ECG 700 Advanced Computer System Architecture Fall 2012

Lecture 6 – Data-Level Parallelism

Mei Yang

Adapted from David Patterson's slides on graduate computer architecture

Outline

- Introduction
- Vector Architecture
- Improvements of Vector Architectures
- SIMD Instruction Set Extensions for Multimedia
- Graphics Processing Units
- Detecting and Enhancing Loop-Level Parallelism
- Crosscutting Issues
- Putting It All Together: Telsa vs. Core i7

2

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

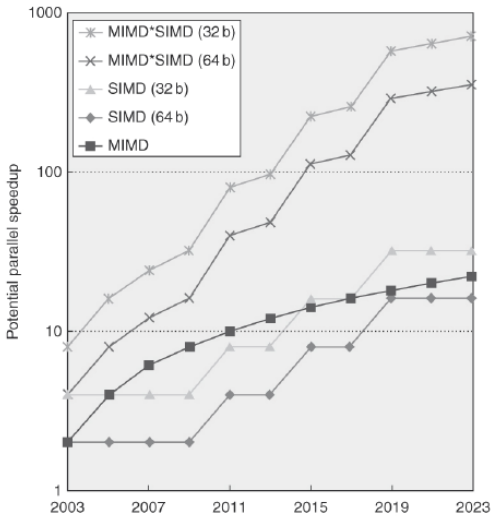
3

SIMD Parallelism

- Three variations of SIMD
  - 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!

4

SIMD vs. MIMD



5

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

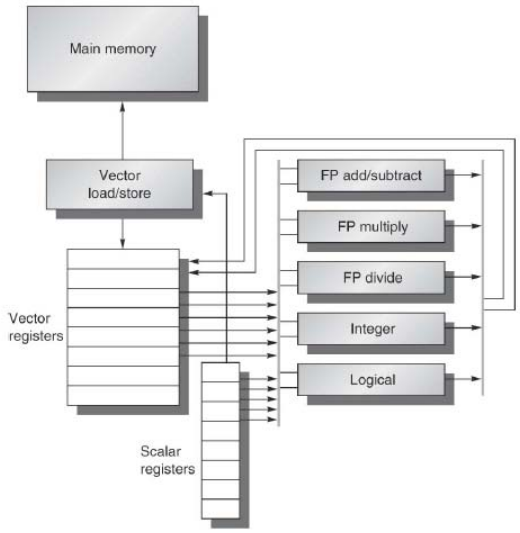
6

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

7

Basic Structure of VMIPS



8

VMIPS Instructions

- ADDVV.D: add two vectors
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address
- Two additional special-purpose registers
  - Vector-length register
  - Vector-mask register

9

Instruction	Operands	Function
ADDVV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2 (i.e., R1 + i × R2).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2).
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--VV.D S--VS.D	V1,V2 V1,F0	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1 MFC1	VLR,R1 R1,VLR	Move contents of R1 to vector-length register VL. Move the contents of vector-length register VL to R1.
MVTM MVFM	VM,F0 F0,VM	Move contents of F0 to vector-mask register VM. Move contents of vector-mask register VM to F0.

10

Example

- Example: DAXPY (Y = a x 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, F4, 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
      DSUBU    R20, R4, Rx    ; compute bound
      BNEZ     R20, Loop      ; check if done
```

Requires 6 instructions vs. almost 600 for MIPS

```
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
```

11

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

12

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* convoys executes in *m* chimes
  - For vector length of *n*, requires *m* x *n* clock cycles

13

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

14

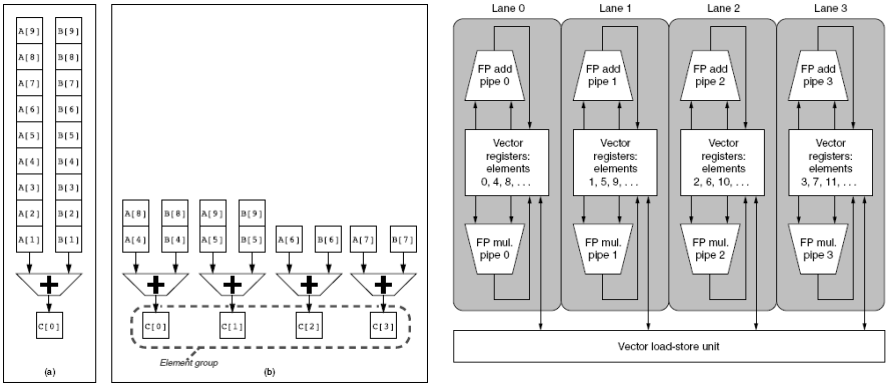
Challenges

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

15

Multiple Lanes

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

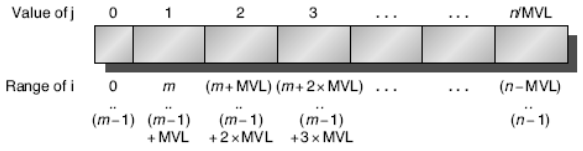


16

Vector Length Register

- 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*/
}
```



