Data-Level Parallelism in Vector and GPU Architectures

Muhamed Mudawar

Computer Engineering Department

King Fahd University of Petroleum and Minerals

Introduction

- **SIMD architectures can exploit significant data-level parallelism for:**
  - **matrix-oriented scientific computing**
  - **media-oriented image and sound processors**
- **SIMD is more energy efficient than MIMD**
  - **Only needs to fetch one instruction per data operation**
  - **Makes SIMD attractive for personal mobile devices**
- **SIMD allows programmer to continue to think sequentially**

SIMD Parallelism

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

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

Vector Architectures

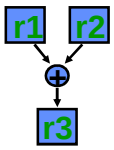
- Basic idea:
  - Read sets of data elements into "vector registers"
  - Operate on those registers
  - Disperse the results back into memory

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

Vector Processing

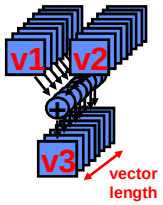
- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"

SCALAR
(1 operation)




```
add r3, r1, r2
```

VECTOR
(N operations)




```
addv v3, v1, v2
```


Vector Supercomputers

Idealized by Cray-1, 1976:

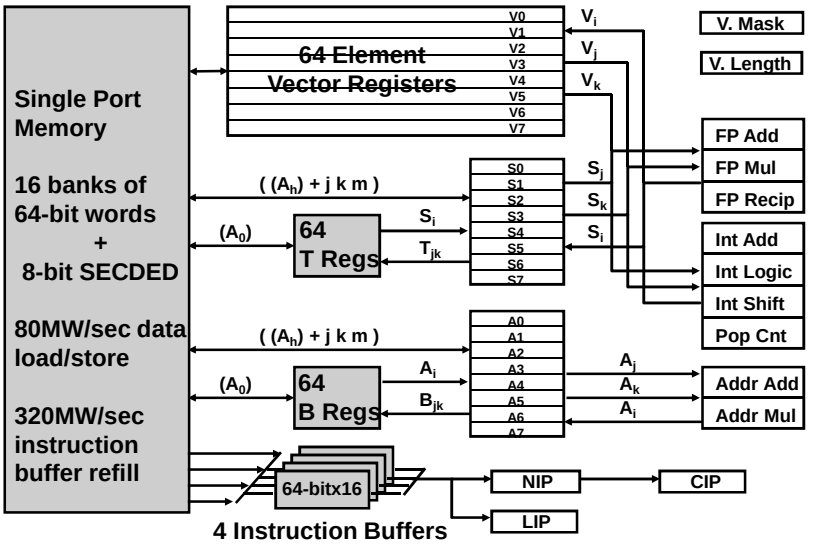
Scalar Unit + Vector Extensions

- Load/Store Architecture
- Vector Registers
- Vector Instructions
- Hardwired Control
- Highly Pipelined Functional Units
- Interleaved Memory System
- No Data Caches
- No Virtual Memory

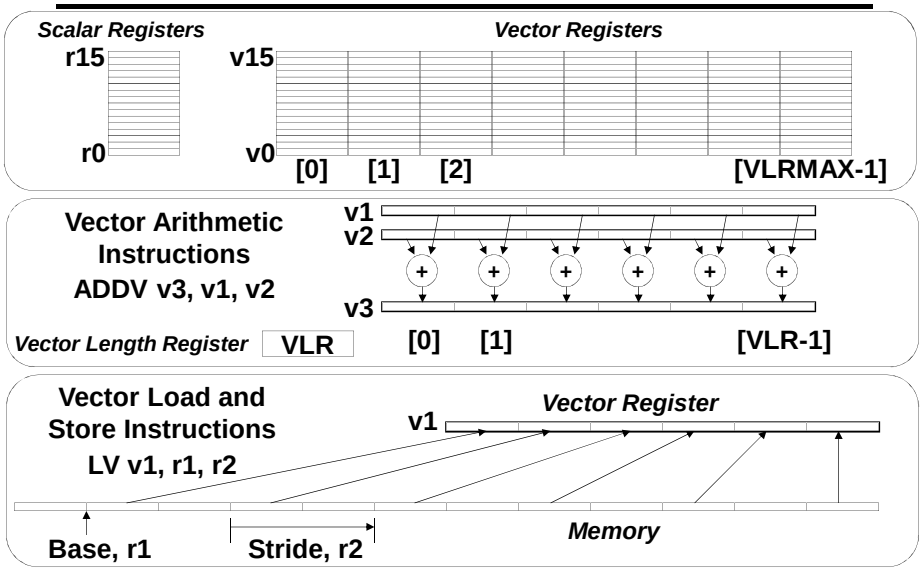
Cray-1 (1976)

