

## Parallel Computer Architecture and Programming

### Written Assignment 2 SOLUTIONS

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”.)

*Solution:  $8 \text{ cores} \times 16 \text{ operations/clock} \times 1 \text{ Ghz} = 128 \text{ Giga-ops/sec}$  (128 operations per clock). In other words, there are 128 single-precision (SIMD) ALUs running at 1 Ghz.*

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  $X[] = \{1.f, 2.f, 3.f, 4.f, 5.f, 6.f, \dots\}$  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.)

*Solution: This workload is too small for the PKPU. This program creates a single thread block of work (128 threads), which will be run on only one of the eight PKPU cores. As a result, it runs at most at 1/8 of the PKPU's peak throughput.*

- C. (2 pts) Yong He says, “Yan, let me take a look”, and runs the same program on bigger output arrays of size  $128 \times 1024$  elements and  $128 \times 1024 \times 1024$  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.)

*Solution: Both workloads generate plenty of work for the PKPU’s 8 cores: 1024 thread blocks and 1M thread blocks respectively. Therefore, while the second program takes longer to execute (it does more work), both programs execute with the same throughput.*

- 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 maximum PKPU throughput does Prof. Kayvon observe? You will need to think 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.)

*Solution: Each warp’s load requests  $16 \text{ threads} \times 4 \text{ bytes} = 64 \text{ bytes}$  of data from memory. Therefore, across the 8 cores 512 bytes are requested. However, there’s divergence in this code, so the PKPU core will have to run both the `if` and `else` branches of the body of the loop. That’s a total of 16 arithmetic cycles per loop body. Therefore, the system is requesting 512 bytes of data every 16 cycles. To keep up, the memory system must provide  $512/16 = 32 \text{ GB}$  of bandwidth, which it in fact does! **The machine runs at about 50% of peak utilization due to instruction stream divergence.** For the nitpicky, since 2 out of 16 arithmetic instructions in the inner loop are fully coherent, the actual throughput is 56% of peak  $((2/16) \times 100 + (14/16) \times 50)$ .*

- 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?*)

*Solution: In this case, even after eliminating instruction stream divergence, and adding the extra memory bandwidth to prevent the program from being bandwidth-bound after the update, **the program does not achieve peak throughput because the PKPU cannot fully hide the 100-cycle memory latency of loads.** When a warp stalls on a memory request, there are seven other warps that can be run to hide the latency of this operation. Each of those warps may issue up to  $7 + 2 = 9$  arithmetic instructions (the fully coherent loop body), so only  $7 \times 9 = 63$  of the 100 cycles of memory latency are hidden—the rest of the time the PKPU core stalls. Therefore the GPU runs at 63% of peak. (We also accepted the answers of  $8 \cdot 9 / 100 = 72\%$  and  $8 \cdot 9 / 109 = 66\%$  of peak for full credit.) YES, THIS WAS A TOUGH QUESTION.*

- 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.)

*Solution: Xu Ji and Ping Xu modified the PKPU to have more warp execution contexts, and thus more latency hiding ability. We also gave credit to answers such as adding support for prefetching.*

## 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

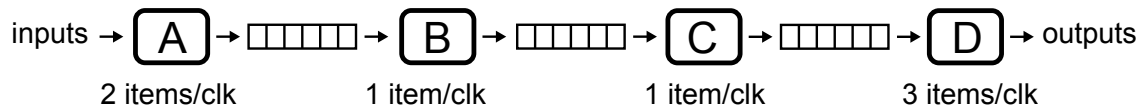
**Assuming that YOU ARE ALLOWED TO REWRITE the program however you wish (provided that it computes exactly the same answer—You can parallelize across cores, vectorize, reorder loops, etc. but you are not permitted to change the math operations to turn adds into multiplies, eliminate common subexpressions 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.)

*Solution: You should go with Computer A since it has more compute capability, and the code can be restructured to take advantage of this peak throughput. The code must be restructured to fuse the various loops. Otherwise, the code is bandwidth bound, and so computer B presents a better solution since it has higher memory bandwidth. However, once the loops are fused the arithmetic intensity of the problem increases enough for Computer A to deliver higher overall performance.*

```
// parallelize and vectorize this loop
for (int i=0; i<N; i++) {
    // three loads, one store, 6 math ops
    // arithmetic intensity = 6 ops / 16 bytes
    result[i] = C[i] - (B[i] * (A[i] + (A[i] + B[i]) * C[i] * A[i])));
}
```

### 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/clock 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.)**

*Solution: The throughput is 1 element per clock, since the throughput of the pipeline is limited by its slowest stage. Computing latency required thinking about the steady-state behavior of the queues. Since Stage A produces elements faster than Stage B can consume them, the A-to-B queue will always be full, contributing to the overall latency. Elements placed in the A-to-B queue will sit in the queue for six cycles. Elements placed in the B-to-C and C-to-D queues are always consumed the cycle after they are produced. Therefore the total latency is: 1 + 6 + 1 + 1 + 1=10 cycles.*

