Stanford CS149: Parallel Computing
Written Assignment 1 SOLUTIONS

Hardware Basics

Problem 1. (33 points):

A. (8 pts) Consider a multi-core processor that has two cores. Each core runs at 1 GHz (1 billion operations per clock). Each core is single-threaded (meaning it only maintains state for a single execution context) and can compete one single-precision floating point arithmetic operation per clock. What is the peak arithmetic throughput of the processor in terms of floating point operations per second?

Solution: 2 billion ops/sec. (2 ops per clock is also a valid answer)

B. (8 pts) Now imagine the cores from part A are upgraded so that they perform 16-wide SIMD instructions. Assuming these cores still complete one of these SIMD instructions per clock, what is the peak arithmetic throughput of the processor (in terms of floating point operations per second)?

Solution: 2 cores $\times$ 16-ops/core $\times$ 1 GHz = 32 billion ops/second (or equivalently, 32 ops/clock)

C. (8 pts) Finally, imagine that each core from part B was a multi-threaded core that maintain execution contexts for up to four hardware threads each. What is the peak arithmetic throughput of the processor in terms of floating operations per second?

Solution: It is still 32 billion ops/second (or 32 ops/clock). Adding multi-threading support does not change the peak arithmetic throughput of the processor. It only increases the ability of the core to hide stalls.

D. (9 pts) Imagine that each core from part C was further modified to support superscalar execution where the core can complete one scalar floating point operation and one 16-wide SIMD instruction per clock from the same thread (if those instructions are independent). What is the peak arithmetic throughput of the processor (in terms of floating point operations per second)?

Solution: 2 cores $\times$ 17-ops/core $\times$ 1 GHz = 34 billion ops/second. Given the appropriate workload a core can now issue two instructions per clock, one does 1 floating point operation, and the other does 16.

Caching Basics

Problem 2. (33 points):

A. (11 pts) Assume we are running a program on a processor with a data cache. All data loaded from memory is first loaded into the processor's data cache, and then transferred from the cache to the processor's registers. (This is true of most systems.) **When new data is brought into the cache, the cache has a policy of evicting the least recently used data** (the data that has been accessed the longest time ago) to make room for the newly accessed data. Imagine that the cache is 32 KB, and consider running the following program:

```
const int SIZE = 64 * 1024;
float mydata[SIZE];  // hint: how much data is this?

float sum = 0.0;
for (int i=0; i<100; i++) {
  for (int j=0; j<SIZE; j++) {
    sum += mydata[j];
  }
}
```

A **cache miss** occurs when a processor accesses data from memory that is not present in the cache. When the program starts running, each each of data accessed during the `i=0` iteration of the outer loop is a cache miss, since that is the first time the data was accessed in the program. Now consider the entire program's execution. Please describe what fraction of the accesses to the `mydata` array will be cache misses. (For those that are familiar with the details of cache operation, please assume that the cache is fully-associative, and has a cache line size of one float. If these terms are unfamiliar to you, you can safely ignore them... or Google it!)

Solution: All accesses are a cache miss. The size of the array is 256 KB (recall a `sizeof(float)` is 4 bytes. By the time the program accesses the data again in a later iteration of the outer loop, the data will have been evicted from the cache. So all accesses are a miss, regardless of the fact that each iteration of the loop is acccessing the same data as prior iterations. In other words the "working set" of this program is larger than the capacity of the cache.

B. (11 pts) Now consider the case where `SIZE = 2048`. Does your answer to part A change with this new array size. Why or why not?

Solution: Indeed it does. Now the program only scans over 8 KB of data each outer loop iteration. Since the data accesses in previous loop iterations is present in the cache for subseuent iterations, all accesses are cache hits after the first iteration of the outer loop. So given the code does a total of 100 iterations, 99% of accesses are hits.

