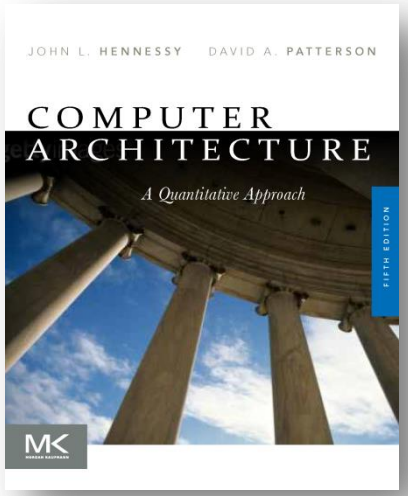


Chapter 4

Data-Level Parallelism in Vector, SIMD, and GPU Architectures

Contents

1. SIMD architecture
2. Vector architectures optimizations: *Multiple Lanes*, *Vector Length Registers*, *Vector Mask Registers*, *Memory Banks, Stride*, *Scatter-Gather*,
3. Programming Vector Architectures
4. SIMD extensions for media apps
5. GPUs – Graphical Processing Units
6. Fermi architecture innovations
7. Examples of loop-level parallelism
8. Fallacies

Flynn's Taxonomy

- *SISD* - Single instruction stream, single data stream
- *SIMD* - Single instruction stream, multiple data streams
  - New: *SIMT* – Single Instruction Multiple Threads (for GPUs)
- *MISD* - Multiple instruction streams, single data stream
  - No commercial implementation
- *MIMD* - Multiple instruction streams, multiple data streams
  - Tightly-coupled MIMD
  - Loosely-coupled MIMD

Advantages of SIMD architectures

1. Can exploit significant data-level parallelism for:
   1. matrix-oriented scientific computing
   2. media-oriented image and sound processors
2. More energy efficient than MIMD
   1. Only needs to fetch one instruction per multiple data operations, rather than one instr. per data op.
   2. Makes SIMD attractive for personal mobile devices
3. Allows programmers to continue thinking sequentially

- SIMD/MIMD comparison. Potential speedup for SIMD twice that from MIMID!
  - x86 processors → expect two additional cores per chip per year
  - SIMD → width to double every four years

SIMD parallelism

- SIMD architectures
  - A. Vector architectures
  - B. SIMD extensions for mobile systems and multimedia applications
  - C. Graphics Processor Units (GPUs)

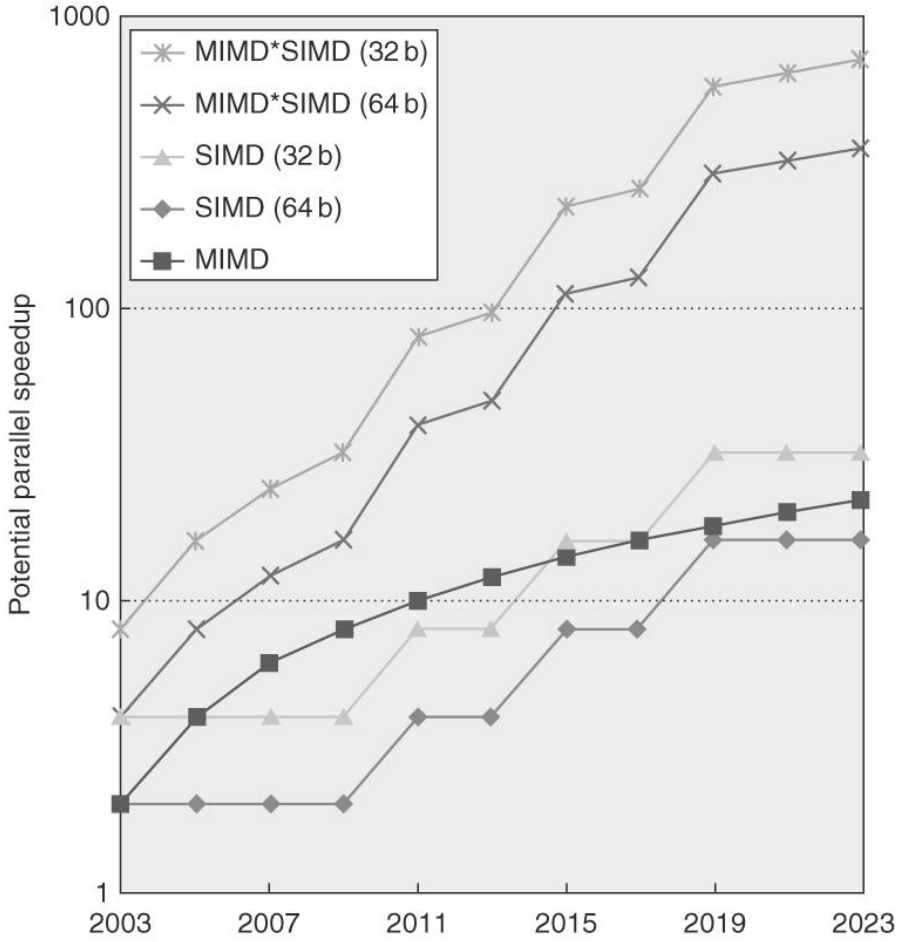


Figure 4.1 Potential speedup via parallelism from MIMD, SIMD, and both MIMD and SIMD over time for x86 computers. This figure assumes that two cores per chip for MIMD will be added every two years and the number of operations for SIMD will double every four years.

A. 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

Example of vector architecture

- VMIPS → MIPS extended with vector instructions
- 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 – FP add and multiply
  - 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

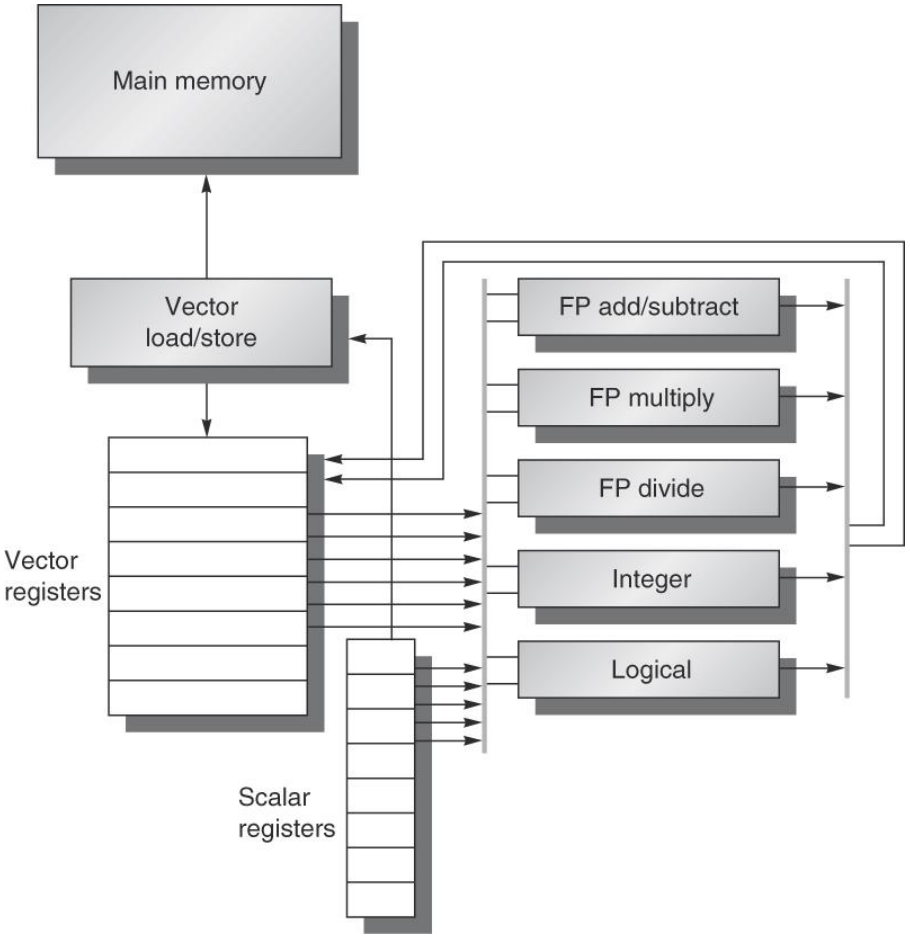


Figure 4.2 The basic structure of a vector architecture, VMIPS. This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. This chapter defines special vector instructions for both arithmetic and memory accesses. The figure shows vector units for logical and integer operations so that VMIPS looks like a standard vector processor that usually includes these units; however, we will not be discussing these units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. A set of crossbar switches (thick grey lines) connects these ports to the inputs and outputs of the vector functional units.

