Stanford CS149: Parallel Computing
Written Assignment 1

Problem 1: Superscalar and Hardware Multi-Threading

Consider the following sequence of instructions. There is a load operation, followed by 9 math operations, followed by a store.

```
1.  LD   R1   <- [R0]
2.  MUL  R2   <- R1, R1
3.  ADD  R3   <- R1, R1
4.  MUL  R4   <- R2, R3
5.  ADD  R5   <- R2, R3
6.  MUL  R2   <- R4, R5
7.  ADD  R3   <- R4, R5
8.  MUL  R4   <- R2, R3
9.  ADD  R5   <- R2, R3
10. ADD  R1   <- R4, R5
11. ST   [R0] <- R1
```

A. Please draw the dependency graph for the instruction sequence.

B. The sequence of instructions on the previous page is the loop body of a program. For example:

```
float src[VERY_BIG];  // let VERY_BIG = 100,000,000
float dst[VERY_BIG];
#pragma omp parallel for          // parallelize this loop using threads
for (int i=0; i<VERY_BIG; i++) {
   dst[i] = foo(src[i]); // assume foo() is the 9 arithmetic instrs above
}
```

We haven't discussed the line "`#pragma omp parallel for`" in class yet, but it's a line of code using OpenMP syntax. OpenMP is a set of extensions to C/C++ that enable parallel execution. You should assume that the directive tells the compiler to parallelize the for loop by dividing the iterations of the loop up among threads. You can also assume that the compiler will create exactly as many threads as there are hardware execution contents on the target CPU.

The loop is run on a single core, single-threaded processor that can perform one instruction per clock. The processor has no cache. Memory stores, like arithmetic instructions, have a latency of 1 cycle. (If an arithmetic instruction executes in clock c, then the next instruction that depends on the result can run in clock $c+1$). However, the latency of a load instruction is 60 cycles. That is, if the processor executes a load instruction in cycle c, then the next instruction that depends on the load will run in cycle $c+60$. In other words, in the single threaded case once a load completes, the processor runs 11 instructions, then must wait for 60 cycles, then runs 11 instructions, etc. **What is the steady-state utilization of the processor? (the fraction of cycles the processor is running an instruction (both arithmetic or memory)?**

C. Now imagine the processor from part A is changed to be a **multi-threaded processor that performs interleaved multi-threading**. How many total hardware threads are needed to obtain full instruction throughput? (i.e., 100% utilization)

D. Now imagine the processor from part A changed to be a **single-threaded, but four-way superscalar processor that is capable of four instructions per clock. The memory latency of a load remains 60 cycles.** Please compute the speedup of running the code (again, remember it's repeated as part of a loop body) compared to the processor from part A (single-threaded, not superscalar). Your speedup calculations should include the costs of memory access. The math is not clean on this problem: show your work, and you can leave your answer as a fraction.

.

E. What is the minimum number of hardware threads needed to achieve maximum instruction throughput when considering the four-way superscalar processor from the previous sub-problem? Please assume that the processor performs strict interleaved multi-threading: **each clock, the core selects one thread to execute, and attempts to run up to four instructions from only that one thread.** Again, please show your work.

F. Your friend looks at the results the number of threads you computed in this assignment and says "Wow, you are working hard to hide that memory latency using a bunch of hardware threads. Why don't you just eliminate the latency by **adding a 16 MB data cache with 4-byte cache lines** to the processor? Then you don't need to have an application that creates many threads!" Do you think your friend's suggestion is a good one? Why or why not? (Hint: please look at the data access patterns given in the code in part A.)

Problem 2: Picking the Right CPU for the Job

You write a bit of ISPC code that modifies a grayscale image of size 32×`height` pixels based on the contents of a black and white "mask" image of the same size. The code brightens input image pixels by a factor of 1000 if the corresponding pixel of the mask image is white (the mask has value 1.0) and by a factor of 10 otherwise.