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

Cray-1 (1976)



Vector Programming Model



Vector Instructions

Instr.	Operands	Operation	Comment
ADDV	V1,V2,V3	V1=V2+V3	vector + vector
ADDSV	V1,F0,V2	V1=F0+V2	scalar + vector
MULTV	V1,V2,V3	V1=V2xV3	vector x vector
MULSV	V1,F0,V2	V1=F0xV2	scalar x vector
LV	V1,R1	V1=M[R1..R1+63]	load, stride=1
LVWS	V1,R1,R2	V1=M[R1..R1+63*R2]	load, stride=R2
LVI	V1,R1,V2	V1=M[R1+V2i,i=0..63]	load, indexed
CeqV	VM,V1,V2	VMASKi = (V1i=V2i)?	comp. setmask
MOV	VLR,R1	Vec. Len. Reg. = R1	set vector length
MOV	VM,R1	Vec. Mask = R1	set vector mask

Properties of Vector Processors

- Each result independent of previous result
  - Long pipeline, compiler ensures no dependencies
  - High clock rate
- Vector instructions access memory with known pattern
  - Highly interleaved memory
  - Amortize memory latency of over 64 elements
  - No (data) caches required! (Do use instruction cache)
- Reduces branches and branch problems in pipelines
- Single vector instruction implies lots of work (loop)
  - Fewer instruction fetches

Vector Code Example

```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  ADDIU R1, 8
  ADDIU R2, 8
  ADDIU R3, 8
  SUBIU R4, 1
  BNEZ R4, loop
```

```
# Vector Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV V3, V1, V2
  SV V3, R3
```

Vector Instruction Set Advantages

- **Compact**
  - **one short instruction encodes N operations**
- **Expressive, tells hardware that these N operations:**
  - **Are independent**
  - **Use the same functional unit**
  - **Access disjoint registers**
  - **Access registers in the same pattern as previous instructions**
  - **Access a contiguous block of memory (unit-stride load/store)**
  - **Access memory in a known pattern (strided load/store)**
- **Scalable**
  - **Can run same object code on more parallel pipelines or *lanes***

Components of a Vector Processor

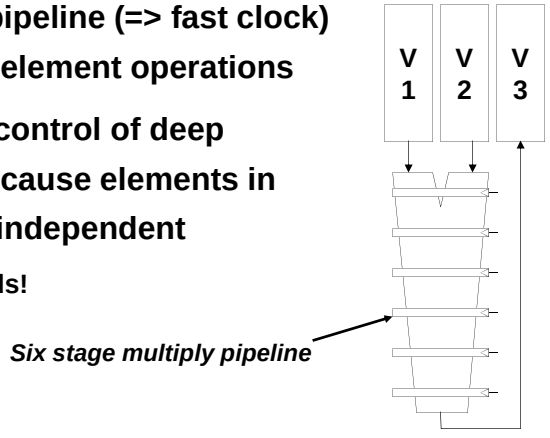
- **Vector Register File**
  - **Has at least 2 read and 1 write ports**
  - **Typically 8-32 vector registers**
  - **Each holding 64 (or more) 64-bit elements**
- **Vector Functional Units (FUs)**
  - **Fully pipelined, start new operation every clock**
  - **Typically 4 to 8 FUs: FP add, FP mult, FP reciprocal**
  - **Integer add, logical, shift (multiple of same unit)**
- **Vector Load-Store Units (LSUs)**
  - **Fully pipelined unit to load or store a vector**
  - **May have multiple LSUs**
- **Scalar registers**
  - **Single element for FP scalar or address**
- **Cross-bar to connect FUs , LSUs, registers**

Examples of Vector Machines

Machine	Year	Clock	Regs	Elements	FUs	LSUs
Cray 1	1976	80 MHz	8	64	6	1
Cray XMP	1983	120 MHz	8	64	8	2L, 1S
Cray YMP	1988	166 MHz	8	64	8	2L, 1S
Cray C-90	1991	240 MHz	8	128	8	4
Cray T-90	1996	455 MHz	8	128	8	4
Conv. C-1	1984	10 MHz	8	128	4	1
Conv. C-4	1994	133 MHz	16	128	3	1
Fuj. VP200	1982	133 MHz	8-256	32-1024	3	2
Fuj. VP300	1996	100 MHz	8-256	32-1024	3	2
NEC SX/2	1984	160 MHz	8+8K	256+var	16	8
NEC SX/3	1995	400 MHz	8+8K	256+var	16	8