17

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:

```
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!

18

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?

19

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

20

Gather–Scatter

- Consider:

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

- 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[]]
```

21

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

22

SIMD Extensions

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

23

SIMD Extensions

Instruction category	Operands
Unsigned add/subtract	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Maximum/minimum	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Average	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Shift right/left	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Floating point	Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit

24

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

25

AVX Instructions

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

26

Example SIMD Code

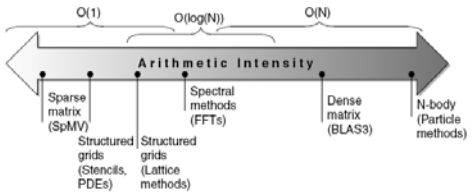
- Example DXPY:

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

27

Roofline Performance Model

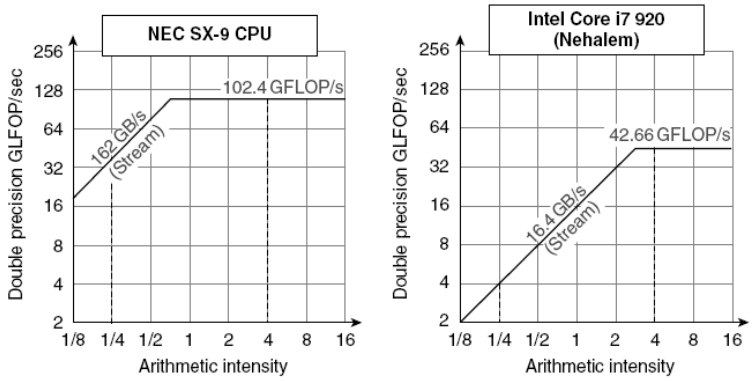
- Basic idea:
  - Plot peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
  - Floating-point operations per byte read



28

Examples

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



29

Graphical Processing Units

- Given the hardware invested to do graphics well, how can be supplement it 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
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model is "Single Instruction Multiple Thread"

30

Threads and Blocks

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

- GPU hardware handles thread management, not applications or OS

31

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

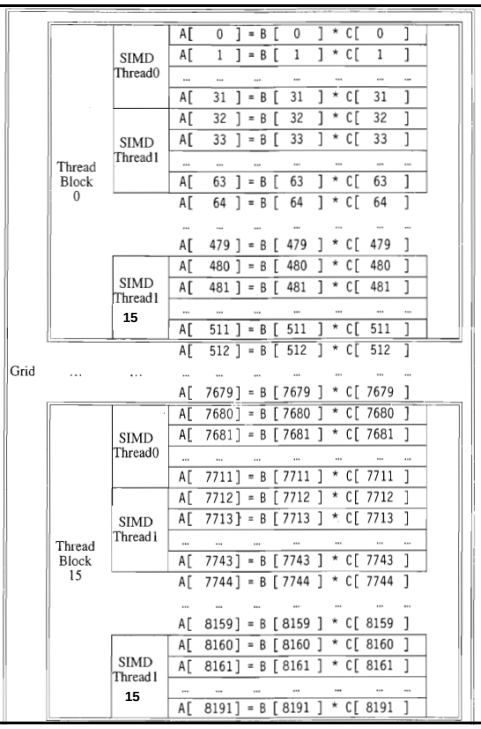
32

Example

- Multiply two vectors of length 8192
  - Code that works over all elements is the grid
  - Thread blocks break this down into manageable sizes
    - 512 threads per block
  - SIMD instruction executes 32 elements at a time
  - Thus grid size = 16 blocks
  - Block is analogous to a strip-mined vector loop with vector length of 32
  - Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
  - Current-generation GPUs (Fermi) have 7-15 multithreaded SIMD processors