VMIPS instructions

- ADDVV.D: add two vectors.
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address
  - Rx → the address of vector X
  - Ry → the address of vector Y
- Example: DAXPY (double precision a*X+Y) → 6 instructions

```
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
```

Assumption: the vector length matches the number of vector operations – no loop necessary.

DAXPY using 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 when the vectors have 64 elements→ 64 elements of a vector x 9 ops

Execution time

- Vector execution time depends on:
  - 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

Chaining and chimes

- *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
- Sequences with read-after-write dependency hazards can be in the same convey via *chaining*

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
```

Three convoys:

1	LV	MULVS.D	→ first chime
2	LV	ADDVV.D	→ second chime
3	SV		→ third chime

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

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

Challenges

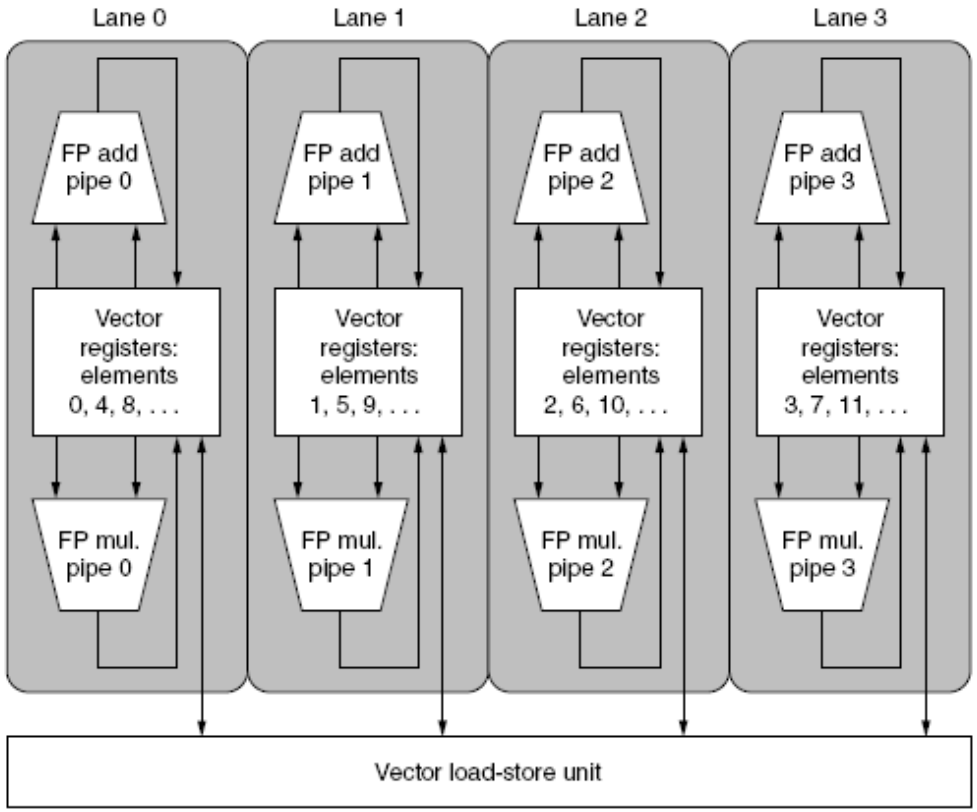
- The chime model ignores the *vector start-up time* determined by the pipelining latency of vector functional units
- Latency of vector functional units. Assume the same as Cray-1
  - Floating-point add ➔ 6 clock cycles
  - Floating-point multiply ➔ 7 clock cycles
  - Floating-point divide ➔ 20 clock cycles
  - Vector load ➔ 12 clock cycles

Optimizations

1. *Multiple Lanes* → processing more than one element per clock cycle
2. *Vector Length Registers* → handling non-64 wide vectors
3. *Vector Mask Registers* → handling IF statements in vector code
4. *Memory Banks* → memory system optimizations to support vector processors
5. *Stride* → handling multi-dimensional arrays
6. *Scatter-Gather* → handling sparse matrices
7. *Programming Vector Architectures* → program structures affecting performance

1. A four lane vector unit

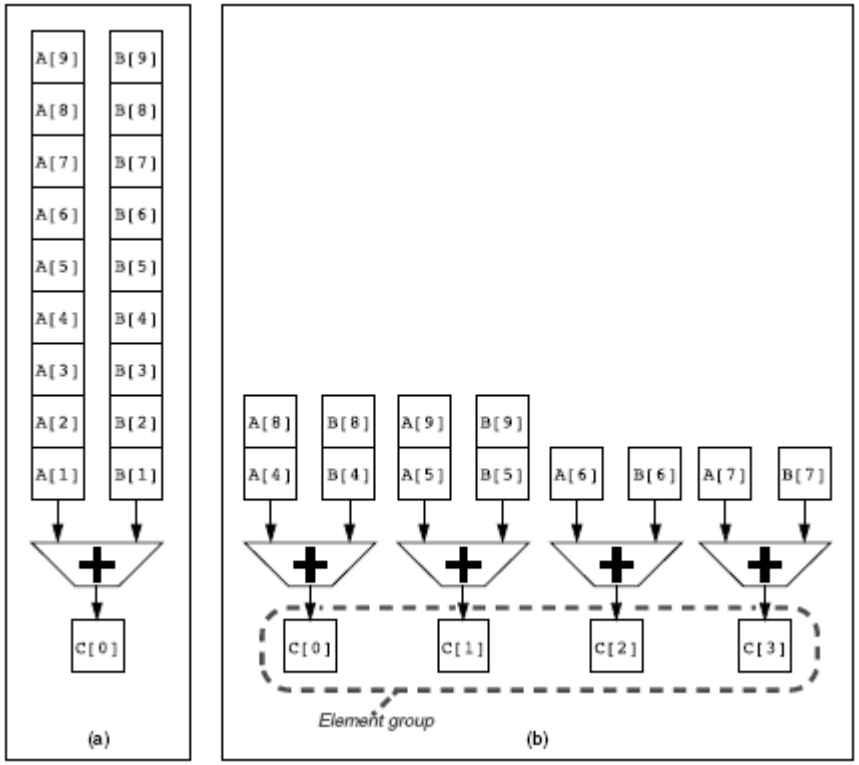
- VIMPS instructions only allov element N of one vector to take part in operations involving element N from other vector registers→ this simplifies the construction of a highly parallel vector unit
  - Line → contains one portion of the vector register file and one execution pipeline from each functional unit
  - Analog with a highway with multiple lanes!!



Single versus multiple add pipelines

- C= A+B
- One versus four additions per clock cycl
- Each pipe adds the corresponding elements of the two vectors

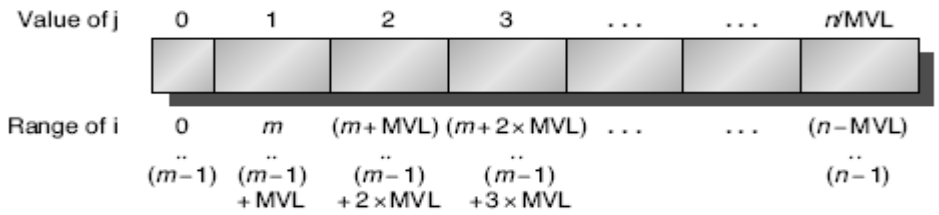
  C(i) = A(i) + B(i)



2. VLR and MVL

- VLR →Vector Length Register; MVL → Max Vector Length
- Vector length:
  - Not known at compile time?
  - Not multiple of 64?
- 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 length to maximum vector length*/
}
```



3. Vector mask registers

- Handling IF statements in a loop

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

- *If conversion* →use vector mask register to “disable/select” vector elements

```
LV          V1,Rx        ;load vector X into V1
LV          V2,Ry        ;load vector Y into V2
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!

4. 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 is 2.167 ns, SRAM cycle time is 15 ns
  - How many memory banks needed?
    - 32 processors x 6 =192 accesses,
    - 15ns SDRAM cycle /2.167ns processor cycle≈7 processor cycles
    - 7 x 192 →1344!

5. Stride → multiple dimensional arrays

- Technique to fetch vector elements that are not adjacent in memory
- Stride → the distance between elements to be gathered in one register.
- Example (recall that in C an array is stored in major row order!!)

