Floating-point performance of the i860

D. Chase

December 4, 2001

1 Exposed pipelines

Best floating-point performance is obtained from the Intel 80860 if the dual-mode pipelined floating point instructions are used. At first I found these rather weird, so I'm including this example to ease the learning curve.

Pipeline instructions feed new operands into the pipeline and extract results from the pipeline. Each pipeline instruction advances the associated pipelines one stage. Use of pipeline instructions typically is divided into three phases called "priming", "continuous operation", and "flushing". The example in figure 1 from the Programmer's Reference Manual illustrates the use of pipelined instructions to obtain the inner product of vectors x and y stored in `f4` through `f11` and `f12` through `f19`. The instruction `ml2apm` is especially suited to computing inner products. The input operands are multiplied together, and the output of the multiplier (contents of final stage prior to executing the instruction) is added together with the output of the adder. That is, the accumulator circulates through the adder pipeline. One unfortunate aspect of this is that the order of the additions is rearranged; this is generally not a good idea for numerical code.

2 A simple example

The most instructive example is provided in the Programmer's Reference Manual on pages 9-12 and 9-13 ("Matrix multiply, cached and pipelined loads"). This example performs the calculation for $C \leftarrow AB^T$, for A an $L \times M$ matrix, B an $M \times N$ matrix, and C conforming. The problem is simplified "to eliminate needless complexity" by assuming that M is a multiple of 8 and that B is stored transposed. Below is the inner loop of this matrix multiplication; this code takes the inner product of a row of A

instructions	Multiplier Stages 1	2	3	1	Adder Stages 2	3	Result
				Priming phase			
ml2apm.ss f4,f12,f0	x_1y_1	?	?	?	?	?	Discard
ml2apm.ss f5,f13,f0	x_2y_2	x_1y_1	?	?	?	?	Discard
ml2apm.ss f6,f14,f0	x_3y_3	x_2y_2	x_1y_1	?	?	?	Discard
pfadd.ss f0,f0,f0				0	?	?	Discard
pfadd.ss f0,f0,f0				0	0	?	Discard
pfadd.ss f0,f0,f0				0	0	0	Discard
				Continuous phase			
ml2apm.ss f7,f15,f0	x_4y_4	x_3y_3	x_2y_2	x_1y_1	0	0	Discard
ml2apm.ss f8,f16,f0	x_5y_5	x_4y_4	x_3y_3	x_2y_2	x_1y_1	0	Discard
ml2apm.ss f9,f17,f0	x_6y_6	x_5y_5	x_4y_4	x_3y_3	x_2y_2	x_1y_1	Discard
ml2apm.ss f10,f18,f0	x_7y_7	x_6y_6	x_5y_5	$x_1y_1+x_4y_4$	x_3y_3	x_2y_2	Discard
ml2apm.ss f11,f19,f0	x_8y_8	x_7y_7	x_6y_6	$x_2y_2+x_5y_5$	$x_1y_1+x_4y_4$	x_3y_3	Discard
				Flushing phase			
ml2apm.ss f0,f0,f0	0	x_8y_8	x_7y_7	$x_3y_3+x_6y_6$	$x_2y_2+x_5y_5$	$x_1y_1+x_4y_4$	Discard
ml2apm.ss f0,f0,f0	0	0	x_8y_8	$x_1y_1+x_4y_4+x_7y_7$	$x_3y_3+x_6y_6$	$x_2y_2+x_5y_5$	Discard
ml2apm.ss f0,f0,f0	0	0	0	$x_2y_2+x_5y_5+x_8y_8$	$x_1y_1+x_4y_4+x_7y_7$	$x_3y_3+x_6y_6$	Discard
pfadd.ss f0,f0,f20				0	$x_2y_2+x_5y_5+x_8y_8$	$x_1y_1+x_4y_4+x_7y_7$	$f20_1 \leftarrow x_3y_3+x_6y_6$
pfadd.ss f20,f21,f21				$f20_1+f21_1$	0	$x_2y_2+x_5y_5+x_8y_8$	$f21_1 \leftarrow x_1y_1+x_4y_4+x_7y_7$
pfadd.ss f0,f0,f20				0	$f20_1+f21_1$	0	$f20_2 \leftarrow x_2y_2+x_5y_5+x_8y_8$
pfadd.ss f0,f0,f0				0	0	$f20_1+f21_1$	Discard
pfadd.ss f0,f0,f20				0	0	0	$f21_2 \leftarrow f20_1+f21_1 = (x_3y_3+x_6y_6)+(x_1y_1+x_4y_4+x_7y_7)$
fadd.ss f20,f21,f20							$f20_3 \leftarrow f21_2+f20_2 = ((x_3y_3+x_6y_6)+(x_1y_1+x_4y_4+x_7y_7))+(x_2y_2+x_5y_5+x_8y_8)$

