Vector chaining

[H&P §B.5] Consider this sequence:

\[
\begin{align*}
\text{ADDVV}1,2,3 \\
\text{MULTVV}4,1,5
\end{align*}
\]

On an \(n\)-element vector, this sequence would take time

\[
\begin{align*}
&\quad nT_{\text{element}} + T_{\text{startup(ADDV)}} + T_{\text{stall}} + T_{\text{startup(MULTV)}} \\
&= 2n + 6 + 4 + 7 \\
&= 2n + 17
\end{align*}
\]

Because the ADDV uses a result from the MULTV, the MULTV has to finish before the ADDV can start.

This would not be necessary if the elements of the vectors could be considered individually.

Then the ADDV could start as soon as the MULTV produced its first result.

The initiation rate would then be one per cycle if the functional units were fully pipelined.

This would allow a sustained rate of 2 operations per clock cycle to be achieved, even though the operations are dependent.

Here is a comparison of the timings with \(n = 64\):

Unchained

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>64</th>
<th>0</th>
<th>6</th>
<th>64</th>
<th>0</th>
<th>6</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>VADD</td>
<td>VADD</td>
<td>VMULT</td>
<td>VADD</td>
<td>VADD</td>
<td>VMULT</td>
<td>VADD</td>
<td>VADD</td>
<td>VMULT</td>
</tr>
</tbody>
</table>

Chained

<table>
<thead>
<tr>
<th>0</th>
<th>6</th>
<th>64</th>
<th>0</th>
<th>6</th>
<th>64</th>
<th>0</th>
<th>6</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>VADD</td>
<td>VADD</td>
<td>VMULT</td>
<td>VADD</td>
<td>VADD</td>
<td>VMULT</td>
<td>VADD</td>
<td>VADD</td>
<td>VMULT</td>
</tr>
</tbody>
</table>
Masking

Just as in array processors, if statements in loops get in the way of vector processors.

Also, sparse matrices can’t be operated on efficiently unless they can be vectorized. But we have not yet studied techniques to do this.

As in our sample array processor, a mask register can be used on a vector processor.

Consider this loop:

```plaintext
for i := 1 to 64 do
```

This loop cannot be vectorized normally because of the conditional execution of the body.

It can be vectorized if the body can be stopped from executing when A[i] = 0.

The *vector-mask register* VM holds a boolean vector of length MVL. Vector operations take place only on elements whose corresponding bits in VM are 1.

Clearing VM sets it to 1, so that no elements are masked off.

In the code below, assume that the starting addresses of A and B are in RA and RB.

1. LV V1,RA Load vector A into V1.
2. LV V2, RB Load vector B into V2.
3. LD F0,#0 Load floating-point 0 into F0.
4. SNESV F0,V1 Sets VM bit to 1 if V1[i] ≠ 0.
5. SUBV V1,V1,V2 Subtract under vector mask.
6. CVM Set the vector mask to all 1s.
7. SV RA,V1 Store the result in vector A.

What are some disadvantages of using a vector mask (relative to not doing the masked-off operations at all)?
Sparse-matrix techniques

[H&P §B.6] In some matrix problems, for example, finite-element problems, sparse matrices occur frequently.

(A sparse matrix is one in which most elements are \text{________}.)

In a two-dimensional continuum program, the typical code accesses only the four nearest non-zero neighbors.

It is more efficient to deal with just the non-zero elements, but they are not distributed predictably through the array. Hence,

- rather extensive communication structures may be needed for efficiency, and
- the interconnections may not be able to be used in parallel.

Here are two approaches to handling sparse matrices.

The CDC Star used \textit{sparse vectors}, which consisted of two vectors:

- a short vector containing just the non-zero elements of the vector, and
- a bit-vector whose 1s indicate where the non-zero elements belong. This has one bit for each non-zero \textit{and} zero element of the vector.

The bit-vector is used to determine which memory words must be accessed. The number of accesses is reduced, but memory conflicts are not prevented.

Clearly, the instruction set can include instructions to translate from sparse vectors to full vectors and back again. Pipelines can be designed to do conversions on the fly.
However, this approach can never give a compaction to less than 1/64th of the full vector.

An alternative approach is—

- again to store only the non-zero elements, but
- to use an array of *indices* into the original matrix.

Here is an example using an index vector

\[
\text{for } i := 1 \text{ to } n \text{ do } \\
A[K[i]] := A[K[i]] + C[M[i]]; \\
\]

Index vectors K and M are used to point to nonzero elements of A and C. (A and C must have the same number of nonzero elements.)

The FPS-164 contains \(T\) registers in its scalar processor that hold the indices of the non-zero elements of a sparse matrix.

*Scatter-gather* operations are usually used to implement sparse matrices.

- The *gather* operation takes an index vector, and fetches the vector whose elements are at the addresses given by adding
  
  \[\text{base} + \text{offsets in index vector}.\]

- The *scatter* operation expands a vector from a register and puts it in memory at the addresses specified by the base and index vector.

