Stanford CS149: Parallel Computing
Written Assignment 1

Multi-Core Architecture

Problem 1:

A. Consider a multi-core processor that runs at 2 GHz and has 4 cores. Each core can perform up to one 8-wide SIMD vector instruction per clock and supports hardware multi-threading with 4 hardware execution context per core. What is the maximum throughput of the processor in units of scalar floating point operations per second? Please show your calculations.

B. Consider the processor from part A running a program that perfectly parallelizes onto many cores, makes perfect usage of SIMD vector processing, and has an arithmetic intensity of 4. (4 scalar floating ops per byte of data transferred from memory.) If we assume the processor has a memory system providing 64 GB/sec of bandwidth, is the program compute-bound or bandwidth bound on this processor? You may assume for math simplicity that 1 GB/sec is a one billion bytes per second (10^9). Please show your work.

C. Consider a cache that contains 32 KB of data, has a cache line size of 4 bytes, is fully associative (meaning any cache line can go anywhere in the cache), and uses an LRU (least recently used—the line evicted is the line that was last accessed the longest time ago) replacement policy. Please describe why the following code will take a cache miss on every data access to the array A.

```
const int SIZE = 1024 * 64;
float A[SIZE];
float sum = 0.0;
for (int reps=0; reps<32; reps++)
  for (int i=0, i<SIZE; i++)
    sum += A[i];
```

Dependency Graphs, ILP, and Superscalar/Multi-Threaded Execution

Problem 2:

Consider the following sequence of scalar instructions running within a single thread.

```
1.  LD   R0 <- mem[R4]    // load memory address given by R4 into R0
2.  ADD  R1 <- R0, R0     // R1 = R0 + R0
3.  ADD  R2 <- R0, R0     // R2 = R0 + R0
4.  ADD  R3 <- R0, R0     // R3 = R0 + R0
5.  MUL  R1 <- R1, R1
6.  MUL  R2 <- R2, R2
7.  MUL  R3 <- R3, R3
8.  ADD  R1 <- R1, R2
9.  ADD  R1 <- R1, R3
10. ST   mem[R5] <- R1    // store R1 into memory at address given by R5
```

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

B. What is the maximum amount of instruction level parallelism (ILP) present in the program. Keep in mind that ILP is entirely a property of the program itself, not the machine it is run on.

C. Consider running this instruction stream on a single core, single-threaded processor that has superscalar execution capability to perform up to two instructions per clock, **PROVIDED THAT EXACTLY ONE INSTRUCTION IS A MUL**. In other words, the processor has two execution units (ALUs): one can execute add/load/store instructions and the other can execute only multiplications. Please assume all instructions (including loads/stores) individually complete in one cycle. You cannot modify the instructions in the program. What is the minimum number of cycles needed to execute this program? (Hint: think carefully about what instructions are allowed to run concurrently. The processing can run two instructions at the same time if they are independent and exactly one is a multiply.)

D. Now assume that the code above is changed so that it is: (1) running in a loop, where instructions 1-10 make up the body of the loop, and in each iteration of the loop the addresses stored in R4 and R5 are different. (2) Loop iterations are perfectly parallelized across `std::threads`. We are running this code on a **single core, multi-threaded processor that can process one instruction per clock** (arithmetic instructions, LDs, STs). However, from the moment a processor begins to execute a load instruction, there is a latency of 25 cycles before the data can be used by a dependent instruction. To be clear: if a core issues a LD in clock c, the LD occupies the core in clock c, but then the core cannot run an instruction that uses the value of the load until clock $c + 25$.

How many threads must be interleaved for the core to run at 100% efficiency? Please justify your answer.

The patterns are pretty, but the SIMD efficiency may not be

Problem 3:

Consider the following ISPC code that processes a 16×16 input image (`input`) containing white, gray, or black pixels. It produces a 16×16 output image (`output`).

```
const int IMAGE_SIZE = 16;
void myfunction(uniform float* input, uniform float* output) {

  for (uniform int row=0; row<IMAGE_SIZE; row++) {
    for (uniform int col=0; col<IMAGE_SIZE; col+=programCount) {

      int idx = row*IMAGE_SIZE + col + programIndex;
      float val = input[idx];    // load four bytes
      float result;

      if (!isWhite(val)) {
        if (isGray(val)) {
          result = foo1(val);    // 10 cycles of arithmetic
        } else {
          result = foo2(val);    // 10 cycles of arithmetic
        }
      } else {
          result = foo3(val);    // 10 cycles of arithmetic
      }
      output[idx] = result;      // store four bytes
    }
  }
}
```

The questions are on the next page...

