DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Computer Architecture
Lecture 8: Vector Processing (Chapter 4)

Chih-Wei Liu 劉志尉

National Chiao Tung University

cwliu@twins.ee.nctu.edu.tw

Introduction

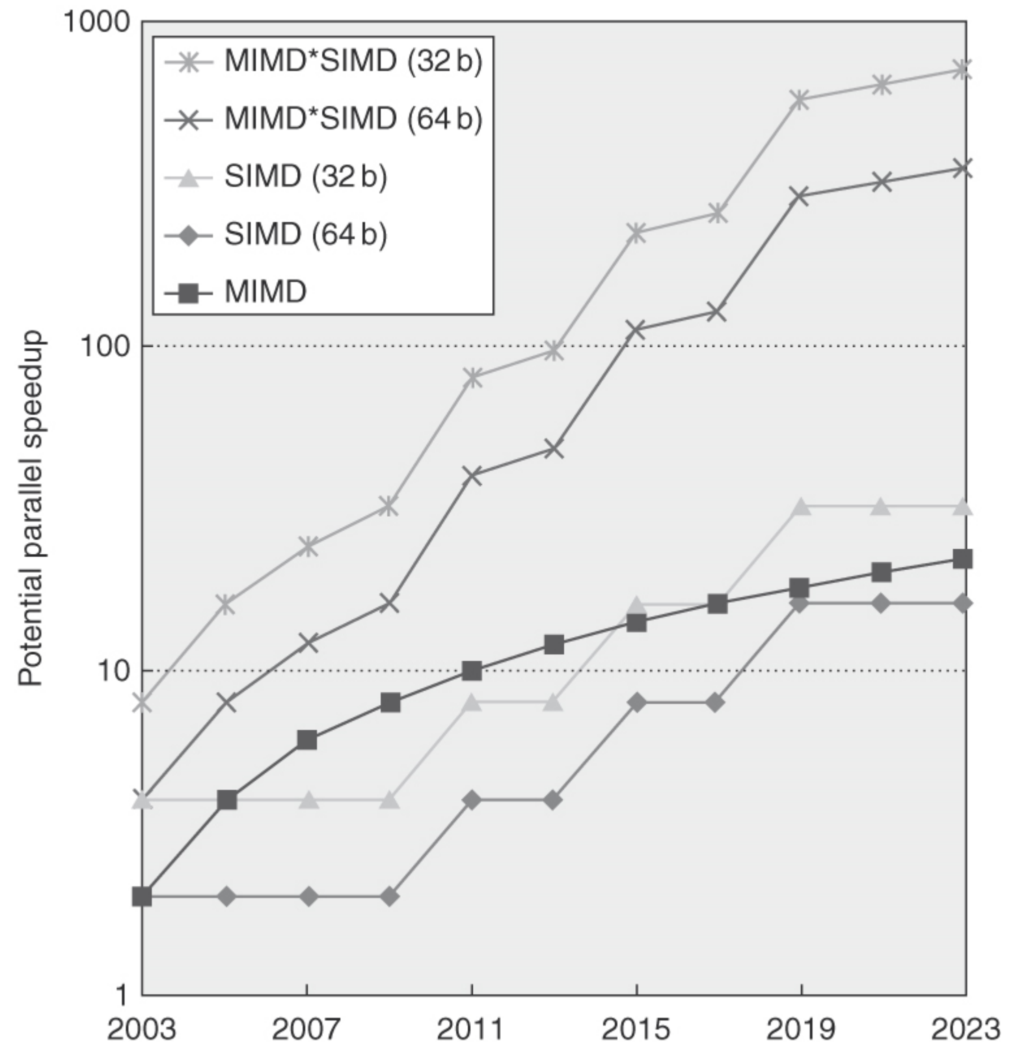
- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

SIMD Variations

- Vector architectures
- SIMD extensions
  - MMX: multimedia extensions (1996)
  - SSE: streaming SIMD extensions
  - AVX: advanced vector extensions
- Graphics Processor Units (GPUs)
  - Considered as SIMD accelerators

SIMD vs. MIMD

- For x86 processors:
  - Expect two additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!!



Vector 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
  - Register files act as compiler controlled buffers
  - Used to hide memory latency
  - Leverage memory bandwidth

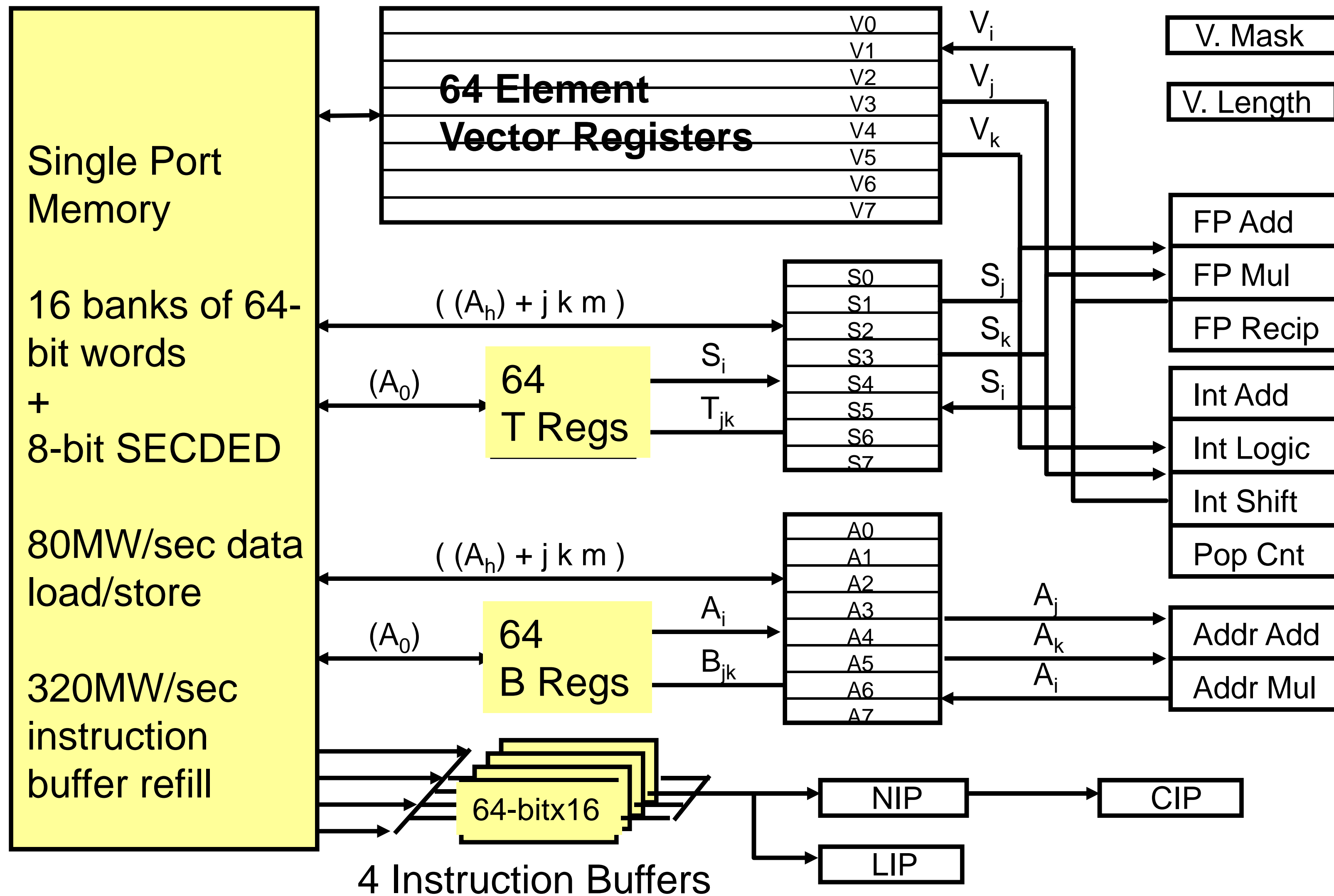
- Vector loads/stores deeply pipelined
  - Pay for memory latency once per vector load/store
- Regular loads/stores
  - Pay for memory latency for each vector element

Vector Supercomputers

- In 70-80s, Supercomputer ≡ Vector machine
- Definition of supercomputer
  - Fastest machine in the world at given task
  - A device to turn a compute-bound problem into an I/O-bound problem
  - CDC6600 (Cray, 1964) is regarded as the first supercomputer
- Vector supercomputers (epitomized by Cray-1, 1976)
  - Scalar unit + vector extensions
    - Vector registers, vector instructions
    - Vector loads/stores
    - Highly pipelined functional units

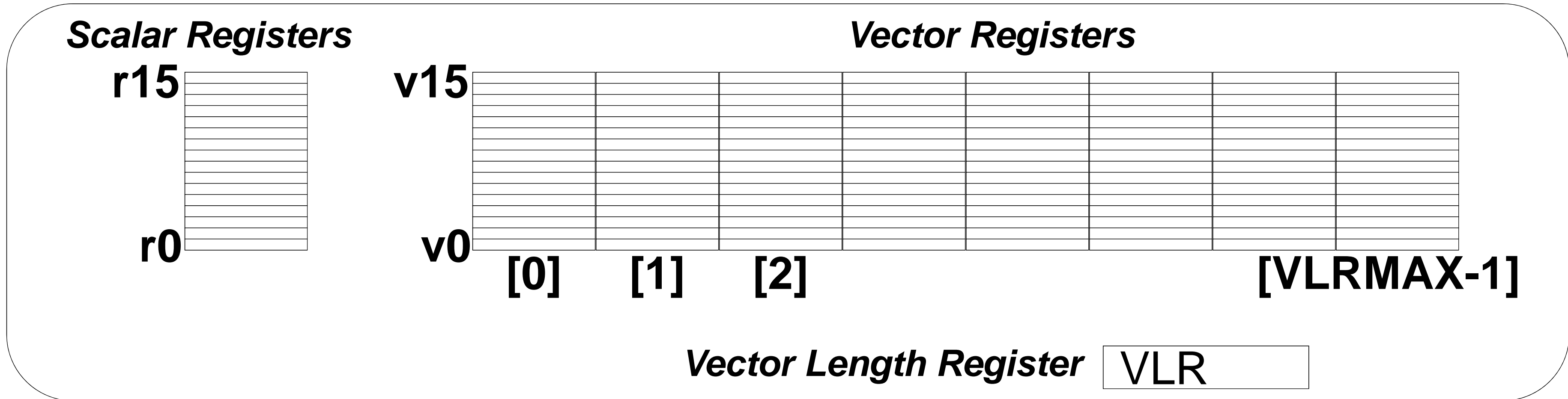
DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

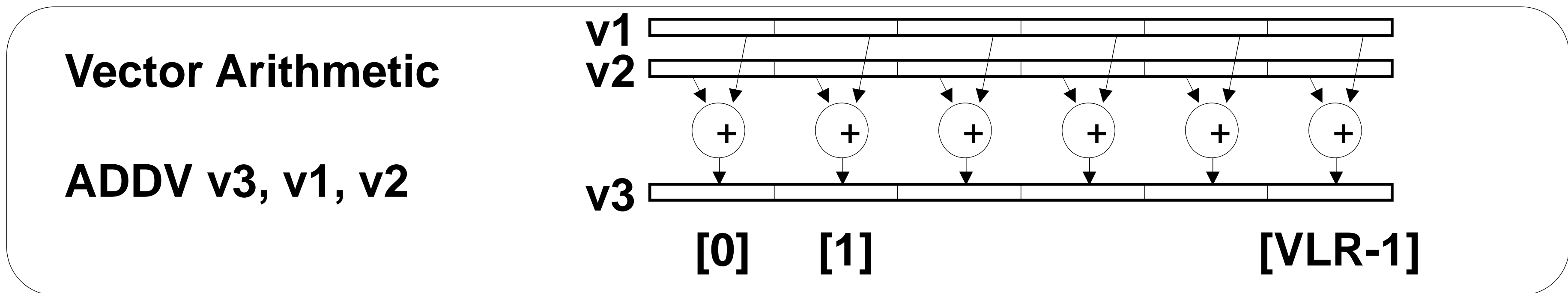
Cray-1 (1976)

