Stanford CS149: Parallel Computing
Written Assignment 2

To Fuse or Not to Fuse

Problem 1: (Graded for Correctness - 20 pts)

Consider the following C program which makes calls to helper functions `f1()`, `f2()`, and `f3()`. You can assume that these functions only perform arithmetic. Please assume that where we write `parallel_for`, we'd like you to assume that the work of the for loop is perfectly parallelized across worker threads. (e.g., You can assume that thread creation/startup occurs at the start of each `parallel_for` loop, and thread joining happens at the end of the loop, and all this work is "free".)

```
// assume N is a billion elements
// assume array a[] is initialized to contain positive values > 1.0.

float a[N], b[N], c[N], d[N], e[N];
float maxValue = 0.0;

parallel_for(int i=0; i<N; i++)    // ****** parallel loop
   b[i] = f1(a[i]);                // f1 performs 5 float ops

parallel_for(int i=0; i<N; i++)    // ****** parallel loop
   c[i] = f2(a[i], b[i]);          // f2 performs 4 float ops

for (int i=0; i<N; i++)            // ****** SEQUENTIAL loop
   maxValue = max(maxValue, c[i]); // 1 float op

parallel_for(int i=0; i<N; i++);   // ****** parallel loop
   d[i] = c[i] / maxValue;         // 1 float op

parallel_for(int i=0; i<N; i++)    // ****** parallel loop
   e[i] = f3(d[i]);                // f3 performs 4 float ops
```

A. (10 pts) Your boss tells you that they are willing to build a machine with an infinite number of cores, infinite memory bandwidth, and zero memory latency... whatever it takes to see a 20× speedup compared to the sequential version of the code (e.g., running the code on one core). Assuming that you cannot change the code, is this possible? If no, explain why not. If yes, please provide the minimum number of cores needed to meet your boss' goal.

[ ] cores are needed to achieve the desired speedup.

B. (10 pts) Now assume that you are running the code on a computer with **ten single-threaded cores**, each running at 1 GHz and capable of executing 1 floating-point instruction per clock. The machine has zero memory latency, **but a memory bandwidth of 4 GB/sec.** (To keep math easy, you can assume 1 GB is 10^9 bytes.) The machine also has a small 256 KB data cache that is shared among all cores.

Most importantly, you are allowed to rewrite the program for maximum performance. To help with this rewrite, you may assume that you have an implementation of the following method:
`atomicMax(float* a, float b)`
which atomically overwrites the value pointed to by `a` with the maximum of the current value at that address and `b`.

Please rewrite the program for maximum performance, and then also **give the total run time (in seconds) of your modified program for N=1 billion.** Please only consider floating point operations. Reads/writes to data stored in cache should be considered "free".

- Hint 1: your solution need only make use of `parallel_for` loops and calls to `atomicMax`.
- Hint 2: is the program compute or bandwidth bound?
- Hint 3: are there opportunities for fusion? dependencies that prevent fusion?
- Please consider reading/writing 1 float from memory as 4 bytes of memory traffic.

seconds

Problem 2: (Graded for Correctness - 20 pts)

A. (10 pts) Consider the following ISPC program compiled with a gang size of 8 and running on a core that can execute 8-wide SIMD instructions.

```
export void myFunc(uniform float* input, uniform float* output, int N) {

   foreach (i=0..N) {
      float x = input[i];
      if (input[i] > 0.0) {
         x *= 2.0;              // one float op
         if (input[i] < 10.0) {
            x -= 6.0;           // one float op
         } else {
            x = x * x + x;      // **two** float ops
         }
      } else {
         x *= -1.0;             // one float op
      }
      output[i] = x;
   }
}
```

If we assume that the only operations are the floating point arithmetic ops labeled in the code (all loads/stores, int operations, and comparisons are free), please describe an input that yields the worst-possible SIMD utilization for the program. What is the utilization this input creates? (Leaving your answer as a fraction is fine.)

SIMD utilization is

B. (correctness pts) **[THIS PROBLEM IS INDEPENDENT OF THE PREVIOUS ONE]** Consider the following CUDA program, where all CUDA threads in a thread block traverse a random path down a binary tree. Each thread computes the maximum element they encounter during this traversal.

```
struct Node {
   Node *left, *right;
   bool isLeaf;
   float value;
};

__global__ void myKernel(Node* n, float* results) {

  float value = VERY_SMALL;
  while (!n->isLeaf) {
    value = max(value, n->value);
    bool x = randBool(); // generate true or false at random
    if (x)
       n = n->left;
    else
       n = n->right;
  }
  // each thread writes result to a unique position in the array.
  results[threadIdx.x] = value;
}
```

Assume the kernel is launched with exactly one thread block, and the thread block size is 1024 CUDA threads. The GPU's warp size (SIMD width) is 32 CUDA threads. Recall that threads in the same warp execute can execute simultaneously if they are executing the same instruction.

If the kernel is run on a **complete binary tree** of depth 10, what is the SIMD utilization of the CUDA kernel? **Please describe why you concluded this.** Recall that a complete binary tree is a tree where all paths from the root to a leaf are the same length. (All nodes of depth 0-9 have 2 children, and all nodes at depth 10 are leaves.)

SIMD utilization is

Problem 3: (Graded for Correctness - 20 pts)

Consider the following code written in an SPMD style. Note this is not ISPC code. It's C-like code, but assume that N threads are running the code. The threads cooperate to compute a histogram for the values in an input array `data`. You can assume that data contains random numbers between 0 and 100, the histogram has 10 bins, and that `bins[i]` is supposed to contain the count of the number of elements in `data` that fall between $10 \times i$ an $10 \times (i + 1)$

```
// These variables are global variables accessible to all threads.

const int N = VERY_LARGE_NUMBER;  // assume N is a very large number
const int NUM_THREADS = 4;
int data[N];
int bins[10];  // assume initialized to 0
Lock myLock;

// This function is run in SPMD fashion by all threads

void run(int threadId) {
  int elsPerThread = N / NUM_THREADS;
  int start = threadId * elsPerThread;
  int end = start + elsPerThread;

  for (int i=start; i<end; i++) {
    int binId = data[i] / 10;
    myLock.lock();
    bins[binId]++;
    myLock.unlock();
  }
}
```

A. (10 pts) You run the program on a four core processor, and observe that it gets the correct answer, and that work is well distributed among the threads. However you don't observe a great speedup compared to a single threaded version of the code. What is a potential significant performance problem?

.