A. Consider running the code on the image below. **Note the full image is 16×16 pixels... the entire image is not shown in the figure but you can assume the pattern repeats as indicated, with only the diagonal being colored.**

Assume that the only arithmetic cycles we want you to count are the arithmetic instructions labeled in the code. (All conditionals and load/stores are "free".) What is the overall SIMD efficiency (50%, 100%, etc.) achieved when running the code on a 1 GHz single-core CPU with a **SIMD-width (and corresponding ISPC gang size) of 4**. In other words, the core can perform one 4-wide SIMD operation per clock. (Hint: start by computing the number of cycles needed to perform one row worth of the computation.)
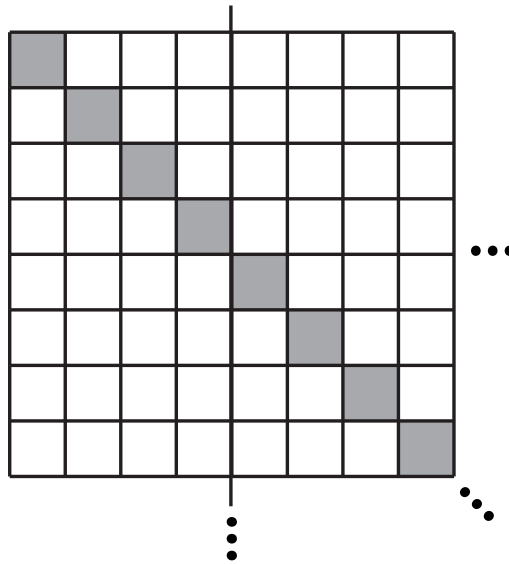
B. What is the SIMD efficiency obtained when running the same code, using the same processor, but on the image below? Again, please keep in mind the image is 16×16 in size (e.g, the pattern of columns below just extends downward another 8 rows).

$\vdots$
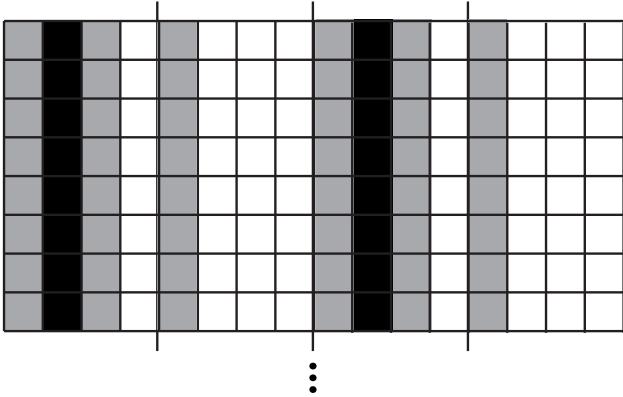
C. Now imagine that you are given a the following function:

```
transpose(uniform float* input, uniform float* output)
```

This function transposes the 16×16 input image and stores the result in a 16×16 output. (Recall that a transpose is `output[i][j] = input[j][i]`.) Write down pseudocode for how you would use calls to `myfunction` and `transpose` to produce a computation that **eliminates all execution divergence** when running `myfunction` **on the image from part B**. Yes, you can allocate temp buffers as needed.

Fill in pseudocode in `myNewFunction` below. Your solution should compute the same output image `output` as `myfunction(input, output)`.

void myNewFunction(float* input, float* output)

D. Imagine that the code is run the same processor, but now with a memory system with a memory bandwidth of **8 bytes per clock**. Please assume that the `transpose` operation is **bandwidth bound** and the cost of the operation is equal to the time needed to transfer required data to and from memory. You should also assume that `myfunction` is **compute bound** and that the required time is a function of the number of cycles of arithmetic performed.

When running on the input image from part B Is the solution you proposed in part C faster or slower than the single call to `myfunction` used in part B? How much faster is it (in clocks)?

Please keep in mind that the processor:

- Can execute one four-wide SIMD operation per clock
- Is attached to a memory bus that can transfer 8 bytes of data per clock
- The input/output images are 16×16 elements in size.
- Each image element (a float) is 4 bytes

Hint: compute the amount of time needed to transpose the data, and then compute the amount of time (in clocks) needed to perform `myfunction` on this modified input.

