Parallel Computer Architecture and Programming
Written Assignment 1 SOLUTIONS

30 points total + 2 pts extra credit. Due Monday, July 3 at the start of class.

Warm Up Problems

Problem 1. (10 points):

A. (5 pts) Complete the ISPC code below to write an if-than-else statement that causes an 8-wide SIMD processor to run at nearly 1/8th its peak rate. (Assume the ISPC gang size is 8. Pseudocode for an answer is fine.)

```
void my_ispc_func() {

   int i = programIndex;

   if (i == 0) {

        very long sequence of instructions here!

   } else {

        very short sequence of instructions here!

   }
}
```

The code above will suffer from execution divergence. When running the 'if' clause, which occupies the bulk of execution since it involves a very long sequence of instructions, only one of the eight program instances will be doing useful work, so the system runs at 1/8 peak performance.

B. (5 pts) You arrive at your new internship at MSRA and on the first day you are asked to improve the performance of the following single-threaded program on a computer with one core running at 2 Ghz that can perform one math operation per clock. **The system has a memory bandwidth of 8 GB/sec, and 0 memory latency.**

```
const int N = 10000000; // very large

void slow_code(float *x, float* y) {

  float tmp[N];

  for (int i=0; i<N; i++)
     tmp[i] = 2.0 * x[i];  // 1 math instruction

  for (int i=0; i<N; i++);
     y[i] = tmp[i] + 2.0;  // 1 math instruction
}
```

Your boss asks you to improve the performance of the code by 2×. Your friend looks at the code, and says, "This is an easily parallelizable program. Let's parallelize the two loops and buy a better CPU that has two cores." You look at the code and say "I think there is a way to get a 2× speedup on our current CPU just by changing the code." Please rewrite the program to get a 2× speedup on the current CPU. Explain why this change yields a speedup.

Solution: The computation is bandwidth bound. Each math operations requires 4 bytes of input and 4 bytes of output. Since the core can perform 1 instruction per clock at 2 GHz, it performs two billion operations per second. Therefore, the memory system must be able to deliver 16 GB/sec of bandwidth to the processor. Since it can only provide 8 GB/sec, the computation is bandwidth bound, and the processor runs at 50% efficiency. ***For this reason, adding the ability to perform math operations at higher throughput by adding more cores would not help performance.*** *Rewriting the code as shown below will improve performance by eliminating the read and write of data from* `tmp`*. The new computation performs 2 math operations for every 8 bytes read, and now requires only 8 GB/sec of bandwidth, so it runs at 100% efficient, 2× faster!*

```
const int N = 10000000; // very large

void fast_code(float *x, float* y) {
   for (int i=0; i<N; i++)
      y[i] = 2.0 * x[i] + 2.0;  // 2 math instruction
}
```

Problem 2. (10 points):

Consider the ISPC code at the bottom of this page, which I will briefly explain below:

The function `brighten_image()` modifies an image of size 32×`HEIGHT` pixels. (The image is a grayscale image represented as an array of floats.) How the image is modified in `brighten_chunk` depends on the contents of a black and white "mask" image of the same size. The code multiplies the value of each input pixel by 1000 if the corresponding pixel of the mask image is white (the mask has value 1.0) and by a factor of 10 otherwise.

```
void brighten_image(uniform int height, uniform float image[], uniform float mask_image[])
{
    uniform int NUM_TASKS = 64;
    uniform int rows_per_task = height / NUM_TASKS;
    launch[NUM_TASKS] brighten_chunk(rows_per_task, image, mask_image);
}


void brighten_chunk(uniform int rows_per_task, uniform float image[], uniform float mask_image[])
{
     //   'programCount' is the ISPC gang size.
     //   'programIndex' is a per-instance identifier between 0 and programCount-1.
     //   'taskIndex' is a per-task identifier between 0 and NUM_TASKS-1

     // compute starting image row for this task
     uniform int start_row = rows_per_task * taskIndex;

     // process all pixels in a chunk of rows
     for (uniform int j=start_row; j<start_row+rows_per_task; j++) {
       for (uniform int i=0; i<32; i+=programCount) {

         int idx = j*32 + i + programIndex;
         int iters = (mask_image[idx] == 1.f) ? 1000 : 10;

         float tmp = 0.f;
         for (int j=0; j<iters; j++)
             tmp += image[idx];

         image[idx] = tmp;
       }
     }
}
```

Notice that the code in `brighten_image` partitions the image processing work into 64 ISPC tasks. **We did not talk about ISPC tasks in class but you will learn about them in part 3 of Programming Assignment 1, or by reading the ISPC documentation.** To make the problem simpler, please assume that the ISPC tasks are perfectly distributed onto the CPU's cores (I do not want you to think about imbalanced work across CPU cores in this problem).

