SIMD & MPP Readings


Lecture Outline

• SIMD introduction
• Automatic Parallelization for SIMD machines
• Vector Architectures
  – Cray-1 case study
# SIMD vs. Alternatives

From [Hughes, SIMD Synthesis Lecture]

<table>
<thead>
<tr>
<th>Hardware</th>
<th>SIMD</th>
<th>Superscalar</th>
<th>Multithreading</th>
<th>Multi-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch/Decode</td>
<td>Single instruction specifies many instances of same operation</td>
<td>Handle multiple instruction per cycle</td>
<td>Handle multiple instruction per cycle</td>
<td>Each core has own fetch/decode logic</td>
</tr>
<tr>
<td>Control Flow</td>
<td>Same code path for many elements, predication</td>
<td>Each element has independent control flow, prediction may be hard</td>
<td>Each element has independent control flow</td>
<td>Each core has independent control flow</td>
</tr>
<tr>
<td>Inter-Element</td>
<td>No needed, only check between instructions</td>
<td>Check all instructions with each other</td>
<td>Intra-thread checks, but no inter-thread checks</td>
<td>Intra-thread checks, but no cross-core checks</td>
</tr>
<tr>
<td>Dependence Check</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUs</td>
<td>Wide ALU, same operation on multiple elements per cycle</td>
<td>Multiple independent ALUs</td>
<td>Multiple independent ALUs</td>
<td>Each core has own ALUs</td>
</tr>
<tr>
<td>Memory System</td>
<td>Wide memory operations, limited non-contiguous support</td>
<td>Multiple narrow operations</td>
<td>Multiple narrow operations</td>
<td>Narrow operation(s) per core, coherence actions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SIMD vs. Superscalar

From [Hughes, SIMD Synthesis Lecture]
Multithreaded vs. Multicore

From [Hughes, SIMD Synthesis Lecture]
SIMD Efficiency

From [Hughes, SIMD Synthesis Lecture]

- Amdahl’s Law...

\[ \text{SIMD efficiency} = \frac{\text{instructions}_{\text{scalar}}}{\text{instructions}_{\text{SIMD}}} \times \frac{\text{vector length}}{} \]
SIMD History

• Vector machines, supercomputing
  – Illiac IV, CDC Star-100, TI ASC,
  – Cray-1: *properly* architected (by Cray-2 gen)

• Incremental adoption in microprocessors
  – Intel Pentium MMX: vectors of bytes
  – Subsequently: SSEx/AVX-y, now AVX-512
  – Also SPARC, PowerPC, ARM, ...
  – *Improperly* architected...
  – Also GPUs from AMD/ATI and Nvidia (later)
Register Overlays

From [Hughes, SIMD Synthesis Lecture]
SIMD Challenges

• Remainders
  – Fixed vector length, software has to fix up
  – Properly architected: VL is supported in HW

• Control flow deviation
  – Conditional behavior in loop body
  – Properly architected: vector masks

• Memory access
  – Alignment restrictions
  – Virtual memory, page faults (completion masks)
  – Irregular accesses: properly architected gather/scatter

• Dependence analysis (next)
Lecture Outline

• SIMD introduction
• Automatic Parallelization for SIMD machines
• Vector Architectures
  – Cray-1 case study
Automatic Parallelization

• Start with sequential programming model
• Let the compiler attempt to find parallelism
  – It can be done...
  – We will look at one of the success stories
• Commonly used for SIMD computing – vectorization
  – Useful for MIMD systems, also -- concurrentization
• Often done with FORTRAN
  – But, some success can be achieved with C
    (Compiler address disambiguation is more difficult with C)
Automatic Parallelization

• Consider operations on arrays of data

  \[
  \text{do } I=1,N \\
  \quad \cdot A(I,J) = B(I,J) + C(I,J) \\
  \text{end do} \\
  \text{– Operations along one dimension involve } \text{vectors}
  \]

• Loop level parallelism

  \text{– Do all} – all loop iterations are independent
    \quad \cdot \text{Completely parallel}
  \text{– Do across} – some dependence across loop iterations
    \quad \cdot \text{Partly parallel}

\[
A(I,J) = A(I-1,J) + C(I,J) \ast B(I,J)
\]
Data Dependence

• Independence $\Rightarrow$ Parallelism
  OR, dependence inhibits parallelism

  $S_1: \ A = B + C$
  $S_2: \ D = A + 2$
  $S_3: \ A = E + F$

• True Dependence (RAW):
  $S_1 \ \delta \ \ S_2$

• Antidependence (WAR):
  $S_2 \ \delta^- \ S_3$

• Output Dependence (WAW):
  $S_1 \ \delta^o \ S_3$
Data Dependence Applied to Loops

