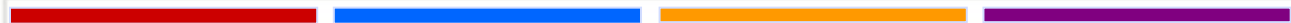


SIMD and Vector architectures



Садржај

- SIMD architecture
- Vector architectures optimizations:
 - Multiple Lanes,
 - Vector Length Registers,
 - Vector Mask Registers,
 - Memory Banks, Stride,
 - Scatter-Gather,
- Programming Vector Architectures
- SIMD extensions for media apps

Problems with conventional approach

1. **pipelined clock rate:** at some point, each increase in clock rate has corresponding CPI increase (branches, other hazards)
2. **instruction fetch and decode:** at some point, its hard to fetch and decode more instructions per clock cycle
3. **cache hit rate:** some long-running (scientific) programs have very large data sets accessed with poor locality; others have continuous data streams (multimedia) and hence poor locality

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

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

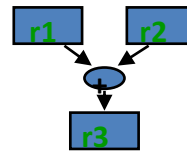
МУПС



A. Vector architectures

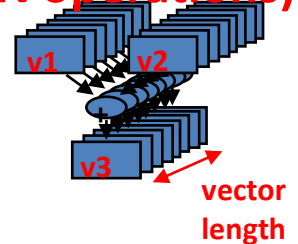
- Basic idea:
 - Read sets of data elements into “vector registers”
 - Operate on those registers
 - Disperse the results back into memory
- Registers are controlled by compiler
 - Used to hide memory latency
 - Leverage memory bandwidth

SCALAR (1 operation)



`add r3, r1, r2`

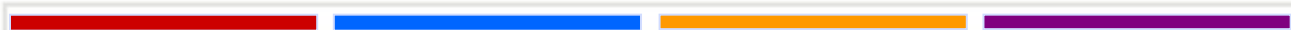
VECTOR (N operations)



`add.vv v3, v1, v2`

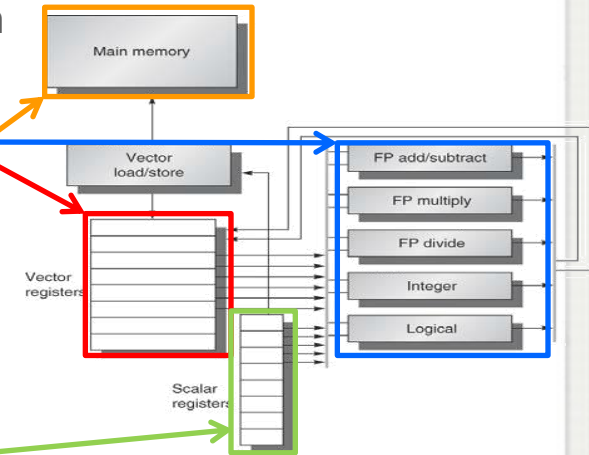
Description of Vector Processors

- CPU that implements an instruction set that operates on 1-D arrays, called *vectors*
- Vectors contain multiple data elements
- Number of data elements per vector is typically referred to as the *vector length*
- Both instructions and data are pipelined to reduce decoding time



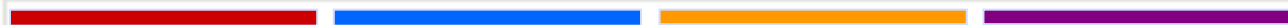
Example of vector architecture

- Vector Registers
 - Typically 8-32 vector registers with 64 - 128 64-bit elements
 - Each contains a vector of double-precision numbers
 - Register size determines the maximum vector length
 - Each includes at least 2 read and 1 write ports
- Vector Functional Units (FUs)
 - Fully pipelined, new operation every cycle
 - Performs arithmetic and logic operations
 - Typically 4-8 different units
- Vector Load-Store Units (LSUs)
 - Moves vectors between memory and registers
- Scalar Registers
 - Single elements for interconnecting FUs, LSUs, and registers



VMIPS instructions

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



DAXPY using VMIPS instructions

SAXPY or DAXPY

(SAXPY stands for single-precision a × X plus Y; DAXPY for double-precision a × X plus Y.)

- Rx -> the address of vector X
- Ry -> the address of vector Y

L.D F0,a ; load scalar a

LV V1,Rx ; load vector X

MULVS.D V2,V1,F0 ; vector-scalar multiply

LV V3,Ry ; load vector Y

ADDVV V4,V2,V3 ; add

SV Ry,V4 ; store the result

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

DAXPY using MIPS instructions

- Example: DAXPY (double precision $a \cdot X + Y$)

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 F4,0(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

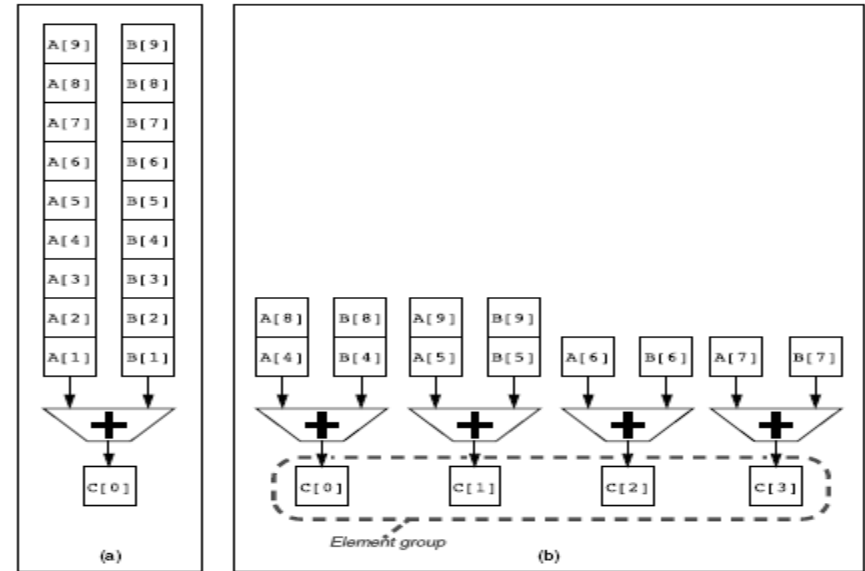
- Requires almost 600 MIPS ops when the vectors have 64 elements -> 64 elements of a vector x 9 ops