```
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; D's stride is 100 double words (800 bytes); B's stride is one double word (8 bytes)*
- 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

- Given
  - 8 memory banks
  - bank busy time of 6 cycles
  - total memory latency of 12 cycles.
- **Questions**: How long will it take to complete a 64-element vector load
  1. With a stride of 1?
  2. With a stride of 32?
- **Answers:**
  1. Stride of 1: number of banks is greater than the bank busy time, so it takes 12 + 64 = 76 clock cycles → 76/64 = 1.2 cycle for each vector element
  2. 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→ 391/64 = 6.1 clock cycles per vector element!

6 Scatter-gather

- Consider sparse vectors A & C and *vector indices* K & M. 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[]]
```

7 Programming vector 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

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

Exercise

Consider the following code, which multiplies two vectors of length 300 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];
```

The processor runs at 700 MHz and has a maximum vector length of 64.

A. What is the arithmetic intensity of this kernel (i.e., the ratio of floating-point operations per byte of memory accessed)?
B. Convert this loop into VMIPS assembly code using strip mining.
C. Assuming chaining and a single memory pipeline, how many chimes are required?

Exercise – arithmetic intensity

This code reads four floats (4 lv) and writes two floats (2 sv) for every six FLOPs (4 mulvv.s + 1 subvv.s + 1 addvv.s).

Arithmetic intensity = (4+2)/6 = 1.

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?
```

Exercise - 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?
```

B. SIMD extensions for media apps

- Media applications operate on data types narrower than the native word size.
  - Graphics: 3x8-bit colors, 8-bit for transparency
  - Audio: 8/16/24 bit/sample
- Disconnect carry chains to “partition” adder.
  - Example: a 256 adder can be partitioned to perform simultaneously:
    - 32 x 8-bit additions
    - 16 x 16-bit additions
    - 8 x 32-bit additions
    - 4 x 64-bit additions
- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers

SIMD extension to x86-64 implementations

- Intel MMX (1996)
  - Eight 8-bit integer ops or four 16-bit integer ops
- Streaming SIMD Extensions: (SSE) (1999), SSE3 (2004),SSE4 (2007)
  - Eight 16-bit integer ops
  - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
- Advanced Vector Extensions (AVE) (2010)
  - Four 64-bit integer/fp ops
- Operands must be consecutive and at 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
  - Easy to implement
  - Require little extra state → easy for context-switching
  - Require little extra memory bandwidth
  - No virtual memory problem of cross-page access and page-fault

AVX Instruction	Description
VADDPD	Add four packed double-precision operands
VSUBPD	Subtract four packed double-precision operands
VMULPD	Multiply four packed double-precision operands
VDIVPD	Divide four packed double-precision operands
VFMADDPD	Multiply and add four packed double-precision operands
VFMSUBPD	Multiply and subtract four packed double-precision operands
VCMPxx	Compare four packed double-precision operands for EQ, NEQ, LT, LE, GT, GE, ...
VMOVAPD	Move aligned four packed double-precision operands
VBROADCASTSD	Broadcast one double-precision operand to four locations in a 256-bit register

Figure 4.9 AVX instructions for x86 architecture useful in double-precision floating-point programs. Packed-double for 256-bit AVX means four 64-bit operands executed in SIMD mode. As the width increases with AVX, it is increasingly important to add data permutation instructions that allow combinations of narrow operands from different parts of the wide registers. AVX includes instructions that shuffle 32-bit, 64-bit, or 128-bit operands within a 256-bit register. For example, BROADCAST replicates a 64-bit operand 4 times in an AVX register. AVX also includes a large variety of fused multiply-add/subtract instructions; we show just two here.

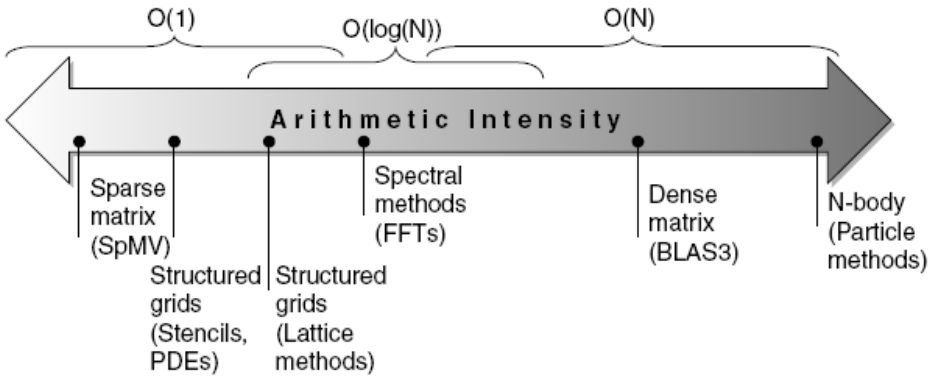
Example: SIMD code for DXPY; Y = Y + a X

- 256 – bit SIMD multimedia instructions added to MIPS.
- .4D → instructions operating on 4 double precision operands at once.

```
    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
```

Roofline performance model

- Basic idea:
  - Peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance
  - Roofline: on the sloped portion of the roof the performance is limited by the memory bandwidth, on the flat portion it is limited by arithmetic intensity
- *Arithmetic intensity* → Floating-point operations per byte read