Vector Arithmetic Execution

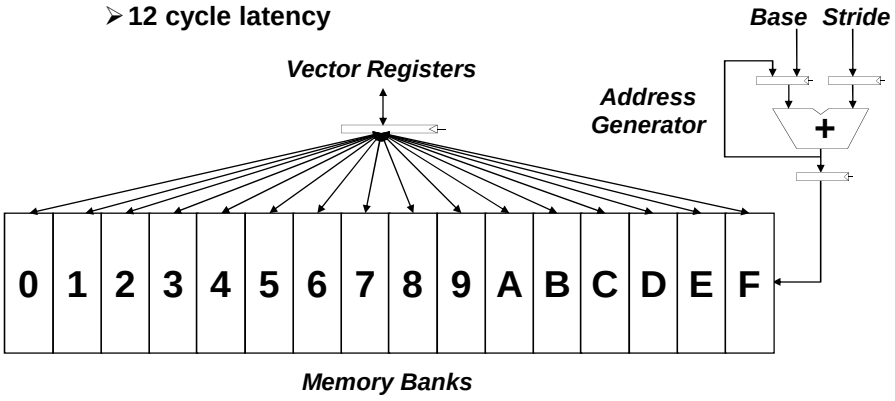
- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent
  - No hazards!



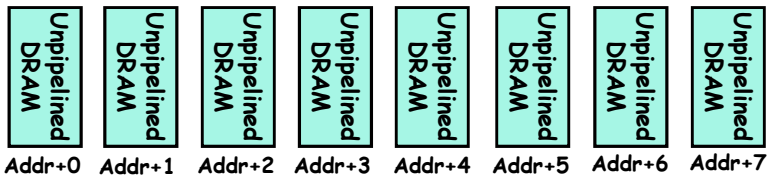
V3 <- v1 * v2

Vector Memory System

- Cray-1: 16 banks
  - 4 cycle bank busy time
    - *Bank busy time*: Cycles between accesses to same bank
  - 12 cycle latency

