CDA 4150 Lecture 4

Vector Processing
CRAY like machines
Amdahl’s Law

\[ \alpha = a + b + c \]

\[ T_S = \text{Time Spent in Sequential Processing} \]

\[ T_P = \text{Time Spent in Parallel Processing} \]

\[ S_P = \text{Speedup} \]

\[ P = \text{Number of Processors} \]
Amdahl’s Law (cont.)

\[
S_p = \frac{T_s}{T_p}
\]

\[
T_p = \alpha T_s + \frac{(1 - \alpha)T_s}{P}
\]

\[
S_p = \frac{T_s}{\alpha T_s + \frac{(1 - \alpha)T_s}{P}}
\]

\[
S_p = \frac{1}{\alpha + \frac{(1 - \alpha)}{P}}
\]

\[
S_p = \frac{P}{P\alpha + (1 - \alpha)}
\]

\[
S_p = \frac{1}{\frac{1}{P} + \left(1 - \frac{1}{P}\right)\alpha}
\]

\[
\lim_{P \to \infty} S_p = \lim_{P \to \infty} \frac{1}{\frac{1}{P} + \left(1 - \frac{1}{P}\right)\alpha}
\]

\[
\lim_{P \to \infty} S_p = \frac{1}{\alpha}
\]
Amdahl’s Law (revisited)

\[ Speedup = \frac{1}{\frac{1}{p} + \left(1 - \frac{1}{p}\right)\alpha} \Rightarrow \lim_{p \to \infty} Sp = \frac{1}{\alpha} \]

- Using \( \alpha \) as a function of \( n \), where \( \alpha(n) = \frac{1}{n} \), then

\[ Speedup = \frac{p}{1 + (p - 1)\alpha(n)} = \lim_{n \to \infty} \frac{p}{1 + (p - 1)\frac{1}{n}} = p \]
An extension of Amdahl’s Law in terms of a matrix multiplication equation \((AX = Y)\).

\[
\begin{bmatrix}
    a_{11} & a_{12} & a_{13} & a_{14} \\
    a_{21} & a_{22} & a_{23} & a_{24} \\
    a_{31} & a_{32} & a_{33} & a_{34} \\
    a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\begin{bmatrix}
    x_1 \\
    x_2 \\
    x_3 \\
    x_4
\end{bmatrix} =
\begin{bmatrix}
    y_1 \\
    y_2 \\
    y_3 \\
    y_4
\end{bmatrix}
\]

\[
y_1 = a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4
\]

\[
y_2 = a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4
\]

\[
y_3 = a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4
\]

\[
y_4 = a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4
\]
Compute each vector element in parallel by partitioning.

\[
\begin{bmatrix}
A_1 & x_1 \\
A_2 & x_2 \\
A_3 & x_3 \\
A_4 & x_4 \\
\end{bmatrix} = 
\begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
\end{bmatrix}
\]

- Introduces CRAY-1 as a vector processing Architecture
CRAY -1

Functional Units

MAIN MEMORY
(16-way interleaving)

64 elements

V0

V1

V2

Vn

FP ADD / SUB

FP MULT

LOGICAL

PIPELINE

SCALAR
REGISTERS
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDV</td>
<td>V₁, V₂, V₃</td>
<td>V₁ ← V₂ + V₃</td>
</tr>
<tr>
<td>ADDSV (add scalar vector)</td>
<td>V₁, F₀*, V₂</td>
<td>V₁ ← V₂ + F₀</td>
</tr>
<tr>
<td>MULTV</td>
<td>V₁, V₂, V₃</td>
<td>V₁ ← V₂ + V₃</td>
</tr>
<tr>
<td>LV (load vector)</td>
<td>V₁, R₁</td>
<td>Load V₁ with memory address location starting at address [R₁]</td>
</tr>
<tr>
<td>SV (store vector)</td>
<td>R₁, V₁</td>
<td>Store V₁ into memory starting at location [R₁]</td>
</tr>
</tbody>
</table>

* F₀ – a floating point number

NOTE: Each vector register (Rₙ) holds floating point numbers.
Timing

- A pipeline machine can initiate several instructions within 1 clock tick, which are then being executed in parallel.

Related Concepts:
- Convoys
- Chimes
Convoy

The set of vector instructions that could potentially begin execution together in one clock period.