B. (10 pts) Imagine that instead of locks, you are allowed to use a single `barrier()` in the code. Please give a solution that yields good work distribution onto all 4 threads, uses no locks, and uses only a single call to `barrier()`. Your solution is allowed to allocate new global or per-thread variables. **Hint: Keep in mind that `N` is assumed to be much, much larger than the number of bins in the histogram.**

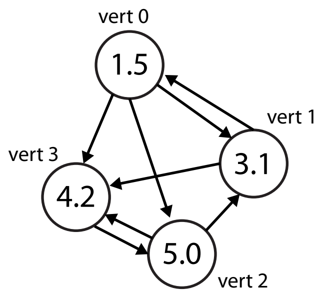
```
const int N = VERY_LARGE_NUMBER;  // assume N is a very large number
const int NUM_THREADS = 4;
int data[N];
int bins[10];  // assume initialized to 0




void run(int threadId) {


  int elsPerThread = N / NUM_THREADS;
  int start = threadId * elsPerThread;
  int end = start + elsPerThread;




  for (int i=start; i<end; i++) {

    int binId = data[i] / 10;




  }




}
```

Problem 4: (Graded on Effort Only - 20 pts)

A very efficient way to represent a directed graph is via arrays:

```
int NUM_VERTS;               // number of verts in the graph
int NUM_EDGES;               // total number of edges in the graph
float vertValues[NUM_VERTS];   // the value stored at each vertex
int edgeStarts[NUM_VERTS+1];   // edgeStarts[v] is the index in 'edges' of the first
                             // edge leaving vertex v. Note: the last (extra) element
                             // of the array is the value NUM_EDGES, you'll find it
                             // useful to avoid boundary conditions.
int edges[NUM_EDGES];          // the index of the vertex on the other side of the edge
                             // (see example below)
```

For example, the following directed graph holds a floating point value at each of its four vertices, and it features eight directed edges. It would be represented like this:



```
int NUM_VERTS = 4;
int NUM_EDGES = 8;
float vertValues = {1.5, 3.1, 5.0, 4.2};
int edgeStarts = {0, 3, 5, 7, 8};

// first 3 ints are the targets of edges from vert 0
// next 2 ints are targets of edges from vert 1
// next 2 ints are targets of edges from vert 2, etc.
int edges = {1,2,3,  0,3,  1,3,   2};
```

Given the data structure above, the following code sets the value of each vertex to the **average of its neighboring vertices**. This is a variant of the grid solver code from class, but on a graph! (It's also a common operation in the popular graph analysis algorithm PageRank!)

```
float newVertValues[NUM_VERTS];
for (int v=0; v<NUM_VERTS; v++) {  // loop over all verts
  int start = edgeStarts[v];
  int end = edgeStarts[v+1];
  float sum = 0.0;
  for (int e=start; e<end; e++) {  // loop over all edges from current vert
    sum += vertValues[edges[e]];
  }
  int numEdges = end-start;
  newVertValues[v] = sum / numEdges;
}
```

We're now going to implement this algorithm using data parallel operators. You have at your disposal the following library. We've discussed all these functions in class, but explanations are given below for your assistance.

```
// gathers from input into output according to indices (for inputs of any type T)
// N is the size of the **output** array (and the indices array)
// -- example: input=[1.0, 2.0, 3.0, 4.0], indices=[3,2,1] --> output=[4.0, 3.0, 2.0]
void gather(T* input, int* indices, T* output, int N);

// scatters from input to output according to indices (for inputs of any type T)
// N is the size of the **input** array (and the indices array)
// The values in indices are assumed to be in range for the output array.
// -- example: input=[1.0, 2.0, 3.0], indices=[4,0,1]
//         --> output=[2.0, 3.0, 0.0, 0.0, 1.0]
//             (assuming output is all zeros before the scatter)
void scatter(T* input, int* indices, T* output, int N);

// performs an **inclusive** segmented scan with the addition operator according
// to segmentStarts (for any type T)
// example: input=[1.0, 2.0, 3.0, 4.0, 5.0], segmentStarts=[1,0,1,0,0]
//      --> output=[1.0, 3.0, 3.0, 7.0, 12.0]
void segmentedScan(T* input, bool* segmentStarts, T* output, int N);

// shifts all values in the input array one element to the left (for any type T).
// decreasing the array size by one.
// N is the size of the input array. The output array is of size N-1.
// example: input=[1.0, 2.0, 3.0, 4.0], output=[2.0, 3.0, 4.0]
void shiftLeft(T* input, T* output, int N);
```

You also have the ability to map a function onto all elements of an array using the syntax:
`output = map(f,input)`. For example given the function:

```
int f(int x) { return x+1;}
```

You can add 1 to all elements of the array `input` using: `output = map(f, input)`.

A. (5 pts) Given `edgeStarts` (a length `NUM_VERTS) array` we'd like you create a length `NUM_EDGES` array `segmentStarts`, that has the value 1 for all elements that correspond to the start of a new vertex, and 0 otherwise. For example, for the graph above, `segmentStarts` should be [1,0,0,1,0,1,0,1]. Assume you can initialize arrays with useful functions like: `allOnes(myArray, N)`

B. (10 pts) Given `segmentStarts` from the prior question (you can assume you have a correct implementation) and all graph arrays, please use only data parallel library above and `map` to output a length `NUM_VERTS` array `vertSums`, where `vertSums[v]` is the sum of values of all verts adjacent to `v`. You do not need to perform division to compute the average, just the sum. Please define any functions you need for calls to `map`.

C. (5 pts) Now, given `vertSums` from part B, compute a length `NUM_VERTS` array `newVertValues` (where `newVertValues` is the same output as the C code given at the top of this problem:
`newVertValues[v] = vertSums[v]/NUM_EDGES_LEAVING_VERT_V`).

To make your code simple you can write `a = b + c` to element-wise add/subtract two arrays, and `a = b / c` to element-wise divide two arrays. **If these arrays are of different sizes, the operation will produce an output that's the size of the smaller array. e.g., adding an array of N elements to an array of N+1 elements will yield an array of N elements (the N+1'th element is ignored).**

Particle Simulation

Problem 5: (Graded on Effort Only - 20 pts)

Consider the following code that uses a simple $O(N^2)$ algorithm to compute forces due to gravitational interactions between all N particles in a particle simulation. One important detail of this algorithm is that force computation is symmetric (`gravity(i,j) = gravity(j,i)`). Therefore, iteration `i` only needs to compute interactions with particles with index `j`, where `i<j`. As a result, there are $N^2/2$ called to `gravity` rather than N^2.

