Stanford CS149: Parallel Computing
Written Assignment 3 SOLUTIONS

Implementing CS149 Spark

Problem 1. (50 points):

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. (10 pts) 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() {
    parent.hasMoreElements();
  }

  string next() {
    // assume l33tify(string s) is a function that returns a string that
    // has all 'e' characters replaced by 3's.
    return l33tify(parent.next());
  }
};
```

B. (10 pts) 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;
  string nextEl = "";                 // assume empty string is reserved
  public:
  RDDFilterLongWords(RDD parentRDD) {
    parent = parentRDD;
  }

  bool hasMoreElements() {
    if (nextEl.compare("") != 0)      // do not advance if next was not called
       return true;
    while (parent.hasMoreElements()) {
      string s = parent.nextElement();
      if (s.length() <= 32) {
         nextEl = s;
         return true;
      }
    }

    return false;
  }

  string next() {
    if (hasMoreElements()) {
      string s = nextEl;
      nextEl = '';
      return s;
    }
  }
};
```

C. (10 pts) 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''
  Iter nextKey = dict.firstKey();

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

  bool hasMoreElements() {
    if (!processParent) {
      while (parent.hasMoreElements()) {
        string s = parent.next();
        dict[s.first()].append(s);
      }
      processParent = true;
      nextKey = dict.firstKey();
    }

    return nextKey != dict.lastKey();
  }

  string next() {
    string s = dict[nextKey];
    nextKey++;
    return s;
  }
};
```

D. (10 pts) 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).

Solution: They can all be streamed and require O(1) space.

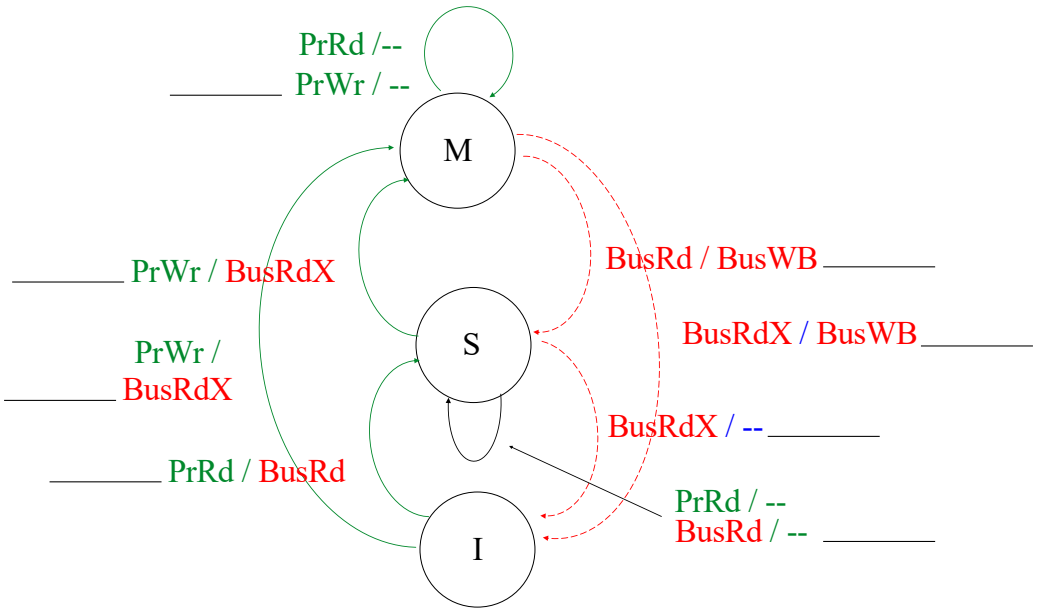
E. (10 pts) Describe why the transformation `GroupByFirstWord` differs from the other transformations in terms of how much memory footprint it requires to implement.

Solution: It requires generating all elements of the parent RDD prior to emitting it's first element. As a result, all its elements must be buffered, and so its memory storage requirements is O(size of RDD), not O(1).

MSI Coherence Protocol Warmup

Problem 2. (10 points):

Below is a state diagram for the MSI protocol.



Consider the follow sequence of operations by processors 1 and 2. Assume that each processor has a local cache carrying out the MSI protocol. Each cache can store two cache lines of data (there's room for both X and Y). For simplicity, please assume that the variables X and Y take an entire cache line.

Please fill out the table below, which indicates the state of the lines X and Y in the local caches of P0 and P1 after each operation. We've given the first four rows for you as examples.

Operation	P0 X state	P1 X state	P0 Y state	P1 Y state
P0 LOAD X	S [MISS]	I	I	I
P0 LOAD X	S [HIT]	I	I	I
P1 STORE Y	S	I	I	M [MISS]
P1 STORE X	I	M [MISS]	I	M
P1 LOAD Y	I	M	I	M [HIT]
P1 STORE Y	I	M	I	M [HIT]
P1 LOAD Y	I	M	I	M [HIT]
P0 STORE X	M [MISS]	I [FLUSH]	I	M
P0 STORE Y	M	I	M [MISS]	I [FLUSH]

Cache Coherence and False Sharing

Problem 3. (40 points):

Consider the following program.

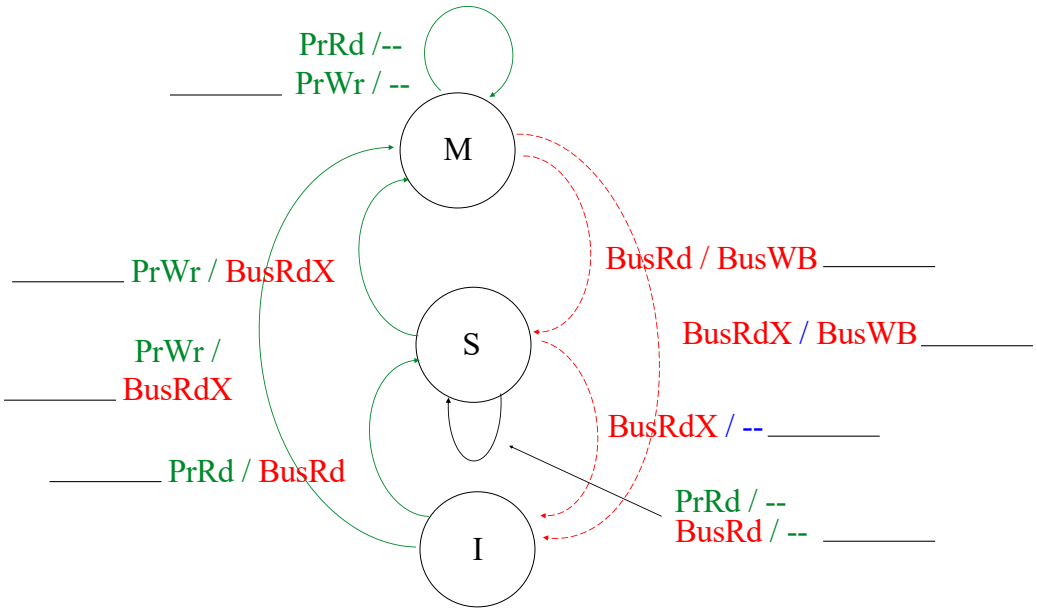
```
void worker(int* counter) {  // each thread runs this function