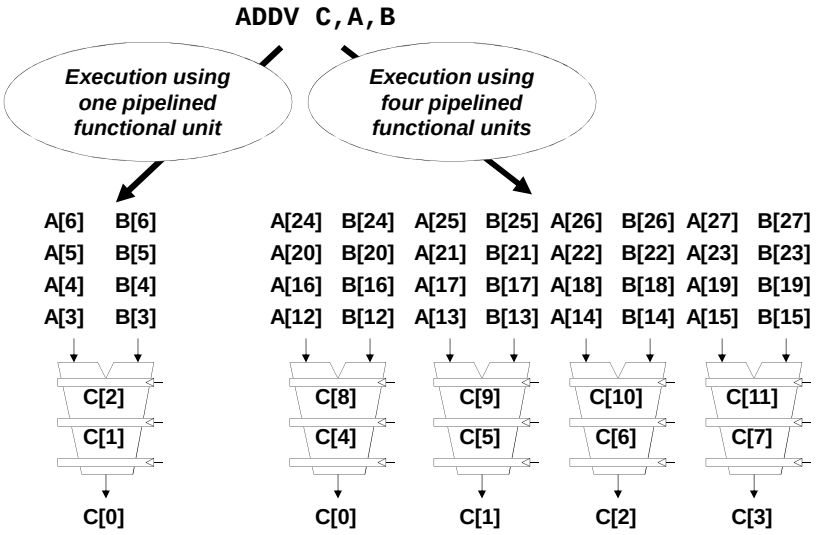


Interleaved Memory Layout

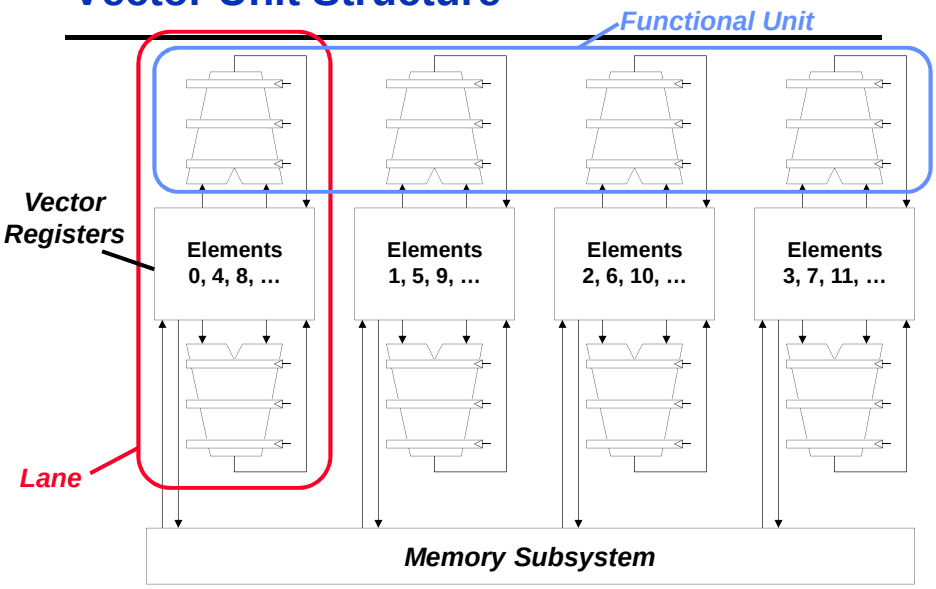


- Great for unit stride:
  - Contiguous elements in different DRAMs
  - Startup time for vector operation is latency of single read
- What about non-unit stride?
  - Above good for strides that are relatively prime to 8
  - Bad for strides = 2, 4 and worse for strides = multiple of 8
  - Better: prime number of banks…!

Vector Instruction Execution



Vector Unit Structure

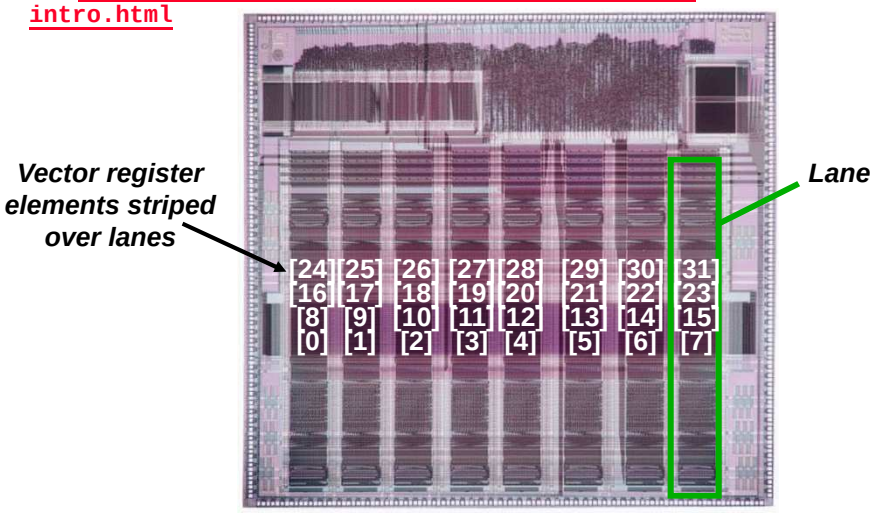


Vector Unit Implementation

- Vector register file
  - Each register is an array of elements
  - Size of each register determines maximum vector length
  - Vector length register determines vector length for a particular operation
- Multiple parallel execution units = "lanes"
  - Sometimes called "pipelines" or "pipes"

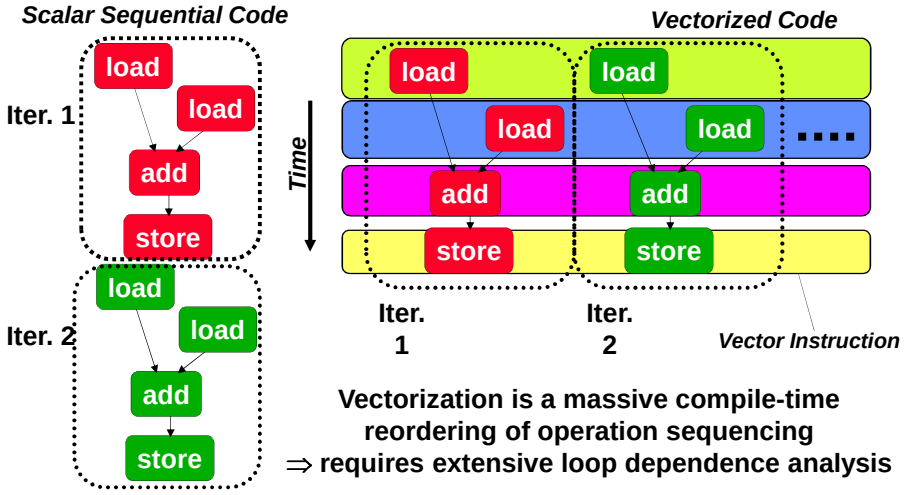
T0 Vector Microprocessor (1995)