In this problem, **assume the code is run on a dual-core processor, with infinite memory bandwidth.**

```
struct Particle {
  float force;  // for simplicity, assume force is represented as a single float
  Lock l;
};

Particle particles[N];

void compute_forces(int threadId) {

  // thread 0 takes first half, thread 1 takes second half
  int start = threadId * N/2;
  int end = start + N/2;

  for (int i=start; i<end; i++) {

    // only compute forces for each pair (i,j) once, then accumulate force
    // into *both* particle i and j

    for (int j=i+1; j<N; j++) {
      float force = gravity(i, j);

      lock(particles[i].l);
      particles[i] += force;
      unlock(particles[i].l);

      lock(particles[j].l);
      particles[j] += force;
      unlock(particles[j].l);
    }
  }
}
```

Question is on the next page...

A. (10 pts) Although the code makes $N^2/2$ calls to `gravity()` it takes N^2 locks. Modify the code so that the number of lock/unlock operations is reduced by $2\times$. You may not allocate additional variables or change how look iterations are mapped to the threads.

B. (10 pts) **(This question can be answered independently from part A)** Looking at the original code, there another major performance problem that does not have to do with the number of lock/unlock operations. Please describe the problem and then describe a solution. Clearly describing an implementable solution strategy is fine, you do not need to write precise pseudocode.

PRACTICE PROBLEM 1:

Imagine you are given the following functions that operate on collections: `doubleSort`, `scatter`, and `transpose`. ISPC signatures for these functions are given below.

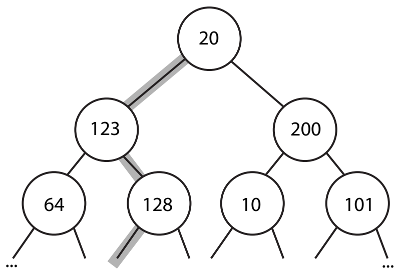
```
// sorts the input array input1, but also permutes the contents of
// the integer array input2 based on the movement of elements in input1.
// If given [2.2 3.1 1.6 3.4] and [0 1 2 3] as input1 and input2,
// then output1 and output2 would be:
    [1.6 2.2 3.1 3.4] and [2 0 1 3]
void doubleSort(uniform float* input1, uniform int* input2,
                uniform float* output1, uniform int* output2,
                uniform int N);

// scatters contents of the array ‘input‘ into the array ‘output‘
// according to the values in the array indices
// If given [1.0 5.0 6.0 2.0] and [3 2 1 0] as input and indices,
// the result in output would be:
     [2.0 6.0 5.0 1.0]
void scatter(uniform float* input, uniform int* indices,
             uniform float* output, uniform int N);

// transposes ’input’ NxM matrix into ’output’ MxN matrix.
// all matrices are stored row major (notation: NxM = HEIGHTxWIDTH)
void transpose(uniform float* input, uniform float* output,
               uniform int N, uniform int M);
```

Please use the methods above (as well as the function `myfunction()` from part A) to implement code that is functionally equivalent to `myfunction()` (e.g., it computes the same output), but will suffer from almost no SIMD divergence when executing `myfunction()`. **Your solution may or may not use all the helper functions above.** You may write pseudocode to allocate and initialize arrays (e.g.) `float A[16] = 1.0, 5.0, 3.0 ...`, but your solution should only perform substantial computation via the sequence operations above and calls to `myfunction`. **You are intentionally given a small amount of space on this page to keep answers brief.**

Running CUDA Code on a GPU

PRACTICE PROBLEM 2:

Consider a complete binary tree of depth 16 that holds one floating point number at each node as shown below. (The figure shows up to depth two.)



Now imagine a CUDA program where each CUDA thread computes the sum of results obtained by applying the functions `f()` or `g()` to all numbers on a path from the root to a leaf. The path taken through the tree is determined by the thread's id, as given in the code below. (In the figure above we highlight the path for thread id 2 which in binary is ...00000010 or left-right-left-left-left...)

```
struct Node {
   Node *left, *right;
   float value;
};

void traverse(Node* root, float* output) {
   int threadId = blockDim.x * blockIdx.x + threadIdx.x;  // compute 1D thread id
   float sum = 0.0;
   int pathBits = threadId;
   int depth = 0;
   Node* curNode = root;
   while (curNode != NULL) {

      // Consider this a single load of 12 bytes
      // Assume processor doesn't use arithmetic cycles to issue loads
      float val = curNode->value;    // 4 bytes
      Node* left = curNode->left;    // 4 bytes
      Node* right = curNode->right;  // 4 bytes

      if (depth % 2 == 0)
        sum += f(val);    // 7 arithmetic instructions
      else
        sum += g(val);    // 7 arithmetic instructions

      // ***** count the lines below as 3 arithmetic instructions
      curNode = (pathBits & 1) ? right : left;  // *** line "A" ***
      pathBits >> 1; // shift right by 1 bit
      depth++;
   }

   output[threadId] = sum;  // each thread writes its result
}
```

Questions are on the next page.

Assume that the program runs on a GPU with a SIMD width (aka CUDA warp size) of 32. Does the program suffer from low utilization due to SIMD divergence, why or why not? (For simplicity, please assume that the conditional "?" operator in line A is a single statement with no divergence.)

PRACTICE PROBLEM 3:

A. A key idea in this course is the difference between *abstraction* and *implementation*. Consider two abstractions we've studied: ISPC's `foreach` and Cilk's `cilk_spawn` construct. **Briefly describe how these two abstractions have similar semantics.** (Hint: what do the constructs declare about the associated loop iterations? Be precise!). **Then briefly describe how their implementations are quite different** (Hint: consider their mapping to execution contexts and SIMD vector instructions on modern CPUs). As a reminder, we give you two syntax examples below:

```
ISPC foreach:                        Cilk:
============================         ============================
                                     void f(int i, float* x, float* y) {
                                       x[i] = y[i]
                                     }

foreach (i = 0 ... 100) {            for (int i=0; i<100; i++) {
  x[i] = y[i];                         cilk_spawn f(i, x, y);
}                                    }
```

B. When we discussed Cilk, we emphasized how `cilk_spawn foo()` differs from a normal C function call `foo()` in that the Cilk call can run asynchronously with the caller. Notice that Cilk doesn't explicitly state that the callee function runs **in parallel with the caller**. Give one reason why the designers of Cilk intentionally designed a language that does not specify when the call will run relative to the caller?

