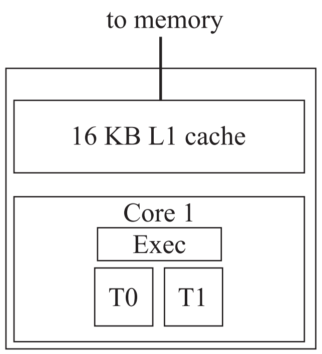
CMU 15-418/618: Parallel Computer Architecture and Programming
Practice Exercise 1

A Task Queue on a Multi-Core, Multi-Threaded CPU

Problem 1. (15 points):

The figure below shows a simple single-core CPU with an 16 KB L1 cache and execution contexts for up to two threads of control. Core 1 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 is 125 clocks. A cache hit is only 1 cycle. A cache line is 4 bytes. The cache implements a least-recently used (LRU) replacement policy.**



You are implementing a task queue for a system with this CPU. The task queue is responsible for executing large batches of 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
int*  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. (3 pts) Consider one possible implementation of `TASK_A` from the code on the previous page:

```
function TASK_A(int task_id, int* X) {
   for (int i=0; i<1000; i++) {
      for (int j=0; j<8192; j++) {
         load X[j]   // assume this is a cold miss when i=0
         // ... 25 non-memory instructions using X
      }
   }
}
```

The inner loop of `TASK_A` scans over 32 KB of elements of array `X`, performing 25 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 16 KB, a cache line is 4 bytes, and the cache implements a LRU replacement policy. Assume the CPU performs no prefetching.**

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.

16 KB = 2^14 bytes

2^2 cache lines

Scans 2^15 Bytes over 2^13 iterations =>
Each elem is 2^2 bytes [let's call it an int]

Performing 25 instructions sequentially versus using two threads w/two execution contexts doesn't really make that much of a difference because the cache will miss quite a bit after the first 2^12 ints, meaning that it will stall while the other context tries to complete, but it too will stall because the other context only has 25 ops to do. There is a good chance that both will stall at the same time after this point. Regardless, speedup from the first 1/4 of the program running time should be about 2x.

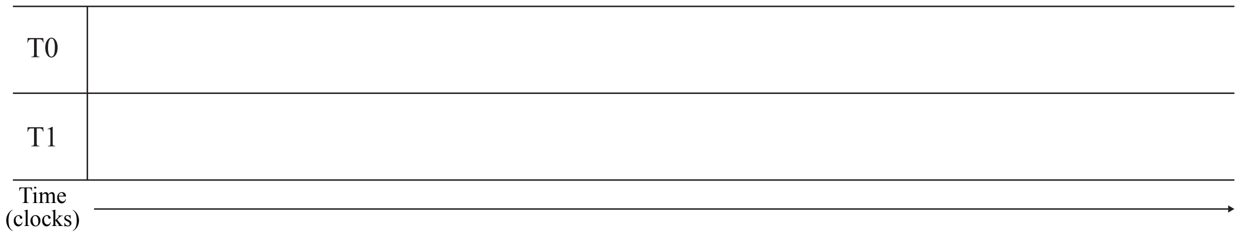
T0	note: I wrote these out instead of drawing them on the computer
T1	
Time (clocks)	→

B. (3 pts) **Now consider the case where the program is modified to contain 10,000 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.

I expect significantly faster execution - roughly 125x - because the 10000 instructions allows more than enough time for the stalled thread to pass control to another thread and get its information from memory before the other thread stalls as well. In this scenario, there would be few if any noticeable stalls.

C. (3 pts) **Now consider the case where the cache size is changed to 128 KB and you are running the original program from Part A (25 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.

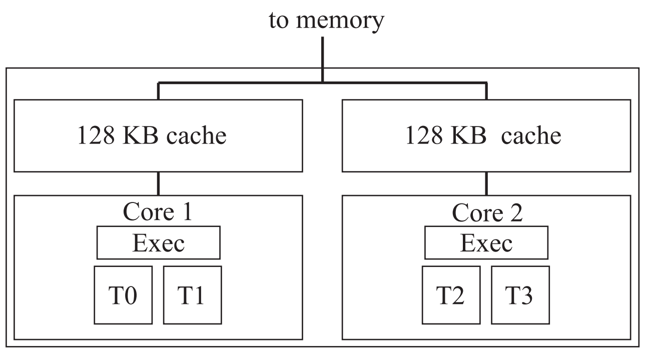
I don't think there would be much of a difference here (roughly 1x), as the entire array can fit in the cache without any misses occurring, so there is little stall time that multithreading can take advantage of.



D. (3 pts) **Now consider the case where the L1 cache size is changed to 48 KB.** Assuming you cannot change the implementation of `TASK_A` from Part A, how should your system schedule tasks to *substantially improve* program performance over the two-worker pool approach? Why does this improve performance and how much higher throughput does your solution achieve?

The system should automatically switch tasks when it stalls, instead of segmenting one task using the two-worker pool approach. This way, instead of having to potentially deal with multiple stalls at the same time, the system can simply run operations on the other one without having to worry about this issue. This achieves roughly a 2x speedup as a result.

E. (3 pts) Now consider the case where the task system is running programs on a dual-core processor. Each core is two-way multi-threaded, so there are a total of four execution contexts (T0-T3). Each core has a 128 KB cache.



If you maintain your two-worker thread implementation of the task system as discussed in prior questions, to which execution contexts do you assign the two worker threads WT0 and WT1? Why? Given your assignment, how much better performance do you expect than if your worker pool contained only one thread?