33

Example



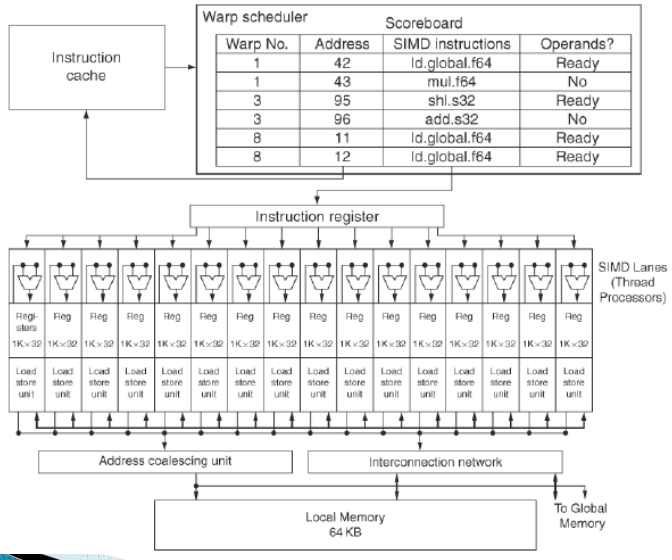
34

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:
  - 16 SIMD lanes
  - Wide and shallow compared to vector processors

35

Multithreaded SIMD Processor



36

Example

- NVIDIA GPU has 32,768 registers
  - Divided into lanes
  - Each SIMD thread is limited to 64 vector 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

37

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
- Format of a PTX instruction
  - Opcode.type d, a, b, c

38

PTX Thread Instructions

Group	Instruction	Example	Meaning	Comments
Arithmetic	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
	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
Special Function	special .type = .f32 (some .f64)			
	rcp.type	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.type	sqrt.f32 d, a	d = sqrt(a);	square root
	rsqrt.type	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.type	sin.f32 d, a	d = sin(a);	sine
	cos.type	cos.f32 d, a	d = cos(a);	cosine
	lg2.type	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.type	ex2.f32 d, a	d = 2 ** a;	binary exponential

39

PTX Thread Instructions (cont'd)

Logical	logic.type = .pred,.b32, .b64			
	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

40

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])
```

- Totally 8192 CUDA threads, using unique address for each element
- No incrementing or branch code

41

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

42

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
```