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];
```

D. Your friend is designing a soccer playing robot for the next RoboCup competition. The software on the robot must send a torque request to the robot's motors every 3 ms in order for the robot to successfully locomote. Unfortunately in your friend's implementation, computing torques takes 20 ms when running serially on a single core of the robot's 2 GHz single-core CPU. As a result your friend's robot constantly falls over without being touched. You laugh at your friend, and call their robot "Neymar"!

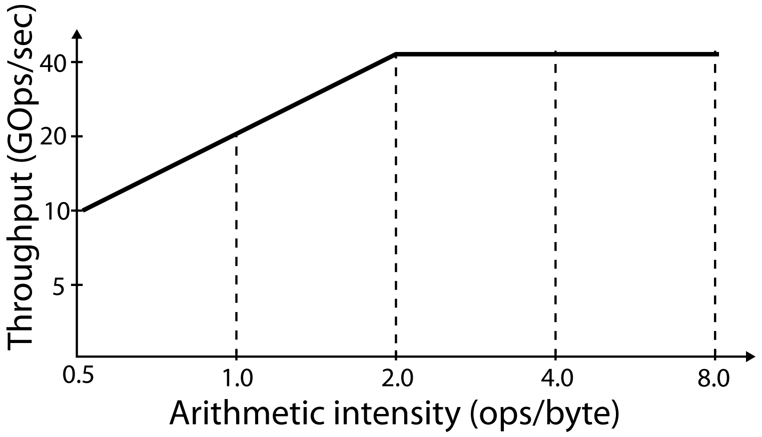
You look at your friend's code and notice that 20% of the code is inherently serial. The rest is perfectly parallelizable. You dig into your desk drawer and find two processors: Processor A is a 64-core processor that runs at 1 GHz. Processor B is a 8-core processor that runs at 4 GHz. Can you solve your friend's performance problem? Justify your answer by computing the maximum performance they can achieve.

More Practice

PRACTICE PROBLEM 4:

A. The figure below shows a "roofline" plot for a specific processor. Each point on the graph corresponds to the performance (the Y axis is ops/sec) of a different program with the given arithmetic intensity (the X-axis gives the arithmetic ops per byte read by the program. **What is the bandwidth available to the processor? Please give a number, and a 1-2 sentence explanation of your answer.**

Hint: consider why the curve slants diagonally upward in the left part of the graph. Why does the curve flatten out on the right side?



B. Your friend suspects that their program is suffering from high communication overhead, so to overlap the sending of multiple messages, they try to change their code to use asynchronous, non-blocking sends instead of synchronous, blocking sends. The result is this code (assume it is run by thread 1 in two-thread program).

```
float mydata[ARRAY_SIZE];
int dst_thread = 2;

update_data_1(mydata); // updates contents of mydata
async_send(dst_thread, mydata, sizeof(float) * ARRAY_SIZE);

update_data_2(mydata); // updates contents of mydata
async_send(dst_thread, mydata, sizeof(float) * ARRAY_SIZE);
```

Your friend runs to you to say “my program no longer gives the correct results.” What is their bug?

Another exercise on understanding the behavior of caches!

PRACTICE PROBLEM 5:

Consider the following C code run by one thread on a CPU featuring a cache with 64 byte (16 float) cache lines and a total capacity of 128 KB. Assume the cache employs a least recently used (LRU) eviction policy and it is fully associative (any cache line can be placed in any available location in the cache). **The processor does not perform any data pre-fetching.**

```
const int SIZE = 1024 * 1024 * 1024;
float input[SIZE];          // sizeof(float) * SIZE bytes
float output[SIZE];         // sizeof(float) * SIZE bytes

// assume input is appropriately initialized here

for (int i=0; i<SIZE; i++) {
  output[i] = input[i] * 2.0;
}
```

A. In the `for` loop, what fraction of loads from `input` are cache misses? Briefly explain why? **(Hint: Please keep in mind the cache line size.)**

B. Imagine the code is changed in the following way to have a nested set of `for` loops:

```
const int SIZE = 8 * 1024;
float input[SIZE];     // sizeof(float) * SIZE bytes
float output[SIZE];    // sizeof(float) * SIZE bytes

// assume input is appropriately initialized here

for (int j=0; j<10000; j++) {
  for (int i=0; i<SIZE; i++) {
    output[i] = output[i] + input[i] * 2.0;
  }
}
```

Assume the processor's cache is the same as in part A. In the "common case" where `j>0`, what fraction of accesses to `input` and `output` (both loads and stores) are cache misses? Briefly explain why?

C. Now consider the following code which works on 2D arrays:

```
const int SIZE_1 = 1024 * 1024 * 1024;
const int SIZE_2 = 1024 * 8;

float input[SIZE_1][SIZE_2];
float output[SIZE_1-1][SIZE_2];

