Stanford CS149: Parallel Computing
Written Assignment 3

An Interesting CUDA Program

Problem 1: (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?

Async Message Ping Pong

Problem 2: (Graded on Effort Only - 35 pts)

Consider the following API for sending and receiving messages. The API supports asynchronous sends and receives, so there are calls to initiate sends/recvs and to check to see if the message send/recv has been completed.

```
HANDLE asyncSend(int* ptr);    // initiate send of int pointed to by ptr
bool   testSendDone(HANDLE h); // test to see if msg h is complete. Once complete
                               // will return true no matter how many times it's called

HANDLE asyncRecv(int* ptr);    // initiate recv of the int pointed to by ptr
bool   testRecvDone(HANDLE h); // test to see if the msg h is complete. Once complete
                               // will return true no matter how many times it's called
```

Also assume you have a function that returns a unique random integer. You are GUARANTEED all values returned from `randomInt` are unique.

```
int randomInt();
```

Using this API write the code for a program that does the following:

- Thread A must generate `COUNT` random integers and send them to thread B.
- Thread B receives the `COUNT` random integers and then sends all `COUNT` values back to thread A
- Thread A must check to make sure it gets the all the same values back, and then prints "DONE" only after it has confirmed it has received all correct values back from thread B.
- The network has high latency, but can support any number of outstanding messages. **Your solution should realize maximum parallelism in network transfers.**
- The network might deliver messages to the receiver IN A DIFFERENT ORDER THAN THE SENDER sent them. In other words if thread 0 sends message A and then message B to thread 1, it is possible for thread 1 to confirm that message B is complete prior to message A.

Your solution may allocate any temporary variables. In your solution "spinning" to check to make sure operations are complete is fine. (There is no way to sleep and be awoken when key events occur.) We started the implementation of `threadA` for you.

Write your solution on the next page.

```
// put logic for thread A here. You may allocate any variables you wish
void threadA() {
  int    outValues[COUNT];
  HANDLE sentHandles[COUNT];



  // initiate the initial sends
  for (int i=0; i<COUNT; i++) {
     outValues[i] = randInt();
     sentHandles[i] = asyncSend(&outValues[i]);
  }

}

// put logic for thread B here. You may allocate any variables you wish
void threadB() {

}
```

Problem 3: (Graded on Effort Only - 35 pts)

In this problem we want you to implement a *very simple* version of Spark, called CS149Spark, that supports only a few operators. You will implement CS149Spark as a simple C++ library consisting of a base class `RDD` as well as subclasses for all CS149Spark transforms.

```
  class RDD {
    public:
    virtual bool hasMoreElements() = 0;   // all RDDs must implement this
    virtual string next() = 0;            // all RDDs must implement this

    int count() {                  // returns number of elements in the RDD
      int count = 0;
      while (hasMoreElements()) {
         string el = next();
         count++:
      }
      return count;
    }

    vector<string> collect() {     // returns STL vector representing RDD
      vector<string> data;
      while (hasMoreElements()) {
         data.append(next());
      }
      return data;
    }
};

class RDDFromTextFile : public RDD {
  ifstream inputFile;             // regular C++ file IO object
  public:
  RDDFromTextFile(string filename) {
    inputFile.open(filename);     // prepares file for reading
  }

  bool hasMoreElements() {
    return !inputFile.eof();      // .eof() returns true if no more data to read
  }

  string next() {
    return inputFile.readLine();  // reads next line from file
  }
};
```

For example, given the two definitions above, a simple program that counts the lines in a text file can be written as such.

```
  RDDFromTextFile r("myfile.txt’’);   // creates an RDD where each element is a string
                                      // corresponding to a line from the text file

  printf("The RDD has length %d\n", r.count());
```

A. Now consider adding a `l33tify` RDD transform to CS149Spark, which returns a new RDD where all instances of the character 'e' in string elements of the source RDD are converted to the character '3'. For example, the following code sequence creates an RDD (`r1`) whose elements are lines from a text file. The RDD `r2` contains a l33tified version of these strings. This data is collected into a regular C++ vector at the end of the program using the call to `collect()`.

```
RDDFromTextFile r1("myfile.txt'');    // creates an RDD where each element is a string
                                      // corresponding to a line from the text file

RDDL33tify r2(r1);                    // l33tify all elements for r1
vector<string> lines = r2.collect();  // lines from the file, but in l33t form
```

Implement the functions `hasMoreElements()` and `next()` for the `l33tify` RDD transformation below. **A full credit solution will use minimal memory footprint and never recompute (compute more than once) any elements of any RDD.**

```
///////////////////////////////////////////////////////////////
class RDDL33tify : public RDD {

  RDD parent;


  RDDL33tify(RDD parentRDD) {
    parent = parentRDD;



  }

  bool hasMoreElements() {









  }

  string next() {











  }
};
```

B. Now consider a transformation `FilterLongWords` that filters out all elements of the input RDD that are strings of greater than 32 characters.

