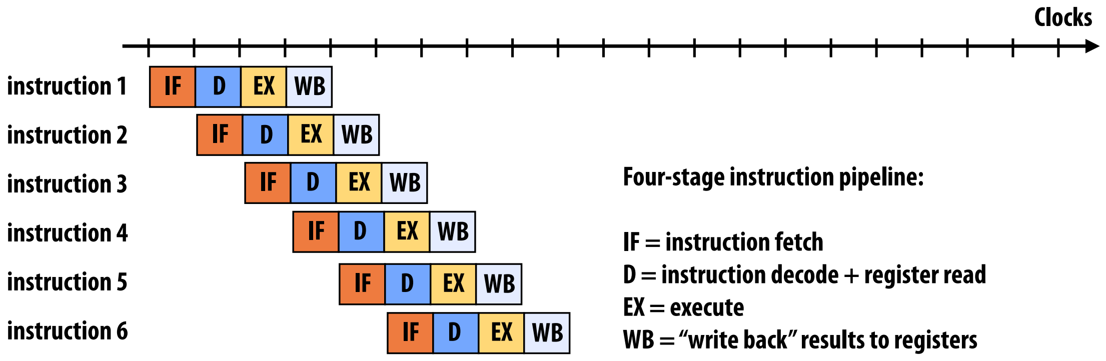
Stanford CS149: Parallel Computing
Written Assignment 2

Problem 1: A Cardinal Processor Pipeline (40 pts)

The fast-growing startup Cardinal Processors, Inc. builds a single core, single threaded processor that executes instructions using a simple four-stage pipeline. As shown in the figure below, each unit performs its work for an instruction **in one clock**. To keep things simple, assume this is the case for all instructions in the program, including loads and stores (memory is infinitely fast).

The figure shows the execution of a program with six **independent instructions** on this processor. *However, if instruction B depends on the results of instruction A, instruction B will not begin the IF phase of execution until the clock after WB completes for A.*



A. (3 pts) Assuming all instructions in a program are **independent** (yes, a bit unrealistic) what is the instruction throughput of the processor?

B. (4 pts) Assuming all instructions in a program are **dependent** on the previous instruction, what is the instruction throughput of the processor?

C. (4 pts) What is the latency of completing an instruction?

D. (4 pts) Imagine the IF stage is modified to improve its throughput to fetch instructions per clock, but no other part of the processor is changed. What is the new overall maximum instruction throughput of the processor?

E. (10 pts) Consider the following C program:

```
float A[500000];
float B[500000];
// assume A is initialized here

for (int i=0; i<500000; i++) {
    float x1 = A[i];
    float x2 = 6 * x1;
    float x3 = 4 + x2;
    B[i] = x3;
}
```

Assuming that we consider only the four instructions in the loop body (for simplicity, disregard instructions for managing the loop or calculating load/store addresses), what is the average instruction throughput of this program? (Hint: You should probably consider instruction dependencies, and at least two loop iterations worth of work).

F. (10 pts) Modify the program to achieve peak instruction throughput on the processor. Please give your answer in C-pseudocode.

G. (5 pts) Now assume the program is reverted to the original code from part E, but the for loop is parallelized using OpenMP. (Recall from written assignent 1 is that openMP is a set of C++ compiler extensions that enable thread-parallel execution. Iterations of the four loop can be carried out in parallel by a pool of worker threads.)

```
// assume iterations of this FOR LOOP are parallelized across multiple
// worker threads in a thread pool.
#pragma omp parallel for
for (int i=0; i<100000; i++) {
  float x1 = A[i];
  float x2 = 2*x1;
  float x3 = 3 + x2;
  B[i] = x3;
}
```

Given this program, imagine you wanted to add multi-threading to the **single-core processor** to obtain **peak instruction throughput** (100% utilization of execution resources). What is the smallest number of threads your processor could support and still achieve this goal? You may not change the program.

Problem 2: Particle Simulation (30 pts)

Consider the following code that uses a simple $O(N^2)$ algorithm to compute forces due to gravitational interactions between all N particles in a particle simulation. One important detail of this algorithm is that force computation is symmetric (`gravity(i,j) = gravity(j,i)`). Therefore, iteration `i` only needs to compute interactions with particles with index `j`, where `i<j`. As a result, there are $N^2/2$ called to `gravity` rather than N^2.

In this problem, **assume the code is run on a dual-core processor, with infinite memory bandwidth.**

```
struct Particle {
  float force;  // for simplicity, assume force is represented as a single float
  Lock l;
};

Particle particles[N];

void compute_forces(int threadId) {

  // thread 0 takes first half, thread 1 takes second half
  int start = threadId * N/2;
  int end = start + N/2;

  for (int i=start; i<end; i++) {

    // only compute forces for each pair (i,j) once, then accumulate force
    // into *both* particle i and j

    for (int j=i+1; j<N; j++) {
      float force = gravity(i, j);

      lock(particles[i].l);
      particles[i] += force;
      unlock(particles[i].l);

      lock(particles[j].l);
      particles[j] += force;
      unlock(particles[j].l);
    }
  }
}
```

Question is on the next page...

