Computer Architecture
Lecture 8: Data-Level Parallelism in Vector, SIMD, and GPU Architectures (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 processing
  - Machine learning algorithm
- 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

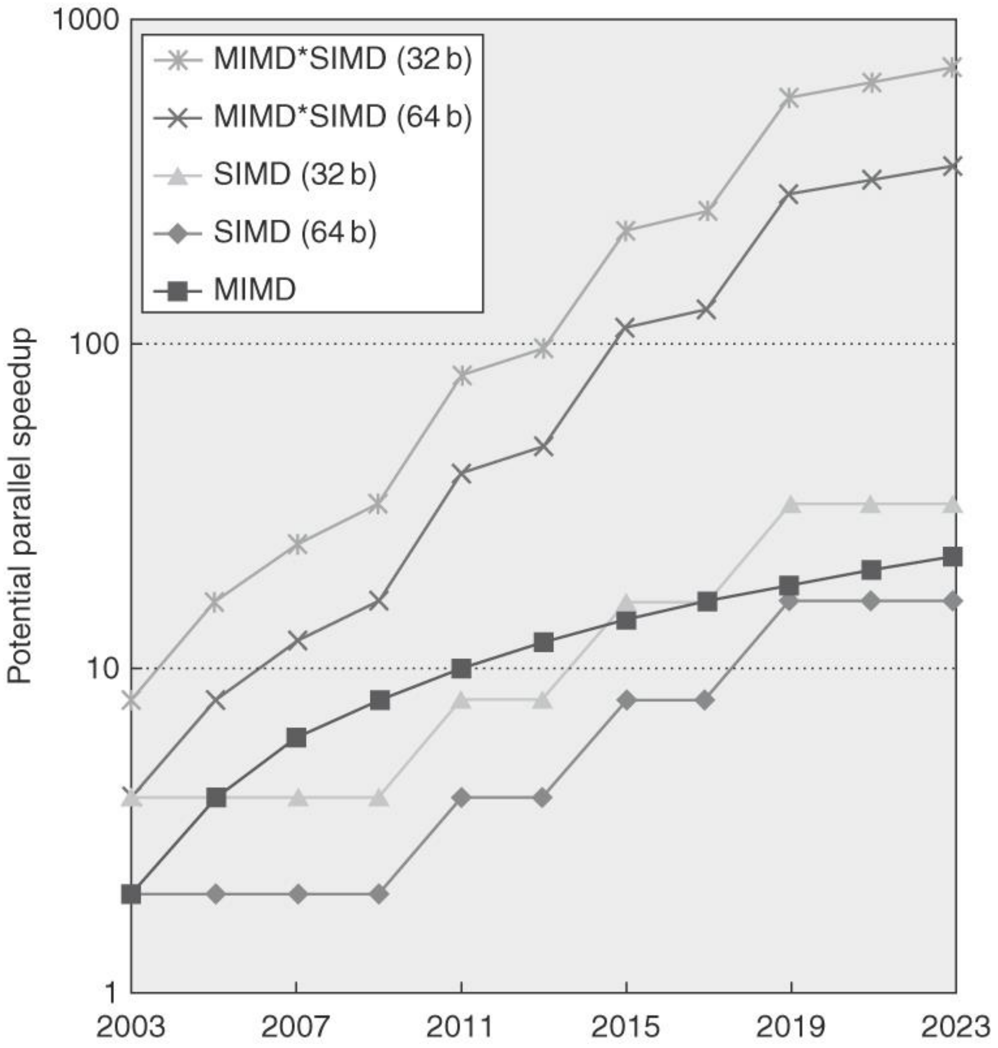
Three SIMD Variations

- Vector architectures
  - Easier to understand but too expensive
- Multimedia SIMD instruction set extensions
  - MMX: multimedia extensions (1996)
  - SSE: streaming SIMD extensions
  - AVX: advanced vector extensions
- Graphics Processor Units (GPUs)
  - Share similar features with vector architectures, but have their own distinguishing characteristics

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

SIMD vs. MIMD

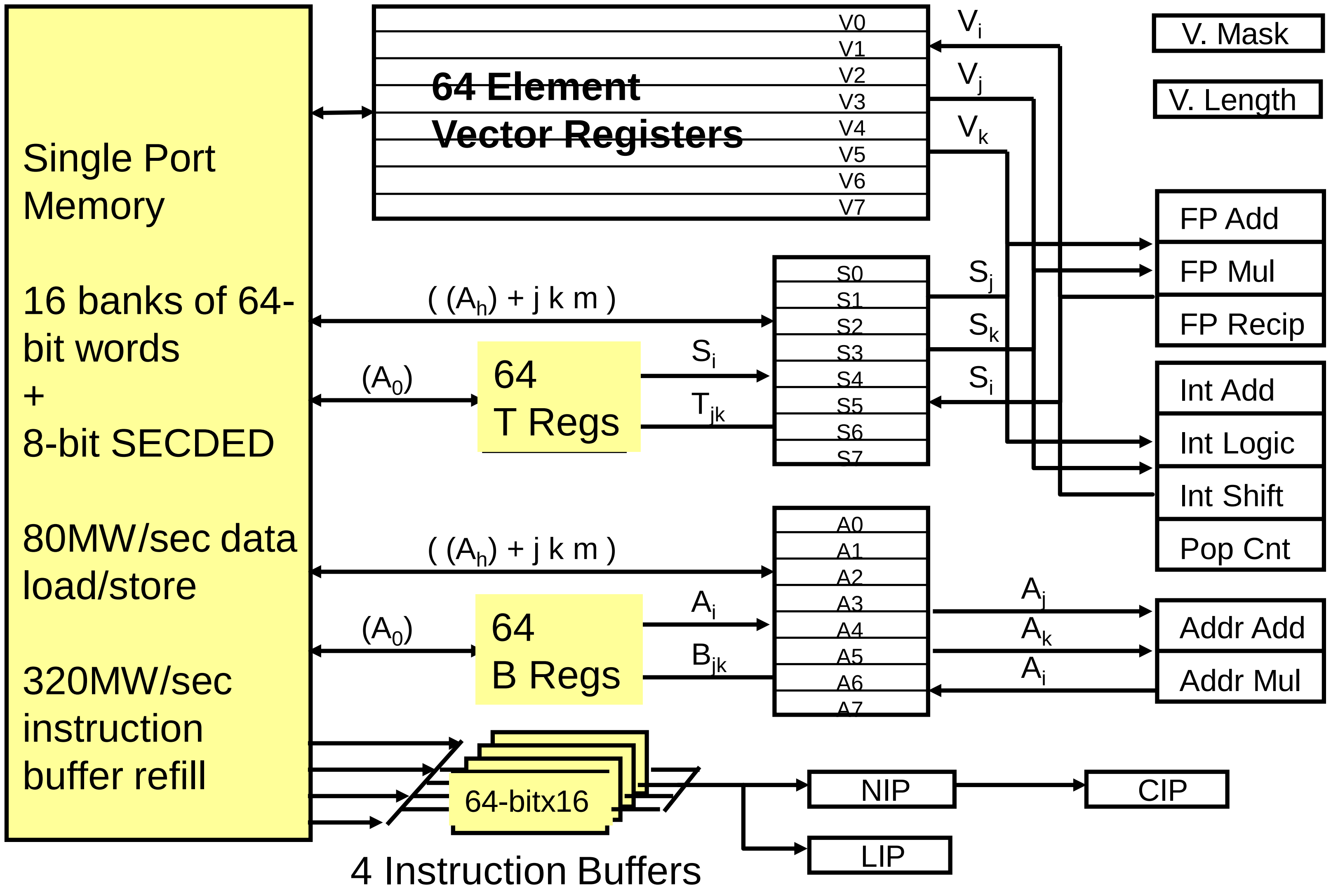
- For x86 processors:
  - Expect two additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!!



Vector 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

Cray-1 (1976)



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

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 (VMM) vs. Vector Register Machines (VRM)

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMAs make it 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*

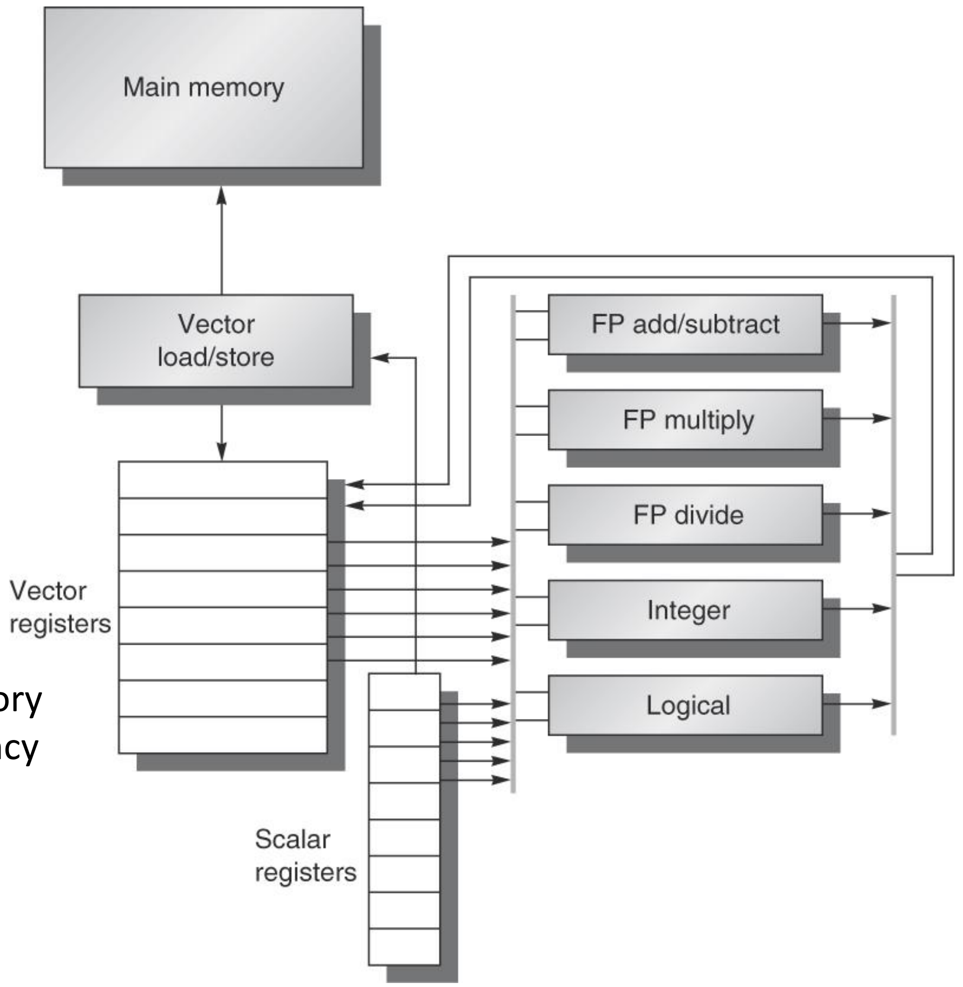
(we ignore vector memory-memory from now on)

VRM Architectures

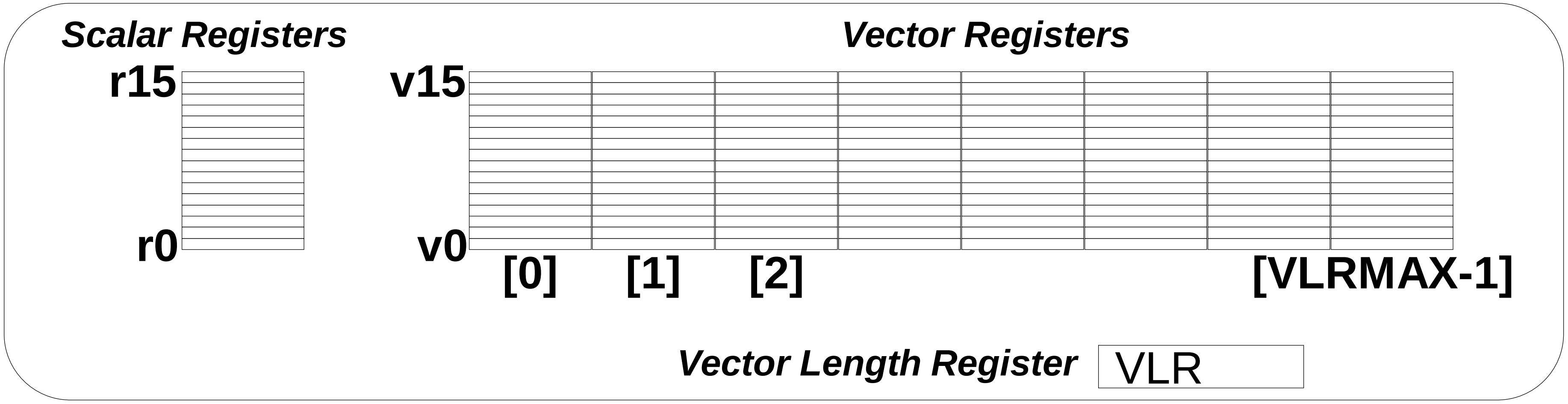
- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on data in those register files
  - Disperse the results back into memory
- Register files are controlled by compiler
  - Register files act as compiler controlled buffers
  - Used to hide memory latency
  - Leverage memory bandwidth
- Vector loads/stores are deeply pipelined
  - Program pays the long memory latency only once per vector load/store vs. latency for each element for regular load/store

RISC-V Vector (RVV) ISA Example: RV64V

- Loosely based on Cray-1
- Vector registers
  - 8 vector registers in this example
  - Each register holds a 32-element, 64 bits/element vector
  - Register file has 16 read ports and 8 write ports
- Vector functional units
  - 5 Fus in this example
  - Fully pipelined
  - Structural and data hazards are detected
- Vector load-store unit
  - Fully pipelined
  - Words move between registers and memory
  - One word per clock cycle after initial latency
- Scalar registers
  - 31 general-purpose registers
  - 32 floating-point registers (RV64V)