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. Single versus multiple add pipelines

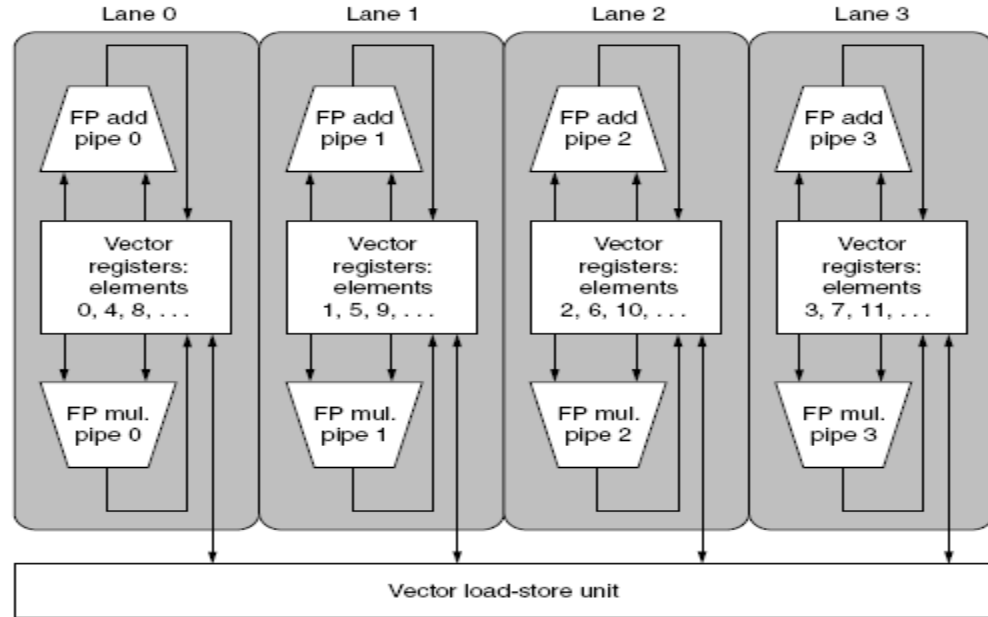
- $C = A + B$
- One versus four additions per clock cycle
- Each pipe adds the corresponding elements of the two vectors
- $C(i) = A(i) + B(i)$



1. A four lane vector unit

VIMPS instructions only allow 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

- Lane -> contains one portion of the vector register file and one execution pipeline from each functional unit
- Analog with a highway with multiple lanes!!



2. VLR and MVL

- VLR -> Vector Length Register;
MVL -> Max Vector Length
- Vector length:
 - Not known at compile time?
 - Not multiple of 64?

- **VMIPS instructions**

MTC1 VLR,R1 ;Move contents of R1 to vector-length register VL.

MFC1 R1,VLR ;Move the contents of vector-length register VL to R1.

2. VLR and MVL

- 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*/

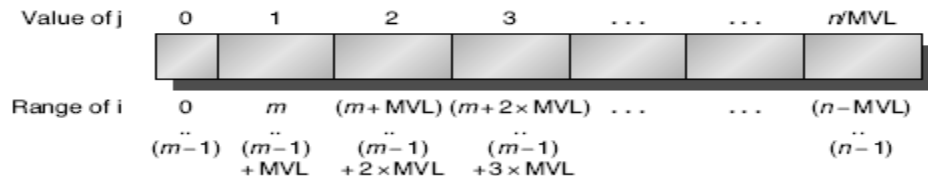
for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/

Y[i] = a * X[i] + Y[i] ; /*main operation*/

low = low + VL; /*start of next vector*/

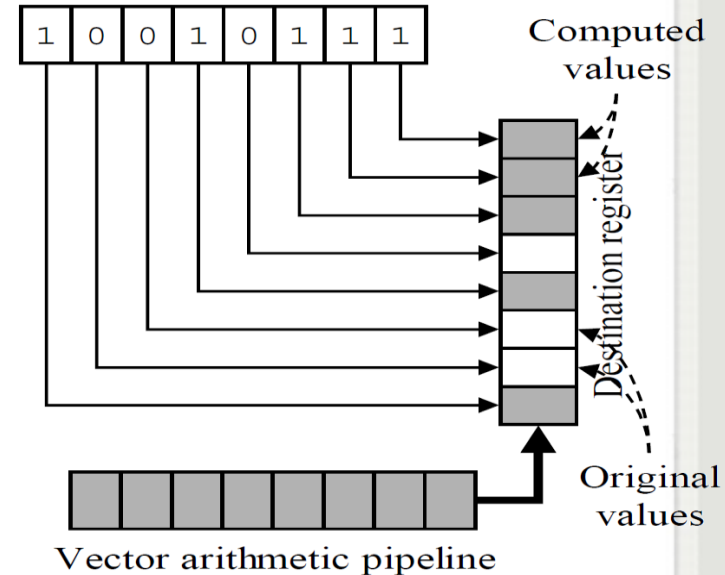
VL = MVL; /*reset length to maximum vector length*/

}



3. Vector mask registers

- Handling IF statements in a loop
for ($i = 0$; $i < 64$; $i=i+1$)
 if ($X[i] \neq 0$)
 $X[i] = X[i] - Y[i]$;
- *If conversion -> use vector mask register to “disable/select” vector elements*
- Vector mask register: one bit for each element in a vector register
 - Bit set => perform operation
 - Bit clear => do no operation
 - Functional unit busy for a cycle regardless of bit set/clear
- Allows the handling of conditions inside loops



3. Vector mask registers

- **VMIPS instructions**

S--VV.D V1,V2

S--VS.D 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.

