Chapter 4

Data-level Parallelism Vector,SIMD, and GPU

Data/Thread level Parallelism

- Data level parallelism
  - Vector Processor
  - GPU
- Thread level Parallelism
  - SMP/DSM
  - Cache coherence
  - Synchronization

SIMD

- SIMD architectures can exploit significant data-level parallelism for:
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors

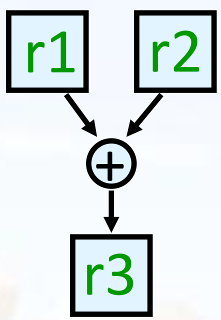
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices

- SIMD allows programmer to continue to think sequentially

Alternative Model: Vector Processing

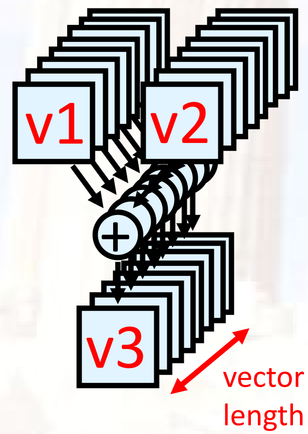
- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"

SCALAR
(1 operation)



`add r3, r1, r2`

VECTOR
(N operations)



`add.vv v3, v1, v2`

Properties of Vector Processors

- Single vector instruction implies lots of work (- loop)
  - fewer instruction fetches
- Each result independent of previous result
  - long pipeline, compiler ensures no dependencies
  - high clock rate
  - hardware does not have to check for data hazards
- Vector instructions that access memory have a known access pattern.
  - highly interleaved memory
  - amortize memory latency of over - 64 elements
  - no (data) caches required!
- Reduces branches and branch problems in pipelines
  - control hazards that would normally arise from the loop branch are nonexistent.

Supercomputer vs. Vector Processor

- CDC6600 (Cray, 1964) regarded as first commercial supercomputer.
- In 70s-80s, Supercomputer ≡ Vector Machine
- **Seymour Cray**
  - Father of supercomputing
  - Founder of company Cray Research

Seymour Cray

Types of Vector Architectures

- *memory-memory vector processors*: all vector operations are memory to memory
  - CDC Star-100 (‘73) , TI ASC (’71)
- *vector-register processors*: all vector operations between vector registers (except load and store)
  - Vector equivalent of load-store architectures
  - Cray-1(1976) was the 1st Vector-Register machine
  - Includes all vector machines since late 1980s: Cray, Convex, Fujitsu, Hitachi, NEC
  - We assume vector-register for rest of lectures

Vector-Register Architectures

- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

Vector Memory-Memory vs. Vector-Register Machines

Example Source Code

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

Vector Memory-Memory Code

```
ADDV C, A, B
SUBV D, A, B
```

Vector Register Code

```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

Vector Memory-Memory Achitecture

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMAs make if difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMAs incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven point was around 2 elements

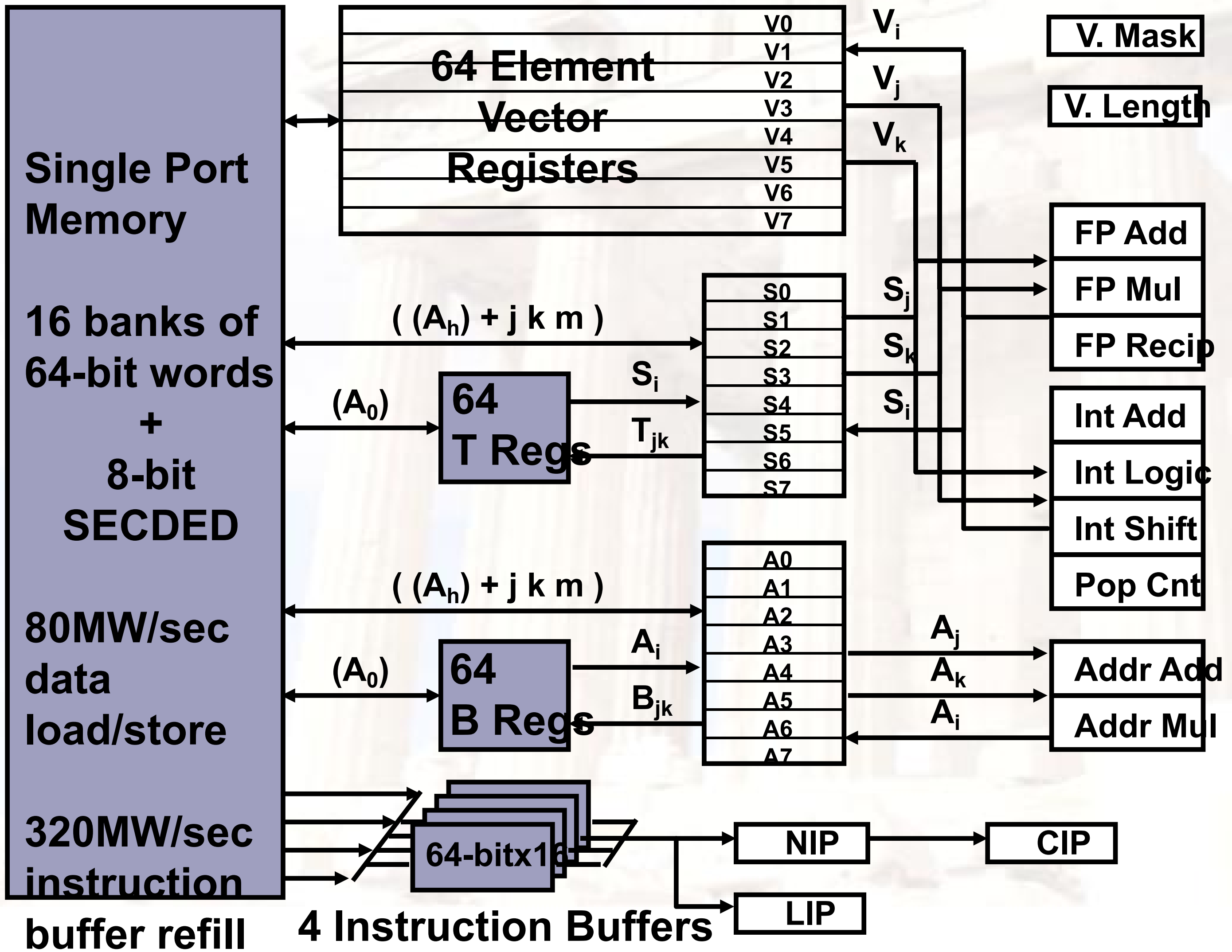
We assume vector-register for rest of the lecture

Cray-1 Breakthrough

- Exquisite electrical and mechanical de
- Semiconductor memory
- Vector register concept
  - vast simplification of instruction set
  - reduced necc. memory bandwidth
- Tight integration of vector and scalar
- Piggy-back off 7600 stacklib
- Later vectorizing compilers developed
- Owned high-performance computing for a decade
  - what happened then?
  - VLIW competition

Cray-1 Block Diagram (1976)

- Scalar Unit + Vector Extensions
- Load/Store Architecture
- Vector Registers
- Simple 16-bit RR Vector Instructions(32-bit with immed)
- Hardwired Control
- Highly Pipelined Functional Units
- Interleaved Memory System
- No Data Caches
- No Virtual Memory



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

Components of Vector Processor

- *Vector Register*: fixed length bank holding a single vector
  - has at least 2 read and 1 write ports
  - typically 8-32 vector registers, each holding 64-128 64-bit elements
- *Vector Functional Units (FUs)*: fully pipelined, start new operation every clock
  - Fully pipelined, start new operation every clock
  - Typically 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift;
  - may have multiple of same unit
- *Vector Load-Store Units (LSUs)*:
  - fully pipelined unit to load or store a vector;
  - Multiple elements fetched/stored per cycle
  - may have multiple LSUs
- *Scalar registers*: single element for FP scalar or address
- Cross-bar to connect FUs , LSUs, registers

Pro. Vs. cons for Vector Processors

- Pro.
  - No dependencies within a vector
    - Pipelining, parallelization, Deep pipelines
  - Each instruction generates a lot of results
    - Small instruction fetch bandwidth
  - Regular Memory access pattern
    - Interleave vector data elements across multiple banks
    - Prefetching a vector is easy
  - Fewer branches in the instruction sequence
- Cons.
  - Very inefficient if parallelism is irregular
  - Memory bandwidth can become a bottleneck

Basic Vector Instructions

Instr.	Operands	Operation	Comment
ADDV	V1,V2,V3	V1=V2+V3	vector + vector
ADDSV	V1,F0,V2	V1=F0+V2	scalar + vector
MULTV	V1,V2,V3	V1=V2xV3	vector x vector
MULSV	V1,F0,V2	V1=F0xV2	scalar x vector
LV	V1,R1	V1=M[R1..R1+63]	load, stride=1
LVWS	V1,R1,R2	V1=M[R1..R1+63*R2]	load, stride=R2
LVI	V1,R1,V2	V1=M[R1+V2i,i=0..63]	indexed "gather"
CeqV	VM,V1,V2	VMASKi = (V1i=V2i)?	comp. setmask
MOV	VLR,R1	Vec. Len. Reg. = R1	set vector length
MOV	VM,R1	Vec. Mask = R1	set vector mask

+ all the regular scalar instructions (RISC style)…

Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies

- RV64V functional units consume one element per clock cycle
  - Execution time is approximately the vector length

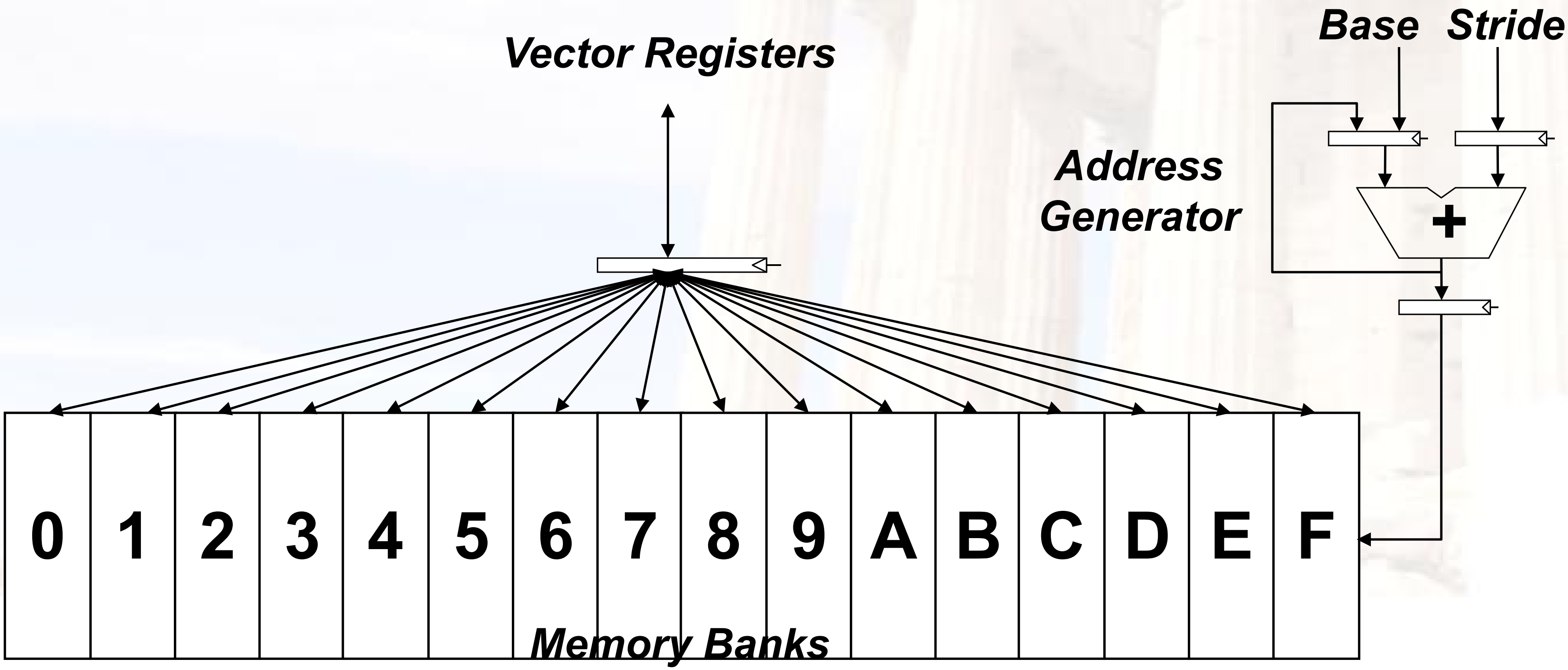
Vector Memory operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - Fastest
  - Non-unit (constant) stride
  - Indexed (gather-scatter)
    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize
    - compress/expand variant also
- Support for various combinations of data widths in memory
  - {.L,.W,.H.,.B} x {64b, 32b, 16b, 8b}

Vector Memory System

Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- ***Bank busy time***: Cycles between accesses to same bank



DAXPY (Y = a * X + Y)

Assuming vectors X, Y are length 64

Scalar vs. Vector

