Vector Processors

- Chapter F of the 4th edition (Chapter G of the 3rd edition)
  - Available in CD attached to the book or
  - available online at [http://www.mkp.com/ca3](http://www.mkp.com/ca3)
    - North America -> companion site
  - Anybody having problems to find it should contact me
- Vector processors big in ’70 and ’80
- Still used today
  - Vector machines: Earth Simulator, NEC SX8, Cray X1
  - Graphics cards
  - MMX, SSE, SSE2 are to some extent ‘vector units’
Main concepts

- Vector processors abstract operations on vectors, e.g. replace the following loop
  
  ```
  for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
  }
  ```

  by
  
  ```
  a = b + c; ADDV.D V10, V8, V6
  ```

- Some languages offer high-level support for these operations (e.g. Fortran90 or newer)

Main concepts (II)

- Advantages of vector instructions
  - A single instruction specifies a great deal of work
  - Since each loop iteration must not contain data dependence to other loop iterations
    - No need to check for data hazards between loop iterations
    - Only one check required between two vector instructions
    - Loop branches eliminated
Basic vector architecture

- A modern vector processor contains
  - Regular, pipelined scalar units
  - Regular scalar registers
  - Vector units - (inventors of pipelining! )
  - Vector register: can hold a fixed number of entries (e.g. 64)
  - Vector load-store units

Comparison MIPS code vs. vector code

Example: \( Y = aX + Y \) for 64 elements

```
L.D  F0, a       /* load scalar a*/
DADDIU R4, Rx, #512 /* load X(i) */
L:  L.D  F2, 0(Rx) /* last address */
MUL.D F2, F2, F0  /* calc. a times X(i)*/
L.D  F4, 0(Ry)   /* load Y(i) */
ADD.D F4, F4, F2 /* aX(i) + Y(i) */
S.D  F4, 0(Ry)   /* store Y(i) */
DADDIU Rx, Rx, #8 /* increment X*/
DADDIU Ry, Ry, #8 /* increment Y */
DSUBU R20, R4, Rx /* compute bound */
BNEZ R20, L
```
Comparison MIPS code vs. vector code (II)

Example: \( Y = aX + Y \) for 64 elements

```
L.D    F0, a        /* load scalar a*/
LV     V1, 0(Rx)   /* load vector X */
MULVS.D V2, V1, F0 /* vector scalar mult*/
LV     V3, 0(Ry)   /* load vector Y */
ADDV.D V4, V2, V3 /* vector add */
SV     V4, 0(Ry)   /* store vector Y */
```

Vector execution time

- **Convoy**: set of vector instructions that could potentially begin execution in one clock cycle
  - A convoy must not contain structural or data hazards
  - Similar to VLIW
  - Initial assumption: a convoy must complete before another convoy can start execution
- **Chime**: time unit to execute a convoy
  - Independent of the vector length
  - A sequence consisting of \( m \) convoys executes in \( m \) chimes
  - A sequence consisting of \( m \) convoys and vector length \( n \) takes approximately \( mn \) clock cycles
Example

- Convoys of the above code sequence:
  1. LV
  2. MULVS.D LV
  3. ADDV.D
  4. SV

Overhead

- Start-up overhead of a pipeline: how many cycles does it take to fill the pipeline before the first result is available?

<table>
<thead>
<tr>
<th>Unit</th>
<th>Start-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/store</td>
<td>12</td>
</tr>
<tr>
<td>Multiply</td>
<td>7</td>
</tr>
<tr>
<td>Add</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Convoy</th>
<th>Starting time</th>
<th>First result</th>
<th>Last result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LV</td>
<td>0</td>
<td>12</td>
<td>11+n</td>
</tr>
<tr>
<td>MULVS</td>
<td>12+n</td>
<td>12+n+12</td>
<td>23+2n</td>
</tr>
<tr>
<td>ADDV</td>
<td>24+2n</td>
<td>24+2n+6</td>
<td>29+3n</td>
</tr>
<tr>
<td>SV</td>
<td>30+3n</td>
<td>30+3n+12</td>
<td>41+4n</td>
</tr>
</tbody>
</table>
Pipelining - Metrics (I)

\( T_c \)  Clocktime, time to finish one segment/sub-operation  
\( m \)  number of stages of the pipeline  
\( n \)  length of the vector  
\( S \)  Startup time in clocks, time after which the first result is available, \( S = m \)  
\( N_{\frac{1}{2}} \)  length of the loop to achieve half of the maximum speed  

Assuming a simple loop such as:

```c
for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
}
```