PRACTICE PROBLEM 1:

A. 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?

B. 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)?

C. 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?

D. 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)?

PRACTICE PROBLEM 2:

A. 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!)

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

C. 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;
}
```

PRACTICE PROBLEM 3:

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. Please draw the dependency graph for the instruction sequence.

B. 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?

C. 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.)

D. 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.
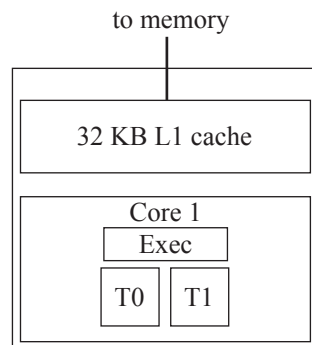
PRACTICE PROBLEM 4:

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.

to memory

```
+-------------------------------------------+
|   +-----------------------------------+   |
|   |         32 KB L1 cache            |   |
|   +-----------------------------------+   |
|                                           |
|   +-----------------------------------+   |
|   |            Core 1                 |   |
|   |   +---------------------------+   |   |
|   |   |           Exec            |   |   |
|   |   +---------------------------+   |   |
|   |   +-------+   +-------+           |   |
|   |   |  T0   |   |  T1   |           |   |
|   |   +-------+   +-------+           |   |
|   +-----------------------------------+   |
+-------------------------------------------+
```

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.

T0

T1

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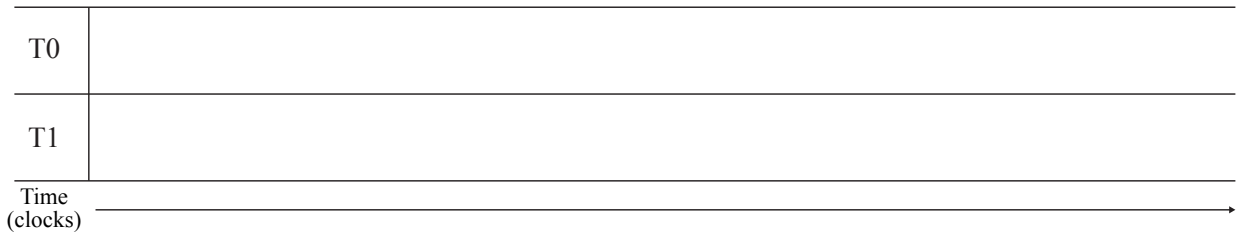
