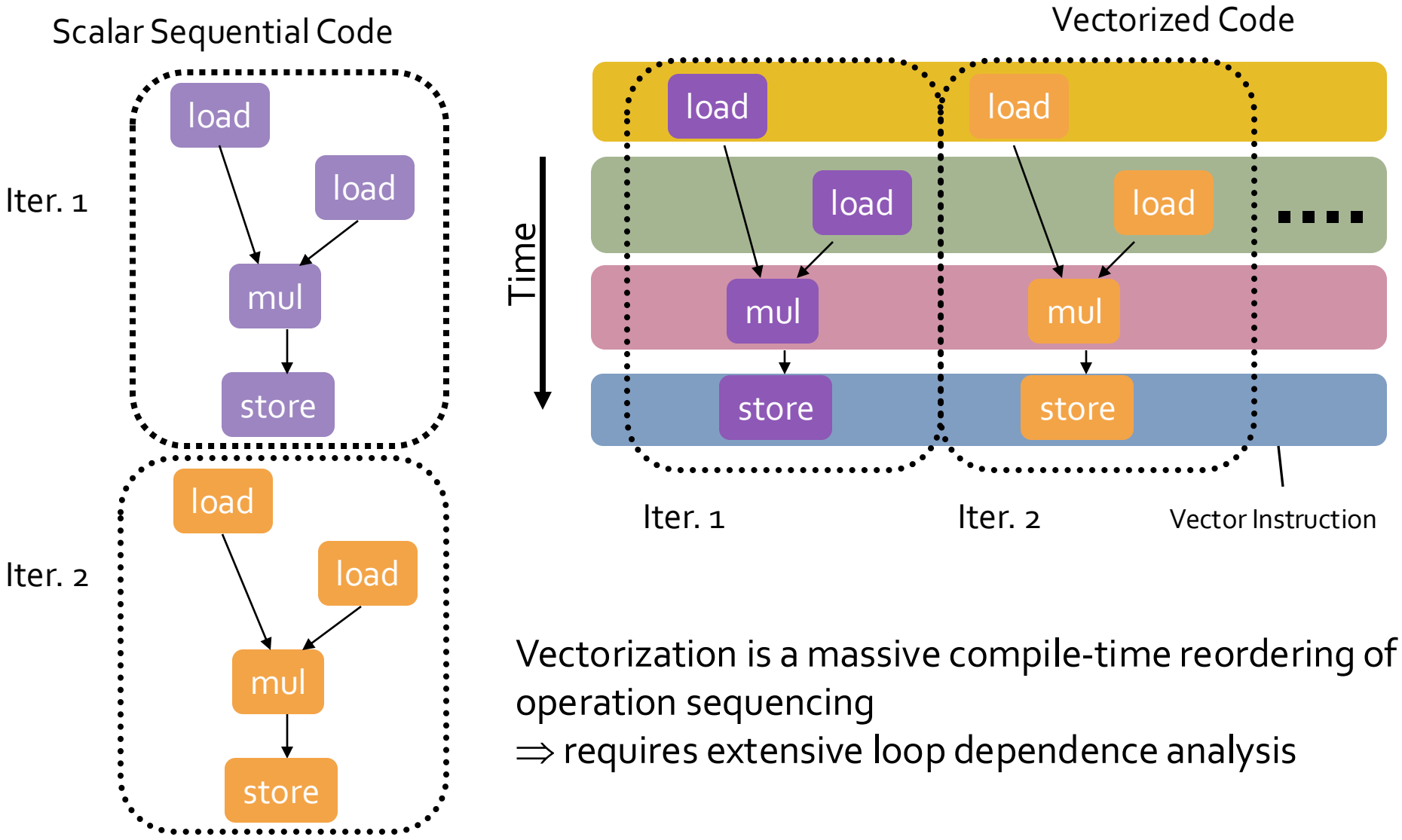


Photo of Berkeley To,
© University of California (Berkeley)
http://www1.icsi.berkeley.edu/Speech/spert/todie.jpg

32

Automatic Code Vectorization

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



Vectorization is a massive compile-time reordering of operation sequencing

⇒ requires extensive loop dependence analysis

Vector Conditional Execution

- Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

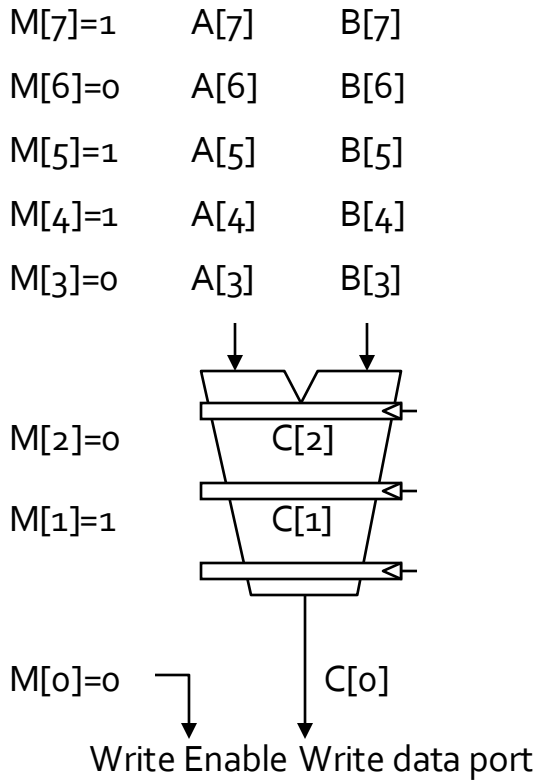
- Solution: Add vector mask (or flag) registers
  - vector version of predicate registers, 1 bit per element
- …and maskable vector instructions
  - vector operation becomes NOP at elements where mask bit is clear

- Code example:

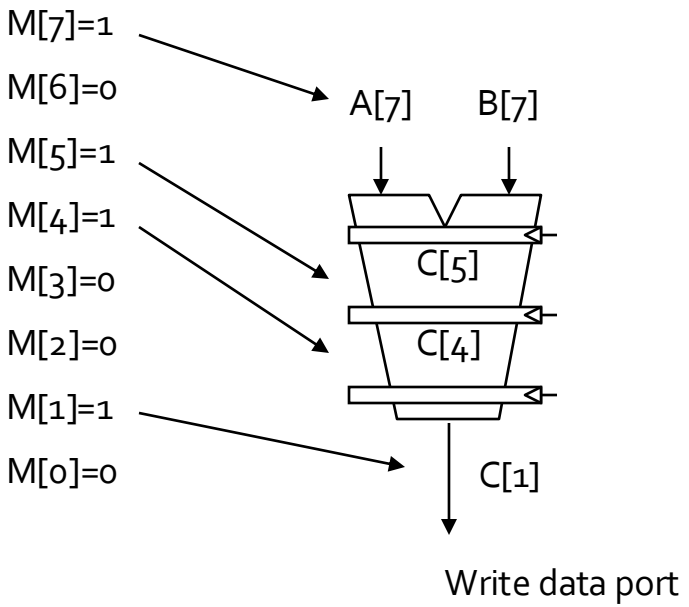
```
vld vA, (xA)       # Load entire A vector
vgt vA, f0         # Set bits in mask register where A>0
vld vA, (xB)       # Load B vector into A under mask
vst vA, (xA)       # Store A back to memory under mask
```

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



C[i]=A[i]+B[i]

Vector Reductions

- Problem: Loop-carried dependence on reduction variables

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  # Loop-carried dependence on sum
```

- Solution: Re-associate operations if possible, use binary tree to perform reduction

```
# Rearrange as:
sum[0:VL-1] = 0                  # Vector of VL partial sums
for(i=0; i<N; i+=VL)             # Stripmine VL-sized chunks
    sum[0:VL-1] += A[i:i+VL-1]; # Vector sum
# Now have VL partial sums in one vector register
do {
    VL = VL/2;                     # Halve vector length
    sum[0:VL-1] += sum[VL:2*VL-1] # Halve no. of partials
} while (VL>1)
```

Vector Scatter/Gather

- Want to vectorize loops with indirect accesses:

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

- Indexed load instruction (**Gather**)

```
vld  vD, (xD)     # Load indices in D vector
vlx  vC, (xC), vD # Load indexed from xC base
vld  vB, (xB)     # Load B vector
vadd vA,vB,vC     # Do add
vst  vA, (xA)     # Store result
```

Is this a correct translation? Yes if A doesn't overlap with the other vectors

Vector Scatter/Gather

- Scatter example:

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

- Is following a correct translation?