A. (15 pts) Although the code makes $N^2/2$ calls to `gravity()` it takes N^2 locks. Modify the code so that the number of lock/unlock operations is reduced by $2\times$. You may not allocate additional variables or change how look iterations are mapped to the threads.

B. (15 pts) **(This question can be answered independently from part A)** Looking at the original code, there another major performance problem that does not have to do with the number of lock/unlock operations. Please describe the problem and then describe a solution. Clearly describing an implementable solution strategy is fine, you do not need to write precise pseudocode.

Problem 3: Running a CUDA Program on the PKPU (30 pts)

The CS149 course staff decides to dip their toes into the GPU design design business and spend their summer creating a new GPU, which, given their poor marketing sense, they call a PKPU (for Prof. Kayvon Processing Unit, or Prof Kunle Processing Unit). The processor runs CUDA programs very similar to NVIDIA GPUs as discussed in class, but it has the following characteristics:

- The processor has eight cores running at 1 GHz.
- Each core provides execution contexts for up to 128 CUDA threads. (Like the GPUs discussed in class, once a CUDA thread is assigned to an execution context, the processor runs the thread to completion before assigning a new CUDA thread to the context.)
- The cores execute threads in an implicit SIMD fashion running 32 consecutively numbered CUDA threads together by executing the same instruction on 32 ALUs in a clock (the PKPU implements 32-wide "warps").
- The cores will fetch/decode one single-precision arithmetic instruction (add, multiply, etc.) per clock. Keep in mind this instruction is executed on an entire warp in that clock.
- All CUDA thread blocks on a single core cannot exceed 16 KB of shared memory storage.

A. (5 pts) When running at peak utilization. What is the processor's **maximum throughput** of single-precision **math operations**? (In your answer, please consider one multiply of two single-precision numbers as one "operation".)

Consider a CUDA kernel launch that executes the following CUDA kernel on the processor. In this program each CUDA thread computes one element of the results array `Y` using 1000 elements from the input array `X` as input. **Assume the program is compiled using a CUDA thread-block size of 128 threads, and that enough thread blocks are created so there is exactly one CUDA thread per output array element.**

```
__global__ void foo(float* X, float* Y) {

   // get array index from CUDA block/thread id
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   int input_idx = 1000 * idx;

   float result = 0.f;

   for (int i=0; i<1000; i++) {                 // 0 cycles (ignore arithmetic here)

      float val = X[input_idx+i];               // memory load (ignore arithmetic here)

      if (((int)val / 1000) % 2 == 0)           // 2 arithmetic cycles
         result += doA(val);                    // 15 arithmetic cycles
      else
         result += doB(val);                    // 15 arithmetic cycles
   }

   Y[idx] = result;                             // memory store

}
```

B. (7 pts) The CAs run the code on a 128,000-element input array initialized as `X[] = {1.f, 2.f, 3.f, 4.f, 5.f, 6.f, ...}`. *The output array `Y[]` is only 128 elements... hint: how many thread blocks is this?*. The CA's observe that the program *does not* realize peak performance on the PKPU. They are devastated! What is the primary reason for the low performance? (Please assume the input and output arrays are resident in GPU memory at the time of the kernel launch. Transfer between host and GPU memory is not relevant in this problem.)

C. (8 pts) Ben steps in and says, "hey, let me take a look", and runs the same program on an output arrays of size 128×1024 elements and $128 \times 1024 \times 1024$ elements. Does he observe *significantly different processing throughput* from the PKPU on the two workloads? Why or why not? (Please assume both the small and large workloads fit comfortably in GPU memory.)

D. (10 pts) The PKPU has a memory system that provides **64 GB/sec of bandwidth**, but all memory operations take 1 cycle, just like a math op. Fait and Matthew look at Ben's code running on the largest dataset and say "Ben, this program is not achieving peak utilization of the processor's execution units." What is the problem limiting performance? What percentage of peak PKPU throughput do they observe? *(Hint: is this a problem bandwidth bound or is it limited by SIMD divergence?) Start by computing how much data is accessed by a load instruction issued by a single warp. Then compute how frequently these loads take place given the provided program. Remember, there are eight cores overall!*

The following problems are PRACTICE PROBLEMS and will not be graded.

Practice Problem 1: Effects of Arithmetic Intensity

Your boss asks you to buy a computer for running the program below. The program uses a math library (`cs149_math`). The library functions should be self-explanatory, but an example implementation of the `cs149math_add` function is given below.