Again, we want you to implement `hasMoreElements()` and `next()`.

You may declare any member variables you wish and assume `.length()` exists on strings. **Careful: `hasMoreElements()` is trickier now! Again a full credit solution will use minimal memory footprint and never recompute any elements of any RDD.**

A sample program using the `FilterLongWords` RDD transformation is below:

```
RDDFromTextFile r1("myfile.txt");   // creates an RDD where each element is a string
                                     // corresponding to a line from the text file

RDDL33tify r2(r1);                   // converts elements to l33t form
RDDFilterLongWords r3(r2);           // removes strings that are greater than 32 characters
print("RDD r3 has length %d\n", r3.count());
```

```
//////////////////////////////////////////////////////////////////////
class RDDFilterLongWords : public RDD {

  RDD parent;


  public:
  RDDFilterLongWords(RDD parentRDD) {
    parent = parentRDD;


  }

  bool hasMoreElements() {













  }

  string next() {














  }
};
```

C. Finally, implement a `groupByFirstWord` transformation which is like Spark's `groupByKey`, but instead (1) it uses the first word of the input string as a key, and (2) instead of building a list of all elements with the same key, concatenates all strings with the same key into a long string.

For example, `groupByFirstWord` on the RDD ["hello world", "hello cs149", "good luck", "parallelism is fun", "good afternoon"] would produce the RDD ["hello world hello cs149", "good luck good afternoon", "parallelism is fun"].

Your implementation can be rough pseudocode, and may assume the existence of a dictionary data structure (mapping strings to strings) to actually perform the grouping, an iterator over the dictionaries keys, and useful string functions like: `.first()` to get the first word of a string, and `.append(string)` to append one string to another.

Rough pseudocode is fine, but your solution should make it clear how you are tracking the next element to return in `next()`. A full credit solution will use minimal memory footprint and never recompute any elements of any RDD.

```
class RDDGroupByFirstWord : public RDD {
  RDD parent;
  Dictionary<string, string> dict;       // assume dict["hello''] returns the string
                                         // associated with key "hello''



  public:
  RDDGroupByFirstWord(RDD parentRDD) {
    parent = parentRDD;


  }

  bool hasMoreElements() {










  }

  string next() {











  }
};
```

D. Describe why the RDD transformations `L33tify`, `FilterLongWords`, and RDD construction from a file, as well as the action `count()` can all execute efficiently on very large files (consider TB-sized files) on a machine with a small amount of memory (1 GB of RAM).

E. Describe why the transformation `GroupByFirstWord` differs from the other transformations in terms of how much memory footprint it requires to implement.

Introducing PKPU2.0: The GPU for the Metaverse

PRACTICE PROBLEM 1:

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.)

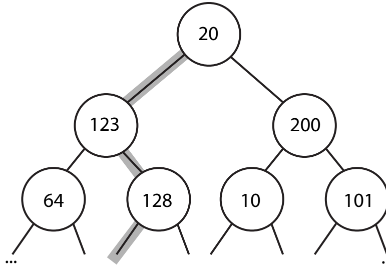
Miscellaneous Short Problems

PRACTICE PROBLEM 2:

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 3:

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 4:

Yes, Prof. Kayvon knows this question is dated. Do you want practice problems or do you want to wait for the Sabrina Carpenter / Chappell Roan question?

Justin Timberlake wants to get back in the news for the write seasons. He hears that Spark is all the rage and decides he's going to code up his own implementation to compete against that of the Apache project. Justin's first test runs the following Spark program, which creates four RDDs. The program takes Justin's lengthy (1 TB!) list of dancing tips and finds all misspelled words.

```
var lines = spark.textFile("hdfs://mydancetips.txt'');     // 1 TB file
var lower = lines.map( x => x.toLower() );                 // convert lines to lower case
var words = lower.flatmap( x => x.split(`` '') );          // convert RDD of lines to RDD of
                                                           // individual words
var misspelled = words.filter( x => !x.isInDictionary() ); // filter to find misspellings

print misspelled.count();   // print number of misspelled words
```

A. Understanding that the Spark RDD abstraction affords many possible implementations, Justin decides to keep things simple and implements his Spark runtime such that each RDD is implemented by a fully allocated array. This array is stored either in memory or on disk depending on the size of the RDD and available RAM. **The array is allocated and populated at the time the RDD is created — as a result of executing the appropriate operator (map, flatmap, filter, etc.) on the input RDD.**

   Justin runs his program on a cluster with 10 computers, each of which has 100 GB of memory. The program gets correct results, but Justin is devastated because the program runs *incredibly slow*. He calls his friend Taylor Swift, ready to give up on the venture. Encouragingly, Taylor says, "shake it off Justin", just run your code on 40 computers. Justin does this and observes a speedup much greater than $4\times$ his original performance. Why is this the case?