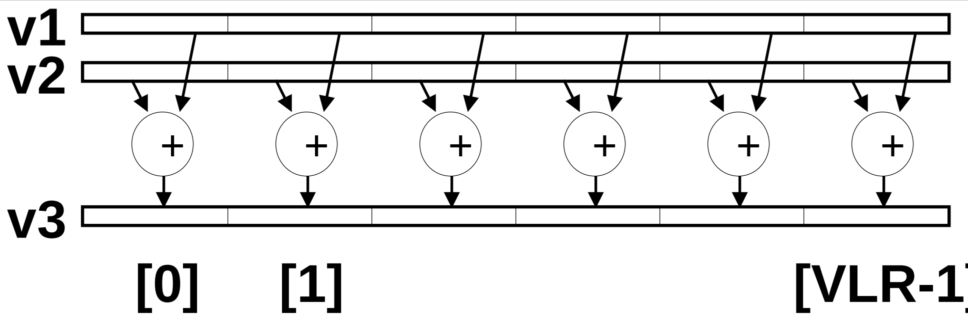


Vector Programming Model



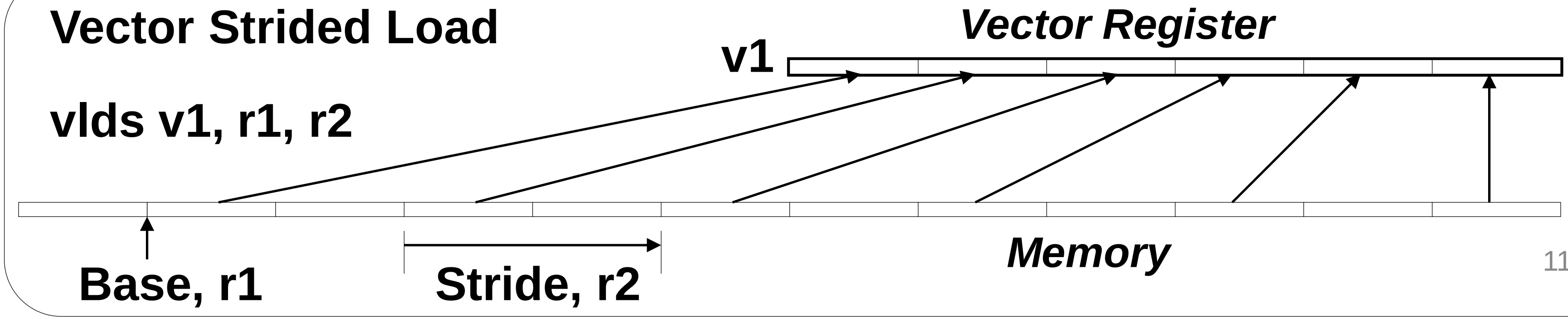
Vector Arithmetic

vadd v3, v1, v2



Vector Strided Load

vlds v1, r1, r2



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

RV64V Vector Instructions

Mnemonic	Name	Description
vadd	ADD	Add elements of V[rs1] and V[rs2], then put each result in V[rd]
vsub	SUBtract	Subtract elements of V[rs2] frpm V[rs1], then put each result in V[rd]
vmul	MULtiply	Multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
vdiv	DIVide	Divide elements of V[rs1] by V[rs2], then put each result in V[rd]
vrem	REMainder	Take remainder of elements of V[rs1] by V[rs2], then put each result in V[rd]
vsqrt	SQuare RooT	Take square root of elements of V[rs1], then put each result in V[rd]
vsll	Shift Left	Shift elements of V[rs1] left by V[rs2], then put each result in V[rd]
vsrl	Shift Right	Shift elements of V[rs1] right by V[rs2], then put each result in V[rd]
vsra	Shift Right Arithmetic	Shift elements of V[rs1] right by V[rs2] while extending sign bit, then put each result in V[rd]
vxor	XOR	Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vor	OR	Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vand	AND	Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd]
vsgnj	SiGN source	Replace sign bits of V[rs1] with sign bits of V[rs2], then put each result in V[rd]
vsgnjn	Negative SiGN source	Replace sign bits of V[rs1] with complemented sign bits of V[rs2], then put each result in V[rd]
vsgnjx	Xor SiGN source	Replace sign bits of V[rs1] with xor of sign bits of V[rs1] and V[rs2], then put each result in V[rd]
vld	Load	Load vector register V[rd] from memory starting at address R[rs1]
vlds	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., R[rs1] + i × R[rs2])
vldx	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2] + V[rs2] (i.e., V[rs2] is an index)
vst	Store	Store vector register V[rd] into memory starting at address R[rs1]
vsts	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1] + i × R[rs2])
vstx	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2] + V[rs2] (i.e., V[rs2] is an index)
vpeq	Compare =	Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpne	Compare !=	Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vplt	Compare <	Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpxor	Predicate XOR	Exclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpor	Predicate OR	Inclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpand	Predicate AND	Logical AND 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
setvl	Set Vector Length	Set vl and the destination register to the smaller of mvl and the source regsiter

Integer	8, 16, 32, and 64 bits	Floating point	16, 32, and 64 bits

Figure 4.3 Data sizes supported for RV64V assuming it also has the single- and double-precision floating-point extensions RVS and RVD. Adding RVV to such a RISC-V design means the scalar unit must also add RVH, which is a scalar instruction extension to support half-precision (16-bit) IEEE 754 floating point. Because RV32V would not have doubleword scalar operations, it could drop 64-bit integers from the vector unit. If a RISC-V implementation didn't include RVS or RVD, it could omit the vector floating-point instructions.

Conti…RV64V Instructions

- Suffix .vv: both operands are vectors
- Suffix .sv (or .vs): the first (or the second) operand is a scalar
  - 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
    - 32 64-bit / 128 16-bit / 256 8-bit
    - Matches the need of multimedia (8bit), scientific applications that require high precision

RV64V Example: DAXPY Loop

- **`for (i=0; i<32; i++)`**

  `Y[i] = a × X[i] + Y[i]`

  adds a scalar multiple of a double precision vector to another double precision vector
- Assume the starting addresses of `X` and `Y` are in `x5` and `x6`
- Assume the number of elements (i.e. Length) of the vectors matches the length of the vector operation.

Here is the RISC-V code:

```
       fld    f0,a
       addi   x28,x5,#256
Loop:  fld    f1,0(x5)
       fmul.d f1,f1,f0
       fld    f2,0(x6)
       fadd.d f2,f2,f1
       fsd    f2,0(x6)
       addi   x5,x5,#8
       addi   x6,x6,#8
       bne    x28,x5,Loop
```

Enable 4 DP FP vector registers

Here is the RV64V code for DAXPY:

```
vsetdcfg  4*FP64
fld       f0,a
vld       v0,x5
vmul      v1,v0,f0
vld       v2,x6
vadd      v3,v1,v2
vst       v3,x6
vdisable
```

Disable vector registers

Remarks

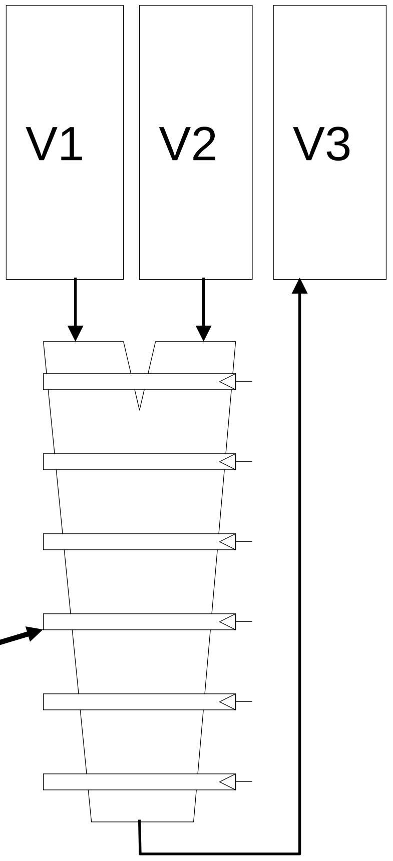
- 8 RV64V vector instructions vs. 258 RV64G scalar instructions
- In RV64G Code
  - Fadd.d must wait for fmul.d
  - fsd must wait for fadd.d
  - Lots of pipeline stalls are necessary for deeply pipelined architecture.
- In RV64V Code
  - Stall once for the first vector element, subsequent elements will flow smoothly down the pipeline (discussed later !!).
  - Pipeline stalls are required only once per vector instruction, rather than once per vector element
- Pipeline stall frequency on RV64G will be about 32× higher than it is on RV64V.

Challenges of Vector Instructions

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

Deeply Pipelined Vector Arithmetic Execution

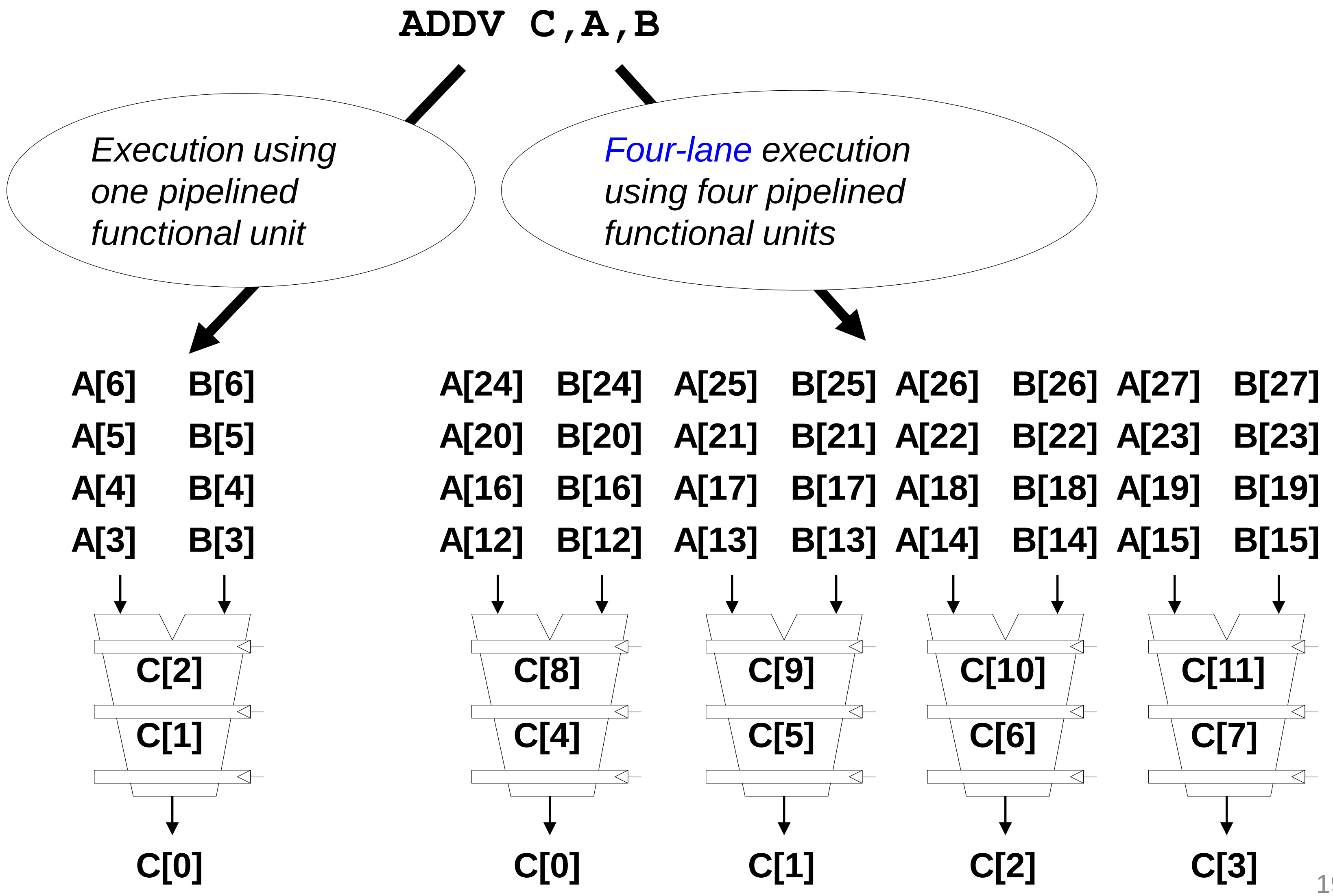
- Use *deep pipeline* (=> fast clock) to execute element operations
  - Hide long instruction latency
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)



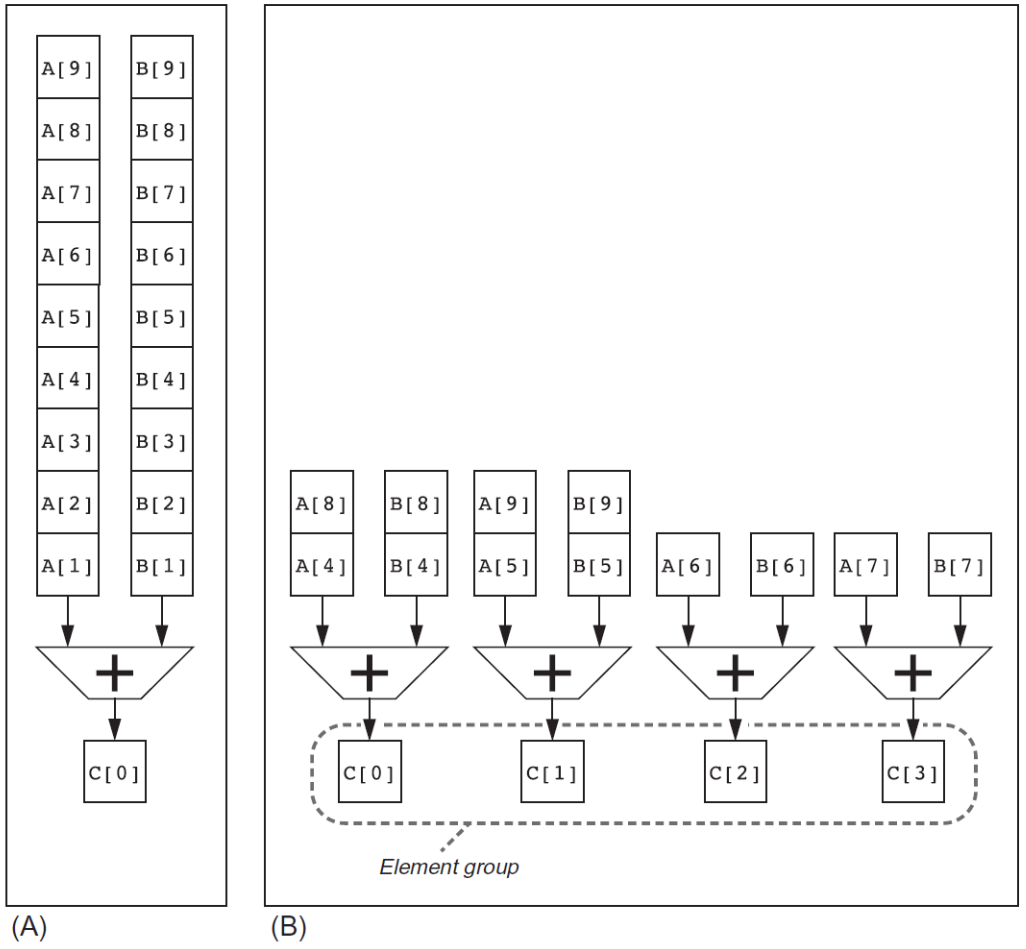
Six stage multiply pipeline

```
V3 <- v1 * v2
```