// assume input is appropriately initialized here

for (int j=0; j<SIZE_1-1; j++) {
  for (int i=0; i<SIZE_2; i++) {
    output[j][i] = input[j+1][i] + input[j][i];
  }
}
```

Now assume that the cache still has the 64 byte line size as before, but **is now only 32 KB in size**. Please rewrite the code so that you minimize the number of cache misses. You can only modify the loop structure (you can add/remove loops and adjust loop bounds) and modify array indexing, but you cannot change the number of mathematical operations performed. **Pseudocode is fine, it need not compile. But to help the grader understand your code, please briefly (in a sentence or two) describe the rough approach used in your solution.**

A Grading Pipeline

PRACTICE PROBLEM 6:

Some of the CS149 CAs get organized to grade the midterm, which you can assume has five questions. To ensure fairness, they decide that each question should be graded by one CA, and that to grade each exam they will organize themselves in a pipeline. James takes question 1, Arden takes question 2, Raj takes question 3, Keshav question 4, and Edmund question 5. The CAs sit in a line at the same table, and for each exam James grades question 1, then passes the exam to Arden who grades Q2, who passes to Raj to grade Q3, etc. Pranil and Drew throw up their hands and say "there seems to only be five questions, well we guess we can sit this one out!

Assume that it takes James 5 minutes to grade each exam's Q1, Arden needs 6 minutes to grade Q2, Raj needs 15 minutes to grade Q3, Keshav needs 5 minutes to grade Q4 and Edmund needs 5 minutes to grade Q5. The CAs grade exams in a pipelined fashion to maximize their throughput.

A. Given this configuration, what is the **latency** of completing the grading of any one exam?

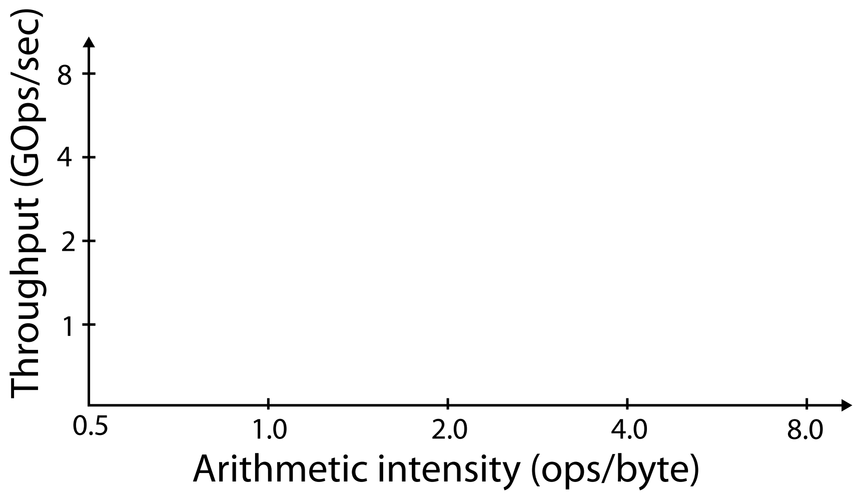
B. What is the **STEADY-STATE THROUGHPUT** of the CAs, in terms of exams per hour? Keep in mind that the CAs are pipelining grading of exams, so James grabs the next exam to grade as soon as he is done with Q1 from his current exam. (While it doesn't matter in the answer to this problem, since we are asking for steady-state throughput, it might be helpful to assume that the pile of ungraded exams between any two CAs is limited to a small fixed size.)

C. The professors start getting anxious because the CAs haven't completed grading, so they send an angry email to the staff. "Let's speed it up already!" they write. The CA's look at Pranil and Drew and, say "Quit surfing the internet reading articles about ML accelerators and please come help!" Assuming that Pranil and Drew grade questions at exactly the same speed as the other CAs, which question should they help with grading? (please choose one) Describe why? What is the new **steady state throughput** of the staff in terms of exams per hour? (Regardless of the question chosen, Assume that Pranil/Drew's help is going to come in the form of grading a different exam in parallel with the other CAs working on the same question. Their help doesn't reduce the amount of time it takes to grade one question on one exam.)

PRACTICE PROBLEM 7:

A. In class we described the usefulness of making roofline graphs, which plots the instruction throughput of a machine (gigaops/sec) as a function of a program's arithmetic intensity (ops performed per byte transferred from memory). Note moving along the X axis is changing the properties of the code being run. The Y axis plots the performane of the machine when running a specified program. Consider the roofline plot below. Please plot the roofline curve for a machine featuring a **1 GHz dual-core processor. Each core can execute one 4-wide SIMD instruction per clock**. This processor is connected to a memory system providing 4 GB/sec of bandwidth. *Hint: what is the peak throughput of this processor? What are its bandwidth requirements when running a piece of code with a specified arithmetic intensity? Recall ops/second × bytes/op is bytes/sec. Arithemetic intensity is 1/(bytes/op).*

Plot the expected throughput of the processor when running code at each arithmetic intensity on the X axis, and draw a line between the points.



B. Consider the following piece of C code.

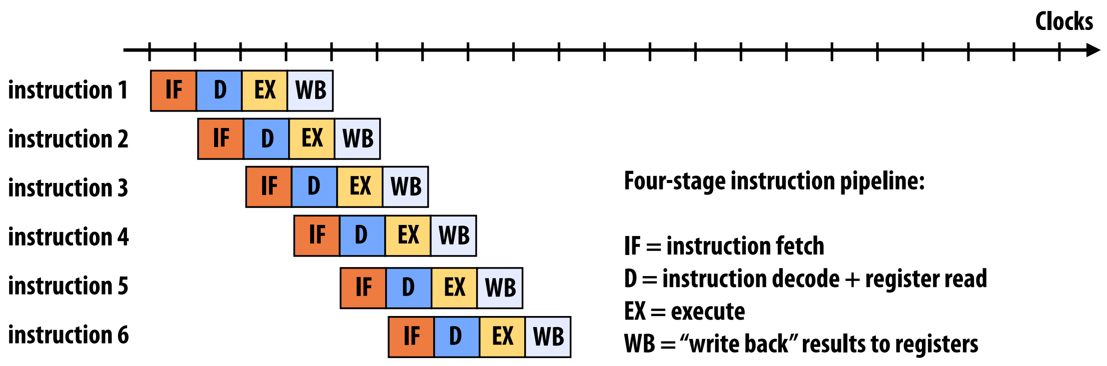
```
float A[VERY_LARGE];
float B[VERY_LARGE];
float C[VERY_LARGE];
float D[VERY_LARGE];
float E[VERY_LARGE];

for (int i=0; i<VERY_LARGE; i++)
  C[i] = A[i] * B[i];
for (int i=0; i<VERY_LARGE; i++)
  D[i] = C[i] + B[i];
for (int i=0; i<VERY_LARGE; i++)
  E[i] = D[i] - A[i];
```

Assume that `VERY_LARGE` is so large that the arrays are hundreds of MBs in size, and that the code is run on a single-core processor with a 8 MB cache. Please modify the program to maximize its arithmetic intensity. You only need to write to the output array `E`, you don't need to fill in `C` and `D` if it is not necessary. However, please DO NOT CHANGE the number of math operations performed. **If we assume the program before and after the modification is bandwidth bound, how much does your modification improve its performance?**

PRACTICE PROBLEM 8:

The fast-growing startup Cardinal Processors, Inc. builds a single core, single threaded processor that executes instructions using a simple four-stage pipeline. As shown in the figure below, each stage of the pipeline (called if the IF, D, EX, and WB stages) performs its work for an instruction **in one clock**. To keep things simple, assume this is the case for all instructions in the program, including loads and stores (memory is infinitely fast).

The figure shows the execution of a program with six **independent instructions** on this processor. *However, if instruction B depends on the results of instruction A, instruction B will not begin the IF phase of execution until the clock after WB completes for A.*



A. Assuming all instructions in a program are **independent** (yes, a bit unrealistic) what is the instruction throughput of the processor?

B. Assuming all instructions in a program are **dependent** on the previous instruction, what is the instruction throughput of the processor?

C. What is the latency of completing an instruction?

D. Imagine the IF stage is modified to improve its throughput to fetch TWO instructions per clock, but no other part of the processor is changed. What is the new overall maximum instruction throughput of the processor?

E. Consider the following C program:

```
float A[500000];
float B[500000];
// assume A is initialized here

for (int i=0; i<500000; i++) {
    float x1 = A[i];
    float x2 = 6 * x1;
    float x3 = 4 + x2;
    B[i] = x3;
}
```

Assuming that we consider only the four instructions in the loop body (for simplicity, disregard instructions for managing the loop or calculating load/store addresses), what is the average instruction throughput of this program? (Hint: You should probably consider instruction dependencies, and at least two loop iterations worth of work).

F. Modify the program to achieve peak instruction throughput on the processor. Please give your answer in C-pseudocode.

G. Now assume the program is reverted to the original code from part E, but the for loop is parallelized using a C++ language extension called OpenMP. OpenMP is a set of C++ compiler extensions that enable thread-parallel execution, and in the case of the code below, iterations of the for loop that's prefixed by `omp parallel for` will be carried out in parallel by a pool of worker threads.)

