Stanford CS149: Parallel Computing
Written Assignment 3

Improving locality on Specialized Hardware

Problem 1: (Graded for Correctness - 30 pts)

CS149 students are always excited to find new ways to optimize the locality of their parallel programs, and so on this final problem of the quarter, let's explore locality optimization in the context of creating specialized hardware.

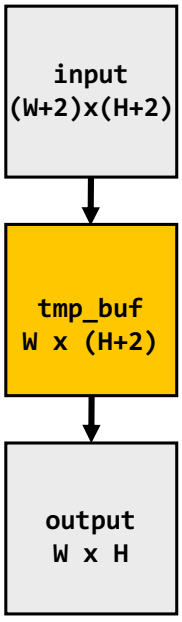
Consider the following C code for computing an image blur in two passes. The code first performs a 1D horizontal blur on each row of the input image, and then performs a 1D vertical blur on each column of the intermediate result stored in `tmp_buf`.

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.0/3, 1.0/3, 1.0/3};

// blur image horizontally
for (int j=0; j<(HEIGHT+2); j++)
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int ii=0; ii<3; ii++)
      tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
    tmp_buf[j*WIDTH + i] = tmp;
  }

// blur tmp_buf vertically
for (int j=0; j<HEIGHT; j++)
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
```



Questions on the next page...

A. What is the arithmetic intensity of the blur image code? Please answer in terms of floating point math ops per byte transferred from off-chip memory (by transferred we want you to consider both reads and writes. Assume a write of a `float` is 4 bytes of memory traffic). You can assume that the system has a cache that can retain a few rows rows of the input image (but not the entire input image), and **in your arithmetic intensity calculation, cache hits should not count as memory traffic.** Assume that the variables `weights` and `tmp` are in registers and generate no memory traffic.

[ ] ops per byte

B. (5 pts) Given a processor with memory bandwidth of 1 TB/sec and a peak arithmetic throughput of 1 TFLOPS, what is the peak performance (in terms of floating point operations per second) that this processor can achieve on the specific blur code given above? (It's fine to treat 1 TB as 10^{12} bytes instead of 1024^4 bytes). **Hint: what constrains the program's performance?**

TFLOPS

Now consider developing a specialized hardware implementation of image blur. Your implementation will use two threads that communicate given a special storage module called a line buffer.

```
// line buffer with storage for numRows rows and numColumns columns
LINEBUFF<T> name[numRows][numColumns];
```

A `LINEBUFF` is a special on-chip data storage module that data for holds `numRows` rows of an image that is `numCols` pixels wide. The line buffer supported enqueuing of individual pixels, and dequeuing of entire rows. Internally, you can think about a line buffer as having state `(cur_column, cur_row)` that marks the next address to add data to on an enqueue. An "API" for a line buffer is given below.

```
// reset buffer's (cur_column, cur_row) cursors to 0
line_buffer.reset();

// *** enq will stall the calling thread if line_buffer is full ***
// if the line_buffer is not full, add data to buffer at the
// position given by internal line buffer state (cur_column, cur_row)
// if cur_column post-increment == numColumns, then:
//   - set cur_column = 0
//   - set cur_row = cur_row + 1;
line_buffer.enq(T data);

// Return the element at position given by (col, row)
T line_buffer.access(int col, int row)

// Discard the oldest row, shift all rows
// Line 0 is discarded, line 1 becomes line 0, line 2 becomes line 1, etc.
// Also set cur_row = cur_row - 1
// *** deqline will stall the calling thread if the line_buffer is not full ***
line_buffer.deqline()

// will stall the calling thread until the line buffer contains 'rows'
// complete rows of pixels. This is the case when cur_row >= rows.
//
// For example line_buffer.waitforrows(1) blocks until the line buffer
// has at least one full row of pixels
line_buffer.waitforrows(int rows)
```

Question is on the next page...

C. (15 pts) Use `LINEBUFF` to improve the performance of image blur. Assume that the `for` loops for horizontal and vertical image blur will execute *concurrently* in different threads, and that communication between the threads should be via inserting and removing data from the line buffer module. Notice that calls to `enq()`, `deqline()`, and `waitforfull()` "block" (a.k.a. stall the calling thread) until conditions are met for them to return to the caller. In your pseudocode, be sure to specifically give the size of your `LINEBUFFER`. You should choose the **minimum size (in number of rows) that allows for a solution that maximizes arithmetic intensity, and in steady state ALLOWS both threads to always be making forward progress (assuming both threads generate pixels at the same rate).**

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.0/3, 1.0/3, 1.0/3};

// *** TODO by student: choose the LINEBUFF's size parameters ***
LINEBUFF<float>line_buf[       ][       ];

// give thread 1 code here ////////////////////////////////
```

```
// give thread 2 code here ////////////////////////////////
```