Figure 1: Inner product from registers

with a row of B. The code to start up and shut down the floating point pipeline is not shown, since its cost is relatively insignificant for large enough matrices.

floating instructions	integer instructions	notes
`inner_loop:`		
`d.ml2apm.ss f4, f12, f0`	`fld.q 16(r29)++, f8`	Load A
`d.ml2apm.ss f5, f13, f0`	`pfld.d 8(r24)++, f16`	Load B
`d.ml2apm.ss f6, f14, f0`	`pfld.d 8(r24)++, f18`	Load B
`d.ml2apm.ss f7, f15, f0`	`fld.q 16(r29)++, f4`	Load A
`d.ml2apm.ss f8, f16, f0`	`nop`	
`d.ml2apm.ss f9, f17, f0`	`pfld.d 8(r24)++, f12`	Load B
`d.ml2apm.ss f10, f18, f0`	`bla r27,r28, inner_loop`	
`d.ml2apm.ss f11, f19, f0`	`pfld.d 8(r24)++, f14`	Load B

This code has been carefully arranged to avoid stalling for register contents. In addition, the `fld` and `pfld` instructions have been somewhat interleaved in order to avoid stalling within the load pipeline. Note that cache misses will slow down the `fld` instructions, and that cache *hits* appear to slow down the `pfld` instructions (PRM, page C-2); therefore, it is very important that the elements of the A matrix be loaded with `fld` (which fills the cache and is optimized for cache hit) and that the elements of the B matrix be loaded with `pfld` (which does *not* fill the cache, and is optimized for cache miss). Each row of A will be loaded into the cache once, and remain in the cache for N inner product operations with columns of B (each of these will yield one element of C). Each element of B is loaded fresh from memory each time it is needed, but using pipelined loads that will not disrupt the cache.

Note, too, how important it is that the elements of A and B are stored in such a way that several elements can be fetched with a single instruction. Each iteration of this loop executes 8 floating-point instructions where each requires two input operands, with none reused, for a total of 16 input operands per loop iteration. If the code is to run at maximum floating point speed, then these 16 operands must somehow be fetched in just 7 core unit instructions.

The result of this careful programming is that the bandwidth to memory is cut in half for large values of N provided that M (the number of columns in A) is small enough for an entire row of A to fit into the cache[1]. It is

[1] 8192 bytes of data cache could contain as many as 2048 single-precision floating point numbers; to provide a little perspective, note that even at 80Mflops the usual algorithm takes 2000 seconds to multiply together two 2048×2048 matrices.

quite reasonable to expect that every 8 clock ticks the code in the example above will perform 8 floating point multiplies, 8 floating point additions, 2 loads totalling 8 words from cache, 4 pipelined loads totalling 8 words from memory, 7 register increments, one register comparison, and one branch. One could reasonably claim that the 80860 delivers 120 million operations per second running this code. Hand-coded versions of floating point library routines (e.g., the BLAS and the GLAS, for Fortran) should run quite well.

3 More complicated example

As noted above, maximum floating point speed requires a steady stream of operands delivered by the core unit. This is easy when several operands can be loaded in a single instruction, but tricky when they cannot.