```
// assume iterations of this FOR LOOP are parallelized across multiple
// worker threads in a thread pool.
#pragma omp parallel for
for (int i=0; i<100000; i++) {
  float x1 = A[i];
  float x2 = 2*x1;
  float x3 = 3 + x2;
  B[i] = x3;
}
```

Given this program, imagine you wanted to add multi-threading to the **single-core processor** to obtain **peak instruction throughput** (100% utilization of execution resources). What is the smallest number of threads your processor could support and still achieve this goal? You may not change the program.

Because The Professor with the Most ALUs (Sometimes) Wins

PRACTICE PROBLEM 9:

Consider the following ISPC code that computes $ax^2 + bx + c$ for elements x of an entire input array.

```
void polynomial(float a, float b, float c,
                uniform float x[], uniform float output[], int elementsPerTask) {
  uniform int start = taskIndex * elementsPerTask;
  uniform int end = start + elementsPerTask;

  foreach (i = start ... end) {
    output[i] = (a * x[i] * x[i]) + (b * x[i]) + c;   // 5 arithmetic ops
  }
}

// assume N is very, very large, and is a multiple of 1024
void run(int N, float a, float b, float c, float* input, float* output) {
  uniform int elementsPerTask = 1024;
  launch[N/elementsPerTask] polynomial(a, b, c, input, output, elementsPerTask);
}
```

Professor Kayvon, seeking to capture the highly lucrative polynomial evaluation market, builds a multi-core CPU packed with ALUs. "The professor with the most ALUs wins, he yells!" The processor has:

- 4 cores clocked at 1 GHz, capable of one 32-wide SIMD floating-point instruction per clock (1 addition, 1 multiply, etc.)
- Two hardware execution contexts per core
- A 1 MB cache per core with 128-byte cache lines (In this problem assume allocations are cache-line aligned so that each SIMD vector load or store instruction will load one cache line). Assume cache hits are 0 cycles.
- The processor is connected to a memory system providing a whopping 512 GB/sec of BW
- The latency of memory loads is 95 cycles. (There is no prefetching.) For simplicity, assume the latency of stores is 0.

A. What is the peak arithmetic throughput of Prof. Kayvon's processor?

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

C. Prof. Kayvon runs the ISPC code on his new processor, the performance of the code is not good. What fraction of peak performance is observed when running this code? Why is peak performance not obtained?

D. Prof. Olukuton sees Kayvon's struggles, and sees an opportunity to start his own polynomial computation processor company that achieves double the performance of Prof. Kayvon's chip. "Oh shucks, now I'll have to double the number of cores in my chip, that will cost a fortune." Kayvon says.

TA Mario writes Kayvon an email that reads "There's another way to achieve peak performance with your original design, and it doesn't require adding cores." Describe a change to Prof. Kayvon's processor that causes it to obtain peak performance on the original workload. Be specific about how you'd realize peak performance (give numbers).

The following year Prof. Kayvon makes a new version of his processor. The new version is the **exact same quad-core processor** as the one described at the beginning of this question, except now the chip **supports 64 hardware execution contexts per core**. Also, the ISPC code is changed to compute a more complex polynomial. In the code below assume that `coeffs` is an array of a few hundred polynomial coefficients and that `expensive_polynomial` involves 100's of arithmetic operations.

```
void polynomial(uniform float coeffs[], uniform float input[],
                uniform float output[], int elementsPerTask) {
  uniform int start = taskIndex * elementsPerTask;
  uniform int end = start + elementsPerTask;
  foreach (i = start ... end) {
    output[i] = expensive_poly(coeffs, input[i]);   // 100's of arithmetic ops
  }
}

void run(int N, float* coeffs, float* input, float* output) {
  uniform int elementsPerTask = 1024;
  launch[N/elementsPerTask] polynomial(coeffs, input, output, elementsPerTask);
}
```

E. What is the peak arithmetic throughput of Prof. Kayvon's new processor?

F. Imagine running the program with N=8×1024 and N = 64×1024. Assuming that the system schedules worker threads onto available execution contents in an efficient manner, do either of the two values of N result in the program achieving near peak utilization of the machine? Why or why not? (For simplicity, assume task launch overhead is negligible.)

G. Now consider the case where N=9×1024. Now what is the performance problem? Describe is simple code change that results in the program obtaining close to peak utilization of the machine. (Assume task launch overhead is negligible.)

PRACTICE PROBLEM 10:

In class we talked about the `barrier()` synchronization primitive. No thread proceeds past a barrier until all threads in the system have reached the barrier. (In other words, the call to `barrier()` will not return to the caller until its known that all threads have called `barrier()`. Consider implementing a barrier in the context of a message passing program that is only allowed to communicate via **blocking sends and receives**. Using only the helper functions defined below, implement a barrier. Your solution should make no assumptions about the number of threads in the system. **Keep in mind that all threads in a message passing program execute in their own address space—there are no shared variables.**

```
// send msg with id msgId and contents msgValue to thread dstThread
void blockingSend(int dstThread, int msgId, int value);

// recv message from srcThread. Upon return, msgId and msgValue are populated
void blockingRecv(int srcThread, int* msgId, int* msgValue);

// returns the id of the calling thread
int getThreadId();