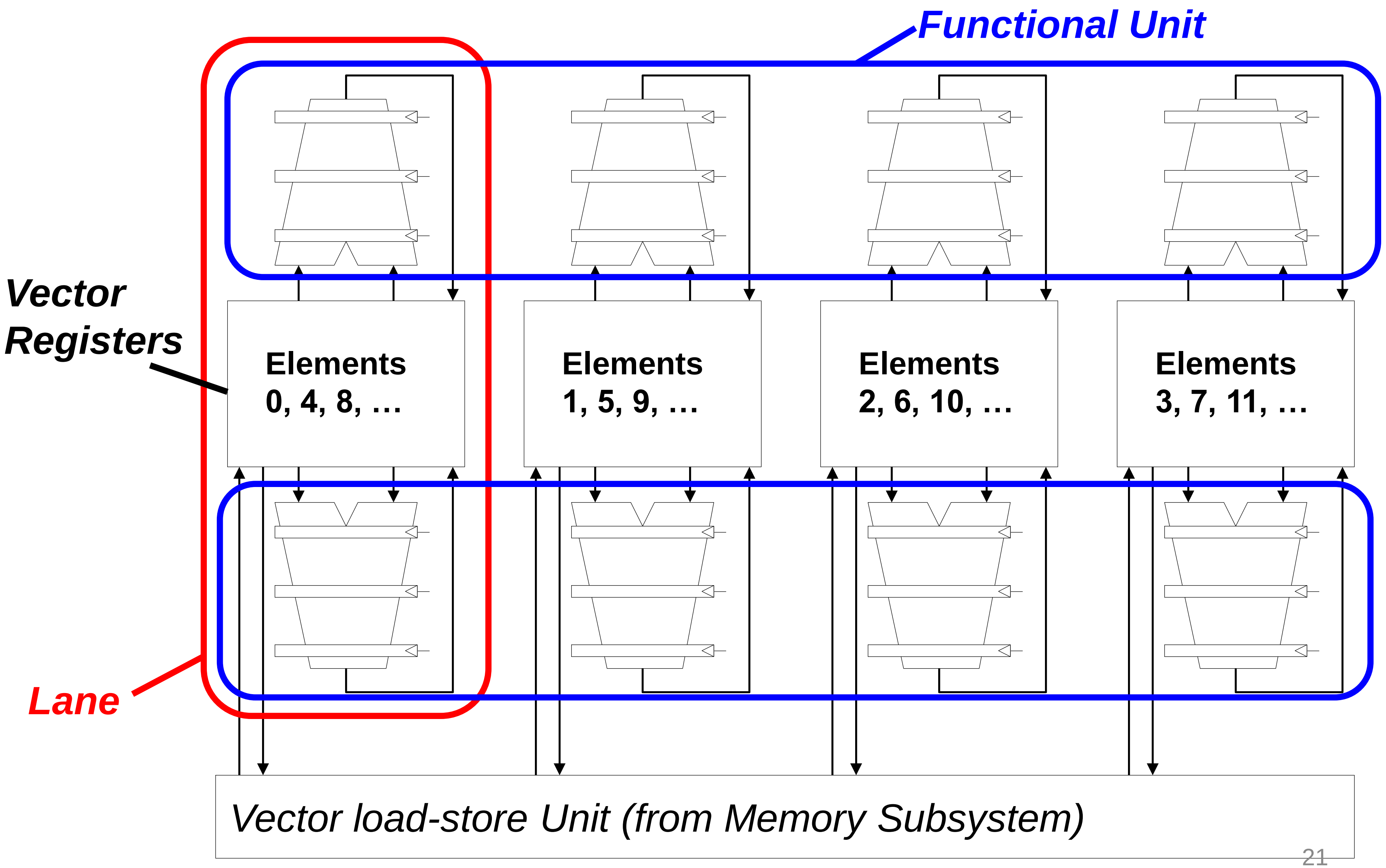
Multiple Lanes for High Throughput


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

Multiple Lanes: Beyond One Element per Clock Cycle



Datapath for Multiple Lanes Structure

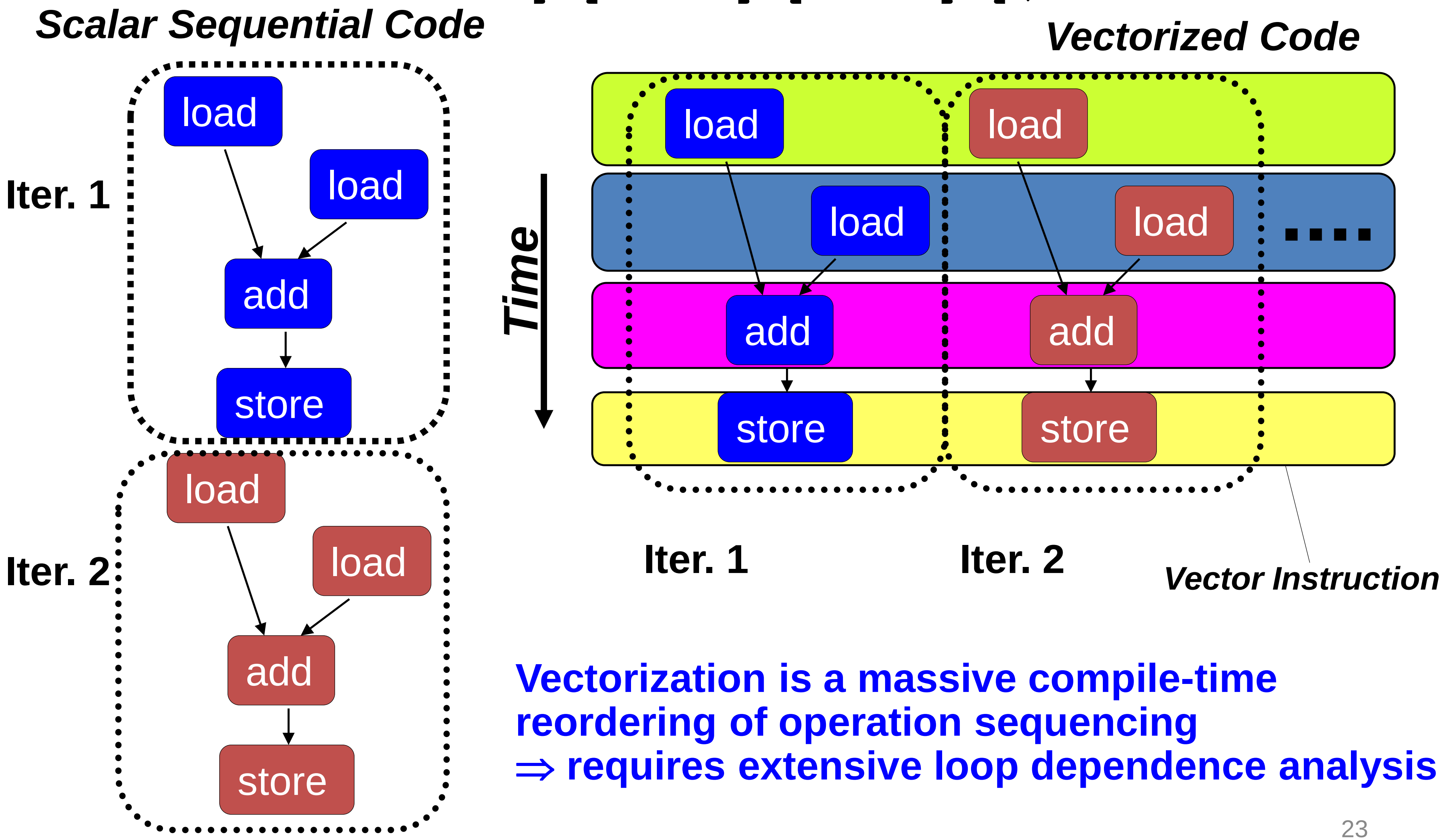


Remarks

- High throughput multiple lanes structure
  - Adding more lanes allows designers to tradeoff clock rate and energy without sacrificing performance!
  - Register files are physically partitioned.
    - Element n of vector register A is “hardwired” to element n of vector register B
  - No communication between lanes
    - The first lane holds the first element (element 0) for all vector registers, and so the first element in any vector instruction will have its source and destination operands located in the first lane.
  - Little increase in control overhead and does not require changes to existing machine code

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

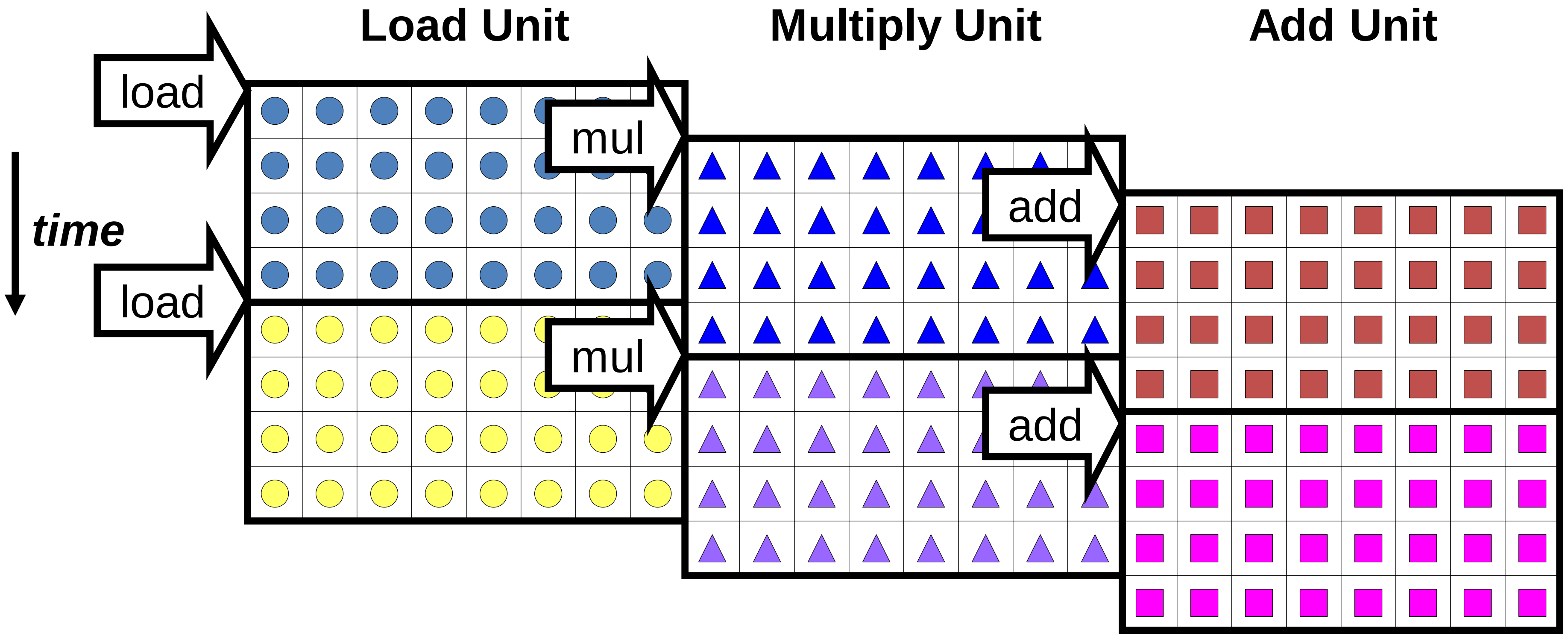
- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies

- RV64V functional units consume one element per clock cycle
  - Execution time is approximately the vector length
- Efficient way to estimate the execution time:
  - *Convoy and chime*

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

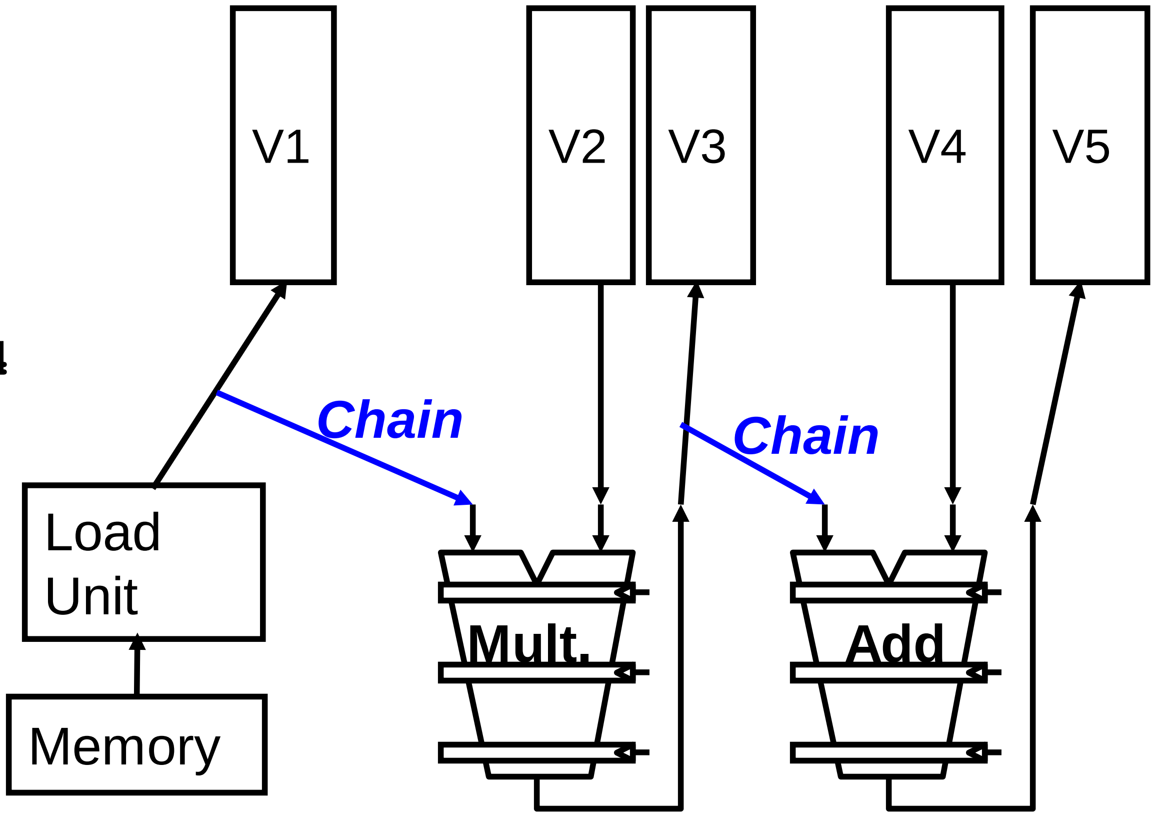
- Convoy: set of vector instructions that could potentially execute together
  - Must not contain structural hazards
  - Sequences with RAW hazards should be in different convoys