See http://www.icsi.berkeley.edu/real/spert/t0-intro.html



Automatic Code Vectorization

```
for (i=0; i < N; i++) C[i] = A[i] + B[i];
```



Vectorization is a massive compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

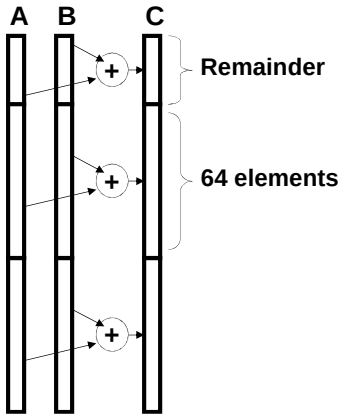
Vector Stripmining

- Problem: Vector registers have fixed length
- What to do if Vector Length > Max Vector Length?
- Stripmining: generate code such that each vector operation is done for a size ≤ MVL
  - First loop iteration: do short piece (n mod MVL)
  - Remaining iterations: VL = MVL

```
index = 0;        /* start at index 0 */
VL = (n mod MVL)  /* find the odd size piece */
while (n > 0) {
  /* do vector instructions on VL elements */
  n = n – VL;
  index = index + VL;
  VL = MVL        /* reset the length to max */
}
```

Vector Stripmining Example

```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```

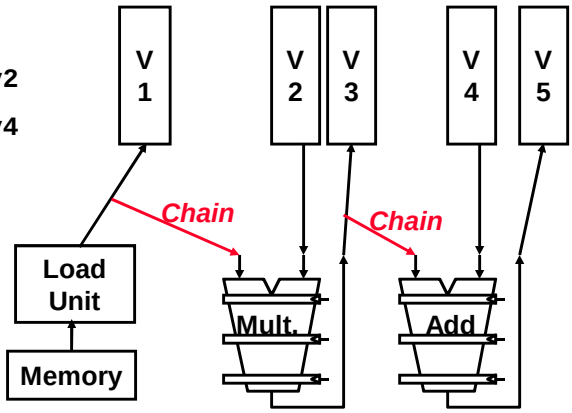



```
 ANDI R1, RN, 63 # N mod 64
 MOV  VLR, R1    # Do remainder
loop:
 LV   V1, RA
 SLL  R2, R1, 3  # Multiply by 8
 ADDU RA, RA, R2 # Advance pointer
 LV   V2, RB
 ADDU RB, RB, R2
 ADDV V3, V1, V2
 SV   V3, RC
 ADDU RC, RC, R2
 SUBU RN, RN, R1 # Subtract elements
 LI   R1, 64
 MOV  VLR, R1    # Reset full length
 BGTZ N, loop    # Any more to do?
```


Vector Chaining

- Vector version of register bypassing
  - Introduced with Cray-1

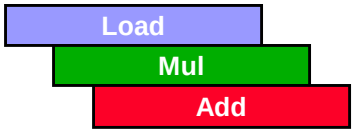
```
LV   v1, r1
MULV v3,v1,v2
ADDV v5,v3,v4
```



Vector Chaining Advantage

- Without chaining, must wait for last element of result to be written before starting dependent instruction

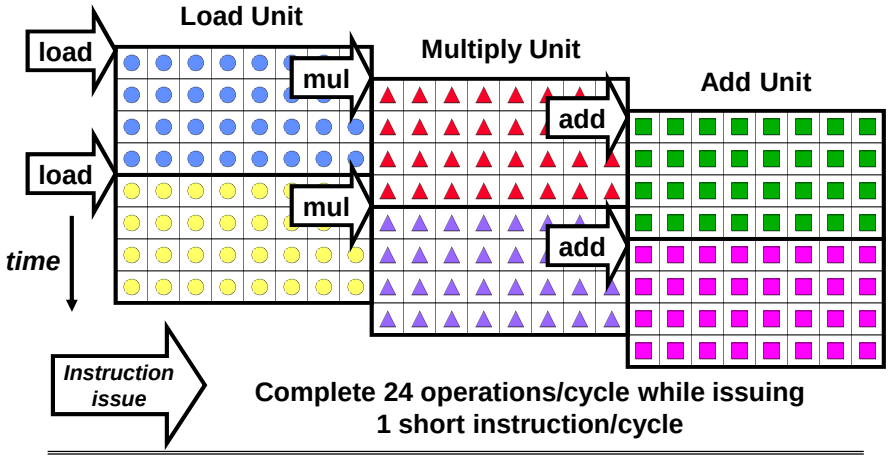
- With chaining, can start dependent instruction as soon as first result appears



Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

Example: 32 elements per vector register and 8 lanes



Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Execution Time

- Vector Execution Time depends on:
  - Vector length, data dependences, and structural hazards
- Initiation rate
  - Rate at which a vector unit consumes vector elements
  - Typically, initiation rate = number of lanes
  - Execution time of a vector instruction = VL / Initiation Rate
- Convoy
  - Set of vector instructions that can execute in same clock
  - No structural or data hazards (similar to VLIW concept)
- Chime
  - Execution time of one convoy
  - *m* convoys take *m* chimes = approximately *m* x *n* cycles
    - If each chime takes *n* cycles and no overlapping convoys

Example on Convoys and Chimes