MVTM VM,F0; Move contents of F0 to vector-mask register VM.

MVFM F0,VM; Move contents of vector-mask register VM to F0.

CVM Set the vector-mask register to all 1s.

POP R1,VM Count the 1s in vector-mask register VM and store count in R1.

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];

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!



3. Vector mask registers

- for (i = 0; i < 64; ++i)
 if (X[i] >= Y[i])
 then Z[i] = X[i]
 else Z[i] = Y[i]

LV V1,Rx ;load vector X into V1

LV V2,Ry ;load vector Y into V2

SGEVV.D V1,V2 ;sets VM(i) to 1 if V1(i)>=V2(i)

MOVVV.D V3,V1 ;move under vector mask

MVFM F0,VM; Move VM to F0

NOT F0

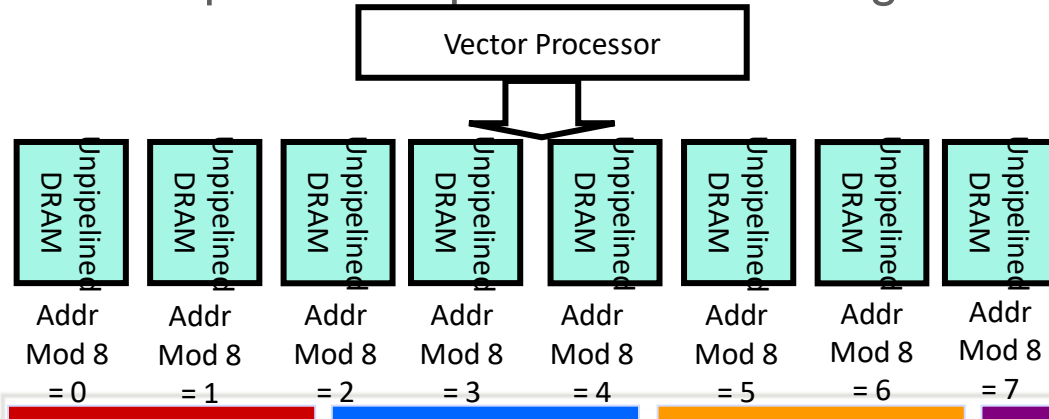
MVTM VM,F0; Move F0 to VM.

MOVVV.D V3,V2 ;move under vector mask

SV Rz,V3 ;store the result in Z

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



4. Memory banks

- 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 \approx 7 processor cycles
 - 7 x 192 \rightarrow 1344!
- The Cray T932 actually has 1024 memory banks, so the early models could not sustain full bandwidth to all processors simultaneously. A subsequent memory upgrade replaced the 15 ns asynchronous SRAMs with pipelined synchronous SRAMs that more than halved the memory cycle time, thereby providing sufficient bandwidth.

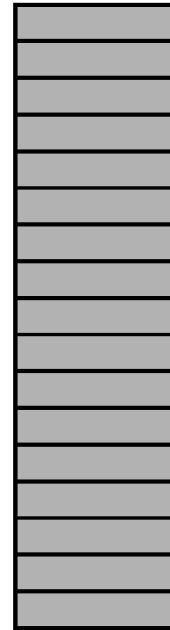
5. Stride - multiple dimensional arrays

- 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:
 - $\#banks / LCM(stride, \#banks) < \text{bank busy time (in \# of cycles)}$

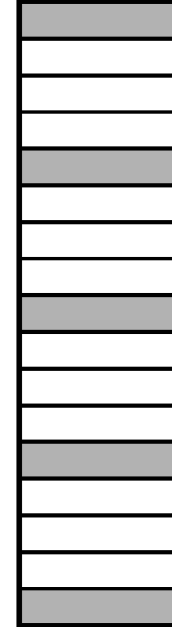
5. Stride - multiple dimensional arrays

- Data in memory may not be sequential
 - Distance between adjacent elements in the vector is called the stride
- Strided access needed only for load and store
 - Vectors held in a register are accessed normally
- Stride & vector address obtained from general purpose registers
- Stride and number of memory banks should be relatively prime
- Spreads accesses evenly for better performance

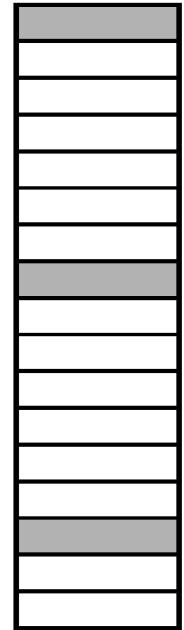
Stride = 1



Stride = 4



Stride = 7



5. Stride - multiple dimensional arrays

- **VMIPS instructions**

LVWS V1,(R1,R2) ; Load V1 from address at R1 with stride in R2 (i.e., $R1 + i \times R2$).

SVWS (R1,R2),V1 ; Store V1 to address at R1 with stride in R2 (i.e., $R1 + i \times R2$).