An example where this is necessary is the row-by-column inner product in standard matrix multiplication (as opposed to row-by-row in the earlier example). Typical code to perform this multiplication looks like:

```
for i ← 1 to l do
    for j ← 1 to n do
        x ← 0.0
        for k ← 1 to m do
            x ← x + A[i, k] × B[k, j]
            end
        C[i, j] ← x
        end
    end
```

The inner loop can be unrolled 4 or 8 times to make use of multi-word loads to obtain elements of A, but the elements of B are no longer accessed in a consecutive fashion and must be loaded individually. For N floating point instructions, this will require $N/4$ instruction to load elements of A, 1 instruction for the test and branch, and N instructions to load elements of B. That is, the inner loop will require $N + N/4 + 1$ instructions to accomodate the operand demands of N floating point instructions. For N equal to 8, this delivers only 72% (8/11) of the potential floating point performance.

Note that this is only true if operands are not reused. If each load of an element of B were used twice, then we'd be ok. However, each inner product operation uses each element only once. Therefore, the inner loop must calculate several inner products at once. That is, code equivalent to the following might be used:

```
for i ← 1 to l by 3 do
    for j ← 1 to n do
        x_0 ← 0.0
        x_1 ← 0.0
        x_2 ← 0.0
        for k ← 1 to m by 4 do
            x_0 ← x_0 + A[i + 0, k + 0] × B[k + 0, j]
            x_1 ← x_1 + A[i + 1, k + 0] × B[k + 0, j]
            x_2 ← x_2 + A[i + 2, k + 0] × B[k + 0, j]
            x_0 ← x_0 + A[i + 0, k + 1] × B[k + 1, j]
            x_1 ← x_1 + A[i + 1, k + 1] × B[k + 1, j]
            x_2 ← x_2 + A[i + 2, k + 1] × B[k + 1, j]
            x_0 ← x_0 + A[i + 0, k + 2] × B[k + 2, j]
            x_1 ← x_1 + A[i + 1, k + 2] × B[k + 2, j]
            x_2 ← x_2 + A[i + 2, k + 2] × B[k + 2, j]
            x_0 ← x_0 + A[i + 0, k + 3] × B[k + 3, j]
            x_1 ← x_1 + A[i + 1, k + 3] × B[k + 3, j]
            x_2 ← x_2 + A[i + 2, k + 3] × B[k + 3, j]
            end
        C[i + 0, j] ← x_0
        C[i + 1, j] ← x_1
        C[i + 2, j] ← x_2
        end
    end
```

The outer (i) loop iterates over rows three at a time, the second loop (j) iterates over columns, and the inner (k) loop accumulates the inner products. The inner products are accumulated into x_0, x_1, and x_2. The reason that three, and not two or four, inner products are accumulated simultaneously is that the variables x_i are not mapped to registers or memory; instead, they circulate around in the adder pipeline. Because the adder pipeline has three stages, the number of intermediate results stored in it must divide three.

Here is some assembly code which might be used to implement the inner loop of the multiplication above.

floating instructions	integer instructions	notes
`inner_loop:`		
`d.ml2apm.ss f6,f16,f0`	`nop`	
`d.ml2apm.ss f10,f16,f0`	`nop`	
`d.ml2apm.ss f14,f16,f0`	`pfld.s r20(r19)++, f18`	
`d.ml2apm.ss f7,f17,f0`	`16(r16)++, fld.q f4`	
`d.ml2apm.ss f11,f17,f0`	`16(r17)++, fld.q f8`	
`d.ml2apm.ss f15,f17,f0`	`16(r18)++, fld.q f12`	
`d.ml2apm.ss f4,f18,f0`	`pfld.s r20(r19)++, f19`	
`d.ml2apm.ss f8,f18,f0`	`nop`	
`d.ml2apm.ss f12,f18,f0`	`pfld.s r20(r19)++, f16`	
`d.ml2apm.ss f5,f19,f0`	`nop`	
`d.ml2apm.ss f9,f19,f0`	`bla r21, r22, inner_loop`	
`d.ml2apm.ss f13,f19,f0`	`pfld.s r20(r19)++, f17`	

The setup code for the loop must ensure that registers `f6`, `f7`, `f10`, `f11`, `f14` and `f15` are loaded from A and that registers `f16` and `f17` are loaded from B. Register `r19` contains the address of the next element of B to load and register `r20` contains the difference in addresses for $B[k, j]$ and $B[k + 1, j]$. Registers `r16`, `r17` and `r18` contain the addresses of the next elements to load from the rows of A, and registers `r21` and `r22` contain values for loop control[2].