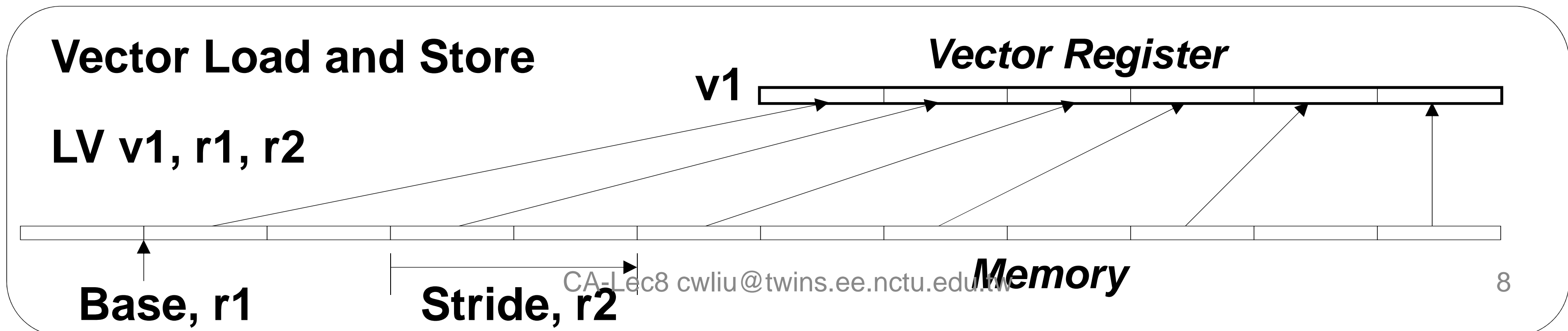


memory bank cycle 50 ns *processor cycle* 12.5 ns (80MHz)

Vector Programming Model

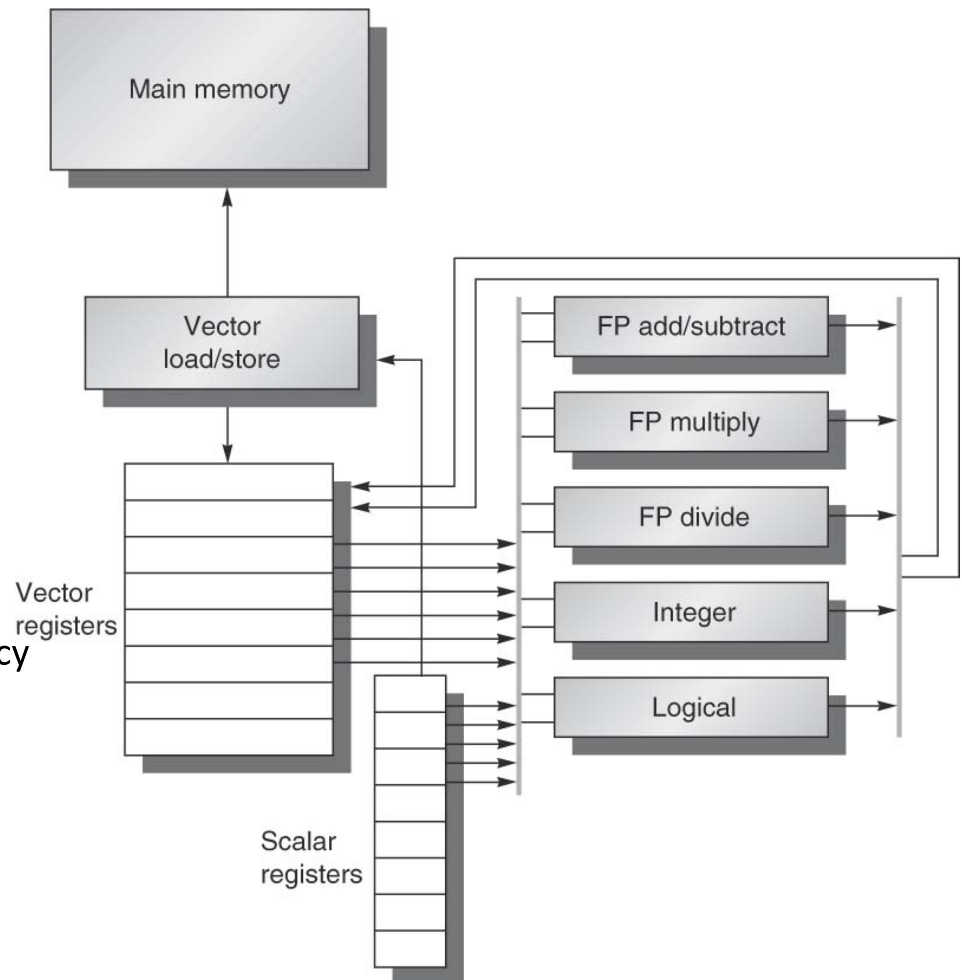






Example: VMIPS

- Loosely based on Cray-1
- Vector registers
  - Each register holds a 64-element, 64 bits/element vector
  - Register file has 16 read ports and 8 write ports
- Vector functional units
  - Fully pipelined
  - Data and control hazards are detected
- Vector load-store unit
  - Fully pipelined
  - Words move between registers
  - One word per clock cycle after initial latency
- Scalar registers
  - 32 general-purpose registers
  - 32 floating-point registers



VMIPS Instructions

- Operate on many elements concurrently
  - Allows use of slow but wide execution units
    - High performance, lower power
- Independence of elements within a vector instruction
  - Allows scaling of functional units without costly dependence checks
- Flexible
  - 64 64-bit / 128 32-bit / 256 16-bit, 512 8-bit
  - Matches the need of multimedia (8bit), scientific applications that require high precision

Vector Instructions

- ADDVV.D: add two vectors
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address
- Vector code example:

# C code	# Scalar Code	# Vector Code
for (i=0; i<64; i++) C[i] = A[i] + B[i];	LI R4, 64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop	LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
- The first vector machines, CDC Star-100 ('73) and TI ASC ('71), were memory-memory machines
- Cray-1 ('76) was first vector register machine

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 vs. Vector Register Machines

- 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

⇒ *Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures*

(we ignore vector memory-memory from now on)

Vector Instruction Set Advantages

- Compact
  - One short instruction encodes *N* operations
- Expressive
  - tells hardware that these *N* operations are independent
  - *N* operations use the same functional unit
  - *N* operations access disjoint registers
  - *N* operations access registers in the same pattern as previous instruction
  - *N* operations access a contiguous block of memory (unit-stride load/store)
  - *N* operations access memory in a known pattern (stridden load/store)
- Scalable
  - Can run same object code on more parallel pipelines or lanes

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Vector Instructions Example

- Example: DAXPY adds a scalar multiple of a double precision vector to another double precision vector

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

Requires 6 instructions only

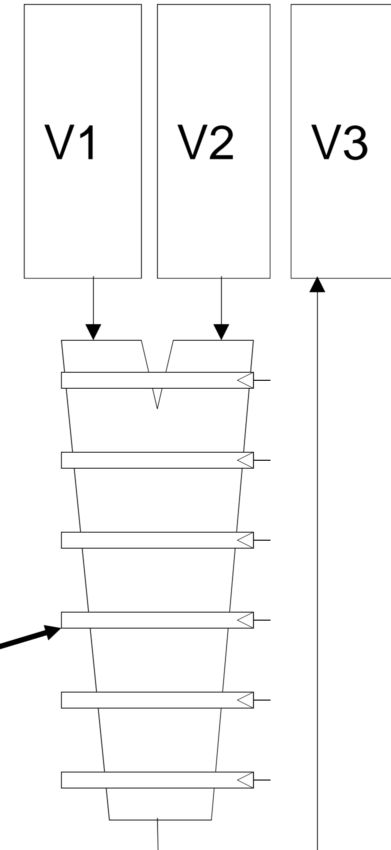
- In MIPS Code
  - ADD waits for MUL, SD waits for ADD
- In VMIPS
  - Stall once for the first vector element, subsequent elements will flow smoothly down the pipeline.
  - Pipeline stall required once per vector instruction!

Challenges of Vector Instructions

- Start up time
  - Application and architecture must support long vectors. Otherwise, they will run out of instructions requiring ILP
  - 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

Vector Arithmetic Execution

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



Six stage multiply pipeline

V3 <- v1 * v2

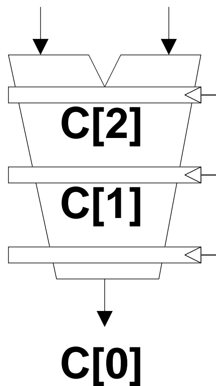
Vector Instruction Execution

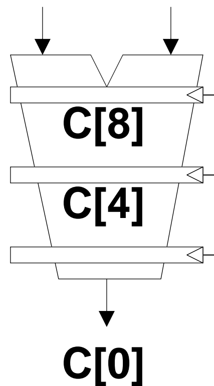
ADDV C,A,B

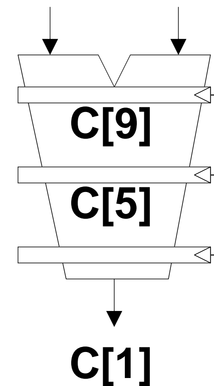
Execution using one pipelined functional unit

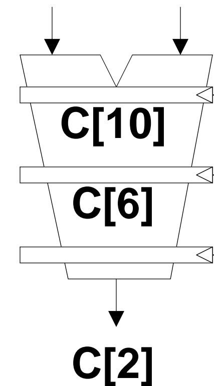
Four-lane execution using four pipelined functional units

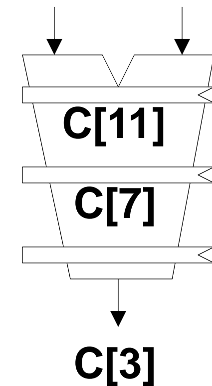
A[6]	B[6]		A[24]	B[24]	A[25]	B[25]	A[26]	B[26]	A[27]	B[27]
A[5]	B[5]		A[20]	B[20]	A[21]	B[21]	A[22]	B[22]	A[23]	B[23]
A[4]	B[4]		A[16]	B[16]	A[17]	B[17]	A[18]	B[18]	A[19]	B[19]
A[3]	B[3]		A[12]	B[12]	A[13]	B[13]	A[14]	B[14]	A[15]	B[15]