5. Stride - example

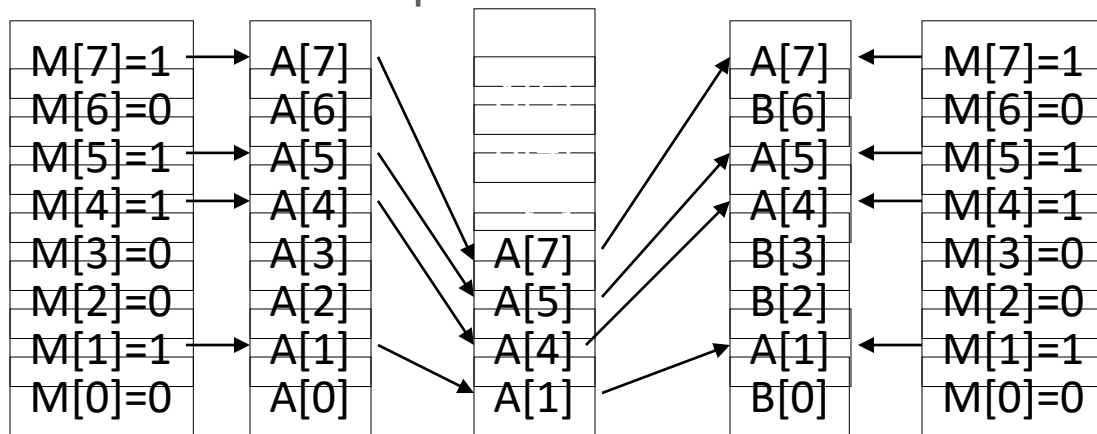
- **Given:**
 - 8 memory banks
 - bank busy time of 6 cycles
 - total memory latency of 12 cycles.
- **Questions: How long will it take to complete a 64-element vector load**
 - 1. With a stride of 1?
- **Answers:**
 - 1. Stride of 1: number of banks is greater than the bank busy time, so it takes $12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

5. Stride - example

- **Questions:** How long will it take to complete a 64-element vector load
 - 2. With a stride of 32?
- **Answers:**
 - 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

- 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
- Expand performs inverse operation



Used for density-time conditionals and also for general selection operations

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 ($i = 0$; $i < n$; $i=i+1$)

$A[K[i]] = A[K[i]] + C[M[i]]$;

R_a , R_c , R_k and R_m the starting addresses of vectors

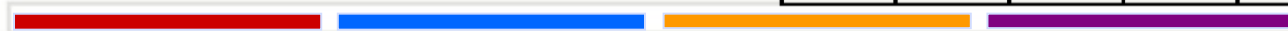
| | | | | | |
|---|----|---|---|----|---|
| 1 | 12 | 7 | 4 | 20 | 5 |
|---|----|---|---|----|---|



+ base address (=40)



| | | | | | |
|---------|---------|---------|---------|---------|---------|
| $M[41]$ | $M[52]$ | $M[47]$ | $M[44]$ | $M[60]$ | $M[45]$ |
|---------|---------|---------|---------|---------|---------|



6. Scatter-gather

VMIPS instructions

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, $1 \times R1$, $2 \times R1$, ..., $63 \times R1$ into V1.



6. Scatter-gather

for (i = 0; i < n; i=i+1)

$A[K[i]] = A[K[i]] + C[M[i]]$;

-

 LV Vk, Rk ;load K

 LVI Va, (Ra+Vk) ;load A[K[]]

 LV Vm, Rm ;load M

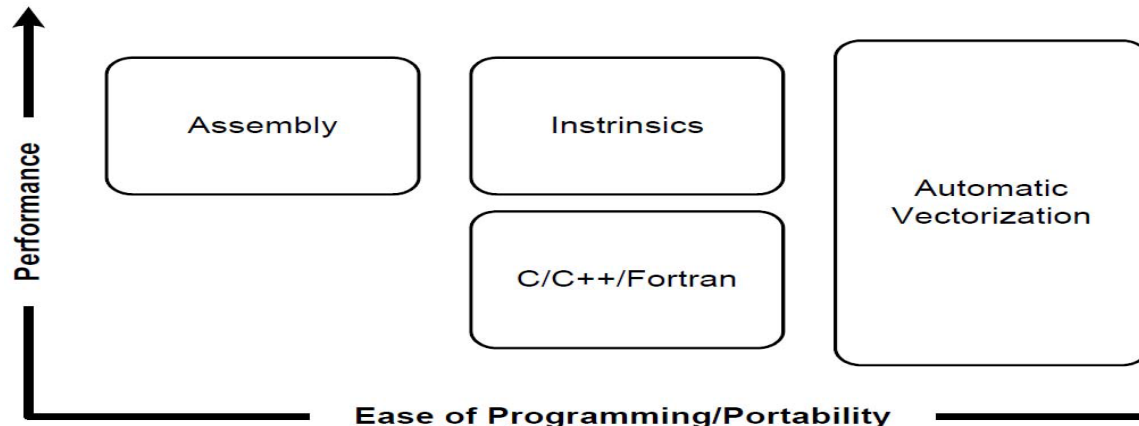
 LVI Vc, (Rc+Vm) ;load C[M[]]

 ADDVV.D Va, Va, Vc ;add them

 SVI (Ra+Vk), Va ;store A[K[]]

7. Programming vector architectures

- Intel and ARM both have vectorizing compilers which will compile code using SIMD instructions
- Many audio/video SIMD libraries available
- To achieve best performance, custom coding at the assembly level should be used