```
LD      F0,a      ;load scalar a
LV      V1,Rx     ;load vector X
MULTS   V2,F0,V1  ;vector-scalar mult.
LV      V3,Ry     ;load vector Y
ADDV    V4,V2,V3  ;add
SV      Ry,V4     ;store the result
```

```
      LD      F0,a
      ADDI    R4,Rx,#512  ;last address to load
loop: LD      F2, 0(Rx)   ;load X(i)
      MULTD   F2,F0,F2    ;a*X(i)
      LD      F4, 0(Ry)   ;load Y(i)
      ADDD    F4,F2, F4   ;a*X(i) + Y(i)
      SD      F4 ,0(Ry)   ;store into Y(i)
      ADDI    Rx,Rx,#8    ;increment index to X
      ADDI    Ry,Ry,#8    ;increment index to Y
      SUB     R20,R4,Rx   ;compute bound
      BNZ     R20,loop    ;check if done
```

578 (2+9*64) vs.
321 (1+5*64) ops (1.8X)

578 (2+9*64) vs.
6 instructions (96X)

64 operation vectors + no loop overhead

also 64X fewer pipeline hazards

ZHEJIANG UNIVERSITY

Vector Length

- A vector register can hold some maximum number of elements for each data width (maximum vector length or MVL)
- What to do when the application vector length is not exactly MVL?
- Vector-length (VL) register controls the length of any vector operation, including a vector load or store
  - E.g. vadd.vv with VL=10 is

```
for (I=0; I<10; I++) V1[I]=V2[I]+V3[I]
```

- VL can be anything from 0 to MVL

- How do you code an application where the vector length is not known until run-time?

Strip Mining

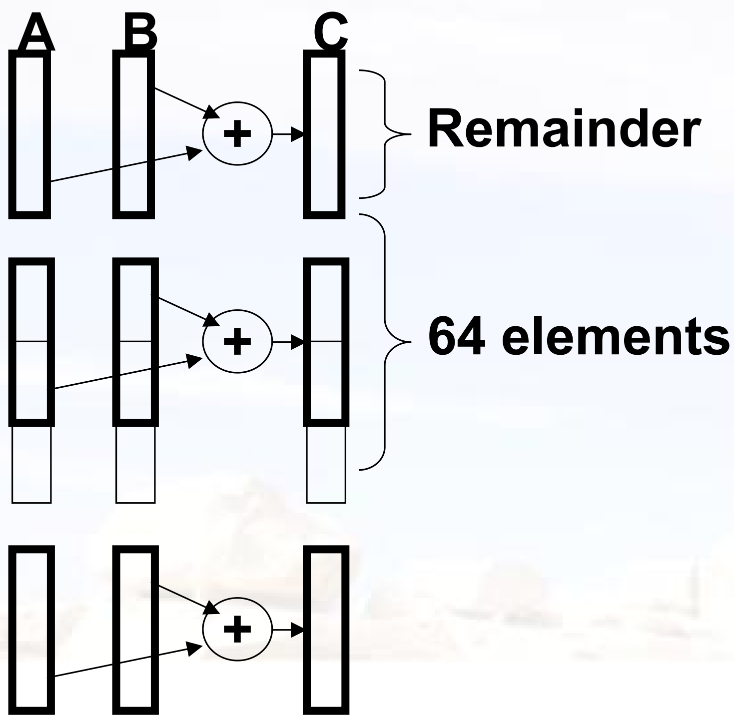
- Suppose application vector length > MVL
- Strip mining
  - Generation of a loop that handles MVL elements per iteration
  - A set operations on MVL elements is translated to a single vector instruction
- Example: vector daxpy of N elements
  - First loop handles (N mod MVL) elements, the rest handle MVL

