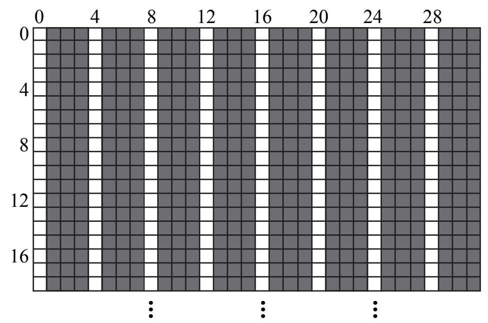
CMU 15-418/618: Exam 1 Practice Exercise SOLUTIONS

1 Buying a New Computer

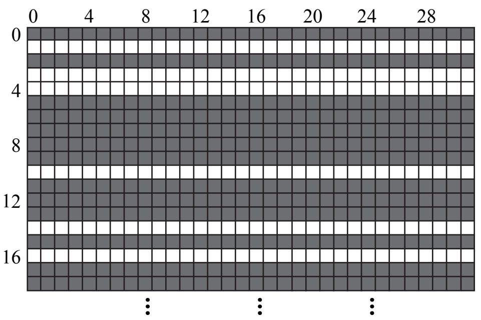
You write a bit of ISPC code that modifies an grayscale image with width 32 and height `height`. The modification is controlled by the contents of a black and white "mask" image of the same size. The code brightens input image pixels by a factor of 1000 if the corresponding pixel of the mask image is white (the mask has value 1.0) and by a factor of 10 otherwise.

The code partitions the image processing work into 64 ISPC tasks, which you can assume balance perfectly onto all available CPU processors.

```
void brighten_image(uniform int height, uniform float image[], uniform float mask_image[])
{
    uniform int NUM_TASKS = 64;
    uniform int rows_per_task = height / NUM_TASKS;
    launch[NUM_TASKS] brighten_chunk(rows_per_task, image, mask_image);
}


void brighten_chunk(uniform int rows_per_task, uniform float image[], uniform float mask_image[])
{
     //   'programCount' is the ISPC gang size.
     //   'programIndex' is a per-instance identifier between 0 and programCount-1.
     //   'taskIndex' is a per-task identifier between 0 and NUM_TASKS-1

     // compute starting image row for this task
     uniform int start_row = rows_per_task * taskIndex;

     // process all pixels in a chunk of rows
     for (uniform int j=start_row; j<start_row+rows_per_task; j++) {
       for (uniform int i=0; i<32; i+=programCount) {

         int idx = j*32 + i + programIndex;
         int iters = (mask_image[idx] == 1.f) ? 1000 : 10;

         float tmp = 0.f;
         for (int j=0; j<iters; j++)
             tmp += image[idx];

         image[idx] = tmp;
       }
     }
}
```



Mask Image 1: 32 x height
(vertical white columns every 4th pixel)



Mask Image 2: 32 x height
(random black or white rows)

Figure 1: Image masks used to govern image manipulation by `brighten_image`

You go to the store to buy a new CPU that runs this computation as fast as possible. On the shelf you see the following three CPUs on sale for the same price:

(A) 4 GHz *single core* CPU capable of performing one floating point addition per clock (no parallelism)

(B) 1 GHz *single core* CPU capable of performing one 32-wide SIMD floating point addition per clock

(C) 1 GHz *dual core* CPU capable of performing one 4-wide SIMD floating point addition per clock

A. If your only use of the CPU will be to run the above code as fast as possible, and assuming the code will execute using mask image 1 above, rank all three machines in order of performance (from best to worst). Please explain how you determined your ranking by comparing execution times on the various processors. When considering execution time, you may assume that (1) the only operations you need to account for are the floating-point additions in the innermost loop. (2) The ISPC gang size will be set to the SIMD width of the CPU. (3) There are no stalls during execution due to data access.

(Hint: it may be easiest to consider the execution time of each row of the image.)

