Προηγμένη Αρχιτεκτονική Υπολογιστών

Στατικός Παραλληλισμός: SIMD – Vector – GPU

Νεκτάριος Κοζύρης & Διονύσης Πνευματικάτος
{nkoziris,pnevmati}@cslab.ece.ntua.gr

8ο εξάμηνο ΣΗΜΜΥ – Ακαδημαϊκό Έτος: 2019-20

http://www.cslab.ece.ntua.gr/courses/advcomparch/

Flynn's Taxonomy

- Single Instruction stream, Single Data stream: SISD
- Single Instruction/Multiple Data streams: SIMD
  - Vector architectures
  - Multimedia extensions
  - Graphics processor units
- Multiple Instruction/Single Data stream: MISD
  - No commercial implementation
- Multiple Instruction/Multiple Data streams: MIMD
  - Tightly-coupled MIMD
  - Loosely-coupled MIMD

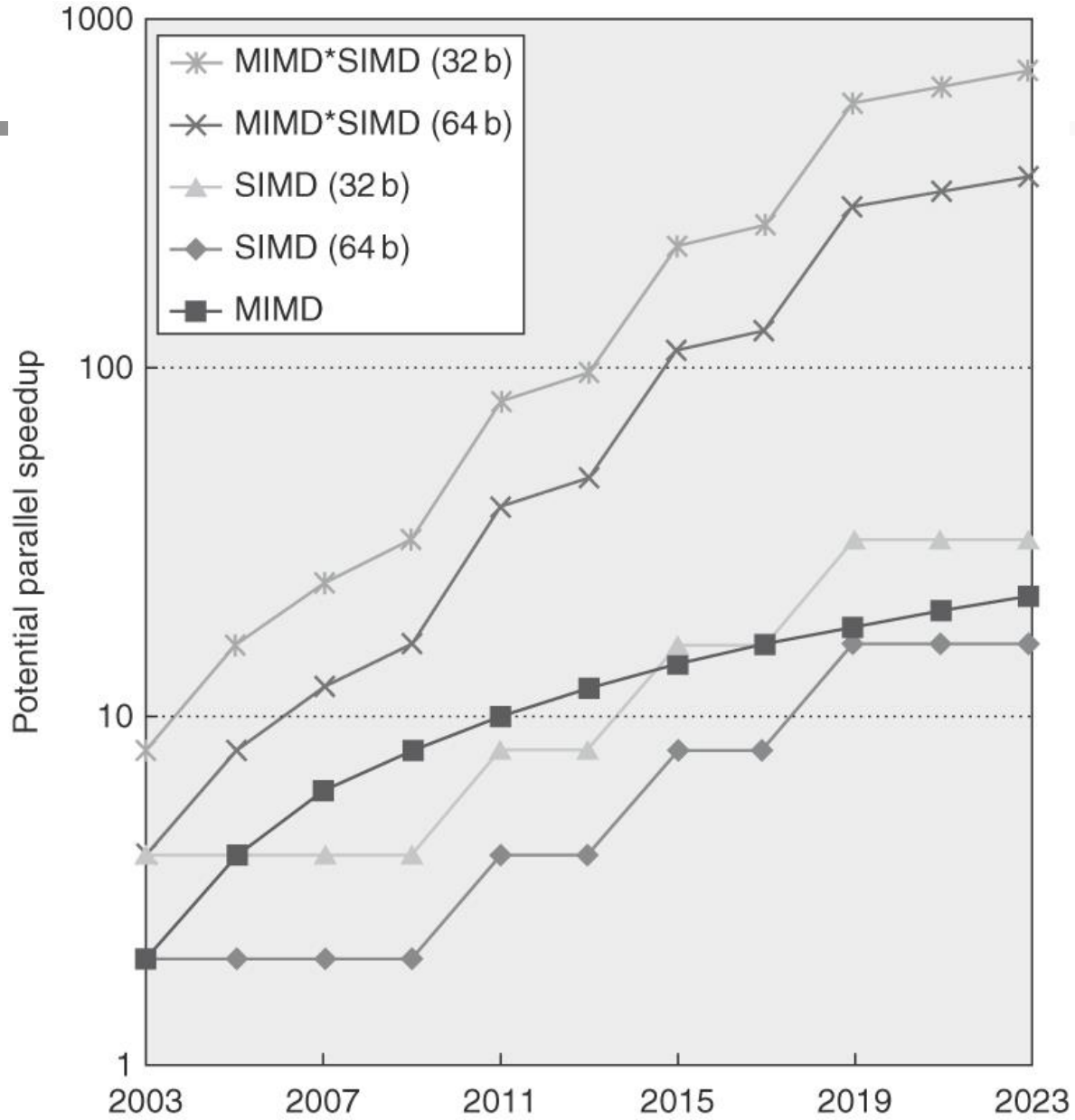
Introduction

- 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 Parallelism

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

- 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!



Potential speedup via parallelism from MIMD, SIMD, and both MIMD and SIMD: ***Assumes*** that two cores per chip for MIMD are added every two years and the number of operations for SIMD will double every four years.

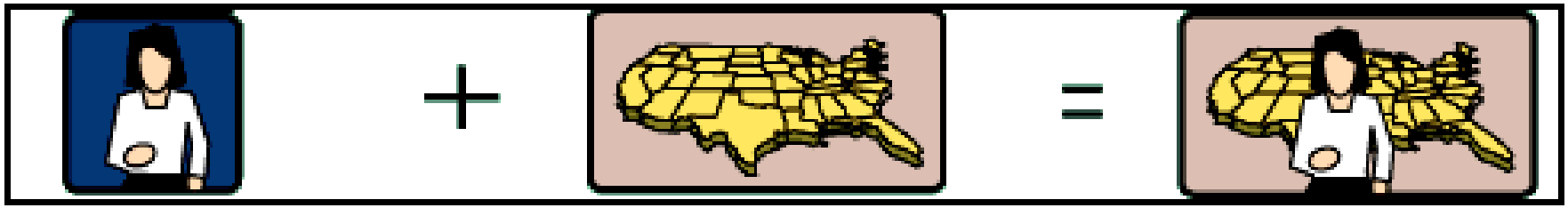
SIMD

- SIMD = Single Instruction, Multiple Data
- (a) sub-word parallelism (RBG pixel = 3 bytes)
  - Assume wide, multi-item registers
  - `add rd, rs, rt` produces multiple results!
  - Fewer instructions, fewer branches!
  - Hard for compiler to use, hand code libraries mostly!
  - Intel MMX, SSD{1-4}, AVX,…
- (b) wide registers: Vector Processing
- Other options:
  - Typical pipelines: SISD
  - Parallel programs: MIMD

Blue/Green Screen Effects

They used to use blue screen back in the day😉 starwarsfacts

MMX example: blue screen image merging

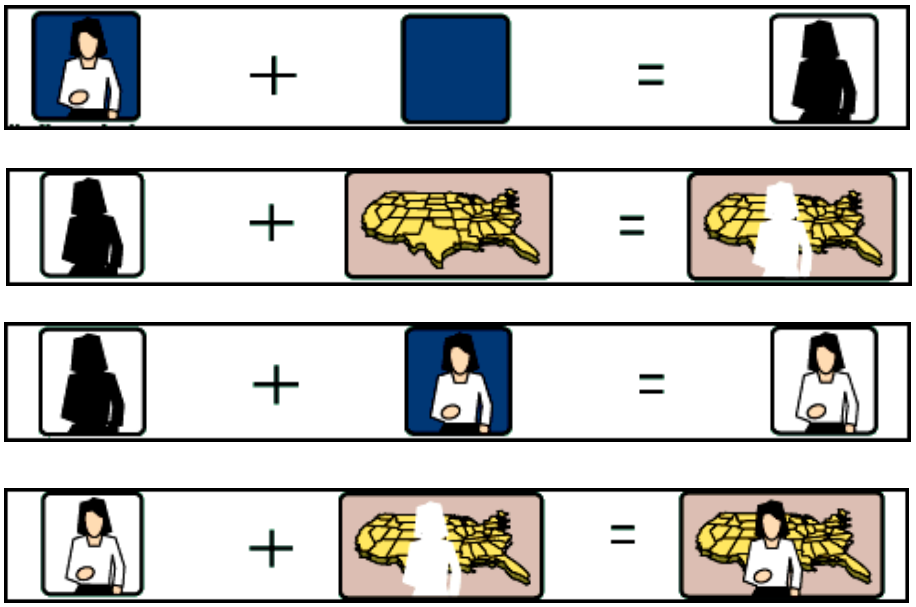


- Array X (blue screen), Y (background), 16-bit pixels
- Result in X
- X86 loop:

```
CMP X[i], BLUE     ;Check if blue
JNE  next_pixel    ;If not, skip ahead
MOV X[i], Y[i]     ;If blue, use second image
+ loop control
```

MMX example: blue screen image merging

```
MOV MM1, X[i]      ;Make a copy of X[i]
;Check X, make mask
PCMPEQW MM1, BLUE
;clear non-blue Y pixels
PAND Y[i], MM1
;Zero out blue pixels in X
PANDN MM1, X[i]
;Combine two images
POR MM1, Y[i]
```



- Process 4 pixels per instruction
  - MMX = 4 results per loop iterations (5 instructions)
  - Eliminate test branch!
  - Speed: >= 2.5x

SIMD Extensions (short vectors)

- Media applications operate on data types narrower than the native word size
  - Example: disconnect carry chains to “partition” adder
- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers

SIMD Implementations

- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
  - Advanced Vector Extensions (2010)
    - Four 64-bit integer/fp ops
  - Operands must be consecutive and aligned memory locations
  - Generally designed to accelerate carefully written libraries rather than for compilers
- Advantages over vector architecture:
  - Cost little to add to the standard ALU and easy to implement
  - Require little extra state → easy for context-switch
  - Require little extra memory bandwidth
  - No virtual memory problem of cross-page access and page-fault

Example SIMD Code

- Example DXPY:

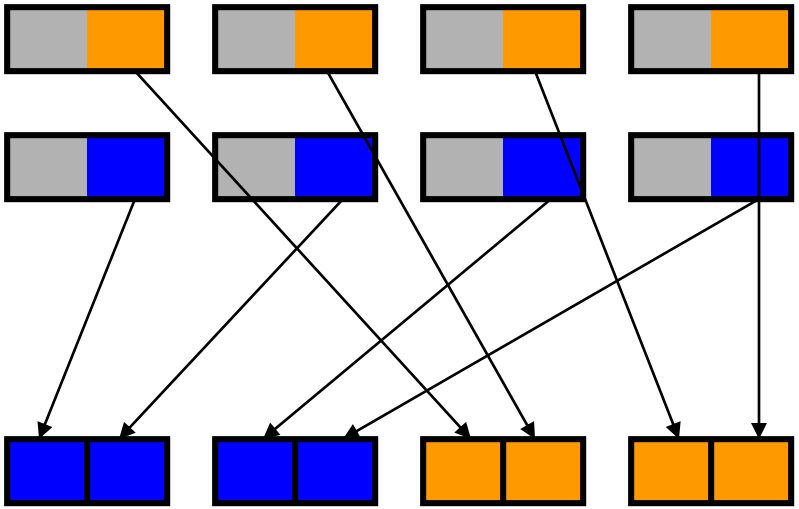
```
        L.D     F0,a        ;load scalar a
        MOV     F1, F0      ;copy a into F1 for SIMD MUL
        MOV     F2, F0      ;copy a into F2 for SIMD MUL
        MOV     F3, F0      ;copy a into F3 for SIMD MUL
        DADDIU  R4,Rx,#512  ;last address to load
Loop:
        L.4D    F4,0[Rx]    ;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    F8,0[Ry]    ;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
```

