UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

EEC171 Parallel Computer Architectures Spring 2009

Homework 3 Due Wednesday 3 June 12:00 pm [turn in to homework box]

(These questions are mostly from final exams given in previous years.)

1. Appendix F provides an excellent overview of vector processing. Make sure you read Appendix F. Recall the definitions of three vector figures of merit from Appendix F:

   - R_∞—The MFLOPS rate on an infinite-length vector. Although this measure may be of interest when estimating peak performance, real problems do not have unlimited vector lengths, and the overhead penalties encountered in real problems will be larger.
   - $N_{1/2}$—The vector length needed to reach one-half of R_∞. This is a good measure of the impact of overhead.
   - N_v—The vector length needed to make vector mode faster than scalar mode. This measures both overhead and the speed of scalars relative to vectors.

   Consider the following measured runtimes for a particular task (with data copied directly from your instructor's dissertation), as a function of strip length, run on a 500 MHz 8-lane vector processor. Recall from class that we process long vectors by dividing them into strips of a given length ("stripmining"), then run each strip sequentially.

Vector length	Number of cycles to finish task (in millions of 2 ns cycles)
8	31.773365
16	16.834815
24	11.723295
32	10.701045
40	8.903265
48	7.752865
64	6.495445
80	5.148655
120	4.550335
160	3.924675
224	3.554295
256	3.434745
288	3.364915

   Assume that the task requires 25 million floating point instructions. In this entire problem you can model the execution of each strip as an (unknown but constant) number of cycles of dead (warmup) time, followed by a certain number of cycles of fully utilized, 8-wide vector operations. (In other words, the warmup time is constant per strip, no matter how many elements are in the strip, and the time actually processing a strip is proportional to the number of elements per strip.)

(a) (5 points) Within 25%, what is R_∞, expressed in MFLOPS? (What performance could we expect if we could process vectors of arbitrary length?)

(b) (5 points) Within 25%, what is $N_{1/2}$, expressed in number of vector elements?

(c) (15 points) Let's say we model the vector processor as we did at the beginning of this problem, running each strip in succession and paying the warmup time per strip. For this part, assume there are 1M elements in the vector to process. Within 25%, what is the warmup time (in cycles) per strip? (You'll have to do some math and use a couple of data points from the table to solve this problem; it doesn't really matter which ones you pick, but I used 16 and 256.)

2. One core kernel that we've covered in this class is dense matrix multiplication. In this problem we consider a 4 × 4 matrix multiplied by a 4-element vector.

$$\begin{bmatrix} o_0 \\ o_1 \\ o_2 \\ o_3 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} i_0 \\ i_1 \\ i_2 \\ i_3 \end{bmatrix}$$

Also recall that Intel's SSE instructions are 4-wide; each SSE instruction operates on 4 data elements in (SIMD) parallel. You can use any reasonable 4-wide instruction (you will find `add4` and `mul4` particularly useful).

(a) (4 points) Write assembly code for a scalar (single-issue) machine to do a matrix multiplication. Assume the vector input is in registers `i0-i3`, the matrix is in `m00-m03` (nomenclature is `mrc`, `r` is row, `c` is column), and the vector output is in `o0-o3`. Feel free to abbreviate as necessary (use as little ink as possible).

(b) (8 points) Write the minimal sequence of SSE instructions to solve this problem. You can lay out the 4-wide registers any way you want, with the exception that the input 4-vector and the output 4-vector both must be packed in a single 4-wide register (register `i` and `o`). If you need to rearrange data within a register, that costs an instruction. Show how your registers are packed as appropriate.

(c) (5 points) Assuming that your processor can issue one scalar instruction per cycle or one SSE (SIMD) instruction per cycle, what is the speedup to move from optimal scalar code to your SSE code?

(d) (5 points) Assuming that your processor can issue four scalar instructions per cycle (with perfect out-of-order execution and infinite register renaming) or one SSE (SIMD) instruction per cycle, what is the speedup to move from optimal scalar code to your SSE code? Note that no two scalar instructions can be issued in the same cycle if they have a dependency between them.

3. (8 points) In a CUDA bitonic sort implementation, on each iteration, each element is paired with another element. Together the two threads associated with those two data elements compare their two data elements and either copy those elements to the output or swap them to the output (a "compare and swap" operation).