C. (11 pts) Now assume that you are running the following code. Would you rather have a processor with a 32 KB data cache, or would you rather add multi-threading to the processor. Why or why not?

```
float sum = 0.0;
for (int i=0; i<1000000; i++) {
  sum += 2.0 * myarray[i] + 1.0;
}
```

Solution: In this example, no data from `myarray` *is ever accessed more than once. Since there is no reuse, there is no latency reduction benefit from a cache. Since latency cannot be reduced by turning accesses into cache hits, it would be more useful to have multi-threading support to hide the potental memory latency stalls by overlapping the execution of math operations with data access.*

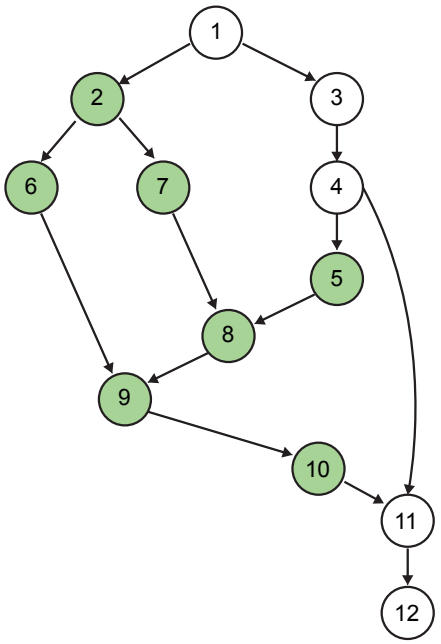
Problem 3. (33 points):

Consider the following sequence of 12 instructions. There is a load operation, followed by 10 math operations, followed by a store. Note that some of the instructions are scalar instructions, and others are vector instructions operating on vector registers (Vx registers). The vector operations have "V" at the beginning of their instruction names.

```
1.   LD      R1   <- [R0]
2.   VSPLAT  V0   <- R1        // copy R1 into all elements of V0
3.   MUL     R2   <- R1, R1
4.   ADD     R2   <- R2, 16    // R2 + 16
5.   VSPLAT  V1   <- R2        // copy R2 into all elements of V1
6.   VMUL    V2   <- V0, V0
7.   VADD    V3   <- V0, V0
8.   VMUL    V3   <- V1, V3
9.   VMUL    V3   <- V2, V3
10.  VRED    R1   <- V3        // reduction: sum all elements of V3 into R1
11.  MUL     R1   <- R1, R2
12.  ST      [R4]  <- R1
```

A. (10 pts) Please draw the dependency graph for the instruction sequence.

Solution: Vector instructions given in green:



B. (8 pts) Imagine the instruction sequence is executed on a *single-core, single-threaded processor*. The processor supports **superscalar execution** in that it can fetch/decode up to **two instructions per clock**, but it has one scalar execution unit and one vector execution unit. Therefore, **it can run instructions IN ANY ORDER THAT RESPECTS THE PROGRAM DEPENDENCY GRAPH**, but it can only run two independent instructions per clock if and only if one instruction is a scalar instruction and the other instruction is a vector instruction. Assuming that all instructions take 1 cycle to complete, how many cycles does it take to complete this instruction sequence?

Solution: 10 cycles. A good solution would be to draw out a schedule of what the scalar and vector units are doing each clock. There is opportunity to execute two instructions at once in only two cycles. For example, one solution is:

```
clock       scalar             vector
0            LD  (1)
1            MUL (3)           VSPLAT (2)
2            ADD (4)           VMUL   (6)
3                              VSPLAT (5)
4                              VADD   (7)
5                              VMUL   (8)
6                              VMUL   (9)
7                              VRED   (10)
8            MUL (11)
9            ST  (12)
```

Other valid solutions might have a different order for vector instructions 5, 6, and 7 (they are independent instructions and so can be carried out in any order) but all schedules will complete in the same amount of time.