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

Vector Chaining

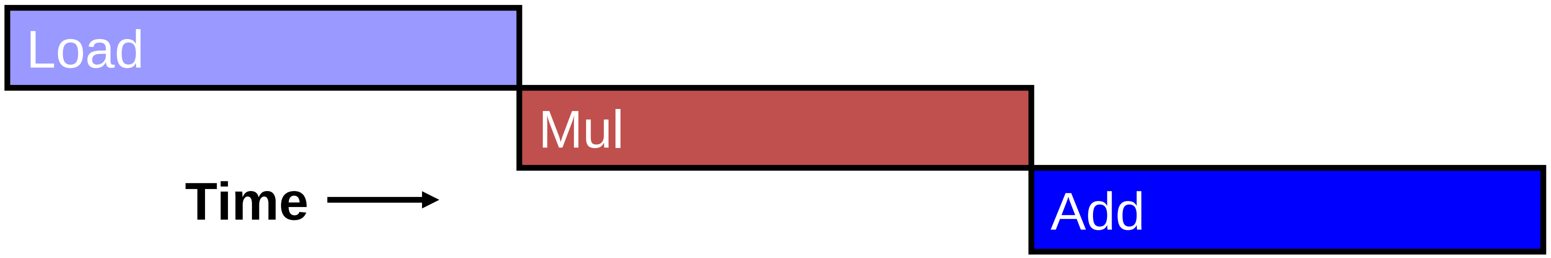
- Vector version of register bypassing
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available

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

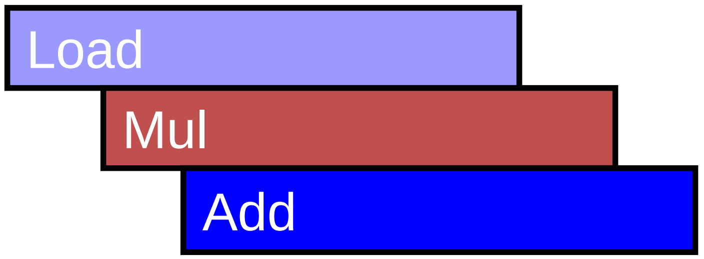


Advantages of Vector Chaining

- 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



Chime

- Time of metric to estimate the length of a convey
- *Chime*: unit of time to execute one convoy
- Assume *m* convoy executes in *m* chimes. For vector length of *n*, it requires totally $m \times n$ clock cycles

- The chime approximation ignores some processor-specific overhead (which are dependent on vector length)
  - Measuring time in chimes is a better approximation for long vectors than a short vector

Execution Time Example

```
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
```

3 Convoys:

1	vld	vmul (via chaining)
2	vld	vadd (via chaining)
3	vst	

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

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

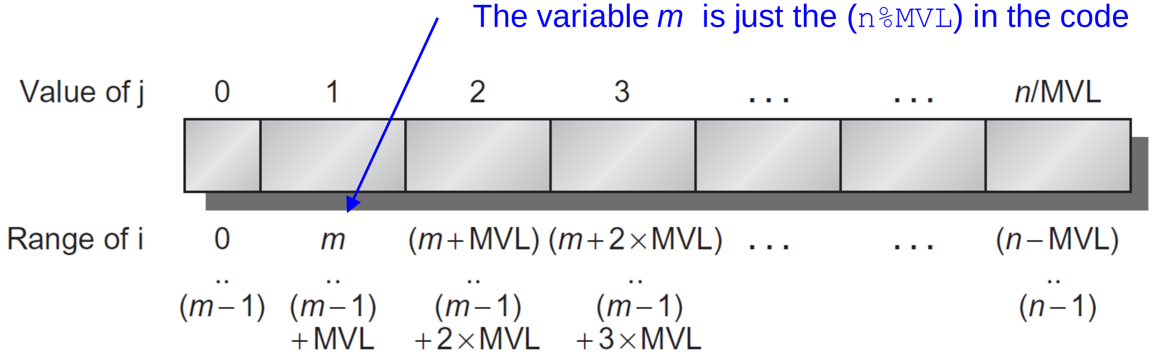
Vector Length Register (`vl`)

- A vector register processor has a natural vector length determined by the *maximum vector length* (`mvl`).
  - For example, mvl = 32 in RV64V
  - The `mvl` is the physical limit of a vector processor
- The length of a particular vector operation is often unknown at compile time.
- The solution to these problems is to add a *vector-length register* (vl) + *strip mining*.
- Vector Strip-Mining
  - Handling loops not equal to `mvl`

A vector of arbitrary length processed with 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*/
}
```

The variable m is just the (`n%MVL`) in the code

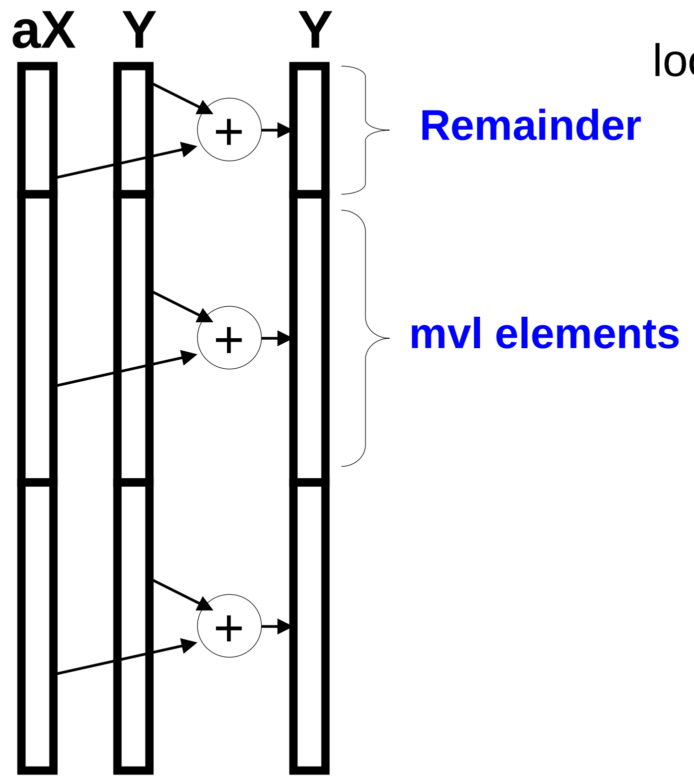


Break loops into pieces that fit into vector registers

- All blocks but the first are of length MVL, utilizing the full power of the vector processor.

Vector Strip-Mining Example

```
for (i=0; i<n; i++)
    Y[i] = a * X[i] + Y[i];
```



```
      vsetdcfg 2 DP FP   # Enable 2 64b Fl.Pt. registers
      fld f0,a           # Load scalar a
loop: setvl t0,a0        # vl = t0 = min(mvl,n)
      vld v0,x5          # Load vector X
      slli t1,t0,3       # t1 = vl * 8 (in bytes)
      add x5,x5,t1       # Increment pointer to X by vl*8
      vmul v0,v0,f0      # Vector-scalar mult
      vld v1,x6          # Load vector Y
      vadd v1,v0,v1      # Vector-vector add
      sub a0,a0,t0       # n -= vl (t0)
      vst v1,x6          # Store the sum into Y
      add x6,x6,t1       # Increment pointer to Y by vl*8
      bnez a0,loop       # Repeat if n != 0
      vdisable           # Disable vector regs}
```

Problems of Low-to-Moderate Levels of Vectorization

- The presence of conditionals (IF statements) inside loops and the use of sparse matrices are two main reasons for lower levels of vectorization.
- Consider:

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

  - This loop cannot normally be vectorized because of the conditional execution of the body;
- However, if the inner loop could be run for the iterations for which X[i] $\neq$ 0, then the subtraction could be vectorized.
- The common extension for this capability is *vector-mask control*.

Vector-Mask Control: Handling IF Statements in Vector Loops

- for (i = 0; i < 64; i=i+1)

  if (X[i] != 0)

  X[i] = X[i] – Y[i];

- Use *predicate register* to "disable" elements:

