ΕΠΛ605

Κεφ. 4: Data-Level Parallelism (DLP) in Vector, SIMD, and GPU Architectures

Slides based on notes from book web page Computer Architecture A Quantitative Approach, Fifth Edition

Parallelism Classification

- SISD: single instruction single data
  - Single PC (single control flow sequencer)
  - one instruction operates on one piece of data
  - Traditional processors
- SIMD: single instruction multiple data <= Today's Lecture
  - Single PC
  - one instruction operates on multiple data pieces
  - Vector, Media extensions, GPUs
- MIMD: multiple instructions multiple data
  - Multiple PCs (multiple sequencers)
  - Multiple instructions each operate on one data
  - Multicores, multiprocessors
- Combinations

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)

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 Vector Architecture and Microarchitecture

- Loosely based on Cray-1
- Vector Register File: 8 Vector registers
  - Each register holds a 64-element, 64 bits/element vector
- Scalar registers
  - 32 general-purpose registers
  - 32 floating-point registers

- Microarchitecture
  - Vector Register file has 16 read ports and 8 write ports
  - Vector functional units
    - Fully pipelined
    - Data and control hazards are detected (dynamically)
  - Vector load-store unit
    - Fully pipelined
    - One word per clock cycle after initial latency
  - Vector Processors no L1 D$. The VRF acts as data cache
    - 8 x 64 x 8 B = 2^{12} B = 4KB

DAXPY

```
double A[64],B[64],a;
int i;
…
…
for(i=0;i<64;++i)
        B[i] = B[i] + A[i]*a
```

Scalar Code

```
R1 <= A
R2 <= B
FR3 <= a
R4 <= A + 64 * 8
L1:
ld.d FR4,0(R1)
mul.d FR5,FR4,FR3
ld.d FR6,0(R2)
add.d FR7,FR6,FR5
sd.d FR7,0(R2)
add R1,R1,8
add R2,R2,8
sub R8,R1,R4
bnz R8, L1
```

For 64 elements, 9x64= 576 instructions

Vector Instructions

- ADDVV.D: add two vectors (64 elements)
- MULVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address

- Example: DAXPY

```
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 vs. almost 600 with non-vector

Vector Execution Time

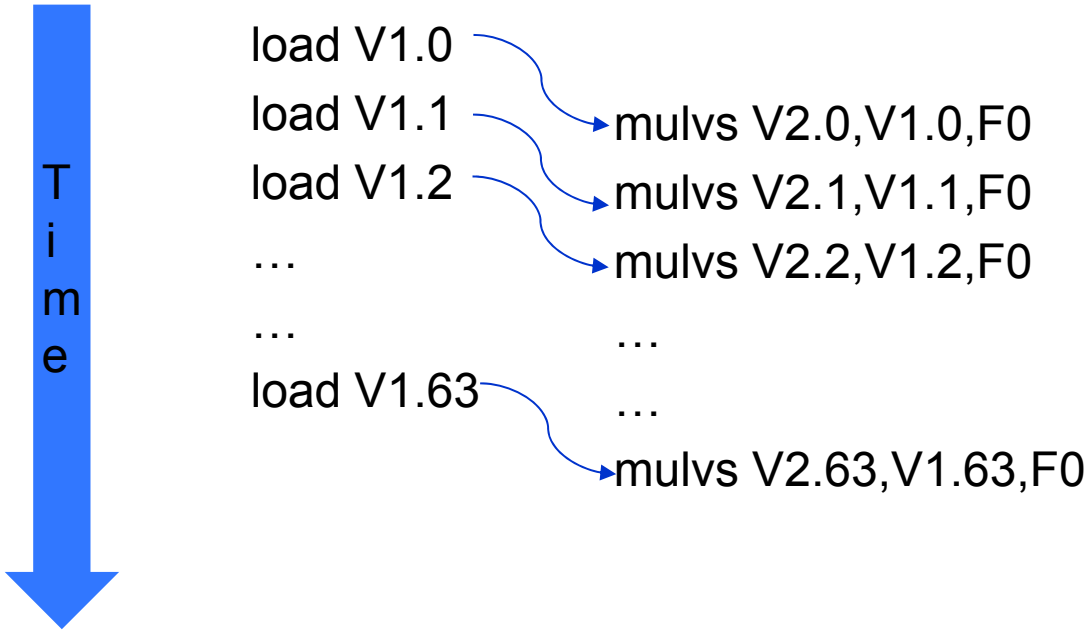
- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- Assume vector functional units consume one element per clock cycle
  - Execution time is approximately the vector length
- *Convey*
  - Set of vector instructions that could potentially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

Chimes

- Sequences with read-after-write dependency hazards can be in the same convey 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* approximately requires *m* x *n* clock cycles (without considering startup latency and chaning)
    - n=64, m=3 =>192 cycles

Convey and Chaining