```
vld vB, (xB)     # Load indices in B vector
vlx vA, (xA), vB # Gather initial A values
vadd vA, vA, 1   # Increment
vsx vA, (xA), vB # Scatter incremented values
```

Incorrect if B may contain repeated values

Vector Supercomputers

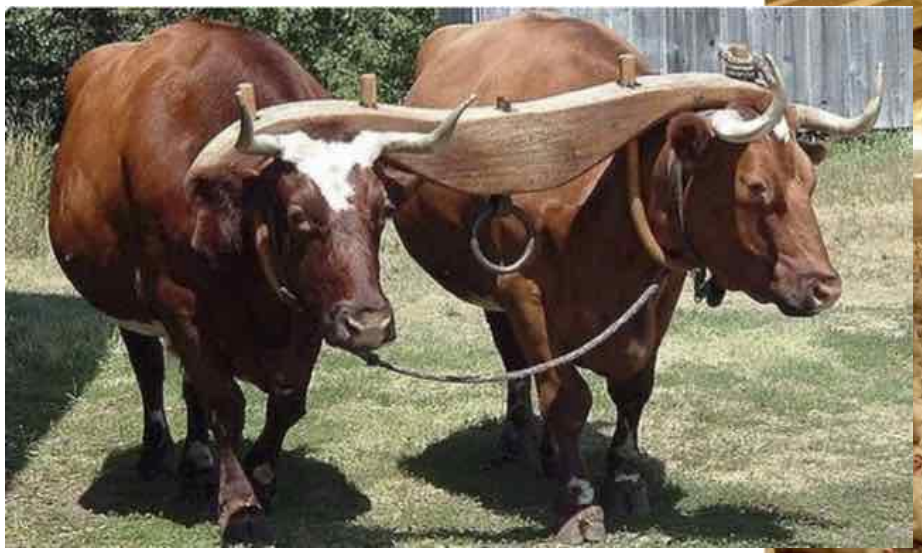
Epitomized by Cray-1, 1976:

- Scalar Unit
  - Load/Store Architecture
- Vector Extension
  - Vector Registers
  - Vector Instructions
- Implementation
  - Hardwired Control
  - Highly Pipelined Functional Units
  - Interleaved Memory System
  - No Data Caches
  - No Virtual Memory

Cray 1 at The Deutsches Museum
Image Credit: Clemens Pfeiffer
http://en.wikipedia.org/wiki/File:Cray-1-deutsches-museum.jpg

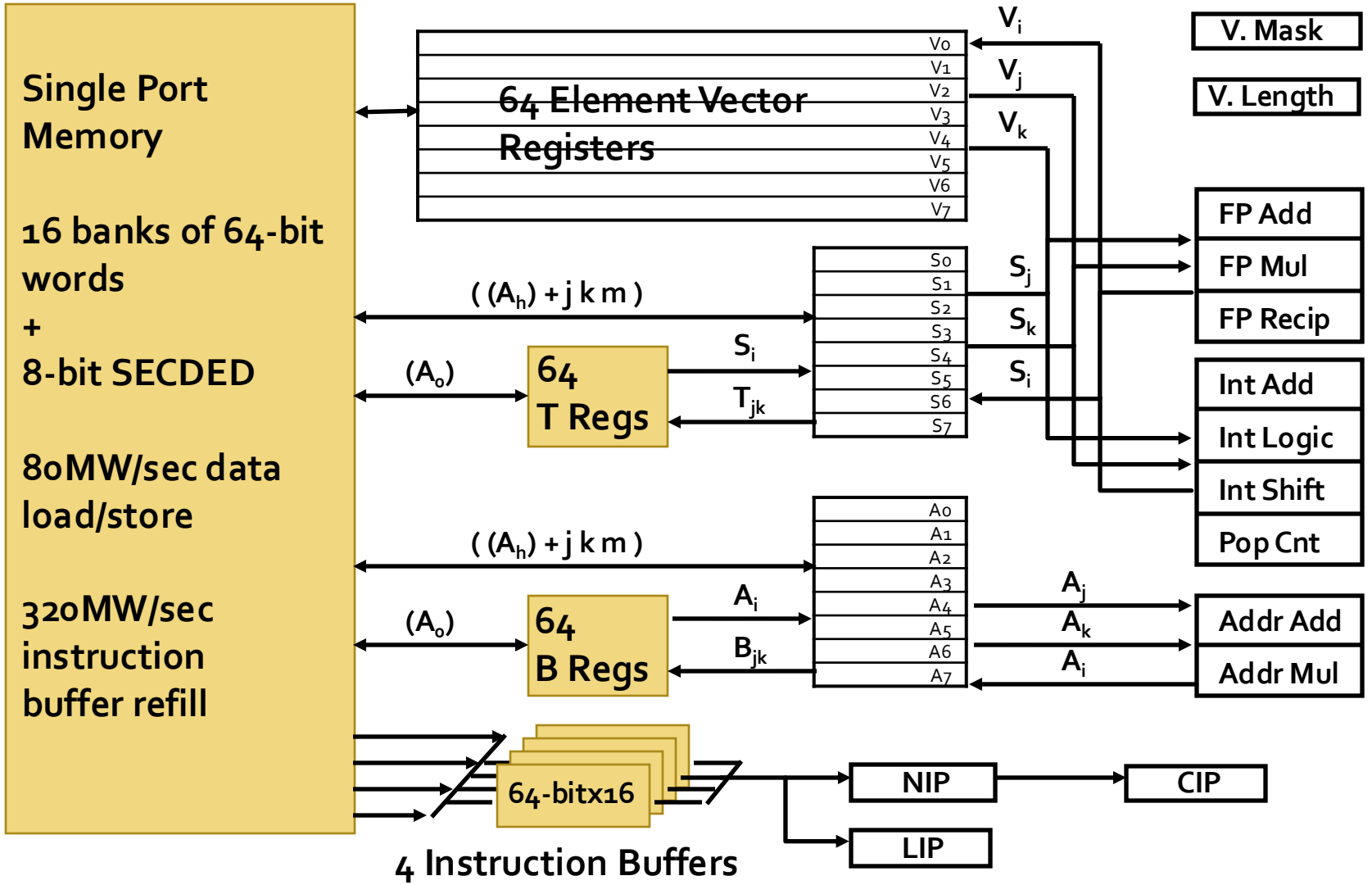
Seymour Cray, Leader in Supercomputer Design

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



Cray-1 (1976)

memory bank cycle 50 ns processor cycle 12.5 ns (80MHz)

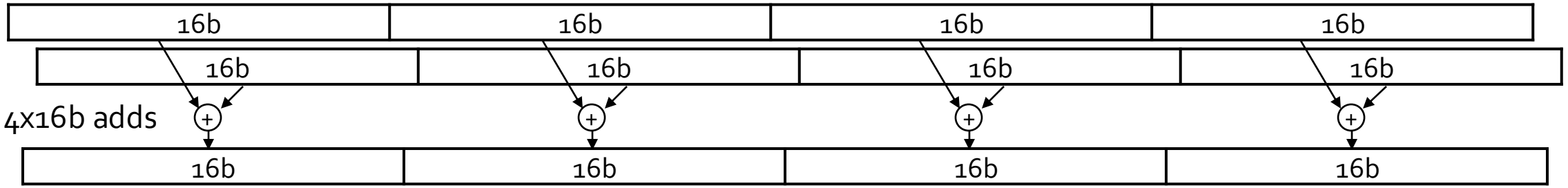


Agenda

- Vector Processors
- Single Instruction Multiple Data (SIMD) Instruction Set Extensions
- Graphics Processing Units (GPU)

Packed SIMD Extensions

- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - This concept first used on Lincoln Labs TX-2 computer in 1957, with 36b datapath split into 2x18b or 4x9b
  - Newer designs have wider registers
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b/512b for Intel AVX, AVX-512
- Single instruction operates on all elements within register



MMX Example:
Image Overlaying (I)

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

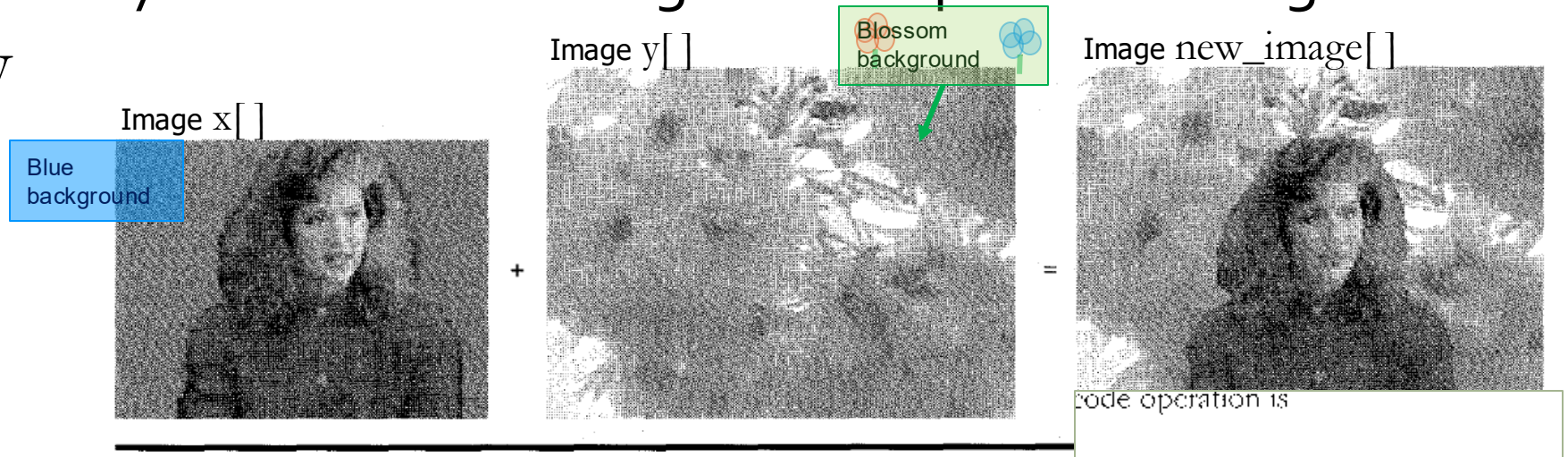


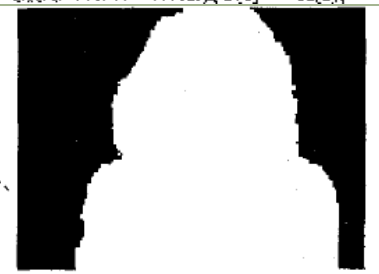
Figure 8. Chroma keying: image overlay using a background color.

code operation is

```
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

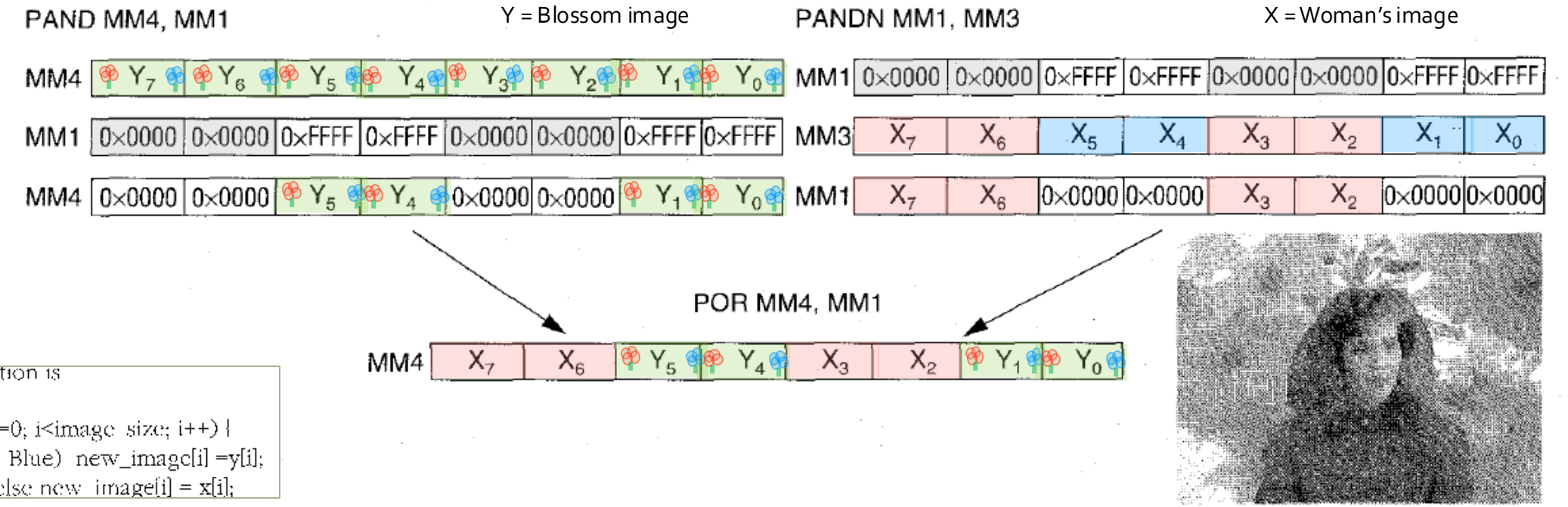
	MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
Image x[ ]	MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
Bit mask	MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF



Bitmask

MMX Example:
Image Overlaying (II)

```
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
```



code operation is

```
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.

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

Intel SIMD Evolution

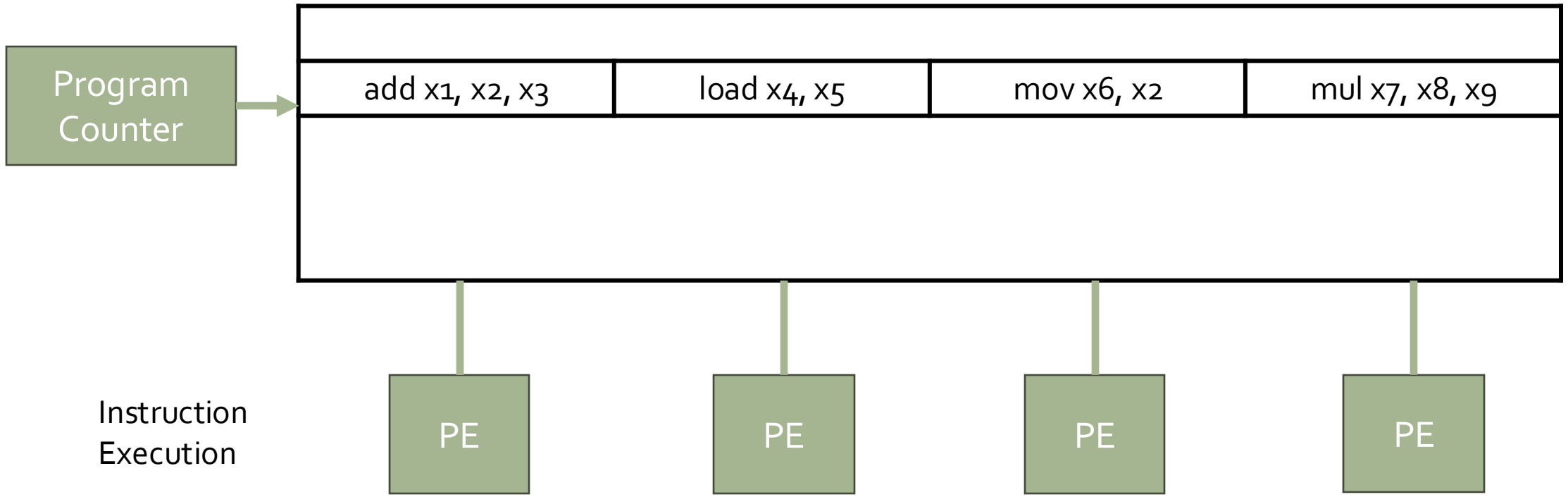
- Implementations:
- Intel MMX (1996) – 64bits
  - Eight 8-bit integer ops, or
  - Four 16-bit integer ops
  - Two 32-bit integer ops
- Streaming SIMD Extensions (SSE) (1999) – 128bits
  - Four 32-bit integer ops (and smaller integer types)
  - Four 32-bit integer/fp ops, or
  - Two 64-bit integer/fp ops
- Advanced Vector Extensions (2010) – 256bits
  - Four 64-bit integer/fp ops (and smaller fp types)
- AVX-512 (2017) – 512bits
  - New instructions: scatter/gather, mask register

Packed SIMD vs Vectors

- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support in microprocessors
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - Support for masked operations

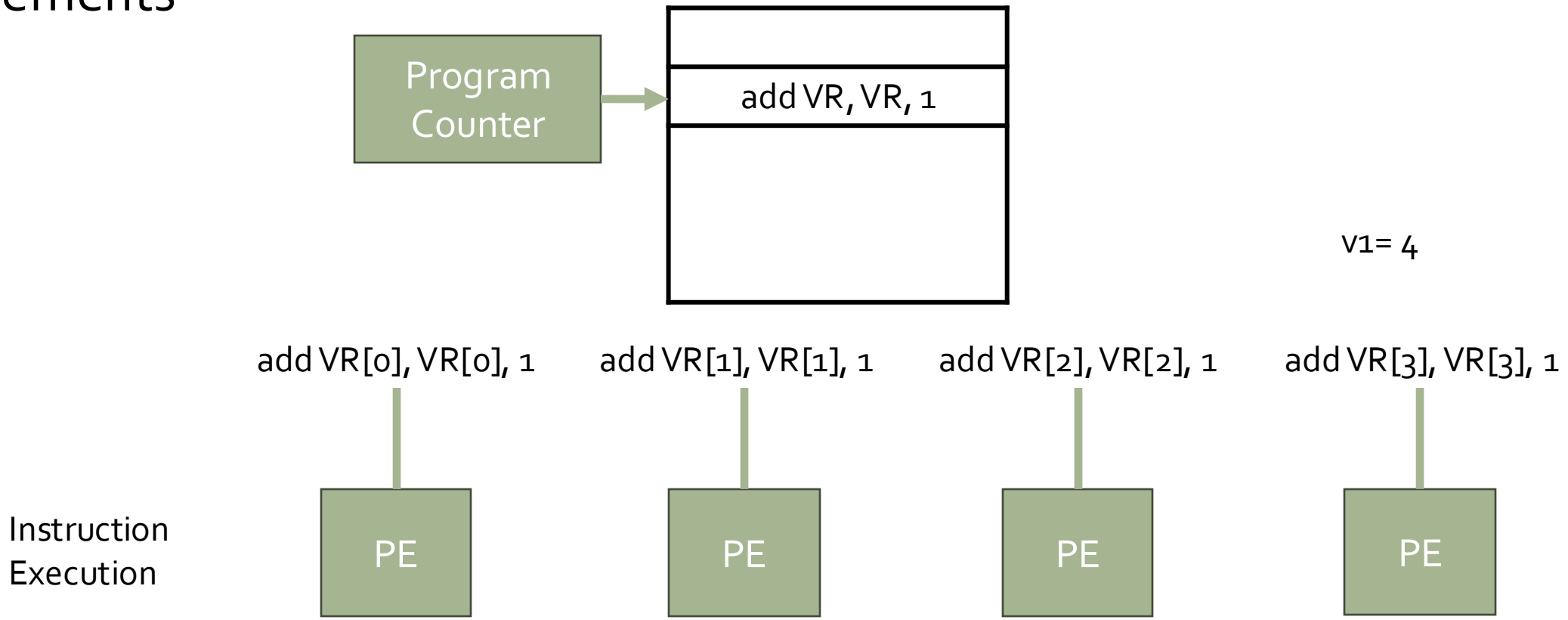
SIMD Array Processing vs. VLIW

- VLIW: **Multiple** independent **operations** packed together into a "long inst."



SIMD Array Processing vs. VLIW

- Array processor: **Single operation** on multiple (different) data elements



Vector/SIMD Processing Summary

- Vector/SIMD machines 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
  - Amdahl's Law
  - CRAY-1 was the fastest SCALAR machine at its time!
- Many existing ISAs include (vector-like) SIMD operations

Agenda

- Vector Processors
- Single Instruction Multiple Data (SIMD) Instruction Set Extensions
- Graphics Processing Units (GPU)

Translation between GPU terms in book and official NVIDIA and OpenCL terms.

Type	More Descriptive Name used in this Book	Official CUDA/ NVIDIA Term	Book Definition and OpenCL Terms	Official CUDA/NVIDIA Definition
Program Abstractions	**Vectorizable Loop**	**Grid**	A vectorizable loop, executed on the GPU, made up of 1 or more "Thread Blocks" (or bodies of vectorized loop) that can execute in parallel. OpenCL name is "index range."	A Grid is an array of Thread Blocks that can execute concurrently, sequentially, or a mixture.
	Body of Vectorized Loop	**Thread Block**	A vectorized loop executed on a "Streaming Multiprocessor" (multithreaded SIMD processor), made up of 1 or more "Warps" (or threads of SIMD instructions). These "Warps" (SIMD Threads) can communicate via "Shared Memory" (Local Memory). OpenCL calls a thread block a "work group."	A Thread Block is an array of CUDA threads that execute concurrently together and can cooperate and communicate via Shared Memory and barrier synchronization. A Thread Block has a Thread Block ID within its Grid.
	Sequence of SIMD Lane Operations	**CUDA Thread**	A vertical cut of a "Warp" (or thread of SIMD instructions) corresponding to one element executed by one "Thread Processor" (or SIMD lane). Result is stored depending on mask. OpenCL calls a CUDA thread a "work item."	A CUDA Thread is a lightweight thread that executes a sequential program and can cooperate with other CUDA threads executing in the same Thread Block. A CUDA thread has a thread ID within its Thread Block.

Resurgence of DLP

- Convergence of application demands and technology constraints drives architecture choice

- New applications, such as graphics, machine vision, speech recognition, machine learning, etc. all require large numerical computations that are often trivially data parallel

- SIMD-based architectures (vector-SIMD, subword-SIMD, SIMT/GPUs) are most efficient way to execute these algorithms

Graphics Processing Units (GPUs)

- Original GPUs were dedicated fixed-function devices for generating 3D graphics (mid-late 1990s) including high-performance floating-point units
  - Provide workstation-like graphics for PCs
  - User could configure graphics pipeline, but not really program it
- Over time, more programmability added (2001-2005)
  - E.g., New language Cg for writing small programs run on each vertex or each pixel, also Windows DirectX variants
  - Massively parallel (millions of vertices or pixels per frame) but very constrained programming model
- Some users noticed they could do general-purpose computation by mapping input and output data to images, and computation to vertex and pixel shading computations
  - Incredibly difficult programming model as had to use graphics pipeline model for general computation

GPUs were originally designed for 3D rendering

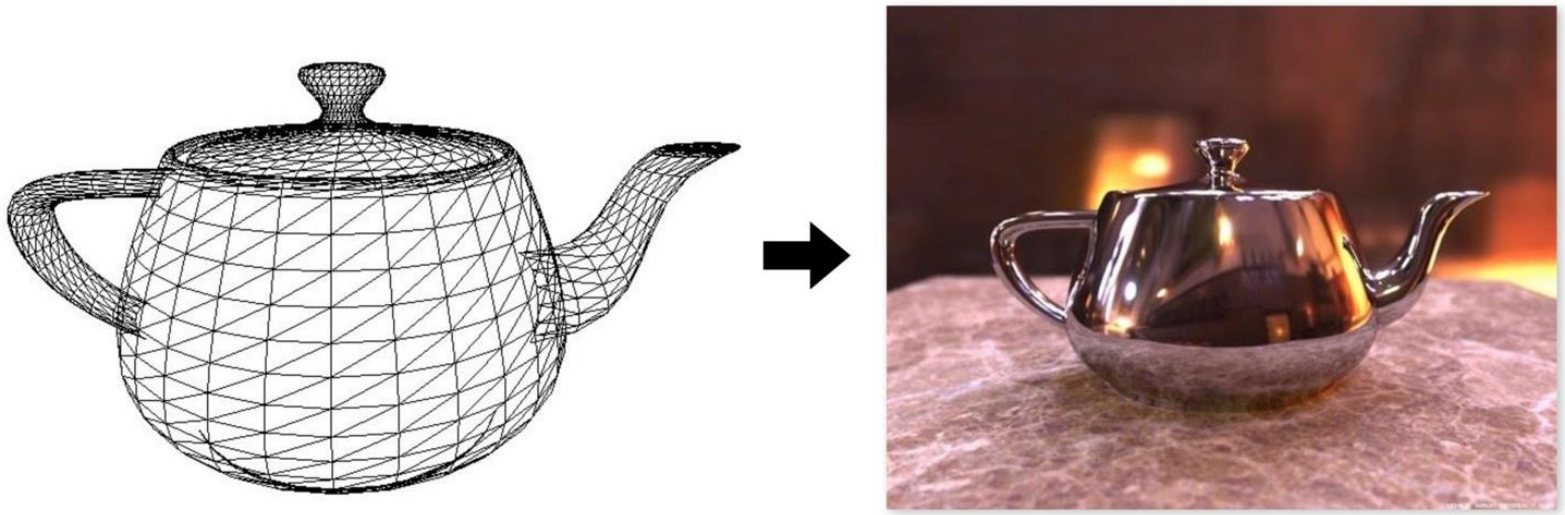


Image credit: Henrik Wann Jensen

Input: description of a scene:

3D surface geometry (e.g., triangle mesh)
surface materials, lights, camera, etc.

Output: image of the scene

[Image Credit: Kayvon Fatahalian]

Simple definition of rendering task: computing how each triangle in 3D mesh contributes to appearance of each pixel in the image?

What GPUs were originally designed to do

- 100's of thousands to millions of triangles in a scene
- Complex material, lighting, and animation computations
- High-resolution screen outputs (2-4 Mpixel + supersampling)
- 30-60 fps

[Unreal Engine Kite Demo (Epic Games 2015)]

General Purpose GPUs (GPGPUs)

- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
  - "Compute Unified Device Architecture"
  - Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution
- This lecture has a simplified version of Nvidia CUDA-style model and only considers GPU execution for computational kernels, not graphics

Simplified CUDA Programming Model

- Computation performed by a very large number of independent small scalar threads (**CUDA threads or microthreads**) grouped into **thread blocks**.