D. (5 pts) Assuming that storage for the line buffer is “on chip" and does not require reads and writes to memory, what is the arithmetic intensity of your hardware implementation?

operations per byte

E. (5 pts) Given a dual-core processor with memory bandwidth of 1 TB/sec and a peak of 1 TFLOPS, what is the peak performance this processor can achieve on the blur code?

TFLOPS

Two Box Blurs are Better Than One

Problem 2: (Graded for Correctness - 35 pts)

Consider the program below, which runs two back-to-back convolutions with a `FILTER_SIZE` by `FILTER_SIZE` filter.

```
float input[HEIGHT][WIDTH];
float temp[HEIGHT][WIDTH];
float output[HEIGHT][WIDTH];

float weight;   // assume initialized to (1/FILTER_SIZE)^2

void convolve(float output[HEIGHT][WIDTH], float input[HEIGHT][WIDTH], float weight) {

  for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
      float accum = 0.f;
      for (int jj=0; jj<FILTER_SIZE; jj++) {
        for (int ii=0; ii<FILTER_SIZE; ii++) {

           // ignore out-of-bounds accesses (assume indexing off the end of image is
           // handled by special case boundary code (not shown)

           // count as one math op (one multiply add)
           accum += weight * input[j-FILTER_SIZE/2+jj][i-FILTER_SIZE/2+ii];
        }
      }
      output[j][i] = accum;
    }
  }
}

// run two convolutions back to back
convolve(temp, input, weight);
convolve(output, temp, weight);
```

A. (5 pts) Assume the code above is run on a processor that can comfortably store `FILTER_SIZE*WIDTH` elements of an image in cache, so that when executing `convolve` each element in the input array is loaded from memory exactly once. What is the arithmetic intensity of the program, in units of math operations per element transferred from memory?

Many times in class Prof. Kayvon emphasized the need to increase arithmetic intensity by exploiting producer-consumer locality. But sometimes it is tricky to do so. Consider an implementation that attempts to double arithmetic intensity of the program above by producing 2D chunks of output at a time. Specifically the loop nest would be changed to the following, **which now evaluates BOTH CONVOLUTIONS.**

```
for (int j=0; j<HEIGHT; j+=CHUNK_SIZE) {
  for (int i=0; i<WIDTH; i+=CHUNK_SIZE) {

    float temp[..][..];  // you must compute the size of this allocation in 6B

    // compute required elements of temp here (via convolution on region of input)

    // Note how elements in the range temp[0][0] -- temp[FILTER_SIZE-1][FILTER_SIZE-1] are the temp
    // inputs needed to compute the top-left corner pixel of this chunk

    for (int chunkj=0; chunkj<CHUNK_SIZE; chunkj++) {
      for (int chunki=0; chunki<CHUNK_SIZE; chunki++) {
        int iidx = i + chunki;
        int jidx = j + chunkj;
        float accum = 0.f;
        for (int jj=0; jj<FILTER_SIZE; jj++) {
          for (int ii=0; ii<FILTER_SIZE; ii++) {
            accum += weight * temp[chunkj+jj][chunki+ii];
          }
        }
        output[jidx][iidx] = accum;
      }
    }
  }
}
```

B. (8 pts) Give an expression for the number of elements in the `temp` allocation.

C. (7 pts) Assuming `CHUNK_SIZE` is 8 and `FILTER_SIZE` is 5, give an expression of the **total amount of arithmetic performed per pixel of output** in the code above. You do not need to reduce the expression to a numeric value.