```
LV        V1,Rx      ; load vector X
MULVS.D   V2,V1,F0   ; vector-scalar multiply
```



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

Why not 1 convoy?

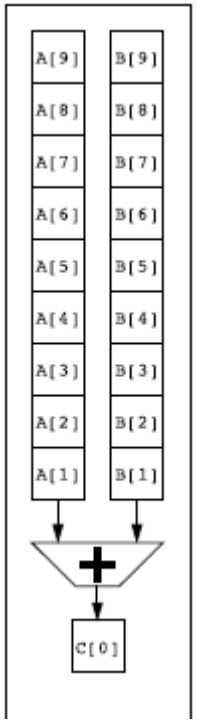
1. LV MULVS.D LV ADDVV.D SV

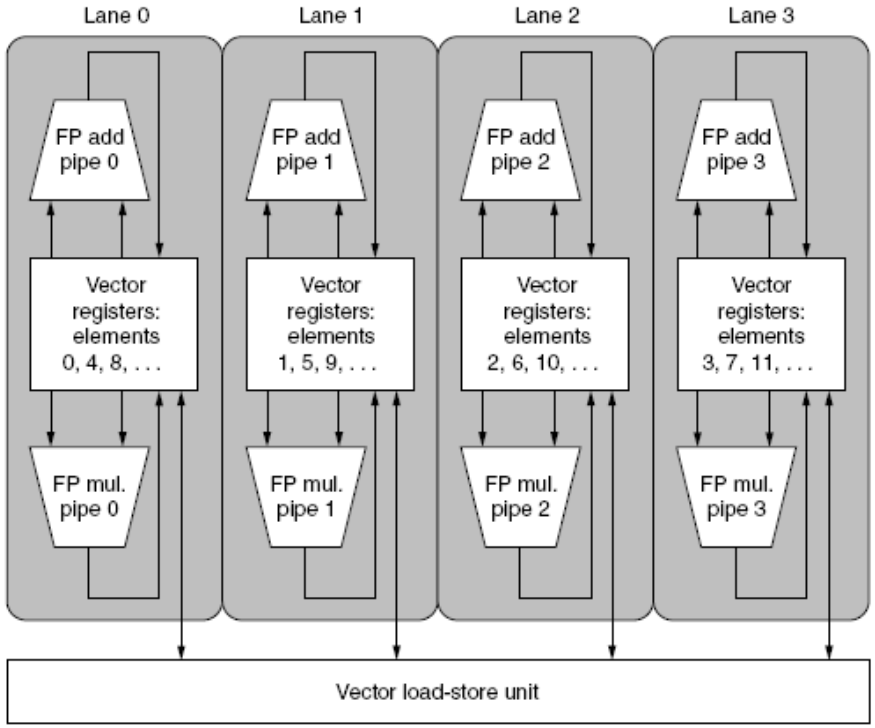
Challenges

- 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 (parallelism)
  - IF statements in vector code
  - Memory system optimizations to support vector processors
  - Non-64 wide vectors
  - Multiple dimensional matrices
  - Sparse matrices
  - Programming a vector computer

Multiple Lanes

- Element i^{th} of vector register *A* is "paired" to element i^{th} of vector register *B*
  - Allows for multiple hardware lanes each with separate functional unit (improve parallelism)
  - E.g. with four lanes vector operations x4 faster





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
```

- Reminds a little of predication
- GFLOPS rate decreases!

Memory Banks

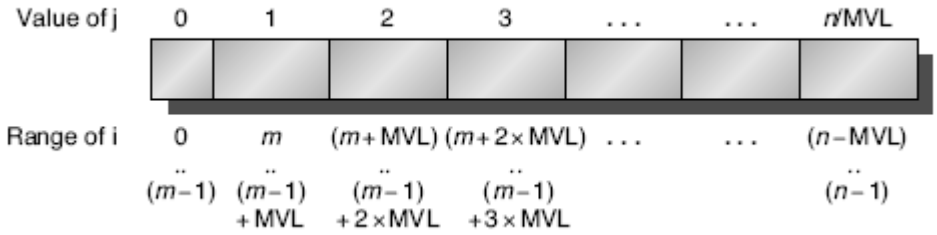
- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words
- Example:
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - How many memory banks needed?