```
vsetdcfg    2*FP64     # Enable 2 64b FP vector regs
vsetpcfgi   1          # Enable 1 predicate register
vld         v0,x5      # Load vector X into v0
vld         v1,x6      # Load vector Y into v1
fmv.d.x     f0,x0      # Put (FP) zero into f0
vpne        p0,v0,f0   # Set p0(i) to 1 if v0(i)!=f0
vsub        v0,v0,v1   # Subtract under vector mask
vst         v0,x5      # Store the result in X
vdisable               # Disable vector registers
vpdisable              # Disable predicate registers
```

These registers use a Boolean vector to control the execution of a vector instruction.

When the predicate register p0 is set, all following vector instructions operate only on the vector elements whose corresponding entries in the predicate register are 1. The entries in the destination vector register that correspond to a 0 in the mask register are unaffected by the vector operation.

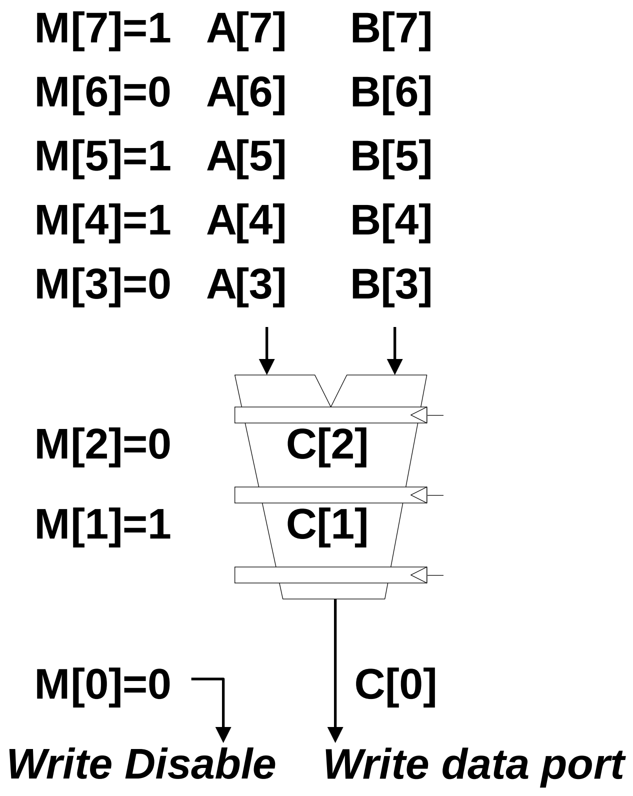
Vector-Mask Control (2)

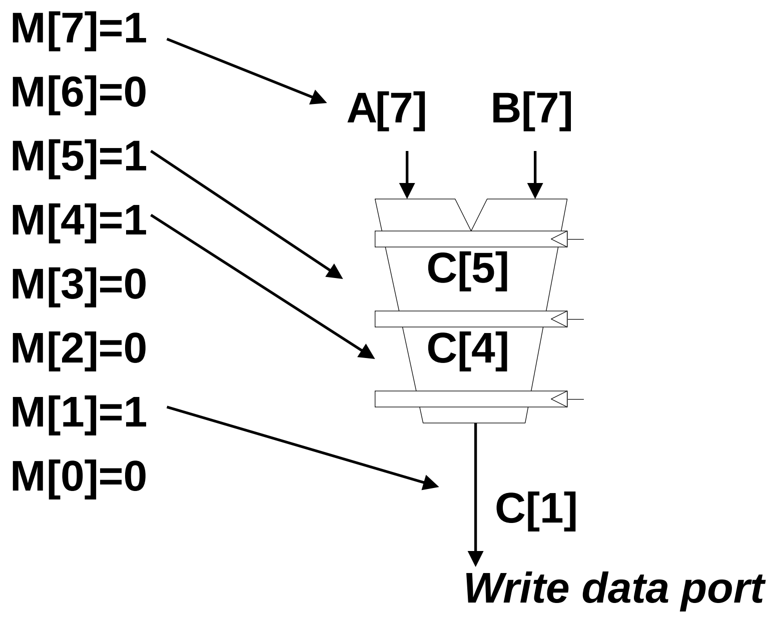
Simple Implementation

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

Density-Time Implementation

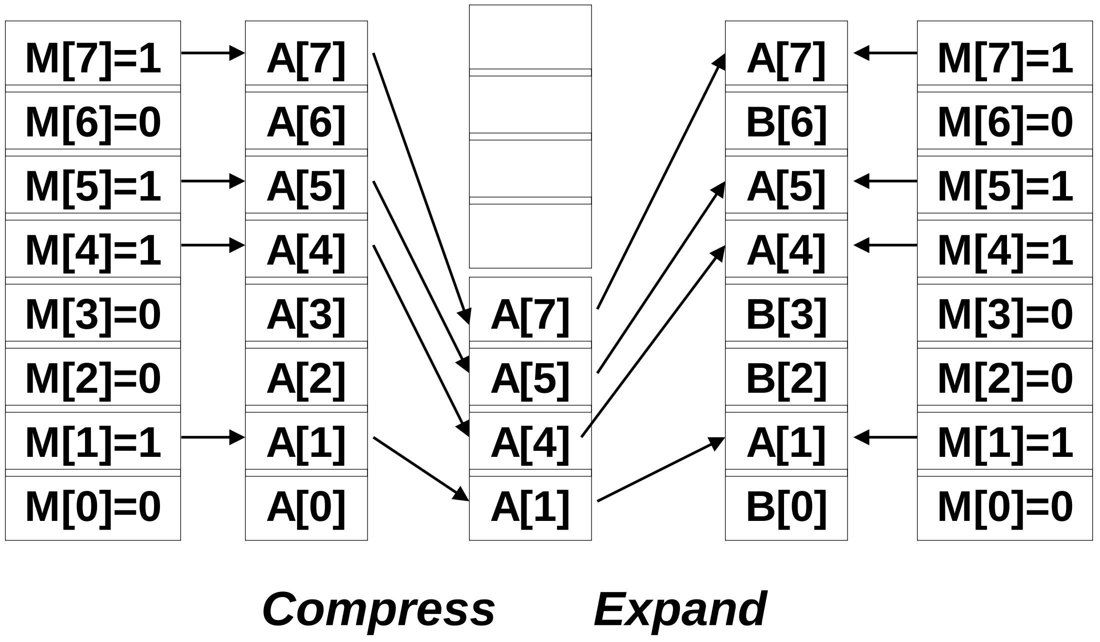
– scan mask vector and only execute elements with non-zero masks





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



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

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
- GFLOPS rate drops when masks are used

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?
  - Penalties for start-ups on load/store units are higher than those for arithmetic unit
  - vector load/store units play a similar role to prefetch units in scalar processors
- 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 to allow all processors to run at the full memory bandwidth?

- Solution:

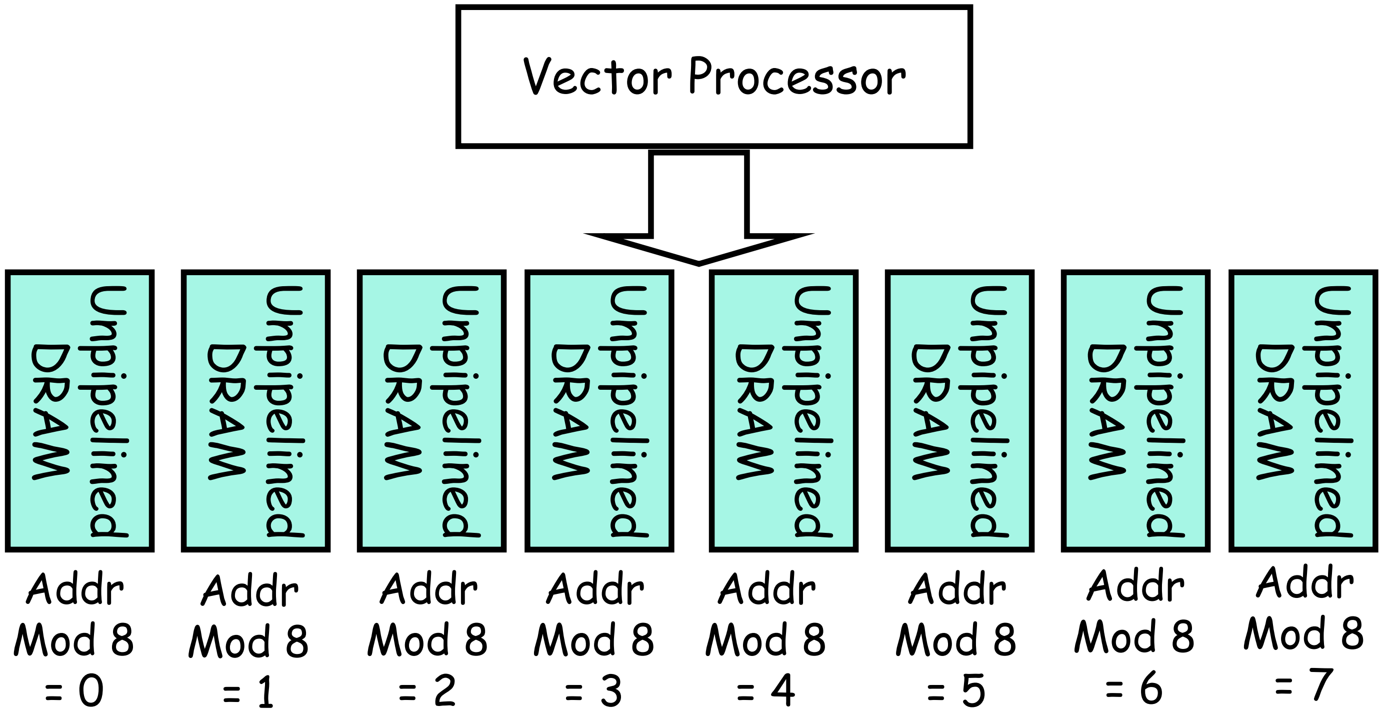
1. The maximum number of memory references each cycle is $32\times6=192$
2. SRAM takes $15/2.167=6.92\approx7$ processor cycles
3. It requires $192\times7=1344$ memory banks at full memory bandwidth!!

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

Handling Multidimensional Arrays in Vector Architectures

- Load/store units move groups of data between vector registers and memory
- **The distance separating elements to be gathered into a single vector register is called the *stride*.**
- Three types of stride addressing
  - ***Unit stride***
    - Contiguous (sequential) 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 for Banked Memory



- 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 (in this example)
  - Bad for: 2, 4

How to get full bandwidth for vector load/store units?

- 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
- Supporting strides greater than one complicates the memory system.
  - Once we introduce nonunit strides, it becomes possible to request accesses from the same bank frequently
  - Memory bank conflict problem !!
- 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

    (or address within bank = address mod number words in bank)
  - modulo & divide per memory access with prime no. banks?

    (prime number of banks ???) (easy if 2^N words per bank)

Example: Multidimensional Arrays in Vector Architectures

```
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];
      }
```

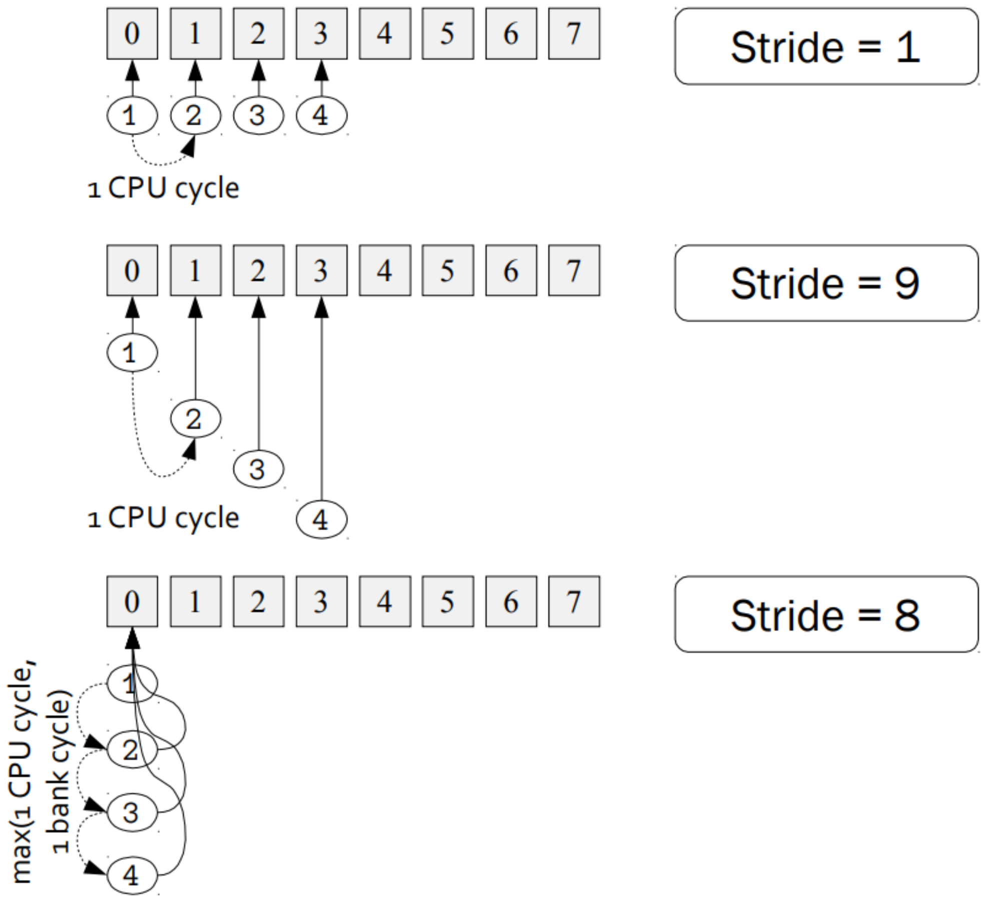
- Must vectorize multiplications of rows of B with columns of D
  - Elements of B accessed in row-major order but elements of D in column-major order!
  - Either the elements in the row or the elements in the column are not adjacent in memory
- Once vector is loaded into the register, it acts as if it has logically adjacent elements

Conti..

- Cache inherently deals with unit stride data
  - Blocking techniques might help non-unit stride data
- In the example (for vector processors without caches)
  - Matrix D has a stride of 100 double words
  - Matrix B has a stride of 1 double word
- Use *non-unit stride addressing* for Matrix 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 VLDS/VSTS: vector load/store with stride
      - The vector stride, like the vector starting address, can be put in a general-purpose register

Problem of Memory Bank Conflict

- Bank busy time: minimum period of time between successive accesses to the same bank
- Bank conflict (stall) occurs when the same bank is hit faster than 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 $12+1+63\times6=391$ cycles, i.e. 6.1 cycles per element

Gather-Scatter: Handling Sparse Matrices in Vector Architecture

- 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 are index vectors to designate the nonzero elements of A and C
- *Gather-scatter* operations are used for handling sparse matrices
  - A *gather operation* (`vldi` (vector load indexed)) takes an index vector (and fetches the vector) whose elements are at the addresses given by adding a base address to the offsets given in the index vector.
  - The result of the gather operation is a dense vector in a vector register.
  - The sparse vector can be stored in an expanded form by a *scatter store* (`vsti` (vector store indexed), using the same index vector

Gather-Scatter Example

- C code

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

- if `x5`, `x6`, `x7`, and `x28` contain the starting addresses of the vectors of `A`, `C`, `K`, and `M`
- RV64V code sequence:

```
Vsetdcfg    4*FP64        # 4 64b FP vector registers
vld         v0, x7        # Load K[]
vldi        v1, x5, v0    # Load A[K[]]
vld         v2, x28       # Load M[]
vldi        v3, x6, v2    # Load C[M[]]
vadd        v1, v1, v3    # Add them
vsti        v1, x5, v0    # Store A[K[]]
vdisable                  # Disable vector registers
```

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!
- Fundamental design issue is memory bandwidth
- Other design issues: number of lanes, functional units and registers, length of vector registers, exception handling, conditional operations, and so on.

Programming Vector Architectures

- Compilers can tell at compile time whether a section of code will vectorize or not
- Programmers can provide hints to compiler
- Cray Y-MP Benchmarks (Level of vectorization)

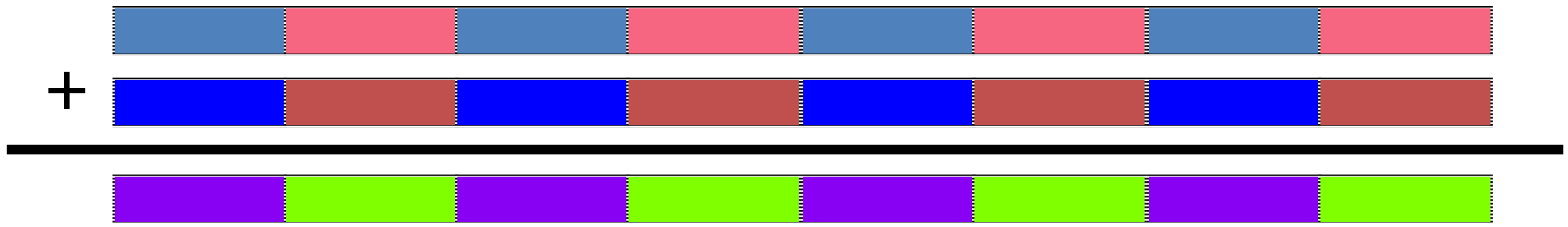
Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid
BDNA	96.1%	97.2%
MG3D	95.1%	94.5%
FLO52	91.5%	88.7%
ARC3D	91.1%	92.0%
SPEC77	90.3%	90.4%
MDG	87.7%	94.2%
TRFD	69.8%	73.7%
DYFESM	68.8%	65.6%
ADM	42.9%	59.6%
OCEAN	42.8%	91.2%
TRACK	14.4%	54.6%
SPICE	11.5%	79.9%
QCD	4.2%	75.1%

SIMD Instruction Set Extensions for Multimedia

- Media applications usually operate on narrower data types (e.g., 8-bit)
- SIMD ISA are added to existing ISAs for multimedia application
- In RV64V, 64-bit registers can split into 2x32b or 4x16b or 8x8b
- Newer designs have 512-bit registers (AVX-512 in 2017)
- In contrast to vector architectures, SIMD extensions have three major limitations:
  - No vector length register
  - No strided or scatter/gather data transfer instructions
  - No mask registers
- Such omissions make it harder for the compiler to generate SIMD code and increase the difficulty of programming in SIMD assembly language.

“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

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

SIMD Implementations Summary

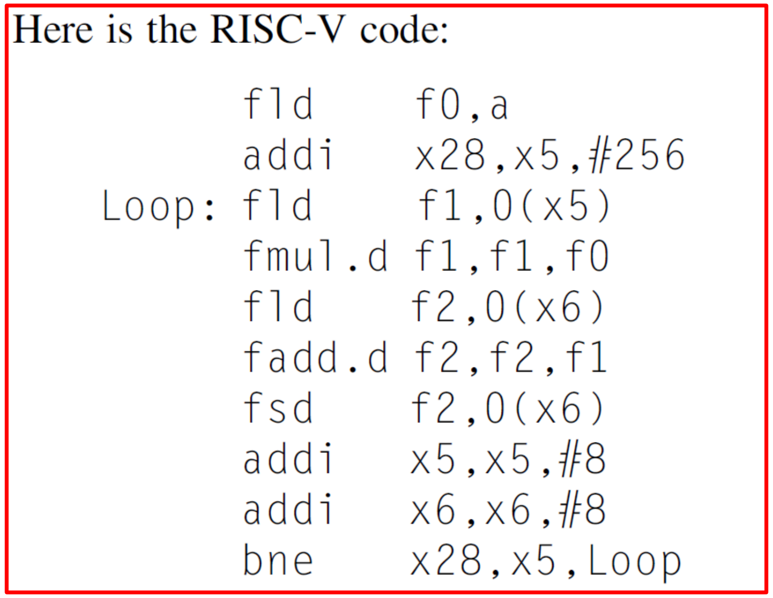
- Intel MMX 1996
  - For the x86 architecture (64-bit floating-point registers)
  - 8 8-bit or 4 16-bit integer ops
- Streaming SIMD Extensions (SSE) 1999, SSE2/3/4 (2001/2004/2007)
  - 128-bit XMM registers
  - 16 8-bit, 8 16-bit, or 4 32-bit integer ops
  - 4 32-bit, or 2 64-bit single-/double- precision FP ops
- Advanced Vector Extensions (AVX) 2010
  - 256-bit YMM registers
  - 4 64-bit dp FP ops
- AVX-512 2017
  - 512-bit registers
  - 8 64-bit dp FP ops

Why SIMD Extensions?

1. Cost little to add to the standard arithmetic unit and easy to implement
2. Require scant extra processor state compared to vector architectures, which is always a concern for context switch times
3. Need smaller memory bandwidth than vector
4. Use separate data transfer aligned in memory. No need to deal with problem in virtual memory (e.g., page fault)
   – Vector: single instruction, 64 memory accesses, page fault in the middle of the vector likely !!
5. Easy to add new instructions for new media operations
6. No need for sophisticated mechanisms of vector architecture

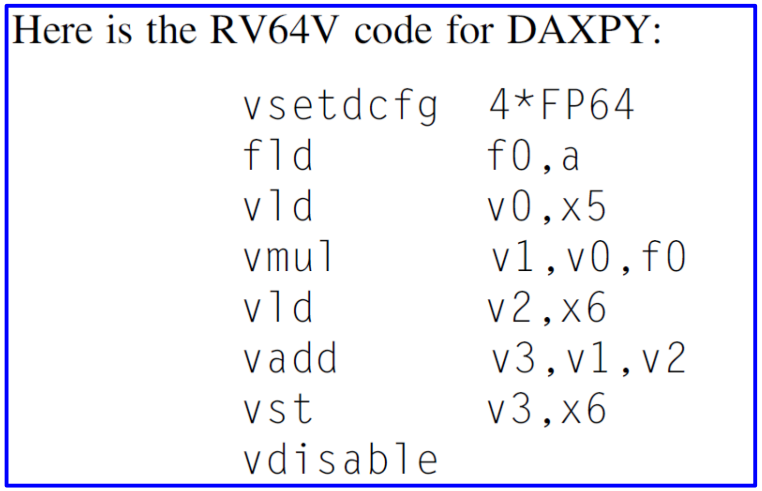
RVP: RISC-V SIMD

- RVP stands for “*packed*“ instruction.
- 256-bit SIMD multimedia instructions
- Add the suffix “4D” on instructions
  - Operate 4 double-precision floating-point operands
- Let’s consider DAXPY loop example



Here is the RISC-V code:

```
        fld     f0,a
        addi    x28,x5,#256
Loop:   fld     f1,0(x5)
        fmul.d  f1,f1,f0
        fld     f2,0(x6)
        fadd.d  f2,f2,f1
        fsd     f2,0(x6)
        addi    x5,x5,#8
        addi    x6,x6,#8
        bne     x28,x5,Loop
```



Here is the RV64V code for DAXPY:

```
        vsetdcfg   4*FP64
        fld        f0,a
        vld        v0,x5
        vmul       v1,v0,f0
        vld        v2,x6
        vadd       v3,v1,v2
        vst        v3,x6
        vdisable
```

Example RISC-V SIMD Code

- Example DXPY loop:

```
      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[i] ... a×X[i+3]
      fld.4D    f2,0(x6)      #Load Y[i] ... Y[i+3]
      fadd.4D   f2,f2,f1      #a×X[i]+Y[i]...
                              #a×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
```

Remarks for RISC-V SIMD Example

- The changes were replacing every RISC-V double-precision instruction with its 4D equivalent, increasing the increment from 8 to 32
- It adds the **`splat` instruction** that makes 4 copies of `a` in the 256 bits of `f0`.
- RISC-V SIMD does get almost a 4× reduction of instruction bandwidth: 67 versus 258 instructions executed for RV64G.
- Note that the RISC-V SIMD code might require an extra strip-mine loop to handle the case when the loop number is not a modulo of 4.

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
- 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

Roofline Performance Model

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

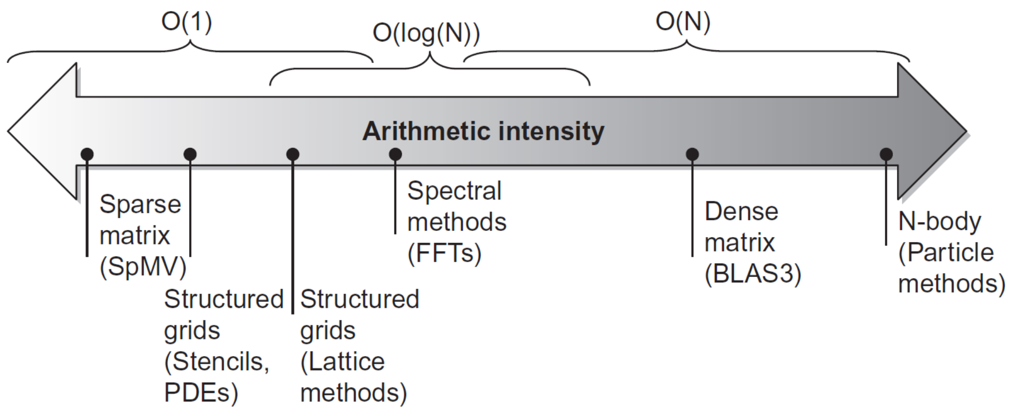
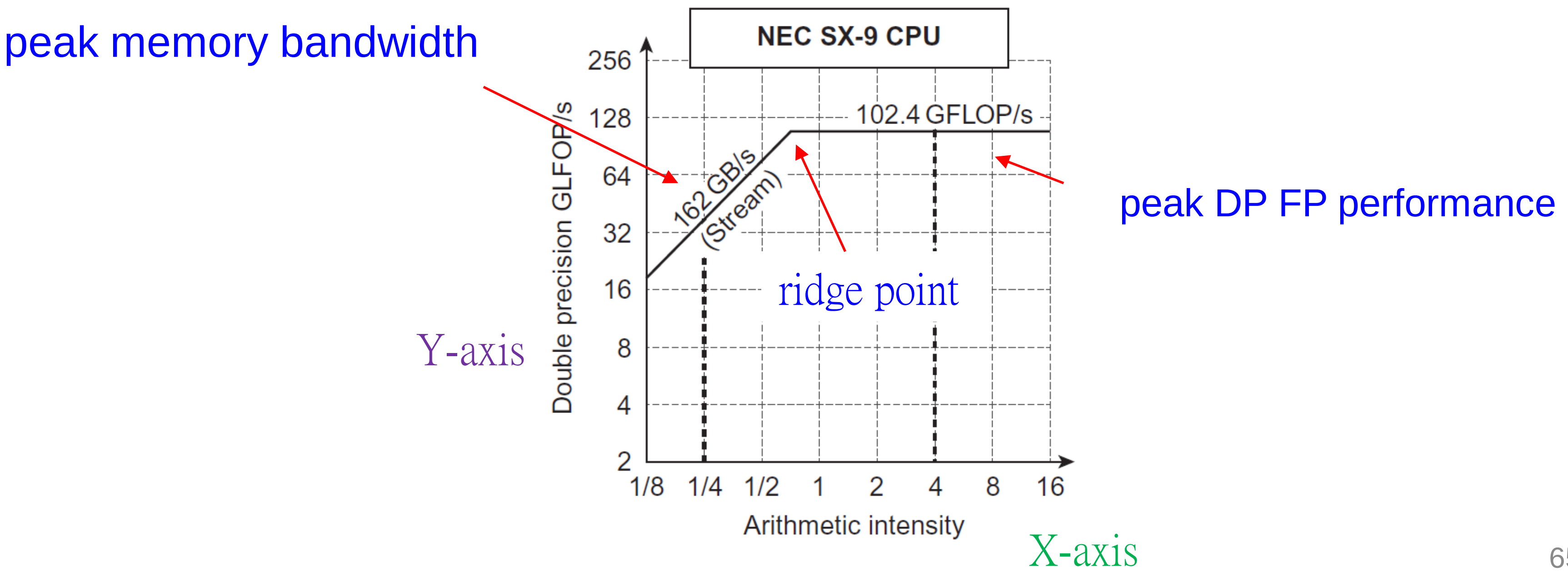


Figure 4.10 Arithmetic intensity, specified as the number of floating-point operations to run the program divided by the number of bytes accessed in main memory (Williams et al., 2009). Some kernels have an arithmetic intensity that scales with problem size, such as a dense matrix, but there are many kernels with arithmetic intensities independent of problem size.

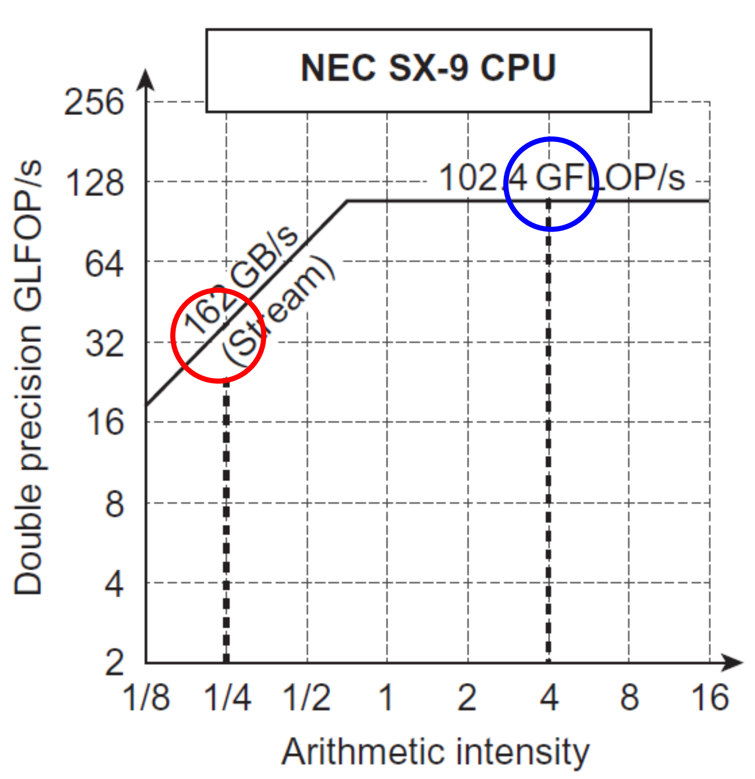
Roofline Model Examples

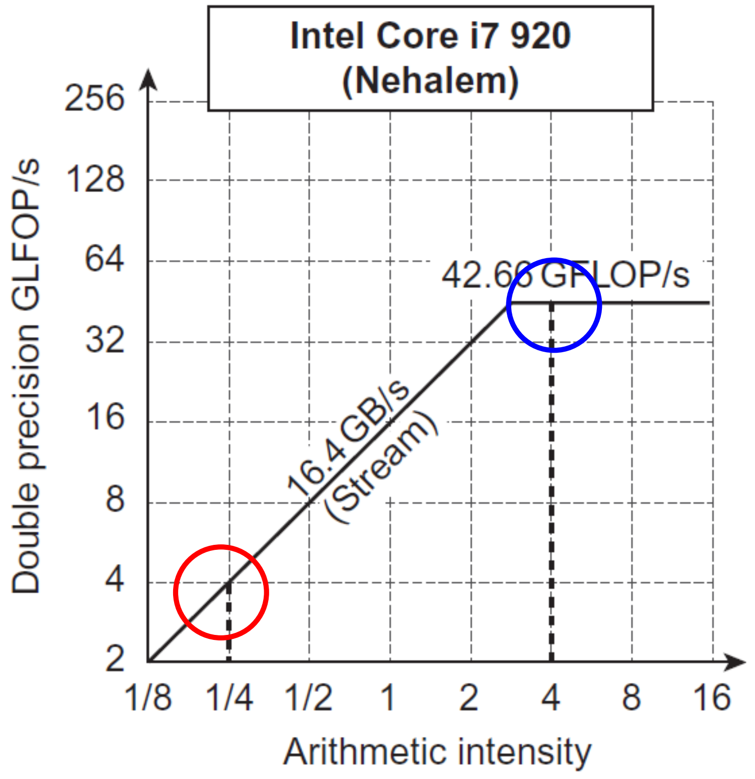
- The “Roofline” sets an upper bound on performance of a kernel depending on its arithmetic intensity.
- Y-axis: attainable fp performance (GFLOPs/sec)

  Attainable GFLOPs/sec = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)
- X-axis: arithmetic intensity (1/8-to 16 FLOP/DRAM byte accessed)



Comparisons on Roofline Models





- The dashed vertical lines at an arithmetic intensity of 4 FLOP/byte: the SX-9 at 102.4 FLOP/s is 2.4× faster than the Core i7 at 42.66 GFLOP/s.
- At an arithmetic intensity of 1/4 FLOP/byte: the SX-9 at 40.5 GFLOP/s is 10× faster than the Core i7 at 4.1 GFLOP/s.

History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output
- 3D graphics processing
  - Originally high-end computers
  - 3D graphics cards for PCs and game consoles
- Graphics Processing Units
  - GPUs are graphics accelerators
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, ray tracing
- This section concentrates on using GPUs for computing.

Graphical Processing Units

- GPUs have virtually every type of parallelism that can be captured by the programming environment:
  - multithreading, MIMD, SIMD, and even instruction-level
- Basic idea of GPUs (NVIDIA):
  - 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* (SIMT)

Programming Model

- Challenges for the GPU programmer for good performance
  - Well schedule the computation on the system and the GPU
  - Efficient transfer of data between system memory and GPU memory