C. (7 pts) Now assume the sequence of instructions on the previous page is run in a loop. For example:

```
float in[VERY_BIG];  // let VERY_BIG = 100,000,000
float out[VERY_BIG];

// parallelize iterations of this loop using threads
#pragma omp parallel for
for (int i=0; i<VERY_BIG; i++) {
   // assume myfunc() is the 10 non-LD/ST instrs in the seq above
   out[i] = myfunc(in[i]);
}
```

The loop is run on the same single core, single-threaded processor as before, **but now the latency of a LD instruction is 20 clocks.** (That is, if the LD instruction begins on clock c, an instruction depending on the LD can begin on clock $c + 20$. (There is one cycle to execute the LD instruction on the processor's scalar unit, followed by 19 cycles of waiting before the dependent instruction can begin.) All other instructions still have a latency of 1 cycle.

In this setup, **what is the utilization of the core's vector execution unit?** (What fraction of cycles is the vector unit executing instructions? Hint: What are all the reasons the vector unit might not be utilized? Note: it's fine to give your answer as a fraction.)

Solution: Now the sequence takes 10+19=29 clocks. Of those 29 clocks, the vector unit is operating for 7 cycles. Therefore the utilization is 7/29.

D. (8 pts) Now imagine the core is multi-threaded, and can choose **to simultaneously execute two instructions from the same thread, or from different threads, provided they meet the one scalar and one vector instruction per clock constraint.** In this design, can you achieve full utilization of the vector unit? If so, how many total threads do you need (please explain why) If not, explain why.

Solution: Yes, it is possible using a total of five threads. Notice that each thread operates in a loop of executing 7 vector instructions, then the vector unit goes idle for 22 cycles. Therefore, we need four additional threads ($4 \times 7 > 22$) to supply instructions to keep the unit busy. Also, with five total threads in the system, there's now more than enough work to do to hide memory latency of a stalled threads.

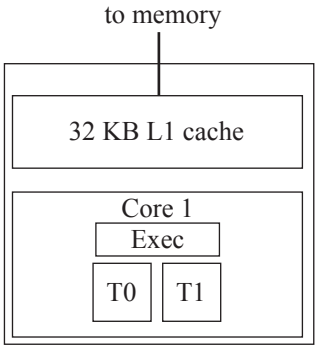
Note that only four total threads are necessary to hide the 19 stall cycles that would otherwise occur due to memory latency (each additional thread can hide 9 cycles of stall). With four threads the core will be executing at least one instruction each clock, but it will not be able to be execute a vector operation every clock.

PRACTICE PROBLEM 1: 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:

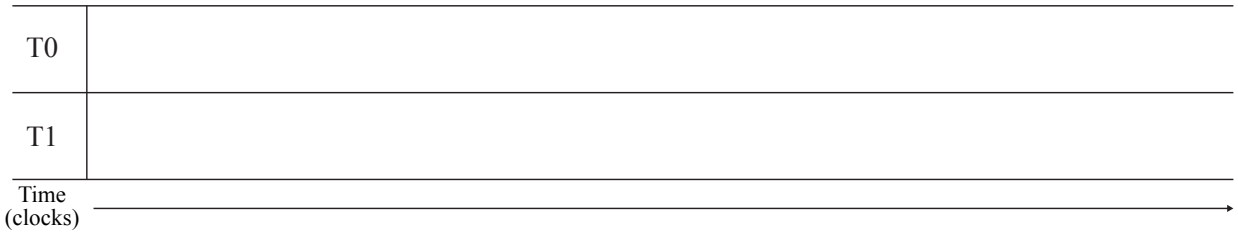
```
function TASK_A(int task_id, float* X) {
   for (int i=0; i<1000; i++) {
      for (int j=0; j<1024*64; j++) {
         load X[j]   // assume this is a cold miss when i=0
         // ... 50 non-memory instructions using X
      }
   }
}
```

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.