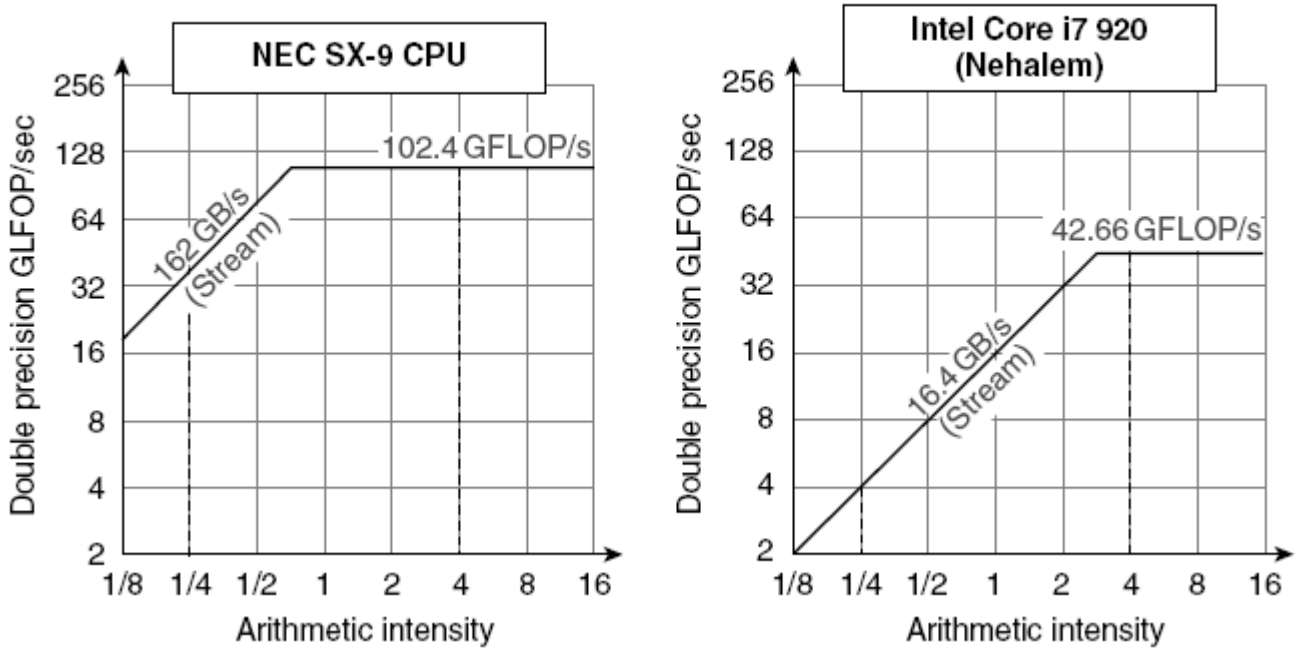


- Dense matrix operations scale with problem size but sparse matrix operations do not!!

Examples

- Attainable GFLOPs/sec

  Min = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Performance)



C. Graphical Processing Unit - GPU

- Given the hardware invested to do graphics well, how can it be supplemented to improve performance of a wider range of applications?
- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - 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*
  - Programming model: “Single Instruction Multiple Thread” (SIMT)

Threads, blocks, and grid

- A *thread* is associated with each data element
  - *CUDA threads* ➔ thousands of threads are 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
- Thread management handled by GPU hardware not by applications or OS
  - A multiprocessor composed of multithreaded SIMD processors
  - A Thread Block Scheduler

NVIDIA GPU architecture

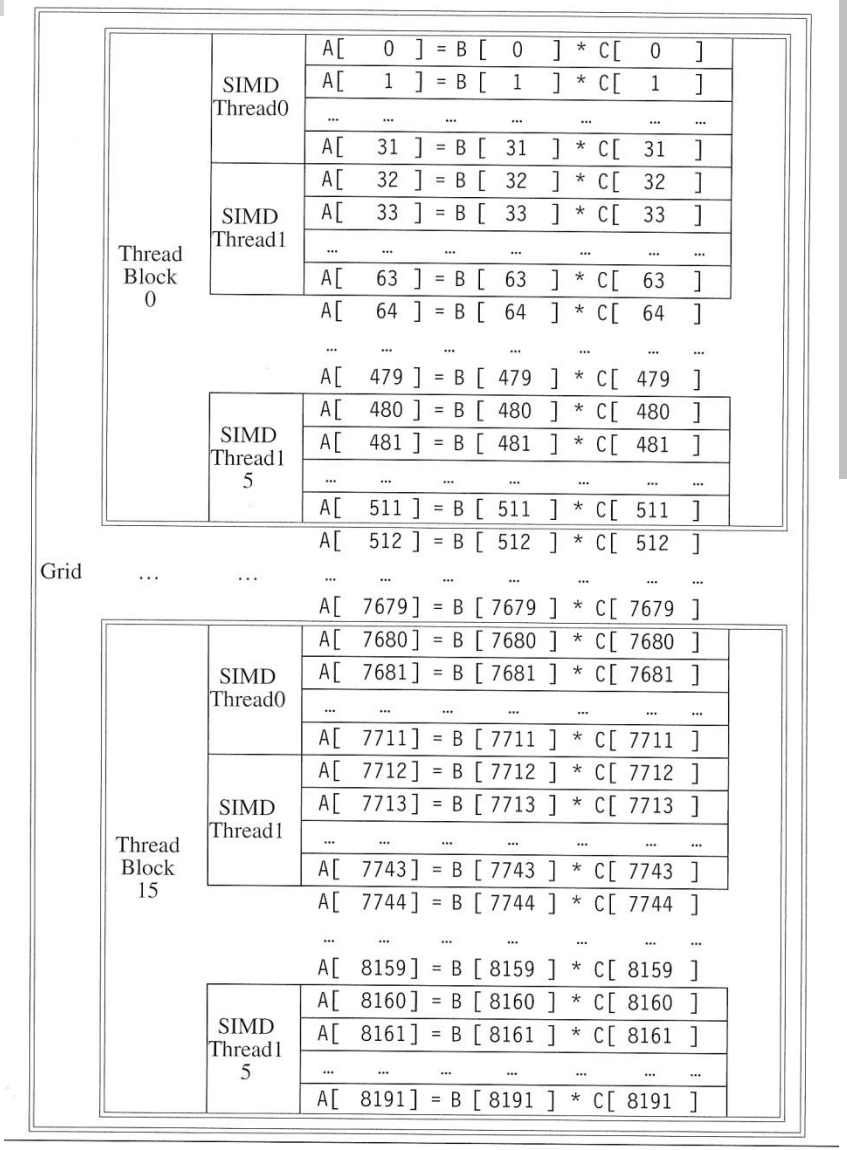
- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files

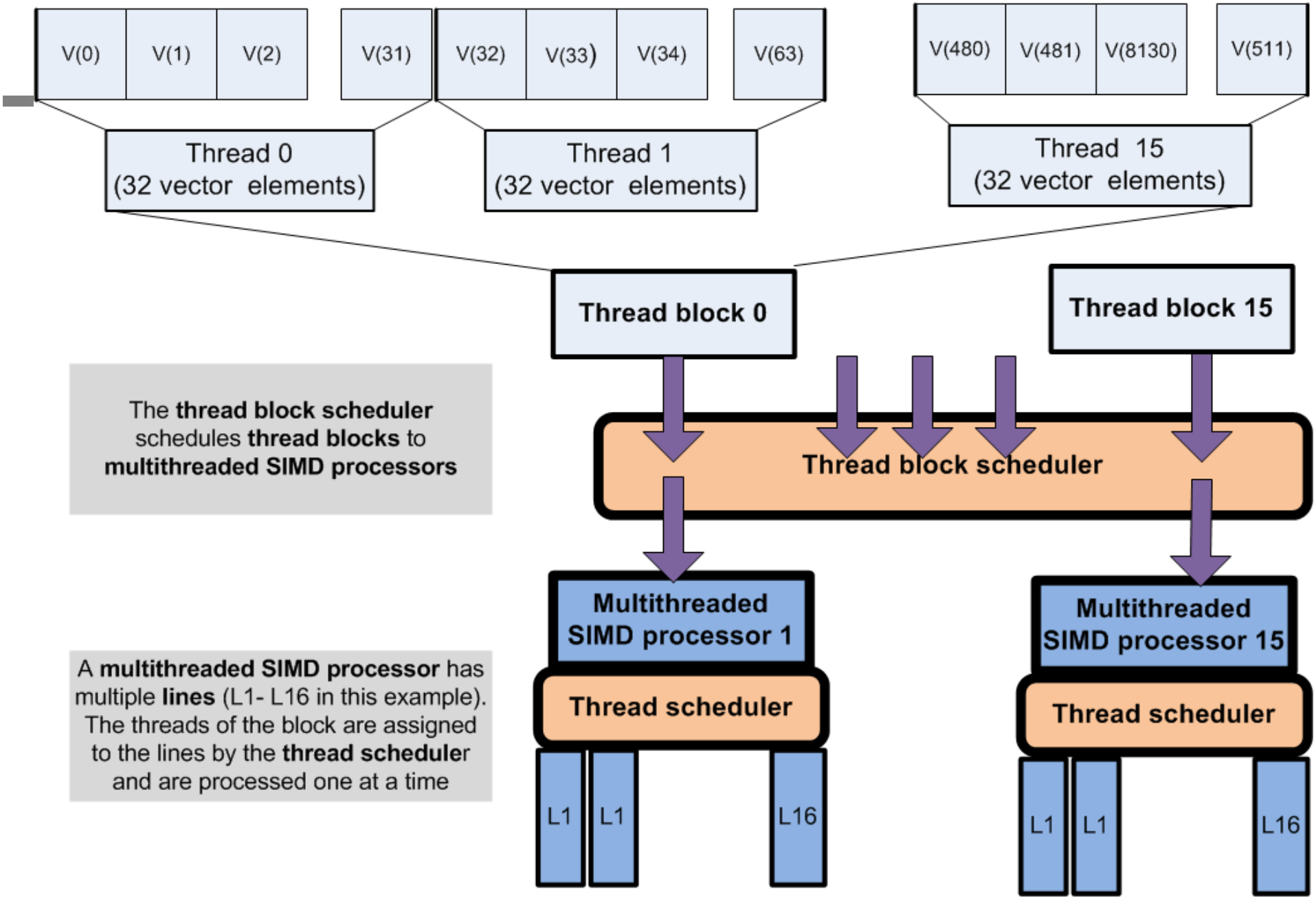
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Example: multiply two vectors of length 8192