```
LV V1, Rx     ; Load vector X
MULVS         V2, V1, F0     ; vector-Scalar multiply
LV V3, Ry     ; Load vector Y
ADDV          V4, V2, V3     ; Add vectors
SV Ry, V4     ; Store result in vector Y
```

- 4 Convoys => 4 Chimes

1. LV
2. MULVS, LV
3. ADDV
4. SV

Suppose VL=64
For 1 Lane: Chime = 64 cycles
For 2 Lanes: Chime = 32 cycles
For 4 Lanes: Chime = 16 cycles

Vector Startup

- Vector startup comes from pipeline latency
- Important source of overhead, so far ignored
- Startup time = depth of pipeline
- Increases the effective time to execute a convoy
- Time to complete a convoy depends
  - Vector startup, vector length, number of lanes

Operation	Start-up penalty (from CRAY-1)
Vector load/store	12 cycles
Vector multiply	7 cycles
Vector add	6 cycles

Startup penalty for load/store can be very high (100 cycles)

Example on Vector Startup

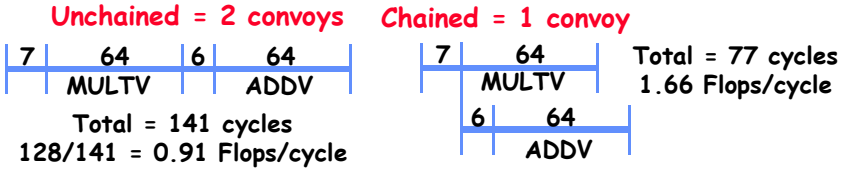
- Consider same example with 4 convoys
- Vector length = n
- Assume Convoys don't overlays
- Show the time of each convoy assuming 1 lane

Convoy	Start time	First result	Last result
1. LV	0	12	11 + n
2. MULVS, LV	12 + n	12 + n + 12	23 + 2n
3. ADDV	24 + 2n	24 + 2n + 6	29 + 3n
4. SV	30 + 3n	30 + 3n + 12	41 + 4n

- Total cycles = 42 + 4n (with extra 42 startup cycles)

Vector Chaining

- Suppose:
  MULV V1, V2, V3
  ADDV V4, V1, V5 ; RAW dependence
- Chaining: Allow a vector operation to start as soon as the individual elements of the vector source operands become available. Forward individual elements of a vector register.
- Dependent instructions can be placed in the same convoy (if no structural hazard)



Vector Stride

- Adjacent elements are not sequential in memory

```
do 10 i = 1,100
     do 10 j = 1,100
          A(i,j) = 0.0
               do 10 k = 1,100
10             A(i,j) = A(i,j) + B(i,k) * C(k,j)
```

- Either B or C accesses are not adjacent
  - 800 bytes between adjacent vector elements
- Stride: distance separating elements that are to be merged into a single vector
  - Caches do unit stride
  - LVWS (load vector with stride) instruction
- Think of addresses per vector element

Memory Addressing Modes

- Load/store operations move groups of data between registers and memory
- Three types of vector addressing
  - Unit stride
    - Contiguous block of information in memory
    - Fastest: always possible to optimize this
  - Non-unit (constant) stride
    - Harder to optimize memory system for all possible strides
    - Prime number of data banks makes it easier to support different strides at full bandwidth
  - Indexed (gather-scatter)
    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize

Vector Scatter/Gather

Want to vectorize loops with indirect accesses

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

Indexed load instruction (Gather)

```
LV  vD, rD      # Load D vector (indices)
LVI vC, rC, vD  # Load C vector indexed
LV  vB, rB      # Load B vector
ADDV vA, vB, vC # Add Vectors
SV vA, rA       # Store A vector
```

Vector Scatter/Gather

Scatter example:

```
for (i=0; i<N; i++) A[B[i]]++;
```

Vector Translation:

```
LV   vB, rB     # Load B vector (indices)
LVI  vA, rA, vB # Load A vector indexed
ADDV vA, vA, 1  # Increment
SVI  vA, rA, vB # Store A vector indexed
```

Load Vector Indexed (Gather)

Store Vector Indexed (Scatter)

Memory Banks

- Most vector processors support large number of independent memory banks
- Memory banks are need for the following reasons
  - Multiple Loads/Stores per cycle
  - Memory bank cycle time > CPU cycle time
  - Ability to load/store non-sequential elements
  - Multiple processors sharing the same memory
  - Each processor generates its stream of load/store instructions

Example on Memory Banks

- The Cray T90 has a CPU cycle = 2.167 ns
- The cycle of the SRAM in memory system = 15 ns
- Cray T90 can support 32 processors
- Each processor is capable of generating 4 loads and 2 stores per CPU clock cycle
- What is the number of memory banks required to allow all CPUs to run at full memory bandwidth
- Solution:
  - Maximum number of memory references per cycle
    32 CPUs x 6 references per cycle = 192
  - Each SRAM busy is busy for 15 / 2.167 = 6.92 ≈ 7 cycles
  - To handle 192 requests per cycle requires
    192 x 7 = 1344 memory banks
  - Cray T932 actually has 1024 memory banks

Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
  if (A[i]>0) then  A[i] = B[i]
```

Solution: Add vector mask registers

- Vector version of predicate registers, 1 bit per element
- Vector operation becomes NOP at elements where mask bit is 0

Code example:

```
CVM           # Turn on all bits in Vector Mask
LV vA, rA     # Load entire A vector
SGTV   vA, 0  # Set bits in mask register where A>0
LV vA, rB     # Load B vector into A under mask
SV vA, rA     # Store A back to memory under mask
```

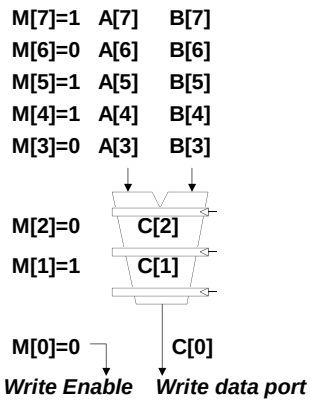
Vector Masks

- Vector masks have two important uses
  - Conditional execution and arithmetic exceptions
- Alternative is conditional move/merge
- More efficient than conditional moves
  - No need to perform extra instructions
  - Avoid exceptions
- Downside is:
  - Extra bits in instruction to specify the mask register
    - For multiple mask registers
  - Extra interlock early in the pipeline for RAW hazards

Masked Vector Instructions

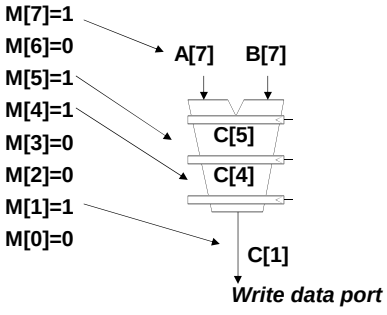
Simple Implementation

Execute all N operations
Turn off result writeback
according to mask



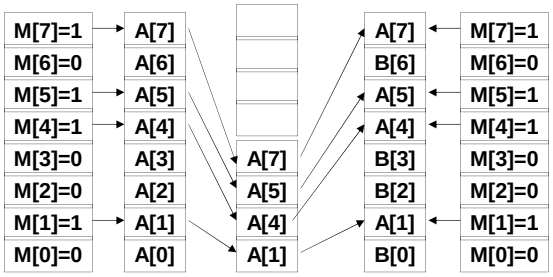
Density-Time Implementation

Scan mask vector and
Execute only elements
with Non-zero masks



Compress/Expand Operations

- **Compress:**
  - Packs non-masked elements from one vector register contiguously at start of destination vector register
  - Population count of mask vector gives packed vector length
  - Used for density-time conditionals and for general selection
- **Expand: performs inverse operation**



Vector Reductions

Problem: Loop-carried dependence on reduction variables

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  # Loop-carried dependence on sum
```

Solution: Use binary tree to perform reduction

