Stanford CS149: Parallel Computing
Written Assignment 4

Two Box Blurs are Better Than One

Problem 1: (Graded for Correctness - 50 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. (10 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 load?

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

C. (10 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. (10 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?

An Exercise in Data-Parallel Thinking

Problem 2: (Graded on Effort Only - 50 pts)

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. (25 pts) 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. (25 pts) 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);

}
```

Paparazzi Camera

PRACTICE PROBLEM 1:

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.

Improving locality on Specialized Hardware

PRACTICE PROBLEM 2:

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.

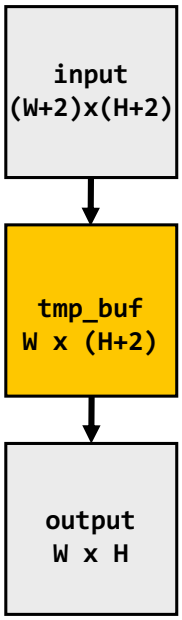
Consider the following C code for computing an image blur in two passes. The code first performs a 1D horizontal blur on each row of the image, and then performs a 1D vertical blur on each column of the result.

```
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;
  }
```



A. What is the arithmetic intensity of the blur image code? Please answer in terms of floating point math ops per byte read from off-chip memory. You can assume that the system has a cache that can retain many rows of the input image, and **in your arithmetic intensity calculation, cache hits should not count as memory traffic.** Assume that `weights` and `tmp` are in registers and generate no memory traffic.

[ ] ops per byte

B. 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 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. 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 use the line buffer module. Notice that calls to `enq()`, `deqline()`, and `waitforfull()` block (aka 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. 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. 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