Parallel Computer Architecture and Programming
Written Assignment 2

40 points total + 6 pts extra credit. Due Monday, July 10 at the start of class.

Problem 3: Running a CUDA Program on a GPU (12 pts)

The TAs decide to start a company that makes GPUs. They spend their summer creating a new GPU, which they call a PKPU (Prof. Kayvon Processing Unit). The processor runs CUDA programs exactly the same manner as the NVIDIA GPUs discussed in class, but it has the following characteristics:

- The processor has eight cores running at 1 GHz.
- Each core provides execution contexts for up to 128 CUDA threads. (Like the GPUs discussed in class, once a CUDA thread is assigned to an execution context, the processor runs the thread to completion before assigning a new CUDA thread to the context.)
- The cores use SIMD execution, running 16 consecutively numbered CUDA threads together using the same instruction stream (the PKPU implements 16-wide "warps", and therefore it has execution contexts for 8 warps per core).
- The cores will fetch/decode one single-precision arithmetic instruction (add, multiply, etc.) per clock. Keep in mind this instruction is executed on an entire warp in that clock.
- All CUDA thread blocks on a single core cannot exceed 16 KB of shared memory storage.

A. (2 pts) When running at peak utilization. What is the processor's **maximum throughput** of single-precision **math operations**? (In your answer, please consider one multiply of two single-precision numbers as one "operation".)

Consider a CUDA kernel launch that executes the following CUDA kernel on the processor. In this program each CUDA thread computes one element of the results array `Y` using 1000 elements from the input array `X` as input. **Assume the program is compiled using a thread-block size of 128 threads.** and that enough thread blocks are created so there is exactly one thread per output array element.

```
__global__ void foo(float* X, float* Y) {

   // get array index from CUDA block/thread id
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   int input_idx = 1000 * idx;

   float result = 0.f;

   for (int i=0; i<1000; i++) {                 // count as 0 arith cycles (ignore arithmetic)

      float val = X[input_idx+i];               // memory load (ignore arithmetic here)

      if (((int)val / 1000) % 2 == 0)           // count as 2 arithmetic cycles
         result += doA(val);                    // count as 7 arithmetic cycles
      else
         result += doB(val);                    // count as 7 arithmetic cycles
   }

   Y[idx] = result;                             // memory store
}
```

B. (2 pts) The TAs hook the PKPU up to a memory system that provides **32 GB/sec of bandwidth** and has a **memory request latency of 100 cycles**. They run the code on a 128,000-element input array initialized as `X[] = {1.f, 2.f, 3.f, 4.f, 5.f, 6.f, ...}` *The output array `Y[]` is only 128 elements*. The TA's observe that the program *does not* realize peak performance (100% efficiency) on the PKPU. Yang Gu is sad. :-(What is the primary reason for the low performance? *Hint, consider the amount of work performed by the program.*

(Please assume the input and output arrays are resident in GPU memory at the time of the kernel launch. Transfer between host and GPU memory is not relevant in any part of this problem.)

C. (2 pts) Yong He says, "Yan, let me take a look", and runs the same program on bigger output arrays of size 128×1024 elements and 128×1024×1024 elements. Does he observe significantly different *processing throughput* from the PKPU on the two workloads? Why or why not? (Please assume both the small and large workloads fit comfortably in GPU memory.)

D. (2 pts) Prof. Kayvon looks at Yong and Yan's experiments on the largest dataset and says "Oh no, this program is still not achieving 100% utilization of the processor's execution units." What is the problem limiting performance? What percentage of maximimum PKPU throughput does Prof. Kayvon observe? You will need to thnk very carefully about how this CUDA code runs on the PKPU cores. Remember:

- The processor runs a instruction on an entire 16-wide warp worth of CUDA threads in a single cycle.
- The memory system provides 32 GB/sec of bandwidth with a request latency of 100 cycles. (When a warp issues a load instruction, the data for all threads will be ready in 100 cycles.)