Vector Length Register

- Vector length not known at compile time?
- Know the Maximum Vector Length (MVL), eg 64
- 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*/
}
```

Which part of loop can be vectorized?

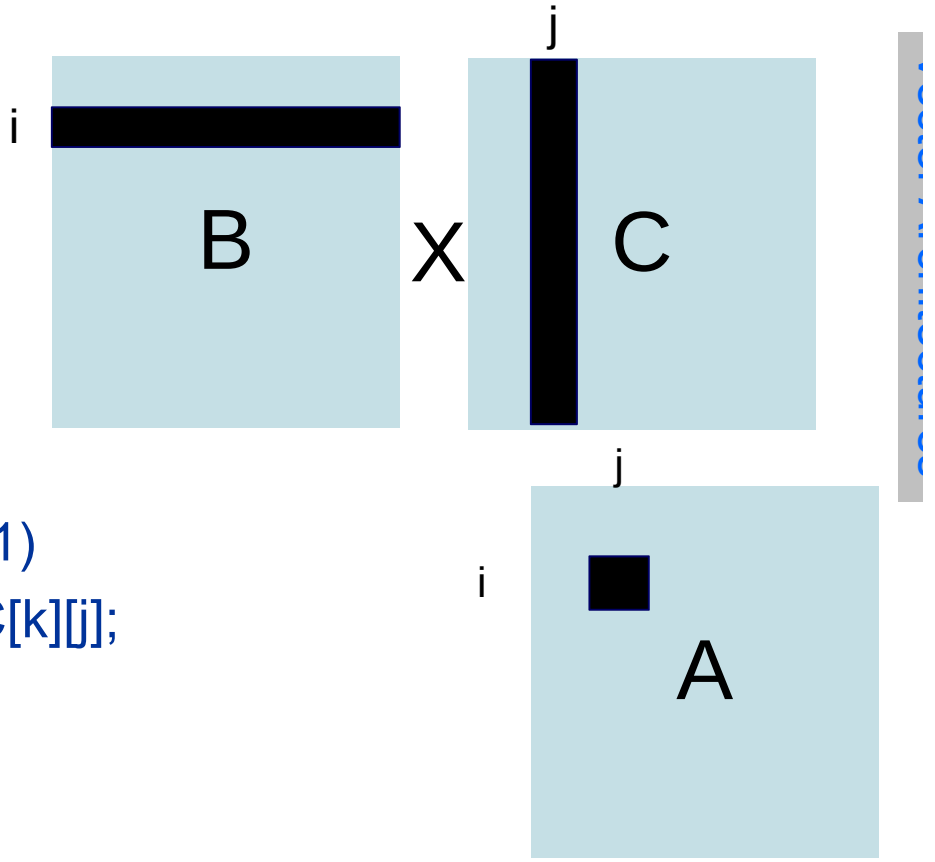


Stride

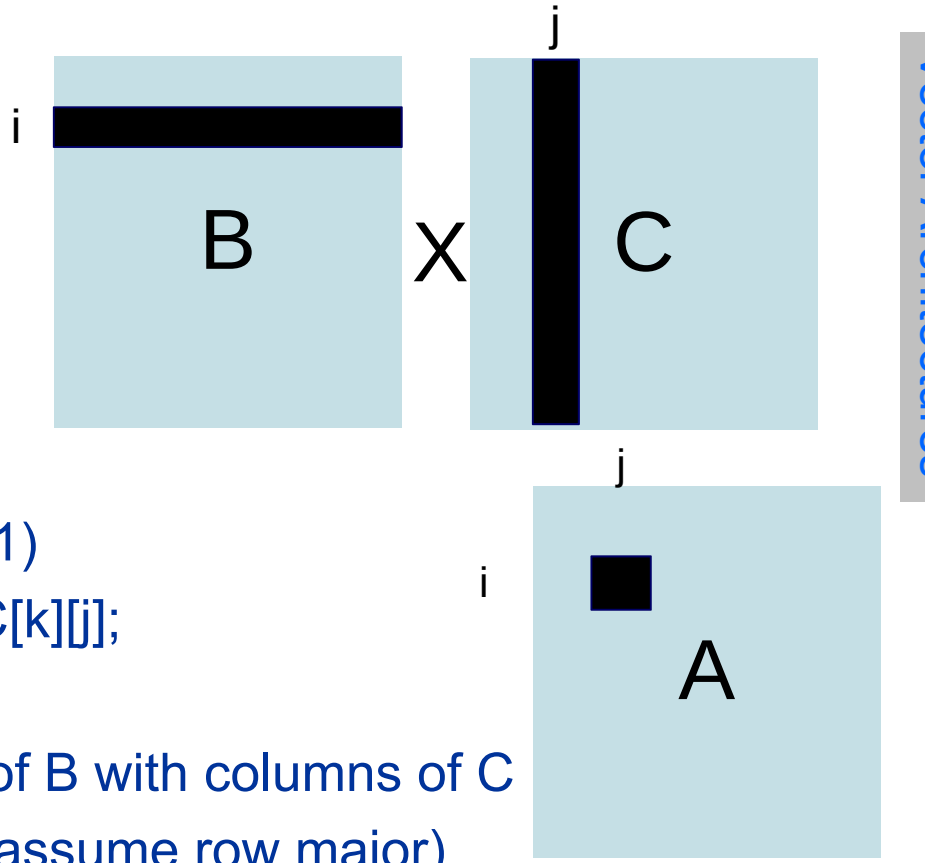
- Consider (matrix multiplication):