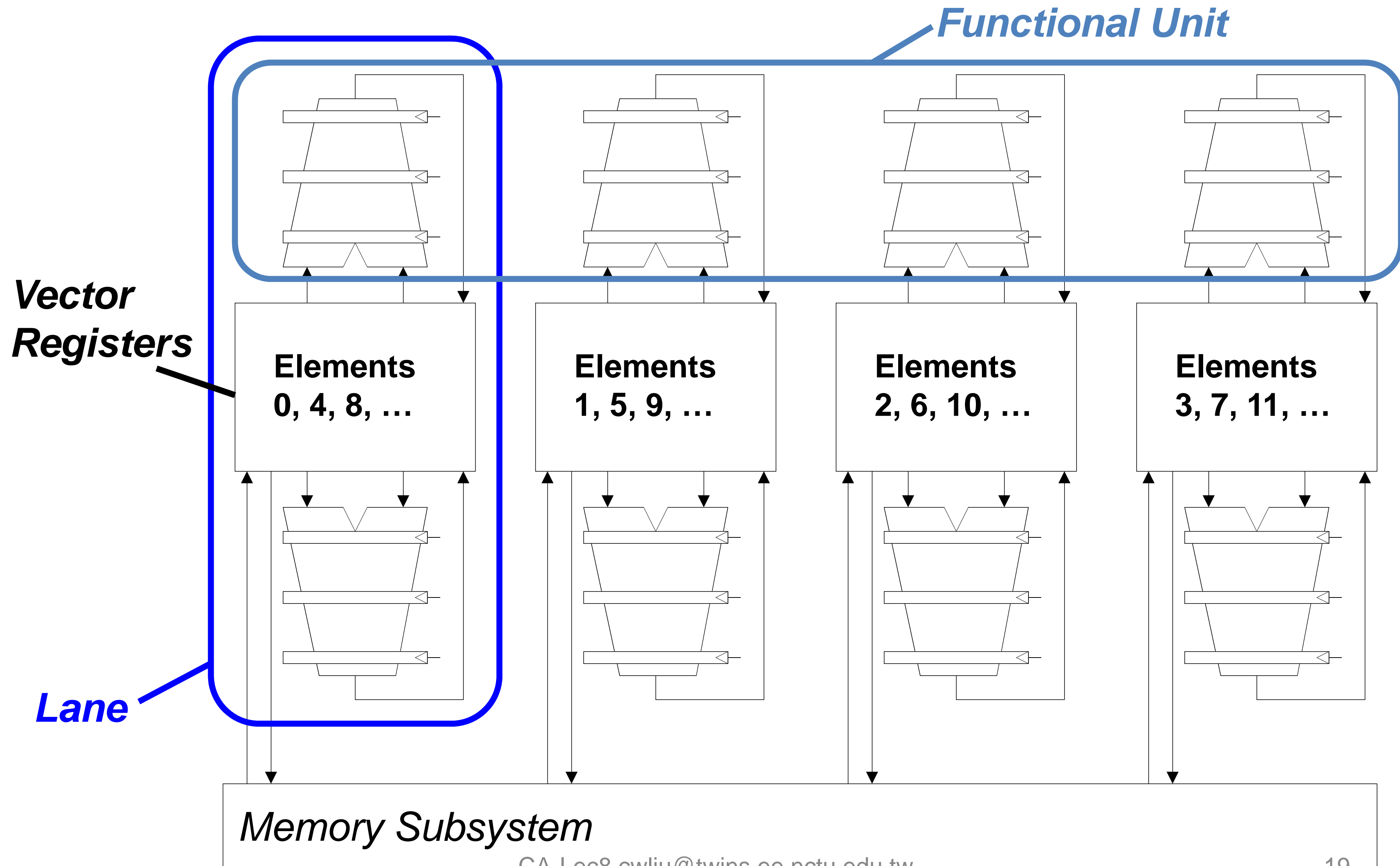






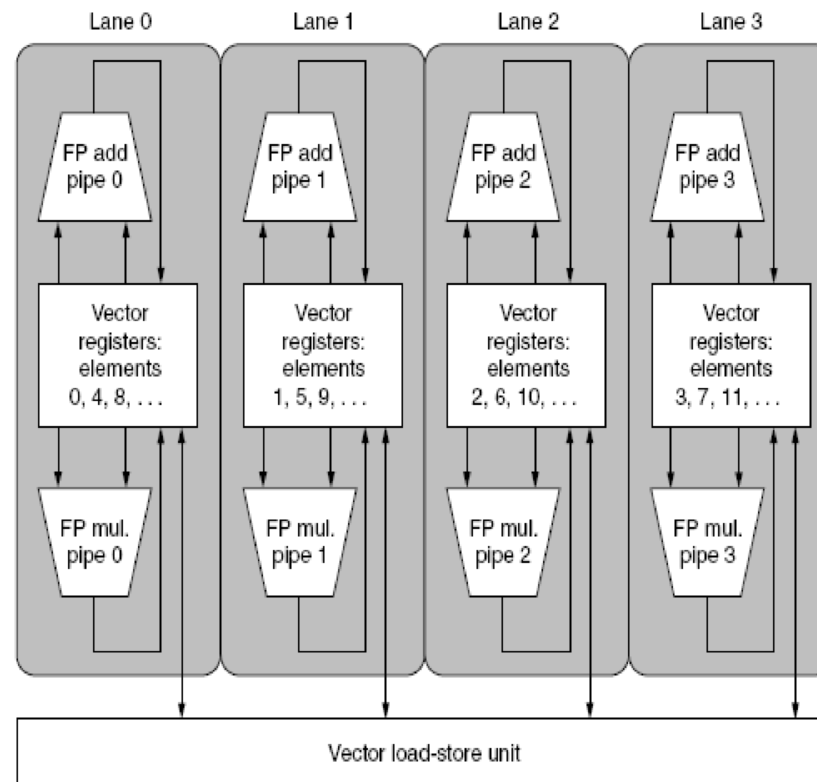
C[0] C[0] C[1] C[2] C[3]

Vector Unit Structure



Multiple Lanes Architecture

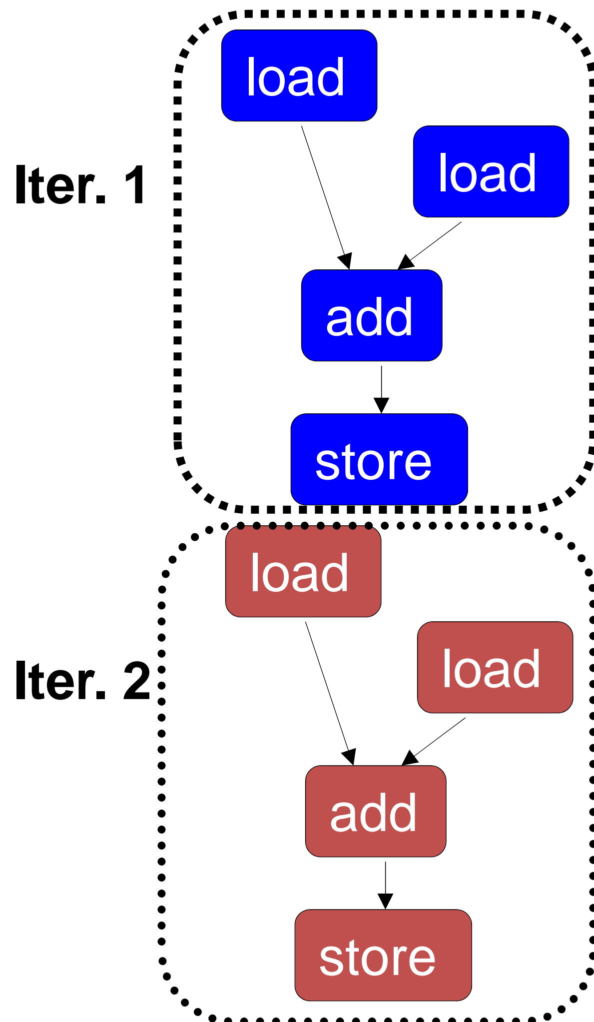
- Beyond one element per clock cycle
- Elements n of vector register A is hardwired to element n of vector B
  - Allows for multiple hardware lanes
  - No communication between lanes
  - Little increase in control overhead
  - No need to change machine code

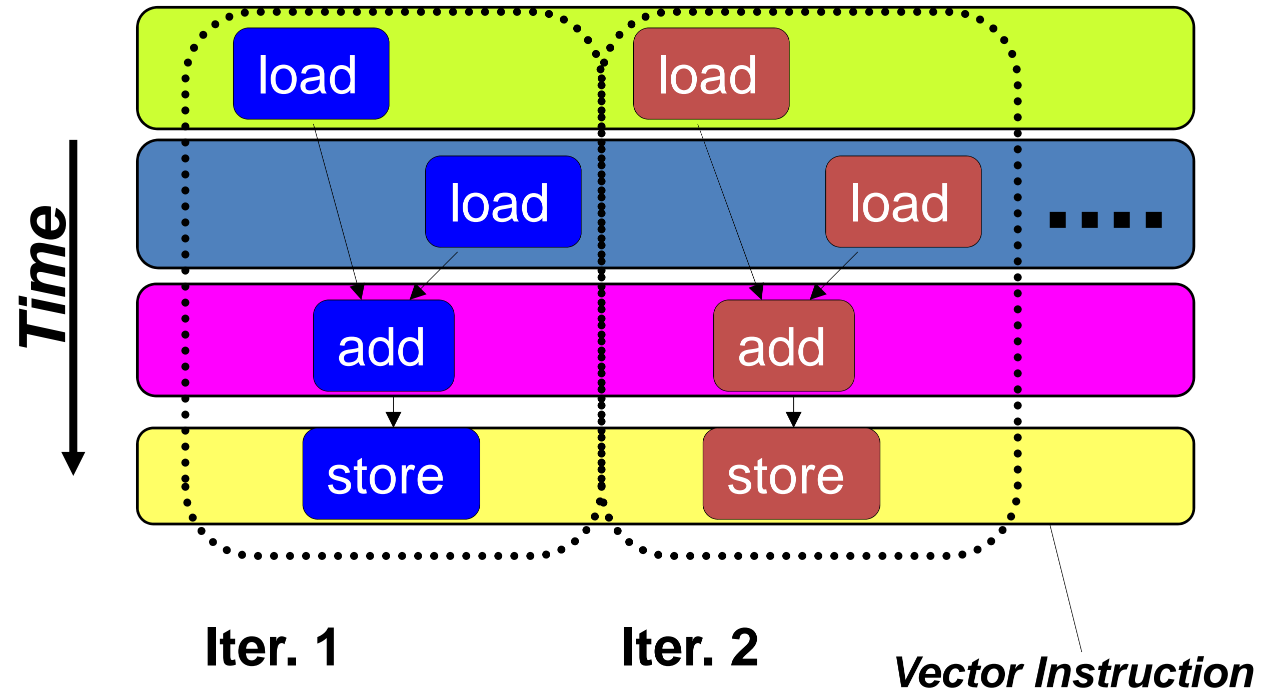


Adding more lanes allows designers to tradeoff clock rate and energy without sacrificing performance!

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

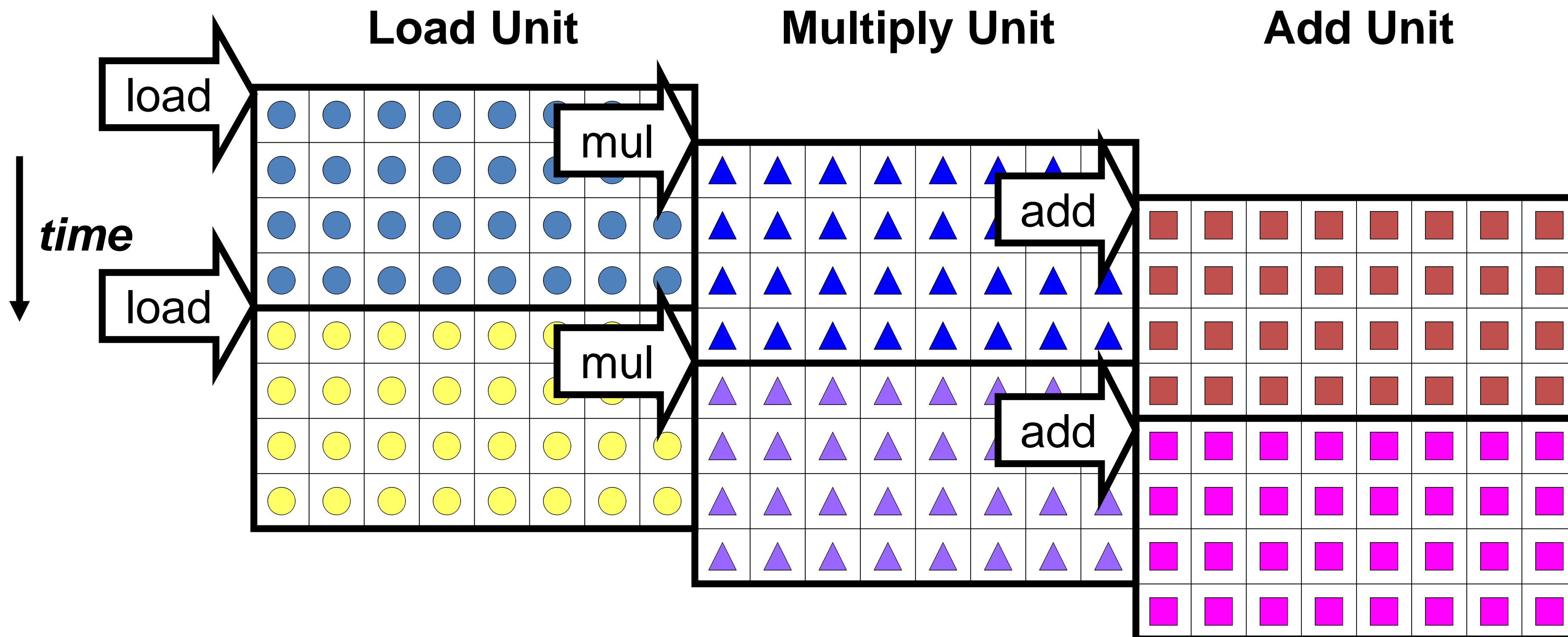
Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- VMIPS functional units consume one element per clock cycle
  - Execution time is approximately the vector length
- *Convoy*
  - Set of vector instructions that could potentially execute together