Matrix A will be "tiled" by the matrix

f6	f7	f4	f5
f10	f11	f8	f9
f14	f15	f12	f13

and matrix B will be tiled by the matrix

f16
f17
f18
f19

In each iteration of the inner loop the two tiles are multiplied together. Each iteration of the loop also loads registers to set up for the next iteration. If it is possible for either matrix to be adjacent to a protected page, then the

[2]The `bla` instruction is "interesting".

loop must terminate early to avoid loading from protected locations. In addition, care must be taken to ensure that the loads are aligned on 128-bit boundaries or else a trap will occur. This constraint is *not* satisfied by the code above if the number of columns in A is not a multiple of 4 (and the rows are not padded). This can be dealt with; if the row length mod 4 is 2, then rows can be grouped by even and odd; if the row length mod 4 is 1 or 3, then four versions of the inner loop can be used (note that there is no constraint that registers `r16`, `r17` and `r18` address elements in adjacent rows). However, these solutions to the problem are rather complicated.

4 Some alternatives

For matrices with dimensions $l \times m$ and $m \times n$, classical matrix multiplication[3] requires $O(lmn)$ multiplications and additions. Transposing the second matrix requires only $O(mn)$ memory operations. Clever coding would use the cache as an extremely long vector register by transposing each column into a row in the cache. Asymptotically, this will win.

5 Other important algorithms

There are other important numerical algorithms that may or may not be able to make use of the peak floating-point performance of the 80860. Complex arithmetic may also change things.

Cholesky decomposition ($A = LL^T$) should run well. The innermost loop is of the form

$$\begin{array}{l} \textbf{for } k \leftarrow 1 \textbf{ to } i-1 \textbf{ do} \\ \quad sum \leftarrow sum - L[i,k] \times L[j,k] \end{array}$$

That is, the innermost loop takes the inner product of two rows in a fashion similar to the first example in this paper.

Gaussian elimination will probably need to run at less than full speed unless the matrices being manipulated can fit within the cache. This problem occurs because the pipelined load and store instructions manipulate only one or two operands (caching loads and stores can handle 4 operands at once), and each iteration of the inner loop requires two inputs and one output (that is, the loop implements "row j gets row j minus α times row i).

[3] Strassen matrix multiplication is somewhat faster asymptotically, but tends to be numerically unstable.

Use of pipelined operations "where appropriate" would lead to $N + N/4 + 1$ core instructions needed to feed N floating point instructions. However, this is still 72% of peak, and clever perturbations to the algorithm might allow further gains.

The QR factorization described in Dennis and Schnabel (after Stewart) contains one inner loop calculating a row inner product, and one inner loop subtracting a multiple of one row from another. The inner product loop will run at full speed, and the row operation loop can run at 72% of full speed.

Note, in all cases, that use of double precision will make obtaining peak speed more difficult. Half as many registers will be available, and twice as many load and store instructions will be needed to feed the floating point unit. Elementary row operations (with the updated row obtained via the pipeline) will obtain at best 40% of the chip's peak speed (that is, $2.5N$ core instructions will be needed to load and store the operands for N double precision instructions). If all operands are cached, then the derating will be only 67% (ignoring cache overflow); if the updated row only is cached, then the derating will be 50%. Clever changes to algorithms using row algorithms may be called for.

6 Conclusions

Generating optimal floating point code for the Intel 80860 from user code will be extraordinarily difficult. Library routines are definitely called for. Upper bounds on the speed of certain "elementary" matrix operations also limits floating point performance. These problems are even more severe for double-precision operations. This bottleneck would be eased if there were more flexible instructions for loading and storing floating point numbers; pipelined quad-word loads would be especially helpful for implementing row operations.