- Grid → Code that works over all elements
- Thread block→ analogous to a strip-mined vector loop with vector length of 32. Breaks down the vector into manageable set of vector elements
  - 32 elements/thread x 16 SIMD threads/block → 512 elements/block
  - SIMD instruction executes 32 elements at a time
  - Grid size = 8192vector elements / 512 elements/block = 16 blocks
- *Thread block scheduler* → assigns a thread block to a *multithreaded SIMD processor*
- Current-generation GPUs (Fermi) have 7-15 multithreaded SIMD processors

Threads, blocks, and grid example





The **thread block scheduler** schedules **thread blocks** to **multithreaded SIMD processors**

A **multithreaded SIMD processor** has multiple **lines** (L1- L16 in this example). The threads of the block are assigned to the lines by the **thread scheduler** and are processed one at a time

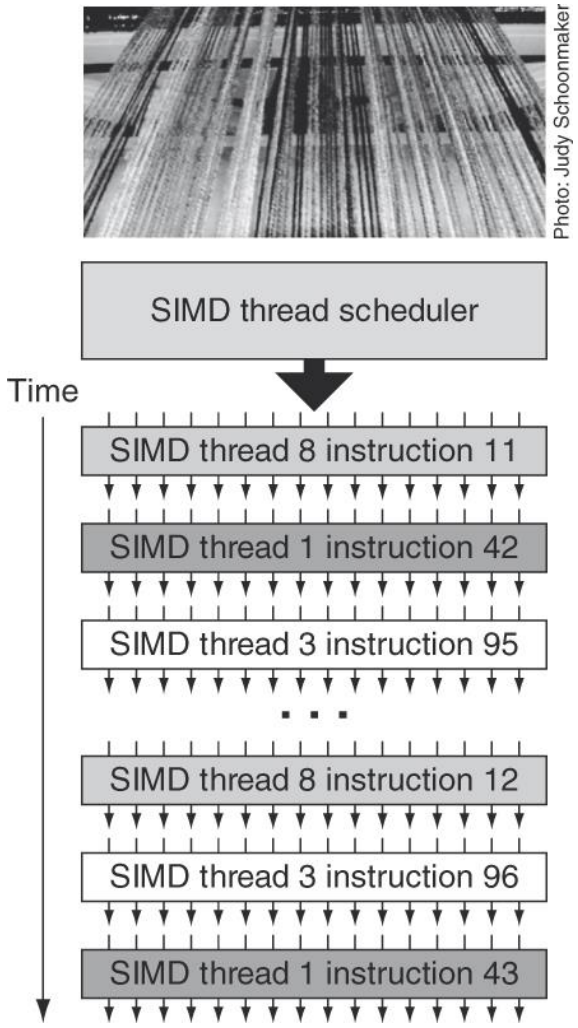
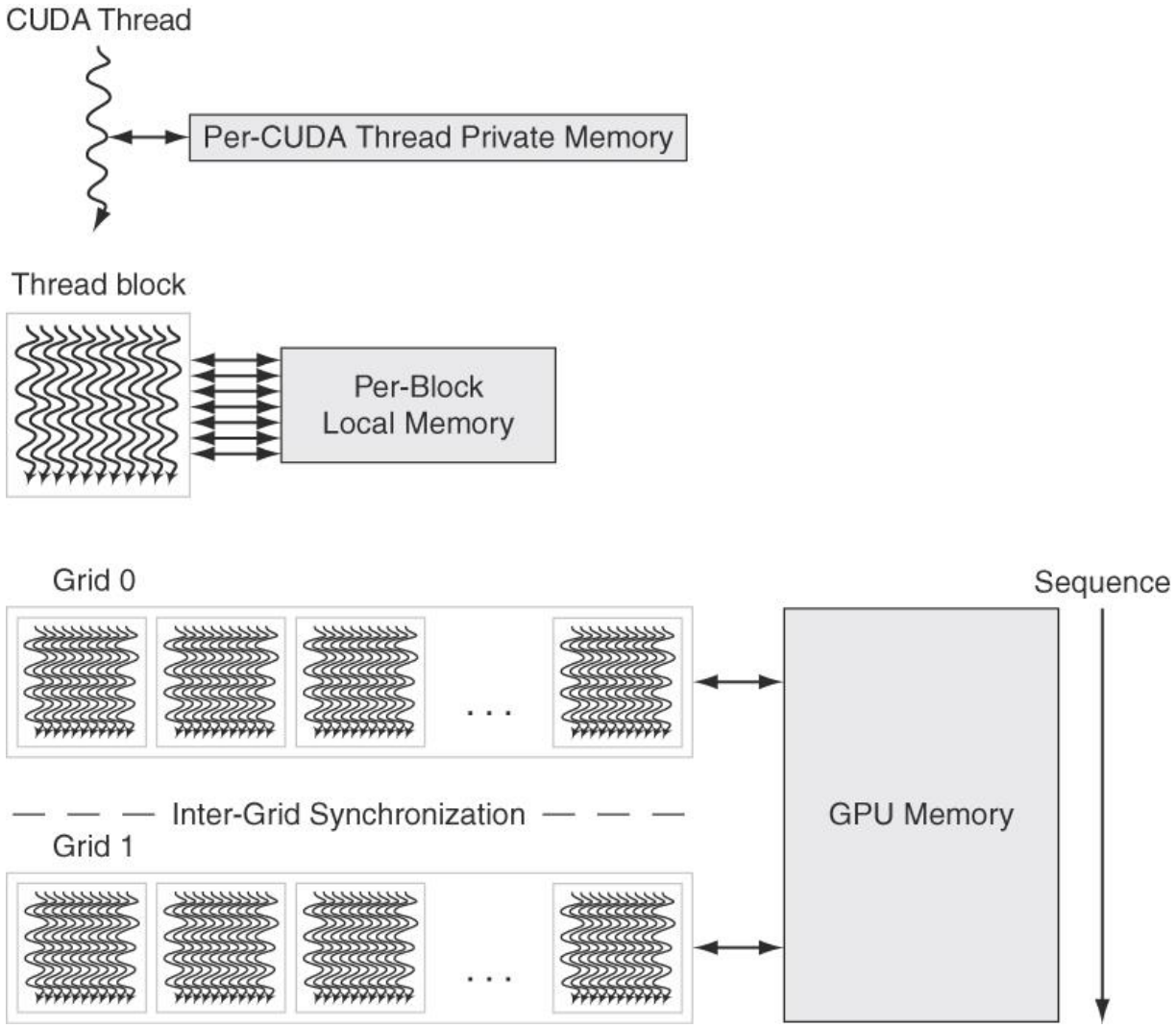


Figure 4.16 Scheduling of threads of SIMD instructions. The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD thread. Because threads of SIMD instructions are independent, the scheduler may select a different SIMD thread each time.