  barrier();

  // <--- invalidate caches: assume all lines in all caches are
  //      invalidated here after leaving barrier --->

  for (int i=0; i<NUM_ITERS; i++)
    (*counter)++;   // work here (load + incr + store)
}

void test(int num_threads) {
  std::thread threads[MAX_THREADS];
  int         counter[MAX_THREADS];  // Assume this is aligned on a cache line boundary

  for (int i=0; i<num_threads; i++)
    threads = std::thread(worker, &counter[i]); // spawn thread
  for (int i=0; i<num_threads; i++)
    threads[i].join();  // wait for thread to terminate
}
```

Consider calling `test()` with `num_threads`=2.

Assuming that code is run on a dual-core processor where:

- Each core has its own private cache
- Caches use the MSI protocol to implement coherence. For reference the MSI diagram is given on the next page.

PROBLEM CONTINUES ON NEXT PAGE...



In your answer to the following questions:

- You should only analyze references to the `counter` array
- Assume all arithmetic is free, you only need to think about memory operations in this problem.
- **Assume that the bus access protocol is such that one core executes the whole C statement involving both the memory read and then the memory write (see the comment "// work here") before allowing bus transactions from the other core.**
- **Assume that between iterations of the `for` loop by one core, the other core is able to execute one iteration of its for loop as well.**
- Each processor action (PrRd, PrWr shown in green) takes 1 cycle. (e.g., A cache hit on a PrWr takes 1 cycle)
- Each bus transaction (BusRd, BusRdX, BusWB shown in red) takes 10 cycles. (Specifically: a PrWr that generates BuxRdX takes a total of 1+10=11 cycles. A PrWr that generates a BusRdX that triggers a remote cache BusWB (write back) requires 1+10+10=21 cycles.)
- No other operations manipulate the state of caches
- Only consider operations that occur after the line <invalidate caches>

PROBLEM CONTINUES ON NEXT PAGE...

A. (20 pts) How many cycles does the memory system take to execute the worker for-loop on one of the threads with the following cache organization. On the figure, please list how many times the various coherence protocol transitions occur.
   - 16 byte cache size
   - Fully associative cache (any line can go anywhere, no conflict misses)
   - 4 byte cache line size

   Solution: in this case, after the initial cold read miss, and then the BusRdX for the write, all memory operations are cache hits. So there is 1 I->S transition, and 1 S->M transition. Therefore there are 2 operations that take 11 cycles, followed by NUM_ITERS-1 iterations that take 2 cycles each. So the total runtime is approximately 22 + 2 $\times$ (NUM_ITERS-1) cycles.

B. (20 pts) How many cycles does the memory system take to execute the worker for-loop on one of the threads with the following cache organization. On the figure, please list how many times the various coherence protocol transitions occur.
   - 16 byte cache size
   - Fully associative cache (any line can go anywhere, no conflict misses)
   - 8 byte cache line size

   Solution: the 8-byte cache line creates a situation of false sharing. Now the cache line containing `counter[0]` *and* `counter[1]` *ping pongs between the two caches. Each iteration through the loop includes a read miss, a write miss, and a write back by the remote processor. In total across the two threads there are 2 $\times$ NUM_ITERS I->S transitions, S->M, M->S, and S->I transitions. The total execution time is now 2 $\times$ (11+21) $\times$ NUM_ITERS cycles.*

PRACTICE PROBLEM 1: Angry Students

Your friend is developing a game that features a horde of angry students chasing after professors for making long exams. Simulating students is expensive, so your friend decides to parallelize the computation using one thread to compute and update the student's positions, and another thread to simulate the student's angriness. The state of the game's N students is stored in the global array `students` in the code below).

