Full Name:
Andrew Id:

15-418/618 Spring 2020
Exercise 1

	Registered students	Waitlist students
Assigned:	Fri., Jan. 17	Fri., Jan. 17
Due:	Fri., Jan. 24, 11:00 pm	Fri., Jan.. 31, 11:00 pm

Overview

This exercise is designed to help you better understand the lecture material and be prepared for the style of questions you will get on the exams. The questions are designed to have simple answers. Any explanation you provide can be brief—at most 3 sentences. You should work on this on your own, since that's how things will be when you take an exam.

You will submit an electronic version of this assignment to Gradescope as a PDF file. For those of you familiar with the LaTeX text formatter, you can download the template and configuration files at:

http://www.cs.cmu.edu/~418/exercises/config-ex1.tex
http://www.cs.cmu.edu/~418/exercises/ex1.tex

Instructions for how to use this template are included as comments in the file. Otherwise, you can use this PDF document as your starting point. You can either: 1) electronically modify the PDF, or 2) print it out, write your answers by hand, and scan it. In any case, we expect your solution to follow the formatting of this document.

Problems

Consider the following code where each line within the function represents a single instruction.

```
typedef struct {
    float x;
    float y;
} point;

inline void innerProduct(point *a, point *b, float *result)
{
    float x1 = a->x; // Uses a load instruction
    float x2 = b->x;
    float product1 = x1*x2;
    float y1 = a->y;
    float y2 = b->y;
    float product2 = y1*y2;
    float inner = product1 + product2;
    *result = inner; // Uses a store instruction
}

void computeInnerProduct(point A[], point B[], float result[], int N)
{
    for (int i = 0; i < N; i++)
        innerProduct(&A[i], &B[i], &result[i]);
}
```

In the following questions, you can assume the following:

- N is very large ($> 10^6$).
- The machines described have modern CPUs, providing out-of-order execution, speculative execution, branch prediction, etc.
  - There are ample resources for fetching, decoding, and committing instructions. The only performance limitations are due to the number, capabilities, and latencies of the execution units.
  - The branch prediction is perfect.
- There are no cache misses.
- The overhead of updating the loop index `i` is negligible.
- The load/store units perform any necessary address arithmetic.
- The overhead due to procedure calls, as well as starting and ending loops, is negligible.

Problem 1: Instruction-Level Parallelism

Suppose you have a machine M_1 with two load/store units that can each load or store a single value on each clock cycle, and one arithmetic unit that can perform one arithmetic operation (e.g., multiplication or addition) on each clock cycle.

A. Assume that the load/store and arithmetic units have latencies of one cycle. How many clock cycles would be required to execute `computeInnerProduct` as a function of N? Explain what limits the performance.

3N. The flow of instructions executed by each unit is shown in the table below.
The limitation is the number of arithmetic units.

B. Now assume that the load/store and arithmetic unit have latencies of 10 clock cycles, but they are fully pipelined, able to initiate new operations every clock cycle. How many clock cycles would be required to execute `computeInnerProduct` as a function of N? Explain how this relates to your answer to part A.

3N. No loop dependencies here, so we can still launches all instructions in order, and use multiple registers to store the temprary results. Thus the times isn't changed.
In part A, due to the short latencies, it only needs one registers each for x1, x2, y1, y2, product1, product2, and inner. But part B needs more.

Arith Unit	L/S Unit1	L/S Unit2
	x1 = a[1].x	x2 = b[1].x
MUL x1 x2	y1 = a[1].y	y2 = b[1].y
MUL y1 y2	x1 = a[2].x	x2 = b[2].y
ADD	y1 = a[2].y	y2 = b[2].y
MUL x1 x2	Store	
MUL y1 y2	x1 = a[3].x	x2 = b[3].x
ADD	y1 = a[3].y	y2 = b[3].y
MUL x1 x2	Store	
MUL y1 y2		
ADD		
	Store	

Problem 2: SIMD with ISPC

Consider running the following ISPC code.

```
export void computeInnerProductISPC(uniform point[] A,
                                    uniform point[] B,
                                    uniform float[] result,
                                    uniform int N)
{
    foreach(i = 0 ... N)
    {
        result[i] = A[i].x * B[i].x + A[i].y * B[i].y;
    }
}
```

Suppose machine M_2 has one 8-wide SIMD load/store unit, and one 8-wide SIMD arithmetic unit. Both have latencies of one clock cycle.

A. How many clock cycles would be required to execute `computeInnerProductISPC` as a function of N? Explain what limits the performance.

$\frac{5}{8}N$. Just like Problem 1A, the limitation is the number of arithmetic units.

SIMD load/store	SIMD Arith
Load .x	
Load .y	MUL
	MUL
	ADD
Store	

B. If we were to run the code shown in `computeInnerProductISPC` on a five-core machine M_3, where each core has the same SIMD capabilities as M_2, what would be the best speedup it could achieve over the single-core performance of part A? Explain.

Only $1\times$. It would only have one thread, thus only can use a signle core.

Problem 3: SIMD, Multicore, and Multi-Threaded Parallelism with ISPC

A. Consider the five-core machine M_3 described in Problem 2B. Suppose you could write multi-threaded code where there is no overhead associated with the threads. Each thread would run the function `computeInnerProductISPC` to compute some subset of the N elements. What is the maximum speedup you could achieve relative to the single-threaded code running on machine M_2?

$5\times$. To achieve the maximum speedup, it could create 5 threads to take advantage of all 5 cores.

B. Now suppose we have a machine M_4, identical to M_3, except that each core supports three-way simultaneous multithreading. What is the maximum speedup your multithreaded code could achieve relative to what it achieved running on machine M_3.

$1\times$. Multithreading can be used to hide the latencies, but there is no latency here.

C. Let $N = 10^6$. Running on machine M_3, if we were to write a Pthreads program that spawns 250 threads, each computing 4000 inner products using `computeInnerProductISPC`, would this program get an overall performance improvement over one that uses a single thread to compute all N inner products? (Use your own intution about the cost of spawning new threads in Pthreads when answering this question.) Explain.

Yes.
If there is only one single thread, only one core is used.
However, with 250 threads, all 5 cores are used. So the time spent on executing instructions is 5 times less.
Even consider the cost of switching threads, the gap is still huge.

No.
Each thread would use only around $4000 \times \frac{5}{8} = 2500$ cycles for the computation. Yet the thread overhead is way more than 7500 cycles (usually takes several μs)

D. Let $N = 10^6$. Running on machine M_3, if we were to write an ISPC program that launches 250 tasks, each computing 4000 inner products using `computeInnerProductISPC`, would this program get an overall performance improvement over one that uses a single task to compute all N inner products? (Consider what you know about the performance characteristics of ISPC tasks.) Explain.

Yes.

ISPC task is more efficient than threads. That each task uses 2500 cycles is still worthwhile for tasking. However, the overhead still exists.