The code partitions the image processing work into 128 ISPC tasks, which you can assume balance perfectly onto all available CPU processors.

```
void brighten_image(uniform int height, uniform float image[], uniform float mask_image[])
{
    uniform int NUM_TASKS = 128;
    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 k=0; k<iters; k++)
             tmp += image[idx];         // these are the ops we want you to count

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

(question continued on next page)

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

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

(B) 1 GHz *12-core* CPU capable of performing one 2-wide SIMD floating point addition per clock

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

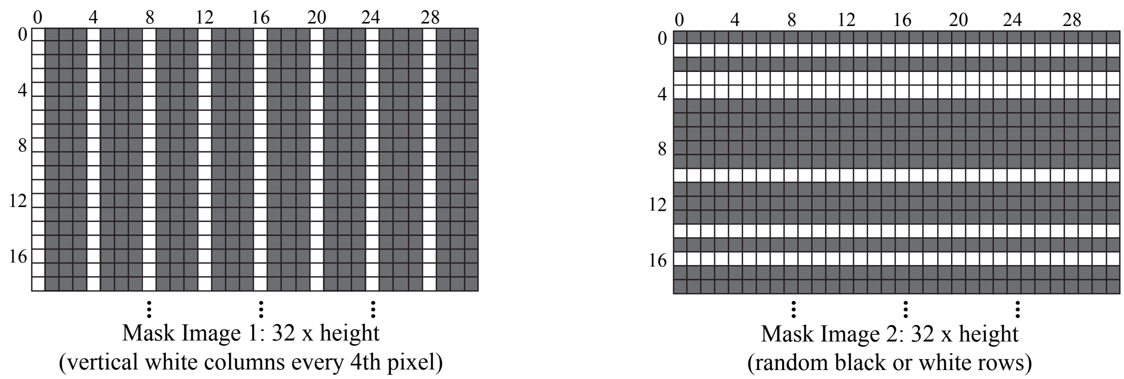


Figure 1: Image masks used to govern image manipulation by `brighten_image`

A. If your only use of the CPU will be to run the above code as fast as possible, and assuming the code will execute using mask image 1 above, rank all three machines in order of performance. Please explain how you determined your ranking by comparing execution times on the various processors. When considering execution time, you may assume that (1) the only operations you need to account for are the floating-point additions in the innermost 'k' 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.)

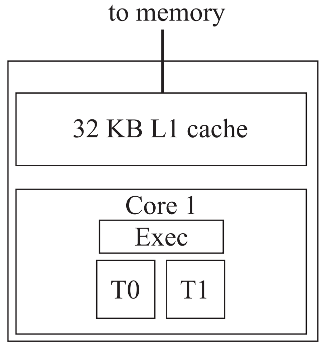
B. Rank all three machines in order of performance for mask image 2? Please justify your answer, but you are not required to perform detailed calculations like in part A.

Problem 3: A Task Queue on a Multi-Core, Multi-Threaded CPU

The figure below shows a single-core CPU with an 32 KB L1 cache and execution contexts for up to two threads of control. The core executes threads assigned to contexts T0-T1 in an interleaved fashion by switching the active thread only on a memory stall); **Memory bandwidth is infinitely high in this system, but memory latency on a cache miss is 200 clocks.**

FAQ about the cache: To keep things simple, assume a cache hit takes only a one cycle. Assume cache lines are 4 bytes (a single floating point value), and the cache implements a least-recently used (LRU) replacement policy—meaning that when a cache line needs to be evicted, the line that was last accessed the furthest in the past is evicted. It may be helpful to think about how this cache behaves when a program reads 33 KB contiguous bytes of memory over and over. Hint: confirm to yourself that in this situation every load will be a cache miss.

In this problem assume the CPU performance no prefetching.



You are implementing a task queue for a system with this CPU. The task queue is responsible for executing independent tasks that are created as a part of a bulk launch (much like how an ISPC task `launch` creates many independent tasks). You implement your task system using a pool of worker threads, all of which are spawned at program launch. When tasks are added to the task queue, the worker threads grab the next task in the queue by atomically incrementing a shared counter `next_task_id`. Pseudocode for the execution of a worker thread is shown below.