Summary of SIMD Operations (1)

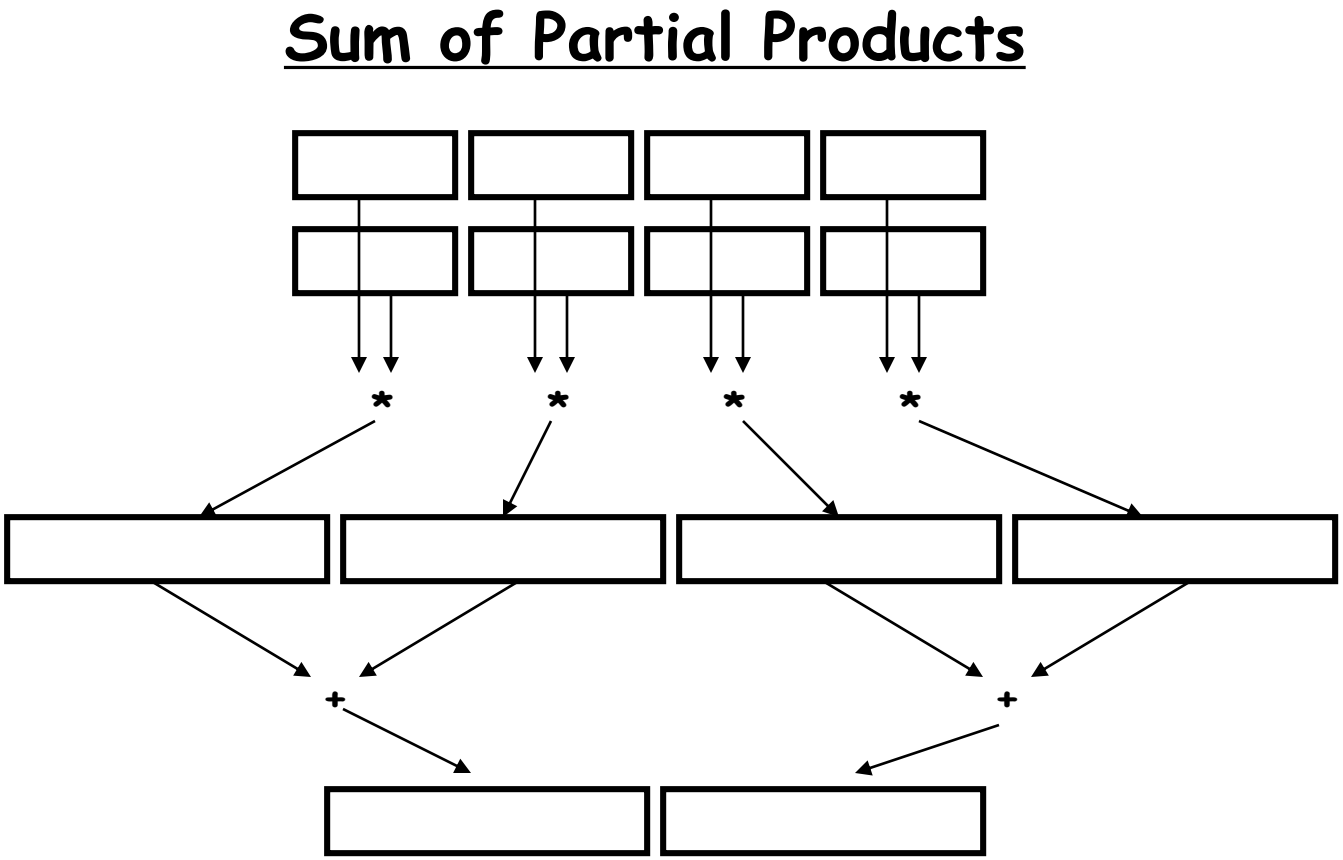
- Integer arithmetic
  - Addition and subtraction with saturation
  - Fixed-point rounding modes for multiply and shift
  - Sum of absolute differences
  - Multiply-add, multiplication with reduction
  - Min, max
- Floating-point arithmetic
  - Packed floating-point operations
  - Square root, reciprocal
  - Exception masks
- Data communication
  - Merge, insert, extract
  - Pack, unpack (width conversion)
  - Permute, shuffle

Example of SIMD Operation (1)

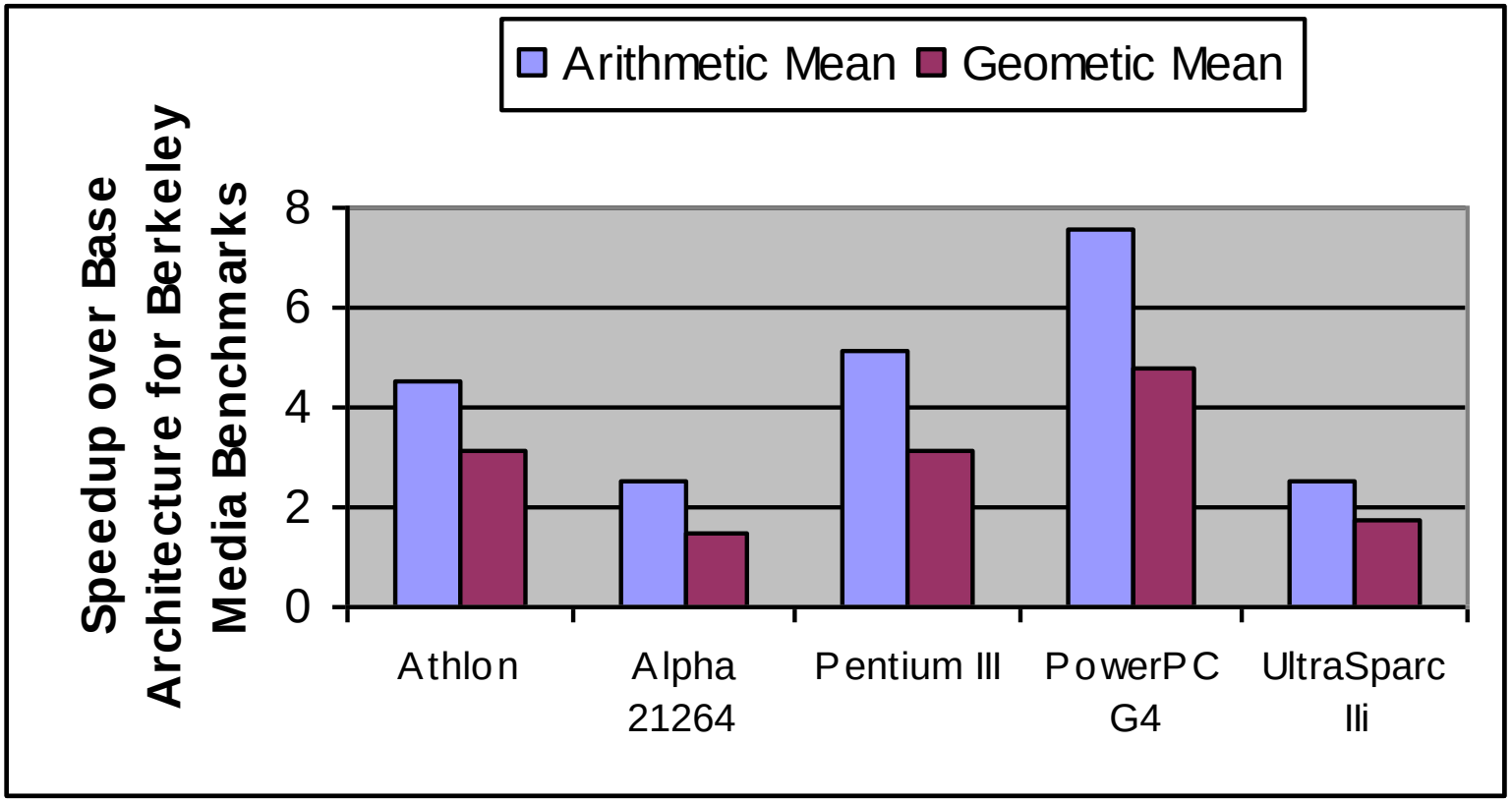
Pack (Int16->Int8)



Example of SIMD Operation (2)