```
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] * C[k][j];
        }
```

Which part of loop can be vectorized?



Stride



- Consider (matrix multiplication):

```
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] * C[k][j];
        }
```

- Must vectorize multiplication of rows of B with columns of C
- Use unit stride to fetch one vector (B assume row major)
- Use *non-unit stride to fetch from memory C*
  - *Special memory operation defines load vector, starting address and stride*
- Bank conflict (stall) occurs when the same bank is hit faster than **bank busy time**:
  - Bank busy time: time needed between able to initiate consecutive accesses

Stride at granularity of 8

- B0: 0, 64, 128,…
- B1: 8, 72, 136,…
- B2: 16, 80, 144,…
- B3: 24, 88, 152,…
- B4: 32, 96, 160,…
- B5: 40, 104, 168,…
- B6: 48, 112, 176,…
- B7: 56, 120, 184,…

Stride	Number of accesses back to same bank	Wait
8	8	0
16	4	2
32	2	4
64	1	5
40	?	?

- With 8 banks, each bank 8 bytes, 6 cycle busy, 1 access initiated per cycle:
  - stride access = 1 double (8bytes), by the time go back (8 cycles) to same bank previous access is done
  - stride access = 8 doubles (64 bytes), every access goes to same bank; consecutive accesses need to wait previous busy time

Scatter-Gather

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

Programming Vec. Architectures

- Compilers can provide feedback to programmers
- Programmers can provide hints to compiler
- Following 1991 analysis

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

SIMD Extensions

- Media applications operate on data types narrower than the native word size
  - Hardware support: disconnect carry chains to “partition” a 64-bit adder into 8 8-bit adders
- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code (no VL)
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers
- Historically SIMD extensions have been improving providing more and more vector operations capabilities MMX, SSE, SSE2, SSE3, SSSE3, SSE4, AVX, AVX2, AVX512

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 - AVX(2010)
    - Four 64-bit integer/fp ops
  - Operands must be consecutive and aligned memory locations

Example SIMD Code using 4-operand SIMD instructions

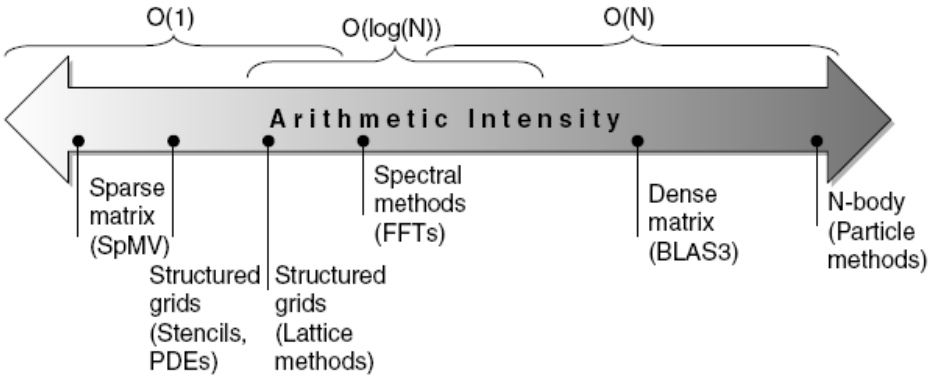
- Example DXPY:

```
        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] F4,F5,F6,F7
        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
```

Scalar: 576 instructions, Vector: 6 instructions, SIMD: 144 instructions

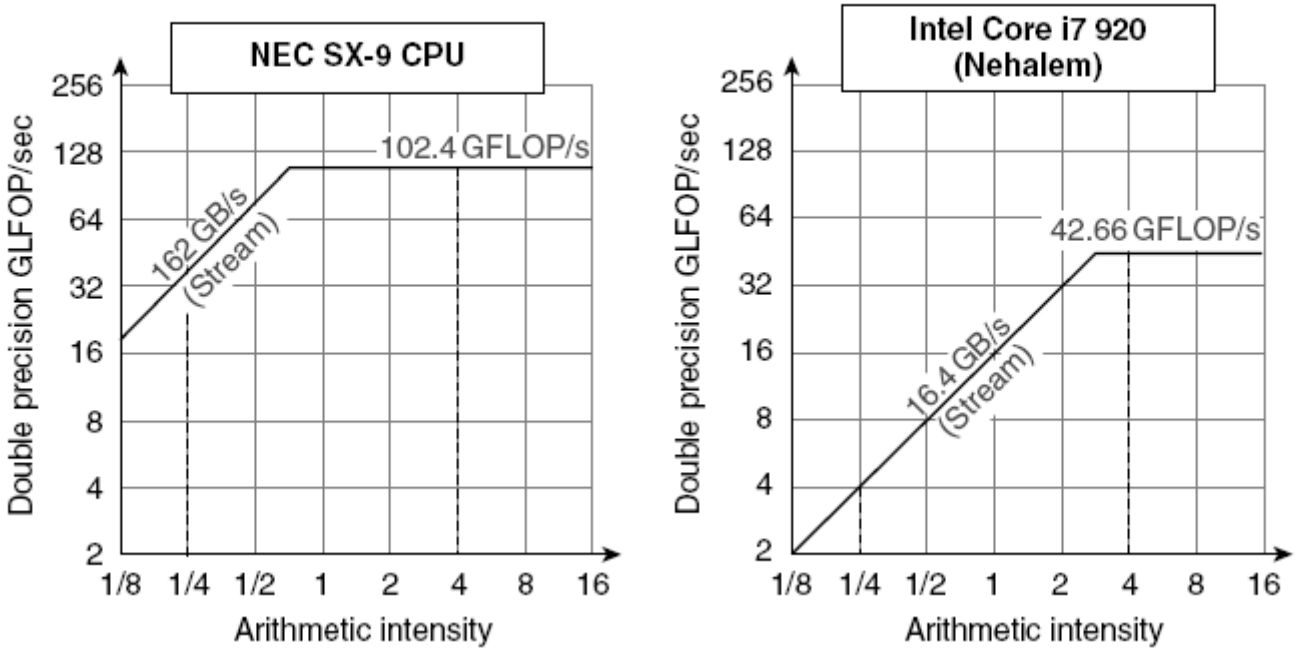
Roofline Performance Model

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



Examples

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



Graphical Processing Units

- Given the hardware invested to do graphics well, how can 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
  - Programming model is “Single Instruction Multiple Thread” (SIMT)

DAXPY in CUDA

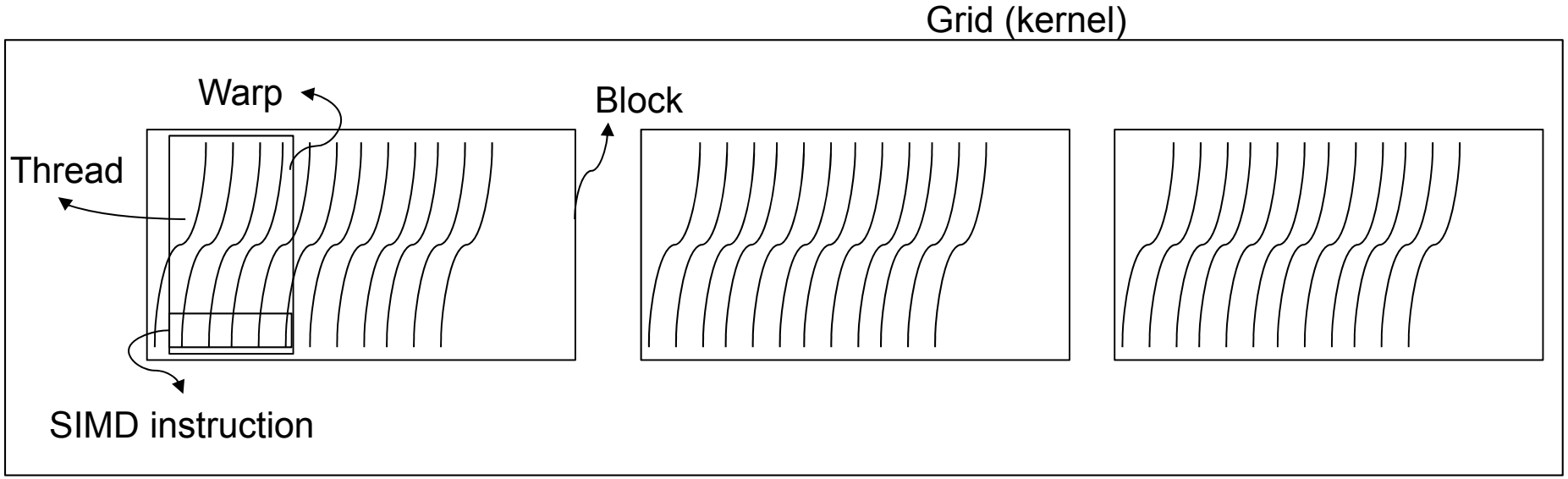
```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
daxpy<<<nblocks, 256>>>(n, 2.0, x, y);