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



Summary VMIPS instructions

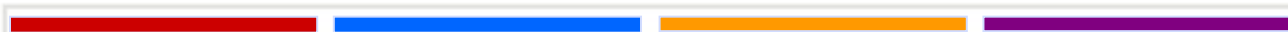
| Instruction | Operands | Function |
|-------------|------------|---|
| ADDVV.D | V1,V2,V3 | Add elements of V2 and V3, then put each result in V1. |
| ADDVS.D | V1,V2,F0 | Add F0 to each element of V2, then put each result in V1. |
| SUBVV.D | V1,V2,V3 | Subtract elements of V3 from V2, then put each 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., $R1 + i \times R2$). |
| SVWS | (R1,R2),V1 | Store V1 to address at R1 with stride in R2 (i.e., $R1 + i \times R2$). |
| 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 $0, 1 \times R1, 2 \times R1, \dots, 63 \times R1$ into V1. |
| S--VV.D | V1,V2 | 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. |
| S--VS.D | V1,F0 | |
| 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 | VLR,R1 | Move contents of R1 to vector-length register VL. |
| MFC1 | R1,VLR | Move the contents of vector-length register VL to R1. |
| MVTM | VM,F0 | Move contents of F0 to vector-mask register VM. |
| MVFM | F0,VM | Move contents of vector-mask register VM to F0. |

RV64V vector instructions (RISC-V 64 vector)

| Mnemonic | Name | Description |
|----------|-------------------------|--|
| vadd | ADD | Add elements of V[rs1] and V[rs2], then put each result in V[rd] |
| vsub | SUBtract | Subtract elements of V[rs2] from V[rs1], then put each result in V[rd] |
| vmul | MULtiply | Multiply elements of V[rs1] and V[rs2], then put each result in V[rd] |
| vdiv | DIVide | Divide elements of V[rs1] by V[rs2], then put each result in V[rd] |
| vrem | REMainder | Take remainder of elements of V[rs1] by V[rs2], then put each result in V[rd] |
| vsqrt | SQUare RooT | Take square root of elements of V[rs1], then put each result in V[rd] |
| vsll | Shift Left | Shift elements of V[rs1] left by V[rs2], then put each result in V[rd] |
| vsrl | Shift Right | Shift elements of V[rs1] right by V[rs2], then put each result in V[rd] |
| vsra | Shift Right Arithmetic | Shift elements of V[rs1] right by V[rs2] while extending sign bit, then put each result in V[rd] |
| vxor | XOR | Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd] |
| vor | OR | Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd] |
| vand | AND | Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd] |
| vsgnj | SiGN source | Replace sign bits of V[rs1] with sign bits of V[rs2], then put each result in V[rd] |
| vsgnjn | Negative SiGN source | Replace sign bits of V[rs1] with complemented sign bits of V[rs2], then put each result in V[rd] |
| vsgnjx | Xor SiGN source | Replace sign bits of V[rs1] with xor of sign bits of V[rs1] and V[rs2], then put each result in V[rd] |
| vld | Load | Load vector register V[rd] from memory starting at address R[rs1] |
| vlds | Strided Load | Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., $R[rs1] + i \times R[rs2]$) |
| vldx | Indexed Load (Gather) | Load V[rs1] with vector whose elements are at $R[rs2] + V[rs2]$ (i.e., V[rs2] is an index) |
| vst | Store | Store vector register V[rd] into memory starting at address R[rs1] |
| vsts | Strided Store | Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., $R[rs1] + i \times R[rs2]$) |
| vstx | Indexed Store (Scatter) | Store V[rs1] into memory vector whose elements are at $R[rs2] + V[rs2]$ (i.e., V[rs2] is an index) |
| vpeq | Compare = | Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0 |
| vpne | Compare != | Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0 |
| vplt | Compare < | Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0 |
| vpxor | Predicate XOR | Exclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd] |
| vpor | Predicate OR | Inclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd] |
| vpand | Predicate AND | Logical AND 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd] |
| setv1 | Set Vector Length | Set vl and the destination register to the smaller of mvl and the source register |

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*



Performance of Vector Processors

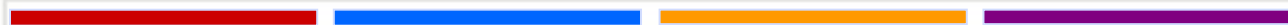
Vector Execution Time

- The execution time of a sequence of vector operations primarily depends on three factors:
 - **the length of the operand vectors**
 - **structural hazards among the operations**
 - **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*