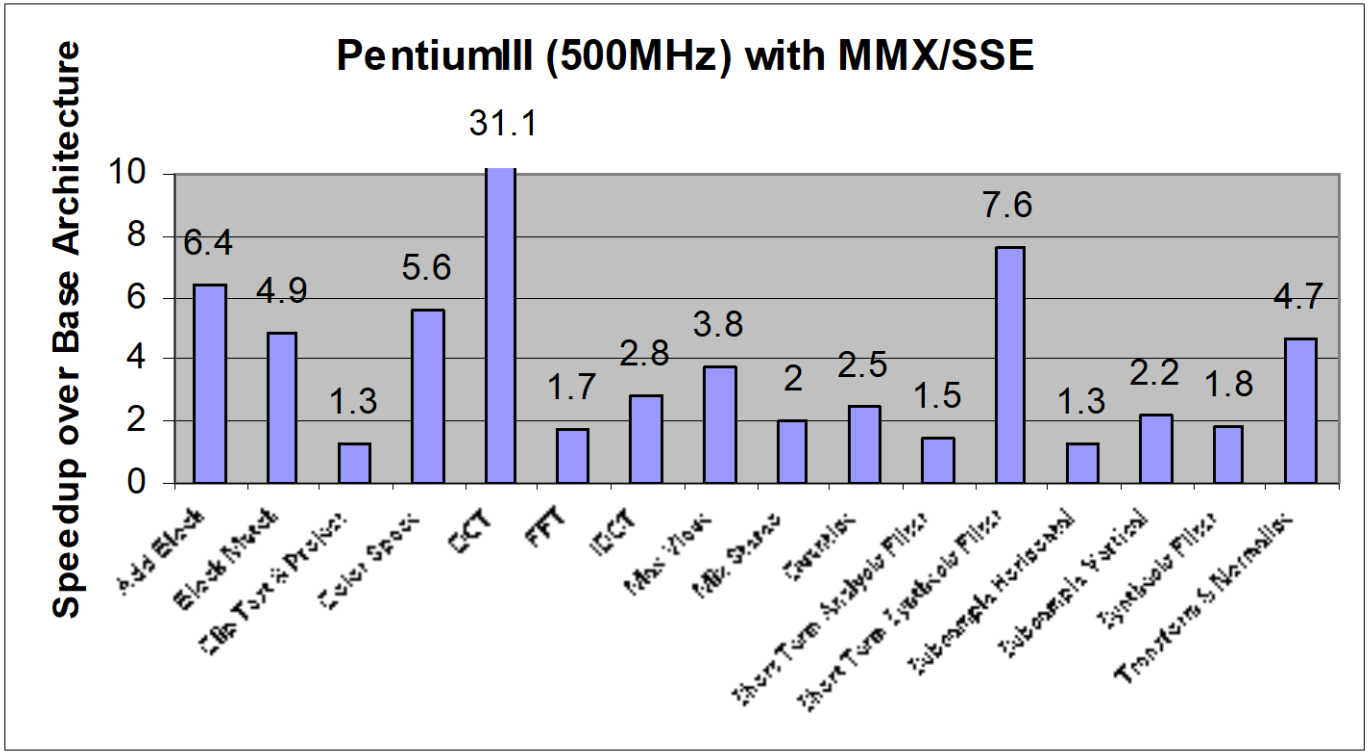
SIMD Performance



Limitations

- Memory bandwidth
- Overhead of handling alignment and data width adjustments

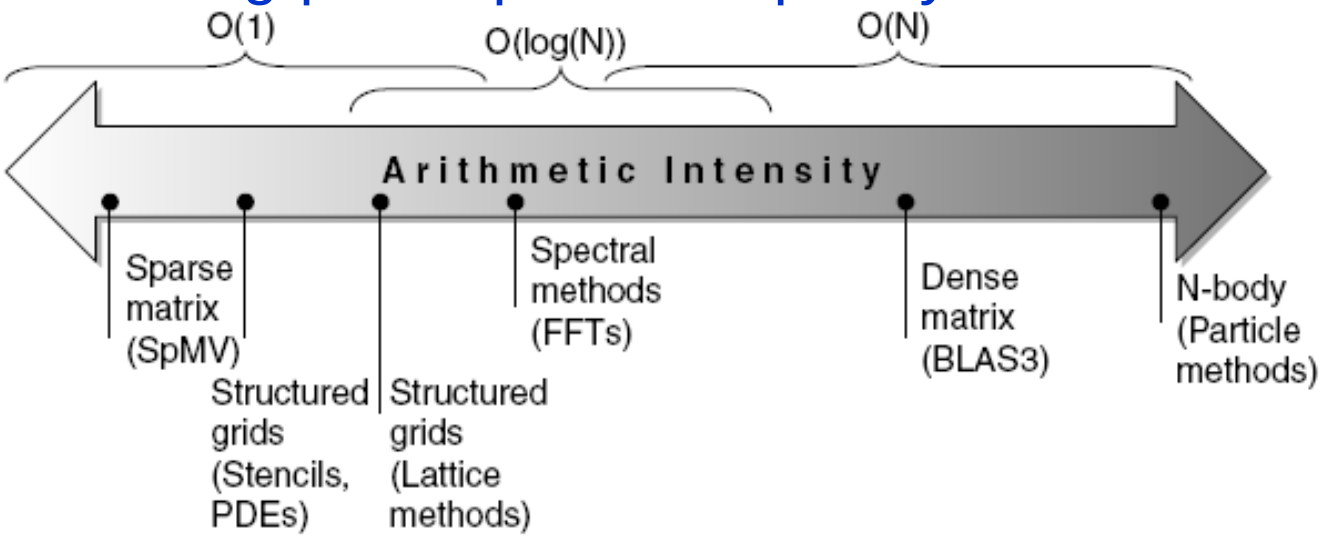
A Closer Look at MMX/SSE



- Higher speedup for kernels with narrow data where 128b SSE instructions can be used
- Lower speedup for those with irregular or strided accesses

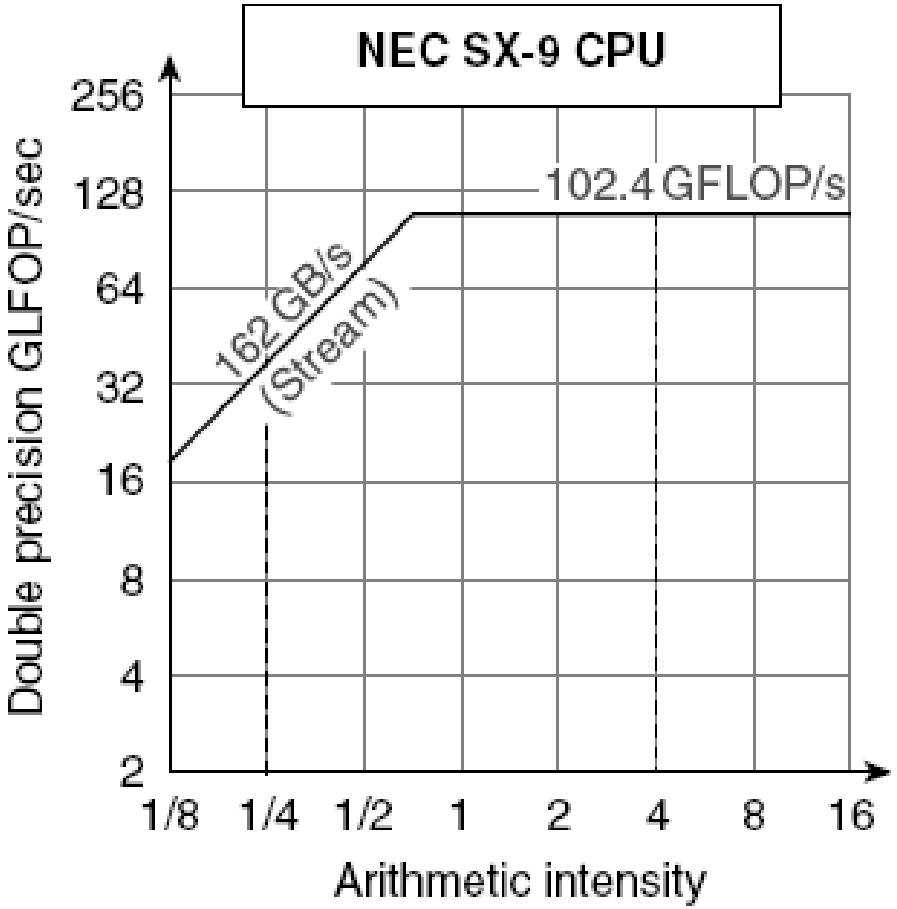
Roofline Performance Model

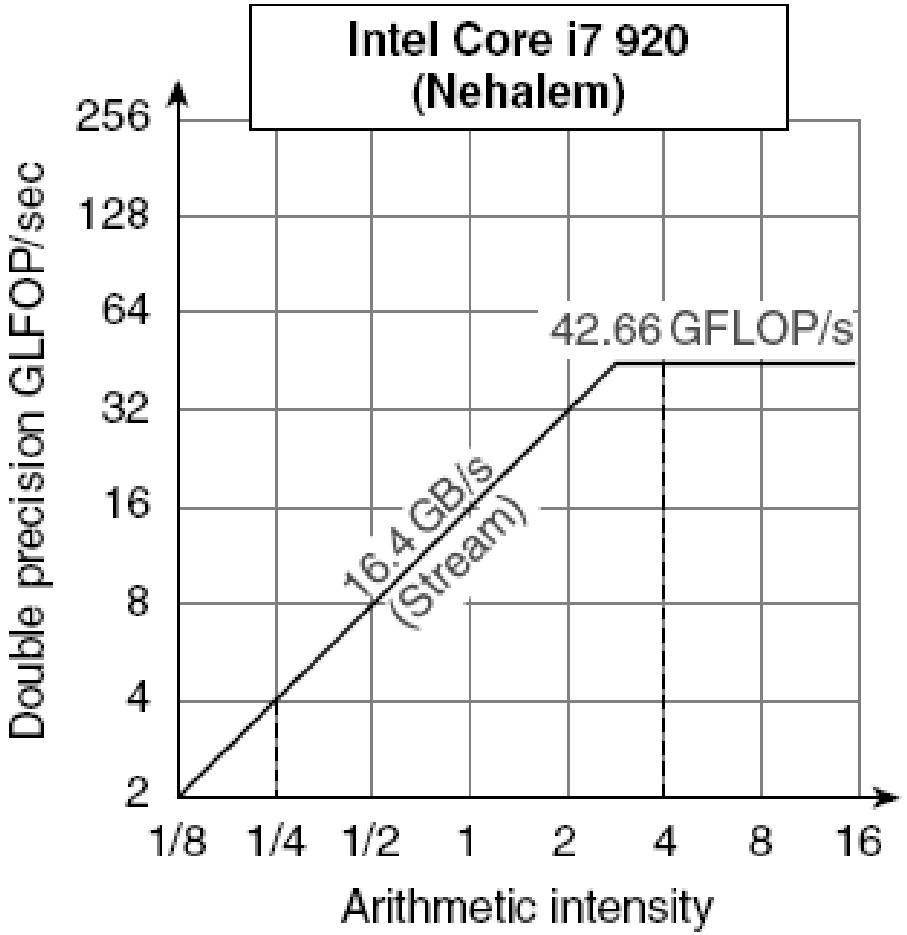
- Basic idea:
  - Plot peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
  - Floating-point operations per byte read



Examples (roof-line plots)

- Attainable GFLOPs/sec Min = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)





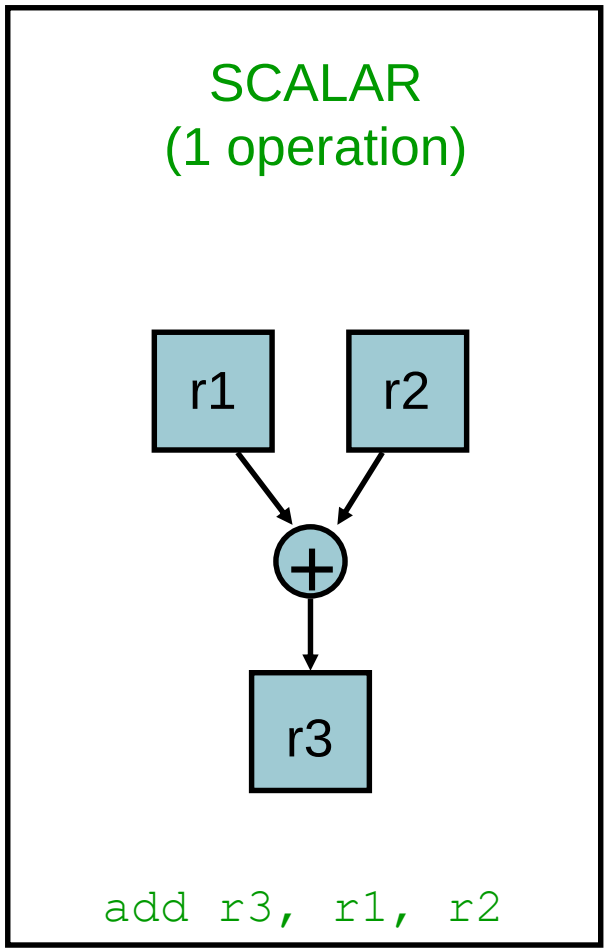
Vector Architectures

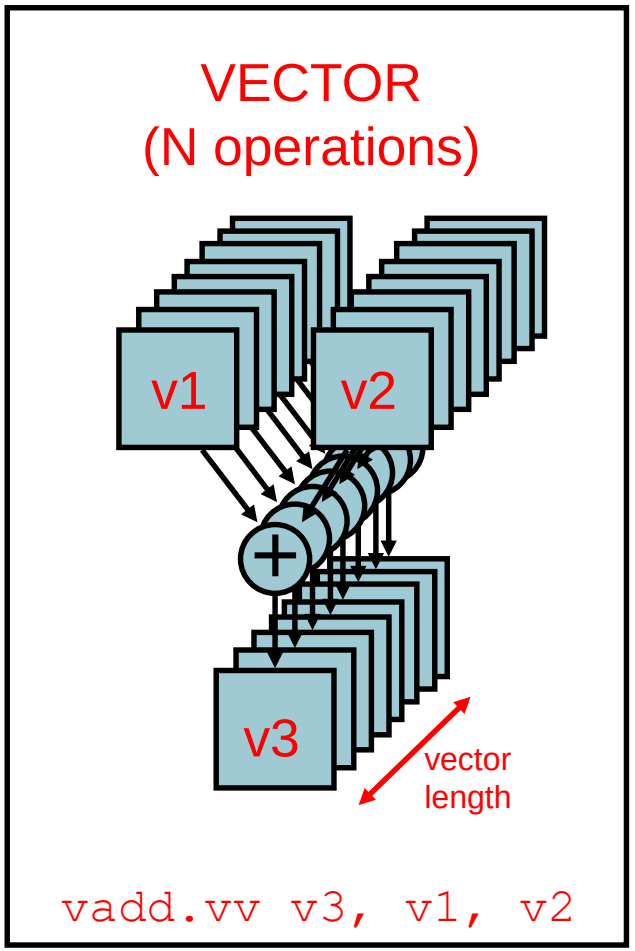
- Basic idea:
  - Read sets of data elements into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory

- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

Vector Instructions

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





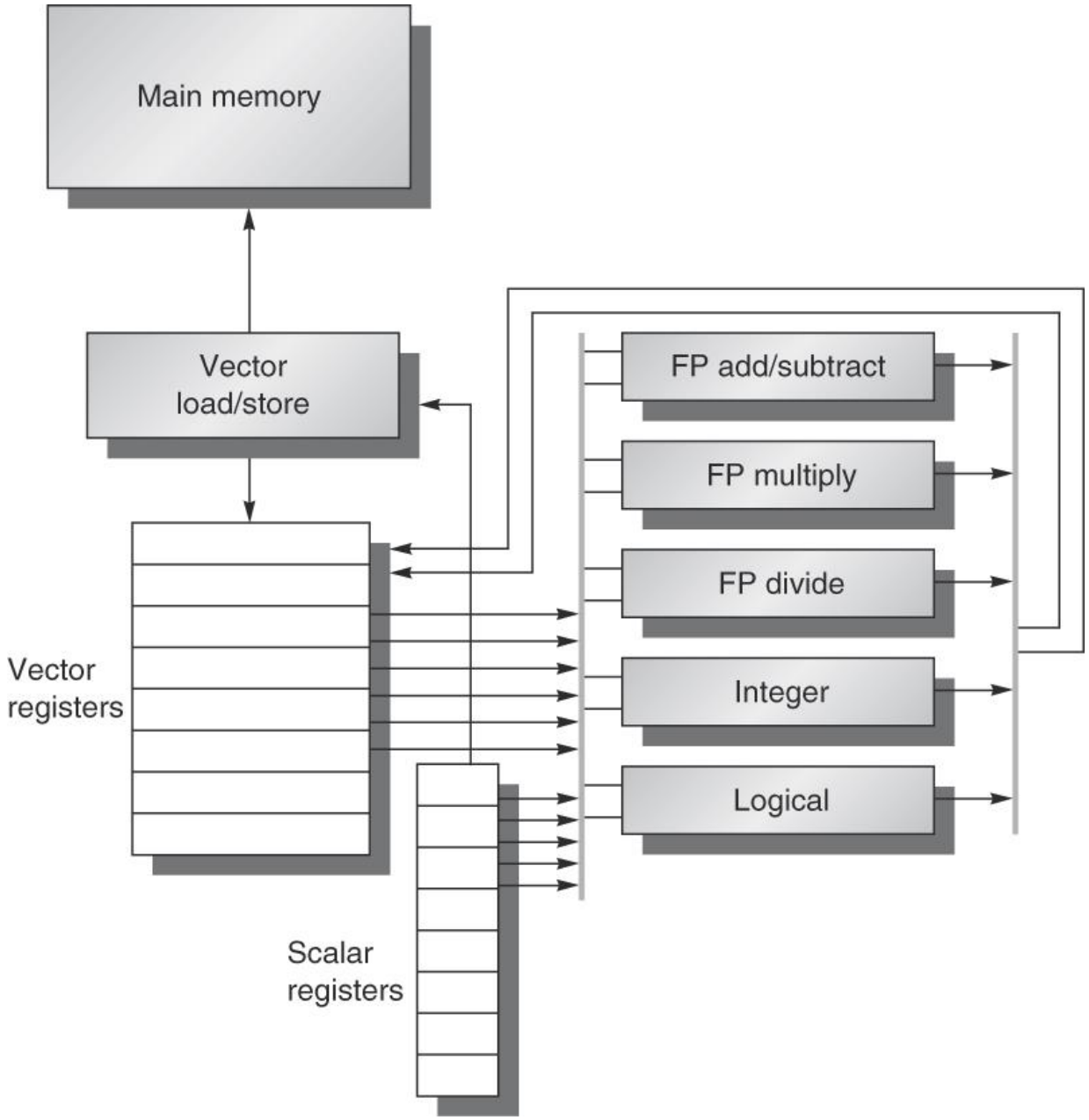
Properties of Vector Architectures

- Single vector instruction implies lots of work (loop)
  - Fewer instruction fetches
- Each result independent of previous result
  - Compiler ensures no dependencies
  - Multiple operations can be executed in parallel
  - Simpler design, high clock rate
- Reduces branches and branch problems in pipelines
- Vector instructions access memory with known pattern
  - Effective prefetching
  - Amortize memory latency of over large number of elements
  - Can exploit a high bandwidth memory system
  - No (data) caches required!

VMIPS

- Example architecture: 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
    - One word per clock cycle after initial latency
  - Scalar registers
    - 32 general-purpose registers
    - 32 floating-point registers

VMIPS vector architecture



DAXPY in MIPS Instructions

Example: DAXPY (double precision a*X+Y)

```
        L.D     F0,a        ; load scalar a
        DADDIU  R4,Rx,#512  ; last address to load
Loop:   L.D     F2,0(Rx)    ; load X[i]
        MUL.D   F2,F2,F0    ; a x X[i]
        L.D     F4,0(Ry)    ; load Y[i]
        ADD.D   F4,F2,F2    ; a x X[i] + Y[i]
        S.D     F4,9(Ry)    ; store into Y[i]
        DADDIU  Rx,Rx,#8    ; increment index to X
        DADDIU  Ry,Ry,#8    ; increment index to Y
        SUBBU   R20,R4,Rx   ; compute bound
        BNEZ    R20,Loop    ; check if done
```

- Requires almost 600 MIPS ops

VMIPS Instructions

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

- Example: DAXPY (double precision a*X+Y)

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

- Requires 6 instructions

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

Chimes

- Sequences with read-after-write dependency hazards can be in the same convoy via *chaining*
- *Chaining*
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
- *Chime*
  - Unit of time to execute one convey
  - *m* conveys executes in *m* chimes
  - For vector length of *n*, requires *m* x *n* clock cycles

Vector Processor Optimizations: Chaining

- Dependencies

```
vmul.vv   V1,V2,V3
vadd.vv   V4,V1,V5     # RAW hazard
```

- *Chaining:* Overlapping dependent vector operations
  - Vector register (V1) is not as a single entity but as a group of individual registers
  - Pipeline forwarding can work on individual vector elements
- Flexible chaining: allow vector to chain to any other active vector operation => more read/write ports

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

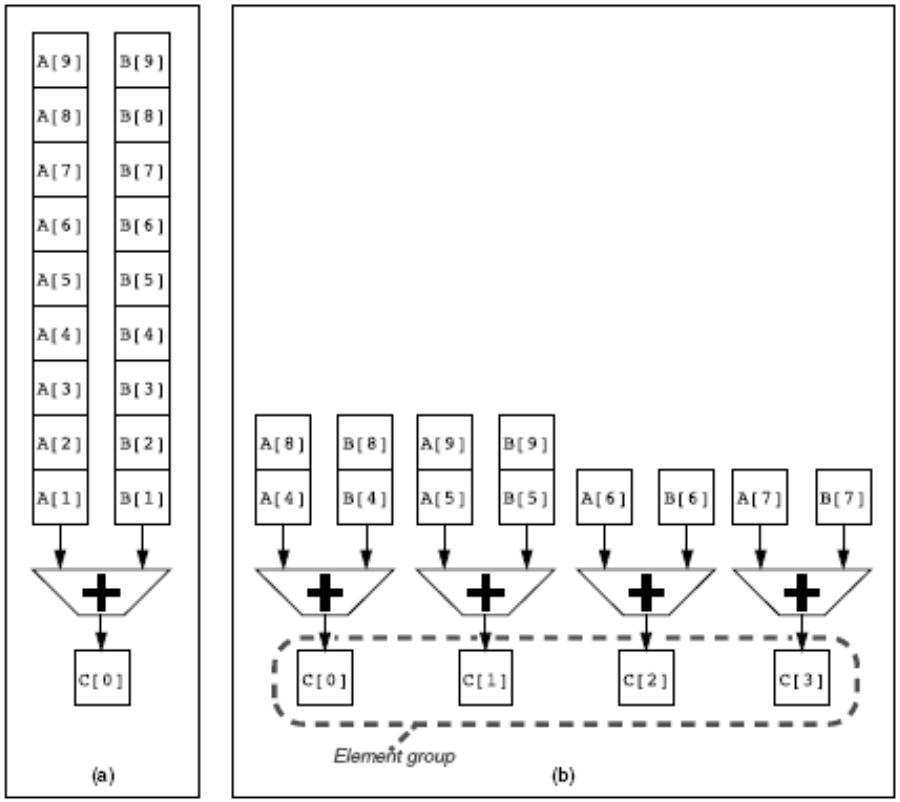
For 64 element vectors, requires 64 x 3 = 192 clock cycles