```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{ for (int i=0; i<n; i++)
      y[i] = a*x[i] + y[i]; }

// CUDA version.
__host__  // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__  // Piece run on GPGPU.
void daxpy(int n, double a, double*x, double*y)
{ int i = blockIdx.x*blockDim.x + threadId.x;
  if (i<n) y[i]=a*x[i]+y[i]; }
```

Programming Model vs. Hardware Execution Model

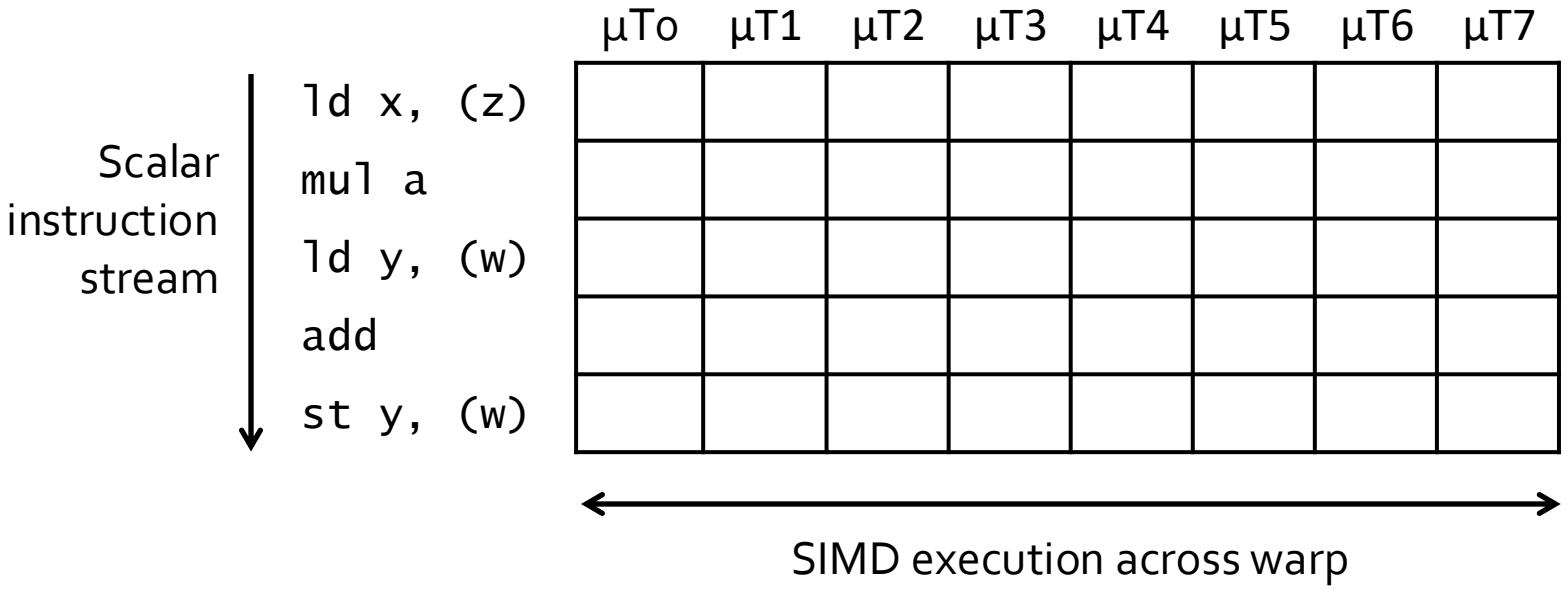
- Programming Model refers to how the programmer expresses the code
  - E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), …
- Execution Model refers to how the hardware executes the code underneath
  - E.g., Out-of-order execution, Vector processor, Array processor,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

A GPU is a SIMD (SIMT) Machine

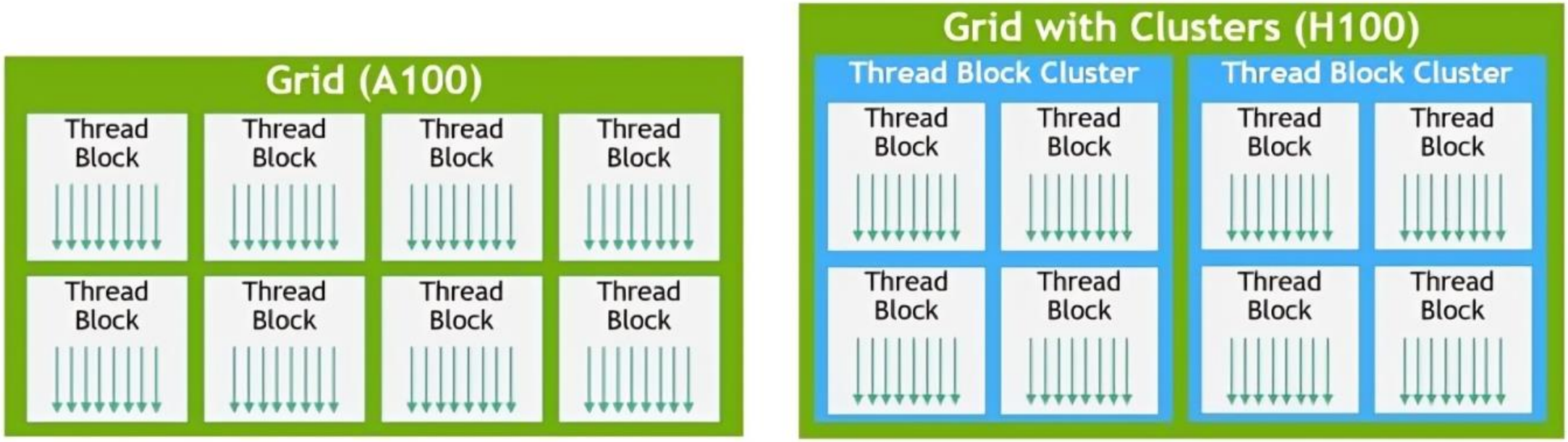
- Except it is **not** programmed using SIMD instructions
- It is programmed using threads (SPMD programming model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)
- A set of threads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

Single Instruction, Multiple Thread (SIMT)

- GPUs use a SIMT model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (Nvidia groups 32 CUDA threads into a **warp**)