• Similar relationships for loops
  – But consider iterations
    
    ```
    do I=1,2
    S1: A(I)=B(I)+C(I)
    S2: D(I)=A(I)
    end do
    ```

• \( S_1 \delta = S_2 \)
  – Dependence involving A, but on same loop iteration
Data Dependence Applied to Loops

• $S_1 \delta_< S_2$
  
  \[
  \begin{array}{l}
  \text{do } \ i = 1, 2 \\
  S_1: \; A(i) = B(i) + C(i) \\
  S_2: \; D(i) = A(i - 1)
  \end{array}
  \]
  
  – Dependence involving $A$, but read occurs on next loop iteration
  – *Loop carried dependence*

• $S_2 \delta^-_< S_1$
  
  – Antidependence involving $A$, write occurs on next loop iteration
  \[
  \begin{array}{l}
  \text{do } \ i = 1, 2 \\
  S_1: \; A(i) = B(i) + C(i) \\
  S_2: \; D(i) = A(i + 1)
  \end{array}
  \]
Loop Carried Dependence

Definition

• do I = 1, N

S1: X(f(i)) = F(...)  
S2: A = X(g(i)) ...

end do

S1 δ S2: is loop-carried

• if there exist i₁, i₂ where
  1 ≤ i₁ < i₂ ≤ N and f(i₁) = g(i₂)

If f and g can be arbitrary functions, the problem is essentially unsolvable.

However, if (for example)

f(i) = c*I + j and g(i) = d*I + k

there are methods for detecting dependence.
Loop Carried Dependences

• GCD test
  
  \[
  \begin{aligned}
  \text{do} & \quad I = 1, N \\
  S1: & \quad X(c*I + j) = F(\ldots) \\
  S2: & \quad A = X(d*I + k) \ldots \\
  \text{end do}
  \end{aligned}
  \]

  \[f(x) = g(y) \text{ if } c*I + j = d*I + k\]

  This has a solution iff \(\gcd(c, d) \mid k-j\)

• Example

  \[
  A(2*I) = \]

  \[= A(2*I + 1)\]

  \[\text{GCD}(2,2) \text{ does not divide } 1 - 0\]

• The GCD test is of limited use because it is very conservative

  \[\text{often } \gcd(c,d) = 1\]

  \[X(4i+1) = F(X(5i+2))\]

• Other, more complex tests have been developed

  e.g. Banerjee's Inequality, polyhedral analysis
Vector Code Generation

• In a vector architecture, a vector instruction performs identical operations on vectors of data

• Generally, the vector operations are independent
  – A common exception is reductions (horizontal ops)

• In general, to vectorize:
  – There should be no cycles in the dependence graph
  – Dependence flows should be downward
    ⇒ some rearranging of code may be needed.
Vector Code Generation: Example

\begin{verbatim}
do I = 1, N
S1:   A(I) = B(I)
S2:   C(I) = A(I) + B(I)
S3:   E(I) = C(I+1)
end do

• Construct dependence graph

S1:
\\downarrow \delta
S2:
\\uparrow \delta^-
S3:

Vectorizes (after re-ordering S2: and S3: due to antidependence)

S1:   A(I:N) = B(I:N)
S3:   E(I:N) = C(2:N+1)
S2:   C(I:N) = A(I:N) + B(I:N)
\end{verbatim}
Multiple Processors (Concurrentization)

• Often used on outer loops

• Example

\[
do \quad I = 1, N \\
\quad do \quad J = 2, N \\
S1: \quad A(I, J) = B(I, J) + C(I, J) \\
S2: \quad C(I, J) = D(I, J)/2 \\
S3: \quad E(I, J) = A(I, J-1)**2 + E(I, J-1)
\]

end do
end do

• Data Dependences & Directions

\[
S1 \delta_{=, <} S3 \\
S1 \delta_{=, =} S2 \\
S3 \delta_{=, <} S3
\]

• Observations

– All dependence directions for I loop are =

\[
\Rightarrow \text{Iterations of the I loop can be scheduled in parallel}
\]
Scheduling

• Data Parallel Programming Model
  – SPMD (single program, multiple data)