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



Convoy

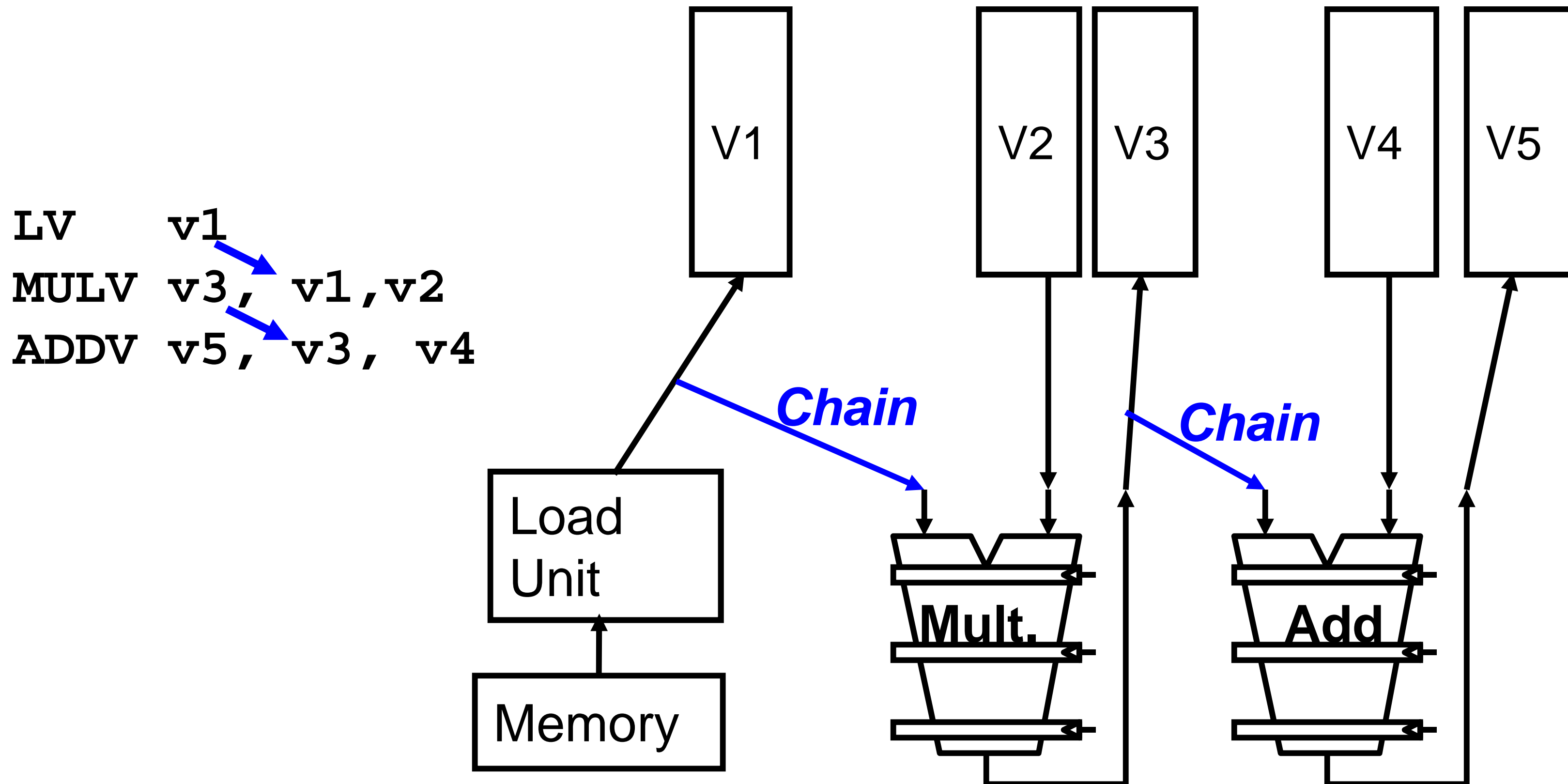
- Convoy: set of vector instructions that could potentially execute together
  - Must not contain structural hazards
  - Sequences with RAW hazards should be in different convoys

- However, sequences with RAW hazards can be in the same convey via chaining

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Vector Chaining

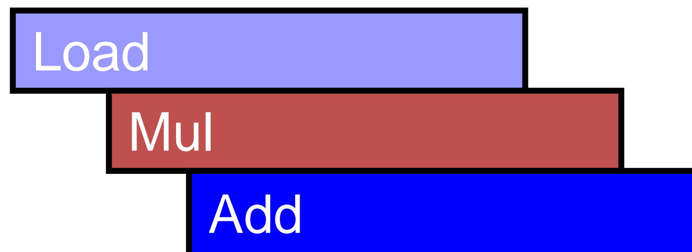
- Vector version of register bypassing
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available



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



Chimes

- Chimes: unit of time to execute one convoy
  - m convoy executes in m chimes
  - For vector length of n, requires $m \times n$ clock cycles

Example

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

Convoys:

1	LV	MULVS.D
2	LV	ADDVV.D
3	SV	

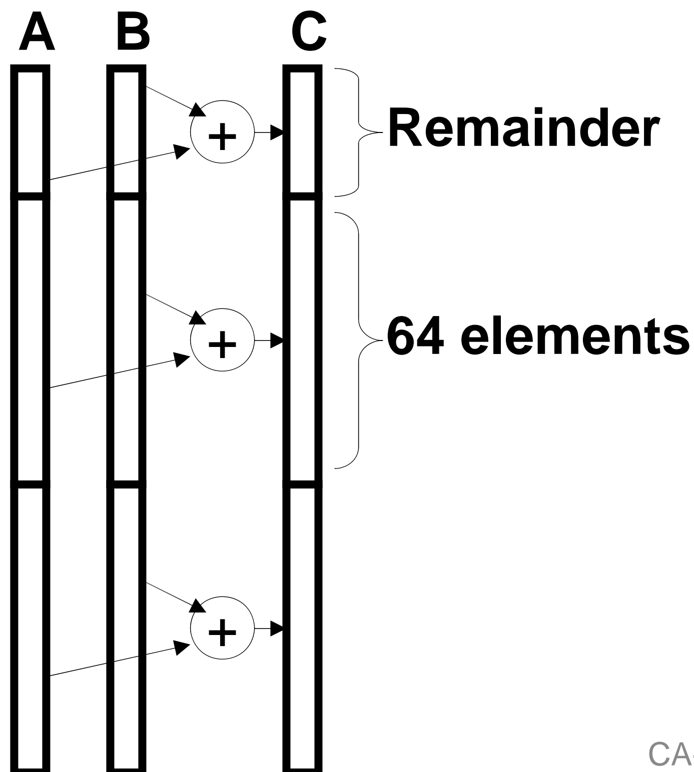
3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

For 64 element vectors, requires 64 x 3 = 192 clock cycles

Vector Strip-mining

Problem: Vector registers have finite length
Solution: Break loops into pieces that fit into vector registers, *"Strip-mining"*

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

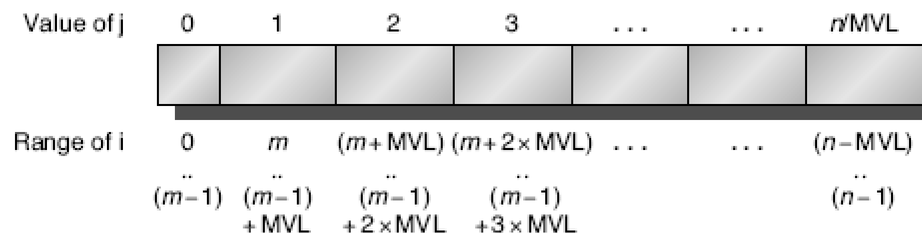



```
 ANDI R1, N, 63     # N mod 64
 MTC1 VLR, R1       # Do remainder
loop:
 LV V1, RA
 DSLL R2, R1, 3     # Multiply by 8
 DADDU RA, RA, R2   # Bump pointer
 LV V2, RB
 DADDU RB, RB, R2
 ADDV.D V3, V1, V2
 SV V3, RC
 DADDU RC, RC, R2
 DSUBU N, N, R1 # Subtract elements
 LI R1, 64
 MTC1 VLR, R1    #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

Vector Length Register

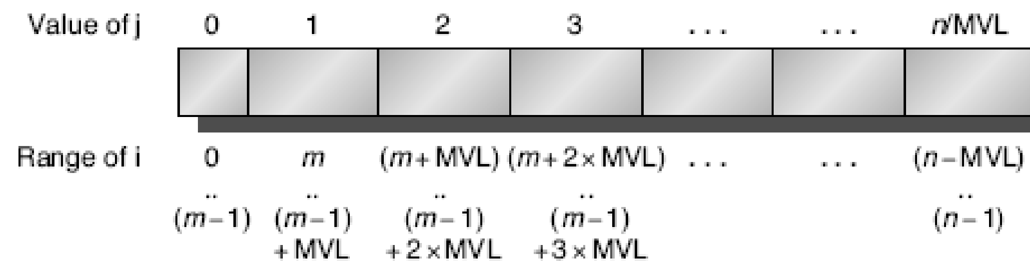
- Handling loops not equal to 64
- Vector length not known at compile time? Use Vector Length Register (VLR) for vectors over the maximum length, strip mining:

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
     for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
        Y[i] = a * X[i] + Y[i] ; /*main operation*/
     low = low + VL; /*start of next vector*/
     VL = MVL; /*reset the length to maximum vector length*/
}
```



Maximum Vector Length (MVL)

- Determine the maximum number of elements in a vector for a given architecture

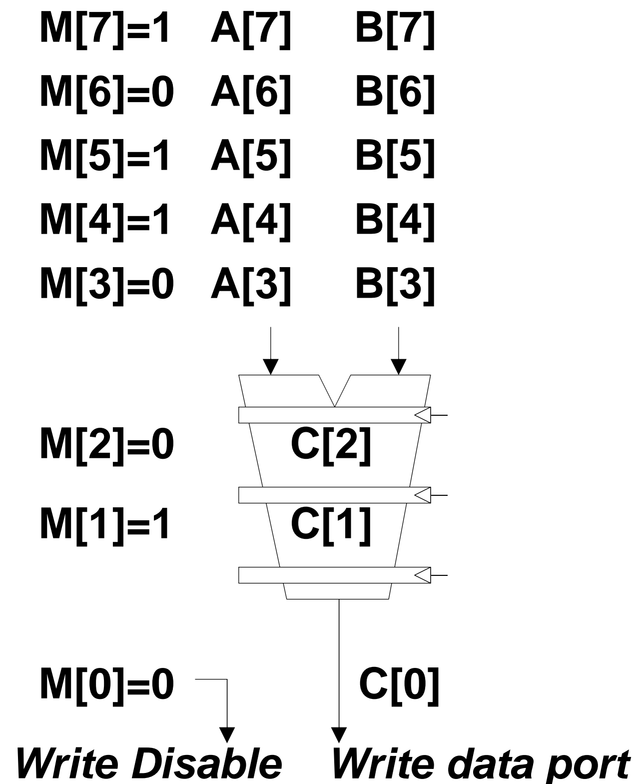


- All blocks but the first are of length MVL
  - Utilize the full power of the vector processor
- Later generations may grow the MVL
  - No need to change the ISA

Vector-Mask Control

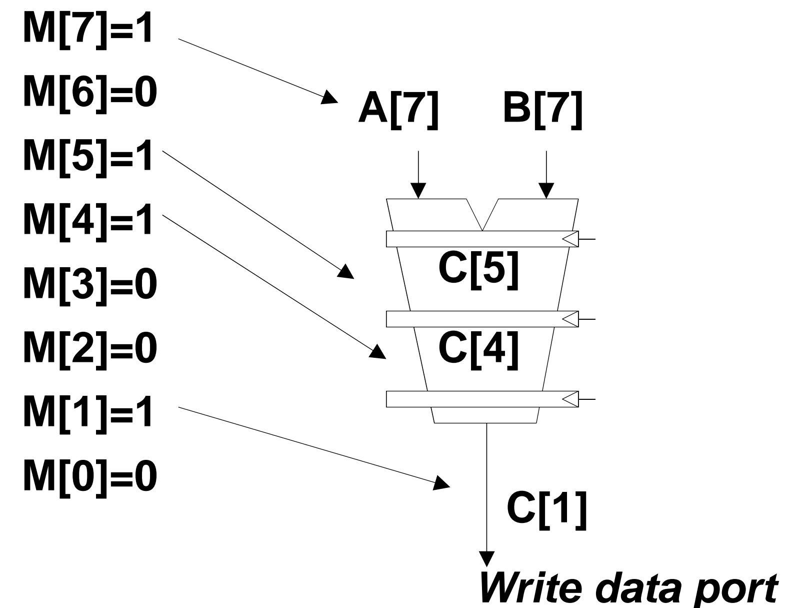
Simple Implementation

– 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



Vector Mask Register (VMR)