B. Consider the same setup as the previous problem. How many hardware threads would the CPU core need in order for the machine to maintain peak throughput (100% utilization) on this workload?

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.

T0	

T1	

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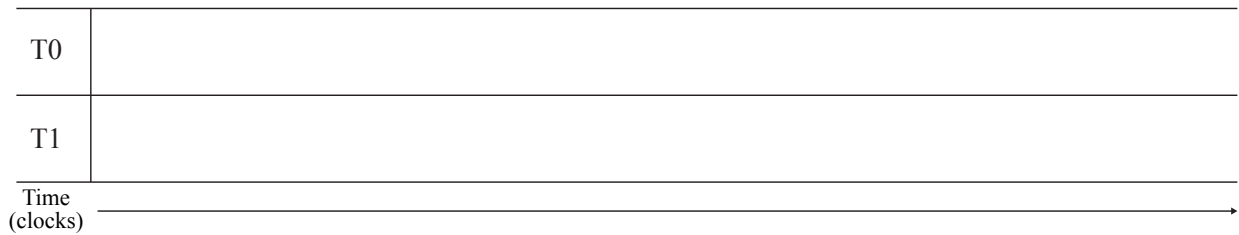
E. **Now consider the case where the L1 cache size is changed to 384 KB.** Assuming you cannot change the implementation of TASK_A from Part A, would you choose to use a worker thread pool of one or two threads? Why does this improve performance and how much higher throughput does your solution achieve?

PRACTICE PROBLEM 5:

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

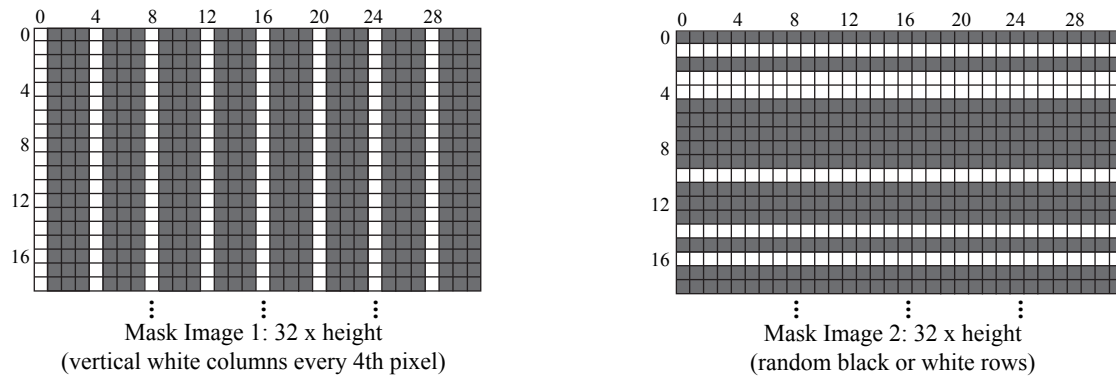(C) 4 GHz *single core* CPU capable of performing one floating point addition per clock (no parallelism)

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.

PRACTICE PROBLEM 6:

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.

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× 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.*)

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?

PRACTICE PROBLEM 7:

NOTE: This question is tricky. If you can answer this question you really understand how the SPMD programming model maps to 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.

Line segments in 1D
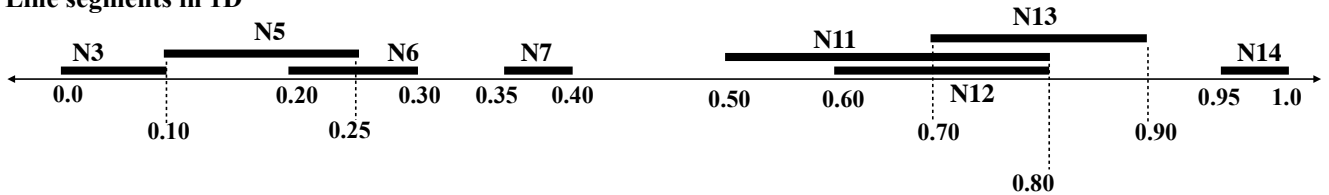
Binary Search Tree:
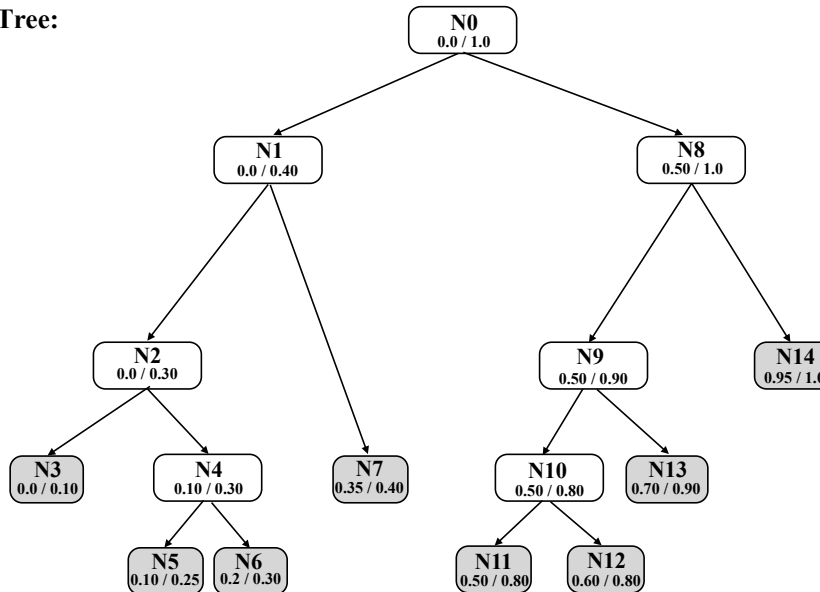

```
struct Node {
    float min, max;    // if leaf: start/end of segment, else: bounds on all child segments.
    bool leaf;         // true if nodes is a leaf node
    int segment_id;    // segment id if this is a leaf
    Node* left, *right; // child tree nodes
};
```

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.)

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?