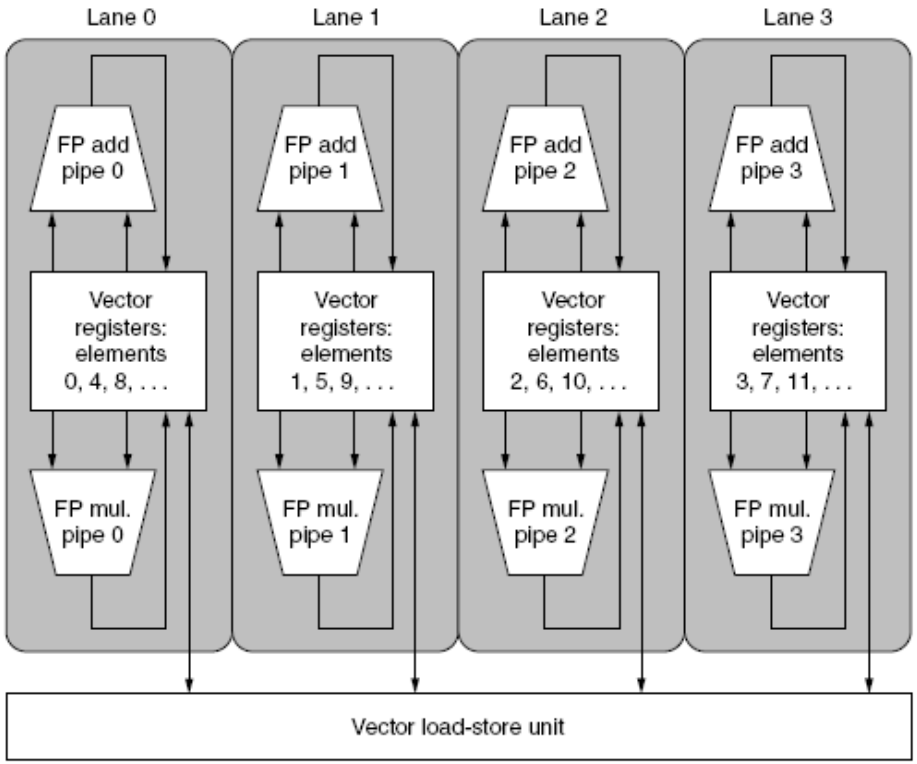
Challenges

- Start up time
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add => 6 clock cycles
    - Floating-point multiply => 7 clock cycles
    - Floating-point divide => 20 clock cycles
    - Vector load => 12 clock cycles
- Optimizations:
  - *Multiple Lanes*: > 1 element per clock cycle
  - *Vector Length Registers*: Non-64 wide vectors
  - *Vector Mask Registers*: IF statements in vector code
  - *Memory Banks*: Memory system optimizations to support vector processors
  - *Stride*: Multiple dimensional matrices
  - *Scatter-Gather*: Sparse matrices
  - *Programming Vector Architectures*: Program structures affecting performance

Multiple Lanes

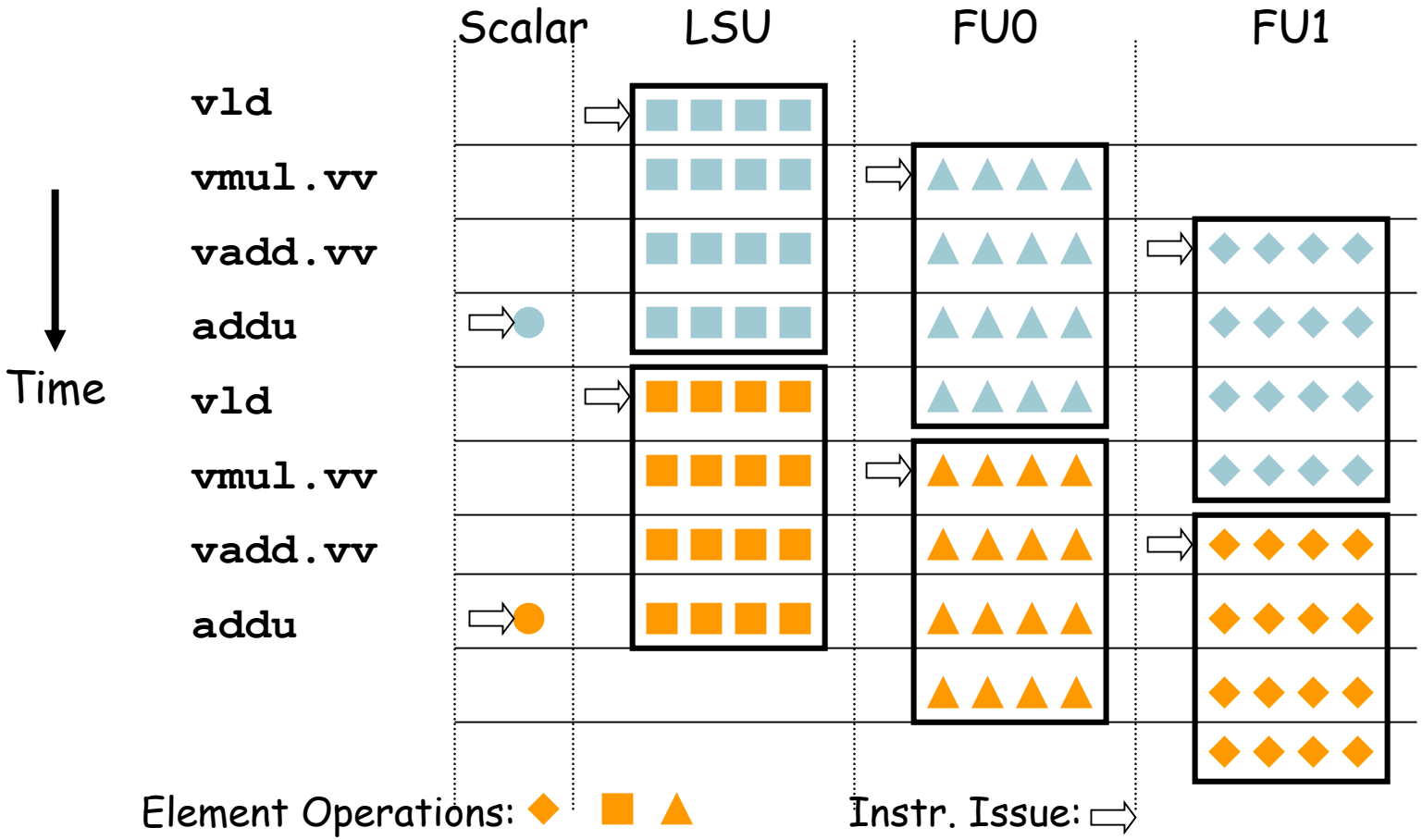
- Element n of vector register A is “hardwired” to element n of vector register B
  - Allows for multiple hardware lanes





Vector Processors: Chaining & Multi-lane Example

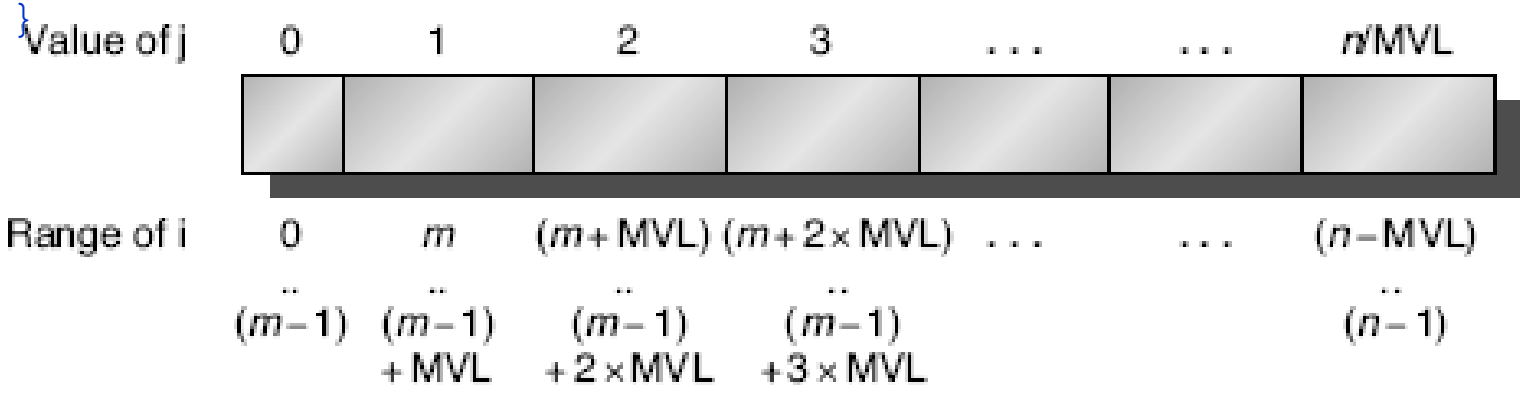
- VL=16, 4 lanes, 2 FUs, 1 LSU, chaining -> 12 ops/cycle
- Just one new instruction issued per cycle!!!!



Vector Length Registers

- Vector length not known at compile time?
- Use Vector Length Register (VLR)
- Use strip mining for vectors over the maximum length:

```
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*/
}
```



Vector Mask Registers

- 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 (*if conversion*):

```
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!

Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing the same memory
- Example:
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time: 2.167 ns, SRAM cycle time: 15 ns
  - How many memory banks needed?
    - 32x6=192 accesses,
    - 15/2.167≈7 processor cycles
    - →1344!

Stride

- 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 multiplication of rows of B with columns of D
- Use non-unit stride
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM(stride, #banks) < bank busy time (in # of cycles)

Stride

- Example:

  8 memory banks with a bank busy time of 6 cycles and a total memory latency of 12 cycles. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?

- **Answer:**
  - Stride of 1: number of banks is greater than the bank busy time, so it takes
    - 12+64 = 76 clock cycles → 1.2 cycle per element
  - Stride of 32: the worst case scenario happens when the stride value is a multiple of the number of banks, which this is! Every access to memory will collide with the previous one! Thus, the total time will be:
    - 12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clock cycles per element!

Scatter-Gather

- Consider sparse vectors A & C and vector indices K & M, and A and C have the same number (n) of non-zeros:

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

Ra, Rc, Rk and Rm the starting addresses of 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[]]
```

Programming Vec. Architectures

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

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

How to Pick Max. Vector Length?

- Vector length => Keep all VFUs busy:

- Vector length >= $\frac{\text{(\# lanes) X (\# VFUs)}}{\text{\# Vector instr. issued/cycle}}$

- Notes:
  - Single instruction issue is always the simplest
  - Don’t forget you have to issue some scalar instructions as well

How to Pick Max Vector Length?

- Longer good because:
  - Lower instruction bandwidth
  - If know max length of app. is < max vector length, no strip mining overhead
  - Tiled access to memory reduce scalar processor memory bandwidth needs
  - Better spatial locality for memory access
- Longer not much help because:
  - Diminishing returns on overhead savings as keep doubling number of elements
  - Need natural app. vector length to match physical register length, or no help
  - Area for multi-ported register file

How to Pick # of Vector Registers?

- More vector registers:
  - Reduces vector register “spills” (save/restore)
  - Aggressive scheduling of vector instructions: better compiling to take advantage of ILP
- Fewer
  - Fewer bits in instruction format (usually 3 fields)
- 32 vector registers are usually enough

Context Switch Overhead?

- The vector register file holds a huge amount of architectural state
  - Too expensive to save and restore all on each context switch
- Extra dirty bit per processor
  - If vector registers not written, don't need to save on context switch
- Extra valid bit per vector register, cleared on process start
  - Don't need to restore on context switch until needed
- Extra tip:
  - Save/restore vector state only if the new context needs to issue vector instructions

Exception Handling: Arithmetic

- Arithmetic traps are hard
- Precise interrupts => large performance loss
  - Multimedia applications don’t care much about arithmetic traps anyway
- Alternative model
  - Store exception information in vector flag registers
  - A set flag bit indicates that the corresponding element operation caused an exception
  - Software inserts trap barrier instructions from SW to check the flag bits as (if/when) needed
  - IEEE floating point requires 5 flag registers (5 types of traps)

Exception Handling: Page Faults

- Page faults must be precise
  - Instruction page faults not a problem
  - Data page faults harder
- Option 1: Save/restore internal vector unit state
  - Freeze pipeline, (dump all vector state), fix fault, (restore state and) continue vector pipeline
- Option 2: expand memory pipeline to check all addresses before send to memory
  - Requires address and instruction buffers to avoid stalls during address checks
  - On a page-fault on only needs to save state in those buffers
  - Instructions that have cleared the buffer can be allowed to complete

Exception Handling: Interrupts

- Interrupts due to external sources
  - I/O, timers etc
- Handled by the scalar core
- Should the vector unit be interrupted?
  - Not immediately (no context switch)
  - Only if it causes an exception or the interrupt handler needs to execute a vector instruction

Summary of Vector Architecture

- Optimizations:
  - *Multiple Lanes*: > 1 element per clock cycle
  - *Vector Length Registers*: Non-64 wide vectors
  - *Vector Mask Registers*: IF statements in vector code
  - *Memory Banks*: Memory system optimizations to support vector processors
  - *Stride*: Multiple dimensional matrices
  - *Scatter-Gather*: Sparse matrices
  - *Programming Vector Architectures*: Program structures affecting performance

Example: Vector Multiplication

- Consider the following code, which multiplies two vectors that contain single-precision complex values:
  - for (i=0; i<300; i++) {
  - c_re[i] = a_re[i] * b_re[i] – a_im[i] * b_im[i];
  - c_im[i] = a_re[i] * b_im[i] + a_im[i] * b_re[i];
- Asumme that the processor runs at 700 MHz and has a maximum vector length of 64.
  - What is the arithmetic intensity of this kernel (i.e., the ratio of floating-point operations per byte of memory accessed)?
  - Convert this loop into VMIPS assembly code using strip mining.
  - Assuming chaining and a single memory pipeline, how many chimes are required?