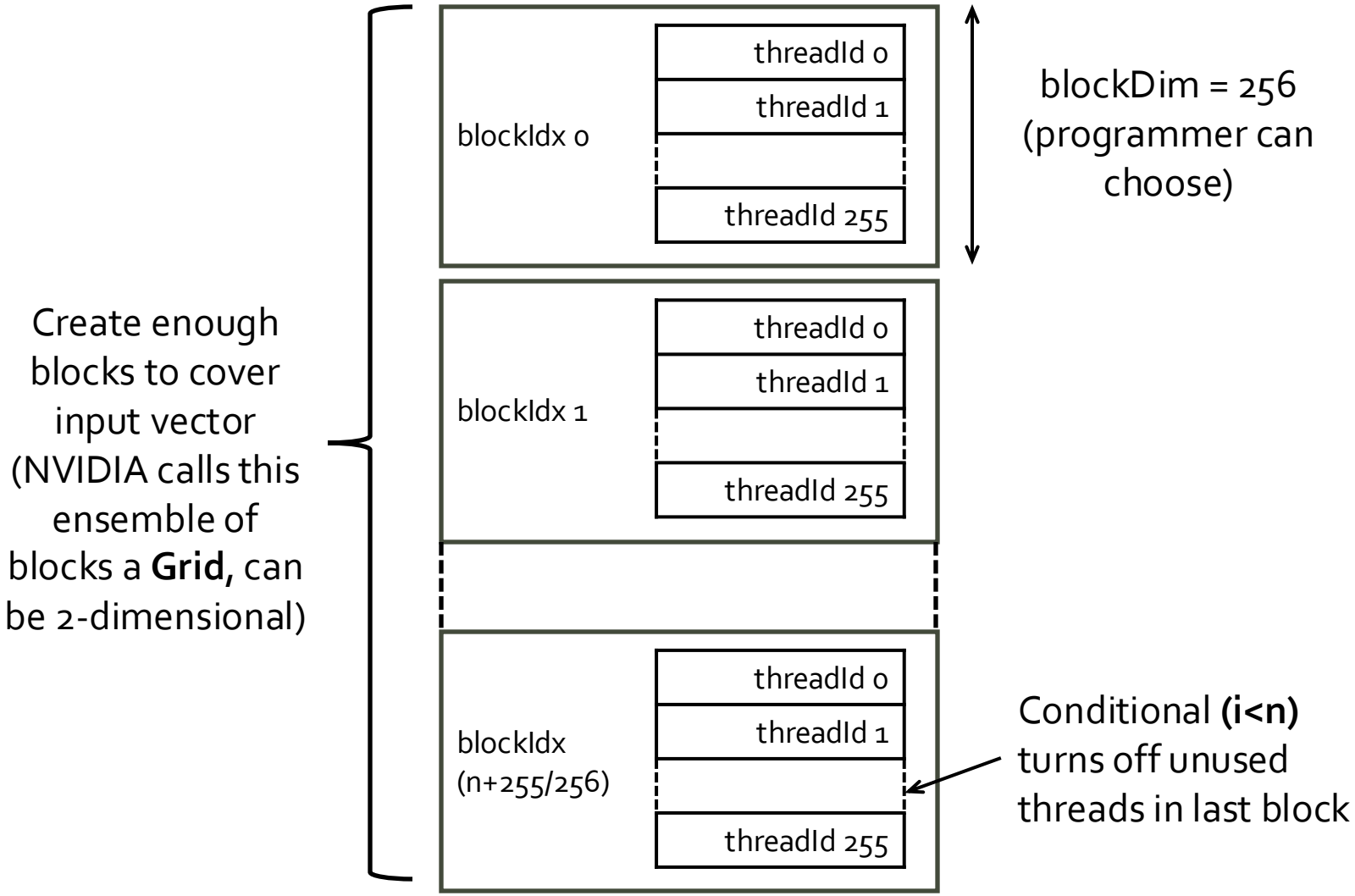


Programmer's View of Execution



https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

Programmer's View of Execution

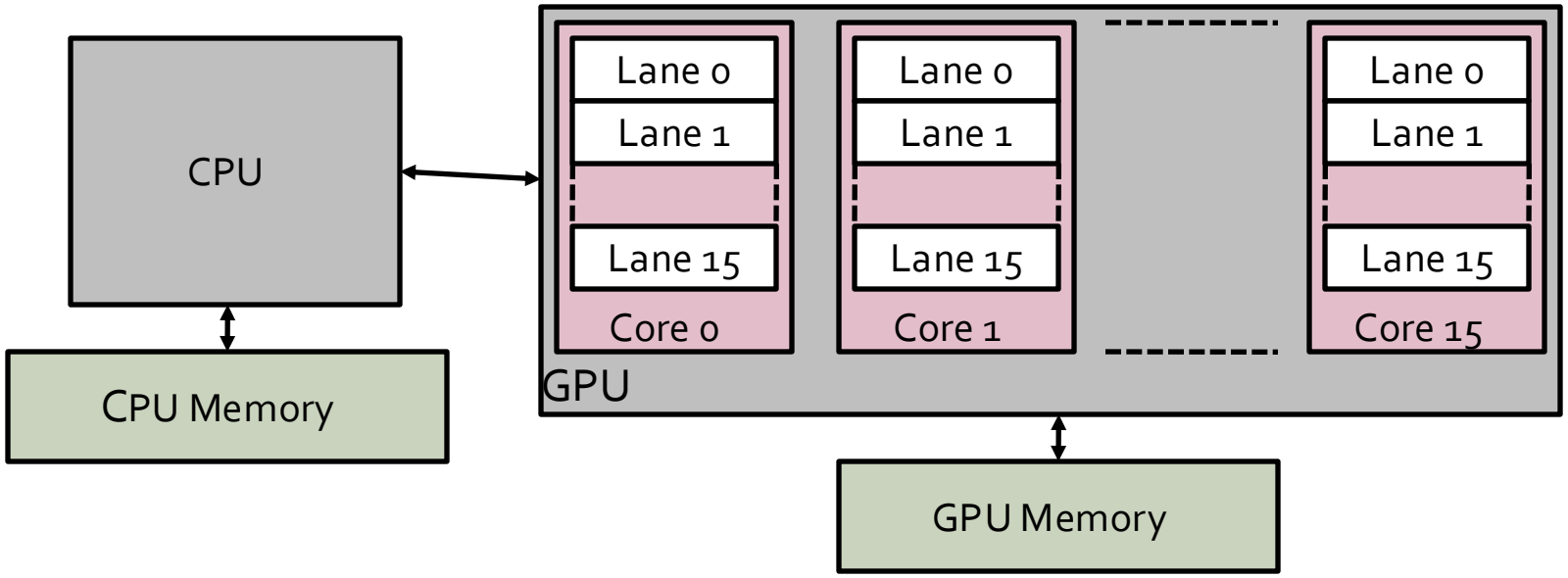


- Single-program multiple data (SPMD) model
- Each context is a thread
  - Threads have registers
  - Threads have local memory
- Parallel threads packed in blocks
  - Blocks have shared memory
  - Threads synchronize with barrier
  - Blocks run to completion (or abort)
- Grids include independent blocks
  - May execute concurrently
  - Share global memory, but
  - Have limited inter-block synchronization

Implications of SIMT Model

- All "vector" loads and stores are scatter-gather, as individual μthreads perform scalar loads and stores
  - GPU adds hardware to dynamically coalesce individual μthread loads and stores to mimic vector loads and stores
- Every μthread has to perform stripmining calculations redundantly ("am I active?") as there is no scalar processor equivalent
- If divergent control flow, need predicates

Hardware Execution Model



- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