- CUDA design goals: C-like language and programming environment
  - Similar to *OpenCL*
  - 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.

Programming the GPU

- CUDA Programming Model
  - Single Instruction Multiple Thread (SIMT)
- A thread is associated with each data element
- Threads are organized into blocks, called thread block
- Thread blocks are organized into a grid

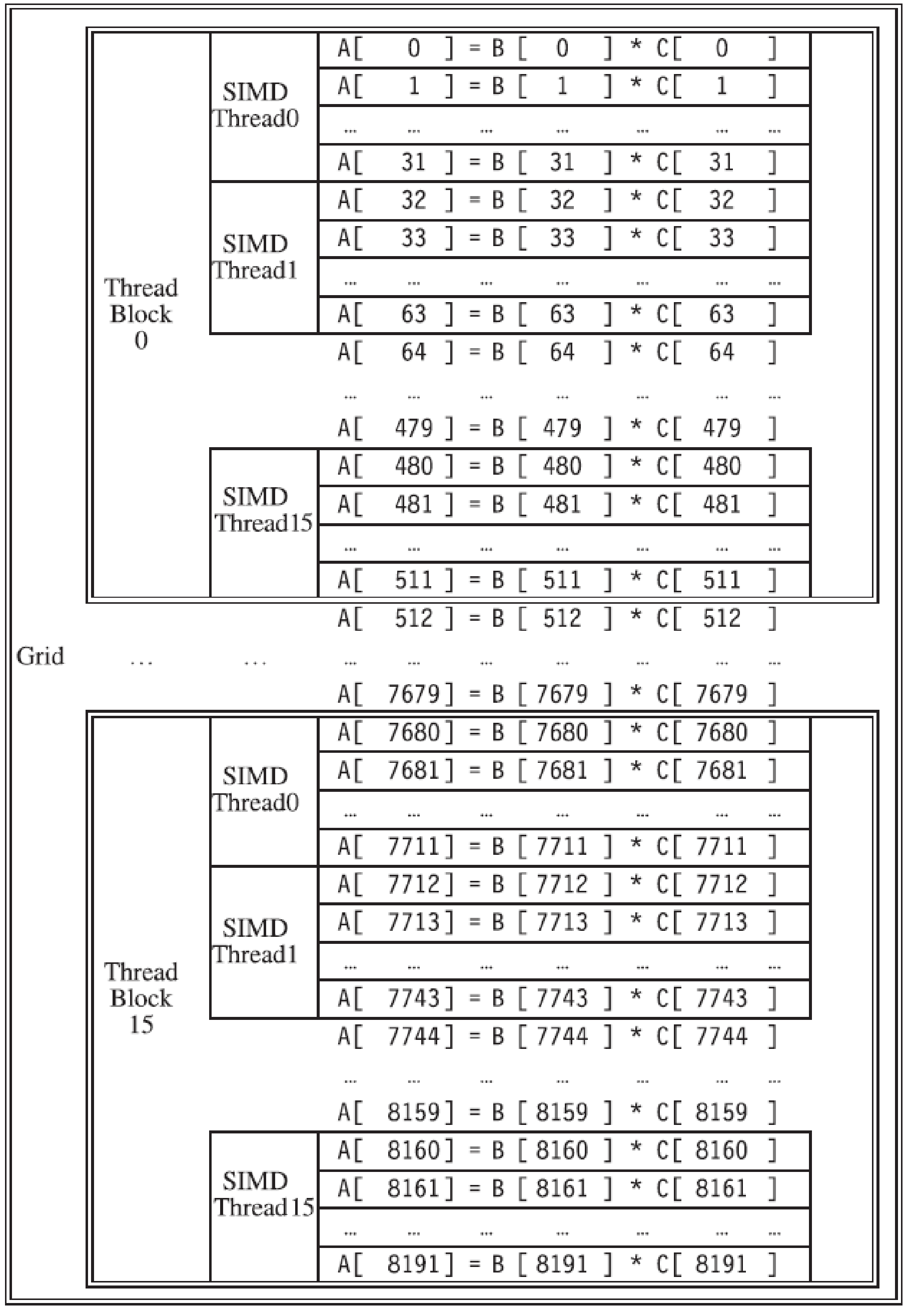
- GPU hardware handles thread management, not applications or OS
  - Given the hardware invested to do graphics well, how can we supplement it to improve performance of a wider range of applications?

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

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*
  - A GPU can have form one to several dozen multithreaded SIMD processors
    - Pascal P100 system has 56



Grid
Thread Block 0
SIMD Thread0
A[0] = B [0] * C[0]
A[1] = B [1] * C[1]
A[31] = B [31] * C[31]
SIMD Thread1
A[32] = B [32] * C[32]
A[33] = B [33] * C[33]
A[63] = B [63] * C[63]
A[64] = B [64] * C[64]
A[479] = B [479] * C[479]
SIMD Thread15
A[480] = B [480] * C[480]
A[481] = B [481] * C[481]
A[511] = B [511] * C[511]
A[512] = B [512] * C[512]
A[7679] = B [7679] * C[7679]
Thread Block 15
SIMD Thread0
A[7680] = B [7680] * C[7680]
A[7681] = B [7681] * C[7681]
A[7711] = B [7711] * C[7711]
SIMD Thread1
A[7712] = B [7712] * C[7712]
A[7713] = B [7713] * C[7713]
A[7743] = B [7743] * C[7743]
A[7744] = B [7744] * C[7744]
A[8159] = B [8159] * C[8159]
SIMD Thread15
A[8160] = B [8160] * C[8160]
A[8161] = B [8161] * C[8161]
A[8191] = B [8191] * C[8191]

Figure 4.13 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector-vector multiply, with each vector being 8192 elements long. Each thread of SIMD instruc-

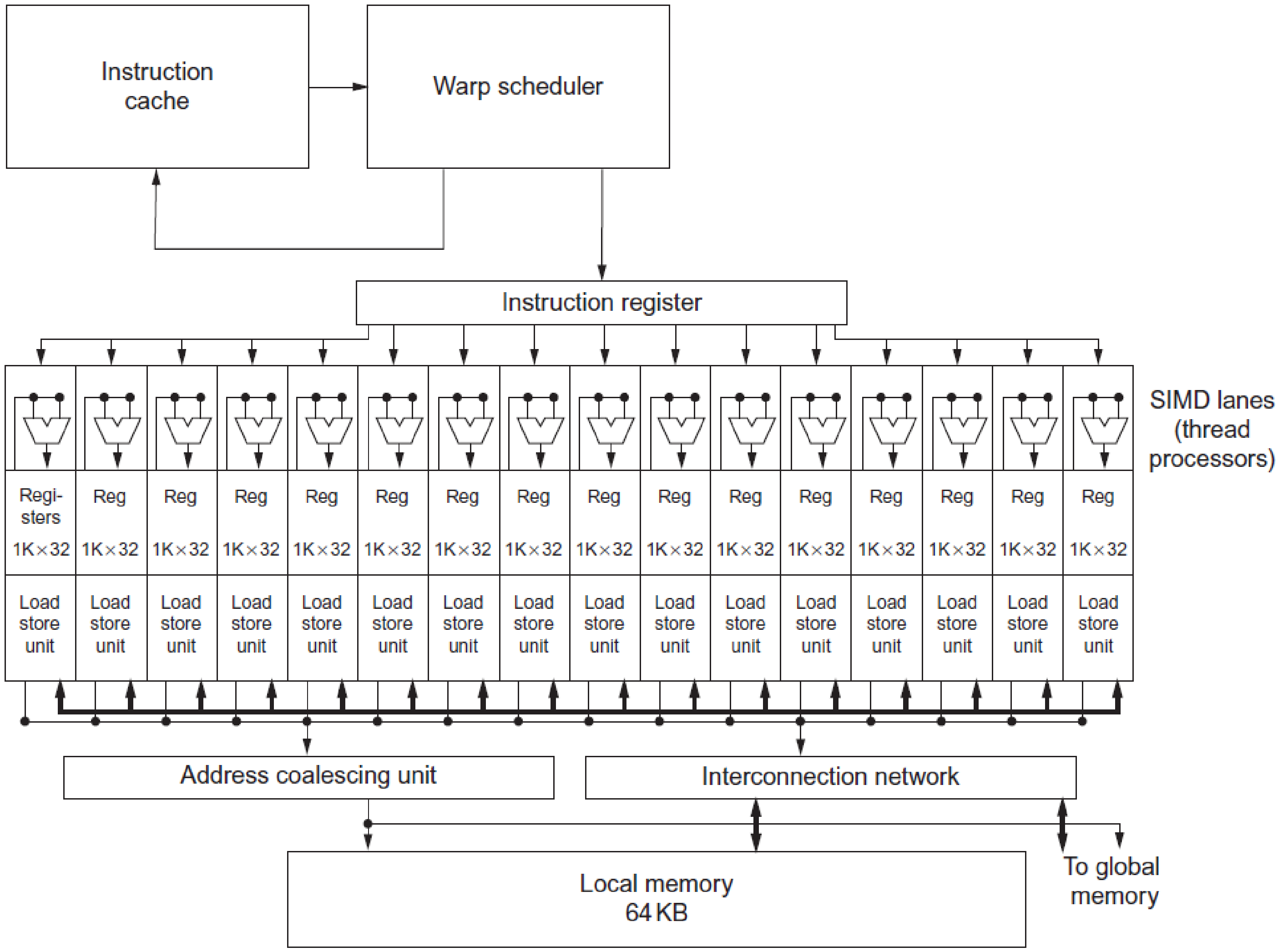


Figure 4.14 Simplified block diagram of a multithreaded SIMD Processor. It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

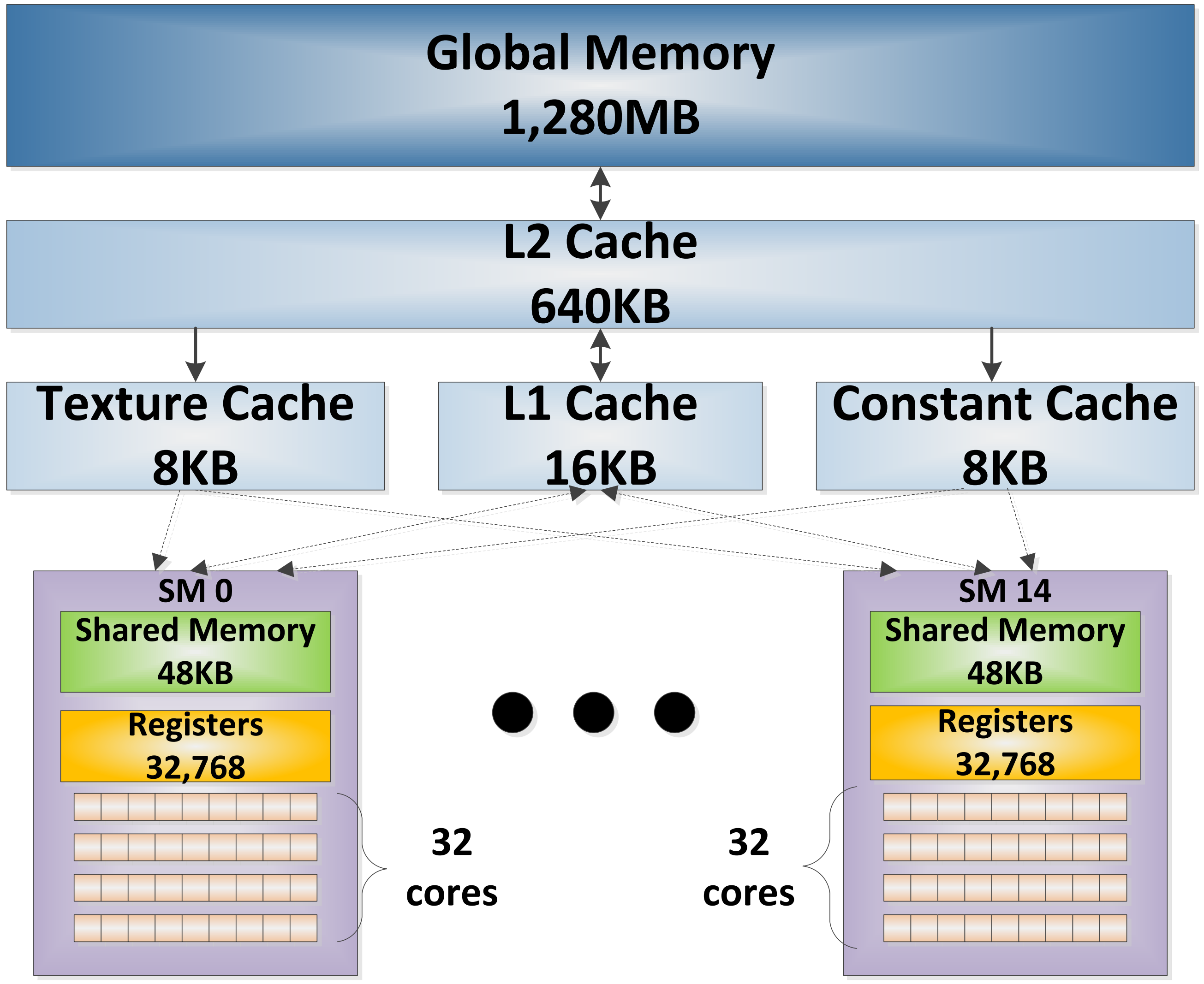
Terminology

- 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

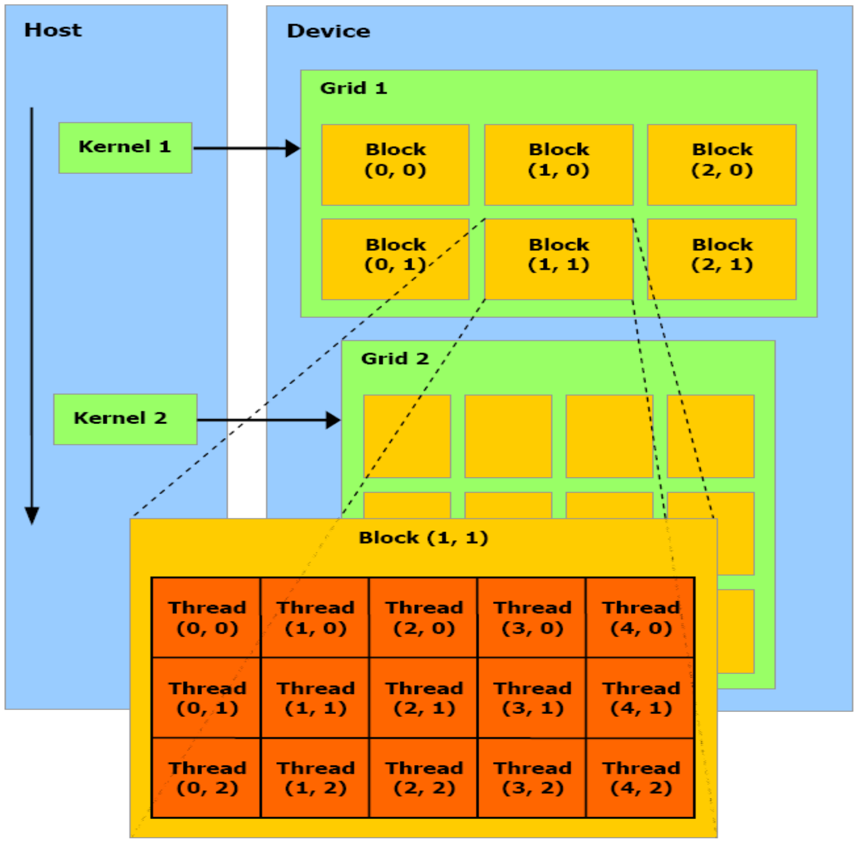
- NVIDIA GPU has 32,768 registers
  - Divided into lanes
  - Each SIMD thread is limited to 64 registers
  - SIMD thread has up to:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
  - Fermi has 16 physical SIMD lanes, each containing 2048 registers

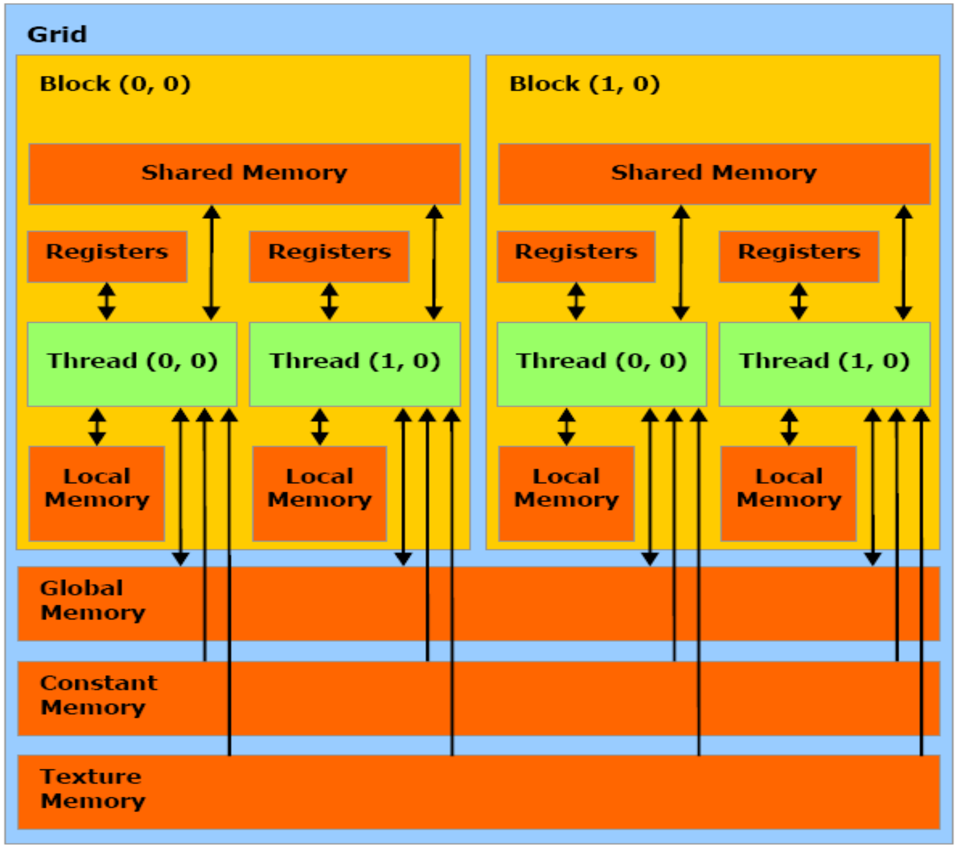
GTX570 GPU



Up to 1536
Threads/SM

GPU Threads in SM (GTX570)





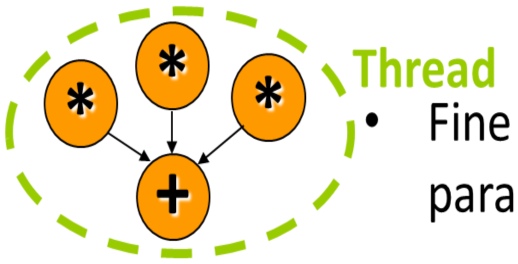
- 32 threads within a block work collectively
  - ✓ Memory access optimization, latency hiding

Matrix Multiplication

$$\underbrace{\begin{bmatrix} C_0 & C_1 & C_2 \\ C_3 & C_4 & C_5 \\ C_6 & C_7 & C_8 \end{bmatrix}}_{\text{Matrix C}} = \underbrace{\begin{bmatrix} A_0 & A_1 & A_2 \\ A_3 & A_4 & A_5 \\ A_6 & A_7 & A_8 \end{bmatrix}}_{\text{Matrix A}} \times \underbrace{\begin{bmatrix} B_0 & B_1 & B_2 \\ B_3 & B_4 & B_5 \\ B_6 & B_7 & B_8 \end{bmatrix}}_{\text{Matrix B}}$$

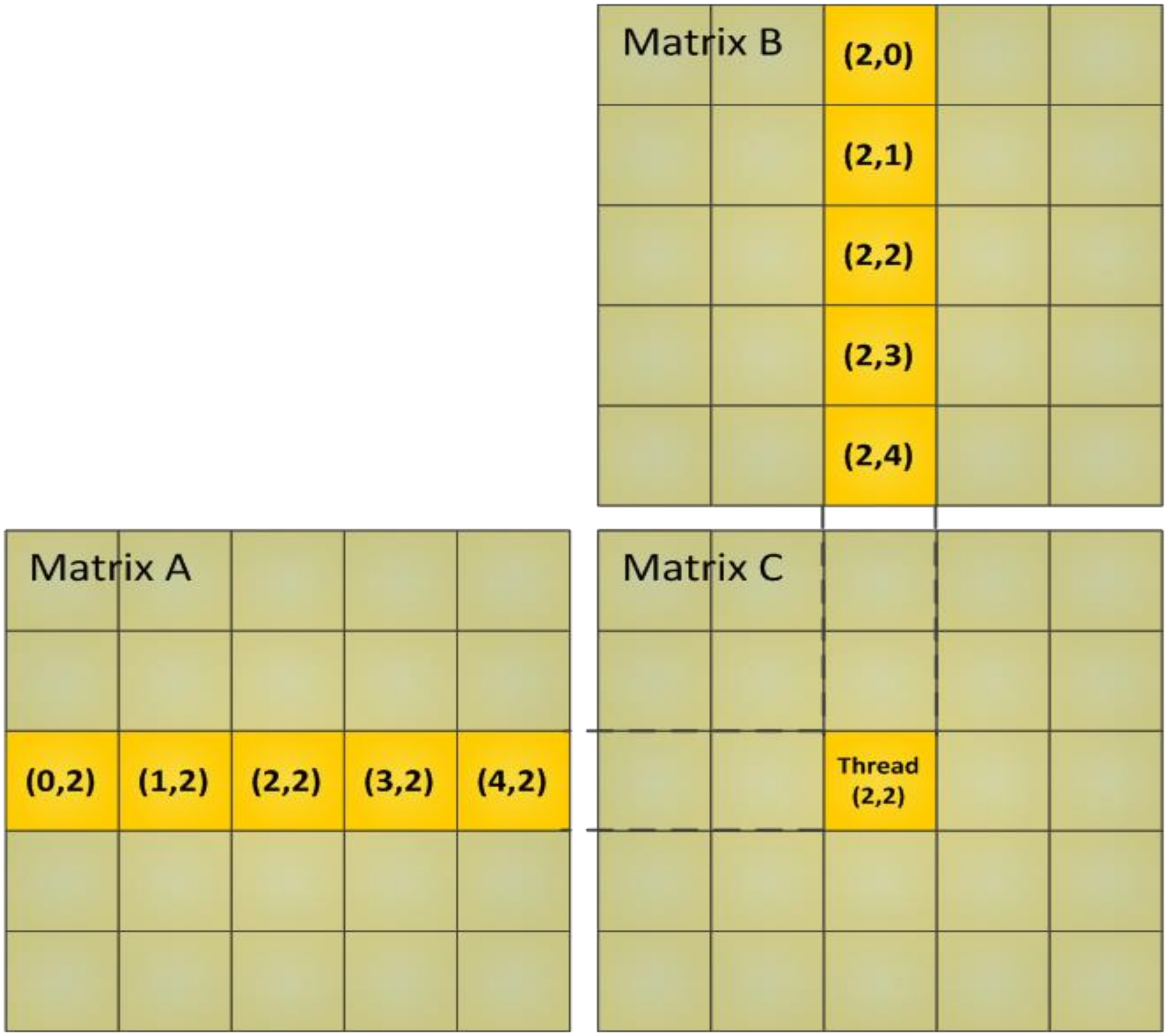
	Thread
$C_0 = (A_0 * B_0) + (A_1 * B_3) + (A_2 * B_6)$	1
$C_1 = (A_0 * B_1) + (A_1 * B_4) + (A_2 * B_7)$	2
$C_2 = (A_0 * B_2) + (A_1 * B_5) + (A_2 * B_8)$	3
$C_3 = (A_3 * B_0) + (A_4 * B_3) + (A_5 * B_6)$	4
$C_4 = (A_3 * B_1) + (A_4 * B_4) + (A_5 * B_7)$	5
$C_5 = (A_3 * B_2) + (A_4 * B_5) + (A_5 * B_8)$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

	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



- 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
```