Solution: The two-thread configuration will execute about 2 times faster. All memory accesses are cache misses and incur a 200 cycle latency. The one thread implementation proceeds with 200 cycles of memory stall, followed by 50 cycles of math, then 200 cycles of memory stall, followed by 50 cycles of more math, etc. Therefore, it results in a core utilization of 20% (it is doing useful processing 1/5 of the time). The two-thread configuration is able overlap 50 of the 200 stall cycles with execution of arithmetic operations from the other thread. As a result, it achieves 40% utilization and runs ***twice as fast****.*



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?

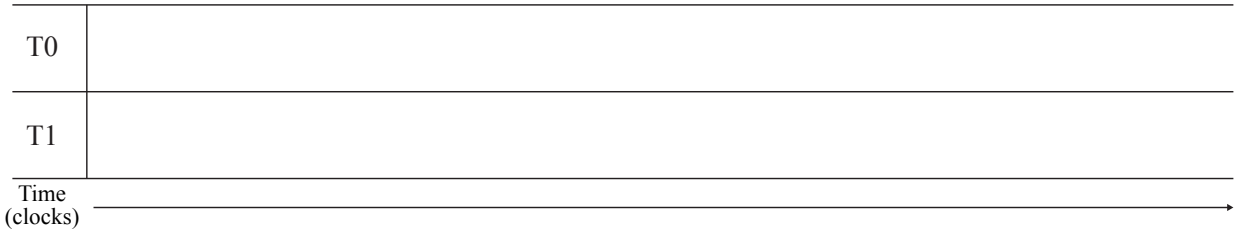
Solution: With five hardware-threads all 200 cycles over memory latency can be hidden. When one thread incurs a memory stall, 4 other threads work of loop body work are needed to fully hide the 200 cycles of memory latency.

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.

*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 200 stall cycles for every 100K arithmetic instructions, which yields neary 100% utilization.*

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.

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.*



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?

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 256 KB (but the CPU's cache is only 384 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 40% core utilization, and thus the ***one thread configuration is 2.5× faster!***

PRACTICE 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)

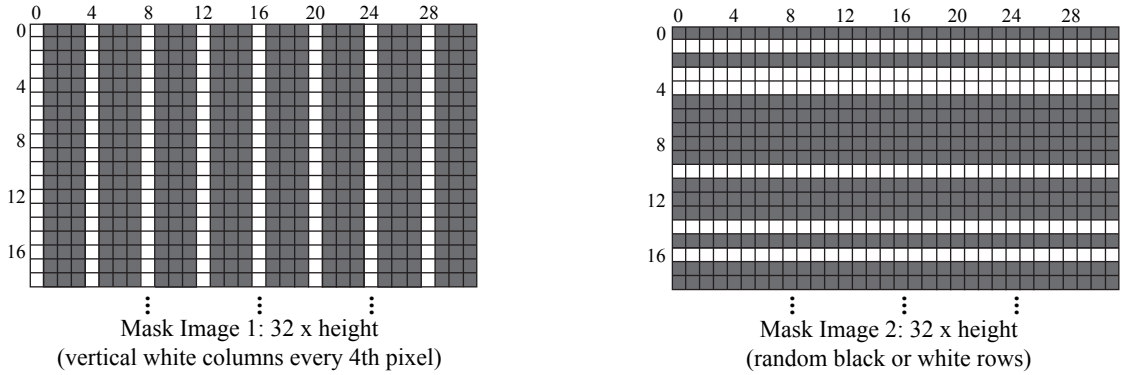


Figure 1: Image masks used to govern image manipulation by `brighten_image`

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.)