Example:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LV</td>
<td>V1, Rx</td>
</tr>
<tr>
<td>MULTSV</td>
<td>V2, F0, V1</td>
</tr>
<tr>
<td>LV</td>
<td>V3, RY</td>
</tr>
<tr>
<td>ADDV</td>
<td>V4, V2, V3</td>
</tr>
<tr>
<td>SV</td>
<td>RY, V4</td>
</tr>
</tbody>
</table>

Load vector X  
Vector scalar multiplication  
Load vector X  
Add  
Storing results
Convoy

Note: `MULTSV V2, F0, V1 || LV V3, RY`

is an example of a convoy, where 2 independent instructions are initiated within same chime.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LV V1, Rx</td>
<td>Load vector X</td>
</tr>
<tr>
<td>MULTSV V2, F0, V1</td>
<td>Vector scalar multiplication</td>
</tr>
<tr>
<td>LV V3, RY</td>
<td>Load vector X</td>
</tr>
<tr>
<td>ADDV V4, V2, V3</td>
<td>Add</td>
</tr>
<tr>
<td>SV RY, V4</td>
<td>Storing results</td>
</tr>
</tbody>
</table>
Chime

- Not a specific amount of time, but rather a timing concept representing the number of clock periods required to complete a vector operation.

- CRAY-1 chime is 64 clock periods.

  - Note: CRAY-1 clock cycle takes 12.5 ns.
  5 chimes would take: \( 5 \times 64 \times 12.5 = 4000 \text{ ns} \)
Chime – Example #1

How many chimes will the vector sequence take?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LV</td>
<td>V1, Rx</td>
<td>Load vector X</td>
</tr>
<tr>
<td>MULTSV</td>
<td>V2, F0, V1</td>
<td>Vector scalar multiplication</td>
</tr>
<tr>
<td>LV</td>
<td>V3, RY</td>
<td>Load vector Y</td>
</tr>
<tr>
<td>ADDV</td>
<td>V4, V2, V3</td>
<td>Add</td>
</tr>
<tr>
<td>SV</td>
<td>RY, V4</td>
<td>Store result</td>
</tr>
</tbody>
</table>
Chime - Example #1

- ANSWER: 4 chimes

1\textsuperscript{st} chime : LV \ V_1, R_x
2\textsuperscript{nd} chime : MULTSV \ V_2, F_0, V_1 || LV \ V_3, R_Y
3\textsuperscript{rd} chime : ADDV \ V_4, V_2, V_3
4\textsuperscript{th} chime : SV \ R_Y, V_4

Note: MULTSV \ V_2, F_0, V_1 || LV \ V_3, R_Y
is an example of a convoy, where 2 independent instructions are initiated within same chime.
Chime - Example #2

- **CRAY-1**
  
  For \( i \leftarrow 1 \) to 64
  
  \[
  A[i] = 3.0 \times A[i] + (2.0 + B[i]) \times C[i]
  \]

- **To execute this:**
  
  1\(^{\text{st}}\) chime : \( V_0 \leftarrow A \)
  
  2\(^{\text{nd}}\) chime : \( V_1 \leftarrow B \)
  
  \[
  \begin{align*}
  V_3 &\leftarrow 2.0 + V_1 \\
  V_4 &\leftarrow 3.0 \times V_0
  \end{align*}
  \]
  
  3\(^{\text{rd}}\) chime : \( V_5 \leftarrow C \)
  
  \[
  \begin{align*}
  V_6 &\leftarrow V_3 \times V_5 \\
  V_7 &\leftarrow V_4 + V_6
  \end{align*}
  \]
  
  4\(^{\text{th}}\) chime : \( A \leftarrow V_7 \)

Can initiate operations to use array values immediately after they have been loaded into vector registers.
Chaining

Building dynamically a larger pipeline by increasing number of stages.
Chaining – Example #1

- For J ← 1 to 64

END

* No chaining - these are independent!!
Chaining – Example #2

- For J ← 1 to 64
  - D[J] ← C[J] * E[J]

END

Diagram:

- A and B connected to V0 and V1
- V0 connected to C (through multiplication)
- C and D connected through a sum (E)
- V2 and V3 connected through E
Latency

It takes 8 units to get the result to here.

V0 and V1 To functional unit

ADD TIME

Result to V2
More Chaining and Storing Matrices

Thanks to Dusty Price
Sequential Approach…

\[ V_0 \rightarrow + \rightarrow V_2 \rightarrow * \rightarrow V_4 \]

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
<th>6</th>
<th>1</th>
<th>1</th>
<th>7</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add</td>
<td></td>
<td></td>
<td>mul</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|       | 8 | 9 |

64 Elements in sequence: \[ T_s = 64 \times (8 + 9) = 1088 \]
Using Pipeline Approach…

Using pipelining it takes 8 units of time to fill pipeline and produce first result, each unit of time after that produces another result

\[ T_{p+} = 8 + 63 \]

The multiplication pipeline takes 9 units of time to fill, and produces another result after each additional unit of time

\[ T_{p*} = 9 + 63 \]

The combination of the two

\[ T_p = T_{p+} + T_{p*} = 8 + 63 + 9 + 63 = 143 \]
Using the chaining technique, we now have one pipeline. This new pipeline takes 17 units of time to fill, and produces another result after each unit of time.

\[ T_c = 17 + 63 = 80 \]
Review of time differences in the three approaches...