// returns the number of threads in the program
int getNumThreads();
```

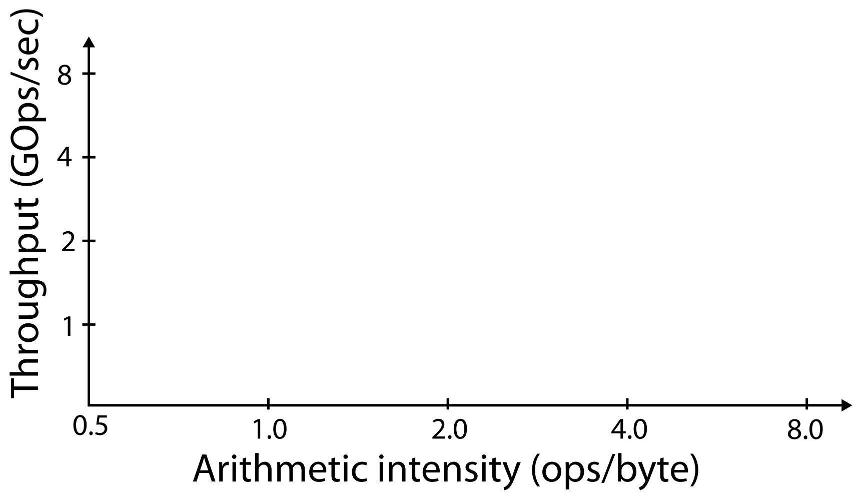
PRACTICE PROBLEM 11:

A. When we discussed Cilk, we emphasized how `cilk_spawn foo()` differs from a normal C function call `foo()` in that the Cilk call can run asynchronously with the caller. Notice that Cilk doesn't explicitly state that the callee function runs **in parallel with the caller**. Give one reason why the designers of Cilk intentionally designed a language that does not specify when the call will run relative to the caller?

PRACTICE PROBLEM 12:

A. In class we described the usefulness of making roofline graphs, which plots the instruction throughput of a machine (gigaops/sec) as a function of a program's arithmetic intensity (ops performed per byte transferred from memory). Note moving along the X axis is changing the properties of the code being run. The Y axis plots the performane of the machine when running a specified program. Consider the roofline plot below. Please plot the roofline curve for a machine featuring a **1 GHz dual-core processor. Each core can execute one 4-wide SIMD instruction per clock**. This processor is connected to a memory system providing 4 GB/sec of bandwidth. *Hint: what is the peak throughput of this processor? What are its bandwidth requirements when running a piece of code with a specified arithmetic intensity? Recall ops/second $\times$ bytes/op is bytes/sec. Arithemetic intensity is 1/(bytes/op).*

Plot the expected throughput of the processor when running code at each arithmetic intensity on the X axis, and draw a line between the points.



B. 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];
```

C. Consider the following piece of C code.

```
float A[VERY_LARGE];
float B[VERY_LARGE];
float C[VERY_LARGE];
float D[VERY_LARGE];
float E[VERY_LARGE];

for (int i=0; i<VERY_LARGE; i++)
  C[i] = A[i] * B[i];
for (int i=0; i<VERY_LARGE; i++)
  D[i] = C[i] + B[i];
for (int i=0; i<VERY_LARGE; i++)
  E[i] = D[i] - A[i];
```

Assume that `VERY_LARGE` is so large that the arrays are hundreds of MBs in size, and that the code is run on a single-core processor with a 8 MB cache. Please modify the program to maximize its arithmetic intensity. You only need to write to the output array `E`, you don't need to fill in `C` and `D` if it is not necessary. However, please DO NOT CHANGE the number of math operations performed. **If we assume the program before and after the modification is bandwidth bound, how much does your modification improve its performance?**

Fusion, Fusion, Fusion

PRACTICE PROBLEM 13:

Your boss asks you to buy a computer for running the program below. The program uses a math library (`cs149_math`). The library functions should be self-explanatory, but example implementations of the `cs149math_add` and `cs149math_sum` functions are given below.

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

void  cs149math_sub(float* A, float* B, float* output);
void  cs149math_mul(float* A, float* B, float* output);

void cs149math_add(float* A, float* B, float* output) {
  // Recall from written asst 1 that this OpenMP directive tells the
  // C compiler that iterations of the for loop are independent, and
  // that implementations of C compilers that support
  // OpenMP will parallelize this loop using multiple threads.
  #omp parallel for
  for (int i=0; i<N; i++)
    output[i] = A[i]+B[i];
}

float cs149math_sum(float* A) {     // compute sum of all elements of the input array
  atomic<float> x = 0.0;
  #omp parallel for
  for (int i=0; i<N; i++)
    x += A[i];
  return x;
}

/////////////////////////////////////////////////////////////////
// The program is below:
/////////////////////////////////////////////////////////////////

// assume arrays are allocated and initialized
float* src1, *src2, *src3, *tmp1, *tmp2, *tmp3, *dst;

cs149math_add(src1, src2, tmp1);     // 1
cs149math_mul(tmp1, src3, tmp2);     // 2
cs149math_mul(tmp2, src1, tmp3);     // 3
float x = cs149math_sum(tmp2) / N;   // 4
if (x > 10.0) {
  cs149math_mul(tmp3, src1, tmp1);   // 5
  cs149math_add(src1, tmp1, tmp2);   // 6
  cs149math_add(src1, tmp2, dst);    // 7
} else {
  cs149math_add(tmp3, src2, tmp1);   // 8
  cs149math_mul(src2, tmp1, tmp2);   // 9
  cs149math_mul(src2, tmp2, dst);    // 10
}
```

The question is on the next page...

You have two computers to choose from, of equal price. (Assume that both machines have the same 16MB cache and 0 memory latency.)

1. Computer A: Four cores 1 GHz, 4-wide SIMD, 192 GB/sec bandwidth
2. Computer B: Four cores 1 GHz, 8-wide SIMD, 128 GB/sec bandwidth

ASSUME THAT YOU ARE ALLOWED TO REWRITE THE CODE, INCLUDING REPLACE LIBRARY CALLS IF DESIRED, (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.). **Please give the arithmetic intensity of your new program assuming that both loads and stores are 4 bytes of data transfer. (You can also assume 1 GB is 10^9 bytes.)** As a result, which machine do you choose? Why? (If you decide to change the program please give a pseudocode description of your changes. What is parallelized, vectorized, what does the loop structure look like, etc.)

An Interesting CUDA Program

Problem 6: (Graded for Correctness - 30 pts)

Consider the CUDA function `sortOfExp()` implemented below. The function is almost like an exponentiation functions, but not quite. Technically, for values of `expValue` of 2 or greater, it computes:
$2 \times 2 \times 4^{\text{expValue}-2} \times \text{baseValue}^{\text{expValue}}$

```
__global__ void sortOfExp(int* inputExps, float* inputValues) {
  int threadId = blockIdx.x * blockDim.x + threadIdx.x;
  int expValue = inputExps[threadId];       // 1 int memory load
  float baseValue = inputValues[threadId];  // 1 float memory load
  float result = 1.0;

  for (int i=0; i<expValue; i++) {  // assume loop arithmetic is "free"
     result *= baseValue;           // 1 arithmetic op
     if (i < 2) {                   // assume this check is "free"
        result *= 2.0f;             // 1 arithmetic op
     } else {
        result *= 4.0f;             // 1 arithmetic op
     }
  }
  resultValues[threadId] = result;  // 1 float memory store
}
```

You run this CUDA program on an `inputExps` array of size $1024 \times 1024 \times 1024$ that is initialized with random values between 1 and 8. In every group of 8 values **there is at least one value 8. For example:**