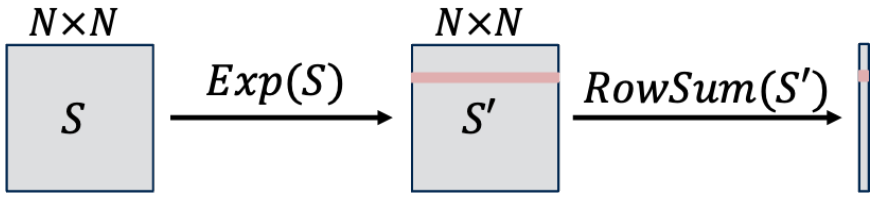
D. (10 pts) Will the transformation given above improve or hurt performance if the original program from part A was *compute bound* for this `FILTER_SIZE`? Why?

E. (5 pts) Why might the chunking transformation described above be a useful transformation in an energy constrained processing setting regardless of whether it notably impacts performance?

Designing Hardware for Softmax

Problem 3: (Graded on Effort Only - 30 pts)

CS149 students are always excited to explore techniques for improving the performance of DNN algorithms. Let's analyze the performance of the exponential and rowsum components of softmax on an AI accelerator.



Here is the code for the accelerator:

```
auto A = IO_REGION("A", (N, N), float);
auto B = O_REGION("B", (N), float);
auto a_tile_shape = std::vector<float>({1, N});   // one row of the matrix

for (i= 0; i < N; i++){
    auto a_tile = LOAD_TILE(A, i, a_tile_shape); // dense load of a_tile from A
    auto out = EXP(a_tile); // EXP calculates exp(i,j) for the tile
    auto out_tile = BUFFER(out);
    STORE_TILE(A, i, out_tile);  // dense store of out_tile to A
}

auto out_sum(N) // length N vector
for (i=0; i < N; i++){
    auto a_tile = LOAD_TILE(A, i, a_tile_shape); // dense load of a_tile from A
    auto out_sum[i] = row_sum(a_tile);  // calculate rowsum for the tile
}
auto out_tile = BUFFER(out_sum);  // just a wrapper, no cost
STORE_TILE(B, out_tile);      // dense store of out_tile to B
```

A. (6 pts) What is the FLOP arithmetic intensity of the EXP loop? Assume that `EXP` counts as 10 FLOPs **per element of a tile**.

[ ] float ops/byte

B. (6 pts) Given a chip with memory bandwidth of 10 MB/sec and a peak of 10 MFLOPS, what is the execution time (in seconds) of the **entire code** for N=1000? Please show calculations. Ignore the final store to B. As before assume that `EXP` counts as 10 FLOPs **per element of a tile**. `row_sum` performs 1 FLOP **per element of a tile**. Please show calculations.

seconds

C. (6 pts) You are unsatisfied with this performance and decide to improve it. Provide modified code that improves the performance. Do not use metapipelining constructs, just regular `for` loops in your solution. (Hint: arithmetic intensity)

D. (6 pts) Given a chip with memory bandwidth of 10 MB/sec and a peak of 10 MFLOPS for N=1000, what is the execution time (in seconds) of the **entire code** for N=1000 for your improved code? Ignore the final store to B. As before assume that `EXP` counts as 10 FLOPs **per element of a tile** and `row_sum` performs 1 FLOP **per element of a tile**. Please show calculations.

seconds

E. (6 pts) You are still not happy with this performance and so you decide to further optimize the code using meta-pipelining. Show your improved code.

F. (6 pts) Given a chip with memory bandwidth of 10 MB/sec and a peak of 10 MFLOPS, for N = 1000, what is the execution time (in seconds) of the **entire code** when running your improved code with metapipelining? Ignore the final store to B. As before assume that `EXP` counts as 10 FLOPs **per element of a tile** and `row_sum` performs 1 FLOP **per element of a tile**. Please show calculations.

[ ] seconds

An Exercise in Data-Parallel Thinking

PRACTICE PROBLEM 1:

Assume you are given a library that can execute a bulk launch of N independent invocations of an application-provided function using the following CUDA-like syntax:

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

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

```
void foo(int* x) {
   printf("Instance %d : %d\n", id, x[id]);
}
int A[] = {10,20,30}
foo<<<3>>>(A);

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

The library also provides the data-parallel function `exclusive_scan` (using the + operator) that works as discussed in class.

```
exclusive_scan(N, in, out);

Example usage:
N    = 6
in   = {1, 2, 3, 4, 5, 6}
====================================
out  = {0, 1, 3, 6, 10, 15}
```

In this problem, we'd like you to design a data-parallel implementation of `largest_segment_size()`, which, given an array of flags that denotes a partitioning of an array into segments, computes the size of the longest segment in the array.