Answer: B > A > C

For image 1, each row of the mask is mixture of white and black pixels: every 1 white pixel is followed by 3 black pixels. Since the SIMD width for CPUs A and B are 32 and 2 respectively these processors will suffer from ***branch divergence*** *when executing this ISPC code. ISPC program instances working on a black pixel will wait for gang instances assigned white pixels to finish their execution of 1000 loop iterations.*

Now let's calculate how many cycles it takes for ONE CORE of each processor to finish rendering a single row:

- A: 1000 cycles
- B: $10 \times 8 + 1000 \times 8 = 8080$ cycles
- C: $10 \times 24 + 1000 \times 8 = 8240$ cycles

However, processor C is clocked at 4 GHz (A and B run at 1 GHz) and processor B has 12 cores. Thus the effective per-row time for each platform will be:

- A: $1000 = 1000$ cycles per row
- B: $8080 \div 12 = 673$ cycles per row
- C: $8240 \div 4 = 2060$ cycles per row

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.

Answer: A > B > C

In image 2, unlike image 1, all the rows are homogeneous. As a result, means processor B and C no longer suffer from ***branch divergence****. Since all processor execute at their peak rates, the processor with the most raw processing power will provide the best processing speed. A provides the most raw processing power, followed by B.*

PRACTICE PROBLEM 3: Be a Parallel Processor Architect

You are hired to start the parallel processor design team at Lagunita Processors, Inc. Your boss tells you that you are responsible for designing the company's first shared address space multi-core processor, which will be constructed by cramming multiple copies of the company's best selling uniprocessor core on a single chip. Your boss expects the project to yield at least a $5\times$ speedup on the performance of the program given below. You are not allowed to change the program, and assume that:

- Each Lagunita core can complete one floating point operation per clock
- Cores are clocked at 1 GHz, and each have a 1 MB cache using LRU replacement.
- All Lagunita processors (both single and multi-core) are attached to a 100 GB/s memory bus
- Memory latency is perfectly hidden (Lagunita processors have excellent pre-fetchers)

```
float A[N]; // let N = 100 million elements
float total = 0;

// ASSUME TIMER STARTS HERE ////////////////////////////////

for (int i=0; i<N; i++)
   total += A[i];

for (int i=0; i<9; i++) {

   // made up syntax for brevity: 'parallel_for'
   // Assume iterations of this loop are perfectly partitioned
   // using blocked assignment among X pthreads each running on
   // one of the processor's X cores.
   parallel_for(int j=0; j<N; j++) {
      A[j] = A[j] / total;
   }
}

// ASSUME TIMER STOPS HERE /////////////////////////////////
```

A. How do you respond to your boss' request? Do you believe you can meet the performance goal? If yes, how many cores should be included in the new multi-core processor? If no, explain why.

Solution: In the provide code, 10% of the program is sequential (the sum reduction computing `total`*), and 90% of the code is perfectly parallelizable. By Amdahl's Law (with* $S = 0.1$*) this means that the maximum parallel speedup under infinite processing capability is* $1/S = 10\times$*.*

Solving $5 = \frac{1}{.1+\frac{.9}{P}}$ *for* P *yields* $P = 9$*. So you tell your boss you can meet the goal with 9 cores.*

Note that 9 cores running at full speed achieve a throughput of 9 GFLOPS, which in this problem would consume about 9 *cores* $\times$ 2 *mem ops* $\times$ 4 *bytes* $\times$ 1 *GHz* $= 72$ *GB/sec of bandwidth. The memory system can sustain this throughput.*

B. You tell your boss that if you were allowed to make a few changes to the code, you could deliver a much better speedup with your parallel processor design. How would you change the code to improve its performance by improving *speedup*? (A simple description of the new code is fine). If your answer was NO in part one, how many processors are required to achieve $5\times$ speedup now? If your answer was YES, approximately what speedup do you expect from your previously proposed machine on the new code? (*Note: we are NOT looking for answers that optimize the program by rolling multiple divisions into one.*)

Solution: The first change you should make is parallelize the computation of `total`*. This could easily be done by computing a partial sum on each core, and then combining the results. Since we're only talking about a few processors in this example, and the array is said to have 100 million elements, combining the partial sums (albeit serial) constitutes only a tiny fraction of execution time. The entire program is now essentially entirely parallelizable, and so the* $5\times$ *speedup can now be obtained using only 5 cores.*