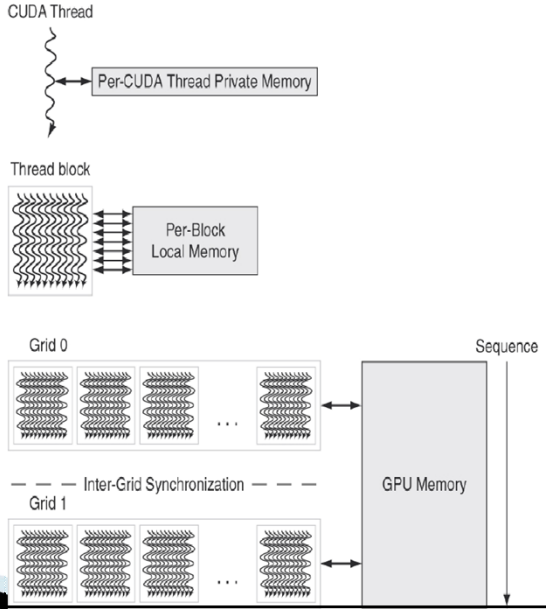
43

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

44

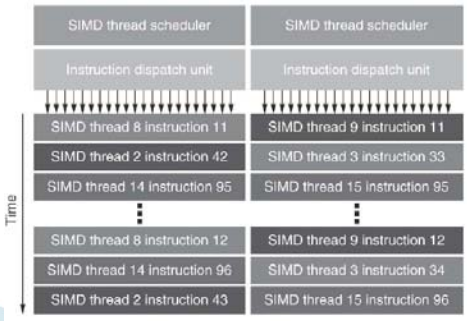
GPU Memory



45

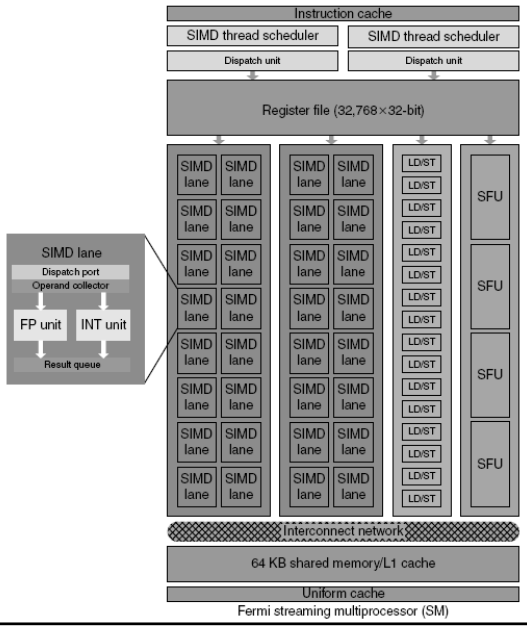
Fermi Architecture Innovations

- 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



46

Fermi Architecture Innovations (cont'd)



Fermi streaming multiprocessor (SM)

47

Fermi Architecture Innovations (cont'd)

- Fast double precision
  - Single precision to double precision: 2:1
- Caches for GPU memory
  - L1 data cache and L1 instruction cache for each multithreaded SIMD processor
  - 768KB L2 cache shared by all multithreaded SIMD processors
- 64-bit addressing and unified address space
- Error correcting codes
- Faster context switching
  - Switching in 25 ms
- Faster atomic instructions

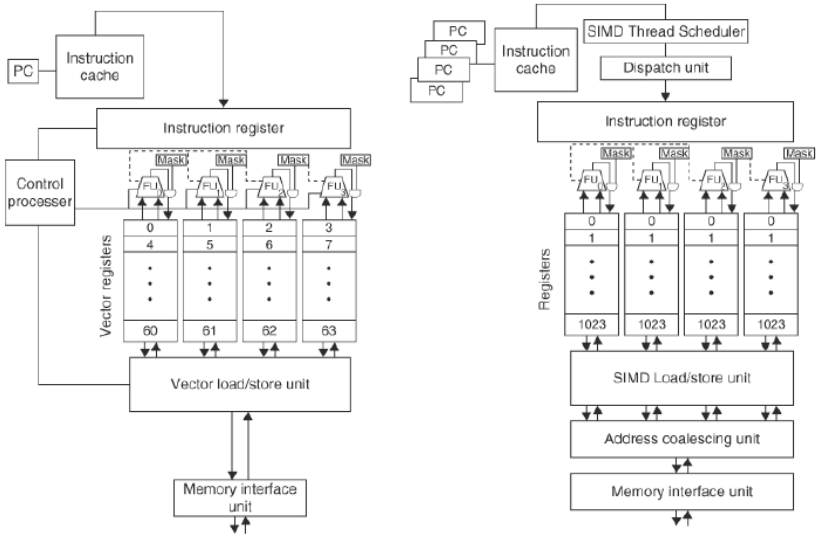
48

Similarities and Differences between Vector Architectures and GPUs

- Processor vise:
  - VA is SIMD
  - Multiple multithreaded SIMD processors act as independent MIMD cores
- Registers
  - VMIP register holds entire vectors
  - A single vector is distributed across the registers of all SIMD lanes

49

VA vs. GPU



50

VA vs. GPU

- # of lanes
- Memory accesses
- Conditional branch instructions
- Control processor
- Scalar processor

51

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

52

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

53

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];
```

54

Loop-Level Parallelism

- Example 4:

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

- Example 5:

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

55

Finding dependencies

- Assume indices are affine:
  - a x i + b (i is loop index)
- 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 x j + b = c x k + d

56

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;
}
```

57

Finding dependencies

- Example 2:

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

- Watch for antidependencies and output dependencies

58

Finding dependencies

- Example 2:

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

- Watch for antidependencies and output dependencies

59

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!

60