```
int largest_segment_size(int N, int* flags);
```

The function takes as input an array of N flags (`flags`) (with 1's denoting the start of segments), and returns the size of the largest segment. The first element of flags will always be 1. For example, the following flags array describes five segments of lengths 4, 2, 2, 1, and 1.

```
N       = 10
flags   = {1,  0,  0,  0,  1,  0, 1,  0, 1,  1}
=============================================
result:  = 4
```

Questions on next page...

A. The first step in your implementation should be to compute the size of each segment. Please use the provided library functions (bulk launch of a function of your choice + `exclusive_scan` to implement the function `segment_sizes()` below. *Hint: We recommend that you get a basic solution done first, then consider the edge cases like how to compute the size of the last segment.*

```
// Example output of segment_sizes(N, flags, num_segs, sizes):
//   N        =  8
//   flags    =  {1, 0, 1, 0, 0, 0, 1, 0}
//   ==================================
//   num_segs = 3
//   sizes    = {2, 4, 2}


// you may wish to define functions used in bulk launches here









// You can allocate any required intermediate arrays in this function
// You may assume that 'seg_sizes' is pre-allocated to hold N elements,
// which is enough storage for the worse case where the flags array
// is all 1's.
void segment_sizes(int N, int* flags, int* num_segs, int* seg_sizes) {









}
```

B. Now implement `largest_segment_size()` using `segment_sizes()` as a subroutine. **NOTE: this problem can be answered even without a valid answer to Part A.** Your implementation may assume that the number of segments described by `flags` is always a power of two. A full credit implementation will maximize parallelism and minimize work when computing the maximum segment size from an array of segment sizes. *Hint: we are looking for solutions with `lg2(num_segs)` span.*

```
// you may want to implement helper functions here that are called via bulk launch
```

```
int largest_segment_size(int N, int* flags) {

  int num_segs;
  int seg_sizes[N];
  segment_sizes(N, flags, &num_segs, seg_sizes);

}
```

Async Message Ping Pong

PRACTICE PROBLEM 2:

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() {









}
```

Introducing PKPU2.0: The GPU for the Metaverse

PRACTICE PROBLEM 3:

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.)

An Interesting CUDA Program

Problem 4: (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?

PRACTICE PROBLEM 4:

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) {
  // We haven't talked about the OpenMP (OMP) extension to C in
  // class, but the following 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 a thread pool
  #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.)

Data-Parallel Grid Solver

Problem 5: (Graded on Effort Only - 18 pts)

Let's look at how the ideas of data-parallel programming and pipelined execution can be used to improve the performance of the Grid-solver application discussed in class.

Consider the following simplified Grid Solver code from Lecture 4 for just the **red-cell update**. Recall that in the grid solver application there was a 2D grid of cells, with a checkerboard pattern of red and black cells. Therefore, 1/2 the cells were red and 1/2 the cells were black. This is essentially the same code as from the lecture. (Note: do not worry about array boundary conditions.)

```
const int N;
float* A = allocate(N, N);   // allocate grid

void simple_solve_red(float* A, int iter_max) {
   for (i=0; i < iter_max; i++){
     for_all (red cells (i,j)) {
         A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] +
                          A[i+1,j] + A[i,j+1]);
     }
}
```

A. (6 pts) What is the arithmetic intensity of the grid solver code (floating point ops performed per byte transferred to/from memory)? Assume that `i` and `j` are in registers and the only data transfer comes from reads/writes to the array `A`.

[ ] FLOPS/byte

B. (6 pts) You decide that you want to see how the code would run on a reconfigurable dataflow architecture (like the SambaNova SN40L discussed in class). You recode the simple Grid Solver code as follows

```
auto A = IO_REGION("A", (N, N), float);   // NxN cell region of memory
auto NN =  100;                           // tile size along N,
auto a_tile_shape = std::vector<int64_t>({NN, N});

for (i= 0; i < iter_max; i++){
    LOOP(N / NN, [&]() {      //sequential loop
        auto a_tile = LOAD_TILE(A, a_tile_shape); // dense load of a_tile from A
        auto out = STENCIL(a_tile);    // STENCIL calculates new A[i,j]s for tile
        auto out_tile = BUFFER(out);   // no work, prepares 'out' for transfer
        STORE_TILE(A, out_tile);       // dense store of out_tile to A
    });
}
```