The DLXV instructions for scatter-gather are \texttt{LVI (Load Vector Indexed)} and \texttt{SVI (Store Vector Indexed)}.

Assume that the \(A\), \(C\), \(K\), and \(M\) vectors in the sequence above are held at addresses \(RA\), \(RC\), \(RK\), and \(RM\). The loop above can be coded—
1. LV VK,RK Load $K$.
3. LV VMR,RM Load $M$. (not VM; that is the vector mask.)
4. LVI VC,(RC+VMR) Load $C[M[i]]$.
5. ADDV VA,VA,VC Add the vectors.
6. SVI (RA+VK),VA Store $A[K[i]]$.

It is hard or impossible for the compiler to find hazards in the above code. Why?

Scatter-gather operations rarely run at one cycle per clock, but they are much faster than the alternatives.

If the non-zero elements change, a new vector must be computed quickly.

On DLXV, a CVI (Create Vector Index) instruction creates an index vector, given a stride $m$, containing the values 0, $m$, 2$m$, … , 63$m$.

Like some other machines, DLXV provides an instruction to create a compressed index vector whose entries correspond to the positions with a 1 in the mask register.

When VM is all 1s, a “standard” index vector is created. Here is vector code for the subtraction loop two pages back:

1. LV V1,RA Load vector $A$ into $V1$.
2. LD F0,#0 Load fl.-pt. 0 into $F0$.
3. SNESV F0,V1 Sets the VM to 1 if $V1[i] \neq 0$.
4. CVI V2,#8 Generates indices in $V2$.
5. POP R1,VM Find the number of 1s in VM.
6. MOV12S VLR,R1 Load vector length register.
7. CVM
8. LVI V3,(RA+V2) Load the non-0 elts. of $A$.
9. LVI V4,(RB+V2) Load corresponding elts. of $B$
10. SUBV V3,V3,V4 Perform the subtraction.
11. SVI V3,(RA+V2) Store $A$ back.
Whether scatter-gather is better than conditional execution depends on—
  • how often the condition holds, and
  • the cost of the operations.

The running time of the masking version is

Let $f$ be the fraction of elements for which the condition is true (i.e., $A[i] \neq 0$). Then the running time of the scatter-gather version is

Let's assume that the constant overheads $c_1$ and $c_2$ are comparable (or negligible). Then we can calculate when

$$5n > 3n + 4fn$$

(What does this mean? That scatter-gather is better.)

This occurs when $2n > 4fn$, or $n > 1/2$.

Thus, scatter-gather is better if ____________ of the elements are nonzero.

If the index vector can be reused, or if the number of vector operations in the then clause grows, the advantage of scatter/gather will increase sharply.

*Storing index vectors instead of zero elements*: How efficient is it to fetch the value of an arbitrary matrix element using index vectors?

Consequently, an inverted representation of this matrix maps indices to pointers.

The lookup can then be done by hashing.

  • Hashing is very similar to a(n associative) cache access.
  • It can be pipelined too, so this method is potentially useful in pipelined computers.
Non-zero items only can be fetched and processed.

The cost of skipping elements and memory contention can be very small compared to the speedup by processing only non-zero elements.

**Vector reduction**

Recursive doubling is an application that is cannot be vectorized straightforwardly.

It is a special case of *reduction*—a loop that reduces an array to a single value by repeated application of an operation.

A common example is a dot product:

\[
\begin{align*}
\text{dot} & := 0.0; \\
\text{for } i := 1 \text{ to } 64 \text{ do} \\
\text{dot} & := \text{dot} + A[i] \times B[i];
\end{align*}
\]

This loop cannot be vectorized without some changes. Why?

That means a pipeline interlock comes into play.

How can the delay be avoided?

This form of the computation is not delayed by any interlocks because \textit{dot}[i–k] is not needed as an input until just after it emerges from the output of the pipeline.

But this strategy requires a slightly more complicated algorithm. Why?

But the performance degradation is negligible when the A and B vectors are long.
The code given in Hennessy and Patterson starts off with 64 partial sums (though this may be overdoing it).

After the partial sums are computed, recursive doubling is used:

\[
dot := 0;
\]
\[
\textbf{for } i := 1 \textbf{ to } 64 \textbf{ do }
\]
\[
dot[i] := A[i] \times B[i];
\]
\[
\text{len} := 32;
\]
\[
\textbf{for } j := 1 \textbf{ to } 6 \textbf{ do }
\]
\[
\textbf{begin}
\]
\[
\textbf{for } i := 1 \textbf{ to } \text{len } \textbf{ do }
\]
\[
dot[i] := \dot{dot}[i] + \dot{dot}[i+len];
\]
\[
\text{len} := \text{len} / 2;
\]
\[
\textbf{end};
\]