E. (2 pts) After some intense discussion, the TAs hook a new memory system up to the PKPU that provides **64 GB/sec of bandwidth** (still with a request latency of 100 cycles). Yong He **rewrites the program to interleave elements of the input array so that they are stored in the following order:**

```
    0...  999
 2000... 2999
 4000... 4999
     ...
30000...30999
 1000... 1999
 3000... 3999
 5000... 5999
     ...
31000...31999
     ...
32000...32999
34000...34999
     ...
```

Yong and Yan congratulate themselves on a job well done and head out to celebrate and eat watermelon. While at dinner, they get a call from Xu Ji and Ping Xu who say, "Why are you celebrating? We are here trying to figure out why Yong He's program still doesn't get peak performance." What is the problem that is limiting performance, and what percentage of peak do Xu Ji and Ping Xu observe? *(Hint: this question is tricky on purpose! It's one of the problems we've talked about in class: workload imbalance? SIMD divergence? memory bandwidth? memory latency?)*

F. (2 pts) When Yong He and Yan Gu return to the office, Xu Ji and Ping Xu are nowhere to be found. Yong and Yan find a note saying "Problem solved! We made a small tweak to the hardware design of the PKPU and now Yong's program runs at 100% utilization now." What was the tweak? (Note: we are looking for a change to the capabilities of the PKPU's cores. You cannot change the design of the memory system.)

Problem 2: Tsinghua Math Library (6 pts)

Your boss asks you to buy a new computer that will be the best possible machine to run the following program, which as been structured into simple data-parallel math library (`Tsinghua_math`), and a main application making calls the library. The library functions are self-explanatory, but the implementation of the `tsinghua_math_add` function is given.

```
const int N = 10000000;   // very large

void tsinghua_math_add(float* A, float* B, float* output) {
#pragma omp parallel for
  for (int i=0; i<N; i++)
    output[i] = A[i]+B[i];
}
void tsinghua_math_sub(float* A, float* B, float* output) { ... }
void tsinghua_math_mul(float* A, float* B, float* output) { ... }

////////////////////////////////////////////////////////////////

float* A, *B, *C, *tmp1, *tmp2, *result;  // assume arrays are allocated and initialized

tsinghua_math_add(A, B, tmp1);
tsinghua_math_mul(tmp1, C, tmp2);
tsinghua_math_mul(tmp2, A, tmp1);
tsinghua_math_add(A, tmp1, tmp2);
tsinghua_math_mul(B, tmp2, tmp1);
tsinghua_math_sub(C, tmp1, result);
```

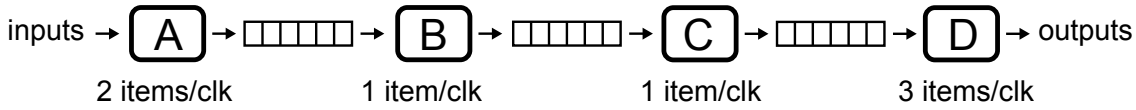
You have two computers to choose from, of equal price. (Assume that both machines have the same 1MB cache and 0 memory latency.)

1. Computer A: Four cores 1 GHz, 8-wide SIMD, 100 GB/sec bandwidth
2. Computer B: Four cores 1 Ghz, 4-wide SIMD, 150 GB/sec bandwidth

Assuming that YOU ARE ALLOWED TO REWRITE the program however you wish (provided that it computes exactly the same answer—You can parallelize across cores, vectorize, reorder loops, etc. but you are not permitted to change the math operations to turn adds into multiplies, eliminate common subexpressions etc.), which machine do you choose? Why? (If you decide to change the program please give a description of your changes. What is parallelized, vectorized, what does the loop structure look like, etc.)

Problem 3: Understanding Pipelining (6 pts)

Consider the four-stage pipeline below. Each stage in the pipeline receives elements from a 6-element input queue, and can process elements from its input queue at the rates shown in the figure. The behavior of each stage generates one output element for each input element. **A stage stalls (does no work in a clock) if there is no room in the output queue to place a result.** You can assume that inputs arrive at stage A infinitely fast, so stage A will always process two items/clk whenever it can.



For example, consider the following behavior of the pipeline:

- t=0: Stage A processes two elements from its input queue, emits two elements to the Stage B input queue (size=2)
- t=1: Stage B processes one element from its input queue and emits one element to the Stage C input queue (size=1). Stage A processes two new elements from its input queue, emits two elements to the Stage B input queue (size=??).
- t=2: Stage C processes one element from its input queue, emits one element to its output, B processes one elements from it's input queue, A processes two elements from it's input queue, ...

