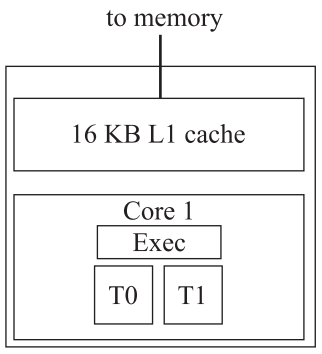
CMU 15-418/618: Parallel Computer Architecture and Programming
Practice Exercise 1

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:

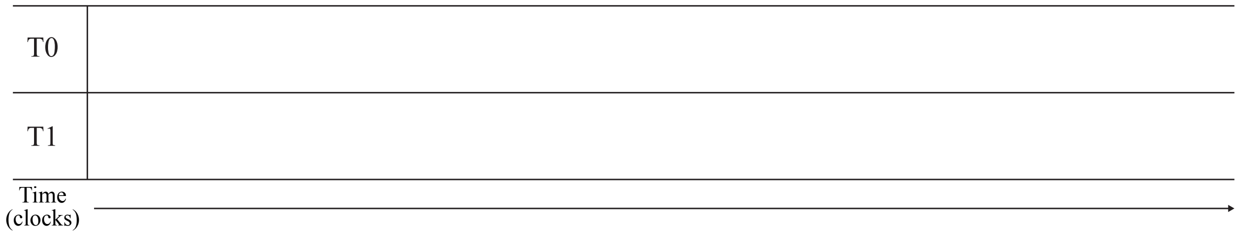
```
function TASK_A(int task_id, int* X) {
   for (int i=0; i<1000; i++) {
      for (int j=0; j<8192; j++) {
         load X[j]   // assume this is a cold miss when i=0
         // ... 25 non-memory instructions using X
      }
   }
}
```

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.

Answer: I think it is 2 speedup from the serial version, even though we do not have spacial locality because the cache line size is 4B for only one integer, which means we need to load every element in array and wait for 125 cycles. On the other hand, we do not have temporal locality also, because the whole cache can only contain 16KB/4B=4K integers, which means before we use the element again in the 2nd loop it is already evicted to leave space to new element. So in this situation, we have no benefit from cache. However, even though we do not have cache as a speed tool, we still will benefit from creating 2 tasks. Because, we can let CPU to send out two load instructions at the same time, and we can get data from memory by only using 125 cycles instead of 250 cycles when doing serial loading. So I think the speedup should be 2X.

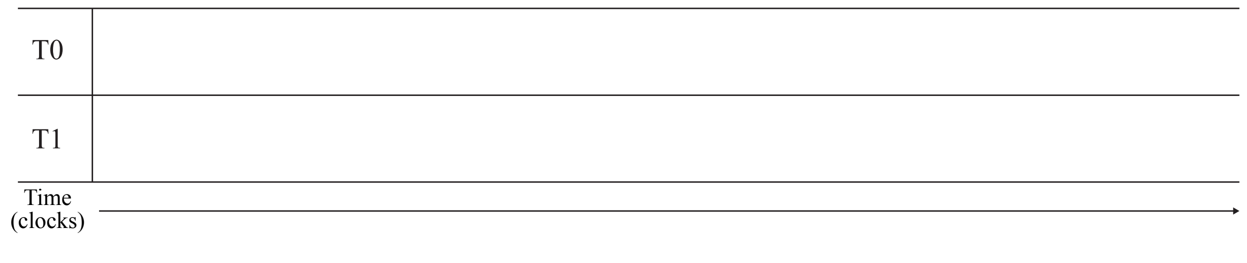


B. (3 pts) **Now consider the case where the program is modified to contain 10,000 instructions in the innermost loop.** Do you expect your two-thread worker pool to execute the program *substantially faster* than a one thread pool? If so, please calculate how much faster (your answer need not be exact,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

I do not expect it will have substantially faster speed than one thread, because this time the execution time is mostly spent on running instructions in the inner loop after loading data. Since we have only execution unit for our system, we cannot run two different instructions stream at the same time, which means we have to wait one thread to running its current instructions, and after some time, the control will return back to another thread to continue. So I think it will not improve the performance.

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.

Yes, it will have a 2X speedup too. Because for the first 8k elements loop we will always wait before every element in array is loaded into cache, so only the first 8k elements will we wait for, and the next 999 outer loop will be very fast because array is all fit into the cache for both threads(32KB * 2 < 128KB). If we use two threads, we will conduct 2 SIMD load instructions during 125 cycles. So it will be as 2 times faster as serial version.