Example: Vector Multiplication

A. The code reads four floats and writes two floats for every six FLOPs, so the arithmetic intensity = 6/6 = 1.

B. Assume MVL = 64 → 300 mod 64 = 44

```
        li      $VL,44          # perform the first 44 ops
        li      $r1,0           # initialize index
loop:   lv      $v1,a_re+$r1    # load a_re
        lv      $v3,b_re+$r1    # load b_re
        mulvv.s $v5,$v1,$v3     # a+re*b_re
        lv      $v2,a_im+$r1    # load a_im
        lv      $v4,b_im+$r1    # load b_im
        mulvv.s $v6,$v2,$v4     # a+im*b_im
        subvv.s $v5,$v5,$v6     # a+re*b_re - a+im*b_im
        sv      $v5,c_re+$r1    # store c_re
        mulvv.s $v5,$v1,$v4     # a+re*b_im
        mulvv.s $v6,$v2,$v3     # a+im*b_re
        addvv.s $v5,$v5,$v6     # a+re*b_im + a+im*b_re
        sv      $v5,c_im+$r1    # store c_im
        bne     $r1,0,else      # check if first iteration
        addi    $r1,$r1,#44     # first iteration,
                                increment by 44
        j loop                  # guaranteed next iteration
else:   addi    $r1,$r1,#256    # not first iteration,
                                increment by 256
skip:   blt     $r1,1200,loop   # next iteration?
```

Example: Vector Multiplication

c. Identify convoys:

1. mulvv.s lv # a_re * b_re
   # (assume already loaded),
   # load a_im
2. lv mulvv.s # load b_im, a_im * b_im
3. subvv.s sv # subtract and store c_re
4. mulvv.s lv # a_re * b_re,
   # load next a_re vector
5. mulvv.s lv # a_im * b_re,
   # load next b_re vector
6. addvv.s sv # add and store c_im

6 chimes

```
        li      $VL,44          # perform the first 44 ops
        li      $r1,0           # initialize index
loop:   lv      $v1,a_re+$r1    # load a_re
        lv      $v3,b_re+$r1    # load b_re
        mulvv.s $v5,$v1,$v3     # a+re*b_re
        lv      $v2,a_im+$r1    # load a_im
        lv      $v4,b_im+$r1    # load b_im
        mulvv.s $v6,$v2,$v4     # a+im*b_im
        subvv.s $v5,$v5,$v6     # a+re*b_re - a+im*b_im
        sv      $v5,c_re+$r1    # store c_re
        mulvv.s $v5,$v1,$v4     # a+re*b_im
        mulvv.s $v6,$v2,$v3     # a+im*b_re
        addvv.s $v5,$v5,$v6     # a+re*b_im + a+im*b_re
        sv      $v5,c_im+$r1    # store c_im
        bne     $r1,0,else      # check if first iteration
        addi    $r1,$r1,#44     # first iteration,
                                increment by 44
        j loop                  # guaranteed next iteration
else:   addi    $r1,$r1,#256    # not first iteration,
                                increment by 256
skip:   blt     $r1,1200,loop   # next iteration?
```

GPU: Graphical Processing Units

- Pixels in frame buffer (video memory?) are many but independent
- Graphics operations touch many pixels => need acceleration => graphics cards with basic pixel operations
  - + memory bandwidth!
- Since they are already there, can we use them for other (general purpose) computation?
  - Only incremental cost, as already there for graphics!
- Data parallel, SIMD?
  - Programming model is “Single Instruction Multiple Thread” (SIMT)

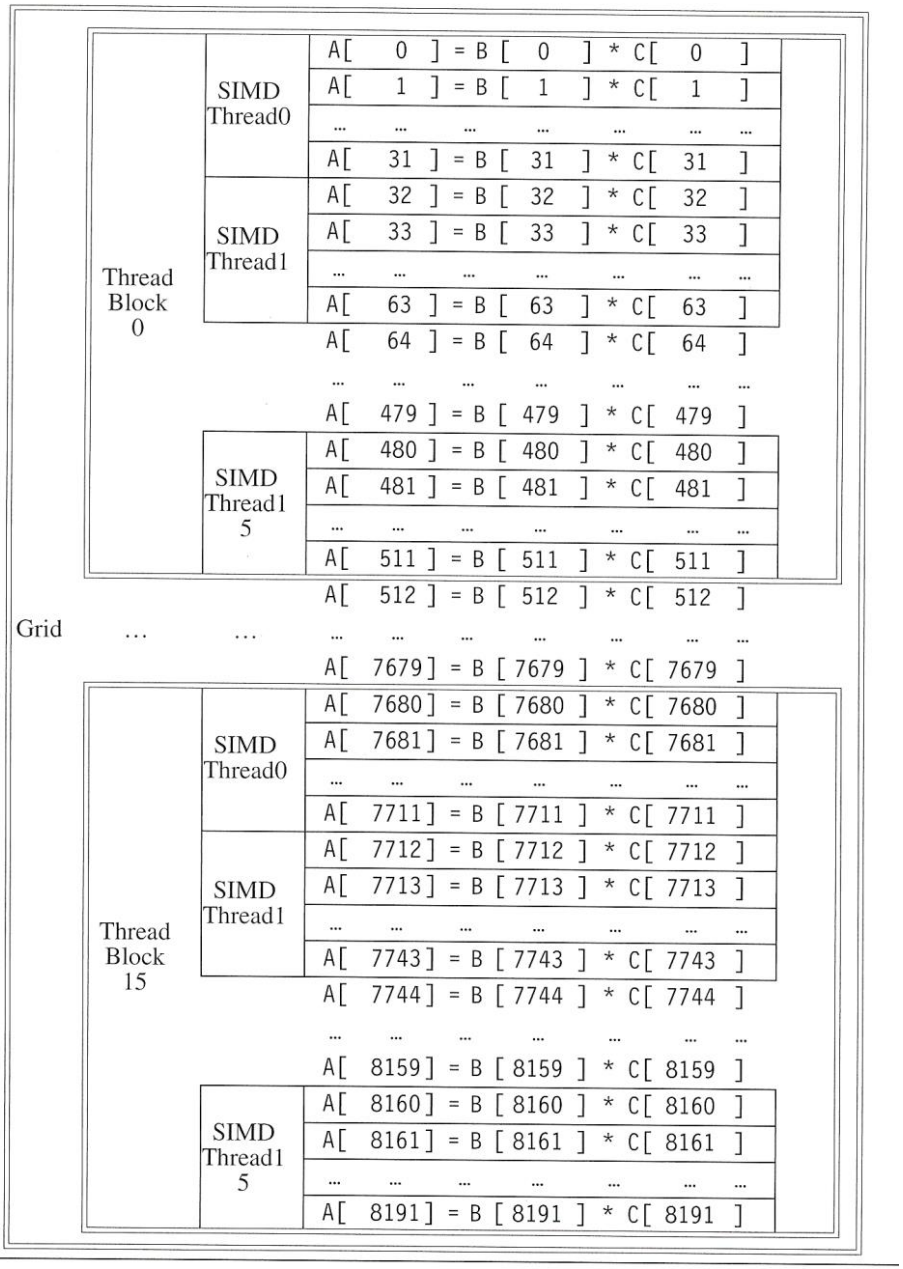
GPU: Graphical Processing Units

- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Initially program in “assembly” (low-level)
  - Develop a C-like programming language for GPU
    - Compute Unified Device Architecture (CUDA)
    - OpenCL for vendor-independent language
  - Unify all forms of GPU parallelism as *CUDA thread*

Threads and Blocks

- A thread is associated with each data element
  - *CUDA threads*, with thousands of which being utilized to various styles of parallelism: multithreading, SIMD, MIMD, ILP
- Threads are organized into blocks
  - *Thread Blocks*: groups of up to 512 elements
  - *Multithreaded SIMD Processor*: hardware that executes a whole thread block (32 elements executed per thread at a time)
- Blocks are organized into a grid
  - Blocks are executed independently and in any order
  - Different blocks cannot communicate directly but can *coordinate* using atomic memory operations in Global Memory
- GPU hardware handles thread management, not applications or OS
  - A multiprocessor composed of multithreaded SIMD processors
  - A Thread Block Scheduler