  - Some adding "scalar coprocessors" now
- CPU sends whole "grid" over to GPU, which distributes thread blocks among cores (each thread block executes on one core)
  - Programmer unaware of number of cores

Multithreaded SIMD Processor

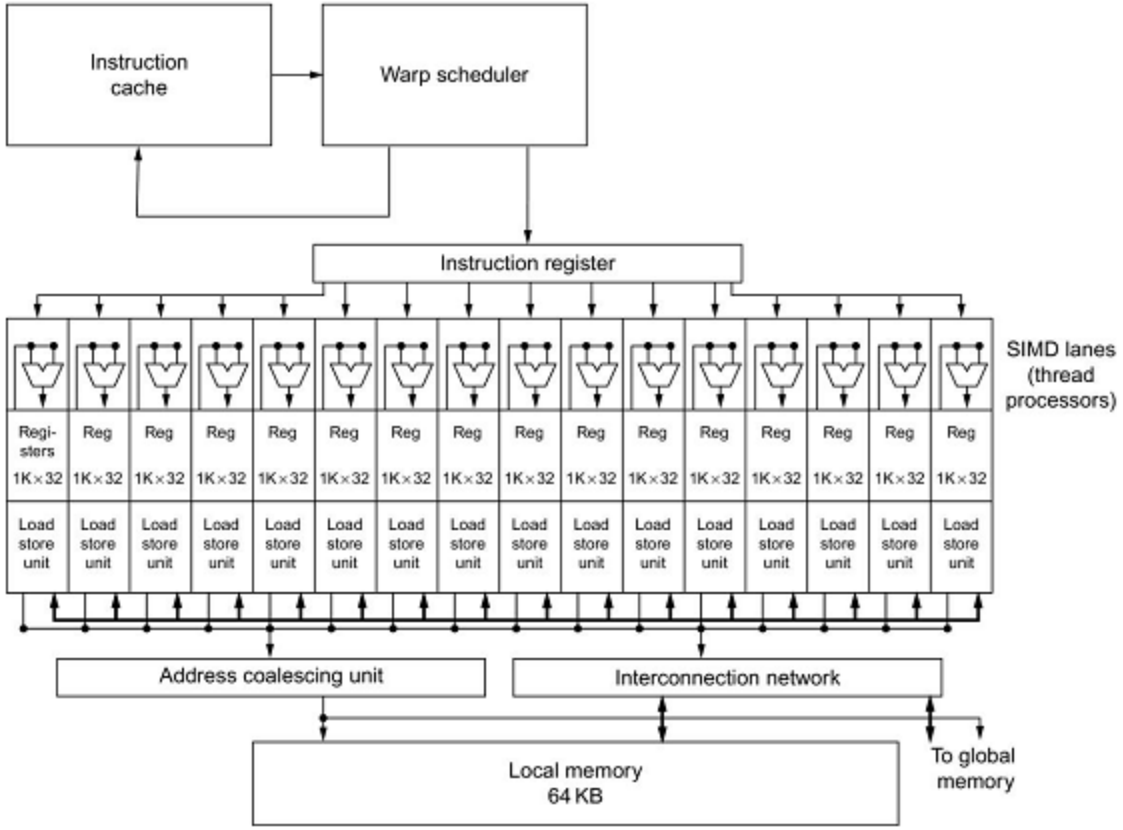
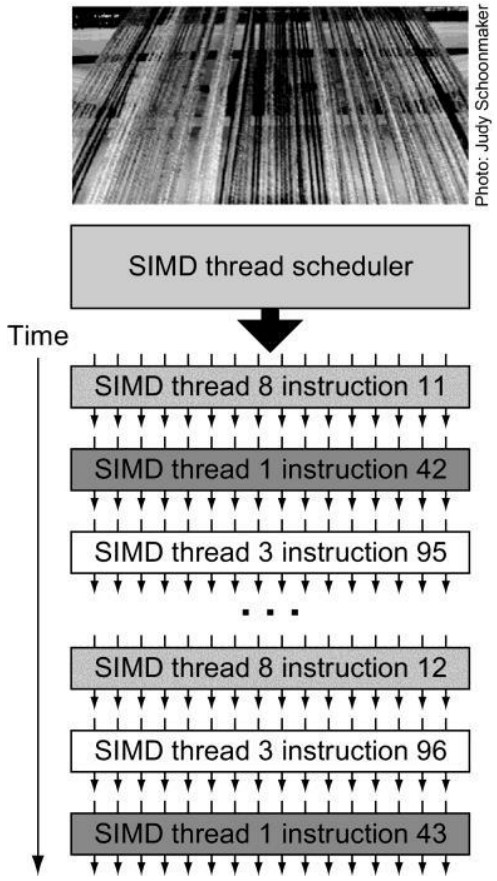


Figure 4.14 Simplified block diagram of a multithreaded SIMD Processor. It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

Warps are Multithreaded on Core



- Warp == SIMD Thread, i.e., a thread of SIMD instructions
- One warp of 32 µthreads is a single thread in the hardware
- Multiple warp threads are interleaved in execution on a single core to hide latencies (memory and functional unit)
- A single thread block can contain multiple warps (up to 512 µT max in CUDA), all mapped to single core
- Can have multiple blocks executing on one core

[Nvidia, 2010]

Vectors vs Multithreaded SIMD

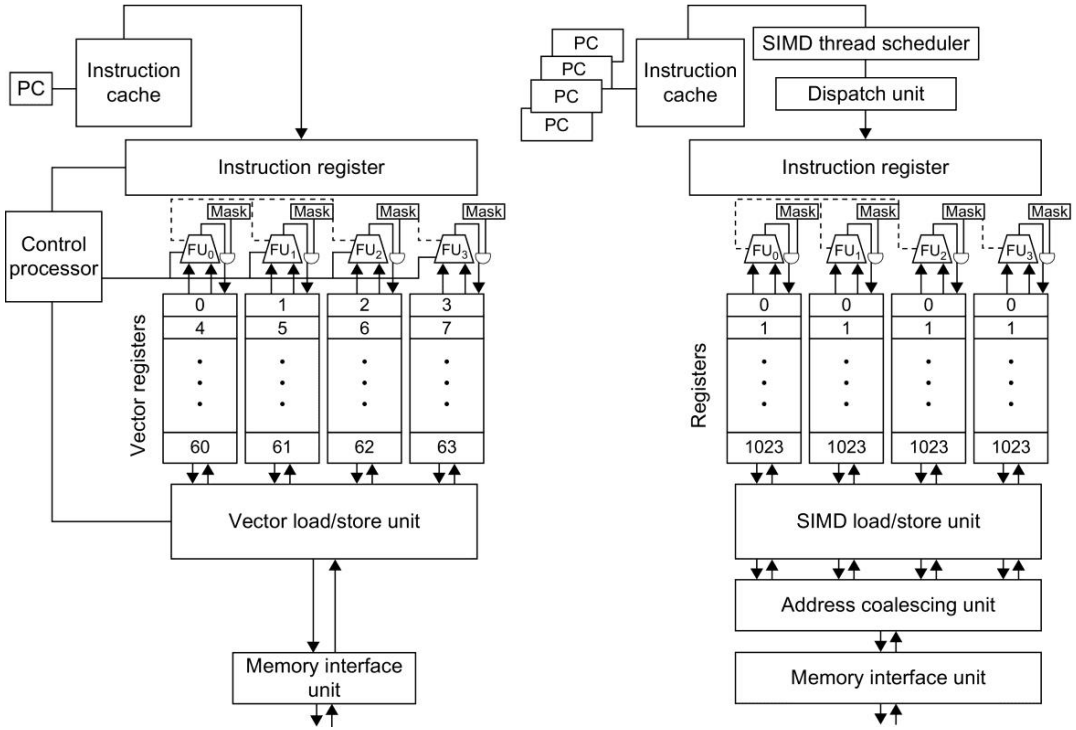
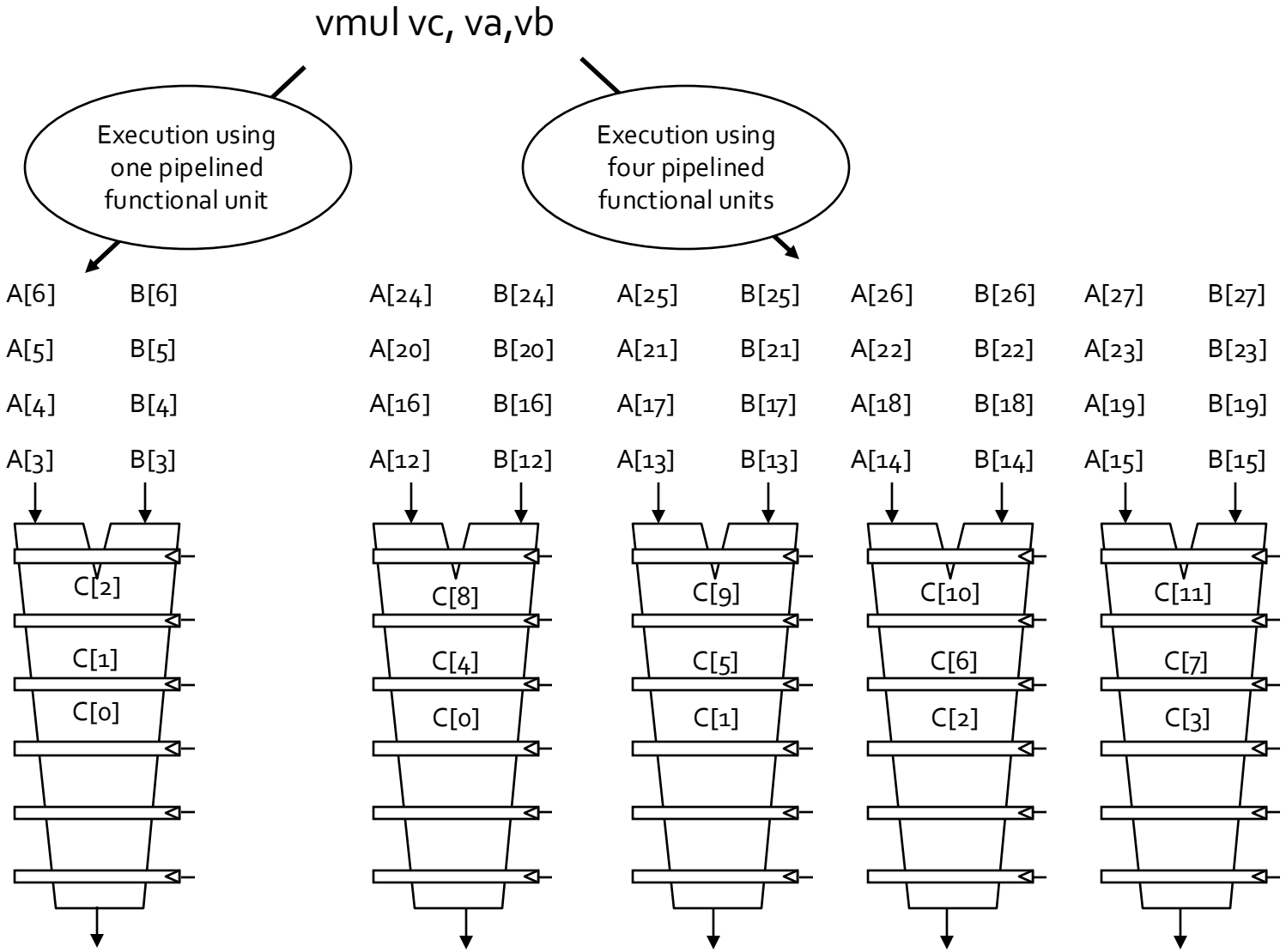
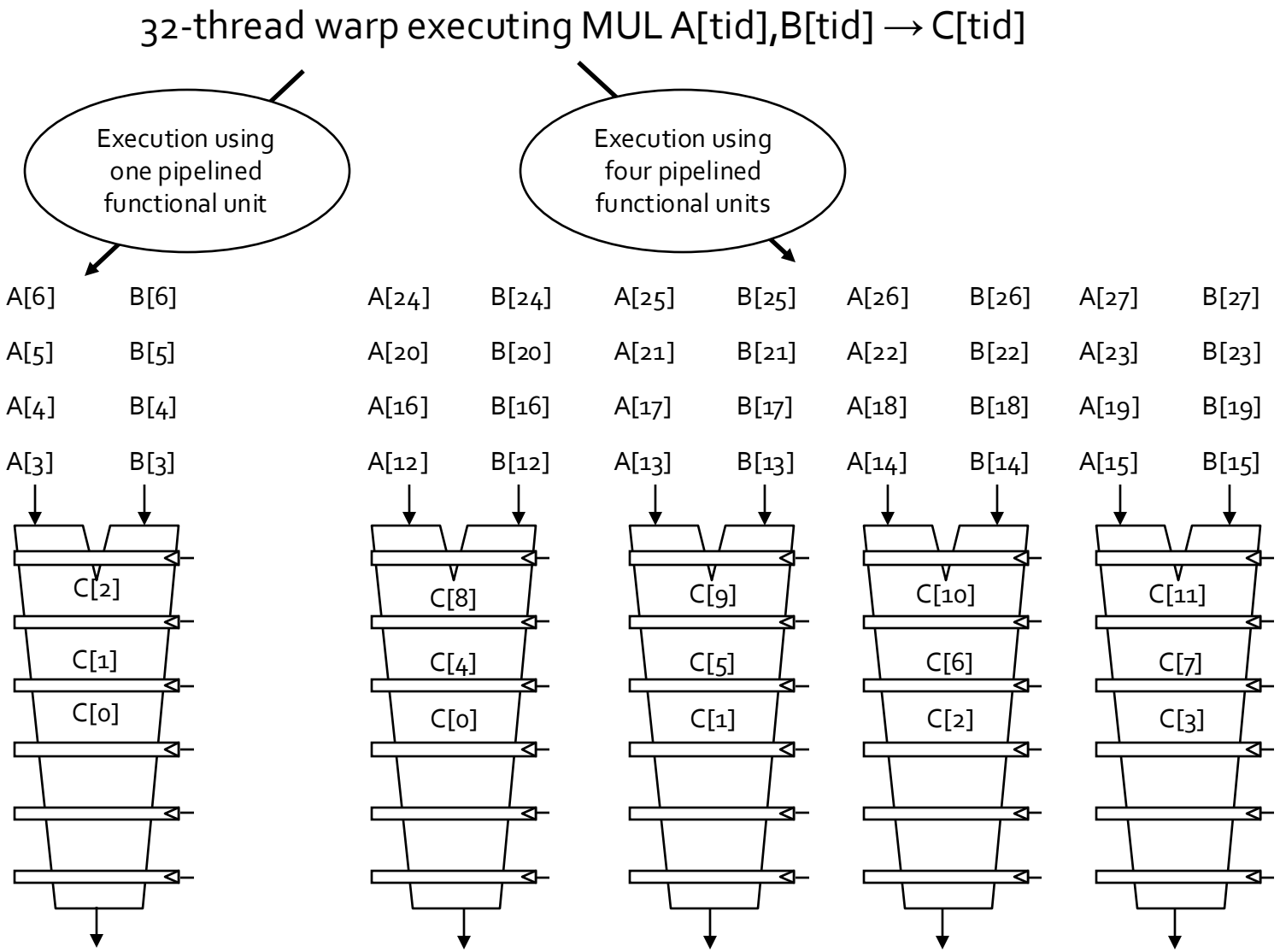