(The question is on the next page)

You want to buy a new CPU that runs this computation as fast as possible. At the store you see the following three CPUs on sale for the same price:

(A) 4 GHz *single core* CPU capable of performing one floating point addition per clock (no parallelism)

(B) 1 GHz *single core* CPU capable of performing one 32-wide SIMD floating point addition per clock

(C) 1 GHz *dual core* CPU capable of performing one 4-wide SIMD floating point addition per clock

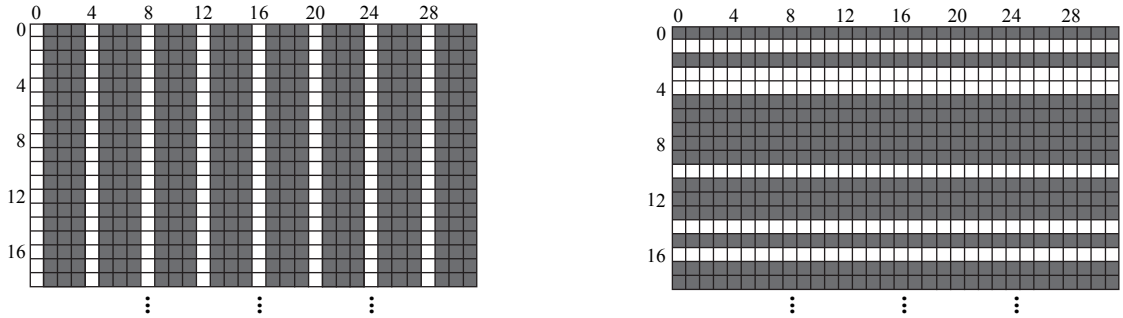

Figure 1: Image masks used to govern image manipulation by `brighten_image`

A. Assuming the code will execute using mask image 1 above, rank all three machines in order of their performance running the ISPC code. Please explain your answer by computing the program's execution time on the various processors. When computing execution time, you may assume that (1) the only operations you need to account for are floating-point additions in the innermost loop. (2) The ISPC gang size will be set to the SIMD width of the CPU. (3) There are no stalls during execution due to data access.

(Hint: it may be easiest to consider the execution time of each row of the image.)

Answer: B > A > C

For image 1, each row of the mask is mixture of white and black pixels: every 1 white pixel is followed by 3 black pixels. Since the SIMD width for CPUs B and C are 32 and 4 respectively these processors will suffer from ***branch divergence*** *when executing this ISPC code. ISPC program instances working on a black pixel will wait for gang instances assigned white pixels to finish their execution of 1000 loop iterations.*

Now let's calculate how many cycles it takes for each processor to finish rendering a single row:

- A: $10 \times 24 + 1000 \times 8 = 8240$ cycles
- B: 1000 cycles
- C: $8 \times 1000 = 8000$ cycles

However, processor A is clocked at 4 GHz and processor C has 2 cores (so its throughput is doubled). Thus the effective per-row cost for each platform, normalized to 1000 cycles of a 1 GHz single core processor, will be:

- A: $8240 \div 1000 \div 4 = 2.06$
- B: $1000 \div 1000 = 1$
- C: $8000 \div 1000 \div 2 = 4$

So B performs better than A and A (despite executing entirely in serial) performs better than C.

B. Rank all three machines in order of performance for mask image 2? Please justify your answer, but you are not required to perform detailed calculations like in part A.