Grid, Threads, and Blocks



NVIDIA GPU Architecture

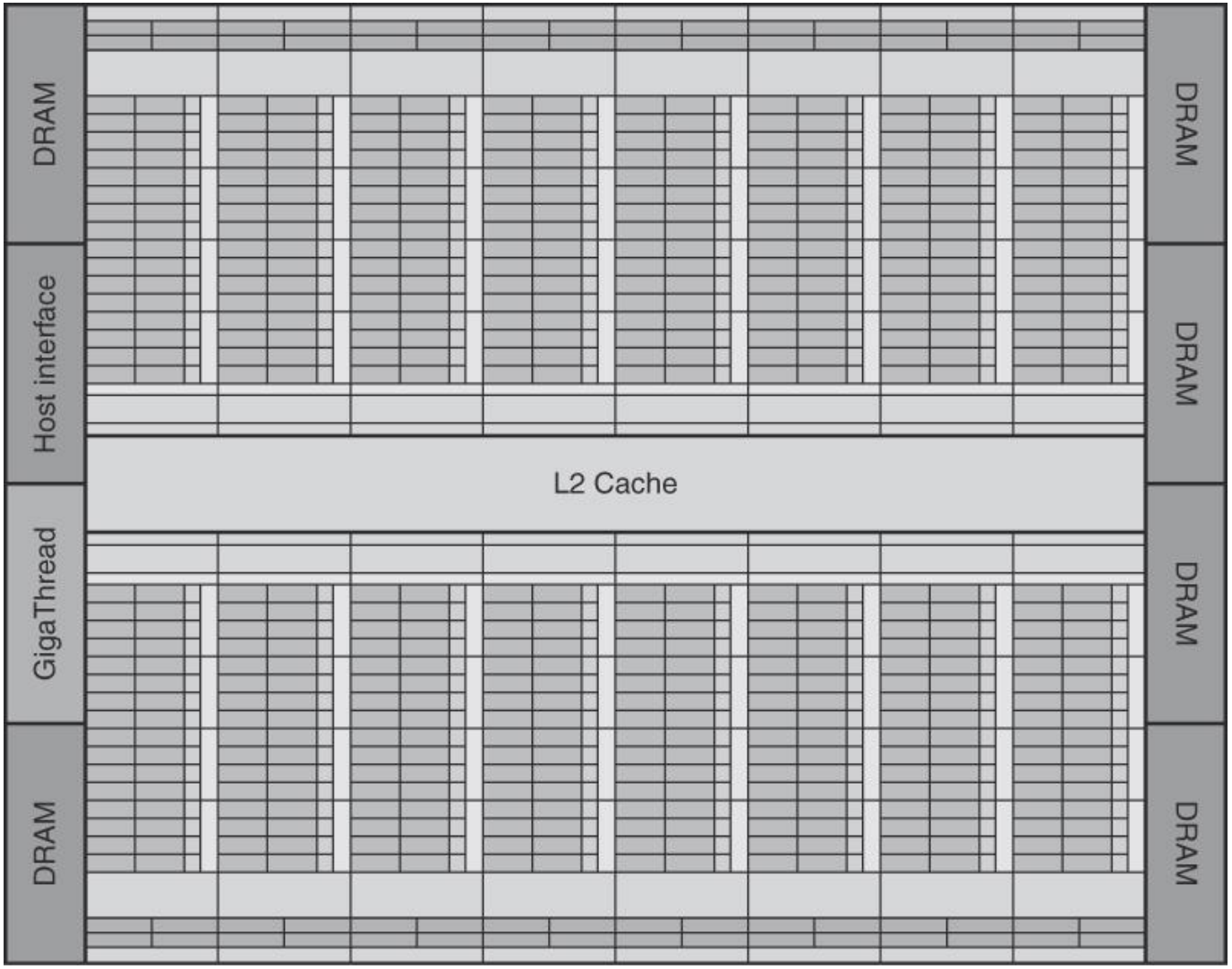
- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files

- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Example

- Multiply two vectors of length 8192
  - Code that works over all elements is the grid
  - Thread blocks break this down into manageable sizes
    - 512 elements/block, 16 SIMD threads/block → 32 ele/thread
  - 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

Fermi GTX 480 GPU Floor plan

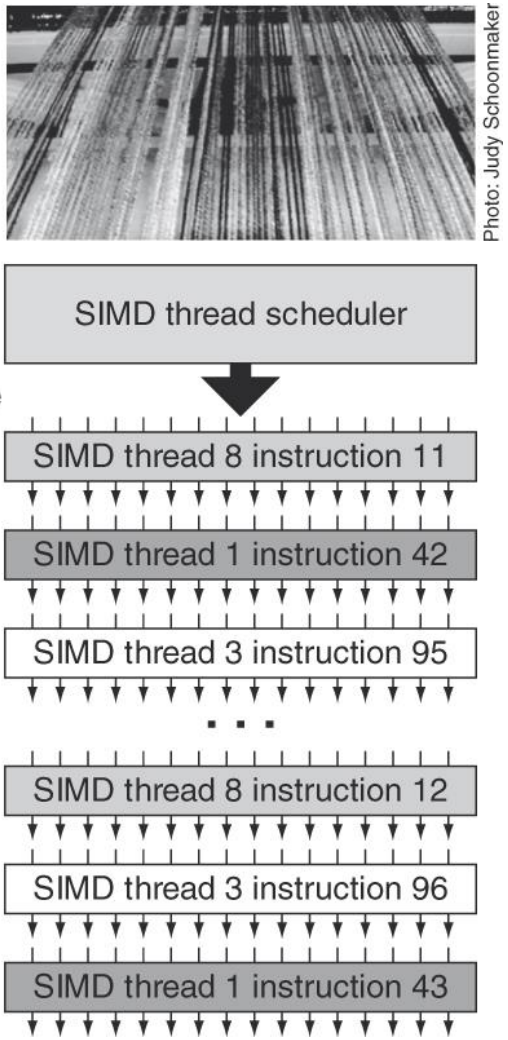


6 GDDR5 ports, 64 bits wide, up to 6 GB size. Thread Block Scheduler shown on the left

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

Scheduling of SIMD instructions

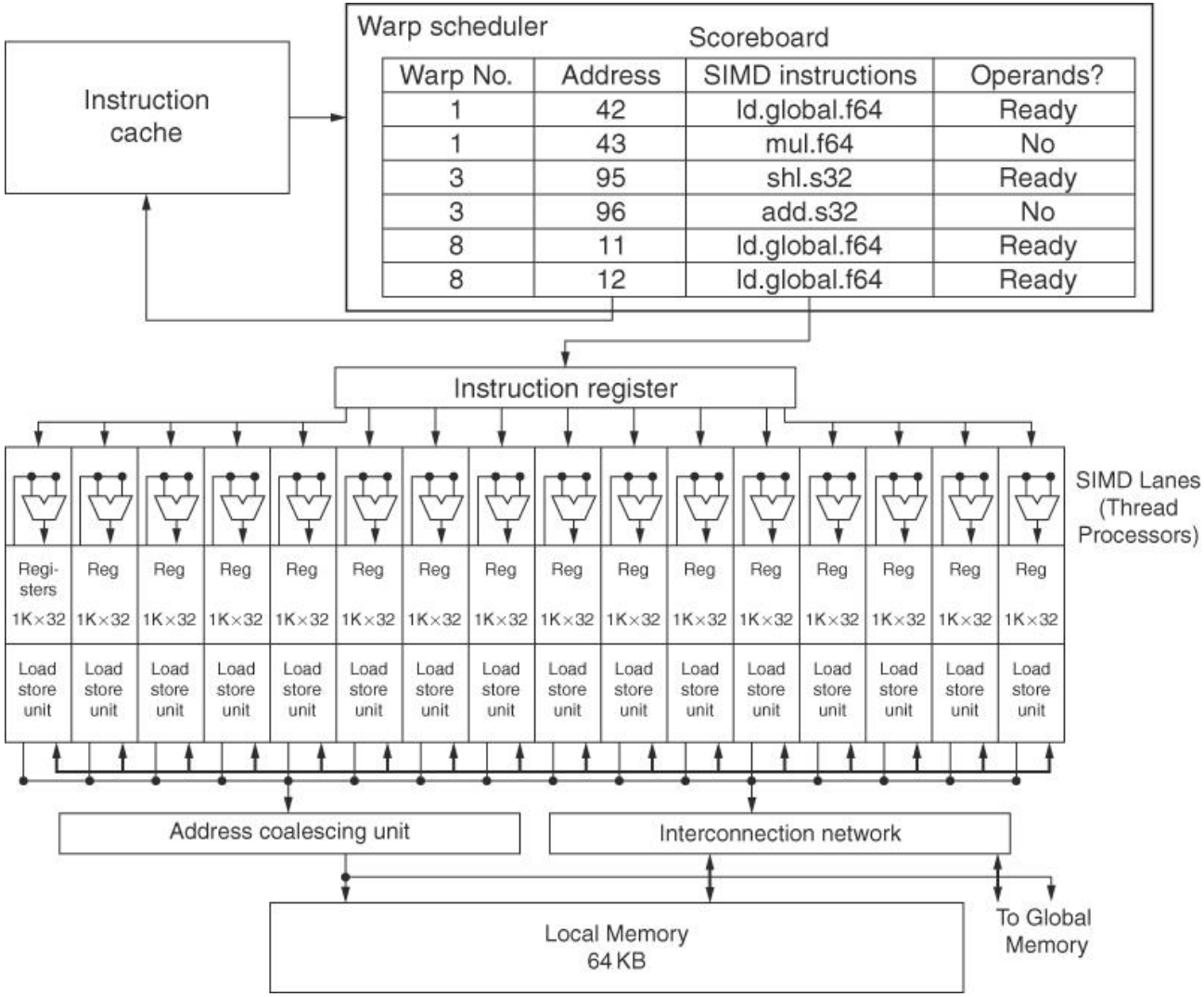


The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD thread. Since threads of SIMD instructions are independent, the scheduler may select a different SIMD thread each time.

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

Multithreaded SIMD Processor



16 SIMD lanes: The SIMD Thread Scheduler has, for example, 48 independent threads of SIMD instructions that it schedules with a table of 48 PCs.

NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
  - “Parallel Thread Execution (PTX)”
  - Uses virtual registers
  - Translation to machine code is performed in software
  - Example: one CUDA thread, 8192 of these created!

```
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

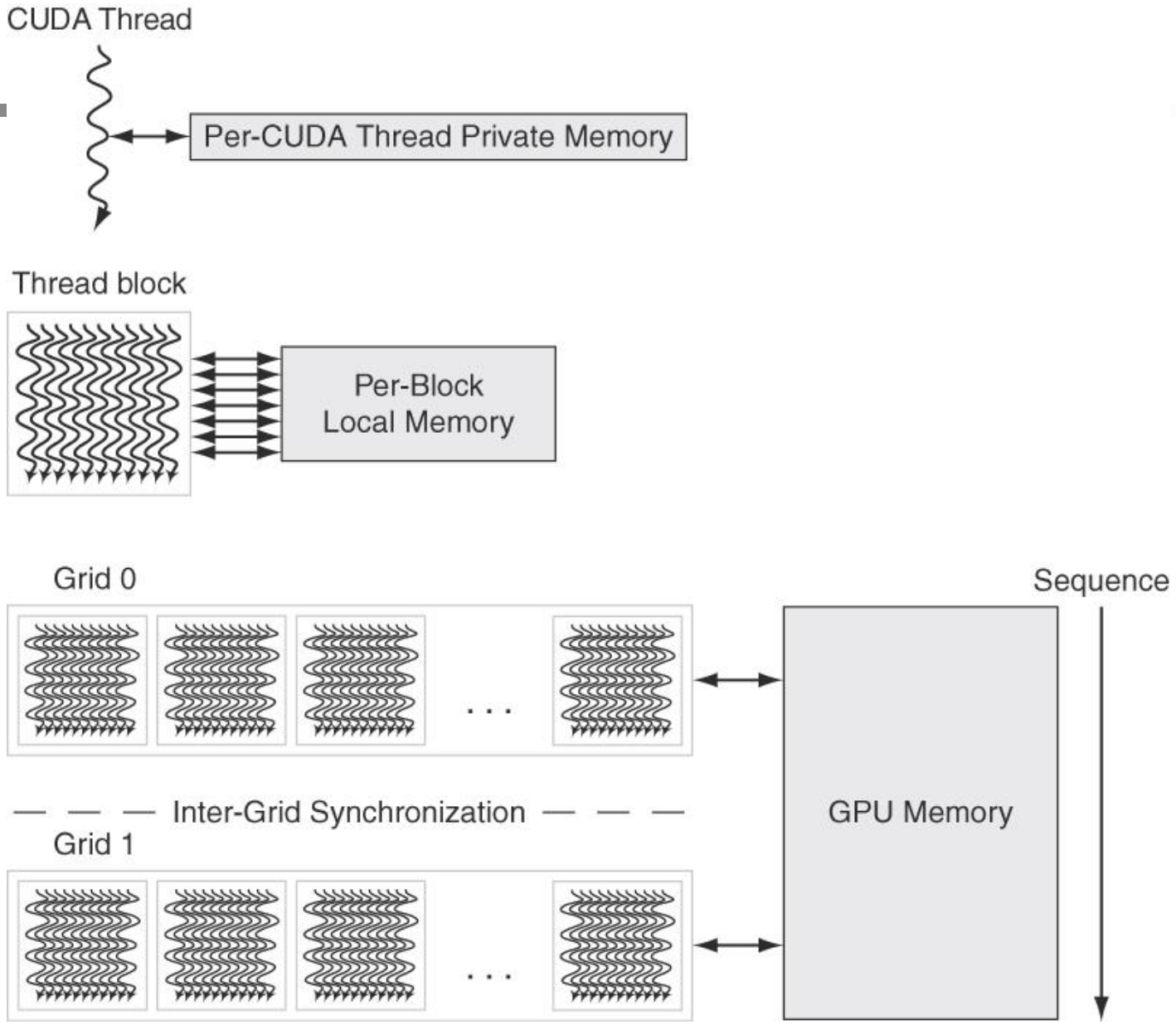
Example