• Compiler can pre-schedule:
  – Processor 1 executes 1st N/P iterations,
  – Processor 2 executes next N/P iterations
  – Processor P executes last N/P iterations
  – Pre-scheduling is effective if execution time is nearly identical for each iteration

• Self-scheduling is often used:
  – If each iteration is large
  – Time varies from iteration to iteration
    - iterations are placed in a "work queue”
    - a processor that is idle, or becomes idle takes the next block of work from the queue (critical section)
Code Generation with Dependences

\[ \text{do } I = 2, N \]
\[ S1: \ A(I) = B(I) + C(I) \]
\[ S2: \ C(I) = D(I) \times 2 \]
\[ S3: \ E(I) = C(I) + A(I-1) \]
\[ \text{end do} \]

• Data Dependences & Directions
  \[ S1 \delta_\leq S2 \]
  \[ S1 \delta_\leq S3 \]
  \[ S2 \delta_\leq S3 \]

• Parallel Code on \( N-1 \) Processors
  \[ S1: \ A(I) = B(I) + C(I) \]
  \[ \text{signal}(I) \]
  \[ S2: \ C(I) = D(I) \times 2 \]
  \[ \text{if } (I > 2) \text{ wait}(I-1) \]
  \[ S3: \ E(I) = C(I) + A(I-1) \]

• Observation
  – Weak data-dependence tests may add unnecessary synchronization.
  \[ \Rightarrow \text{Good dependence testing crucial for high performance} \]
Reducing Synchronization

\[
\begin{align*}
\text{do } & \quad I = 1, N \\
S1: & \quad A(I) = B(I) + C(I) \\
S2: & \quad D(I) = A(I) \times 2 \\
S3: & \quad \text{SUM } = \text{SUM } + A(I)
\end{align*}
\]
\quad \text{end do}

• Parallel Code: Version 1

\[
\begin{align*}
\text{do } & \quad I = p, N, P \\
S1: & \quad A(I) = B(I) + C(I) \\
S2: & \quad D(I) = A(I) \times 2 \\
& \quad \text{if } (I > 1) \text{ wait}(I-1) \\
S3: & \quad \text{SUM } = \text{SUM } + A(I) \\
& \quad \text{signal}(I)
\end{align*}
\]
\quad \text{end do}
Reducing Synchronization, contd.

• Parallel Code: Version 2

\[
\text{SUMX}(p) = 0 \\
\text{do } \ I = p, N, P \\
S1: \quad A(I) = B(I) + C(I) \\
S2: \quad D(I) = A(I) * 2 \\
S3: \quad \text{SUMX}(p) = \text{SUMX}(p) + A(I) \\
\text{end do} \\
\text{barrier synchronize} \\
\text{add partial sums}
\]

• Not always safe (bit-equivalent): why?
Vectorization vs Concurrentization

• When a system is a vector MP, when should vector/concurrent code be generated?

```plaintext
  do J = 1,N
    do I = 1,N
      S1:  A(I,J+1) = B(I,J) + C(I,J)
      S2:  D(I,J) = A(I,J) * 2
    end do
  end do
```

• Parallel & Vector Code: Version 1

```plaintext
  doacross J = 1,N
    S1:  A(1:N,J+1) = B(1:N,J)+C(1:N,J)
    signal(J)
    if (J > 1) wait (J-1)
    S2:  D(1:N,J) = A(1:N,J) * 2
  end do
```
Vectorization vs Concurrentization

• Parallel & Vector Code: Version 2
  Vectorize on J, but non-unit stride memory access
  (assuming Fortran Column Major storage order)

  \[
  \begin{align*}
  \text{doall } & I = 1,N \\
  S1: \quad & A(I,2:N+1) = B(I,1:N) + C(I,1:N) \\
  S2: \quad & D(I,1:N) = A(I,1:N) \times 2 \\
  \text{end do}
  \end{align*}
  \]

• Need support for gather/scatter
Summary

• Vectorizing compilers have been a success
• Dependence analysis is critical to any auto-parallelizing scheme
  – Software (static) disambiguation
  – C pointers are especially difficult
• Can also be used for improving performance of sequential programs
  – Loop interchange
  – Fusion
  – Etc.
Aside: Thread-Level Speculation

• Add hardware to resolve difficult concurrentization problems

• Memory dependences
  – Speculate independence
  – Track references (cache versions, r/w bits, similar to TM)
  – Roll back on violations