```
mutex  queue_lock;
int    next_task_id;             // set to zero at time of bulk task launch
int    total_tasks;              // set to total number of tasks at time of bulk task launch
float* task_args[MAX_NUM_TASKS]; // initialized elsewhere

while (1) {

   int my_task_id;

   LOCK(queue_lock);
   my_task_id = next_task_id++;
   UNLOCK(queue_lock);

   if (my_task_id < total_tasks)
       TASK_A(my_task_id, task_args[my_task_id]);
   else
       break;
}
```

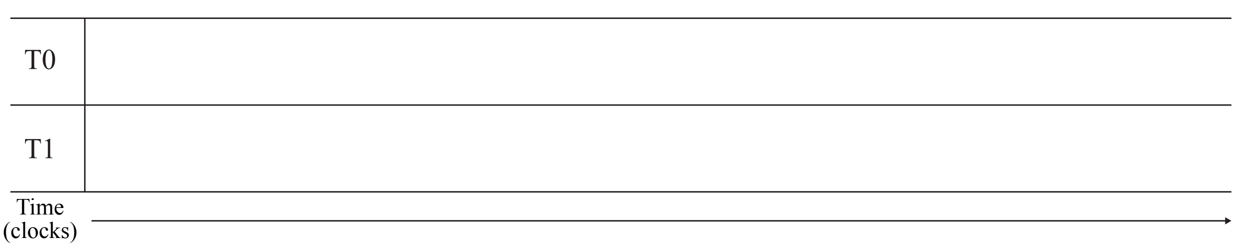
A. Consider one possible implementation of TASK_A from the code on the previous page:

```
function TASK_A(int task_id, float* X) {
   for (int i=0; i<1000; i++) {
      for (int j=0; j<1024*64; j++) {
         load X[j]   // assume this is a cold miss when i=0
         // ... 50 non-memory instructions using X
      }
   }
}
```

The inner loop of TASK_A scans over 64K elements = 256 KB of elements of array X, performing 50 arithmetic instructions after each load. This process is repeated over the same data 1000 times. **Assume there are no other significant memory instructions in the program and that each task works on a completely different input array X (there is no sharing of data across tasks). Remember the cache is 32 KB.**

In order to process a bulk launch of TASK_A, you create two worker threads, WT0 and WT1, and assign them to CPU execution contexts T0 and T1. Do you expect the program to execute *substantially faster* using the two-thread worker pool than if only one worker thread was used? If so, please calculate how much faster. (Your answer need not be exact, a back-of-the envelop calculation is fine.) If not, explain why.

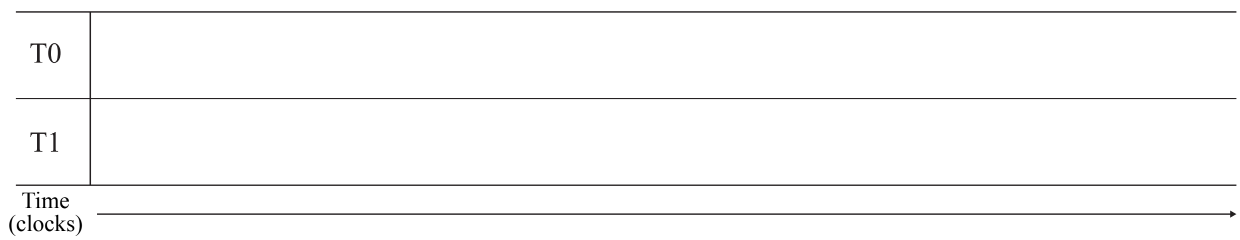
(Careful: please consider the program's execution behavior on average over the entire program's execution ("steady state" behavior). Past students have been tricked by only thinking about the behavior of the first loop iteration of the first task.) It may be helpful to draw when threads are running and stalled waiting for a load on the diagram below.



B. Consider the same setup as the previous problem. How many hardware threads would the CPU core need in order for the machine to maintain peak throughput (100% utilization) on this workload?

C. **Now consider the case where the program is modified to contain 100,000 instructions in the innermost loop.** Do you expect your two-thread worker pool to execute the program *substantially faster* than a one thread pool? If so, please calculate how much faster (your answer need not be exact, a back-of-the envelop calculation is fine). If not, explain why.

D. **Now consider the case where the cache size is changed to 1 MB and you are running the original program from Part A (50 math instructions in the inner loop).** When running the program from part A on this new machine, do you expect your two-thread worker pool to execute the program *substantially faster* than a one thread pool? If so, please calculate how much faster (your answer need not be exact, a back-of-the envelop calculation is fine). If not, explain why.



E. **Now consider the case where the L1 cache size is changed to 384 KB.** Assuming you cannot change the implementation of `TASK_A` from Part A, would you choose to use a worker thread pool of one or two threads? Why does this improve performance and how much higher throughput does your solution achieve?