A few clever students also pointed out that the order of the `i` *and* `j` *loops in the parallel section of the code could be interchanged. The result is a* `parallel_for` *loop whose body performs nine divisions on the same array element before moving onto the next one. This operation dramatically increases the arithmetic intensity of the computation. It does not impact the answer to this question since the program is not bandwidth bound, it but did does impact your answer to part C.*

C. Assume that the following year, Lagunita Processors, Inc. decides to produce a 32-core version of your parallel CPU design. In addition to adding cores, your boss gives you the opportunity to further improve the processor through one of the following three options.

- You may double each processor's cache to 2 MB.
- You may increase memory bandwidth by 50%
- You may add a 4-wide SIMD unit to the core so that each core can perform 4 floating point operations per clock.

If each of these options has the same cost, given the code you produced in part B (and what you learned from assignment 1), which option do you recommend to your boss? Why?

Solution: If you provided the most common answer to part B (parallelize the computation of `total`*), then the correct answer is B, take the extra memory bandwidth. The 32-core processor would be bandwidth bound since it would require about* $32 \times 2 \times 4 = 256$ *GB/sec of bandwidth, and the previous memory system could only provide 100 GB/sec. Thus, adding more compute capability with SIMD instructions would not be helpful.*

However, if students also included the loop-reordering optimization in part B, then the correct answer to the problem is to take the SIMD unit. The loop reordering increases arithmetic intensity sufficiently that even the 32-core version of the problem would not be bandwidth bound. That's why it's a great optimization!

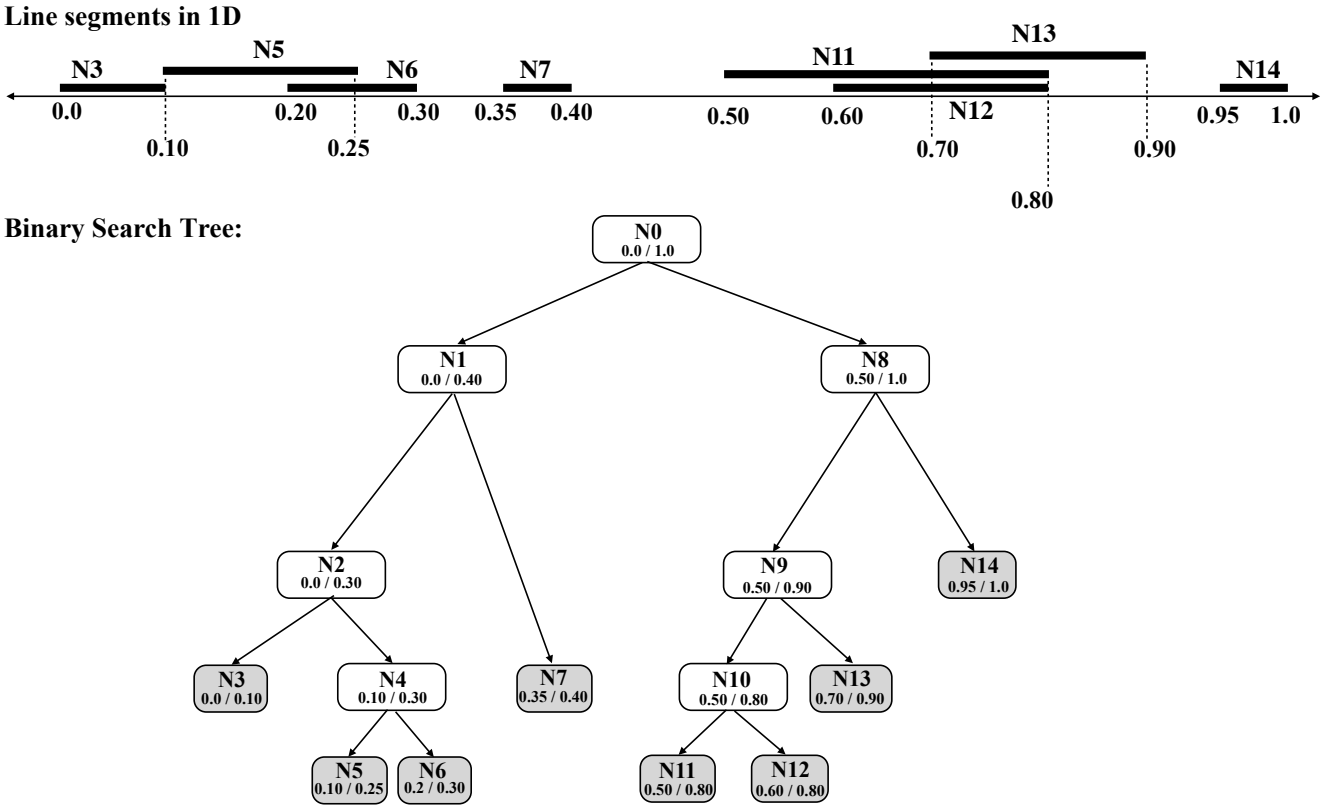
A 2 MB data cache would not help in this problem. The working set for the program is over 400 MB, so a 2MB cache incurs the same number of misses as the original 1 MB cache.