What is the throughput of the pipeline in terms of completed items/clock? What is the start-to-end latency of the entire pipeline? **(Be careful: In both cases, make sure you give answers for the steady-state behavior of the pipeline—including time intermediate data is stored in queues)—not its initial startup.)**

Problem 4: Miscellaneous Short Problems (6 pts + 2 pts EXTRA CREDIT)

A. (2 pts) On your first day of work at NVIDIA you propose adding a new concept to CUDA: `global_syncthreads()`. The semantics of this function is that it is a **global barrier across all threads in all thread blocks** created as a result of a single CUDA thread launch from the host. Your boss looks at you and says, "I love your ambition, but what you proposed requires major changes to how we currently schedule CUDA thread blocks onto a GPU." Describe the most significant change to how CUDA schedules work that must occur to correctly implement the `global_syncthreads` primitive. (Hint: consider running a program that creates a few thousand thread blocks, much like you did in Assignment 2.)

B. (2 pts) What is the depth of the following piece of code?

```
void f(int n) {
  if (n<10)
    return;
  f(99 * n / 100);
  f(n / 2);
}
```

C. (2 pts) What is the **depth** of the following code?

```
void f(int n) {
  if (n<10)
    return;
  f(sqrt(n));
}
```

D. (2 pts EXTRA CREDIT) In the analyzing parallel program performance lecture we discussed the following matrix divide-and-conquer algorithm multiplication algorithm. (It is also called "blocked matrix multiplication" because it computes the output C in blocks.) Please assume all matrices are square $n \times n$ matrices. the the pseudocode below, C_{00} is the top-left submatrix of C, C_{01} is the top-right submatrix, etc.

```
Function matmul(A, B) {
  if A is small then
    BaseCase
  else
    In Parallel
      C11 = matmul(A11, B11) + matmul(A12, B21)
      C12 = matmul(A11, B12) + matmul(A12, B22)
      C21 = matmul(A21, B11) + matmul(A22, B21)
      C22 = matmul(A21, B12) + matmul(A22, B22)

  return C
}
```

Show that the cache complexity of the algorithm is $O(n^3/(B\sqrt{M}))$, using the external-memory model. Recall that M is the size of the cache (in units of matrix elements), and B is the size of a block of data in the case (in units of matrix elements). *(Note: this complexity turns out to be optimal for matrix multiplication. A much harder challenge is to show that it is optimal.)*

Problem 5: Parallel Histogram (10 pts)

A sequential algorithm for generating a histogram from the values in a large input array `input` is given below. For each element of the input array, the code uses the function `bin_func` to compute a "bin" the element belongs to (`bin_func` always returns an integer between 0 and `NUM_BINS-1`) and increments the count of elements in that bin.

```
int bin_func(float value);      // external function declaration
float input[N];                 // assume input is initialized and N is a very large

int histogram_bins[NUM_BINS];  // assume bins are initialized to 0

for (int i=0; i<N; i++) {
   histogram_bins[bin_func(input[i])]++;
}
```

You are given a massively parallel machine with N processors (yes, one per input element) and asked by a colleague to produce an efficient parallel histogram routine. To help you out, your colleague hands you a library with a highly optimized parallel sort routine.

```
void sort(int count, int* input, int* output);
```

The library also has the ability to execute a bulk launch of N independent invocations of an application-provided function using the following CUDA-like syntax:

```
my_function<<<N>>>(arg1, arg2, arg3...);
```

For example the following code (assuming `current_id` is a built-in id for the current function invocation) would output:

```
void foo(int* x) {
   printf("Instance %d : %d\n", current_id, x[current_id]);
}

int A[] = {10,20,30}
foo<<<3>>>(A);

"Instance 0 : 10"
"Instance 1 : 20"
"Instance 2 : 30"
```

(question continued on next page)

Using only `sort`, `bin_func` and bulk launch of any function you wish to create, implement a data-parallel version of histogram generation that makes good use of N processors. You may assume that the variable `current_id` is in scope in any function invocation resulting from a bulk launch and provides the number of the current invocation.