Pipelining - Metrics (II)

\( op \)  Number of operations per loop iteration  
\( op_{total} \)  total number of operations for the loop, with  
\[ op_{total} = op \times n \]

Speed of the loop is

\[ F = \frac{op_{total}}{time} = \frac{op \times n}{T_c (m + (n-1))} \]

For \( n \to \infty \) we get

\[ F_{max} = \frac{op}{T_c} \]
Pipelining - Metrics (III)

Because of the Definition of \( N_{1/2} \) we now have

\[
\frac{op}{T_c \left( \frac{m-1}{N_{1/2}} + 1 \right)} = \frac{1}{2} F_{\text{max}} = \frac{1}{2} \frac{op}{T_c}
\]

or

\[
\frac{m-1}{N_{1/2}} + 1 = 2
\]

and

\[
N_{1/2} = m - 1
\]

⇒ length of the loop required to achieve half of the theoretical peak performance of a pipeline is equal to the number of segments (stages) of the pipeline.

Pipelining - Metrics (IV)

More general: \( N_{\alpha} \) is defined through

\[
\frac{op}{T_c \left( \frac{m-1}{N_{\alpha}} + 1 \right)} = \alpha \frac{op}{T_c}
\]

and leads to

\[
N_{\alpha} = \frac{m-1}{\frac{1}{\alpha} - 1}
\]

E.g. for \( \alpha = \frac{3}{4} \) you get

\[
N_{3/4} = 3(m-1) \approx 3m
\]

⇒ the closer you would like to get to the maximum performance of your pipeline, the larger the iteration counter of your loop has to be.
Vector length control

- What happens if the length is not matching the length of the vector registers?
- A vector-length register (VLR) contains the number of elements used within a vector register
- Strip mining: split a large loop into loops less or equal the maximum vector length (MVL)

Vector length control (II)

```c
low = 0;
VL = (n mod MVL);
for (j=0; j < n/MVL; j++) {
    for (i=low; i < VL; i++) {
        Y(i) = a * X(i) + Y(i);
    }
    low += VL;
    VL = MVL;
}
```
Vector stride

- Memory on vector machines typically organized in multiple banks
  - Allow for independent management of different memory addresses
  - Memory bank time an order of magnitude larger than CPU clock cycle
- Example: assume 8 memory banks and 6 cycles of memory bank time to deliver a data item
  - Overlapping of multiple data requests by the hardware
Vector stride (II)

- What happens if the code does not access subsequent elements of the vector
  ```c
  for (i=0; i<n; i+=2) {
      a[i] = b[i] + c[i];
  }
  ```

  - Vector load ‘compacts’ the data items in the vector register (gather)
    - No affect on the execution of the loop
    - You might however use only a subset of the memory banks -> longer load time
    - Worst case: stride is a multiple of the number of memory banks

Summary 1: Vector processors

- Support for operations on vectors using ‘special’ instructions
  - Each iteration has to be data independent from other iterations
  - Multiple vector instructions are organized in convoy’s in the absence of data and structural hazards
  - Each convoy can be executed in one chime
  - So far: assume that a convoy has to finish before another convoy can start
    - Start-up cost of a pipeline
    - Strip-mining costs in case loop iteration count does not match the length of the vector registers
Enhancing Vector Performance

- Five techniques to further improve the performance
  - Chaining
  - Conditional Execution
  - Support for sparse matrices
  - Multiple lanes
  - Reducing start-up costs by pipelining

Chaining

- Example:
  
  MULV.D V1, V2, V3
  ADDV.D V4, V1, V5

- Second instruction has a data dependence on the first instruction: two convoys required
- Once the element V1(i) is has been calculated, the second instruction could calculate V4(i)
  - no need to wait until all elements of V1 are available
  - could work similarly as forwarding in pipelining
  - Technique is called chaining
Chaining (II)

- Recent implementations use *flexible chaining*
  - Vector register file has to be accessible by multiple vector units simultaneously
- Chaining allows operations to proceed in parallel on separate elements of vectors
  - Operations can be scheduled in the same convoy
  - Reduces the number of chimes
  - Does not reduce the startup-overhead

Chaining (III)

- Example: chained and unchained version of the **ADDV.D** and **MULV.D** shown previously for 64 elements
  - Start-up latency for the FP MUL vector unit: 7 cycles
  - Start-up latency for FP ADD vector unit: 6 cycles
  - NOTE: different results than in book in fig. G.10

  Unchained version:
  \[ 7 + 63 + 6 + 63 = 139 \text{ cycles} \]