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

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



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
    - Memory gap wider today than 1970s
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Example

- Multiply two vectors of length 8192
  - Code that works over all elements is the grid
  - Thread blocks break this down into manageable sizes
    - 512 threads per block
    - Influenced by implementation: memory latency, number of lanes, depth of pipeline, scheduler
  - Thus grid size = 16 blocks
  - Block is assigned to a *multithreaded SIMD processor (Streaming Multiprocessor - SM)* by the *thread block scheduler - TBS*
  - Current-generation GPUs many (4-15) multithreaded SIMD processors

Terminology

- Thread block scheduler (TBS) schedules blocks to SIMD processors
- Within each SIMD processor:
  - Many (2x16) SIMD lanes
  - Wide and shallow compared to vector processors
- *Threads of SIMD instructions*
  - Each has its own PC
  - Thread scheduler (TS) uses scoreboard to dispatch
  - No data dependencies between thread of SIMDs!
  - SIMD processor keeps track of up to 48 threads of SIMD instructions
    - Hides memory latency with multithreading
  - SIMD instruction executes on 32 elements at a time

Example

- GPU with 32768x32b registers/SM
  - RF Divided into lanes
  - Fermi has 16 physical SIMD lanes, each containing 2048x32b registers or 1024x64b registers
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
  - Each SIMD thread is limited to 64 registers
    - Software does spilling