```
struct Student {
   float position;   // assume position is 1D for simplicity
   float angriness;
};

Student students[N];

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

void update_positions() {
   for (int i=0; i<N; i++) {
      students[i].position = compute_new_position(i);
   }
}

void update_angriness() {
   for (int i=0; i<N; i++) {
      students[i].angriness = compute_new_angriness(i);
   }
}

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

// ... initialize students here

std::thread t0, t1;
t0 = std::thread(update_positions);
t1 = std::thread(update_angriness);
t0.join();
t1.join();
```

Questions are on the next page...

A. Since there is no synchronization between thread 0 and thread 1, your friend expects near a perfect 2× speedup when running on two-core processor that implements invalidation-based cache coherence **using 64-byte cache lines**. She is shocked when she doesn't obtain it. Why is this the case? (For this problem assume that there is sufficient bandwidth to keep two cores busy – "the code is bandwidth bound" is not an answer we are looking for.) HINT: consider how data is laid out in memory in a C struct.

Solution: This is a classic false-sharing situation. Assuming the threads iterate through the array at equal rates (that is, they are on the same loop iteration at about the same time), both threads will be writing to elements on the same cache line at about the same time. The cache line will bounce back and forth between the caches of the two processors. In the worst case, every write is a miss.

B. Modify the program to correct the performance problem. You are allowed to modify the code and data structures as you wish, **but you are not allowed to change what computations are performed by each thread and your solution should not substantially increase the amount of memory used by the program.** You only need to describe your solution in pseudocode (compilable code is not required).

A simple solution is to change the data structure from an array of `Student` *structures to two arrays, one for each field. As a result, each thread works on its own array and scans over it contiguously.*

```
float position[N];
float angriness[N];
```

Some students mentioned that an alternative solution was to offset the position of the threads in the arrays to ensure that, at any one moment, each thread operating in distant parts of the array. One example was to have one thread iterate from i=0 to N, and the other iterate backwards from N-1 to 0. This solution eliminates the false sharing and was given full credit. It should be noted that the spatial locality of data access is not as good (by a factor of 2) in this scenario than for the solution described above since each thread only makes use of 1/2 of the data in each cache line it loads.

PRACTICE PROBLEM 2: Fusion, Fusion, Fusion

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.)

Solution: The code as written is bandwidth bound on computer B (12 bytes per math op, or arithmetic intensity of 1/12) for 9 of 10 operations. Running a full speed on computer B would require $4 \times 8 \times 12 = 384$ GB of BW, but it only has 128. Running at full speed on computer A would require $4 \times 4 \times 12 = 192$ GB of BW. Note that the `cs149math_sum()` operation has arithmetic intensity=1/4, and is compute bound on both machines (and thus runs better on machine B), but this is only 1/10 of the computation.

*However, the code can be restructured to have arithmetic intensity=1/4 everywhere. As a result, the computation is always compute bound (384 GB of BW * ((1/12)/(1/4)) = 128 GB of BW) and the best machine is machine B.*