Problem 4: SPMD Tree Search

NOTE: This question is tricky. If you can answer this question you really understand SIMD execution!

The figure below shows a collection of line segments in 1D. It also shows a binary tree data structure organizing the segments into a hierarchy. Leaves of the tree correspond to the line segments. Each interior tree node represents a spatial extent that bounds all its child segments. Notice that sibling leaves can (and do) overlap. Using this data structure, it is possible to answer the question "what is the largest segment that contains a specified point" without testing the point against all segments in the scene.

For example, the answer for point $p = 0.15$ is segment 5 (in node N5). The answer for the point $p = 0.75$ is segment 11 in node N11.




```
struct Node {
   float min, max;     // if leaf: start/end of segment, else: bounds on all child segments.
   bool leaf;          // true if nodes is a leaf node
   int segment_id;     // segment id if this is a leaf
   Node* left, *right; // child tree nodes
};
```

On the following two pages, we provide you two ISPC functions, `find_segment_1` and `find_segment_2` that both compute the same thing: they use the tree structure above to find the id of the largest line segment that contains a given query point.

```
struct Node {
   float min, max;     // if leaf: start/end of segment, else: bounds on all child segments.
   bool leaf;          // true if nodes is a leaf node
   int segment_id;     // segment id if this is a leaf
   Node* left, *right; // child tree nodes
};

// -- computes segment id of the largest segment containing points[programIndex]
// -- root_node is the root of the search tree
// -- each program instance processes one query point
export void find_segment_1(uniform float* points, uniform int* results, uniform Node* root_node) {

  Stack<Node*> stack;
  Node* node;
  float max_extent = 0.0;

  // p is point this program instance is searching for
  float p = points[programIndex];
  results[programIndex] = NO_SEGMENT;

  stack.push(root_node);

  while(!stack.size() == 0) {
    node = stack.pop();

    while (!node->leaf) {
      // [I-test]: test to see if point is contained within this interior node
      if (p >= node->min && p <= node->max) {
        // [I-hit]: p is within interior node... continue to child nodes
        push(node->right);
        node = node->left;
      } else {
        // [I-miss]: point not contained within node, pop the stack
        if (stack.size() == 0)
          return;
        else
          node = stack.pop();
      }
    }

    // [S-test]: test if point is within segment, and segment is largest seen so far
    if (p >= node->min && p <= node->max && (node->max - node->min) > max_extent) {
       // [S-hit]: mark this segment as ‘‘best-so-far’’
       results[programIndex] = node->segment_id;
       max_extent = node->max - node->min;
    }
  }
}
```

```
export void find_segment_2(uniform float* points, uniform int* results, uniform Node* root_node) {

  Stack<Node*> stack;
  Node* node;
  float max_extent = 0.0;

  // p is point this program instance is sarch for
  float p = points[programIndex];

  results[programIndex] = NO_SEGMENT;

  stack.push(root_node);

  while(!stack.size() == 0) {
    node = stack.pop();

    if (!node->leaf) {
       // [I-test]: test to see if point is contained within interior node
       if (p >= node->min && p <= node->max) {
          // [I-hit]: p is within interior node... continue to child nodes
          push(node->right);
          push(node->left);
       }
    } else {
       // [S-test]: test if point is within segment, and segment is largest seen so far
       if (p >= node->min && p <= node->max && (node->max - node->min) > max_extent) {
         // [S-hit]: mark this segment as ''best-so-far''
         results[programIndex] = node->segment_id;
         max_extent = node->max - node->min;
       }
    }
  }
}
```