- A Boolean vector to control the execution of a vector instruction
- VMR is part of the architectural state
- For vector processor, it relies on compilers to manipulate VMR explicitly
- For GPU, it gets the same effect using hardware
  - Invisible to SW
- Both GPU and vector processor spend time on masking

Vector Mask Registers Example

- Programs that contain IF statements in loops cannot be run in vector processor
- Consider:

```
for (i = 0; i < 64; i=i+1)
        if (X[i] != 0)
                X[i] = X[i] – Y[i];
```

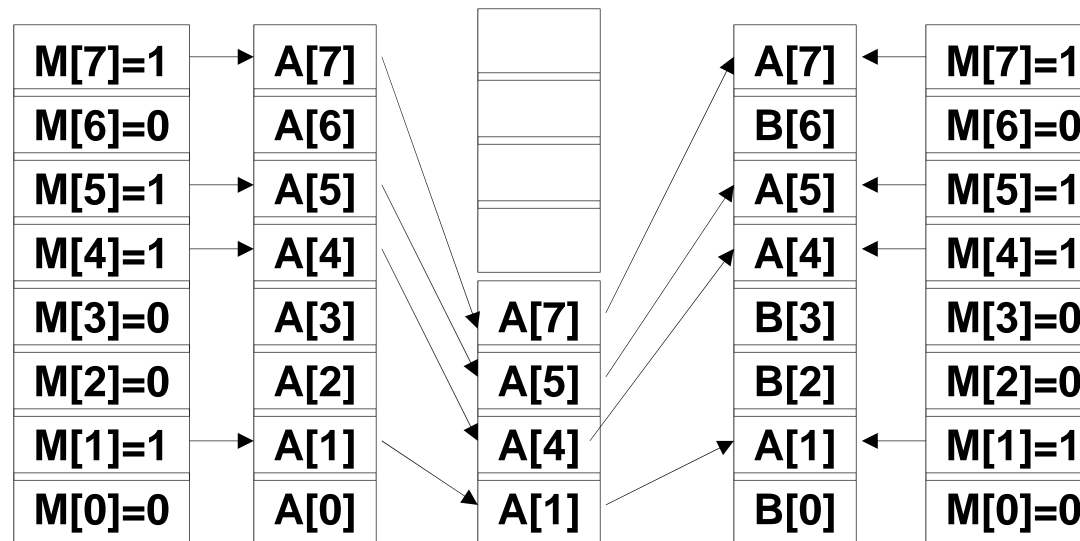
- Use vector mask register to “disable” elements:

```
LV        V1,Rx       ;load vector X into V1
LV        V2,Ry       ;load vector Y
L.D       F0,#0       ;load FP zero into F0
SNEVS.D   V1,F0       ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D   V1,V1,V2    ;subtract under vector mask
SV        Rx,V1       ;store the result in X
```

- GFLOPS rate decreases!

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



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

Memory Banks

- The start-up time for a load/store vector unit is the time to get the first word from memory into a register. How about the rest of the vector?
  - Memory stalls can reduce effective throughput for the rest of the vector
- Penalties for start-ups on load/store units are higher than those for arithmetic unit
- Memory system must be designed to support high bandwidth for vector loads and stores
  - Spread accesses across multiple banks

  1. Support multiple loads or stores per clock. Be able to control the addresses to the banks independently
  2. Support (multiple) non-sequential loads or stores
  3. Support multiple processors sharing the same memory system, so each processor will be generating its own independent stream of addresses

Example (Cray T90)

- 32 processors, each generating 4 loads and 2 stores per cycle
- Processor cycle time is 2.167ns
- SRAM cycle time is 15ns
- How many memory banks needed?

- Solution:

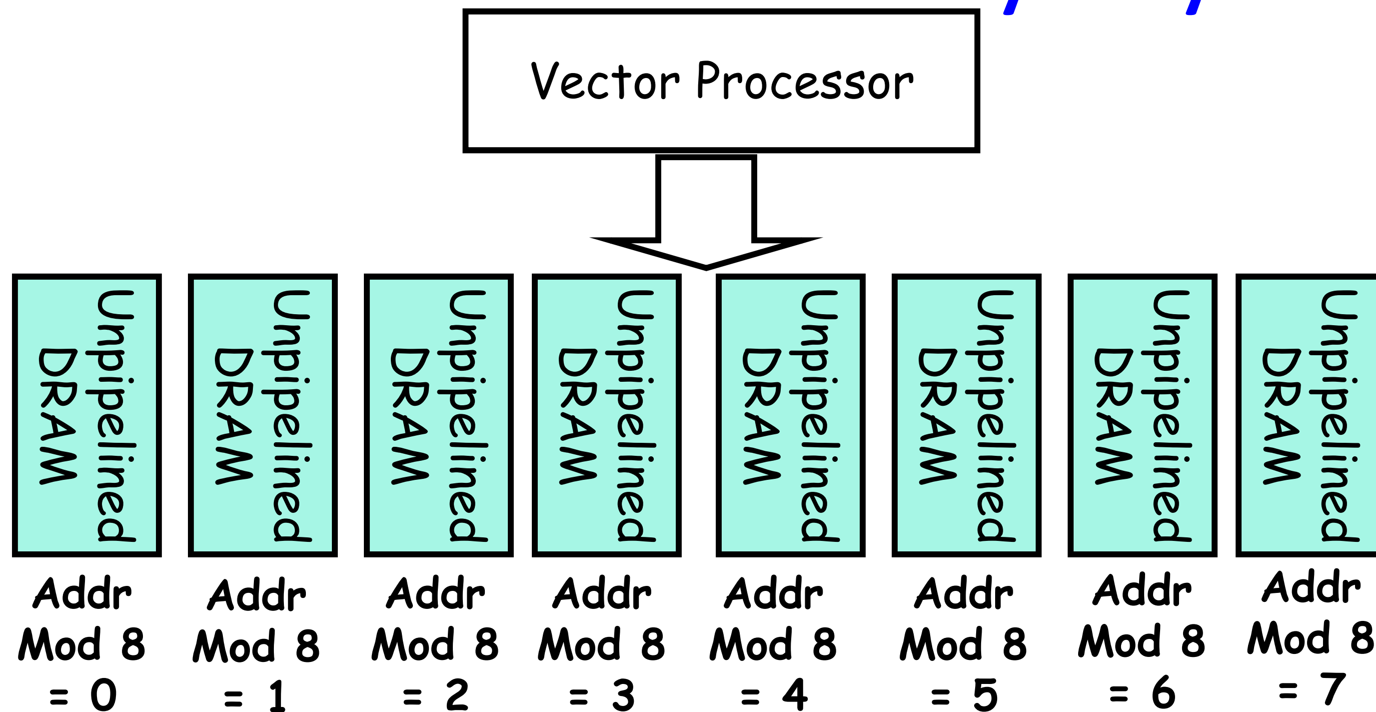
1. The maximum number of memory references each cycle is $32\times6=192$
2. SRAM takes $15/2.167=6.92\approx7$ processor cycles
3. It requires $192\times7=1344$ memory banks at full memory bandwidth!!

Cray T932 actually has 1024 memory banks (not sustain full bandwidth)

Memory Addressing

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - Contiguous block of information in memory
    - Fastest: always possible to optimize this
  - Non-unit (constant) stride
    - Harder to optimize memory system for all possible strides
    - Prime number of data banks makes it easier to support different strides at full bandwidth
  - Indexed (gather-scatter)
    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize

Interleaved Memory Layout



- Great for unit stride:
  - Contiguous elements in different DRAMs
  - Startup time for vector operation is latency of single read
- What about non-unit stride?
  - Above good for strides that are relatively prime to 8
  - Bad for: 2, 4

Handling Multidimensional Arrays in Vector Architectures

- Consider:

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
        A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

- Must vectorize multiplications of rows of B with columns of D
  - Need to access adjacent elements of B and adjacent elements of D
  - Elements of B accessed in row-major order but elements of D in column-major order!
- Once vector is loaded into the register, it acts as if it has logically adjacent elements

(Unit/Non-Unit) Stride Addressing

- The distance separating elements to be gathered into a single register is called stride.
- In the example
  - Matrix D has a stride of 100 double words
  - Matrix B has a stride of 1 double word
- Use non-unit stride for D
  - To access non-sequential memory location and to reshape them into a dense structure
- The size of the matrix may not be known at compile time
  - Use LVWS/SVWS: load/store vector with stride
    - The vector stride, like the vector starting address, can be put in a general-purpose register (dynamic)
- Cache inherently deals with unit stride data
  - Blocking techniques helps non-unit stride data

How to get full bandwidth for unit stride?

- Memory system must sustain (# lanes x word) /clock
- # memory banks > memory latency to avoid stalls
- If desired throughput greater than one word per cycle
  - Either more banks (start multiple requests simultaneously)
  - Or wider DRAMS. Only good for unit stride or large data types
- # memory banks > memory latency to avoid stalls
- More numbers of banks good to support more strides at full bandwidth
  - can read paper on how to do prime number of banks efficiently

Memory Bank Conflicts

```
int x[256][512];
        for (j = 0; j < 512; j = j+1)
                for (i = 0; i < 256; i = i+1)
                        x[i][j] = 2 * x[i][j];
```

- Even with 128 banks, since 512 is multiple of 128, conflict on word accesses
- SW: loop interchange or declaring array not power of 2 (“array padding”)
- HW: Prime number of banks
  - bank number = address mod number of banks
  - address within bank = address / number of words in bank
  - modulo & divide per memory access with prime no. banks?
  - address within bank = address mod number words in bank
  - bank number? easy if 2^N words per bank

Problems of Stride Addressing

- Once we introduce non-unit strides, it becomes possible to request accesses from the same bank frequently
  - Memory bank conflict !!
    - Stall the other request to solve bank conflict
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM(stride,#banks) < bank busy time

Example

- 8 memory banks, Bank busy time: 6 cycles, Total memory latency: 12 cycles for initialized
- What is the difference between a 64-element vector load with a stride of 1 and 32?