```
#omp parallel for
for (int i=0; i<N; i++) {
   // 3 loads + 1 store + 4 math ops (AI = 4/16 = 1/4)
   // the variable t is stored in a register, so it doesn't contribute to loads and stores.
   float t = (src1[i] + src2[i]) * src3[i];
   x += t;
   tmp3[i] = t * src1[i];
}

if (x / N > 10.f) {
  #omp parallel for
  for (int i=0; i<N; i++) {
    // 2 loads + 1 store + 3 math ops (AI = 3/12 = 1/4)
    dst[i] = tmp3[i] * src1[i] + src1[i] + src1[i];
  }
} else {
  #omp parallel for
  for (int i=0; i<N; i++) {
    // 2 loads + 1 store + 2 math ops (AI = 3/12 = 1/4)
    dst[i] = (tmp3[i] + src2[i]) * src2[i] * src2[i];
  }
}
```

PRACTICE PROBLEM 3: Introducing PKPU2.0: The GPU for the Metaverse

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**?

Solution: 16 cores $\times$ 32 operations/clock $\times$ 1 GHz = 512 Giga-ops/sec (512 operations per clock). In other words, there are 512 single-precision ALUs running at 1 GHz. (and groups of 32 of those ALUs execute in SIMD lockstep)

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.

Solution: There is no divergence. All threads in the same warp operate on input data with the same sign, therefore all threads take the "if" path and there is no divergence.

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!)

Solution: 16 cycles of work per 256 bytes read. 256 comes from 32 threads $\times$ (4 bytes loaded + 4 bytes stored).

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?

Solution: Yes it can. When one warp issues a load, there are seven other warps $\times$ 16 instructions to cover the latency of the load. Therefore, given this input array, the PKPU2.0 can cover up to 112 cycles of memory latency.

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.**

Solution: It is bandwidth bound. Every 16 cycles each core will generate 32 $\times$ (4 bytes + 4 bytes) = 256 bytes of memory traffic. There are 16 cores, so that's 4096 bytes of memory traffic across the entire PKPKU2.0. 4096 $\times$ 1 GHz / 16 = 256 GB/sec of bandwidth. The processor achieves 50% utilization since it has half the bandwidth it needs.

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.)

Solution: Triple the memory bandwidth. You need to at least double the memory bandwidth to become compute bound. Previously the PKPU was memory bound and ran at approximately 50% efficiency. (It had half the bandwidth it needed to run at peak rate.) But adding bandwidth to become compute bound, you speed up the problem by $2\times$*. Since PKPKU could already hide all latency, adding executing contexts won't help performance. Nor does increasing the SIMD width, since the program is bandwidth bound. Note that even if the PKPKU had infinite bandwidth, doubling the SIMD width would not help very much since although* $2\times$ *more work could be done per clock, the core would now suffer from divergence, and therefore it would require almost* $2\times$ *more clocks to execute the "if" and "else" clauses of the code.*

Practice Problem 4: Bringing Locality Back

Justin Timberlake wants to get back in the news. 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?

   Solution: The program creates four RDDs, and since Justin's implementation elects to evaluate and materialize the contents RDD's immediately, the implementation will require up to 4 TB of memory to store all of these structures. There is only 1 TB of memory in aggregate across the 10 nodes of the cluster, so they will need to be stored and loaded from disk to implement each operation. The computation will be disk I/O limited. By increasing the cluster size to 40 nodes, there is now 4 TB of memory across the cluster, and the RDDs can be stored in memory and not out on disk. This will yield a significant performance improvement. Note: Even if the RDD were freed immediately after use, the system would need about 2 TB of memory and still require on-disk storage.

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.

Solution: Taylor is pointing out that the semantics of Spark RDDs permit a much more efficient implementation. The computation can be reordered to that instead of performing one operation on all elements of an RDD before proceeding to the new, the entire pipeline of operations can be performed on a chunk of the dataset all at once, with different chunks running in parallel on all the nodes. For example, an implementation could load 1 MB of lines of the input file from disk, run the entire sequence of RDD operations on this chunk (storing intermediate data in cache). This would be exceptionally memory efficient, and take advantage of the producer-consumer locality in the problem.

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.**

Solution: Assuming the elements of the RDDs are partitioned equally across the cluster, losing one node results in a loss of 1/10th of the output. Since the system partitioned the data across the cluster, the system knows exactly which partition of elements where lost. Given the lineage of the RDD `misspelled`, this partition can be recomputed in parallel across all remaining 9 nodes. If one node took 60 minutes to compute this partition, 9 nodes would so the work in 60/9 = 6.6 minutes. We also accepted 6 minutes as a perfectly valid answer, since if the failed node was replaced then the lost partition could be recomputed in 60/10=6 minutes.