```
VL = (N mod MVL);             // set VL = N mod MVL
for (I=0; I<VL; I++)          // 1st loop is a single set of
    Y[I]=A*X[I]+Y[I];         //   vector instructions
low = (N mod MVL);
VL = MVL;                     // set VL to MVL
for (I=low; I<N; I++)         // 2nd loop requires N/MVL
    Y[I]=A*X[I]+Y[I];         //   sets of vector instructions
```

example for Strip Mining

Max Vector Length is fixed ！

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




```
 ANDI R1, N, #63      ; N mod 64
 MTC1 VLR, R1       ; Do remainder
loop:
 LV V1, RA
 DSLL R2, R1, #3      ; Multiply by 8
 DADDU RA, RA, R2  ; Bump pointer
 LV V2, RB
 DADDU RB, RB, R2
 ADDV.D V3, V1, V2
 SV V3, RC
 DADDU RC, RC, R2
 DSUBU N, N, R1 ; Subtract elements
 LI R1, #64
 MTC1 VLR, R1     ; Reset full length
 BGTZ N, loop     ; Any more to do?
```

New in RISC V

- MVL is decided by Hardware

```
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
  0:  li t0, 2<<25
  4:  vsetdcfg t0              # enable 2 64b Fl.Pt. registers
loop:
  8:  setvl  t0, a0            # vl = t0 = min(mvl, n)
  c:  vld    v0, a1            # load vector x
 10:  slli   t1, t0, 3         # t1 = vl * 8 (in bytes)
 14:  vld    v1, a2            # load vector y
 18:  add    a1, a1, t1        # increment C pointer to x by vl*8
 1c:  vfmadd v1, v0, fa0, v1   # v1 += v0 * fa0 (y = a * x + y)
 20:  sub    a0, a0, t0        # n -= vl (t0)
 24:  vst    v1, a2            # store Y
 28:  add    a2, a2, t1        # increment C pointer to y by vl*8
 2c:  bnez   a0, loop          # repeat if n != 0
 30:  ret                      # return
```

Challenges

- Start up time
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add => 6 clock cycles
    - Floating-point multiply => 7 clock cycles
    - Floating-point divide => 20 clock cycles
    - Vector load => 12 clock cycles
- Improvements:
  - > 1 element per clock cycle
  - Non-64 wide vectors
  - IF statements in vector code
  - Memory system optimizations to support vector processors
  - Multiple dimensional matrices
  - Sparse matrices
  - Programming a vector computer

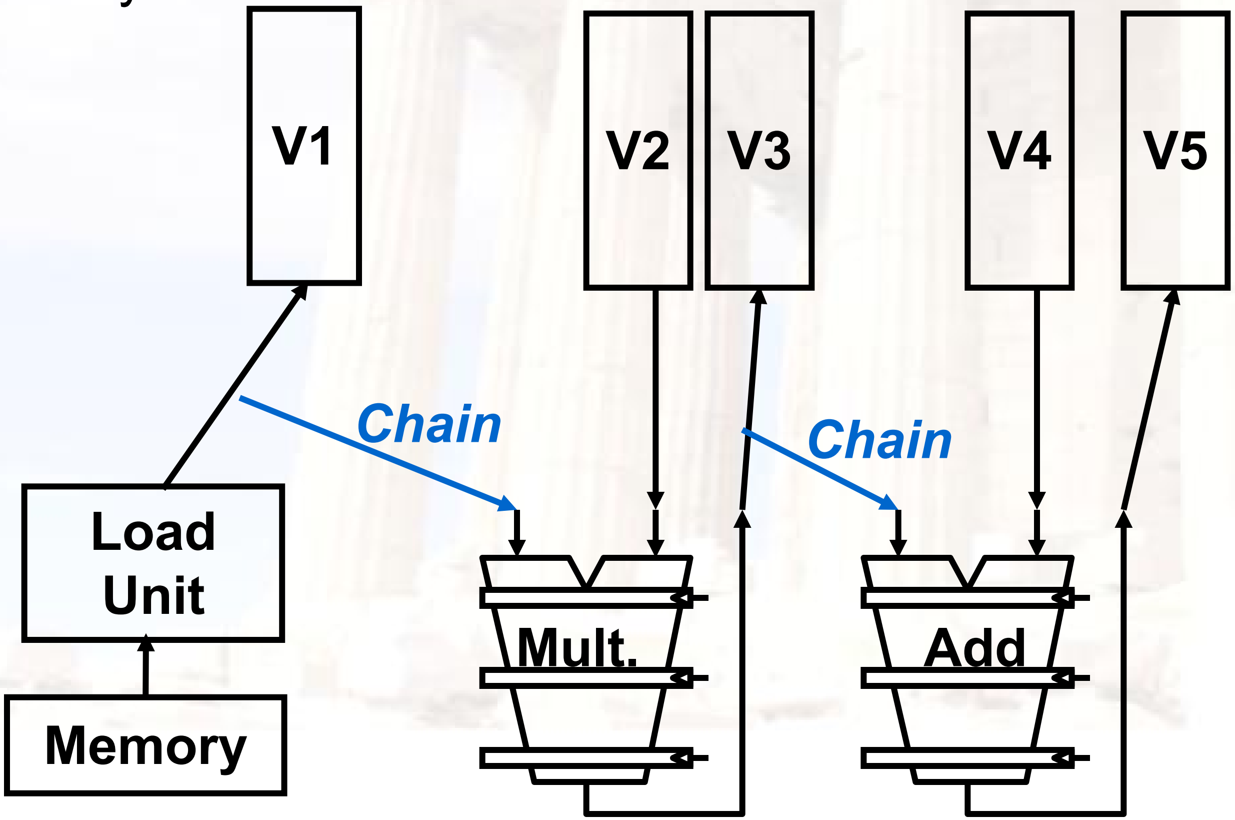
Optimizing Vector Performance

- Vector Chaining
- Conditionally Executed Statements
- Sparse Matrices
- Multiple Lanes

Optimization 1: Vector Chaining

- the Concept of Forwarding Extended to Vector Registers
- Vector version of register bypassing
  - introduced with Cray-1

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



ZHEJIANG UNIVERSITY 浙江大學

Vector Chaining Advantage

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

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

ZHEJIANG UNIVERSITY

Convey & Chimes

- Convey: Set of vector instructions that could potentially execute together
- Chimes: Sequences with read-after-write dependency hazards placed in same convey via *chaining*
- *Chaining*
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
- *Chime*
  - Unit of time to execute one convey
  - m conveys executes in m chimes for vector length n
  - For vector length of n, requires m x n clock cycles

Example (single mem access unit)

```
vld     v0,x5       # Load vector X
vmul    v1,v0,f0    # Vector-scalar multiply
vld     v2,x6       # Load vector Y
vadd    v3,v1,v2    # Vector-vector add
vst     v3,x6       # Store the sum
```

Convoys:

1	vld	vmul
2	vld	vadd
3	vst	

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

For 32 element vectors, requires 32 x 3 = 96 clock cycles

浙江大學 ZHEJIANG UNIVERSITY

Optimization 2: Conditional Execution

- Suppose you want to vectorize this:

```
for (I=0; I<N; I++)
    if (A[I]!= B[I]) A[I] -= B[I];
```

- Solution: vector conditional execution
  - Add vector flag registers with single-bit elements
  - Use a vector compare to set the a flag register
  - Use flag register as mask control for the vector sub
    - Addition executed only for vector elements with corresponding flag element set

RISCV:
Vpeq, Vpne, vplt, vpgt

- Vector code

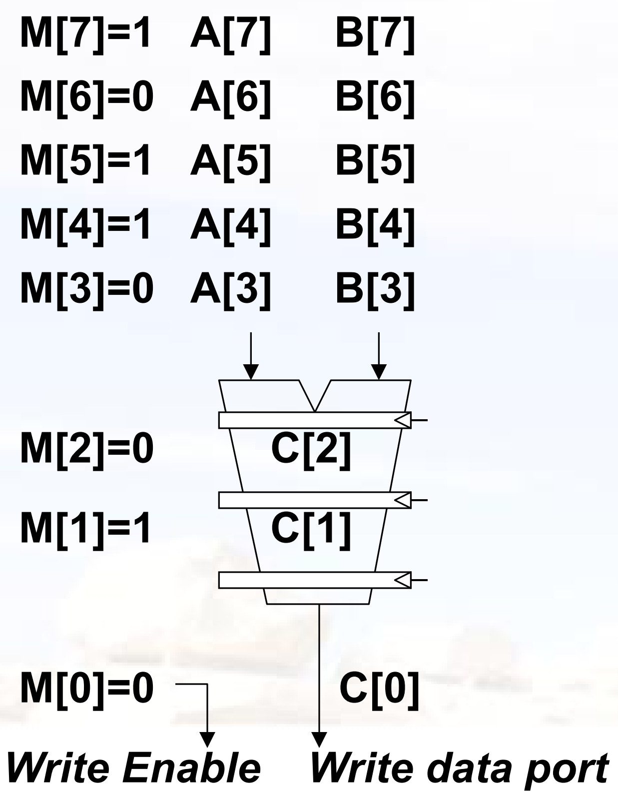
```
vld           V1, Ra
vld           V2, Rb
vcmp.neq.vv    F0, V1, V2     # vector compare
vsub.vv       V3, V2, V1, F0 # conditional vadd
vst           V3, Ra
```

–Cray uses vector mask & merge

Masked Vector Instructions

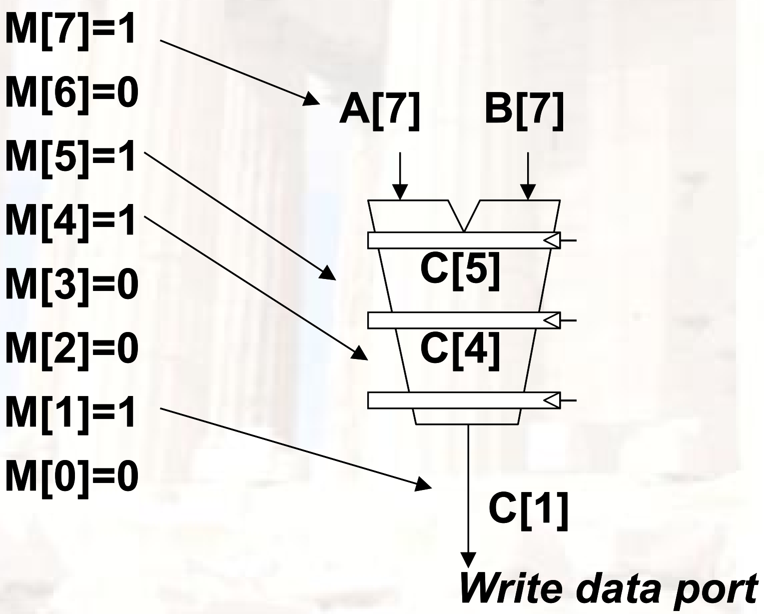
Simple Implementation

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



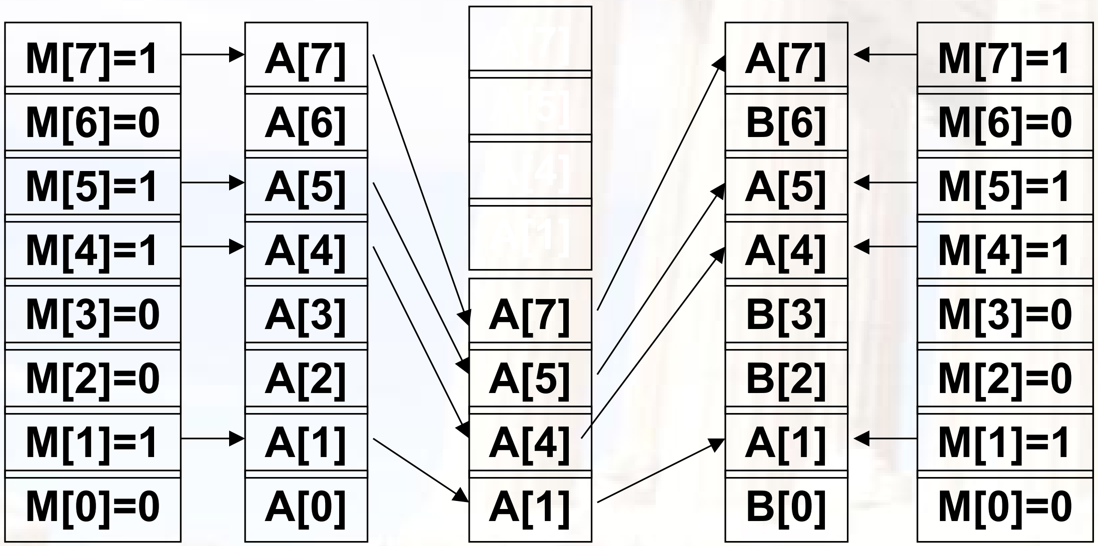
Density-Time Implementation

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



Compress/Expand Operations

- Compress packs non-masked elements from one vector register contiguously at start of destination vector register
  - population count of mask vector gives packed vector length
- Expand performs inverse operation



Compress ***Expand***

Used for density-time conditionals and also for general selection operations

Optimization 3: sparce matrices

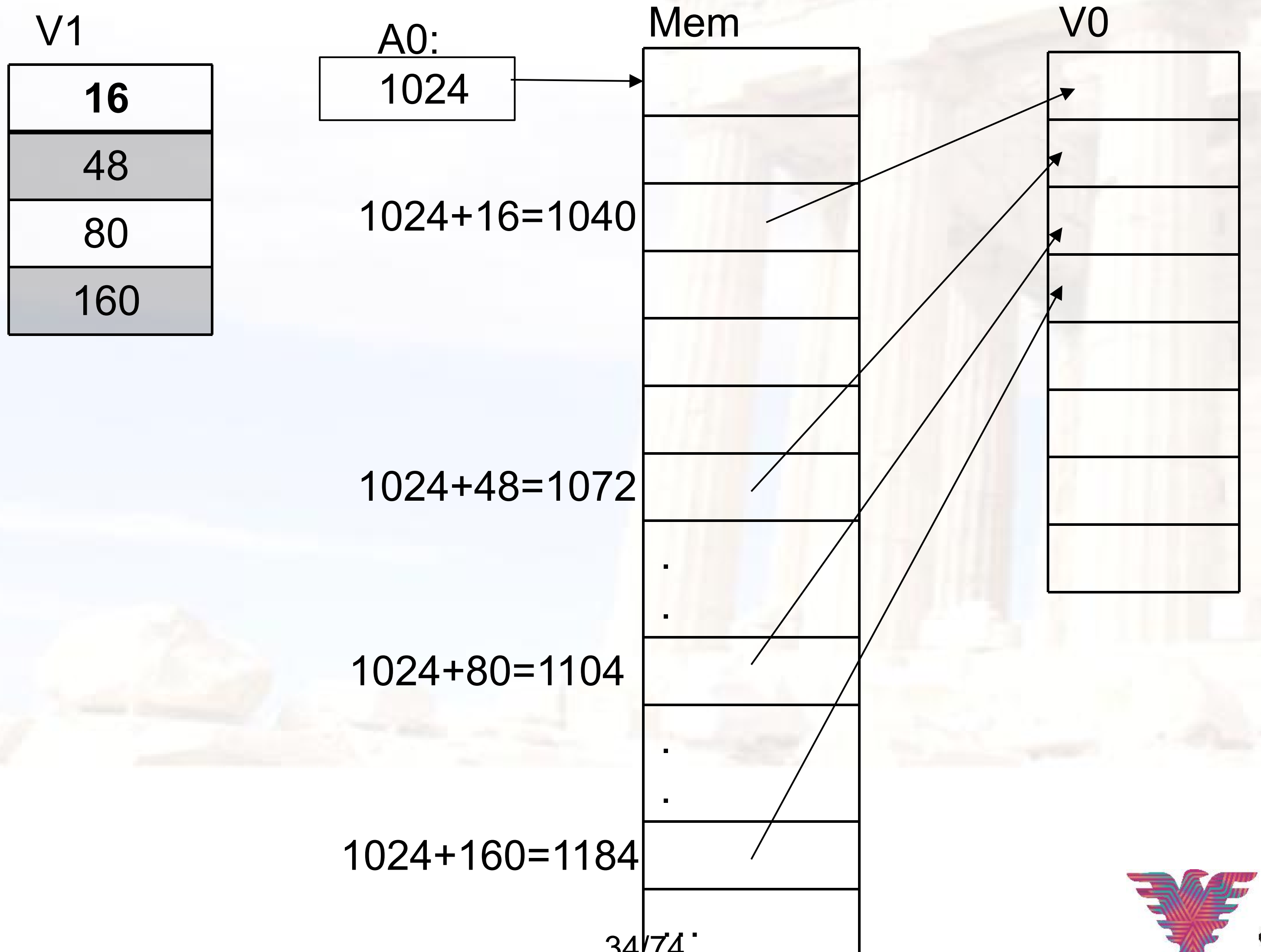
$$A_{4\times 5}=\begin{bmatrix} 0 & 5 & 0 & 0 & 5 \\ 1 & 0 & 3 & 0 & 0 \\ 0 & -2 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a)

	i	j	v
0	0	1	5
1	0	4	5
2	1	0	1
⋮	1	2	3
⋮	2	1	-2
a→t-1	3	0	6
⋮			
MaxSize-1			

Indexed load (gather) vs indexed store (scatter)

- vldx v0, a0, v1 vstx v0, a0, v1



Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

(index vector D designate the nonzero elements of *C*)