```
# Rearrange as:
sum[0:VL-1] = 0          # Vector of VL partial sums
for(i=0; i<N; i+=VL)     # Stripmine VL-sized chunks
    sum[0:VL-1] += A[i:i+VL-1]; # Vector sum
# Now have VL partial sums in one vector register
do {
    VL = VL/2;           # Halve vector length
    sum[0:VL-1] += sum[VL:2*VL-1]
} while (VL>1)
```

New Architecture Direction?

- **"…media processing will become the dominant force in computer architecture & microprocessor design."**
- **"... new media-rich applications... involve significant real-time processing of continuous media streams, and make heavy use of vectors of packed 8-, 16-, and 32-bit integer and FP"**
- **Needs include high memory BW, high network BW, continuous media data types, real-time response, fine grain parallelism**
  - **"How Multimedia Workloads Will Change Processor Design", Diefendorff & Dubey, *IEEE Computer* (9/97)**

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**
    - **No strided, No scatter-gather memory access**
  - **No mask registers**

SIMD 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

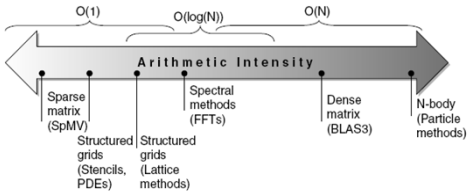
Example SIMD Code

- Example DAXPY:

```
      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
```

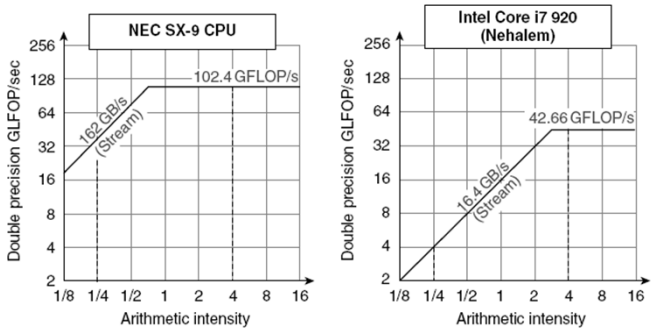
Roofline Performance Model

- **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.)**



GPU Architectures

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - OpenGL
  - Compute Unified Device Architecture (CUDA)

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

Threads and Blocks

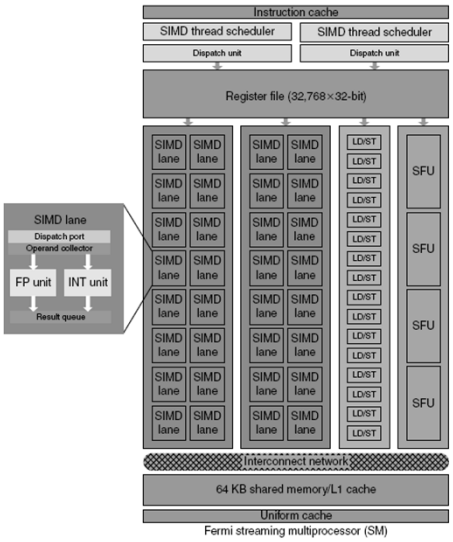
- A thread is associated with each data element
- Threads are organized into blocks
- Blocks are organized into a grid

- GPU hardware handles thread management, not applications or OS

Example: NVIDIA Fermi

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

Fermi Streaming Multiprocessor



Fermi streaming multiprocessor (SM)

Fermi Architecture Innovations

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

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:**

```
shl.s32      R8, blockIdx, 9   ; Thread Block ID * Block size (512)
add.s32      R8, R8, threadIdx         ; R8 = i = my CUDA thread ID
ld.global.f64          RD0, [X+R8]     ; RD0 = X[i]
ld.global.f64          RD2, [Y+R8]     ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4       ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2       ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0     ; Y[i] = sum (X[i]*a + Y[i])
```


Conditional Branching

- **Like vector architectures, GPU branch hardware uses internal masks**
- **Also uses**
  - **Branch synchronization stack**
    - **Entries consist of masks for each SIMD lane**
    - **I.e. which threads commit their results (all threads execute)**
  - **Instruction markers to manage when a branch diverges into multiple execution paths**
    - **Push on divergent branch**
  - **…and when paths converge**
    - **Act as barriers**
    - **Pops stack**
- **Per-thread-lane 1-bit predicate register, specified by programmer**

Example

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

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


NVIDIA GPU Memory Structures

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

Summary

- Vector is a model for exploiting Data Parallelism
- If code is vectorizable, then simpler hardware, more energy efficient, and better real-time model than Out-of-order machines
- Design issues include number of lanes, number of functional units, number of vector registers, length of vector registers, exception handling, and conditional operations
- Fundamental design issue is memory bandwidth
  - With virtual address translation and caching