  Chained version:
  \[ 7 + 6 + 63 = 76 \text{ cycles} \]
Conditional execution

- Consider the following loop
  ```c
  for (i=0; i< N; i++) {
    if ( A(i) != 0 ) {
      A(i) = A(i) - B(i);
    }
  }
  ```
- Loop can usually not been vectorized because of the conditional statement
- Vector-mask control: boolean vector of length MLV to control whether an instruction is executed or not
  - Per element of the vector

Conditional execution (II)

```
LV       V1, Ra    /* load vector A into V1 */
LV       V2, Rb    /* load vector B into V2 */
L.D      F0, #0    /* set F0 to zero */
SNEVS.D  V1, F0    /* set VM(i)=1 if V1(i)!=F0 */
SUBV.D   V1, V1, V2 /* sub using vector mask*/
CVM      /* clear vector mask to 1 */
SV       V1, Ra    /* store V1 */
```
Support for sparse matrices

- Access of non-zero elements in a sparse matrix often described by
  \[ A(K(i)) = A(K(i)) + C(M(i)) \]
  - \( K(i) \) and \( M(i) \) describe which elements of \( A \) and \( C \) are non-zero
  - Number of non-zero elements have to match, location not necessarily
- Gather-operation: take an index vector and fetch the according elements using a base-address
  - Mapping from a non-contiguous to a contiguous representation
- Scatter-operation: inverse of the gather operation

Support for sparse matrices (II)

```
LV  Vk, Rk /* load index vector K into V1 */
LVI Va, (Ra+Vk) /* Load vector indexed A(K(i)) */
LV  Vm, Rm /* load index vector M into V2 */
LVI Vc, (Rc+Vm) /* Load vector indexed C(M(i)) */
ADDV.D Va, Va, Vc /* set VM(i)=1 if V1(i)!=F0 */
SVI Va, (Ra+Vk) /* store vector indexed A(K(i)) */
```

- Note:
  - Compiler needs the explicit hint, that each element of \( K \) is pointing to a distinct element of \( A \)
  - Hardware alternative: a hash table keeping track of the address acquired
    - Start of a new vector iteration (convoy) as soon as an address appears the second time
Multiple Lanes

- Further performance improvements if multiple functional units can be used for the same vector operation

\[
\begin{array}{ccc}
\end{array}
\]

Pipelined instruction startup

- Start of one vector instruction can overlap with another vector instruction
  - Theoretically: instruction of new vector operation could start in the next cycle after the last instruction of the previous operation
  - Practically: *dead time* between two instructions, in order to simplify the logic of the pipeline
    - E.g. 4 cycles before the next vector instruction can start
## Effectiveness of Compiler Vectorization

<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>Operations executed in vector mode, compiler-optimized</th>
<th>Operations executed in vector mode, hand-optimized</th>
<th>Speedup from hand optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDNA</td>
<td>96.1%</td>
<td>97.2%</td>
<td>1.52</td>
</tr>
<tr>
<td>MG3D</td>
<td>95.1%</td>
<td>94.5%</td>
<td>1.00</td>
</tr>
<tr>
<td>FLO82</td>
<td>91.5%</td>
<td>88.7%</td>
<td>N/A</td>
</tr>
<tr>
<td>ARC3D</td>
<td>91.1%</td>
<td>92.0%</td>
<td>1.01</td>
</tr>
<tr>
<td>SPEC77</td>
<td>90.3%</td>
<td>90.4%</td>
<td>1.07</td>
</tr>
<tr>
<td>MDG</td>
<td>87.7%</td>
<td>94.2%</td>
<td>1.49</td>
</tr>
<tr>
<td>TRFD</td>
<td>69.8%</td>
<td>73.7%</td>
<td>1.67</td>
</tr>
<tr>
<td>DYFESM</td>
<td>68.8%</td>
<td>65.6%</td>
<td>N/A</td>
</tr>
<tr>
<td>ADM</td>
<td>42.9%</td>
<td>59.6%</td>
<td>3.60</td>
</tr>
<tr>
<td>OCEAN</td>
<td>42.8%</td>
<td>91.2%</td>
<td>3.92</td>
</tr>
<tr>
<td>TRACK</td>
<td>14.4%</td>
<td>54.6%</td>
<td>3.52</td>
</tr>
<tr>
<td>SPICE</td>
<td>11.5%</td>
<td>79.9%</td>
<td>4.06</td>
</tr>
<tr>
<td>OCD</td>
<td>4.2%</td>
<td>75.1%</td>
<td>2.15</td>
</tr>
</tbody>
</table>