Answer: B > A > C

For image 1, each row of the mask is mixture of white and black pixels: every 1 white pixel is followed by 3 black pixels. Since the SIMD width for CPUs B and C are 32 and 4 respectively these processors will suffer from ***branch divergence*** *when executing this ISPC code. ISPC program instances working on a black pixel will wait for gang instances assigned white pixels to finish their execution of 1000 loop iterations.*

Now let's calculate how many cycles it takes for each processor to finish rendering a single row:

- A: $10 \times 24 + 1000 \times 8 = 8240$ cycles
- B: 1000 cycles
- C: $8 \times 1000 = 8000$ cycles

However, processor A is clocked at 4 GHz and processor C has 2 cores (so its throughput is doubled). Thus the effective per-row cost for each platform, normalized to 1000 cycles of a 1 GHz single core processor, will be:

- A: $8240 \div 1000 \div 4 = 2.06$
- B: $1000 \div 1000 = 1$
- C: $8000 \div 1000 \div 2 = 4$

So B performs better than A and A (despite executing entirely in serial) performs better than C.

B. Rank all three machines in order of performance for mask image 2? Please justify your answer, but you are not required to perform detailed calculations like in part A.

Answer: B > C > A

*In image 2, unlike image 1, all the rows are homogeneous. As a result, means processor B and C no longer suffer from **branch divergence**. Since all processor execute at their peak rates, the processor with the most raw processing power will provide the best processing speed. B provides the most raw processing power, followed by C.*

2 Buying a New Computer, Again

You plan to port the following sequential C++ code to ISPC so you can leverage the performance benefits of modern parallel processors.

```
float input[LARGE_NUMBER];
float output[LARGE_NUMBER];
// initialize input and output here ...

for (int i=0; i<LARGE_NUMBER; i++) {
   int iters;
   if (i % 16 == 0)
      iters = 256;
   else
      iters = 8;
   for (int j=0; j<iters; j++)
      output[i] += input[i];
}
```

Before sitting down to hack, you go the store, and see the following CPUs all for the same price:

- 4 GHz single core CPU capable of performing one floating point addition per clock (no parallelism)
- 1 GHz quad-core CPU capable of performing one 4-wide SIMD floating point addition per clock
- 1 Ghz dual-core CPU capable of performing one 16-wide SIMD floating point addition per clock

If your only use of the CPU will be to run your future ISPC port of the above code as fast as possible, which machine will provide the best performance for your money? Which machine will provide the least? Please explain why by comparing expected execution times on the various processors. When considering execution time, you may assume that (1) the only operations you need to account for are the floating-point additions in the innermost loop. (2) the ISPC gang size will be set to the SIMD width of the CPU.

(Hint: consider the execution time of groups of 16 elements of the input and output arrays).

Solution: To compare the three machines for the specific workload we will calculate how many groups of 16 elements each machine can process.

- *4 Ghz single core CPU: A group of 16 elements will take 376 (15*8+256) cycles on the single core system. Given the 4 GHz clock rate this machine can process* $4/376 * 10^9$ *groups of 16 elements per second.*
- *1 GHz quad-core 4-wide SIMD CPU: 16 elements will be scheduled in 4 sets of 4 elements on the cores. The first group requires 256 iterations through the loop (256 cycles), and incurs the inefficiency of divergent execution (3 of the 4 lanes only require 8 iterations). The last three groups suffer no divergence, and require 8 iterations (8 cycles) through the inner loop. Therefore each group of 16 elements requires (3*8 + 256) cycles. There are a total of four cores operating simultaneously at 1 GHz, so the machine can process* $(4/280) * 10^9$ *groups of 16 elements per second.*
- *1 GHz dual-core 16-wide SIMD CPU: 16 elements will take 256 cycles to execute on each core (every group must have to wait for the long-running element, so the 16-wide machine suffers from significant divergence). There are two 1GHz cores, so the machine can process* $(2/256) * 10^9$ *groups of 16 elements per second.*

According to these results, the best performing core for this workload is the 1 GHz quad-core 4-wide SIMD CPU. The 1 GHz dual-core CPU will provide the lowest performance.