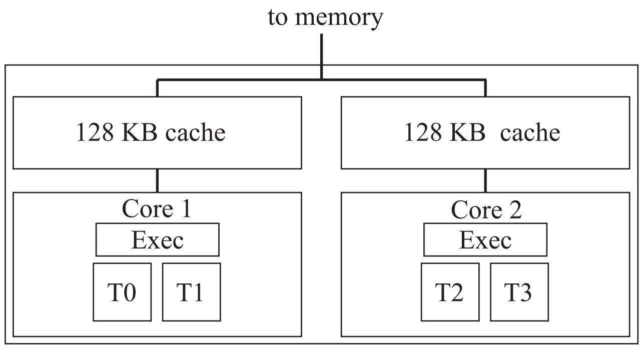


D. (3 pts) **Now consider the case where the L1 cache size is changed to 48 KB.** Assuming you cannot change the implementation of `TASK_A` from Part A, how should your system schedule tasks to *substantially improve* program performance over the two-worker pool approach? Why does this improve performance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

I will choose run only one thread over the entire array, because 48KB can hold the entire working set(32KB) by one thread instead of 68KB working set from two thread. Because the penalty for loading evicted data is very huge, for one thread, we just need to load 32KB data in the first loop, and then the rest of 999 loops will be calculated very fast, However, if we use two threads at the same time, there will be a conflict between two threads, and maybe there will many many reload operation due to the conflict, which will reduce the performance a lot.
For two-worker approach: assume every thread can get half of cache, which is 24KB, so the 32KB cannot fit into 24KB, which means we will miss every time we load the data:
total cycles = (125 + 50) cycles/inner loop * 8192 inner loop/outer loop * 1000 outer loop = 128800000 cycles.
For one-worker approach: Since now we have 48KB cache that can hold entire array:
total cycles = (150 cycles / 1st inner loop + (1 + 26) cycles / rest of 999 loop) * 2 = 2509548 cycles

So the speedup = 128800000 cycles / 2509548 cycles = 5.1X

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 will assign two threads on different cores because by doing that I can truly run two instruction streams simultaneously, and they can have much more cache independently.
I think the speedup is still 2X, because it just make two threads to run at the same time to double the throughput.

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?

32 wide means we can perform 8 float arithmetic operation at the same time.
We have 4 cores at the speed of 1Ghz, so for 1 second we can perform 4 * 8 * 1000000000 ops.
So we can perform 32000000000 / 5 = 6400000000 iterations/second.
For each iteration, we will have move 8B + 4B = 12B.
Total throughput = 6400000000 * 12B = 76.8GB/sec

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

32/4 = 8

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?

Since 1M cache has 128KB cache line, which means it can hold 32 float number at the same time, since we never reuse the data, so we do not consider the eviction behavior. 1M will have 8192 cache lines, so both output and x can be placed into cache. a cache line holding 32 float number at the same means that we will encounter a cache miss for every 32 iterations:
So the total number of cycles for ever 32 iteration is (95 + 5) + 31 * 5 = 255 cycles instead of 32 * 5 = 160 cycles, so the real throughput for this system is:
4 * 8 * 1000000000 / 255 * 32 * 12B = 48.2 GB/sec

So the fraction is (48.2GB/sec) / (76.8GB/sec) * 100% = 62%

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).

I would to increase the number of each core can handle concurrently such as hyper threading, which we can use 2 thread contexts per core to handle four threads at the same time, and every time when the thread get into stall, another thread is switched into core and then run another loading action. With this modification, we can running 4 SIMD loading instructions at the same time with only one load delay(because they happen at the same time). In that way we can save much time on loading data from memory to cache and can have a better performance.

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?

If assume coeffs[] does not the data transmission in this calculation.
4 * 8 * 1000000000 /100 * 12B = 3.84GB/sec

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.)

No, the latter one get a better utilization. Because now the chip can support more contexts, it can switch to another thread when encounter a stall to hide the latency of loading instruction. So the latter case will be less sensitive to the influence of high latency caused by memory referencing.

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.)

If the tasks are assigned to every core equally, and the first 8 tasks will be assigned to 4 cores, and each core has 2 tasks to perform, however the remaining will be added to a core making its workload unbalanced which means the core that is assigned to do 9th work will still run after other cores have finished. It does not utilize the 4 cores most effectively.

My change is that modify elementsPerTask for the run() function from 1024 to 9 * 1024 / 8 = 1152. In that way, we can assign a balanced workload to every core to achieve peak uitilization.