Given a chip with memory bandwidth of 4 MB/sec and a peak floating point arithmetic throughput of 1 MFLOPS for `N`=1000, how long **(in seconds)** will it take to execute one iteration of the `i` loop? (This involves 10 iterations of `LOOP`.) Please show your work. (Hint: determine the run times for the individual components of the loop.)

	seconds

C. (6 pts) You are unsatisfied with this performance and want to ascend the leader board. You decide to write an optimized Grid Solver kernel using meta-pipelining. You come up with the following code. **Reminder: please assume that the stages of the `METAPIPE` block execute as a pipeline!**

```
auto A = IO_REGION("A", (N, N), float);
auto NN =  100; // Tile size along N,
auto a_tile_shape = std::vector<int64_t>({NN, N});

for (i= 0; i < iter_max; i++){
    METAPIPE(N / NN, [&]() {
        auto a_tile = LOAD_TILE(A, a_tile_shape); // dense load of a_tile from A
        auto out = STENCIL(a_tile);  // STENCIL calculates new A[i,j]s for tile
        auto out_tile = BUFFER(out); // no work, prepares 'out' for transfer
        STORE_TILE(A, out_tile);     // dense store of out_tile to A
    });
}
```

Given a chip with memory bandwidth of 4 MB/sec and a peak arithmetic throughput of 1 MFLOPS, for `N` = 1000, how long will it take to execute one iteration of the `i` loop. (This involves 10 iterations of `METAPIPE`.) Briefly describe how you computed this answer. (Hint: what limits the overall throughput of the pipeline?)

[ ] seconds

Paparazzi Camera

PRACTICE PROBLEM 5:

You are designing a heterogeneous multi-core processor to perform real-time "celebrity detection" on a future camera. The camera will continuously process low-resolution live video and snap a high-resolution picture whenever it identifies a subject in a database of 100 celebrities. Pseudocode for its behavior is below:

```
void process_video_frame(Image input_frame)
{
  Image face_image = detect_face(input_frame);
  for (int i=0; i<100; i++)
     if (match_face(face_image, database_face[i]))
        take_high_res_photo();
}
```

In order to not miss the shot, the camera MUST call `take_high_res_photo` within 500 ms of the start of the original call to `process_video_frame`! To keep things simple:

- Assume the code loops through all 100 database images regardless of whether a match is found (e.g., we want to find all matches).
- The system has plenty of bandwidth for any number of cores.

Two types of cores are available to use in your chip. One is a fixed-function unit that accelerates `detect_face`, the other is a general-purpose processor. The cost (in chip resources) of the cores and their performance (in ms) executing important functions in the pseudocode are given below:

Operation	Core Type	
	C1 (fixed-function)	C2 (Programmable)
Resource Cost	1	1
Perf (ms): `detect_face`	100	400
Perf (ms): `match_face`	N/A	20

A. Assume a video frame arrives exactly every 500 ms. **If you only use cores of type C2**, how many cores do you need to meet the performance requirement for the video stream? (You cannot change the algorithm, and please justify your answer).

B. Your team has just built a multi-core processor that contains a large number of cores of type C2. It achieves 5.9× speedup on the camera workload discussed above. Amdahl's Law says that the maximum speedup of the camera pipeline in this problem should be $2400/400 = 6\times$, so your team is happy. They are shocked when your boss demands a speedup of $10\times$. Your team is on the verge of quitting due "unreasonable demands". How do you argue to them that the goal is reasonable one if they consider all the possibilities in the above table? (Hint: What assumption are they making in their Amdahl's Law calculations, and why does it not hold?)

.

C. Now assume you can use both cores of types C1 and C2 in your design. How many of each core do you choose to minimize resource usage, while still meeting the same performance requirements as in part A? Does your new chip use more or fewer total resources than your solution in part A (by how much)?

D. Beyonce is about to release a new album, and your paparazzi customers want to follow her around all day. They request a camera that is more energy efficient. Energy efficiency is so important they are willing to relax their performance requirement and allow high-res photos to be taken within 1500 ms (not 500 ms) of a video frame arriving. You find that the primary consumer of power in your application is loading database faces from memory. Describe how you would change the pseudocode to **approximately double** the energy efficiency of the camera while still meeting the performance requirements. You should assume you use the same processor design as in part C (or part A for that matter), and that your processor has a cache that holds up to 4 database images.