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 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


CUDA Thread
Per-CUDA Thread Private Memory
Thread block
Per-Block
Local Memory
Grid 0
Sequence
Inter-Grid Synchronization
GPU Memory
Grid 1

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 thread 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
  - A 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 (2048 x 16 = 32,768)

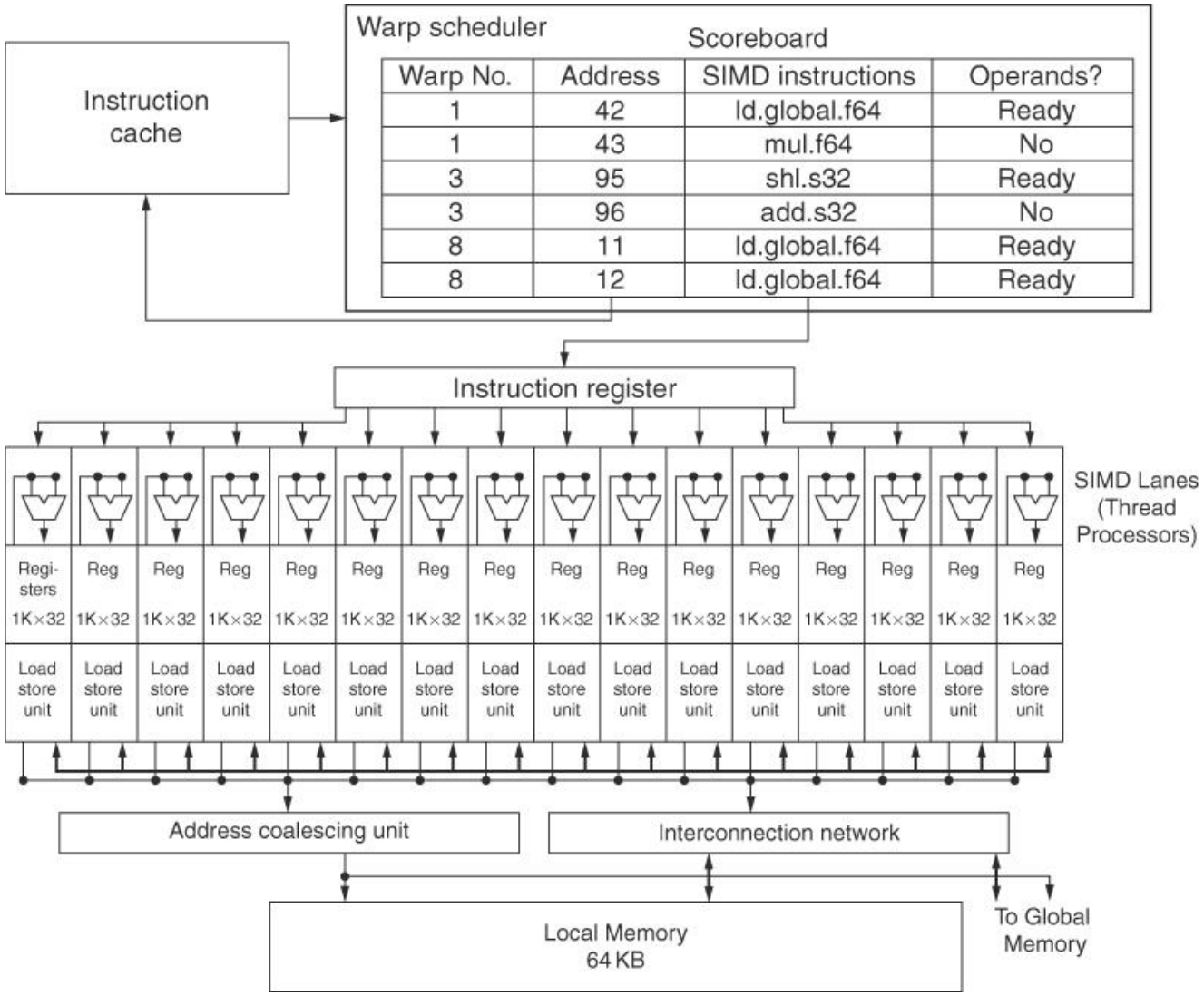


Figure 4.14 Simplified block diagram of a Multithreaded SIMD Processor with 16 SIMD lanes. The SIMD Thread Scheduler has, say, 48 independent threads of SIMD instructions that it schedules with a table of 48 PCs.

NVIDIA ISA -- PTX

- PTX → Parallel Thread Execution
- A PTX instruction describes the operation of a single CUDA thread!!
- Like x86 instructions PTX instructions translate to an internal format
  - X86 → translation done by hardware at execution time
  - PTX → translation done by software at compile time.
- The format of a PTX instruction: **opcode.type d,a,b,c**
  - d → destination operand
  - a, b, c → source operands

Type	.type Specifier
Untyped bits 8, 16, 32, and 64 bits	.b8, .b16, .b32, .b64
Unsigned integer 8, 16, 32, and 64 bits	.u8, .u16, .u32, .u64
Signed integer 8, 16, 32, and 64 bits	.s8, .s16, .s32, .s64
Floating Point 16, 32, and 64 bits	.f16, .f32, .f64

- Use virtual registers
- NVIDIA act as co-processors. Similar to I/O units

PTX arithmetic instructions

Group	Instruction	Example	Meaning	Comments
	arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
	add.type	add.f32 d, a, b	d = a + b;	
	sub.type	sub.f32 d, a, b	d = a – b;	
	mul.type	mul.f32 d, a, b	d = a * b;	
	mad.type	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
	div.type	div.f32 d, a, b	d = a / b;	multiple microinstructions
	rem.type	rem.u32 d, a, b	d = a % b;	integer remainder
Arithmetic	abs.type	abs.f32 d, a	d = |a|;	
	neg.type	neg.f32 d, a	d = 0 - a;	
	min.type	min.f32 d, a, b	d = (a < b)? a:b;	floating selects non-NaN
	max.type	max.f32 d, a, b	d = (a > b)? a:b;	floating selects non-NaN
	setp.cmp.type	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
	numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.type	mov.b32 d, a	d = a;	move
	selp.type	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	convert atype to dtype

PTX logical, memory access, and control flow

	logic.type = .pred,.b32, .b64			
Logical	and.type	and.b32 d, a, b	d = a & b;	
	or.type	or.b32 d, a, b	d = a | b;	
	xor.type	xor.b32 d, a, b	d = a ^ b;	
	not.type	not.b32 d, a, b	d = ~a;	one's complement
	cnot.type	cnot.b32 d, a, b	d = (a==0)? 1:0;	C logical not
	shl.type	shl.b32 d, a, b	d = a << b;	shift left
	shr.type	shr.s32 d, a, b	d = a >> b;	shift right
Memory Access	memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory space
	st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory space
	tex.nd.dtyp.btype	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup
	atom.spc.op.type	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b,	atomic { d = *a; *a = cop(*a, b); }	atomic read-modify-write operation
	atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32			
Control Flow	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

Parallel Thread Execution (PTX) example

- One CUDA thread, 8192 of these created!

```
shl.s32       R8, blockIdx, 9  ; Thread Block ID * Block size (512 or 2^9)
add.s32       R8, R8, threadIdx          ; R8 = i = my CUDA thread ID
ld.global.f64RD0, [X+R8]      ; RD0 = X[i]
ld.global.f64RD2, [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

- GPU branch hardware uses:
  - Internal masks
  - 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
```

Note: a thread has 64 vector components, each a 32 bit floating point

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 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