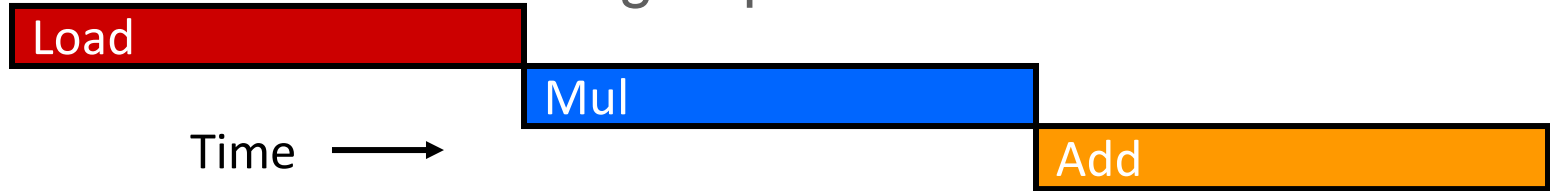
Chaining and Chimes

- *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*

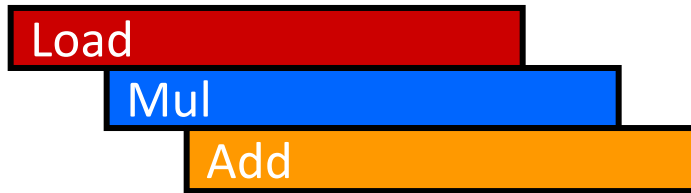


Vector Chaining

- 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

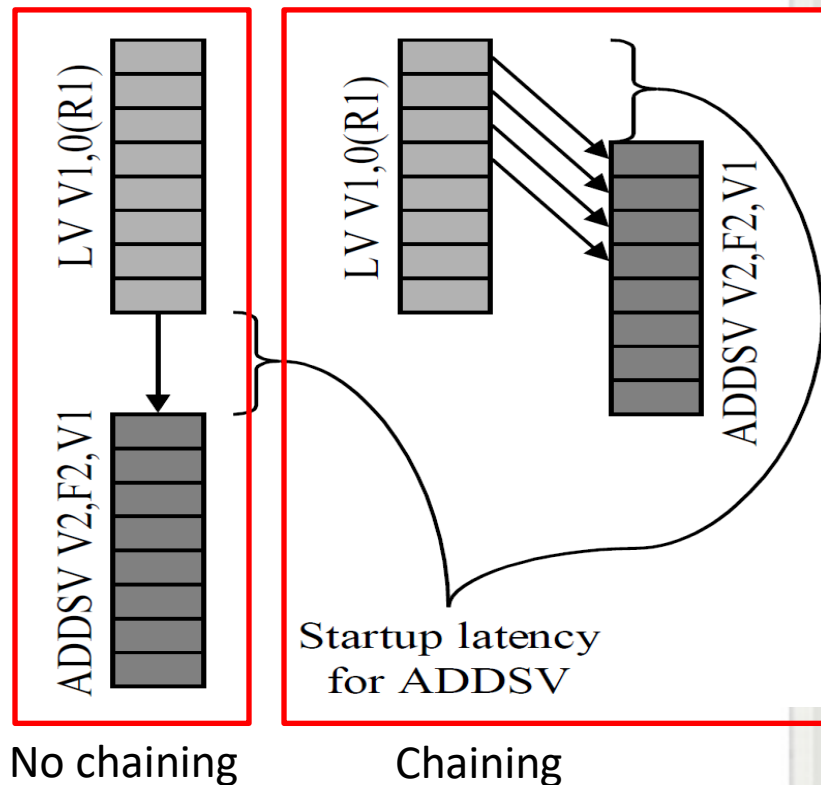


Implementations of Chaining

- Early implementations worked like forwarding, but this restricted the timing of the source and destination instructions in the chain.
- Recent implementations use *flexible chaining*, which requires simultaneous access to the same vector register by different vector instructions, which can be implemented either by adding more read and write ports or by organizing the vector-register file storage into interleaved banks in a similar way to the memory system.

Vector Chaining

- Chaining forwarding for vector operations
 - Entire result from a vector operation may take 64+ cycles to produce
 - First element is ready after the startup latency
 - Could be fed into a second operation that uses the vector register as a source,
 - Time to do two chained operations is (length + startupop1 + startupop2)
- Chaining reduces overall time by overlapping vector instructions



Vector 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 (*flexible chaining*):
 - 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

Start-up overhead

- The most important source of overhead ignored by the chime model is *vector start-up time*.
- The start-up time comes from the pipelining latency of the vector operation and is principally determined by how deep the pipeline is for the functional unit used.

Start-up overhead - Example

- Latency of vector functional units. (Cray-1)
 - Floating-point add \rightarrow 6 clock cycles
 - Floating-point multiply \rightarrow 7 clock cycles
 - Floating-point divide \rightarrow 20 clock cycles
 - Vector load \rightarrow 12 clock cycles

| Convoy | Starting time | First result time | Last result time |
|------------|---------------|-------------------|------------------|
| LV MULVS.D | 0 | $12 + 7$ | $18 + n$ |
| LV ADDVV.D | $19 + n$ | $19 + n + 12 + 6$ | $36 + 2*n$ |
| SV | $37 + 2*n$ | $37 + 2*n + 12$ | $48 + 3*n$ |

- The time per result for a vector of length 64 is $3 + (48/64) = 3.75$ clock cycles, while the chime approximation would be 3.

Performance for odd-size vectors

- Estimate loop performance using same method as before
 - Include time per element
 - Include vector instruction startup time
 - Include loop overhead
 - Doesn't include loop startup (paid once per execution, not once per loop)
- Compute T_{start} by adding up all of the vector startup latencies (excluding those that overlap in convoys)
- Compute T_{chime} by counting the convoys

$$T_n = \left\lceil \frac{n}{\text{max vector length}} \right\rceil \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}}$$

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 has a maximum vector length of 64.
 - What is the arithmetic intensity of this kernel (i.e., the ratio of floating-point operations per byte of memory accessed)?
 - Convert this loop into VMIPS assembly code using strip mining.
 - Assuming chaining and a single memory pipeline, how many chimes are required?

Exercise – arithmetic intensity

```

                                # perform the first 44 ops
                                # initialize index
loop:  li      $VL, #44
      li      $r1, #0
      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
      li      $VL, MVL        # perform 64 ops
      j      loop            # guaranteed next iteration
else:  addi    $r1, $r1, #256  # not first iteration,
                                # increment by 256
skip:  blt     $r1, #1200, loop # next iteration?
```

MVL = 64 -> 300 mod 64 = 44

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 = $(4+2)/6 = 1$.