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

void cs149math_add(float* A, float* B, float* output) {
  // Recall from written asst 1 that this OpenMP directive tells the
  // C compiler that iterations of the for loop are independent, and
  // that implementations of C compilers that support
  // OpenMP will parallelize this loop across CPU cores.
  #omp parallel for
  for (int i=0; i<N; i++)
    output[i] = A[i]+B[i];
}
void cs149math_sub(float* A, float* B, float* output) { ... }
void cs149math_mul(float* A, float* B, float* output) { ... }

/////////////////////////////////////////////////////////////////

float* A, *B, *C, *tmp1, *tmp2, *result;  // assume arrays are allocated and initialized

cs149math_add(A, B, tmp1);          // 1
cs149math_mul(tmp1, C, tmp2);       // 2
cs149math_mul(tmp2, A, tmp1);       // 3
cs149math_add(A, tmp1, tmp2);       // 4
cs149math_mul(B, tmp2, tmp1);       // 5
cs149math_mul(B, tmp1, tmp2);       // 6
cs149math_mul(B, tmp2, tmp1);       // 7
cs149math_sub(C, tmp1, result);     // 8
```

You have two computers to choose from, of equal price. (Assume that both machines have the same 1MB cache and 0 memory latency.)

1. Computer A: Four cores 1 GHz, 32-wide SIMD, 128 GB/sec bandwidth
2. Computer B: Eight cores 1 GHz, 8-wide SIMD, 256 GB/sec bandwidth

ASSUME THAT YOU ARE ALLOWED TO REWRITE THE PROBLEM, INCLUDING THE LIBRARY IF DESIRED, (provided that it computes exactly the same answer—You can parallelize across cores, vectorize, reorder loops, etc. but you are not permitted to change the math operations to turn adds into multiplies, eliminate common subexpressions etc.), which machine do you choose? Why? (If you decide to change the program please give a rough description of your changes. What is parallelized, vectorized, what does the loop structure look like, etc.) Hint: begin with a computation of the current program's arithmetic intensity.

Additional space for your answer...

Practice Problem 2: Parallel Histogram

Consider the following code which computes a histogram for all values in the array `A`. Assume the code is run by T threads. Throughout this problem, assume that `N` is a very large value.

```
int A[N];  // contains values between 0 and 16

// assume all bins are initialized to 0
int bins[16];

// the following code is run by each thread. Assume thread_id
// takes on a value between 0 and T-1.  You can assume T divides N.

int elements_per_thread = N/T;
int start = thread_id * elements_per_thread;
int end = start + elements_per_thread;

for (int i=start; i<end; i++)  // assume loop management has 0 cost
   bins[A[i]]++;
```

A. There is a correctness bug in this code. Assuming you only have the single synchronization primitive `atomicAdd(int* x, int amount)` please fix the bug in the code.

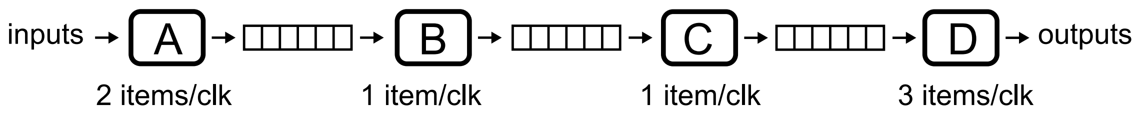
B. Assume that all loop body operations (loads, stores, adds) take 1 cycle, and ignore any costs of loop iteration. How many cycles does the body of the ORIGINAL UNMODIFIED loop take? (be careful, there is one integer add in the loop, but how many loads and stores are there?)

   If we assume that an `atomicAdd` operation (the load/add/store) takes 7 cycles, what is the speedup of your solution from part A **compared to a single-threaded version of the original code**, assuming your part B solution is run using four threads on four cores?

C. Assume that you are also given a `barrier()` as a synchronization primitive. How would you modify the code to improve the speedup. Hint: consider a common trick in this class for removing fine-grained synchronization.

Practice Problem 3: Understanding Pipelines

Consider the four-stage pipeline below. Each stage in the pipeline receives elements from a 6-element input queue, and can process elements from its input queue at the rates shown in the figure. The behavior of each stage generates one output element for each input element. **A stage stalls (does no work in a clock) if there is no room in the output queue to place a result.** You can assume that inputs arrive at stage A infinitely fast, so stage A will always process two items/clk whenever it can.



For example, consider the following behavior of the pipeline:

- t=0: Stage A processes two elements from its input queue, emits two elements to the Stage B input queue (size=2)
- t=1: Stage B processes one element from its input queue and emits one element to the Stage C input queue (size=1). Stage A processes two new elements from its input queue, emits two elements to the Stage B input queue (size=??).
- t=2: Stage C processes one element from its input queue, emits one element to its output, B processes one elements from it's input queue, A processes two elements from it's input queue, ...

What is the throughput of the pipeline in terms of completed items/clock? What is start-to-end latency of the entire pipeline processing one element? (define latency as the time between the clock where an element is first processed by A and the clock that D finishes processing it. Do not consider time spent waiting in the queue before A) **(Be careful: In both cases, make sure you give answers for the steady-state behavior of the pipeline (including its queues), not its initial startup.)**