- Solution:

1. Since 8 > 6, for a stride of 1, the load will take 12+64=76 cycles, i.e. 1.2 cycles per element
2. Since 32 is a multiple of 8, the worst possible stride
   - every access to memory (after the first one) will collide with the previous access and will have to wait for 6 cycles
   - The total time will take 12+1+63×6=391 cycles, i.e. 6.1 cycles per element

Handling Sparse Matrices in Vector architecture

- Sparse matrices in vector mode is a necessity
- Example: Consider a sparse vector sum on arrays A and C

```
for (i = 0; i < n; i=i+1)
        A[K[i]] = A[K[i]] + C[M[i]];
```

  - where K and M and index vectors to designate the nonzero elements of A and C

- Sparse matrix elements stored in a compact form and accessed indirectly
- Gather-scatter operations are used
  - Using LVI/SVI: load/store vector indexed or gathered

Scatter-Gather

- Consider:
  for (i = 0; i < n; i=i+1)
  A[K[i]] = A[K[i]] + C[M[i]];
- Ra, Rc, Rk, and Rm contain the starting addresses of the vectors

- Use index vector:

```
LV        Vk, Rk          ;load K
LVI       Va, (Ra+Vk)     ;load A[K[]]
LV        Vm, Rm          ;load M
LVI       Vc, (Rc+Vm)     ;load C[M[]]
ADDVV.D   Va, Va, Vc      ;add them
SVI       (Ra+Vk), Va     ;store A[K[]]
```

- A and C must have the same number of non-zero elements (sizes of k and m)

Vector Architecture Summary

- Vector is alternative model for exploiting ILP
  - If code is vectorizable, then simpler hardware, energy efficient, and better real-time model than out-of-order
    - More lanes, slower clock rate!
  - Scalable if elements are independent
  - If there is dependency
    - One stall per vector instruction rather than one stall per vector element
- Programmer in charge of giving hints to the compiler!
- Design issues: number of lanes, functional units and registers, length of vector registers, exception handling, conditional operations
- Fundamental design issue is memory bandwidth
  - Especially with virtual address translation and caching

Programming Vector Architectures

- Compilers can provide feedback to programmers
- Programmers can provide hints to compiler
- Cray Y-MP Benchmarks

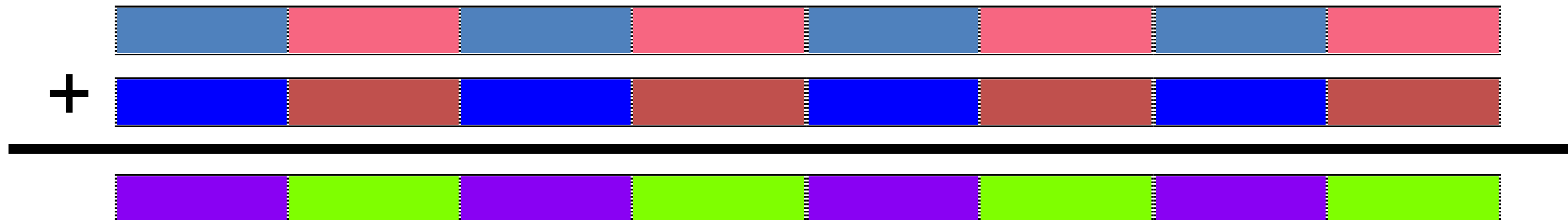
Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	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

Multimedia Extensions

- Very short vectors added to existing ISAs
- Usually 64-bit registers split into 2x32b or 4x16b or 8x8b
- Newer designs have 128-bit registers (Altivec, SSE2)
- Limited instruction set:
  - no vector length control
  - no load/store stride 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

"Vector" for Multimedia?

- Intel MMX: 57 additional 80x86 instructions (1st since 386)
  - similar to Intel 860, Mot. 88110, HP PA-71000LC, UltraSPARC
- 3 data types: 8 8-bit, 4 16-bit, 2 32-bit in 64bits
  - reuse 8 FP registers (FP and MMX cannot mix)
- short vector: load, add, store 8 8-bit operands



- Claim: overall speedup 1.5 to 2X for 2D/3D graphics, audio, video, speech, comm., ...
  - use in drivers or added to library routines; no compiler

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

MMX Instructions

- Move 32b, 64b
- Add, Subtract in parallel: 8 8b, 4 16b, 2 32b
  - opt. signed/unsigned saturate (set to max) if overflow
- Shifts (sll,srl, sra), And, And Not, Or, Xor in parallel: 8 8b, 4 16b, 2 32b
- Multiply, Multiply-Add in parallel: 4 16b
- Compare = , > in parallel: 8 8b, 4 16b, 2 32b
  - sets field to 0s (false) or 1s (true); removes branches
- Pack/Unpack
  - Convert 32b<–> 16b, 16b <–> 8b
  - Pack saturates (set to max) if number is too large

SIMD Implementations: IA32/AMD64

- Intel MMX (1996)
  - Repurpose 64-bit floating point registers
  - Eight 8-bit integer ops or four 16-bit integer ops
- Streaming SIMD Extensions (SSE) (1999)
  - Separate 128-bit registers
  - Eight 16-bit integer ops, Four 32-bit integer/fp ops, or two 64-bit integer/fp ops
  - Single-precision floating-point arithmetic
- SSE2 (2001), SSE3 (2004), SSE4(2007)
  - Double-precision floating-point arithmetic
- Advanced Vector Extensions (2010)
  - 256-bits registers
  - Four 64-bit integer/fp ops
  - Extensible to 512 and 1024 bits for future generations

SIMD Implementations: IBM

- VMX (1996-1998)
  - 32 4b, 16 8b, 8 16b, 4 32b integer ops and 4 32b FP ops
  - Data rearrangement
- Cell SPE (PS3)
  - 16 8b, 8 16b, 4 32b integer ops, and 4 32b and 8 64b FP ops
  - Unified vector/scalar execution with 128 registers
- VMX 128 (Xbox360)
  - Extension to 128 registers
- VSX (2009)
  - 1 or 2 64b FPU, 4 32b FPU
  - Integrate FPU and VMX into unit with 64 registers
- QPX (2010, Blue Gene)
  - Four 64b SP or DP FP

Why SIMD Extensions?

- Media applications operate on data types narrower than the native word size
- Costs little to add to the standard arithmetic unit
- Easy to implement
- Need smaller memory bandwidth than vector
- Separate data transfer aligned in memory
  - Vector: single instruction, 64 memory accesses, page fault in the middle of the vector likely !!
- Use much smaller register space
- Fewer operands
- No need for sophisticated mechanisms of vector architecture

Example SIMD Code

- Example DXPY:

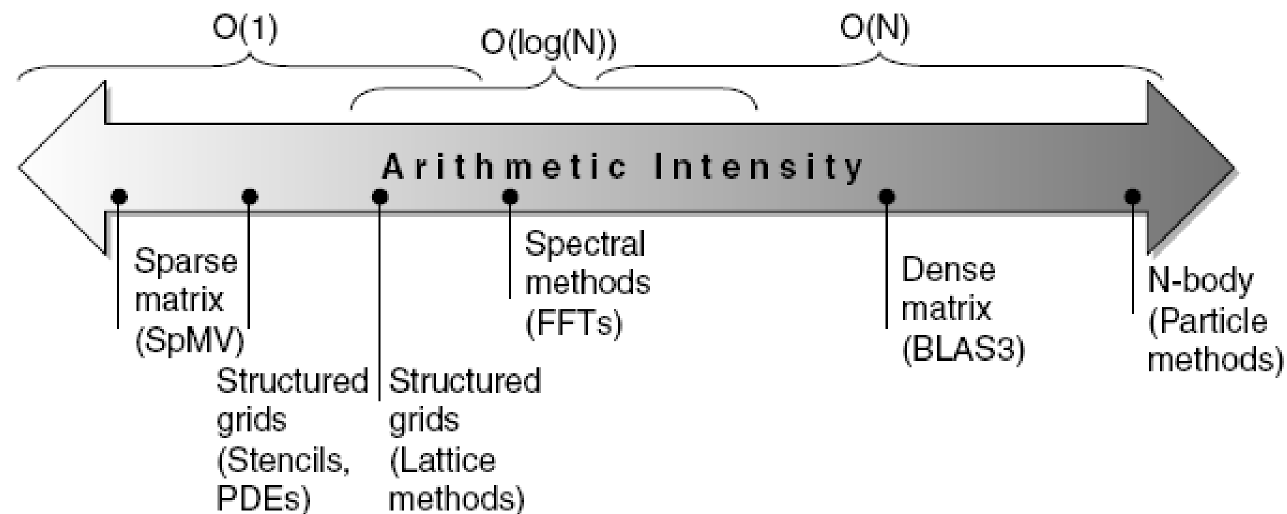
```
        L.D       F0,a          ;load scalar a
        MOV       F1, F0        ;copy a into F1 for SIMD MUL
        MOV       F2, F0        ;copy a into F2 for SIMD MUL
        MOV       F3, F0        ;copy a into F3 for SIMD MUL
        DADDIU    R4,Rx,#512    ;last address to load
Loop:   L.4D      F4,0[Rx]      ;load X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D    F4,F4,F0      ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D      F8,0[Ry]      ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D    F8,F8,F4      ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D      0[Ry],F8      ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU    Rx,Rx,#32     ;increment index to X
        DADDIU    Ry,Ry,#32     ;increment index to Y
        DSUBU     R20,R4,Rx     ;compute bound
        BNEZ      R20,Loop      ;check if done
```

Challenges of SIMD Architectures

- Scalar processor memory architecture
  - Only access to contiguous data
    - No efficient scatter/gather accesses
  - Significant penalty for unaligned memory access
  - May need to write entire vector register
- Limitations on data access patterns
  - Limited by cache line, up to 128-256b
- Conditional execution
  - Register renaming does not work well with masked execution
  - Always need to write whole register
  - Difficult to know when to indicate exceptions
- Register pressure
  - Need to use multiple registers rather than register depth to hide latency

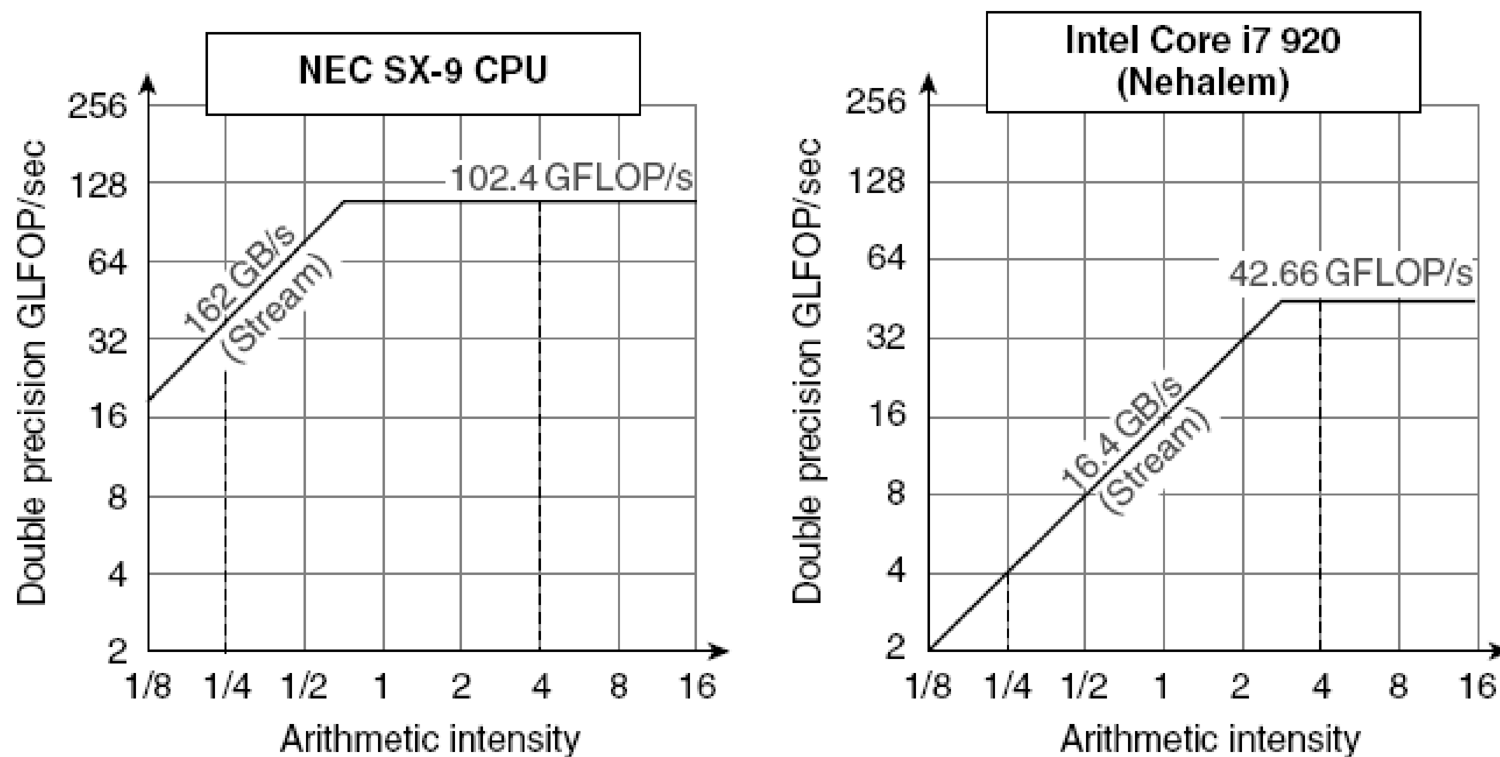
Roofline Performance Model

- Basic idea:
  - Plot peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
  - Floating-point operations per byte read



Examples

- Attainable GFLOPs/sec



History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output
- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - 3D graphics cards for PCs and game consoles
- Graphics Processing Units
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, ray tracing

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"

Programming Model

- CUDA's design goals
  - extend a standard sequential programming language, specifically C/C++,
    - focus on the important issues of parallelism—how to craft efficient parallel algorithms—rather than grappling with the mechanics of an unfamiliar and complicated language.
  - minimalist set of abstractions for expressing parallelism
    - highly scalable parallel code that can run across tens of thousands of concurrent threads and hundreds of processor cores.