Figure 4.22 A vector processor with four lanes on the left and a multithreaded SIMD Processor of a GPU with four SIMD Lanes on the right. (GPUs typically have 16 or 32 SIMD Lanes.) The Control Processor supplies scalar operands for scalar-vector operations, increments addressing for unit and nonunit stride accesses to memory, and performs other accounting-type operations. Peak memory performance occurs only in a GPU when the Address Coalescing Unit can discover localized addressing. Similarly, peak computational performance occurs when all internal mask bits are set identically. Note that the SIMD Processor has one PC per SIMD Thread to help with multithreading.

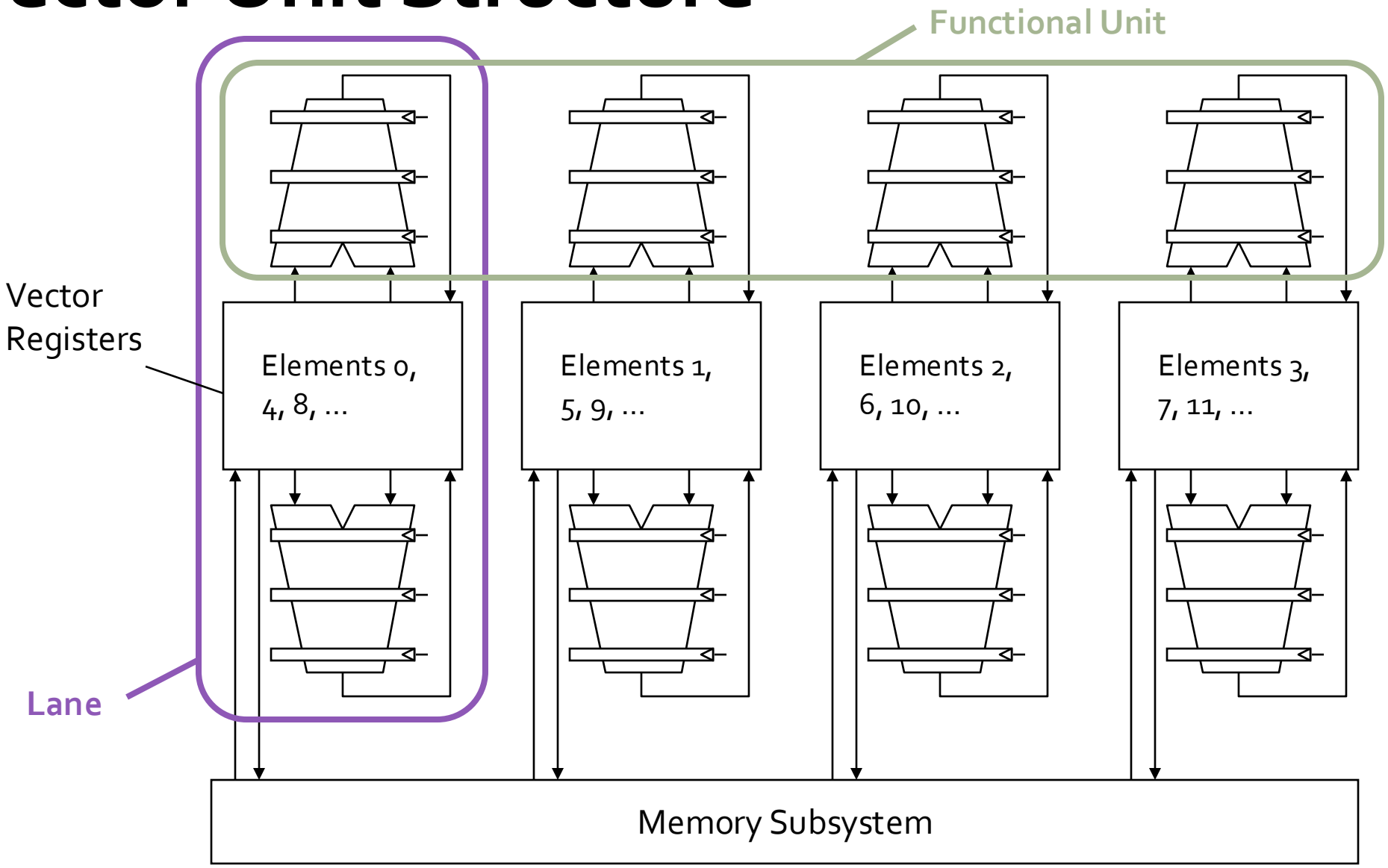
Vector Instruction Execution



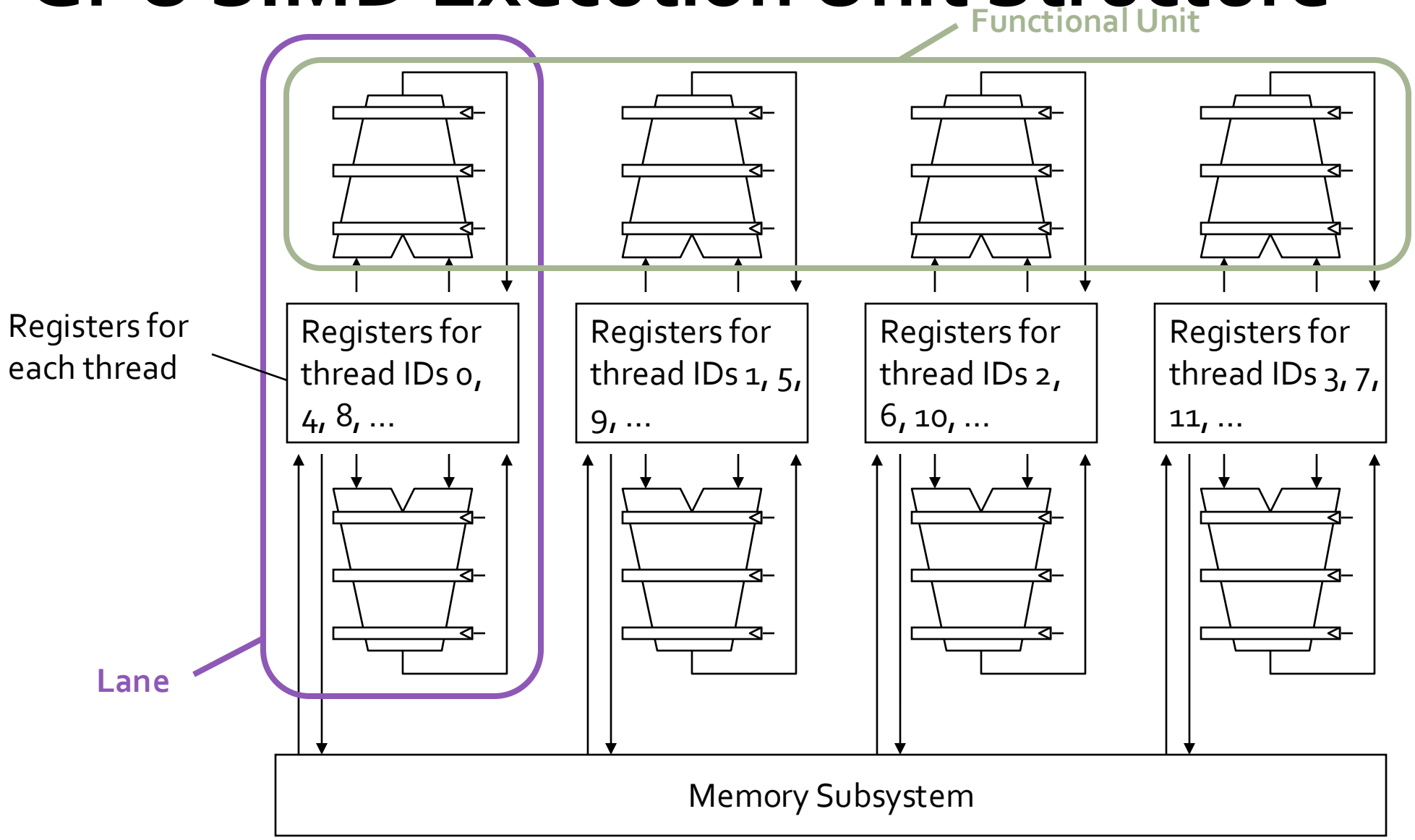
Warp Execution



Vector Unit Structure



GPU SIMD Execution Unit Structure



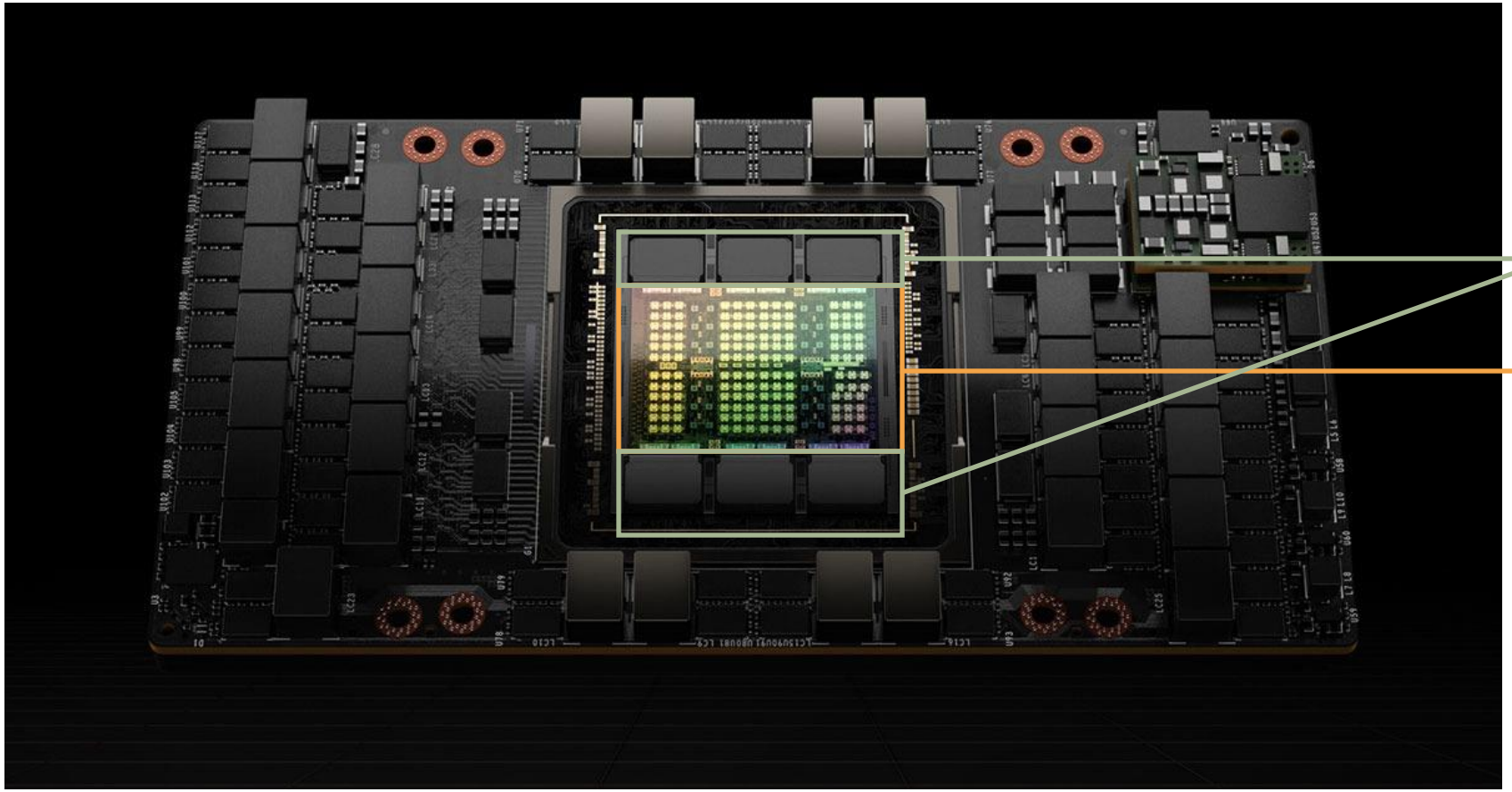
Importance of Machine Learning for GPUs

GPU in Modern Systems

- Discrete GPUs
  - Accelerator separate from CPU
  - PCIe or custom-link connection
  - Separate GPU memory

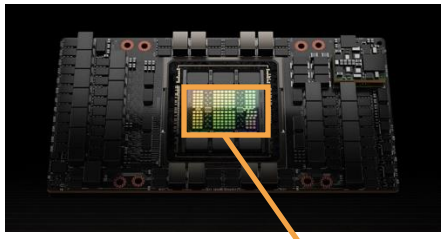
- Integrated GPUs
  - CPU and GPU on same die
  - Shared main memory and last-level cache

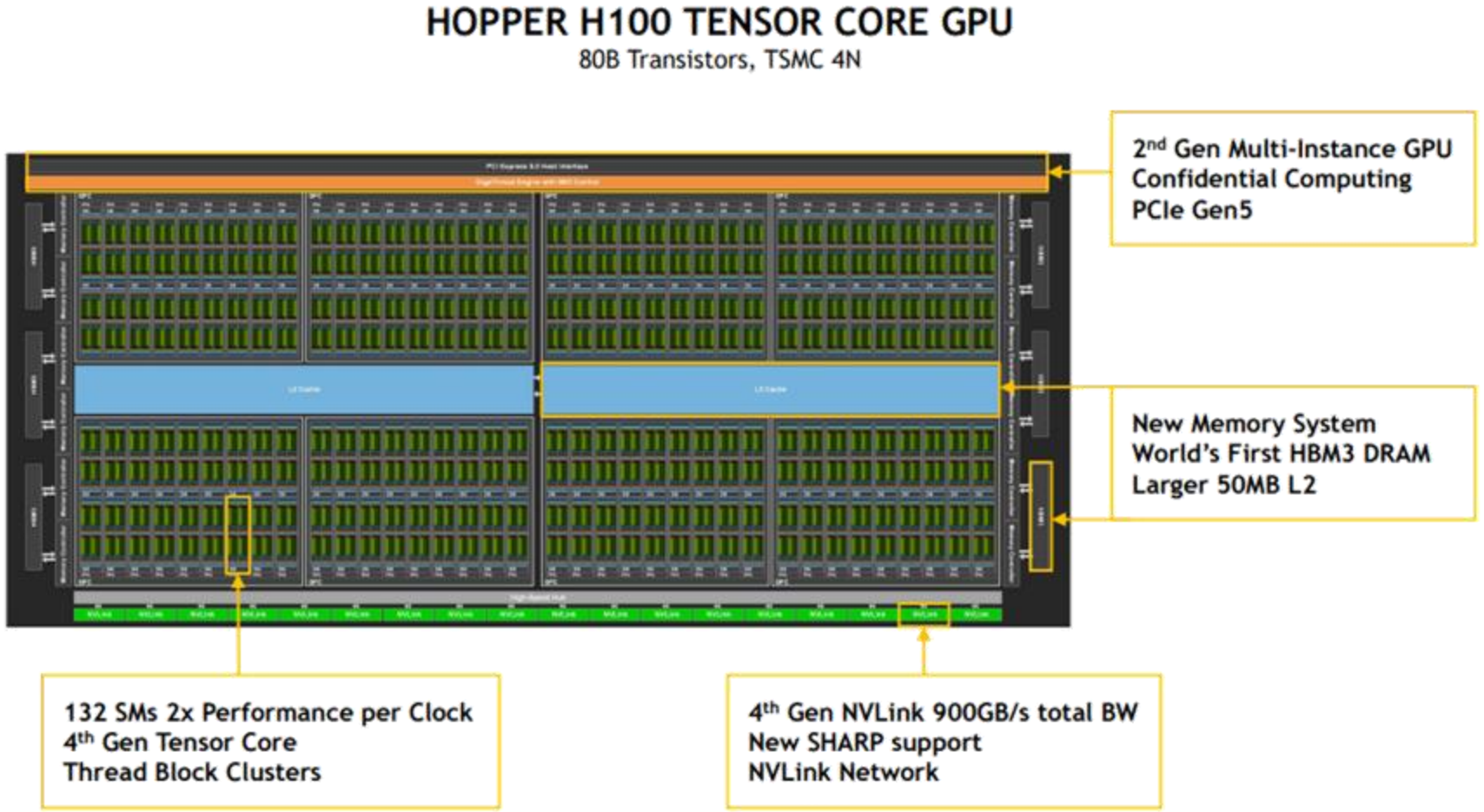
Inside a GPU: NVIDIA H100



[Nvidia]

Inside a GPU: NVIDIA H100

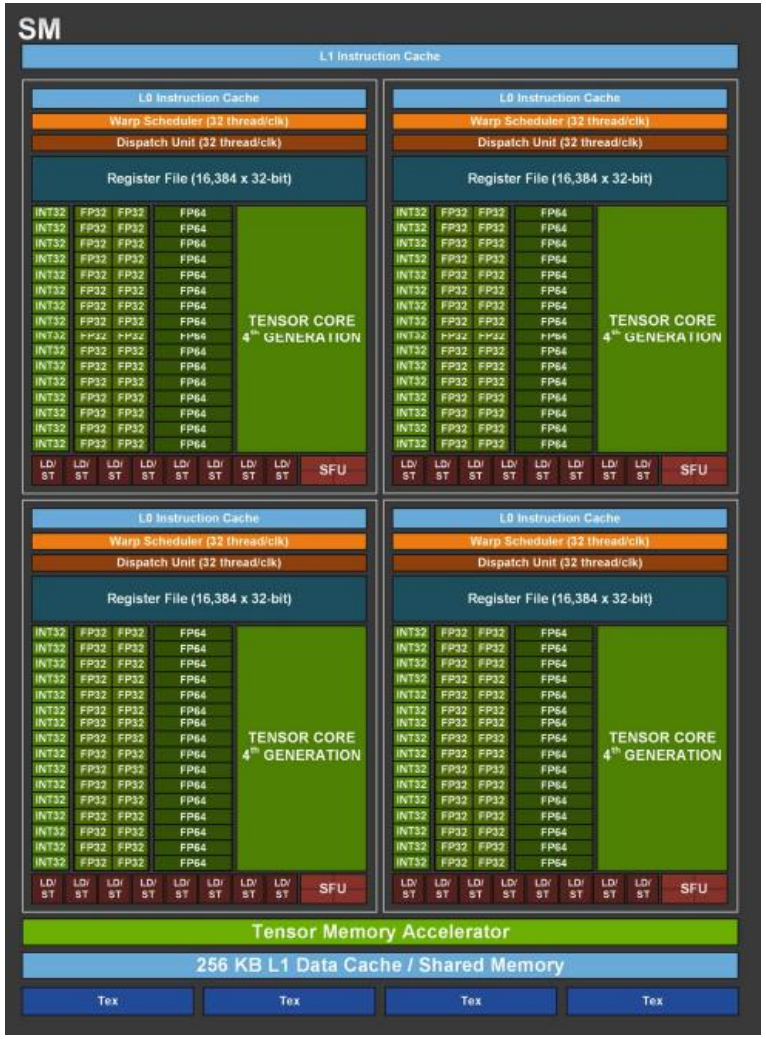