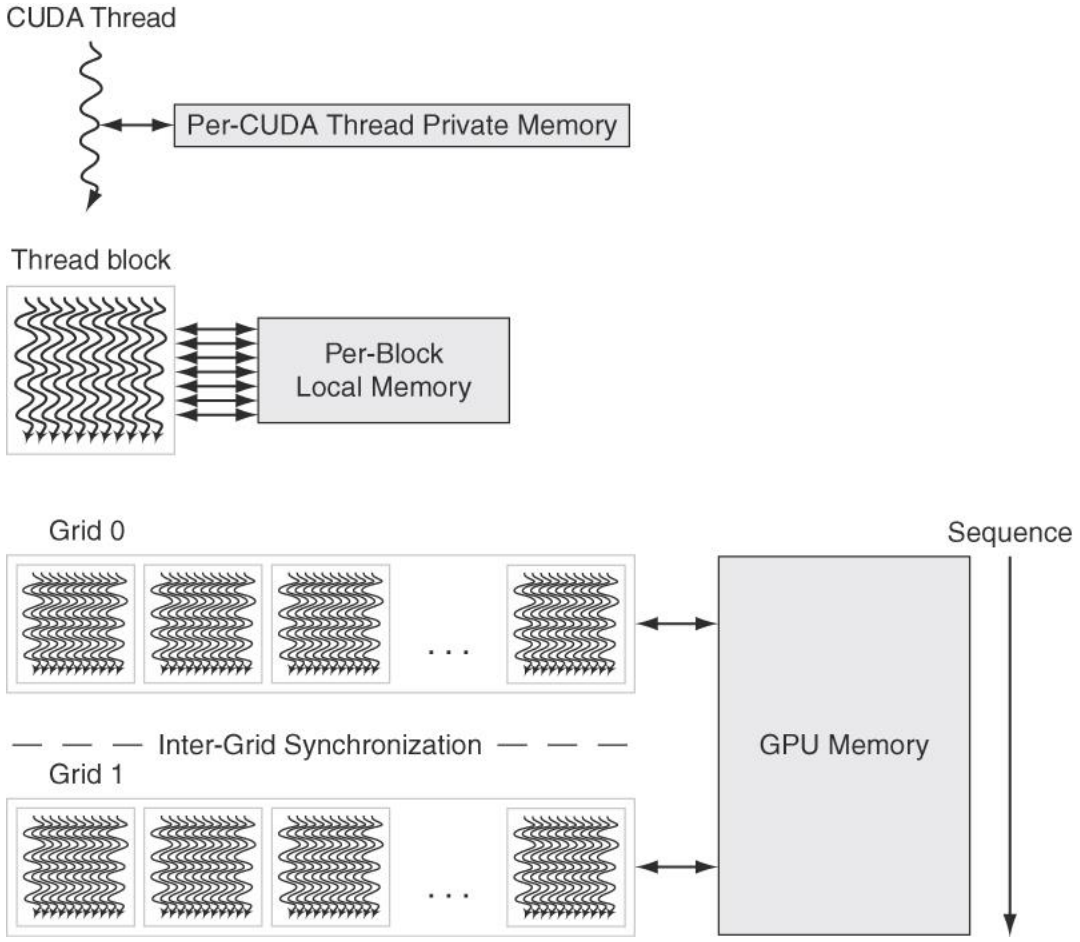


Figure 4.18 GPU Memory structures.

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

Fermi architecture innovations

- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - Two sets of 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 per SIMD processor 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 gen-

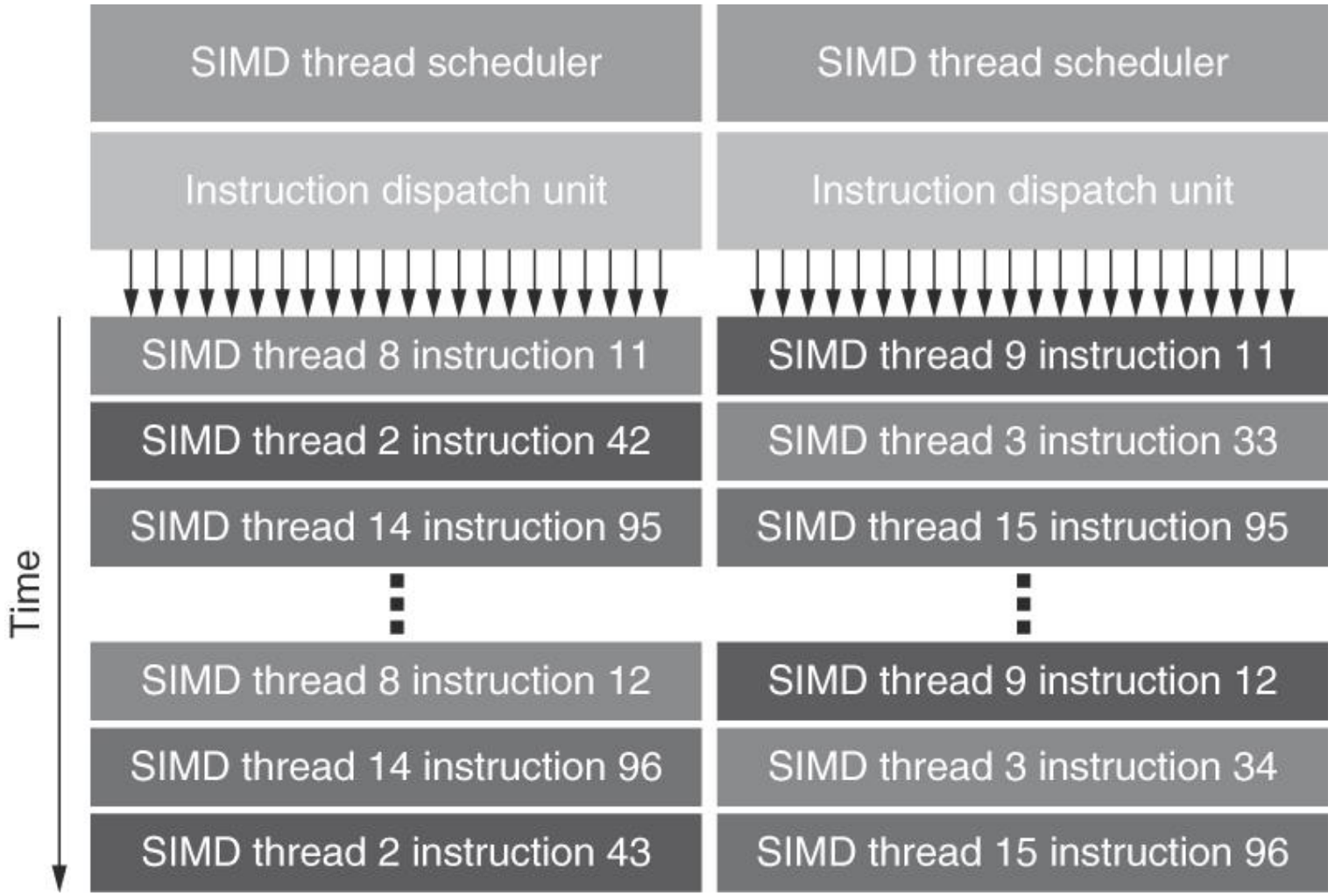
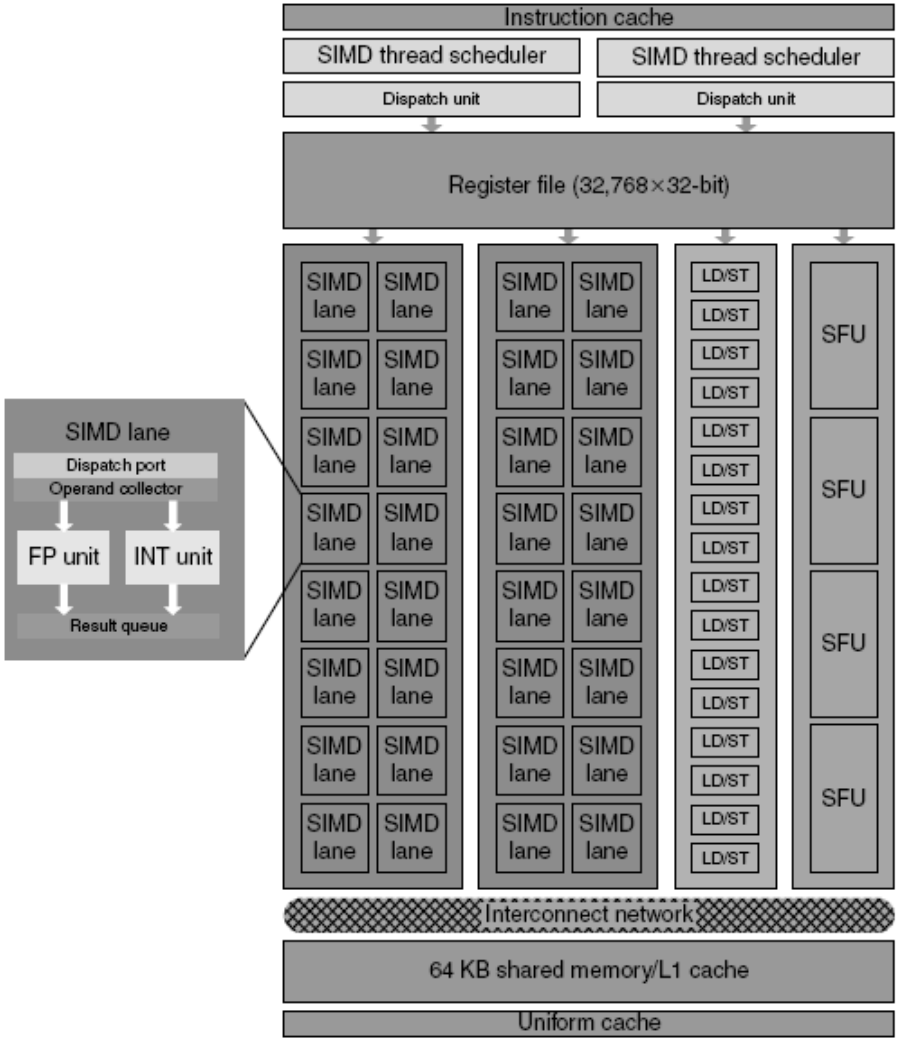