NVIDIA GPU Architecture

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

Programming the GPU

- CUDA Programming Model
  - Single Instruction Multiple Thread (SIMT)
- 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
  - Given the hardware invested to do graphics well, how can we supplement it to improve performance of a wider range of applications?

Example

- Multiply two vectors of length 8192
  - 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 (Fermi) have 7-15 multithreaded SIMD processors

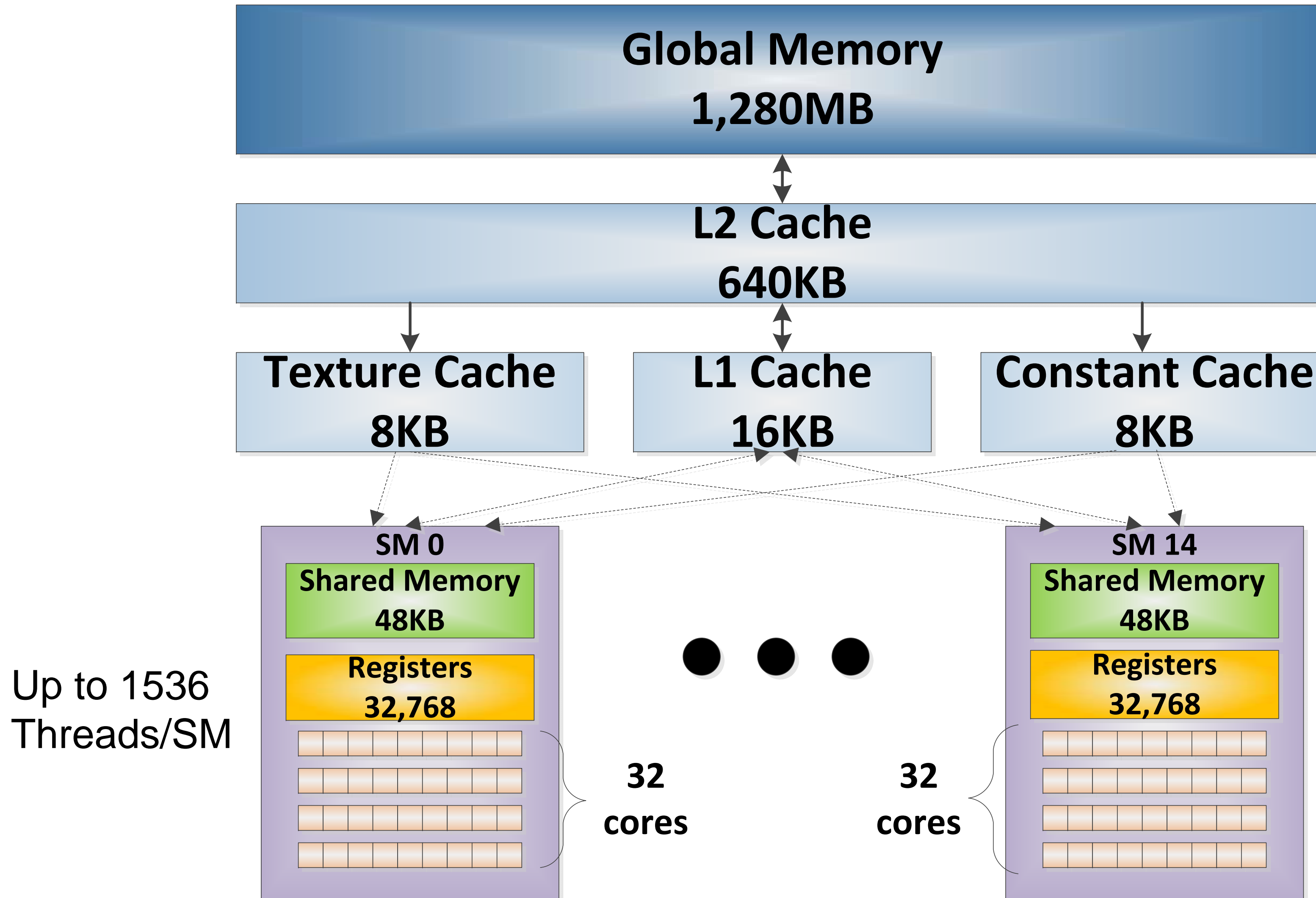
Terminology

- *Threads of SIMD instructions*
  - Each has its own PC
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Keeps track of up to 48 threads of SIMD instructions
    - Hides memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - 32 SIMD lanes
  - Wide and shallow compared to vector processors

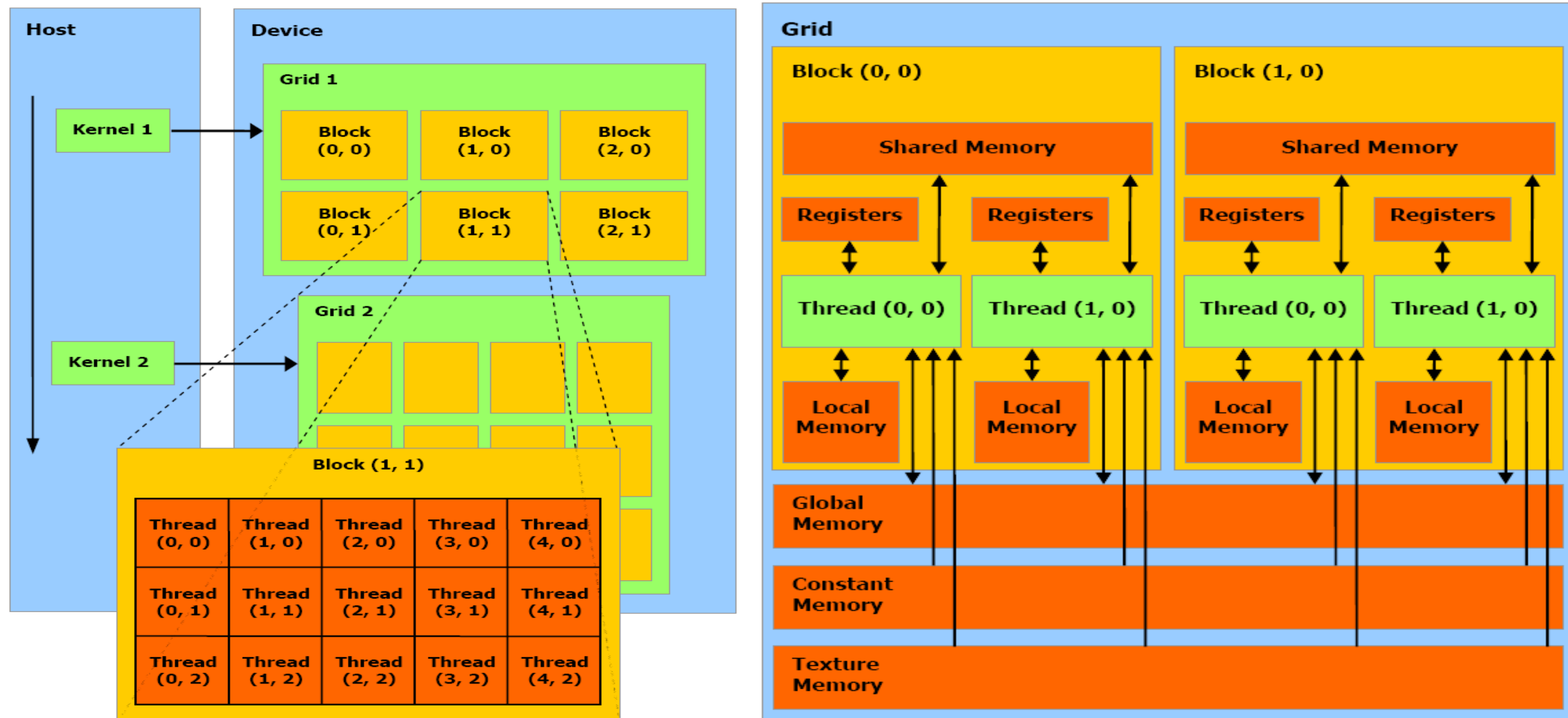
Example

- NVIDIA GPU has 32,768 registers
  - Divided into lanes
  - Each SIMD thread is limited to 64 registers
  - SIMD thread has up to:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
  - Fermi has 16 physical SIMD lanes, each containing 2048 registers

GTX570 GPU



GPU Threads in SM (GTX570)



- 32 threads within a block work collectively
  - ✓ Memory access optimization, latency hiding

Matrix Multiplication

Matrix C		
C_0	C_1	C_2
C_3	C_4	C_5
C_6	C_7	C_8

=

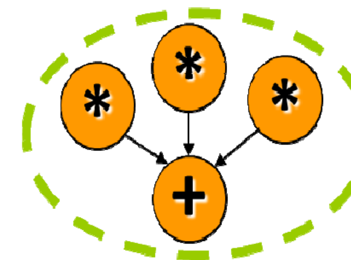
Matrix A		
A_0	A_1	A_2
A_3	A_4	A_5
A_6	A_7	A_8

X

Matrix B		
B_0	B_1	B_2
B_3	B_4	B_5
B_6	B_7	B_8

Equation	Thread
$C_0 = (A_0 * B_0) + (A_1 * B_3) + (A_2 * B_6)$	1
$C_1 = (A_0 * B_1) + (A_1 * B_4) + (A_2 * B_7)$	2
$C_2 = (A_0 * B_2) + (A_1 * B_5) + (A_2 * B_8)$	3
$C_3 = (A_3 * B_0) + (A_4 * B_3) + (A_5 * B_6)$	4
$C_4 = (A_3 * B_1) + (A_4 * B_4) + (A_5 * B_7)$	5
$C_5 = (A_3 * B_2) + (A_4 * B_5) + (A_5 * B_8)$	6

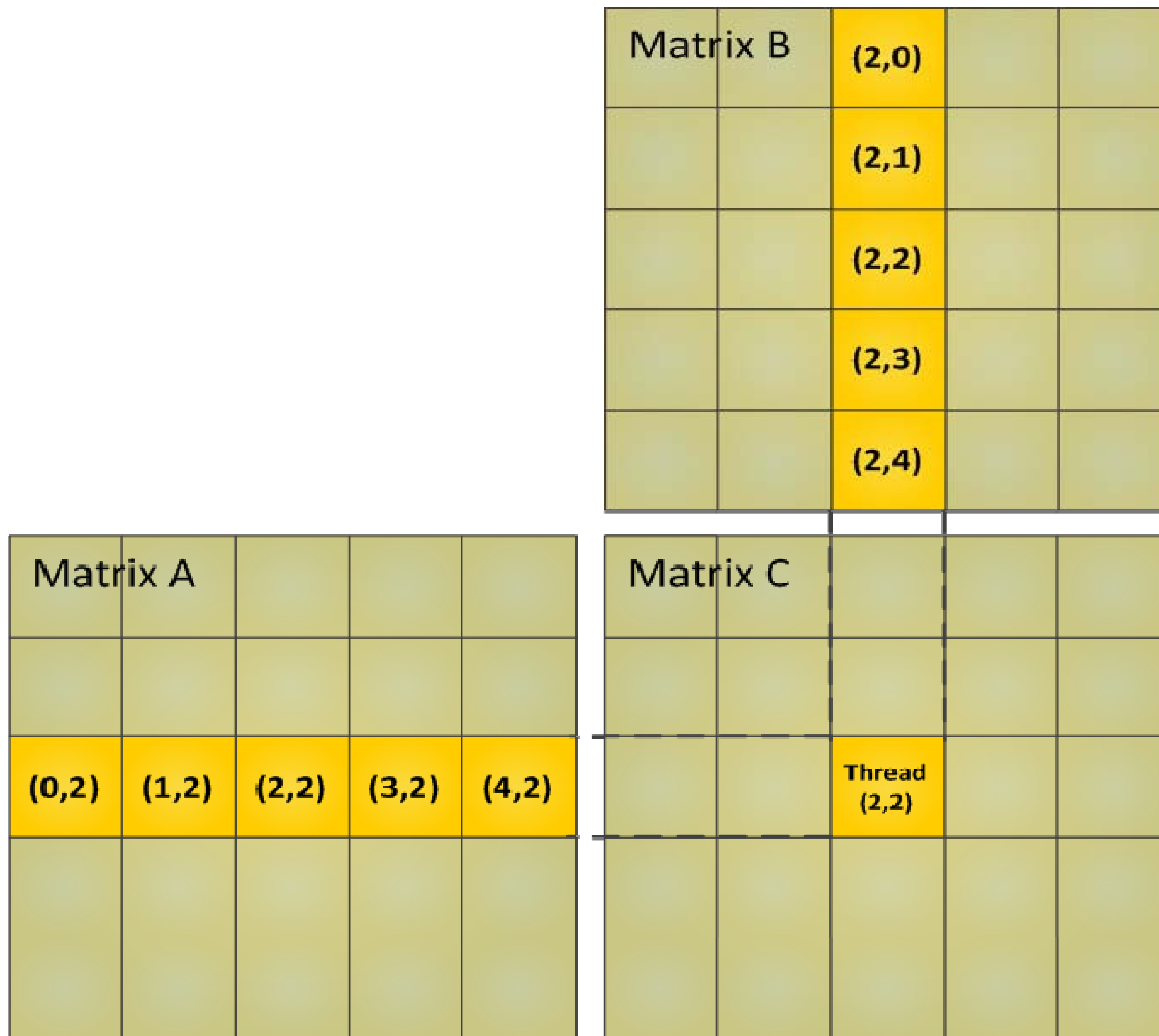
Equation	Thread
$C_6 = (A_6 * B_0) + (A_7 * B_3) + (A_8 * B_6)$	7
$C_7 = (A_6 * B_1) + (A_7 * B_4) + (A_8 * B_7)$	8
$C_8 = (A_6 * B_2) + (A_7 * B_5) + (A_8 * B_8)$	9