```
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
```

Vector Scatter/Gather

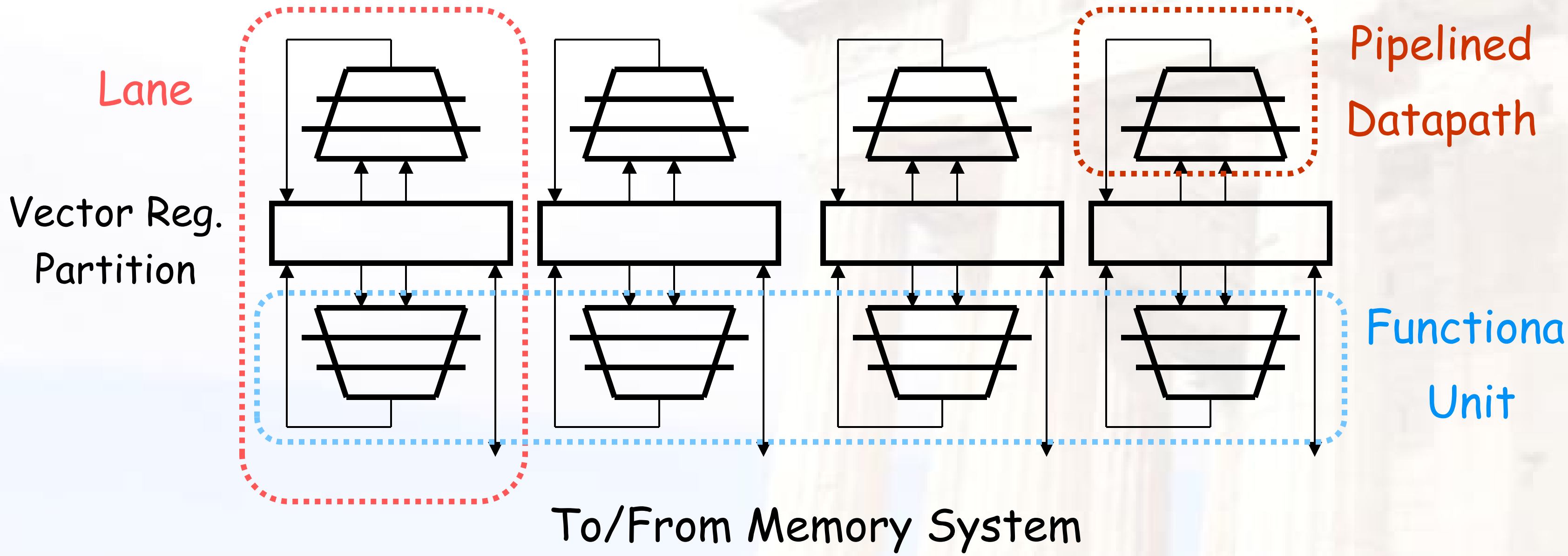
Scatter example:

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

Is following a correct translation?