Ben Bitdiddle only writes code in SIMD-style with no conditionals at all. His thread code to evaluate `compareAndSwap()` at each element looks like this:

```
int * data_in;
int * data_out;
void compareAndSwap() {
  int myAddr, otherAddr;     /* you can assume these are predefined */
  int myData = data_in[myAddr];
  int otherData = data_in[otherAddr];
  int keepThisData = figureOutWhichDataToKeep(myData, otherData);
  data_out[myAddr] = keepThisData;
}
```

Write the same routine assuming that you have MIMD hardware per thread (but still run the same program on all threads in parallel). Your goal is to write a program that minimizes memory bandwidth compared to the SIMD case. You can assume that if you do not run a particular segment of code, it will not incur any memory traffic. You can introduce new functions (like `figureOutWhichDataToKeep`) without defining them if it's clear what they will do.

4. (10 points) (This is F.9 from your book, rewritten in C.)

   Here is a tricky piece of code with two-dimensional arrays. Does this loop have dependencies? Can these loops be written so they are parallel? If so, how? Rewrite the source code so that it is clear that the loop can be vectorized, if possible.

```
for (int j = 0; j < n; j++) {
  for (int i = 0; i < j; i++) {
    aa[i][j] = aa[i-1][j] * aa[i-1][j] + bb[i][j];
  }
}
```

5. For this question you may not consult any external sources for the answers to these questions, but you may use external sources if necessary to understand how the algorithms work. I would encourage you to try to figure them out yourself though!

   In this question we will look at a particular parallel primitive called "scan" (also known as "parallel prefix"). We are looking at a "sum-scan" (a scan with addition as the operator). Sum-scan has two variants. In an "exclusive sum-scan", the output at each data element is the sum of all the input elements that came before that element. In an "exclusive sum-scan", the output at each data element is the sum of all the input elements up to and including that element. For instance, the exclusive sum-scan of the vector [1 2 3 4] is [0 1 3 6], and the inclusive sum-scan of the vector [1 2 3 4] is [1 3 6 10].

   We will look at three different formulations of scan. In the algorithmic descriptions below, x is the input array, n is the number of items to scan, and d is the step number.

   - First, the serial algorithm (Algorithm 1) is simple.
   - The first parallel formulation is from Hillis and Steele (Algorithm 2). This is an inclusive sum-scan.

- The second parallel formulation is from Blelloch (Algorithm 3). This is an exclusive sum-scan. It has two phases, first an up-sweep phase, then a down-sweep phase.

(a) It's important that you be able to interpret algorithms such as those above. Show the step-by-step result of both a Hillis and Steele scan and a Blelloch scan on the input array [1 2 3 4 5 6 7 8].

(b) A *step* comprises all operations that can be completed simultaneously. Loosely speaking, the number of steps in a parallel algorithm is the critical path for that algorithm. For an input array of size n (where n is a power of two), as a function of n, how many steps are necessary in a Hillis and Steele scan, how many steps are necessary in a Blelloch scan, and how many steps are necessary in a serial implementation of scan? Also express these in big-O notation ($O(n^2)$, etc.). This is called the *step complexity* of the algorithm.

(c) For an input array of size n (where n is a power of two), as a function of n, how many additions are necessary in a Hillis and Steele scan, how many additions are necessary in a Blelloch scan, and how many additions are necessary in a serial implementation of scan? Also express these in big-O notation ($O(n^2)$, etc.). This is called the *work complexity* of the algorithm.

Algorithm 1 Serial inclusive scan

```
for $k = 1$ to $n - 1$ do
    $x[k] = x[k] + x[k-1]$
end for
```

Algorithm 2 Hillis and Steele parallel scan

```
for $d = 1$ to $\log_2 n$ do
    for all $k$ in parallel do
        if $k \geq 2^d$ then
            $x[k] = x[k - 2^{d-1}] + x[k]$
        end if
    end for
end for
```

Algorithm 3 Blelloch parallel scan

Blelloch scan, up-sweep (reduce) phase:

```
for $d = 0$ to $\log_2 n - 1$ do
    for all $k = 0$ to $n - 1$ by $2^{d+1}$ in parallel do
        $x[k + 2^{d+1} - 1] = x[k + 2^d - 1] + x[k + 2^{d+1} - 1]$
    end for
end for
```

Blelloch scan, down-sweep phase:

```
$x[n - 1] = 0$
for $d = \log_2 n - 1$ down to $0$ do
    for all $k = 0$ to $n - 1$ by $2^{d+1}$ in parallel do
        $t = x[k + 2^d - 1]$
        $x[k + 2^d - 1] = x[k + 2^{d+1} - 1]$
        $x[k + 2^{d+1} - 1] = t + x[k + 2^{d+1} - 1]$
    end for
end for
```