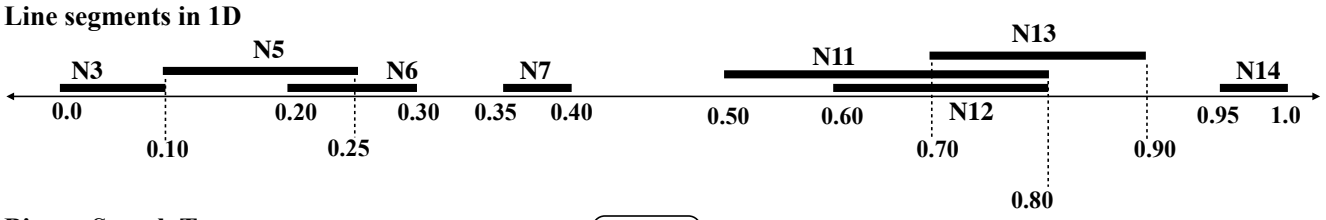
```
// External function declarations.  Your solution may or may not use all these functions.
void  sort(int count, int* input, int* output);
int   bin_func(float value);

// input: array of numbers, assume input is initialized
float input[N];

// output: assume all bins are initialized to 0
int   histogram_bins[NUM_BINS];
```

Problem 6: SIMD Tree Search (4 pts EXTRA CREDIT)

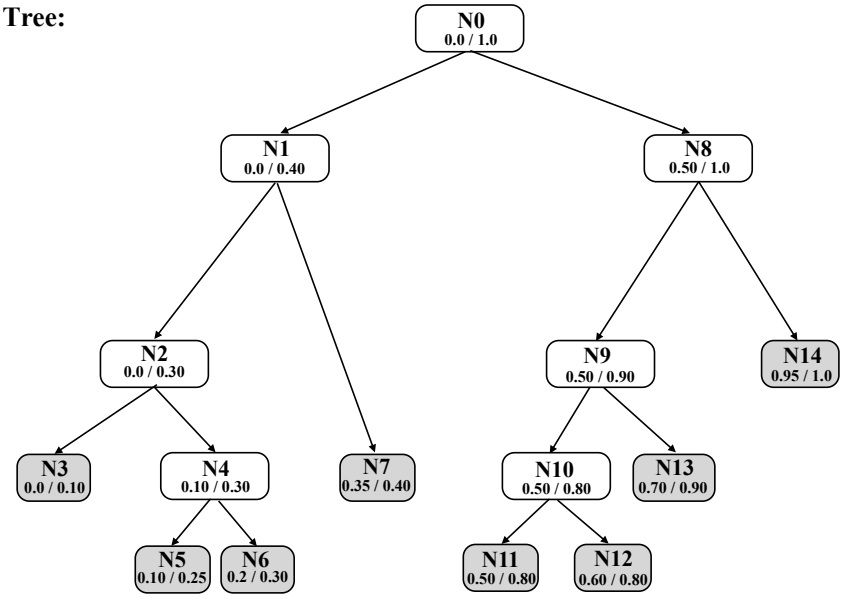
NOTE: This question is tricky. We recommend you attempt it last since there is a LOT OF READING TO DO. If you can answer this question you really understand SIMD execution!

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.



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 CUDA 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[threadId]
// -- root_node is the root of the search tree
// -- each CUDA thread processes one query point
__global__ void find_segment_1(float* points, int* results, Node* root_node) {

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

  // p is point this CUDA thread is searching for
  int threadId = threadIdx.x + blockIdx.x * blockDim.x;
  float p = points[threadId];

  results[threadId] = 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-inside]: mark this segment as ‘‘best-so-far’’
       results[threadId] = node->segment_id;
       max_extent = node->max - node->min;
    }
  }
}
```

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

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

  // p is point this CUDA thread is searching for
  int threadId = threadIdx.x + blockIdx.x * blockDim.x;
  float p = points[threadId];

  results[threadId] = 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-inside]: 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-inside]: mark this segment as ''best-so-far''
         results[threadId] = node->segment_id;
         max_extent = node->max - node->min;
       }
    }
  }
}
```

Begin by studying `find_segment_1`.

Given the input $p = 0.1$, the a single CUDA thread 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. (5 pts) Confirm you understand the above, then consider the behavior of a **4-wide warp** executing the above two CUDA 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 warp and each column shows each thread's execution. Hint 2: It may help to consider which solution is better in the case of large, heavily unbalanced trees.)

B. (5 pts) 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?