Exercise - convoys

| | | | | | |
|------------|----------------------------|-------|---------|-------------------|-----------------------------|
| 1. mulvv.s | lv | | li | \$VL, #44 | # perform the first 44 ops |
| | # a_re * b_re | | li | \$r1, #0 | # initialize index |
| | # (assume already loaded), | loop: | lv | \$v1, a_re+\$r1 | # load a_re |
| | # load a_im | | lv | \$v3, b_re+\$r1 | # load b_re |
| | | | mulvv.s | \$v5, \$v1, \$v3 | # a_re*b_re |
| 2. lv | mulvv.s | | lv | \$v2, a_im+\$r1 | # load a_im |
| | # load b_im, a_im * b_im | | lv | \$v4, b_im+\$r1 | # load b_im |
| | | | mulvv.s | \$v6, \$v2, \$v4 | # a_im*b_im |
| 3. subvv.s | sv | | subvv.s | \$v5, \$v5, \$v6 | # a_re*b_re - a_im*b_im |
| | # subtract and store c_re | | sv | \$v5, c_re+\$r1 | # store c_re |
| 4. mulvv.s | lv | | mulvv.s | \$v5, \$v1, \$v4 | # a_re*b_im |
| | # a_re * b_re, | | mulvv.s | \$v6, \$v2, \$v3 | # a_im*b_re |
| | # load next a_re vector | | addvv.s | \$v5, \$v5, \$v6 | # a_re*b_im + a_im*b_re |
| 5. mulvv.s | lv | | sv | \$v5, c_im+\$r1 | # store c_im |
| | # a_im * b_re, | | bne | \$r1, #0, else | # check if first iteration |
| | # load next b_re vector | | addi | \$r1, \$r1, #44 | # first iteration, |
| | | | | | # increment by 44 |
| 6. addvv.s | sv | else: | li | \$VL, MVL | # perform 64 ops |
| | # add and store c_im | skip: | j | loop | # guaranteed next iteration |
| | | | addi | \$r1, \$r1, #256 | # not first iteration, |
| | | | | | # increment by 256 |
| | | | blt | \$r1, #1200, loop | # next iteration? |

6 chimes

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 (AVX) (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

Summary of SIMD implementations on PC

| Year | Description |
|------|--|
| 1994 | Hewlett-Packard introduced the “ Multimedia Acceleration eXtensions ” (MAX). It was 64-bit wide. |
| 1995 | Sun Microsystems introduced the “Visual Instruction Set” (VIS). It was used by SPARC processors. It was 64-bit wide. |
| 1996 | Intel launched the MMX. This unit was not so popular due to its technical limitations. The original name was supposed to be “Sub-word Parallelism” but marketing team decided to change it. MMX shared registers with the FPU. It was 64-bit wide. |
| 1996 | MIPS Technologies developed its own SIMD implementation. It was called “MIPS Digital Media eXtension” (MDMX). It was pretended to be launched as coprocessor of MIPS-V instruction set. Unfortunately MIPS-V was never launched neither MDMX. It was 64-bit wide and share registers with the FPU. |
| 1997 | AIM (Apple, IBM and Motorola) introduced the AltiVec instruction set on its G3 processors. It was 128-bit wide and supported SIMD floating-point operations. |

Summary of SIMD implementations on PC

| Year | Description |
|------|--|
| 1999 | Intel developed a new SIMD implementation called “ Streaming SIMD Extension ” (SSE). It doubled the size of the register respect MMX from 64-bit to 128-bit wide. Also SSE introduced support to perform floating-point operations of single-precision. SSE were more popular than MMX. |
| 2000 | AMD launched its own SIMD implementation called 3DNow! that was similar to SSE. Because AMD market was relatively small and 3DNow! was implemented only in AMD processors, 3DNow! was not popular. 3DNow! shared registers with the FPU. |
| 2002 | Intel released the next generation of SSE (SSE2). It fixed many problems with previous implementation and introduced support to perform double-precision floating-point operations. It was launched with the Pentium IV and became very popular. |
| 2004 | Intel launched SSE3. It introduced 13 new instructions over SSE2 which are primarily designed to improve thread synchronization and specific application areas such as media and gaming. |
| 2006 | Intel launched SSE4. It introduced 54 new instructions over SSE3. It introduced support to perform floating-point dot products. |

Summary of SIMD implementations on PC

| Year | Description |
|------|---|
| 2008 | Intel announced a new SIMD set instruction for x86-64 architecture. It was called “ Advanced Vector Extension ” (AVX). One of the main improvements was to double the size of the SIMD register from 128-bits to 256-bits, support up to four operand instruction and fused operations. |
| 2011 | Intel launched Sandy Bridge microarchitecture under the Core brand. Sandy Bridge was the first microarchitecture that implements AVX. |
| 2013 | Intel launched Haswell microarchitecture which includes the second generation of AVX (AVX2). |
| 2015 | AVX-512 (AVX3) launched with Knights Landing Xeon Phi processor. AVX3 doubles the size of the registers from 256-bits to 512-bits. |
| 2020 | Advanced Matrix Extensions (AMX), also known as Intel Advanced Matrix Extensions (Intel AMX), are extensions to the x86 instruction set architecture (ISA) for microprocessors from Intel designed to work on matrices to accelerate artificial intelligence (AI) and machine learning (ML) workloads. |
| | |

SIMD extension to x86-64 implementations

| 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 four 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

- 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 |

Vectors Are Inexpensive

Superscalar

- N ops per cycle =>
 $O(N^2)$ circuitry
- HP PA-8000
 - 4-way issue
 - reorder buffer alone:
850K transistors
incl. 6,720 5-bit register
number comparators

Vector

- N ops per cycle =>
 $O(N + \epsilon N^2)$ circuitry
- UCB-T0 Integer vector μP
 - 24 ops per cycle
 - 730K transistors total
only 23 5-bit register
number comparators
 - Integer, no floating point

Energy Efficiency

Single-issue Scalar

- One instruction fetch, decode, dispatch per operation
- Arbitrary register accesses, adds area and power
- Loop unrolling and software pipelining for high performance increases instruction cache footprint
- All data passes through cache; waste power if no temporal locality
- One TLB lookup per load or store
- Off-chip access is via whole cache lines