Thread

- Fine grained parallelism

Programming the GPU



Matrix Multiplication

- For a ***4096x4096*** matrix multiplication
  - Matrix C will require calculation of ***16,777,216*** matrix cells.
- On the GPU each cell is calculated by its own thread.
- We can have ***23,040 active threads (GTX570)***, which means we can have this many matrix cells calculated in parallel.
- On a general purpose processor we can only calculate one cell at a time.
- Each thread exploits the GPUs fine granularity by computing one element of Matrix C.
- Sub-matrices are read into shared memory from global memory to act as a buffer and take advantage of GPU bandwidth.

Programming the GPU

- Distinguishing execution place of functions:
  - _device_ or _global_ => GPU Device
    - Variables declared are allocated to the GPU memory
  - _host_ => System processor (HOST)
- Function call
  - Name<<dimGrid, dimBlock>>(..parameter list..)
  - blockIdx: block identifier
  - threadIdx: threads per block identifier
  - blockDim: threads per block

CUDA Program Example

```
//Invoke DAXPY
daxpy(n,2.0,x,y);

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

```
//Invoke DAXPY with 256 threads per Thread Block
_host_
int nblocks = (n+255)/256;
daxpy<<<nblocks, 256>>> (n,2.0,x,y);

//DAXPY in CUDA
_device_
void daxpy(int n,double a,double* x,double* y){
        int i=blockIDx.x*blockDim.x+threadIdx.x;
        if (i<n)
                y[i]= a*x[i]+ y[i]
}
```

NVIDIA Instruction Set Arch.

- "Parallel Thread Execution (PTX)"
- 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 R0D, RD0, RD4               ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, 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
```

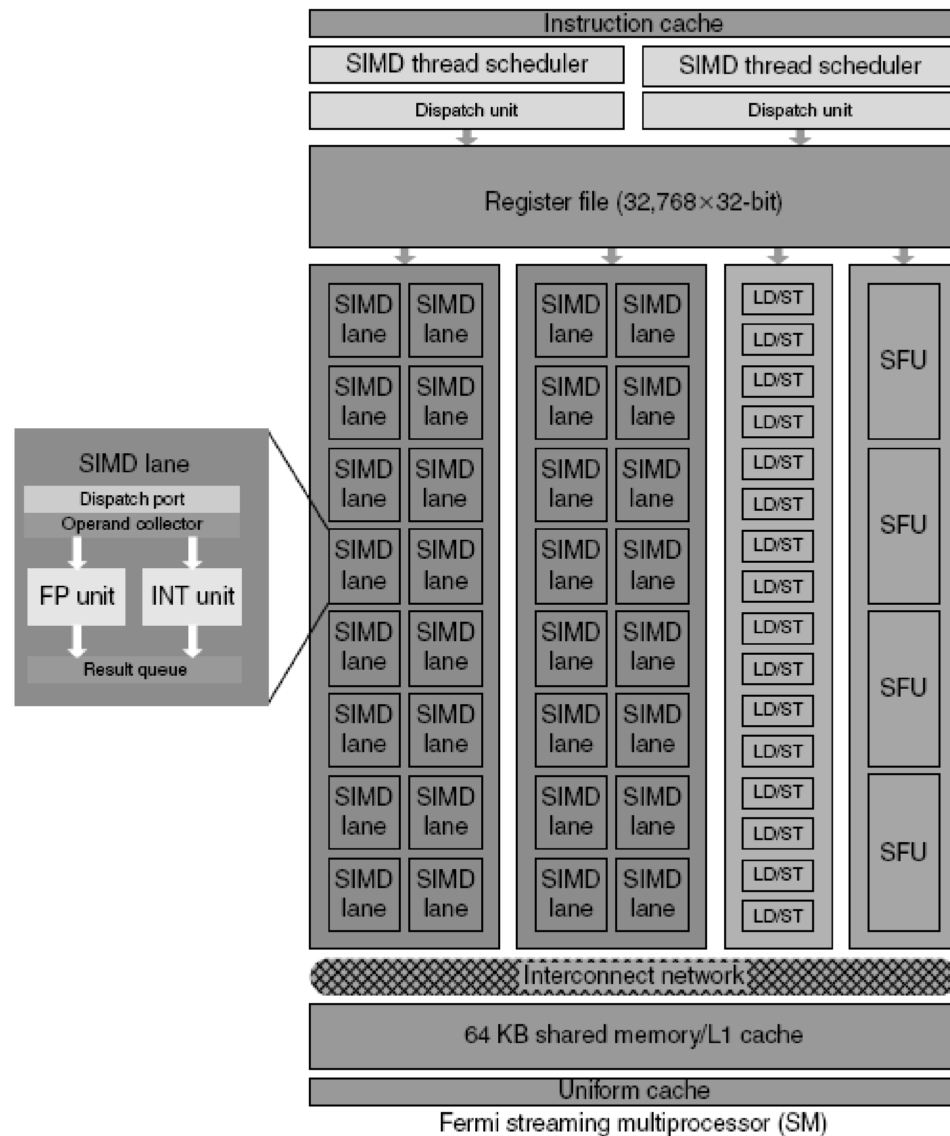
NVIDIA GPU Memory Structures

- 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

Fermi Architecture Innovations

- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles
- Fast double precision
- Caches for GPU memory
- 64-bit addressing and unified address space
- Error correcting codes
- Faster context switching
- Faster atomic instructions

Fermi Multithreaded SIMD Proc.



Compiler Technology for Loop-Level Parallelism

- Loop-carried dependence
  - Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
- Loop-level parallelism has no loop-carried dependence
- Example 1:

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

- No loop-carried dependence

Example 2 for 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
  – Loop-carried dependence
- S2 uses value computed by S1 in same iteration
  – No loop-carried dependence

Remarks

- Intra-loop dependence is not loop-carried dependence
  - A sequence of vector instructions that uses chaining exhibits exactly intra-loop dependence
- Two types of S1-S2 intra-loop dependence
  - Circular: S1 depends on S2 and S2 depends on S1
  - Not circular: neither statement depends on itself, and although S1 depends on S2, S2 does not depend on S1
- A loop is parallel if it can be written without a cycle in the dependences
  - The absence of a cycle means that the dependences give a partial ordering on the statements

Example 3 for Loop-Level Parallelism (1)

- 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 this dependence is not circular.
- There is no dependence from S1 to S2, interchanging the two statements will not affect the execution of S2

Example 3 for Loop-Level Parallelism (2)

- Transform to

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
        B[i+1] = C[i] + D[i];               /*S2*/
        A[i+1] = A[i+1] + B[i+1];           /*S1*/
}
B[100] = C[99] + D[99];
```

- The dependence between the two statements is no longer loop carried.

More on Loop-Level Parallelism

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

- The second reference to A needs not be translated to a load instruction.
  – The two reference are the same. There is no intervening memory access to the same location
- A more complex analysis, i.e. Loop-carried dependence analysis + data dependence analysis, can be applied in the same basic block to optimize

Recurrence

- Recurrence is a special form of loop-carried dependence.
- Recurrence Example:

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

- Detecting a recurrence is important
  – Some vector computers have special support for executing recurrence
  – It my still be possible to exploit a fair amount of ILP

Compiler Technology for Finding Dependences

- To determine which loop might contain parallelism ("inexact") and to eliminate name dependences.
- Nearly all dependence analysis algorithms work on the assumption that array indices are affine.
  - A one-dimensional array index is *affine* if it can written in the form $a \times i + b$ (i is loop index)
  - The index of a multi-dimensional array is affine if the index in each dimension is affine.
  - Non-affine access example: $x[\ y[i]\ ]$
- Determining whether there is a dependence between two references to the same array in a loop is equivalent to determining whether two affine functions can have the same value for different indices between the bounds of the loop.

Finding dependencies Example

- Assume:
  - Load an array element with index $c \times i + d$ and store to $a \times i + b$
  - i runs from m to n
- Dependence exists if the following two conditions hold
  1. Given j, k such that $m \le j \le n$, $m \le k \le n$
  2. $a \times j + b = c \times k + d$
- In general, the values of a, b, c, and d are not known at compile time
  - Dependence testing is expensive but decidable
  - GCD (greatest common divisor) test
    - If a loop-carried dependence exists, then $\mathrm{GCD}(c,a) \mid |d-b|$

Example

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

- Solution:
  1. a=2, b=3, c=2, and d=0
  2. GCD(a, c)=2, |b-d|=3
  3. Since 2 does not divide 3, no dependence is possible

Remarks

- The GCD test is sufficient but not necessary
  - GCD test does not consider the loop bounds
  - There are cases where the GCD test succeeds but no dependence exists
- Determining whether a dependence actually exists is NP-complete

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 */
}
```

- True dependence: S1->S3 (Y[i]), S1->S4 (Y[i]), but not loop-carried.
- Antidependence: S1->S2 (X[i]), S3->S4 (Y[i]) (Y[i])
- Output dependence: S1->S4 (Y[i])

Renaming to Eliminate False (Pseudo) Dependences

- Before:

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

- After:

```
for (i=0; i<100; i=i+1) {
    T[i] = X[i] / c;
    X1[i] = X[i] + c;
    Z[i] = T[i] + c;
    Y[i] = c - T[i];
}
```

Eliminating Recurrence Dependence

- Recurrence example, a dot product:

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

- The loop is not parallel because it has a loop-carried dependence.
- Transform to…

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

This is called scalar expansion.
Scalar ===> Vector
Parallel !!

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

This is called a reduction.
Sums up the elements of the vector
Not parallel !!

Reduction

- Reductions are common in linear algebra algorithm
- Reductions can be handled by special hardware in a vector and SIMD architecture
  - Similar to what can be done in multiprocessor environment
- Example: To sum up 1000 elements on each of ten processors

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

  - Assume p ranges from 0 to 9

Multithreading and Vector Summary

- Explicitly parallel (DLP or TLP) is next step to performance
- Coarse-grained vs. Fine-grained multithreading
  - Switch only on big stall vs. switch every clock cycle
- Simultaneous multithreading, if fine grained multithreading based on OOO superscalar microarchitecture
  - Instead of replicating registers, reuse rename registers
- Vector is alternative model for exploiting ILP
  - If code is vectorizable, then simpler hardware, more energy efficient, and better real-time model than OOO machines
  - Design issues include number of lanes, number of FUs, number of vector registers, length of vector registers, exception handling, conditional operations, and so on.
- Fundamental design issue is memory bandwidth