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 (DAXPY):

```
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.f64 RD0, [X+R8]         ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]         ; RD2 = Y[i]
mul.f64 RD0, RD0, RD4             ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 RD0, 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

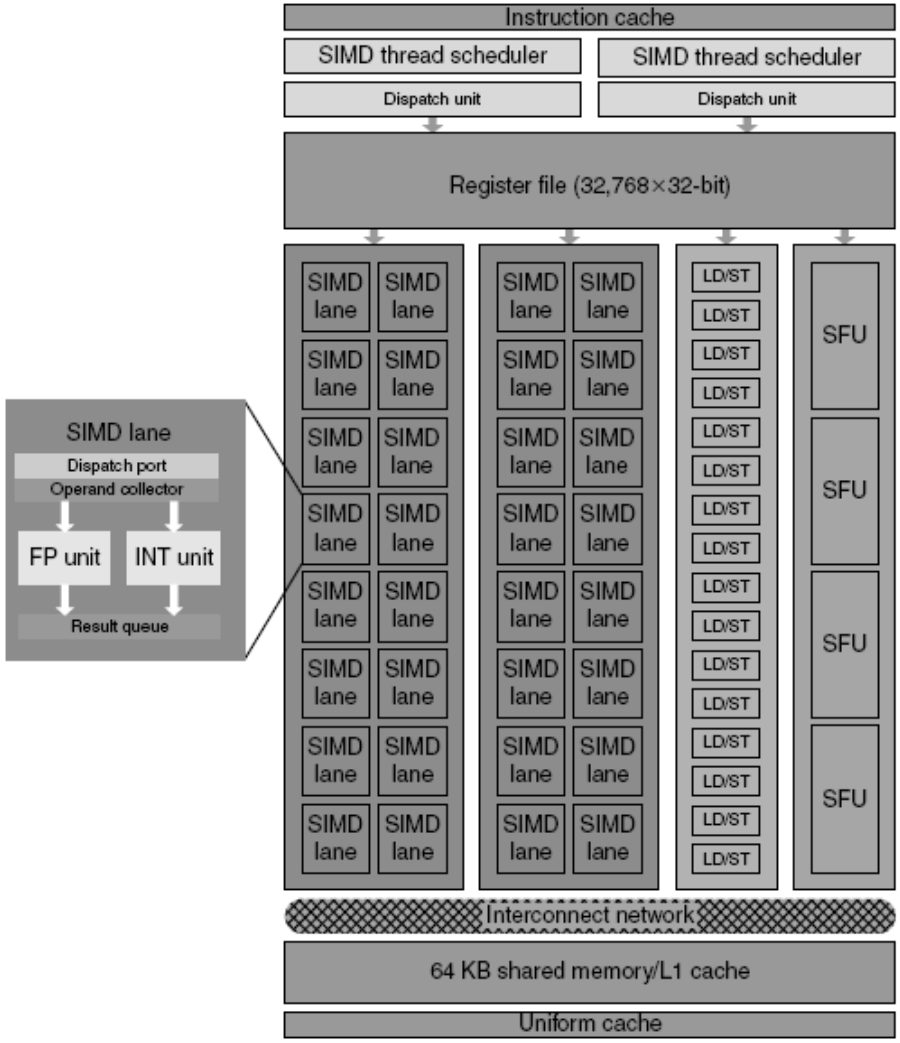
```
if (X[i] != 0)
        X[i] = X[i] – Y[i];
else X[i] = Z[i];
```

```
    ld.global.f64     RD0, [X+R8]              ; RD0 = X[i] – assume R8 is known
    setp.neq.s32      P1, RD0, #0              ; P1 is predicate register 1
    @!P1, bra         ELSE1, *Push             ; Push old mask, set new mask bits
                                               ; if P1 false, go to ELSE1
    ld.global.f64     RD2, [Y+R8]              ; RD2 = Y[i]
    sub.f64           RD0, RD0, RD2            ; Difference in RD0
    st.global.f64     [X+R8], RD0              ; X[i] = RD0
    @P1, bra          ENDIF1, *Comp            ; complement mask bits
                                               ; if P1 true, go to ENDIF1
ELSE1:              ld.global.f64 RD0, [Z+R8]  ; RD0 = Z[i]
                    st.global.f64 [X+R8], RD0  ; X[i] = RD0
ENDIF1:  <next instruction>, *Pop        ; pop to restore old mask
```

NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - “Private memory”
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write GPU memory
- Private, Local, Shared

Fermi Multithreaded SIMD Proc.



Fermi streaming multiprocessor (SM)

Fermi Architecture Innovations

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

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:

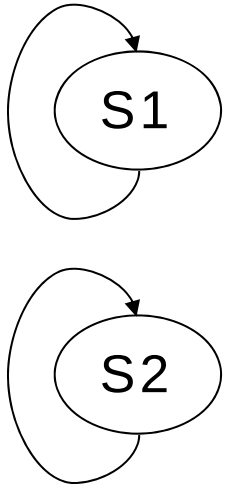
```
for (i=999; i>=0; i=i-1)
        x[i] = x[i] + s;
```

- No loop-carried dependence => can parallelize

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];
```

```
A[0] <=A[0] + B[0]
B[1] <=C[0]+D[0]

A[1]<= A[1]+B[1]
B[2]<=C[1]+D[1]

A[2]<=A[2]+B[2]
```

```
A[0] <=A[0] + B[0]

B[1] <=C[0]+D[0]
A[1]<= A[1]+B[1]

B[2]<=C[1]+D[1]
A[2]<=A[2]+B[2]
```

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];
}
```

Finding dependencies

- Assume indices are affine:
  - $a \times i + b$ (i is loop index)
- 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;
}
```

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!

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

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