B. With things looking good, Justin runs off to write a new single "Bringing Locality Back" to use in the marketing his product. At that moment, Taylor calls back, and says "Actually, Justin, I think you can schedule the computations much more efficiently and get very good performance with less memory and far fewer nodes." Describe how you would change how Justin schedules his Spark computations to improve memory efficiency and performance.

C. After hacking until midnight, which in term inspired Taylor's recent album), Justin and Taylor run the optimized program on 10 nodes. The program runs for 1 hour, and then right before `misspelled.count()` returns, node 6 crashes. Justin is devastated! He says, "Taylor, I have a single to release, and I don't have time to deal with rerunning programs from scratch." Taylor gives Justin a stink eye and says, "Don't worry, it will be complete in just a few minutes." Approximately how long will it take after the crash for the program to complete? You should assume the `.count()` operation is essentially free. But please **clearly state any assumptions about how the computation is scheduled in justifying your answer.**

One more question about Spark

PRACTICE PROBLEM 5:

Consider the following program written using Spark RDDs, in a C-like syntax. Assume that `readRDDFromFile()` generates an RDD with elements of type `int` by reading numbers from a file, and that the functions `addOne()` and `addTwo()` are defined as given below. **You may also assume that `map()`, `readRDDFromFile()`, and `writeRDDToFile()` are THE ONLY transformations allowed on RDDs.**

```
int addOne(int x) { return x+1; }
int addTwo(int x) { return x+2; }

RDD r1 = readRDDFromFile();
RDD r2 = r1.map(addOne);
RDD r3 = r2.map(addTwo);
writeRDDToFile(r3);
```

Assume that there are N numbers in the file, and consider two potential implementations of this program. In the code below, `readIntFromFile()` and `writeIntToFile()` read/write exactly one integer to/from the file.

```
// IMPLEMENTATION 1

int array1[N];
int array2[N];
int array3[N];

for (int i=0; i<N; i++)
   array1[i] = readIntFromFile();
for (int i=0; i<N; i++)
   array2[i] = addOne(array1[i]);
for (int i=0; i<N; i++)
   array3[i] = addTwo(array2[i]);
for (int i=0; i<N; i++)
   writeIntToFile(array3[i]);

// IMPLEMENTATION 2

for (int i=0; i<N; i++) {
  writeIntToFile(addTwo(addOne(readIntFromFile())));
}
```

The second implementation computes elements of the three RDDs in a different order than the first implementation. It also clearly uses far less memory than the first. Are both implementations correct implementations of the Spark RDD abstraction? (In other words do they both compute the expected result?) If your answer is yes, please describe WHAT properties of RDDs and RDD transformations allow for both of these two different implementations. If your answer is no, please describe why. **(Please ignore robustness to node failure in this problem.)**

Fusion, Fusion, Fusion

PRACTICE PROBLEM 6:

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

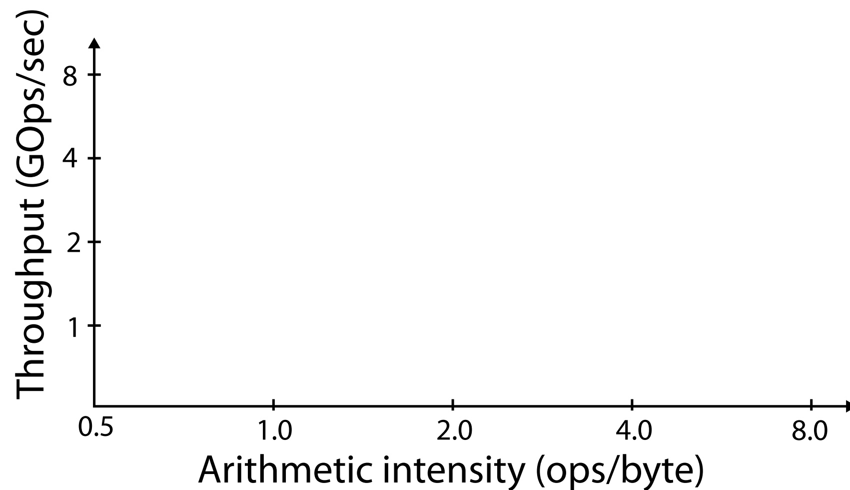
ASSUME THAT YOU ARE ALLOWED TO REWRITE THE CODE, INCLUDING REPLACE LIBRARY CALLS IF DESIRED, (provided that it computes exactly the same answer—You can parallelize across cores, vectorize, reorder loops, etc. but you are not permitted to change the math operations to turn adds into multiplies, eliminate 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.)

Misc Problems

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 $\times$ 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?**