• Thread/task generation
  – Dynamic task generation/spawn (Multiscalar)

• References
Cray-1 Architecture

• Circa 1976
• 80 MHz clock
  – When high performance mainframes were 20 MHz
• Scalar instruction set
  – 16/32 bit instruction sizes
    • Otherwise conventional RISC
  – 8 S register (64-bits)
  – 8 A registers (24-bits)
• In-order pipeline
  – Issue in order
  – Can complete out of order (no precise traps)
Cray-1 Vector ISA

- 8 vector registers
  - 64 elements
  - 64 bits per element (word length)
  - Vector length (VL) register

- RISC format
  - $V_i \leftarrow V_j \text{ OP } V_k$
  - $V_i \leftarrow \text{mem}(A_j, \text{disp})$

- Conditionals via vector mask (VM) register
  - $VM \leftarrow V_i \text{ pred } V_j$
  - $V_i \leftarrow V_2$ conditional on VM

Figure 3-1. Computation section
Do 10 i=1,looplength
   a(i) = b(i) * x + c(i)
10  continue

A1  \leftarrow  \text{looptlength}  \quad \text{.initial values:}
A2  \leftarrow  \text{address(a)}  \quad \text{.for the arrays}
A3  \leftarrow  \text{address(b)}  \quad \text{.}
A4  \leftarrow  \text{address(c)}  \quad \text{.}
A5  \leftarrow  0  \quad \text{.index value}
A6  \leftarrow  64  \quad \text{.max hardware VL}
S1  \leftarrow  x  \quad \text{.scalar x in register S1}
VL  \leftarrow  A1  \quad \text{.set VL – performs mod function}

\text{BrC} \quad \text{done, A1<=0}  \quad \text{.branch if nothing to do

more:  \quad V3  \leftarrow  \text{A4,A5}  \quad \text{.load c indexed by A5 – addr mode not in Cray-1}
        \quad V1  \leftarrow  \text{A3,A5}  \quad \text{.load b indexed by A5}
        \quad V2  \leftarrow  \text{V1 * S1}  \quad \text{.vector times scalar}
        \quad V4  \leftarrow  \text{V2 + V3}  \quad \text{.add in c}
        \quad A2,A5 \leftarrow  \text{V4}  \quad \text{.store to a indexed by A5}
        \quad A7  \leftarrow  \text{VL}  \quad \text{.read actual VL}
        \quad A1  \leftarrow  \text{A1 – A7}  \quad \text{.remaining iteration count}
        \quad A5  \leftarrow  \text{A5 + A7}  \quad \text{.increment index value}
        \quad VL  \leftarrow  \text{A6}  \quad \text{. set VL for next iteration}
        \quad \text{BrC} \quad \text{more, A1>0}  \quad \text{.branch if more work

done:
Compare with Scalar

Do 10 i=1,looplength
   a(i) = b(i) * x + c(i)
10  continue

2 loads
1 store
2 FP
1 branch
1 index increment (at least)
1 loop count increment

total -- 8 instructions per iteration

4-wide superscalar => up to 1 FP op per cycle
vector, with chaining => up to 2 FP ops per cycle (assuming mem b/w)

Also, in a CMOS microprocessor would save a lot of energy
Vector Conditional Loop

do 80 i = 1,looplen
    if (a(i).eq.b(i)) then
        c(i) = a(i) + e(i)
    endif
80 continue

V1 ← A1 .load a(i)
V2 ← A2 .load b(i)
VM ← V1 == V2 .compare a and b; result to VM
V3 ← A3; VM .load e(i) under mask
V4 ← V1 + V3; VM .add under mask
A4 ← V4; VM .store to c(i) under mask
Vector Conditional Loop

Gather/Scatter Method (used in later Cray machines)

do 80 i = 1,looplen
    if (a(i).eq.b(i)) then
        c(i) = a(i) + e(i)
    endif
80    continue

V1 ← A1 .load a(i)
V2 ← A2 .load b(i)
VM ← V1 == V2 .compare a and b; result to VM
V5 ← IOTA(VM) .form index set
VL ← pop(VM) .find new VL (population count)
V6 ← A1, V5 .gather a(i) values
V3 ← A3, V5 .gather e(i) values
V4 ← V6 + V3 .add a and e
A4,V11 ← V4 .scatter sum into c(i)
Lecture Summary

• SIMD introduction
• Automatic Parallelization
• Vector Architectures
  – Cray-1 case study