Vector

- One instruction fetch, decode, dispatch per vector
- Structured register accesses
- Smaller code for high performance, less power in instruction cache misses
- Bypass cache
- One TLB lookup per group of loads or stores
- Move only necessary data across chip boundary



Energy Efficiency

Superscalar

- Control logic grows quadratically with issue width (n \times n hazard checks)
- Control logic consumes energy regardless of available parallelism
- Speculation to increase visible parallelism wastes energy

Vector

- Control logic grows linearly with issue width
- Vector unit switches off when not in use
- Vector instructions expose parallelism without speculation
- Software control of speculation when desired:
 - Whether to use vector mask or compress/expand for conditionals



Examples

Loop 1: DO i = 1, N

$q(i) = g(i) * b(i) + c(i)$

END DO

Loop 2: DO i = 1, N

$q(i) = q(i) + f * b(i)$

END DO

Loop 3: DO i = 1, N

$q(i) = 0.5 * (q(i-1) + q(i+1))$

END DO

Loop 4: DO i = 1, N, 2

$q(i) = 0.5 * (q(i-1) + q(i+1))$

END DO

Loop 5: DO i = 1, Nx

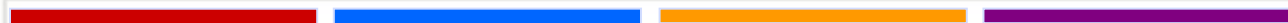
 IF (F(i+1) .LT. 1.0E-10) then

$F(i+1) = 0.0$

 ELSE

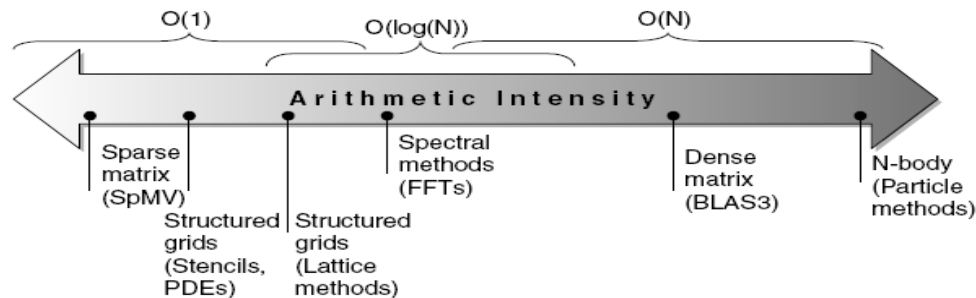
$q(i) = q(i) - dtdx * (F(i+1) - F(i))$

END DO



Roofline performance model

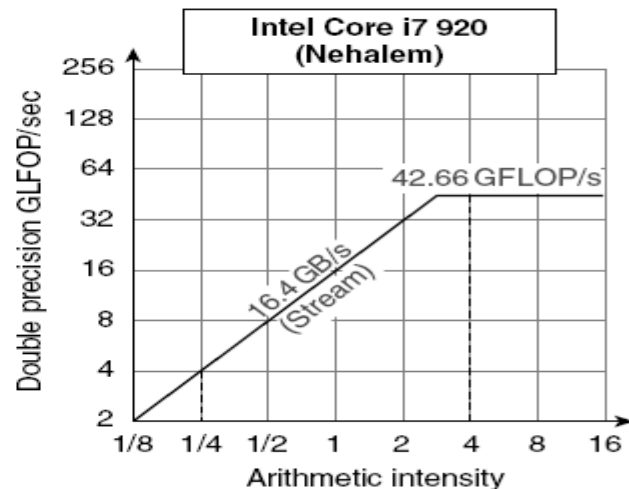
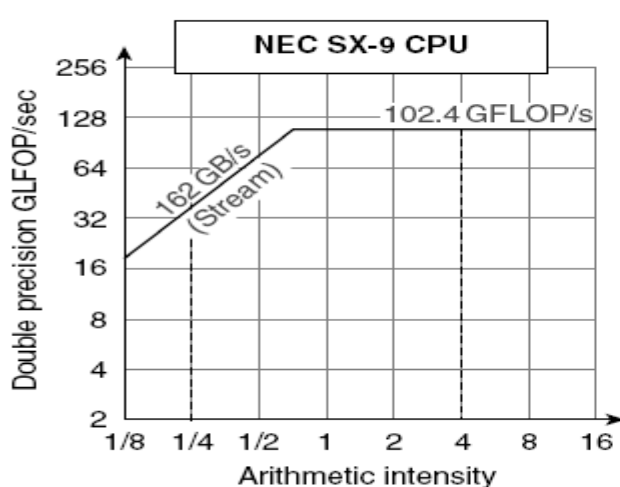
- Basic idea:
 - Peak floating-point throughput as a function of arithmetic intensity
 - Ties together floating-point performance and memory performance
 - Roofline: on the sloped portion of the roof the performance is limited by the memory bandwidth, on the flat portion it is limited by arithmetic intensity
- *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
- $\text{Min} = (\text{Peak Memory BW} \times \text{Arithmetic Intensity}, \text{Peak Floating Point Performance})$



Thanks

Slides developed with sources that included slides by:

- Hennessy, Patterson “Computer Architecture A Quantitative Approach,” Fifth Edition, Morgan Kaufman
- Dan C. Marinescu from University of Central Florida
- Eric Welch & James Evans, Multiple Processor Systems
- Ethan Miller, University of Maryland, Baltimore County...
- Yiorgos Makris, University of Texas at Dallas
- Eduardo Jonathan Martínez Montes, Universitat Politècnica de Catalunya
- Larry Wittie, SBU, Krste Asanovic of UCB and David Patterson of UCB

Питања?

Електротехнички Факултет
Универзитет у Београду