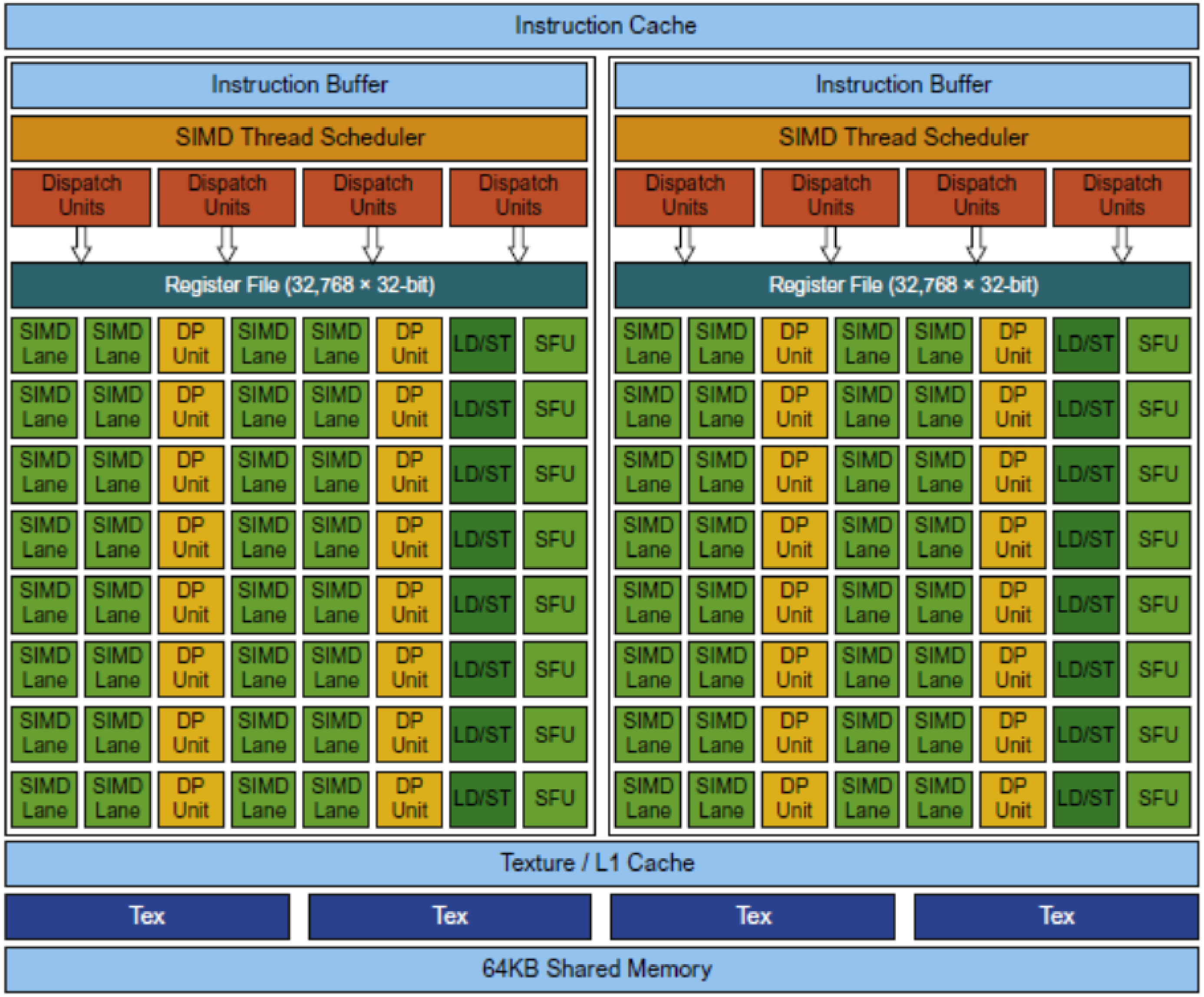
NVIDIA GPU Memory Structures

- 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

Pascal Architecture Innovations

- Each SIMD processor has
  - Two or four SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Two threads of SIMD instructions are scheduled every two clock cycles
- Fast single-, double-, and half-precision
- High Bandwith Memory 2 (HBM2) at 732 GB/s
- NVLink between multiple GPUs (20 GB/s in each direction)
- Unified virtual memory and paging support

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

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

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

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 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