Answer: B > C > A

In image 2, unlike image 1, all the rows are homogeneous. As a result, means processor B and C no longer suffer from ***branch divergence****. Since all processor execute at their peak rates, the processor with the most raw processing power will provide the best processing speed. B provides the most raw processing power, followed by C.*

Mixing Superscalar, Threads, and Cores

Problem 3. (10 points):

Consider the following sequence of 10 instructions. (2 memory operations, 8 arithmetic ops)

```
1.  LD    R0 <- [R3]      // load from address given by R3
2.  ADD   R0 <- R0, R0
3.  MUL   R0 <- R0, R0
4.  MUL   R0 <- R0, R0
5.  ADD   R1 <- R0, R0
6.  MUL   R2 <- R0, R0
7.  ADD   R0 <- R0, R1
8.  ADD   R1 <- R1, R2
9.  ADD   R0 <- R0, R1
10. ST   [R3] <- R0       // store to address given by R3
```

A. (2 pts) Consider running this instruction sequence on a **single-core, but superscalar processor that can execute up to two independent instructions per clock.** Assume that all ten instructions (including loads and stores) complete in a single single cycle (latency = 1 clock). What is the **speedup** observed by running this instruction stream on the 2-way superscalar processor compared to a single-core processor that can only issue one instruction per clock (no superscalar)? **(Strong Hint: drawing the graph of instruction dependencies might be helpful.)**

Solution: The superscalar processor can complete the instruction sequence in 8 cycles. (Dependencies in the program allow both execution units to be used in only two clocks.) Therefore the speedup is $10/8 = 1.25\times$.

B. (2 pts) What is the speedup of a superscalar processor that is able to issue **three instructions per clock compared to a non-superscalar processor that completes one instruction per clock?**

Solution: Still 1.25×. The instruction sequence never presents a situation where ILP=3, so the third hardware unit can never be used.

C. (2 pts) Now consider a loop where the 10-instruction sequence from part A is the body of a loop. Assume the loop is parallelized using two threads as shown below. Thread 0 computes the first N/2 iterations, and thread 1 computes the second N/2 iterations.

```
void runme(float* x, int start, int end) {
  for (int i=start; i<end; i++) {
    // let the value of R3 be the address of x[i]
    // the 10-instruction sequence from part A goes here.
  }
}

float x[1000];

// initialize values of x here...

std::thread t0(runme, x, 0, 500);
std::thread t1(runme, x, 500, 1000);
```

The two-threaded program is run on a **single-core processor with support for two hardware threads (execution contexts)**. Assume the processor works as follows: each clock it ALWAYS switches which thread it can draw instructions from. It runs up to two independent instructions from ONLY THAT ONE THREAD (if available), then switches to the next thread in the next clock. What is the speedup of this processor compared to the **2-way superscalar, single-threaded processor from part A?** Why?

Solution: The performance is the same. The processor is performing strict interleaved multi-threading and there is no latency to hide. Since instructions from only one thread can be issued in a cycle, the problem of lack of ILP in a single thread still exists.

D. (2 pts) Now consider a small modification to the single core, multi-threaded processor from the previous problem: **each clock the single-core processor can choose to execute any mixture of up to two independent instructions from (if needed) both hardware threads** (not just one thread like in part C). Given this new design, when running the code from part C, what is the expected speedup compared to **a single-core non-superscalar processor that completes one instruction per clock?**

Solution: The new processor runs at peak rate at all times since it can always draw at least one instruction from each thread. It will run $2\times$ faster than the baseline processor and $8/5 = 1.4\times$ faster than the superscalar processor.

E. (2 pts) Now consider a situation where the load operation has a latency of 20 cycles. (There is no latency to the store, it is still one cycle.) Consider running the two-threaded program on the single-core, **2-threaded processor that is only capable of issuing at most one instruction from one thread per clock**. What percentage of peak execution capability does the code achieve?

Solution: About 66.67% of peak. There are two threads that we can run at any one time. In the steady state, while the first thread is stalling for memory, we can issue 10 ALU instructions, then issue a store. For the next 10 cycles, both threads are stalled for memory. After this, we can do 10 cycles of ALU instructions from the other thread. In total, this is 20 cycles of work for 30 cycles total.

F. (2 pts EXTRA CREDIT) **Tricky! (And depends on a correct understanding of part D.)** Now consider the case where the processor is the same as in Part D (**two-way superscalar, can execute up to two instructions from potentially different threads**), but you are allowed to choose the number of hardware execution contexts the core has. How many threads are needed to get very near peak performance? (Hint: Recall that peak is two instructions per clock.)

Solution: Two threads must be runnable to fill the core's execution units. When two threads are runnable, those two threads take 10 cycles between memory accesses, so so four additional threads (two sets of two) are needed to hide the 20 cycle memory latency AND maintain two runnable threads at a time to fill the execution units. So a total of six threads are necessary.