#### 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.)

*Solution: A major assumption of how CUDA schedules thread blocks onto GPU SM cores is that they can be "streamed" onto the cores. For example, if a program creates a large number of thread blocks, the blocks can be assigned to the cores dynamically as cores become free. The block is run to completion, and then its resources are freed for the next block. Implementing a global barrier would require an implementation to be able to support all thread blocks live on the GPU at once (since all blocks need to make progress to hit the barrier, otherwise deadlock will occur). Since CUDA programs can make large numbers of thread blocks, many more blocks than SM cores on the GPU, the CUDA implementation would have to change to time interleave thread blocks on the GPU. This would involve preempting running thread blocks much like a normal operating system preempts processes.*

- 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);  
}
```

*Solution: It's  $\lg(n)$  but with base 100/99.*



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

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

*Solution:  $\lg(\lg(n))$*

*Solution:  $\sqrt{n}$*

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.*)

*Solution: TODO*

### 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];
```

*Solution: (Full credit would be given if edge cases for empty bins weren't handled correctly in `compute_counts`.)*

```
int bin_ids[N];
int sorted_bin_ids[N];
int bin_starts[NUM_BINS]; // initialized to -1

void compute_bin(float* input, int* bin_ids) {
    bin_ids[current_id] = bin_func(input[current_id]);
}

void find_starts(int* bin_ids, int* starts) {
    if (current_id == 0 || bin_ids[current_id] != bin_ids[current_id-1])
        starts[bin_ids[current_id]] = current_id;
}

void compute_counts(int* bin_starts, int* histogram_bins, int num_items, int num_bins) {
    if (histogram_bins[current_id] == -1) {
        histogram_bins[current_id] = 0; // edge case for no items in this bin
    } else {
        // find start of next bin. Tricky edge case: this is not just
        // current_id+1 if next bin in the histogram is empty
        int next_id = current_id+1;
        while(next_id < num_bins && bin_starts[next_id] == -1)
            id++;

        if (next_id < num_bins)
            histogram_bins[current_id] = bin_starts[next_id] - bin_starts[current_id];
        else
            histogram_bins[current_id] = num_items - bin_starts[current_id];
    }
}

int main () {
    // every element computes the bin it falls in
    launch<<<N>>>compute_bin(input, bin_ids);

    // find starting point of each bin in sorted list
    sort(N, bin_ids, sorted_bin_ids);
    launch<<<N>>>find_starts(sorted_bin_ids, bin_starts);

    // compute differences in bin starts to get bin counts
    launch<<<NUM_BINS>>>compute_counts(bin_starts, histogram_bins, N);
}
```

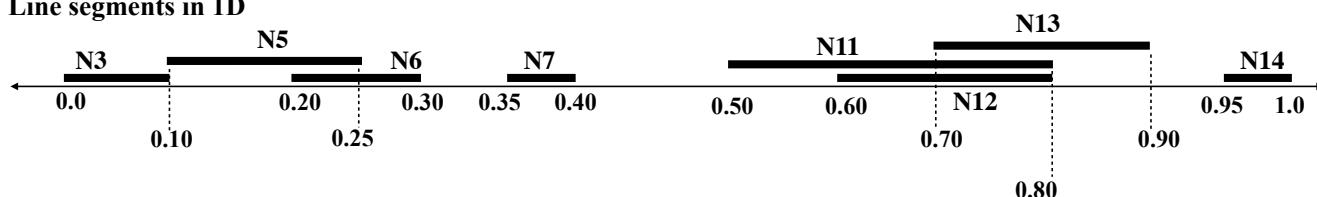
## Problem 6: SIMD Tree Search (4 pts EXTRA CREDIT)

**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!**

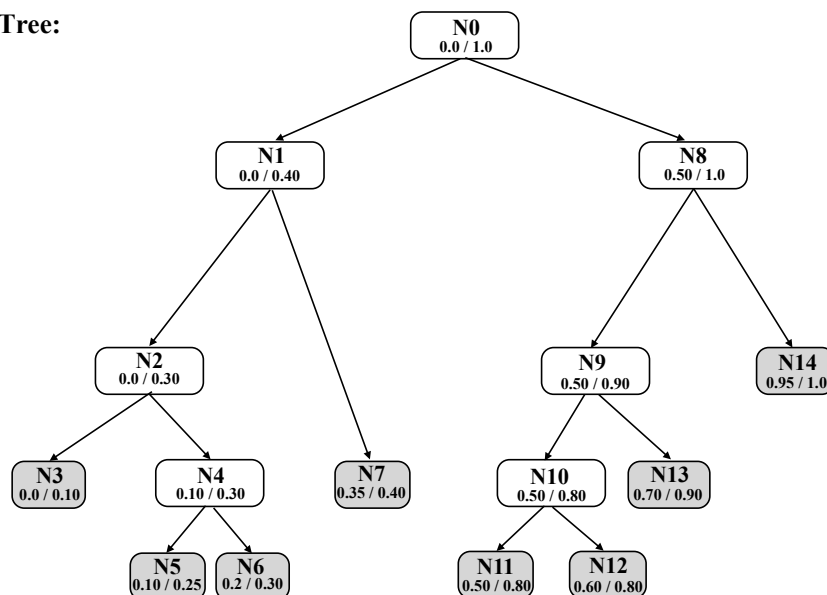
The figure below shows a collection of line segments in 1D. It also shows a binary tree data structure organizing the segments into a hierarchy. Leaves of the tree correspond to the line segments. Each interior tree node represents a spatial extent that bounds all its child segments. Notice that sibling 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.**

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 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 node 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.)

*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. (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?

*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.*