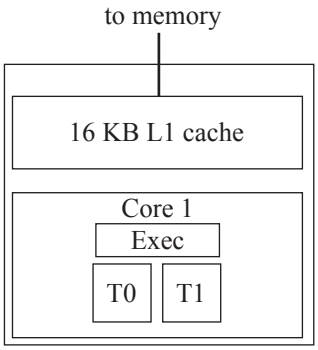
**CMU 15-418/618: Parallel Computer Architecture and Programming**
**Practice Exercise 1 SOLUTIONS**

**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:

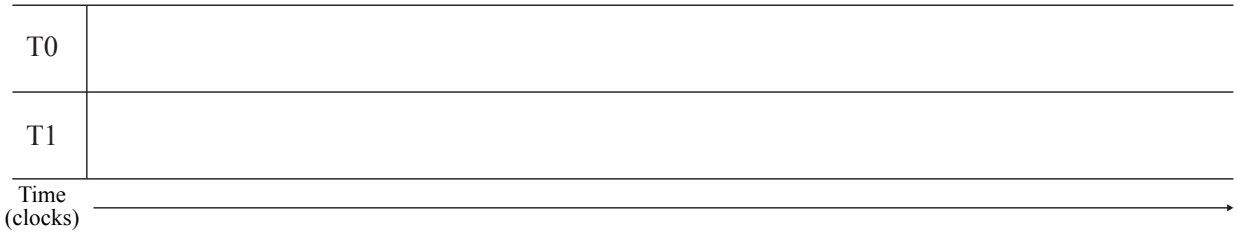
```
function TASK_A(int task_id, int* X) {
   for (int i=0; i<1000; i++) {
      for (int j=0; j<8192; j++) {
         load X[j]   // assume this is a cold miss when i=0
         // ... 25 non-memory instructions using X
      }
   }
}
```

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.*

*Solution: The two-thread configuration will execute about 2 times faster. All memory accesses are cache misses and incur a 125 cycle latency. The one thread implementation proceeds with 125 cycles of memory stall, followed by 25 cycles of math, then 125 cycles of memory stall, followed by 25 cycles of more math, etc. Therefore, it results in a core utilization of 16.6% (it is doing useful processing 1/6 of the time). The two-thread configuration is able overlap 25 of the 125 stall cycles with execution of arithmetic operations from the other thread. As a result, it achieves 33% utilization and runs* ***twice as fast****. An interesting note is that a processor supporting six interleaved hardware threads would be required to reach 100% ALU utilization.*

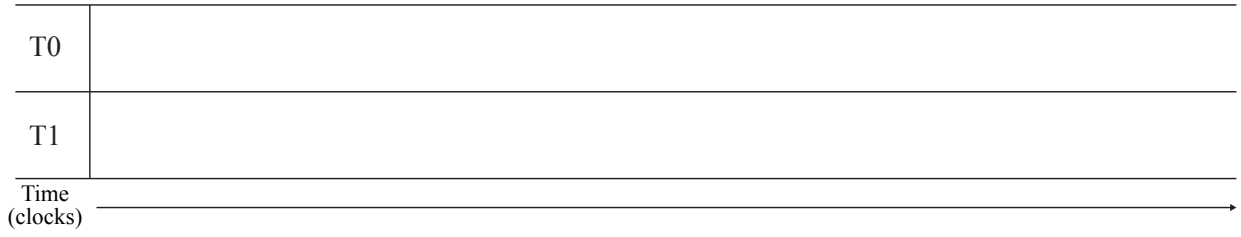


B. (3 pts) **Now consider the case where the program is modified to contain 10,000 instructions in the innermost loop.** Do you expect your two-thread worker pool to execute the program *substantially faster* than a one thread pool? If so, please calculate how much faster (your answer need not be exact, a back-of-the envelop calculation is fine). If not, explain why.

*Solution: There is **not** a substantial difference in performance because both the one and two-thread configurations will run at near 100% utilization of the processor (and thus operate at about the same speed). The reason for this is that the runtime of the program is now dominated by arithmetic, not memory access, so that even though the single threaded implementation stalls waiting on memory, there are only 125 stall cycles for every 10K arithmetric instructions, which yields about 99% utilization.*

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.

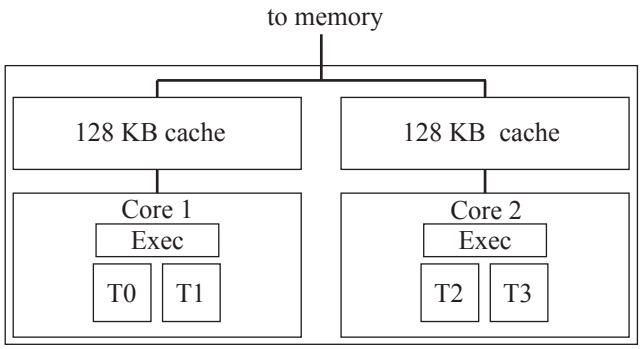
*Solution: There is **not** a substantial difference in performance because both the one and two-thread configurations will run at near 100% utilization of the processor (and thus operate at about the same speed). The reason for this is that there are essentially no cache misses in this scenario. For all loops where `i>0`, all memory accesses are serviced by the cache. As a result, there is no latency to hide and thus no benefit from hardware multi-threading in this situation.*



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, 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?

*Solution: Now, when running one thread, the thread's working set fits in the cache. The thread takes essentially no cache misses and runs at 100% utilization (just like in Part A). The two-thread configuration has a working set size of 64 KB (but the CPU's cache is only 48 KB) and thus, just like in part A, all memory accesses are cache misses. We know from part A that the two-thread configuration will realize 33% core utilization, and thus the* ***one thread configuration is three times as fast!***

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?

*Solution: You want to schedule the worker threads to execution contexts T0 and T2 (on different cores). This assignment allows you to DOUBLE performance over the one-thread configuration since twice as many execution resources could be used. (Two instructions per clock can be executed, not just one! Note that caching effects were not relevant on this problem since with a 128 KB cache like in part B, the working set for two threads can fit in the cache on a single core—although spreading the worker threads out to different cores essentially doubles the effective cache size for each thread, the program's working set is not large enough to benefit from the additional capacity.*

**Because The Professor with the Most ALUs (Sometimes) Wins**

## 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?

*Solution: 4 cores $\times$ 32-wide SIMD ALUs $\times$1 GHz = 128 GFLOPS*

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

*Solution: It should be (at least) 32, since that is the SIMD width of the machine.*

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?

*Solution: The code is memory latency bound. In steady state, a thread waits 95 cycles for a load, then performs 5 math ops, then waits another 95 cycles for a load, etc. Only 5 of these 95 stall cycles can be hidden by the other thread, so it runs at 10/100, or 1/10 of peak. It should be noted that this code is not bandwidth bound—even if the core was somehow running at peak rate, there would be sufficient bandwidth to feed the processor.*

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).

*Solution: Add more hardware multi-threading. With 20 threads, memory latency will be fully hidden an the system will run at peak rate.*

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?

*Solution: Still 128 GFLOPS. Peak compute capability has not changed by adding more hardware execution contexts.*

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.)

*Solution: In all cases, yes. Even though all thread slots are not utilized, there is no benefit to more than one thread of memory latency hiding in this scenario.*

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.)

*Solution: Drop the elements per task so that there are at least as many tasks as execution contexts. A solution should also make sure the number of tasks is an exact multiple of the number of execution contexts, or at least several times greater.*