```
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

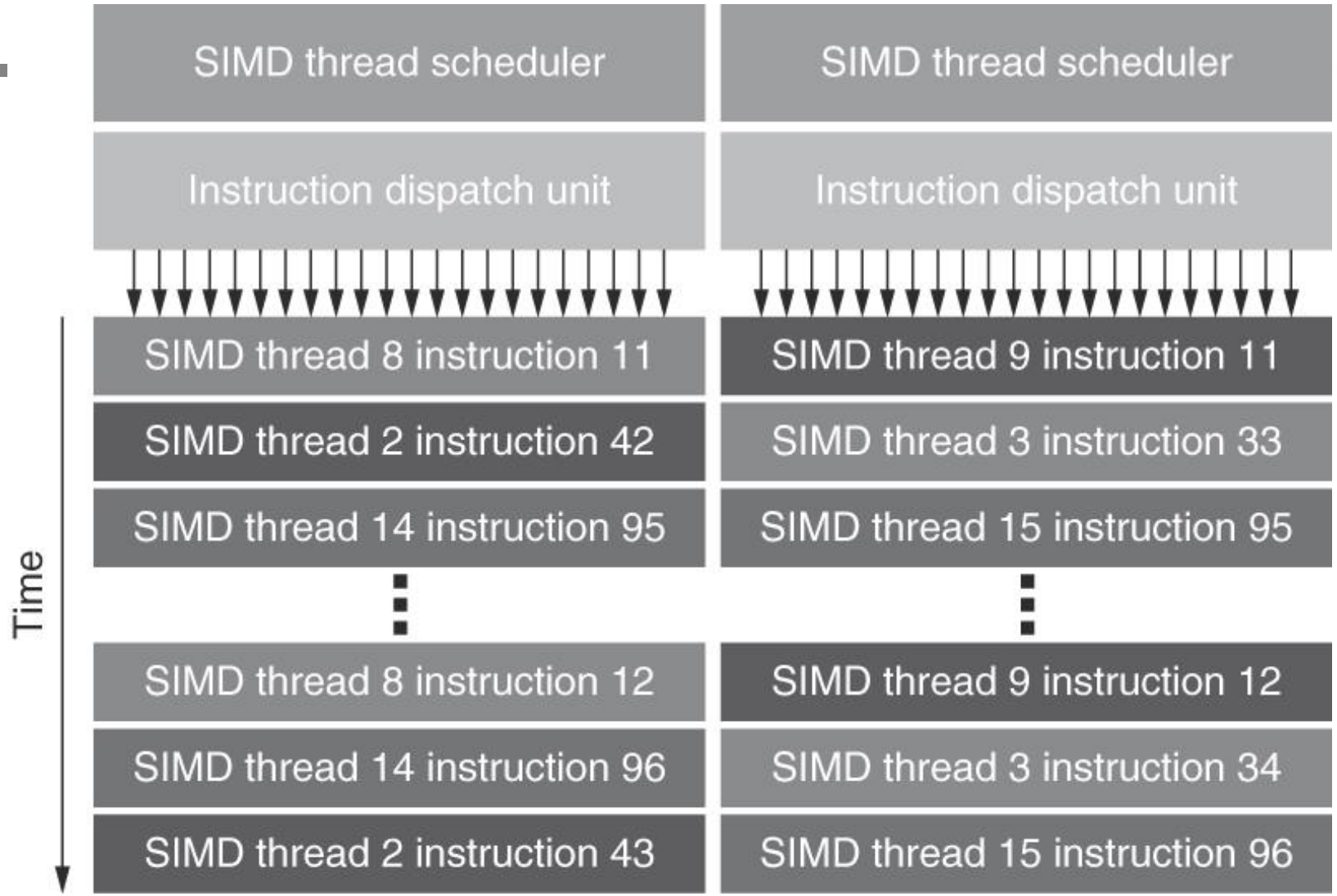
- Each SIMD Lane has private section of *off-chip* DRAM
  - "Private memory", not shared by any other lanes
  - Contains stack frame, spilling registers, and private variables
  - Recent GPUs cache this in L1 and L2 caches
- Each multithreaded SIMD processor also has local memory that is *on-chip*
  - Shared by SIMD lanes / threads *within a block only*
- The *off-chip* memory shared by SIMD processors is *GPU Memory*
  - Host can read and write GPU memory



GPU Memory structures. GPU Memory is shared by all Grids (vectorized loops), Local Memory is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), and Private Memory is private to a single CUDA Thread.

Fermi Architecture Innovations

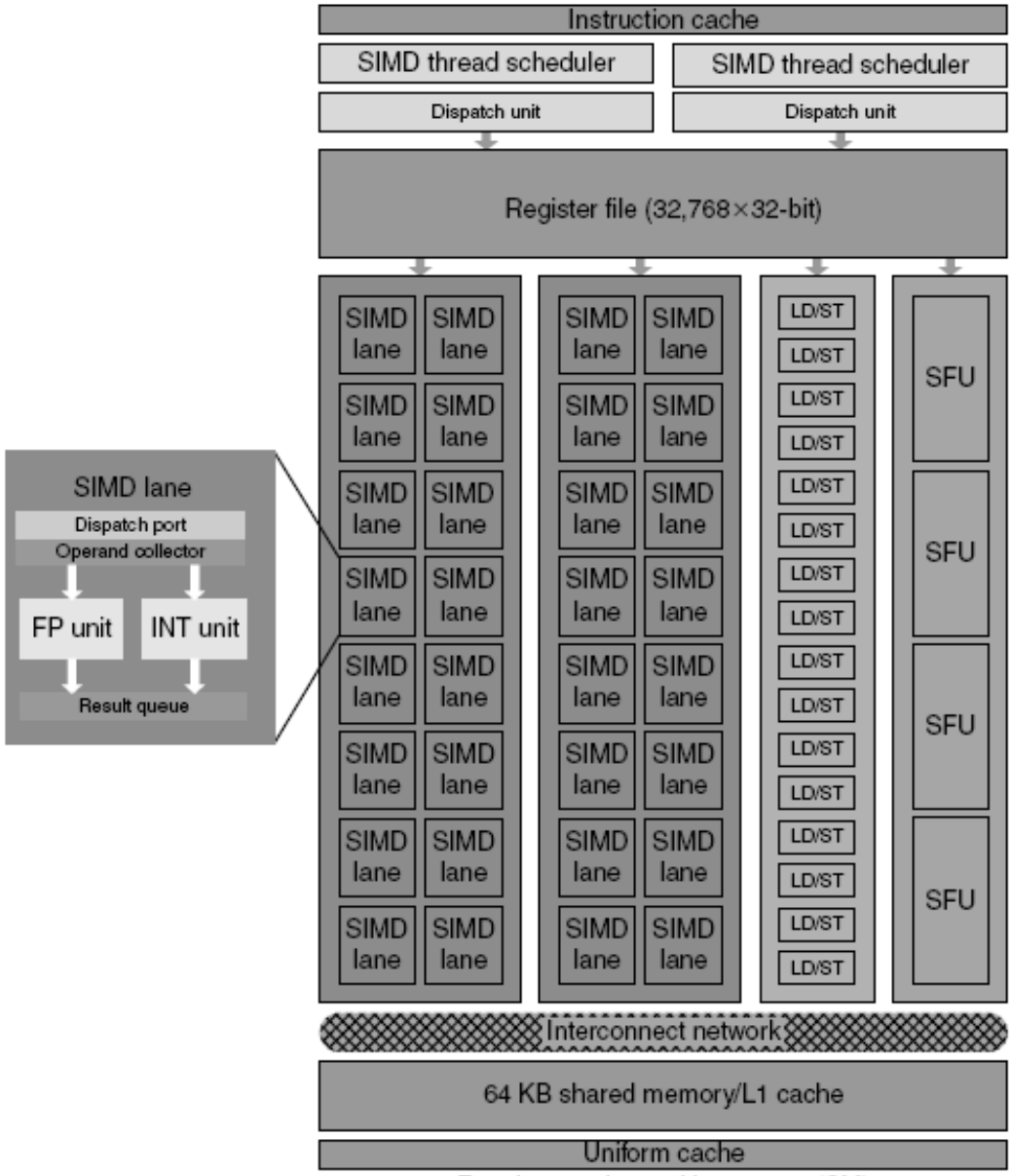
- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles
- Fast double precision: gen- 78 →515 GFLOPs for DAXPY
- Caches for GPU memory: I/D L1/SIMD proc and shared L2
- 64-bit addressing and unified address space: C/C++ ptrs
- Error correcting codes: dependability for long-running apps
- Faster context switching: hardware support, 10X faster
- Faster atomic instructions: 5-20X faster than ealier



Block Diagram of Fermi's Dual SIMD Thread Scheduler

Compare this design to the single SIMD Thread Design

Fermi Multithreaded SIMD Proc.



Fermi streaming multiprocessor (SM)

Loop-Level Parallelism

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

- Example 1:

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

- No loop-carried dependence

Loop-Level Parallelism

- Example 2:

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

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

Loop-Level Parallelism

- Example 3:

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

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

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

Loop-Level Parallelism

- Example 4:

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

  - No loop-carried dependence

- Example 5:

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

  - Loop-carried dependence in the form of *recurrence*

Finding dependencies

- Assume that a *1-D* array index *i* is *affine*:
  - $a \times i + b$ (with constants a and b)
- An index in an *n-D* array index is *affine* if it is affine in each dimension
- Assume:
  - Store to $a \times i + b$, then
  - Load from $c \times i + d$
  - i runs from m to n
  - Dependence exists if:
    - Given j, k such that $m \leq j \leq n$, $m \leq k \leq n$
    - Store to $a \times j + b$, load from $a \times k + d$, and $a \times j + b = c \times k + d$

Finding dependencies

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

- Example:

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

- Answer: a=2, b=3, c=2, d=0 →GCD(c,a)=2, d-b=-3 → no dependence possible.

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 */
}
```

```
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];
}
```

- Watch for antidependencies and output dependencies:
  - RAW: S1→S3, S1→S4 on Y[i], not loop-carried
  - WAR: S1→S2 on X[i]; S3→S4 on Y[i]
  - WAW: S1→S4 on Y[i]

Reductions

- Reduction Operation:

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

- Transform to…

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

- Do on p processors:

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

- Note: assumes associativity!