(Actually 144 here; only 132 are enabled due to yield)

[Nvidia, Hotchips, 2022]

Inside a GPU: NVIDIA H100



H100 Streaming Multiprocessor

- 256 KB L1 cache, 256 KB registers

128 FP32 Cores

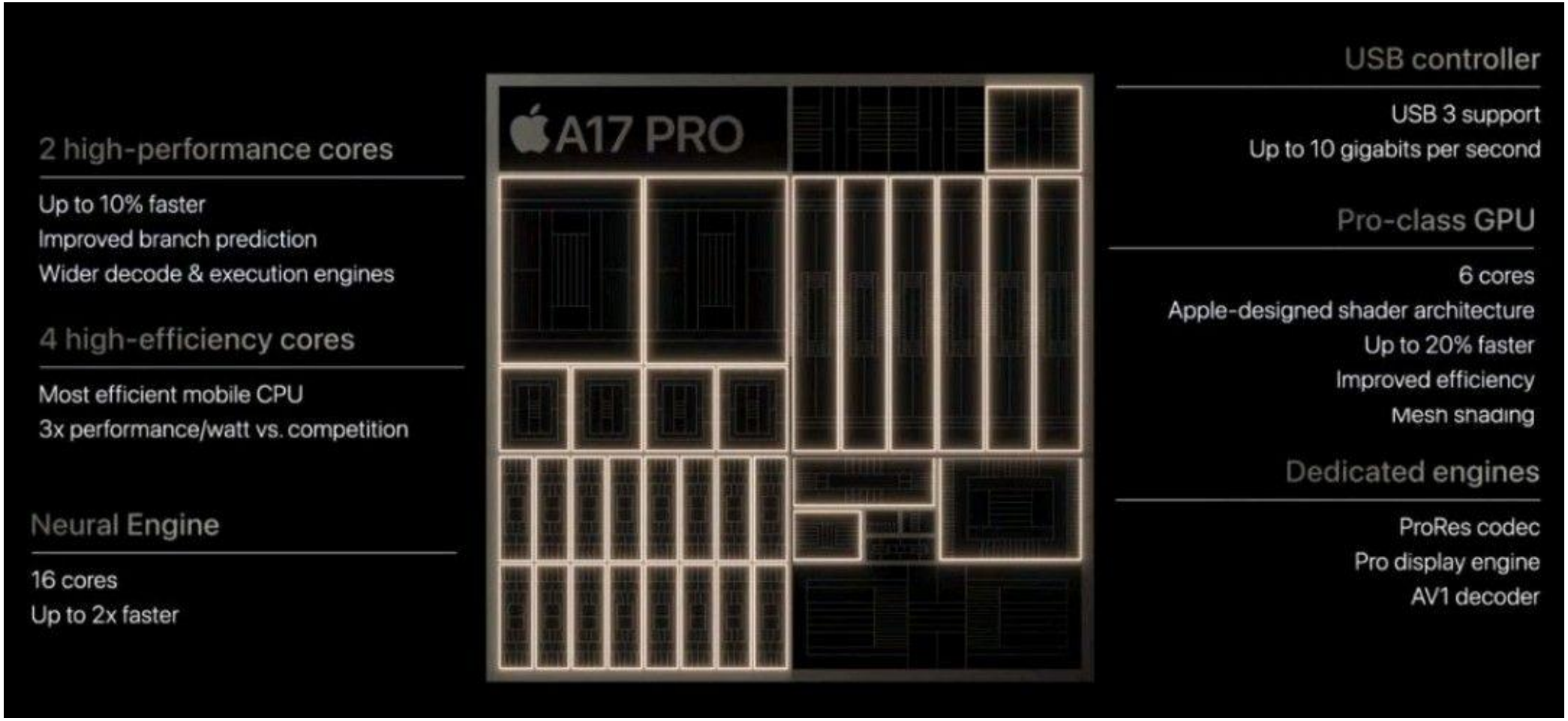
- Computes a*x + b per clock cycle
- 2 FLOPs = Floating Point Operations
- 256 FLOP/cycle per SM

4 Tensor Cores

- Computes AX + B per clock cycle
- Matrix operation: [16x4][4x8] + [16x8]
- 16*4*8*2 = 1024 FLOPs
- 4096 FLOP/cycle per SM
- Mixed precision: 16-bit / 32-bit

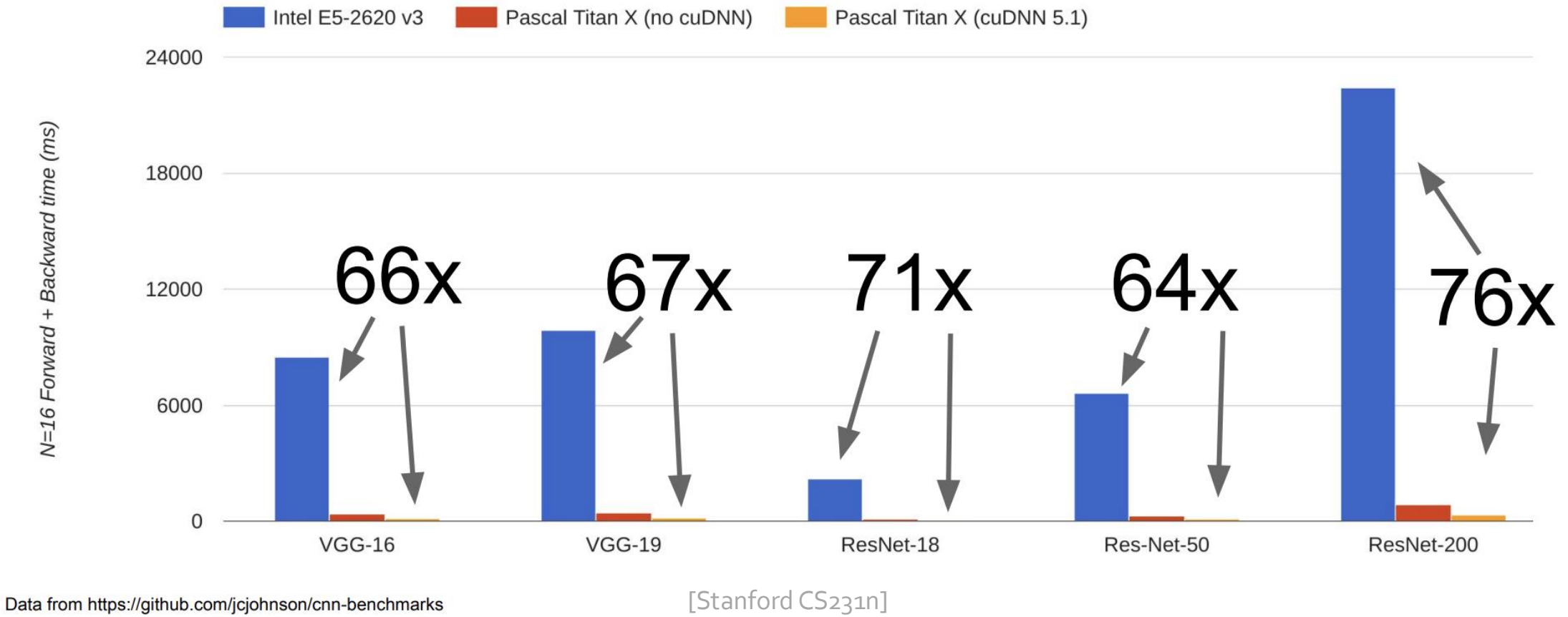
[Nvidia, Hotchips, 2022]

Apple A17 PRO [2023]



[Apple, 2023]

CPU vs GPU Performance



Data from https://github.com/jcjohnson/cnn-benchmarks

[Stanford CS231n]

Ratio of (partially-optimized) CPU vs. CUDA library (cuDNN)

Summary

- Vector Processors
- Single Instruction Multiple Data (SIMD) Instruction Set Extensions
- Graphics Processing Units (GPU)

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Christopher Batten (Cornell)
  - David Wentzlaff (Princeton)
- MIT material derived from course 6.823
- UCB material derived from course CS252
- Cornell material derived from course ECE 4750
- Princeton material derived from course ECE 475