I would assign WT0 to T0 and WT1 to T2, or put them on separate cores. I expect better performance from this compared to previous question with one thread because there’s only a joint period of stall time before both can perform the operations they need concurrently with very little stall time afterwards, since the cache can fit the whole array. This is more than twice as fast as the one thread version.

Problem 2. (15 points):

Consider the following ISPC code that computes $ax^2 + bx + c$ for elements x of an entire input array.

```
void polynomial(float a, float b, float c,
                uniform float x[], uniform float output[], int elementsPerTask) {
  uniform int start = taskIndex * elementsPerTask;
  uniform int end = start + elementsPerTask;

  foreach (i = start ... end) {
    output[i] = (a * x[i] * x[i]) + (b * x[i]) + c;   // 5 arithmetic ops
  }
}

// assume N is very, very large, and is a multiple of 1024
void run(int N, float a, float b, float c, float* input, float* output) {
  uniform int elementsPerTask = 1024;
  launch[N/elementsPerTask] polynomial(a, b, c, input, output, elementsPerTask);
}
```

Professor Kayvon, seeking to capture the highly lucrative polynomial evaluation market, builds a multi-core CPU packed with ALUs. "The professor with the most ALUs wins, he yells!" The processor has:

- 4 cores clocked at 1 GHz, capable of one 32-wide SIMD floating-point instruction per clock (1 addition, 1 multiply, etc.)
- Two hardware execution contexts per core
- A 1 MB cache per core with 128-byte cache lines (In this problem assume allocations are cache-line aligned so that each SIMD vector load or store instruction will load one cache line). Assume cache hits are 0 cycles.
- The processor is connected to a memory system providing a whopping 512 GB/sec of BW
- The latency of memory loads is 95 cycles. (There is no prefetching.) For simplicity, assume the latency of stores is 0.

A. (1 pt) What is the peak arithmetic throughput of Prof. Kayvon's processor?

The peak arithmetic throughput is 4 * 8 * 1 * 2 = 32 gigaflops (assuming it takes 8 ALUs/ core to be able to do one 32-wide SIMD floating point instruction per clock)

B. (1 pt) What should Prof. Kayvon set the ISPC gang size to when running this ISPC program on this processor?

He should set the gang size to 32.

C. (3 pts) Prof. Kayvon runs the ISPC code on his new processor, the performance of the code is not good. What fraction of peak performance is observed when running this code? Why is peak performance not obtained?

The latency for memory access prevents the code from running at its peak performance because it requires numerous memory accesses in any execution of this program.

D. (3 pts) Prof. Bryant sees Kayvon's struggles, and sees an opportunity to start his own polynomial computation processor company, RandyNomial that achieves double the performance of Prof. Kayvon's chip. "Oh shucks, now I'll have to double the number of cores in my chip, that will cost a fortune." Kayvon says.

TA Ravi writes Kayvon an email that reads "There's another way to achieve peak performance with your original design, and it doesn't require adding cores." Describe a change to Prof. Kayvon's processor that causes it to obtain peak performance on the original workload. Be specific about how you'd realize peak performance (give numbers).

The way to go about this would be to implement new execution contexts. A higher number like 16 would work, since the working set is comprised of a couple of floats. This would help achieve peak performance because it takes advantage of multithreading to, while doing multiple tasks at the same time, hide stalls and memory latency in any one process by simple doing another task when this happens.

The following year Prof. Kayvon makes a new version of his processor. The new version is the **exact same quad-core processor** as the one described at the beginning of this question, except now the chip **supports 64 hardware execution contexts per core**. Also, the ISPC code is changed to compute a more complex polynomial. In the code below assume that `coeffs` is an array of a few hundred polynomial coefficients and that `expensive_polynomial` involves 100's of arithmetic operations.

```
void polynomial(uniform float coeffs[], uniform float input[],
                uniform float output[], int elementsPerTask) {
  uniform int start = taskIndex * elementsPerTask;
  uniform int end = start + elementsPerTask;
  foreach (i = start ... end) {
    output[i] = expensive_poly(coeffs, input[i]);   // 100's of arithmetic ops
  }
}

void run(int N, float* coeffs, float* input, float* output) {
  uniform int elementsPerTask = 1024;
  launch[N/elementsPerTask] polynomial(coeffs, input, output, elementsPerTask);
}
```

E. (1 pt) What is the peak arithmetic throughput of Prof. Kayvon's new processor?

The peak arithmetic throughput is still 4 * 8 * 1 * 2 = 32 gigaflops (assuming it takes 8 ALUs/ core to be able to do one 32-wide SIMD floating point instruction per clock) [not sure if the 64 execution contexts/core makes a difference]

F. (3 pts) Imagine running the program with N=8×1024 and N = 64×1024. Assuming that the system schedules worker threads onto available execution contents in an efficient manner, do either of the two values of N result in the program achieving near peak utilization of the machine? Why or why not? (For simplicity, assume task launch overhead is negligible.)

Neither of the values gets peak utilization because the first value underutilizes the multithreading capabilities of the processors, which would cost a lot especially for the more expensive polynomial operation. Similarly, multithreading isn't really a factor for the second one either.

G. (3 pts) Now consider the case where N=9×1024. Now what is the performance problem? Describe is simple code change that results in the program obtaining close to peak utilization of the machine. (Assume task launch overhead is negligible.)

The performance problem is now that one processor would be stuck with one more thread than the other ones, which would prevent the machine from getting peak utilization.