Figure 4.19 Block Diagram of Fermi's Dual SIMD Thread Scheduler.
Compare this design to the single SIMD Thread Design in Figure 4.16.

Fermi multithreaded SIMD processor



Fermi streaming multiprocessor (SM)

	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
Number of processing elements (cores or SMs)	4	30	15	7.5	3.8
Clock frequency (GHz)	3.2	1.3	1.4	0.41	0.44
Die size	263	576	520	2.2	2.0
Technology	Intel 45 nm	TSMC 65 nm	TSMC 40 nm	1.6	1.0
Power (chip, not module)	130	130	167	1.0	1.3
Transistors	700 M	1400 M	3030 M	2.0	4.4
Memory bandwidth (GBytes/sec)	32	141	177	4.4	5.5
Single-precision SIMD width	4	8	32	2.0	8.0
Double-precision SIMD width	2	1	16	0.5	8.0
Peak single-precision scalar FLOPS (GFLOP/Sec)	26	117	63	4.6	2.5
Peak single-precision SIMD FLOPS (GFLOP/Sec)	102	311 to 933	515 or 1344	3.0–9.1	6.6–13.1
(SP 1 add or multiply)	N.A.	(311)	(515)	(3.0)	(6.6)
(SP 1 instruction fused multiply-adds)	N.A.	(622)	(1344)	(6.1)	(13.1)
(Rare SP dual issue fused multiply-add and multiply)	N.A.	(933)	N.A.	(9.1)	--
Peak double-precision SIMD FLOPS (GFLOP/sec)	51	78	515	1.5	10.1

Figure 4.27 Intel Core i7-960, NVIDIA GTX 280, and GTX 480 specifications. The rightmost columns show the ratios of GTX 280 and GTX 480 to Core i7. For single-precision SIMD FLOPS on the GTX 280, the higher speed (933) comes from a very rare case of dual issuing of fused multiply-add and multiply. More reasonable is 622 for single fused multiply-adds. Although the case study is between the 280 and i7, we include the 480 to show its relationship to the 280 since it is described in this chapter. Note that these memory bandwidths are higher than in Figure 4.28 because these are DRAM pin bandwidths and those in Figure 4.28 are at the processors as measured by a benchmark program. (From Table 2 in Lee et al. [2010].)

Kernel	Application	SIMD	TLP	Characteristics
SGEMM (**SGEMM**)	Linear algebra	Regular	Across 2D tiles	Compute bound after tiling
Monte Carlo (**MC**)	Computational finance	Regular	Across paths	Compute bound
Convolution (**Conv**)	Image analysis	Regular	Across pixels	Compute bound; BW bound for small filters
FFT (**FFT**)	Signal processing	Regular	Across smaller FFTs	Compute bound or BW bound depending on size
SAXPY (**SAXPY**)	Dot product	Regular	Across vector	BW bound for large vectors
LBM (**LBM**)	Time migration	Regular	Across cells	BW bound
Constraint solver (**Solv**)	Rigid body physics	Gather/Scatter	Across constraints	Synchronization bound
SpMV (**SpMV**)	Sparse solver	Gather	Across non-zero	BW bound for typical large matrices
GJK (**GJK**)	Collision detection	Gather/Scatter	Across objects	Compute bound
Sort (**Sort**)	Database	Gather/Scatter	Across elements	Compute bound
Ray casting (**RC**)	Volume rendering	Gather	Across rays	4-8 MB first level working set; over 500 MB last level working set
Search (**Search**)	Database	Gather/Scatter	Across queries	Compute bound for small tree, BW bound at bottom of tree for large tree
Histogram (**Hist**)	Image analysis	Requires conflict detection	Across pixels	Reduction/synchronization bound

Figure 4.29 Throughput computing kernel characteristics (from Table 1 in Lee et al. [2010].) The name in parentheses identifies the benchmark name in this section. The authors suggest that code for both machines had equal optimization effort.

Kernel	Units	Core i7-960	GTX 280	GTX 280/ i7-960
SGEMM	GFLOP/sec	94	364	3.9
MC	Billion paths/sec	0.8	1.4	1.8
Conv	Million pixels/sec	1250	3500	2.8
FFT	GFLOP/sec	71.4	213	3.0
SAXPY	GBytes/sec	16.8	88.8	5.3
LBM	Million lookups/sec	85	426	5.0
Solv	Frames/sec	103	52	0.5
SpMV	GFLOP/sec	4.9	9.1	1.9
GJK	Frames/sec	67	1020	15.2
Sort	Million elements/sec	250	198	0.8
RC	Frames/sec	5	8.1	1.6
Search	Million queries/sec	50	90	1.8
Hist	Million pixels/sec	1517	2583	1.7
Bilat	Million pixels/sec	83	475	5.7

Figure 4.30 Raw and relative performance measured for the two platforms. In this study, SAXPY is just used as a measure of memory bandwidth, so the right unit is GBytes/sec and not GFLOP/sec. (Based on Table 3 in [Lee et al. 2010].)

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 examples 4 and 5

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

- No loop-carried dependence

```
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* x *i* + *b* (with constants *a* and *b*)
- An *n-D* array index is *affine* if it is affine in each dimension
- Assume:
  - Store to *a* x *i* + *b*, then
  - Load from *c* x *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* x *j* + *b*, load from *a* x *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!

Fallacies

1. GPUs suffer from being co-processors.

   The I/O device nature of GPUs creates a level of indirection between the compiler and the hardware and gives more flexibility to GPU architects who can try new innovations and drop them if not successful. For example, the Fermi architecture changed the hardware instruction set without disturbing the NVIDIA software stack:

   (1) from memory-oriented as x86 to register-oriented like MIPS;

   (2) from 32-bit to 64-bit addressing

2. One can get good performance without providing good memory bandwidth.

3. Add more threads to improve performance.

   If memory accesses of CUDA threads are scattered or not correlated the memory system will get progressively slower. The CUDA threads must enjoy locality of memory access.