Sequential: $T_s = 17 \times 64 = 1088$

Pipelining: $T_p = 8 + 63 + 9 + 63 = 143$

Chaining: $T_c = 17 + 63 = 80$
Storing Matrixes for Parallel Access (Memory Interleaving)

Matrix

<table>
<thead>
<tr>
<th>A_{11}</th>
<th>A_{12}</th>
<th>A_{13}</th>
<th>A_{14}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{21}</td>
<td>A_{22}</td>
<td>A_{23}</td>
<td>A_{24}</td>
</tr>
<tr>
<td>A_{31}</td>
<td>A_{32}</td>
<td>A_{33}</td>
<td>A_{34}</td>
</tr>
<tr>
<td>A_{41}</td>
<td>A_{42}</td>
<td>A_{43}</td>
<td>A_{44}</td>
</tr>
</tbody>
</table>

4 Memory Modules

<table>
<thead>
<tr>
<th>M_1</th>
<th>M_2</th>
<th>M_3</th>
<th>M_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_{11}</td>
<td>A_{21}</td>
<td>A_{31}</td>
<td>A_{41}</td>
</tr>
</tbody>
</table>

One column of the matrix can be accessed in parallel.
Storing the Matrix by Column...

<table>
<thead>
<tr>
<th>Matrix</th>
<th>4 Memory Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_{11} ) A_{12} A_{13} A_{14}</td>
<td>( M_1 ) ( M_2 ) ( M_3 ) ( M_4 )</td>
</tr>
<tr>
<td>( A_{21} ) A_{22} A_{23} A_{24}</td>
<td>( A_{11} ) ( A_{12} ) ( A_{13} ) ( A_{14} )</td>
</tr>
<tr>
<td>( A_{31} ) A_{32} A_{33} A_{34}</td>
<td>( A_{21} ) ( A_{22} ) ( A_{23} ) ( A_{24} )</td>
</tr>
<tr>
<td>( A_{41} ) A_{42} A_{43} A_{44}</td>
<td>( A_{31} ) ( A_{32} ) ( A_{33} ) ( A_{34} )</td>
</tr>
</tbody>
</table>

One Row can be accessed in parallel with this storage technique.
Sometimes we need to access both rows and columns fast…

By using a skewed matrix representation, we can now access each row and each column in parallel.
Sometimes we need access to the main diagonal as well as rows and columns…

At the cost of adding another memory module and wasted space, we can now access the matrix in parallel by row, column, and main diagonal.
Program Transformation

FOR I ← 1 TO n do
    X ← A[I] + B[I]
    .
    .
    Y[I] ← 2 * X
    .
    .
    X ← C[I] / D[I]
    .
    .
    P ← X + 2
ENDFOR

⇒ removes data dependency

data dependency

FOR I ← 1 TO n do
    X ← A[I] + B[I]
    .
    .
    Y[I] ← 2 * X
    .
    .
    XX ← C[I] / D[I]
    .
    .
    P ← XX + 2
ENDFOR
Scalar Expansion

FOR I ← 1 TO n do
  X ← A[I] + B[I]
  .
  .
  Y[I] ← 2 * X
  .
  .
ENDFOR

⇒

FOR I ← 1 TO n do
  X ← A[I] + B[I]
  .
  .
  Y[I] ← 2 * X[I]
  .
  .
ENDFOR

removes data dependency
Loop Unrolling

FOR I ← 1 TO n do
ENDFOR

⇒

\[
\begin{align*}
\vdots \\
X[n] & \leftarrow A[n] \times B[n]
\end{align*}
\]
Loop Fusion or Jamming

FOR I ← 1 TO n do
  X[I] ← Y[I] * Z[I]
ENDFOR
FOR I ← 1 TO n do
  M[I] ← P[I] + X[I]
ENDFOR

a) FOR I ← 1 TO n do
    X[I] ← Y[I] * Z[I]
    M[I] ← P[I] + X[I]
ENDFOR
b) FOR I ← 1 TO n do
    M[I] ← P[I] + Y[I] * Z[I]
ENDFOR