Begin by studying `find_segment_1`.

Given the input $p = 0.1$, the a single program instance will execute the following sequence of steps: (I-test,N0), (I-hit,N0), (I-test, N1), (I-hit, N1), (I-test, N2), (I-hit, N2) (S-test,N3), (S-hit, N3), (I-test, N4), (I-hit, N4), (S-test, N5), (S-hit, N5), (S-test, N6), (S-test,N7), (I-test, N8), (I-miss, N8). Where each of the above "steps" represents reaching a basic block in the code (see comments):

- (I-test, Nx) represents a point-interior node test against node `x`.
- (I-hit, Nx) represents logic of traversing to the child nodes of node `x` when p is determined to be contained in `x`.
- (I-miss, Nx) represents logic of traversing to sibling/ancestor nodes when the point is not contained within node `x`.
- (S-test, Nx) represents a point-segment (left node) test against the segment represented by node `x`.
- (S-hit, Nx) represents the basic block where a new largest node is found `x`.

The question is on the next page...

A. Confirm you understand the above, then consider the behavior of a **gang of 4 program instances** executing the above two ISPC functions `find_segment_1` and `find_segment_2`. For example, you may wish to consider execution on the following array:

```
points = {0.15, 0.35, 0.75, 0.95}
```

Describe the difference between the traversal approach used in `find_segment_1` and `find_segment_2` in the context of SIMD execution. Your description might want to specifically point out conditions when `find_segment_1` suffers from divergence. (Hint 1: you may want to make a table of four columns, each row is a step by the entire gang and each column shows each program instance's execution. Hint 2: It may help to consider which solution is better in the case of large, heavily unbalanced trees.)

Solution: The main difference between `find_segment_1` and `find_segment_2` is in how they handle the fact that different program instances can reach leaf nodes at different points in time. In `find_segment_1` all program instances in a gang must wait for all other program instances to reach their leaf nodes in order to proceed to the S-test. In `find_segment_2`, program instances are allowed to proceed to the S-test at different times and will do the S-test while the others wait. Because of these differences, `find_segment_2` will perform better on large unbalanced trees. With `find_segment_2`, there will be less divergence than if we had used `find_segment_1` where we will often have to hold up several program instances because at least one program instance in a gang still has not reached a leaf node.

B. Consider a slight change to the code where as soon as a best-so-far line segment is found (inside [S-hit]) the code makes a call to a **very, very expensive function**. Which solution might be preferred in this case? Why?

Solution: Now `find_segment_1` is preferable because the most significant code block now is where the S-test is performed. The amount of work done in the S-test now significantly trumps the amount of work done to get to a leaf node. Therefore, we would rather hold up all program instances until they've all reached their leaf nodes, and then perform the really expensive S-test together to maximize utilization. If we had used `find_segment_2` instead, it would often be the case that some program instances will have reached their leaf nodes while others have not, causing all other program instances to be held up for a really long time while one or a few program instances in the gang perform their S-tests.