```
LV      VB, RB       ; Load indices in B vector
LVI     VA,(RA, VB)  ; Gather initial A values
ADDV    VA, RA, 1    ; Increment
SVI     VA,(RA, VB)  ; Scatter incremented values
```

Optimization 4: Multi-lane Implementation



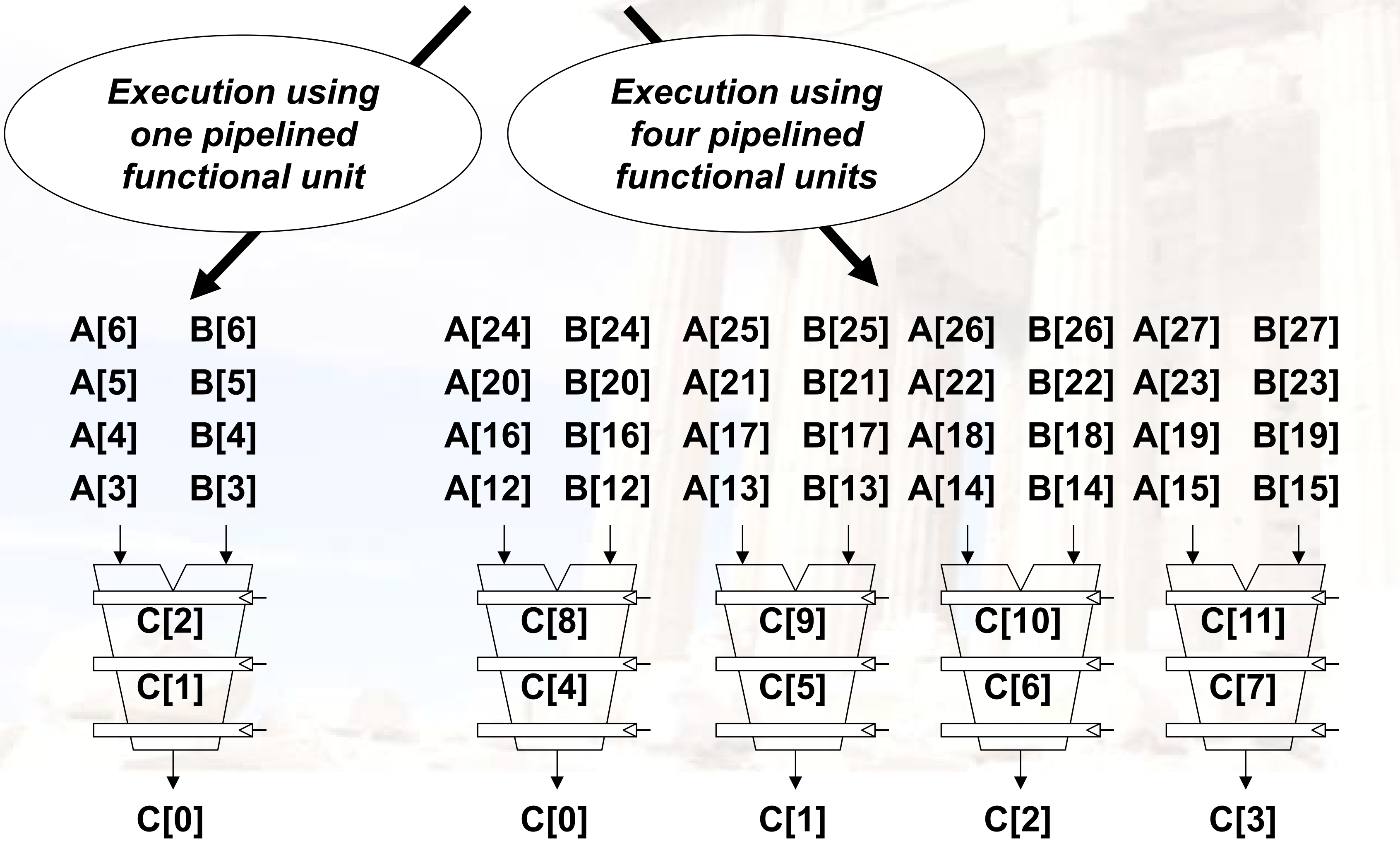
- Elements for vector registers interleaved across the lanes
- Each lane receives identical control
- Multiple element operations executed per cycle
- Modular, scalable design
- No need for inter-lane communication for most vector instructions

ZHEJIANG UNIVERSITY

Multiple Lanes

Vector Instruction Execution

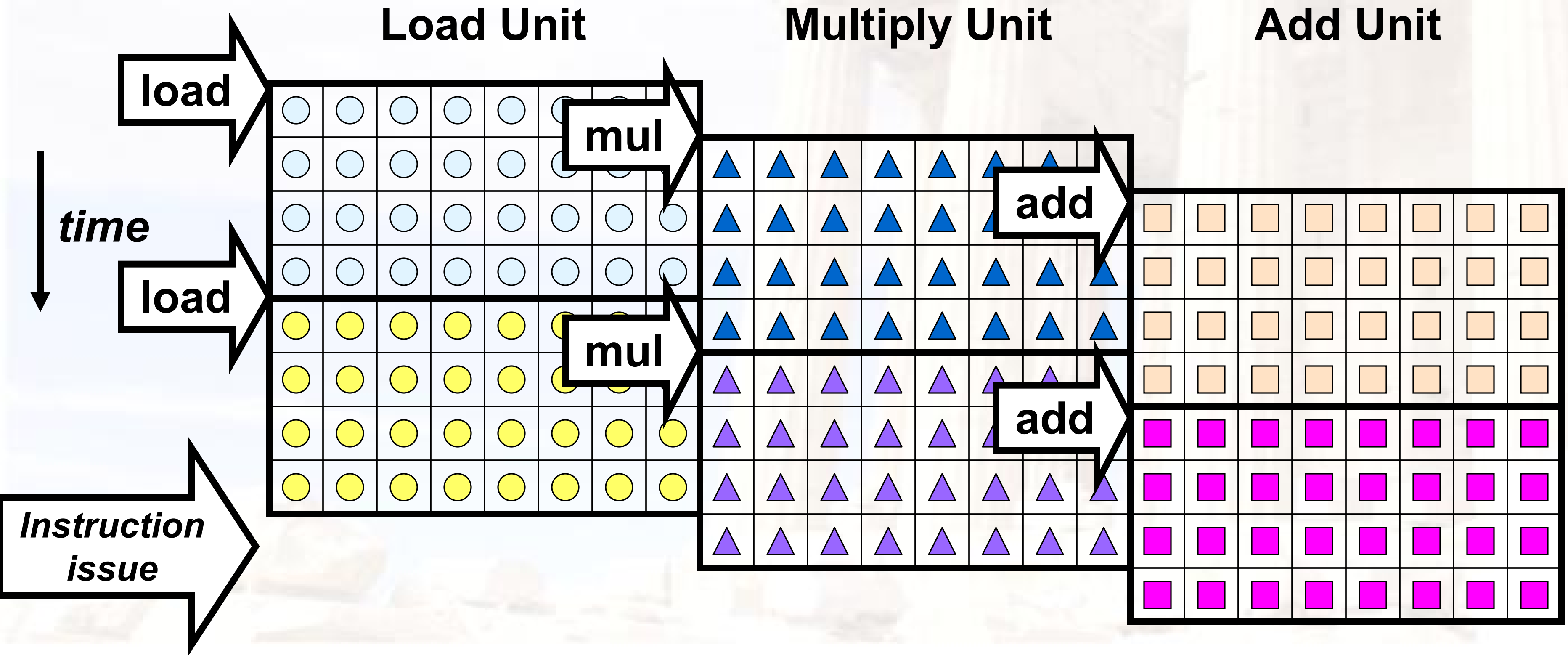
`ADDV C,A,B`



Chain & Multiple Lane

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

Two Ways to View Vectorization

- Inner loop vectorization (Classic approach)
  - Think of machine as, say, 32 vector registers each with 16 elements
  - 1 instruction updates 32 elements of 1 vector register
  - Good for vectorizing single-dimension arrays or regular kernels (e.g. saxpy)
- Outer loop vectorization (post-CM2)
  - Think of machine as 16 “virtual processors” (VPs) each with 32 scalar registers! (- multithreaded processor)
  - 1 instruction updates 1 scalar register in 16 VPs
  - Good for irregular kernels or kernels with loop-carried dependences in the inner loop
- These are just two compiler perspectives
  - The hardware is the same for both

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words (need independent bank addressing
  - Support multiple vector processors sharing the same memory
- Example:
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - How many memory banks needed?
    - 32x(4+2)x15/2.167 = ~1330 banks

Example Vector Machines

Machine	Year	Clock	Regs	Elements	FUs	LSUs
Cray 1	1976	80 MHz	8	64	6	1
Cray XMP	1983	120 MHz	8	64	8	2 L, 1 S
Cray YMP	1988	166 MHz	8	64	8	2 L, 1 S
Cray C-90	1991	240 MHz	8	128	8	4
Cray T-90	1996	455 MHz	8	128	8	4
Conv. C-1	1984	10 MHz	8	128	4	1
Conv. C-4	1994	133 MHz	16	128	3	1
Fuj. VP200	1982	133 MHz	8-256	32-1024	3	2
Fuj. VP300	1996	100 MHz	8-256	32-1024	3	2
NEC SX/2	1984	160 MHz	8+8K	256+var	16	8
NEC SX/3	1995	400 MHz	8+8K	256+var	16	8

Vector Linpack Performance(MFLOPS)

Machine	Year	Clock	100x100	1kx1k	Peak(Procs)
Cray 1	1976	80 MHz	12	110	160(1)
Cray XMP	1983	120 MHz	121	218	940(4)
Cray YMP	1988	166 MHz	150	307	2,667(8)
Cray C-90	1991	240 MHz	387	902	15,238(16)
Cray T-90	1996	455 MHz	705	1603	57,600(32)
Conv. C-1	1984	10 MHz	3	--	20(1)
Conv. C-4	1994	135 MHz	160	2531	3240(4)
Fuj. VP200	1982	133 MHz	18	422	533(1)
NEC SX/2	1984	166 MHz	43	885	1300(1)
NEC SX/3	1995	400 MHz	368	2757	25,600(4)

Operation & Instruction Count: RISC v. Vector Processor

Spec92fp	Operations (Millions)			Instructions (M)		
Program	RISC	Vector	R / V	RISC	Vector	R / V
swim256	115	95	1.1x	115	0.8	142x
hydro2d	58	40	1.4x	58	0.8	71x
nasa7	69	41	1.7x	69	2.2	31x
su2cor	51	35	1.4x	51	1.8	29x
tomcatv	15	10	1.4x	15	1.3	11x
wave5	27	25	1.1x	27	7.2	4x
mdljdp2	32	52	0.6x	32	15.8	2x

Vector reduces ops by 1.2X, instructions by 20X

Vector Advantages

- Easy to get high performance; N operations:
  - are independent
  - use same functional unit
  - access disjoint registers
  - access registers in same order as previous instructions
  - access contiguous memory words or known pattern
  - can exploit large memory bandwidth
  - hide memory latency (and any other latency)
- Scalable: (get higher performance by adding HW resources)
- Compact: Describe N operations with 1 short instruction
- Predictable: performance vs. statistical performance (cache)
- Multimedia ready: N * 64b, 2N * 32b, 4N * 16b, 8N * 8b
- Mature, developed compiler technology

Programming Vec. Architectures

- Compilers can provide feedback to programmers
- Programmers can provide hints to compiler

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Vector Pitfalls

- Pitfall: Concentrating on peak performance and ignoring start-up overhead:

  N_V (length faster than scalar) > 100!

- Pitfall: Increasing vector performance, without comparable increases in scalar performance (Amdahl's Law)
  - failure of Cray competitor (ETA) from his former company
- Pitfall: Good processor vector performance without providing good memory bandwidth
  - MMX?

Vector Disadvantage: Out of Fashion?

- Hard to say. Many irregular loop structures seem to still be hard to vectorize automatically.
- Theory of some researchers that SIMD model has great potential.

SIMD Extensions

- Media applications operate on data types narrower than the native word size
  - Example: disconnect carry chains to “partition” adder

- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers

SIMD Implementations

- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
  - Advanced Vector Extensions (2010)
    - Four 64-bit integer/fp ops
  - AVX-512 (2017)
    - Eight 64-bit integer/fp ops
  - Operands must be consecutive and aligned memory locations

Example SIMD Code

- Example DXPY:

```
      fld        f0,a            # Load scalar a
      splat.4D   f0,f0           # Make 4 copies of a
      addi       x28,x5,#256     # Last address to load
Loop: fld.4D     f1,0(x5)        # Load X[i] ... X[i+3]
      fmul.4D    f1,f1,f0        # a x X[i] ... a x X[i+3]
      fld.4D     f2,0(x6)        # Load Y[i] ... Y[i+3]
      fadd.4D    f2,f2,f1        # a x X[i]+Y[i]...
                                 # a x X[i+3]+Y[i+3]
      fsd.4D     f2,0(x6)        # Store Y[i]... Y[i+3]
      addi       x5,x5,#32       # Increment index to X
      addi       x6,x6,#32       # Increment index to Y
      bne        x28,x5,Loop     # Check if done
```

Graphical Processing Units

- **Basic idea:**
  - Heterogeneous execution model
    - CPU is the *host*,   GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model is "Single Instruction Multiple Thread"

Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks
- Blocks are organized into a grid

- GPU hardware handles thread management, not applications or OS

NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

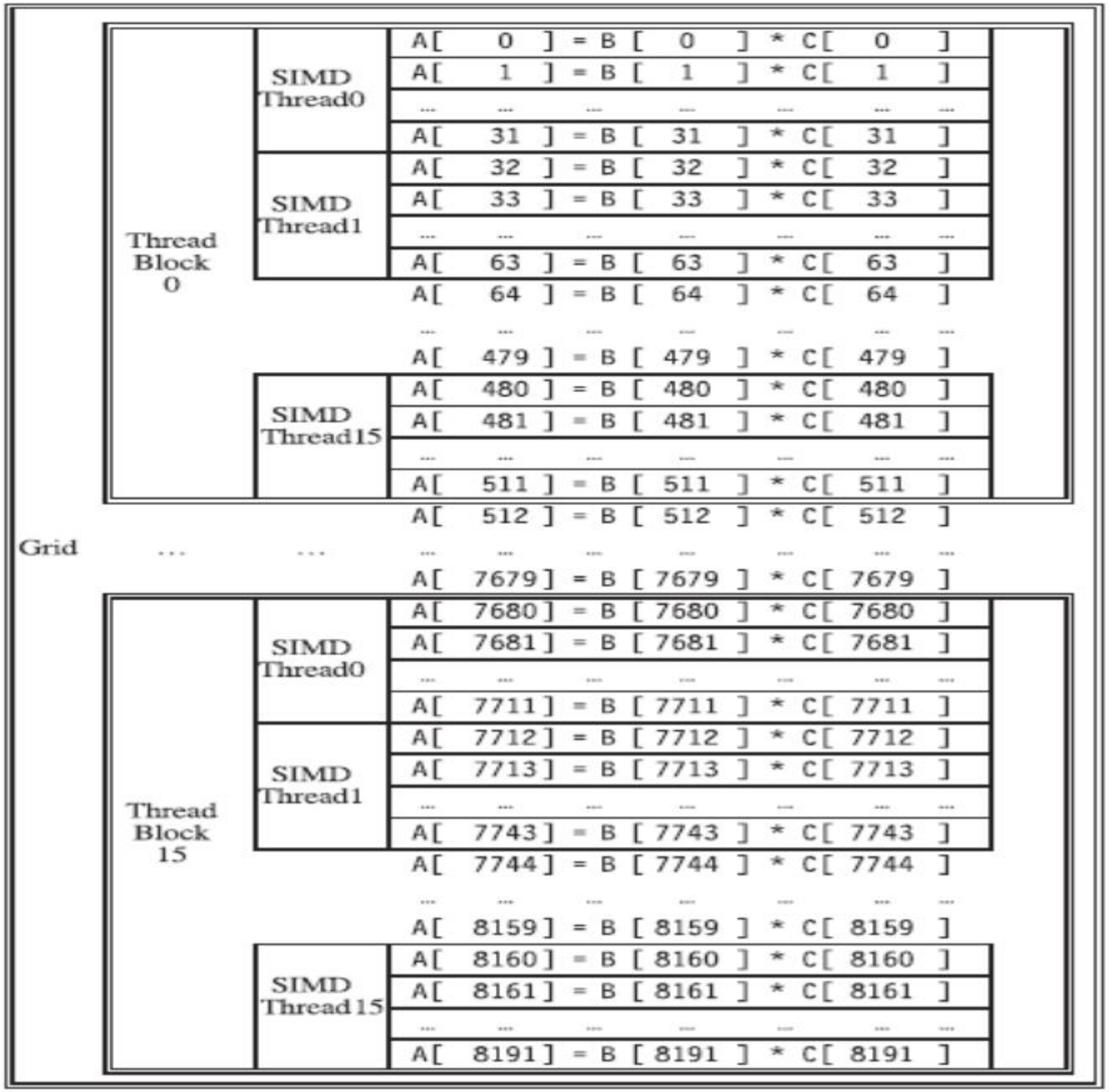
Example

- Code that works over all elements is the grid
- Thread blocks break this down into manageable sizes
  - 512 threads per block
- SIMD instruction executes 32 elements at a time
- Thus grid size = 16 blocks
- Block is analogous to a strip-mined vector loop with vector length of 32
- Block is assigned to a multithreaded SIMD processor by the thread block scheduler
- Current-generation GPUs have 7-15 multithreaded SIMD processors

Terminology

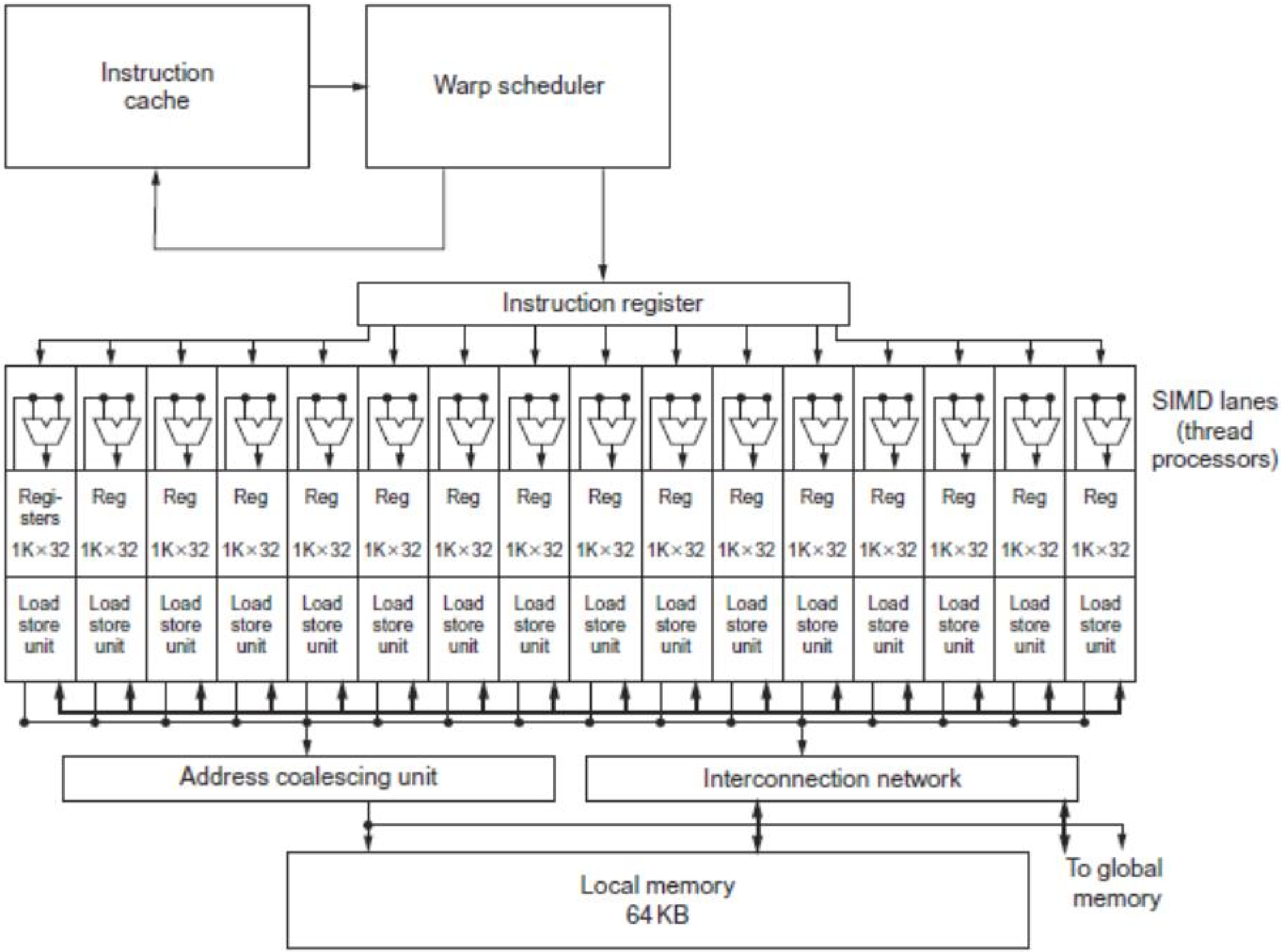
- Each thread is limited to 64 registers
- Groups of 32 threads combined into a SIMD thread or "warp"
  - Mapped to 16 physical lanes
- Up to 32 warps are scheduled on a single SIMD processor
  - Each warp has its own PC
  - Thread scheduler uses scoreboard to dispatch warps
  - By definition, no data dependencies between warps
  - Dispatch warps into pipeline, hide memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - 32 SIMD lanes
  - Wide and shallow compared to vector processors

Example



ZHEJIANG UNIVERSITY

GPU Organization



NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
  - "Parallel Thread Execution (PTX)"
    - opcode.type d,a,b,c;
  - Uses virtual registers
  - Translation to machine code is performed in software
  - Example:

```
shl.s32         R8, blockIdx, 9       ; Thread Block ID * Block size (512 or 29)
add.s32         R8, R8, threadIdx     ; R8 = i = my CUDA thread ID
ld.global.f64   RD0, [X+R8]           ; RD0 = X[i]
ld.global.f64   RD2, [Y+R8]           ; RD2 = Y[i]
mul.f64         RD0, RD0, RD4         ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64         RD0, RD0, RD2         ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64   [Y+R8], RD0           ; Y[i] = sum (X[i]*a + Y[i])
```

Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

Example

```
if (X[i] != 0)
        X[i] = X[i] – Y[i];
else X[i] = Z[i];

     ld.global.f64     RD0, [X+R8]              ; RD0 = X[i]
     setp.neq.s32      P1, RD0, #0              ; P1 is predicate register 1
     @!P1, bra         ELSE1, *Push             ; Push old mask, set new mask bits
                                                ; if P1 false, go to ELSE1
     ld.global.f64     RD2, [Y+R8]              ; RD2 = Y[i]
     sub.f64           RD0, RD0, RD2            ; Difference in RD0
     st.global.f64     [X+R8], RD0              ; X[i] = RD0
     @P1, bra          ENDIF1, *Comp            ; complement mask bits
                                                ; if P1 true, go to ENDIF1
ELSE1:                 ld.global.f64 RD0, [Z+R8]  ; RD0 = Z[i]
                       st.global.f64 [X+R8], RD0  ; X[i] = RD0
ENDIF1: <next instruction>, *Pop                ; pop to restore old mask
```

NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - "Private memory"
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write GPU memory

Pascal Multithreaded SIMD Proc.

Vector Architectures vs GPUs

- SIMD processor analogous to vector processor, both have MIMD
- Registers
  - RV64V register file holds entire vectors
  - GPU distributes vectors across the registers of SIMD lanes
  - RV64 has 32 vector registers of 32 elements (1024)
  - GPU has 256 registers with 32 elements each (8K)
  - RV64 has 2 to 8 lanes with vector length of 32, chime is 4 to 16 cycles
  - SIMD processor chime is 2 to 4 cycles
  - GPU vectorized loop is grid
  - All GPU loads are gather instructions and all GPU stores are scatter instructions

SIMD Architectures vs GPUs

- GPUs have more SIMD lanes
- GPUs have hardware support for more threads
- Both have 2:1 ratio between double- and single-precision performance
- Both have 64-bit addresses, but GPUs have smaller memory
- SIMD architectures have no scatter-gather support

Loop-Level Parallelism

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence

- Example 1:

```
for (i=999; i>=0; i=i-1)
        x[i] = x[i] + s;
```

- No loop-carried dependence

Loop-Level Parallelism

- Example 2:

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i]; /* S1 */
        B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

- S1 and S2 use values computed by S1 in previous iteration
- S2 uses value computed by S1 in same iteration

Loop-Level Parallelism

Example 3:

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i]; /* S1 */
        B[i+1] = C[i] + D[i]; /* S2 */
}
```

- S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel
- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Loop-Level Parallelism

- Example 4:

```
for (i=0;i<100;i=i+1) {
        A[i] = B[i] + C[i];
        D[i] = A[i] * E[i];
}
```

- Example 5:

```
for (i=1;i<100;i=i+1) {
        Y[i] = Y[i-1] + Y[i];
}
```

Finding dependencies

- Assume indices are affine:
  - $a \times i + b$ (i is loop index)

- Assume:
  - Store to $a \times i + b$, then
  - Load from $c \times i + d$
  - i runs from m to n
  - Dependence exists if:
    - Given j, k such that $m \le j \le n$, $m \le k \le n$
    - Store to $a \times j + b$, load from $a \times k + d$, and $a \times j + b = c \times k + d$

浙江大學 ZHEJIANG UNIVERSITY

Finding dependencies

- Generally cannot determine at compile time
- Test for absence of a dependence:
  - GCD test:
    - If a dependency exists, GCD(*c*,*a*) must evenly divide (*d*-*b*)
- Example:

```
for (i=0; i<100; i=i+1) {
   X[2*i+3] = X[2*i] * 5.0;
}
```

Finding dependencies

- Example 2:

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

- Watch for antidependencies and output dependencies

ZHEJIANG UNIVERSITY

Reductions

- Reduction Operation:

```
for (i=9999; i>=0; i=i-1)
        sum = sum + x[i] * y[i];
```

- Transform to…

```
for (i=9999; i>=0; i=i-1)
        sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
        finalsum = finalsum + sum[i];
```

- Do on p processors:

```
for (i=999; i>=0; i=i-1)
        finalsum[p] = finalsum[p] + sum[i+1000*p];
```

- Note: assumes associativity!

Fallacies and Pitfalls

- GPUs suffer from being coprocessors
  - GPUs have flexibility to change ISA
- Concentrating on peak performance in vector architectures and ignoring start-up overhead
  - Overheads require long vector lengths to achieve speedup
- Increasing vector performance without comparable increases in scalar performance
- You can get good vector performance without providing memory bandwidth
- On GPUs, just add more threads if you don’t have enough memory performance