```
1 3 5 8 1 1 2 8   1 1 1 1 1 2 8 1   3 8 8 8 3 1 7 2   8 8 8 8 8 8 8 8 ...
```

Your application will initiate a single bulk launch of 1024^3 CUDA threads (each thread generates one output.) Although it is not relevant to the problem, you can assume a thread block size of 32.

Please assume that you are running on a GPU running at 1 GHz with 8 cores ("SMs in NV-speak"). Each core has a SIMD width of 32 (like NVIDIA GPUs), has 32 CUDA threads worth of execution contexts (one "warp"), and can one run 1 instruction (arithmetic, load, or store) for all threads in the warp in a clock in a SIMD fashion. (Loads and stores, like arithmetic, take 1 cycle.)

A. (10 pts) Assuming that CUDA threads with consecutive thread IDs execute in a single warp, what are the number of cycles needed for each warp's worth of CUDA threads to complete execution? Please include the cycles used to issue loads and stores as part of your computation.

B. (10 pts) Regardless of the answer you computed above, let's assume that program above needs 30 processor cycles to issue all the instructions for a warp. If that's the case (Note, that's **not the right answer to Part A**, but we want you to assume 30 for now to avoid coupling answers.) Assuming that the GPU is the same as it was for Part A (1 GHz, 8 SM cores, 32-wide SIMD, 32 execution contexts per core operating together as a warp), but now the GPU's memory system has 0 memory latency and 32 GB/sec of memory bandwidth.

Given this setup, **is the program memory bound or compute bound on this GPU?** Please show your calculations, and **if you conclude it is bandwidth bound what fraction of time do you estimate the GPU is waiting on memory?** You may count reads as 4 bytes of memory traffic and writes as 4-bytes of memory traffic, and for easy make treat 1 GB as 10^9 bytes.

C. (10 pts) Now imagine we keep the program the same, and like in part B assert that there are 30 cycles of (load-/store/arithmetic) instructions for each warp's worth of kernel execution. But now the GPU has changed so that it has infinite memory bandwidth and memory latency of 75 cycles. (A load issued on cycle 0 is ready to be used on cycle 75). Assume that the GPU can support an unlimited number of outstanding memory transactions and that GPU cores NEVER stall waiting for stores to complete. Under these conditions, what is the minimum number of CUDA thread execution contexts (or equivalently, the minimum number of warps worth of execution contexts) needed to ensure that the GPU cores are NEVER stalled waiting on memory?

Introducing PKPU2.0: The GPU for the Metaverse

PRACTICE PROBLEM 14:

Inspired by their early success documented in prior practice problems, the midterm practice problems, your CS149 instructors decide to take on NVIDIA in the GPU design business, and launch PKPU2.0... the GPU designed (their marketing team claims) for metaverse applications! (PKPU stands for Prof. Kayvon Processing Unit, or Prof Kunle Processing Unit). The PKPU2.0 runs CUDA programs exactly the same manner as the NVIDIA GPUs discussed in class, but it has the following characteristics:

- The processor has 16 cores (akin to NVIDIA SMs) running at 1 GHz.
- The cores execute CUDA threads in an implicit SIMD fashion running 32 consecutively numbered CUDA threads together using the same instruction stream (PKPU2.0 implements 32-wide "warps").
- Each core provides execution contexts for up to 256 CUDA threads (eight PKPKU2.0 warps). 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 will fetch/decode one single-precision floating point arithmetic instruction (add, multiply, compare, etc.) per clock (one fp operation completes per clock per ALU). Keep in mind this instruction is executed on an entire warp in that clock, so exactly one warp can make progress each clock. As we've often done in prior problems, you can assume that all other instructions (integer ops, load/stores are "free" in that they are executed on other hardware units in the core, not the main floating point ALUs.)

A. When running at peak utilization. What is the PKPU2.0's **maximum throughput** for executing **floating-point math operations**?

B. 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 one element from the input array X. Assume that (1) the program is run on large arrays of size 128 million elements, (2) the CUDA program is compiled using a CUDA thread-block size of 128 threads, and (3) enough thread blocks are created in the bulk thread launch so that there is exactly one CUDA thread per output array element.

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

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

   float val = X[idx];         // load instr

   float output;
   float val2 = 2.0 * val;     // 1 arithmetic cycle
   if (val2 > 0.0) {           // 1 arithmetic cycle
      output = f1(val);        // 14 arithmetic cycles
   } else {
      output = f2(val);        // 14 arithmetic cyles
   }

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

The input array contains values with the following pattern: (recall there are 128M elements)

```
  [ 1.0,  2.0, ...,  32.0,
   -1.0, -2.0, ..., -32.0,
    1.0,  2.0, ...,  32.0,
   -1.0, -2.0, ..., -32.0, ...]
```

Does this workload suffer from instruction stream divergence? Please state YES or NO and explain why.

C. Given the input values shown in the previous problem, what is the arithmetic intensity of the program, **in terms of PKPKU2.0 cycles of floating point arithmetic (accounting for the potential of divergence) per bytes transferred from memory?** Please write your answer as a fraction. (Hint: This is best computed at the granularity of a warp!)

.

D. **Assume that on the PKPKU2.0, the memory latency of loads is 50 cycles.** (Assume stores have 0 latency and assume (for now) that the memory system has very high bandwidth.) Does the PKPKU2.0 have the ability to hide all memory latency from loads? Why or why not?

E. Now assume that the PKPU2.0 memory system has 128 GB/sec of bandwidth (and still has a load latency of 50 cycles. Is this program compute bound or bandwidth bound on the PKPU2.0? (show calculations underlying your answer) If you conclude the PKPU2.0 is bandwidth bound running this code, tell us what the utilization of the processor will be. **Remember the PKPKU2.0 has 16 cores operating at 1 GHz.**

F. You are hired to improve the PKPKU's performance on this workload. You have four options.

(a) Increase the maximum number of CUDA thread execution contexts by $2\times$.

(b) Triple the memory bandwidth.

(c) Add a data cache that can hold 1/2 of the elements in the input and output arrays.

(d) Double the SIMD width (aka warp size) to 64 (while still maintaining the ability to run exactly one instruction per warp per clock.

Which option do you choose to get the best performance on the given input data, and what speedup do you expect to observe (compared to the original unmodified PKPKU2.0) from this change? Explain why. (Note: assume that at the start of the CUDA program's execution, all the input/output data is located in main memory, and is not resident in cache.)