3 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 1D position 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

pthread_t t0, t1;
pthread_create(&t0, NULL, updatePositions, NULL);
pthread_create(&t1, NULL, updateAngriness, NULL);
pthread_join(t0, NULL);
pthread_join(t1, NULL);
```

A. Since there is no synchronization between thread 0 and thread 1, your friend expects near a perfect $2\times$ speedup when running on two-core processor that implements invalidation-based cache coherence. 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.)

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.

4 Oh, the Students Remain Angry

Due to the great success of the hit iPhone app "Angry Students", your professors decide to release "Angry Students 2: They are Still Angry", which uses ISPC to take advantage of the SIMD instructions on the iPhone's ARM processor. The code is written like this:

```
struct Student {
  float position;
  float angriness;
};

Student students[N];

// ispc function
void updateStudents(int N, Student* students) {
  foreach (i = 0 ... N) {
    students[i].position = compute_new_position(i);
    students[i].angriness = compute_new_angriness(i);
  }
}
```

Performance is lower than expected, so the professors change the code to this:

```
float positions[N];
float angriness[N];

// ispc function
void updateStudents(int N, float* positions, float* angriness) {
  foreach (i = 0 ... N) {
    position[i] = compute_new_position(i);
    angriness[i] = compute_new_angriness(i);
  }
}
```

The resulting code runs significantly faster. Why?

Solution: The first version requires an expensive scatter and gather instruction where as the second version can perform the same vector instruction on contiguous floats in memory. This is the difference between having an array of structs versus a struct of arrays. (There is no struct of arrays in this case but you could imagine a Students struct that holds the positions and angriness arrays).

5 Parallel Histogram Generation

Your friend implements the following parallel code for generating a histogram from the values in a large input array `input`. For each element of the input array, the code uses the function `bin_func` to compute a "bin" the element belongs to (`bin_func` always returns an integer between 0 and `NUM_BINS-1`), and increments a count of elements in that bin. His port targets a small parallel machine with only two processors. *This machine features 64-byte cache lines and uses an invalidation-based cache coherence protocol.* Your friend's implementation is given below.

```
float input[N];                 // assume input is initialized and N is a very large
int   histogram_bins[NUM_BINS];   // output bins
int   partial_bins[2][NUM_BINS];  // assume bins are initialized to 0
                                // assume partial_bins is 64-byte aligned

////////////////////////// Code executed by thread 0 //////////////////////////
for (int i=0; i<N/2; i++)
   partial_bins[0][bin_func(input[i])]++;

barrier();  // wait for both threads to reach this point

for (int i=0; i<NUM_BINS; i++)
    histogram_bins[i] = partial_bins[0][i] + partial_bins[1][i];

////////////////////////// Code executed by thread 1 //////////////////////////
for (int i=N/2; i<N; i++)
   partial_bins[1][bin_func(input[i])]++;

barrier();  // wait for both threads to reach this point
```

A. Your friend runs this code on an input of 1 million elements (`N`=1,000,000) to create a histogram with eight bins (`NUM_BINS`=8). He is shocked when his program obtains far less than a linear speedup, and glumly asserts believe he needs to completely restructure the code to eliminate load imbalance. You take a look and recommend that he not do any coding at all, and just create a histogram with 16 bins instead. Explain why.

   *Solution: With a 64-bit-wide cache line and 8 bins, the `partial_bins` arrays for each thread lie on the same cache line (8 integers = 32 bytes). As a result, although there is no data sharing between the two threads when computing the partial results, significant **false sharing** will occur. Increasing the number of bins to 16 causes, `partial_bins[0]` and `partial_bins[1]` to reside on a separate cache lines, and false sharing is eliminated. It could also be noted that there is very little load imbalance in the current solution. The threads each process 500,000 elements in parallel. Then the serial part of the code is a simple summation of eight numbers.*

B. Inspired by his new-found great performance, your friend concludes that more bins is better. He tries to use the provided code from part A to compute a histogram of 10,000 elements with 2,000 bins. He is shocked when the speedup obtained by the code drops. Improve the existing code to scale near linearly with the larger number of bins. (Please provide pseudocode as part of your answer – it need not be compilable C code.)

   Now, with a large number of bins (and fewer total elements), the serial combination of the partial results is a significant fraction (20%) of execution time, significantly limiting speedup! Correct solutions sought to parallelize the combination step. Example code is given below:

```
float input[N];                   // assume input is initialized and N is a very large
int   histogram_bins[NUM_BINS];   // output bins
int   partial_bins[2][NUM_BINS];  // assume bins are initialized to 0

////////////////////////// Code executed by thread 0 //////////////////////////

for (int i=0; i<N/2; i++) {
   partial_bins[0][bin_func(input[i])]++;
}

barrier();  // wait for both threads to reach this point

for (int i=0; i<NUM_BINS/2; i++) {
    histogram_bins[i] = partial_bins[0][i] + partial_bins[1][i];
}


////////////////////////// Code executed by thread 1 //////////////////////////

for (int i=N/2; i<N; i++) {
   partial_bins[1][bin_func(input[i])]++;
}

barrier();  // wait for both threads to reach this point

for (int i=NUM_BINS/2; i<NUM_BINS; i++) {
    histogram_bins[i] = partial_bins[0][i] + partial_bins[1][i];
}
```

C. Your friend changes `bin_func` to a function with *extremely high arithmetic intensity*. (The new function requires 100000's of instructions to compute the output bin for each input element). If the histogram code **provided in part A** is used with this new `bin_func` do you expect scaling to be better, worse, or the same as the scaling you observed using the old `bin_func` in part A? Why? (Please ignore any changes you made to the code in part B for this question.)

Solution: Yes. Scaling is likely to be better. With an extremely high arithmetic intensity `bin_func`, most instructions are non-memory instructions. The execution time will be dominated by `bin_func` and not the cost of the increment of the bin counter. As a result, the effect of false sharing (which still exists here as it did in part A) will be negligible.

6 Reduction with CUDA

You want to write code to sum all of the elements in a vector of length N. You consider two options: *binary* reduction and *square* reduction.

The kernel for binary reduction reduces the size of the vector by a factor of 2 on each step:

```
// Parameter N2 = ceil(N/2.0)
__global__ void
binaryReduceKernel(int N, int N2, float *src, float *dst) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float val;
    if (i < N2) {
        val = src[i];          // 1 cycle
        if (i+N2 < N)
            val += src[i+N2];  // 1 cycle
        dst[i] = val;          // 1 cycle
    }
}
```

The kernel for square reduction treats the vector as an $M \times M$ matrix where $M = \left\lceil \sqrt{N} \right\rceil$, and reduces the vector to M elements by summing each row in the array:

```
// Generate position of matrix element i,j
#define RM(i,j,W) ((i)*(W)+(j))

// Parameter M = ceil(sqrt(N))
__global__ void
squareReduceKernel(int N, int M, float *src, float *dst) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float val = 0.0;
    for (int j = 0; j < M; j++) {
        int idx = RM(i,j,M);
        if (idx < N)
            val += src[idx];  // 1 cycle
    }
    if (i < M)
        dst[i] = val;         // 1 cycle
}
```

The general scheme for both kernels is to apply them repeatedly until it becomes better to do a serial summation:

```
    for (; N >= NMIN; N = M) {
        M = next(N);          // Either ceil(N/2.0) or ceil(sqrt(N))
        ... reduce vec from N to M ...
    }
    return serialReduce(vec, N);   // N cycles
```

A. What is the minimum value that can be used for NMIN for performing binary reduction? (and still get a correct result)

Solution: 2

B. What is the minimum value that can be used for `NMIN` for performing square reduction? (and still get a correct result)

Solution: 3. Since sqrt(2) = 1.414, which rounds up to 2, trying to set NMIN to 2 would result in an infinite loop.

C. In experimenting with the square reduction code, you replace the computation `idx = RM(i,j,M)` with `idx = RM(j,i,M)`, so that it computes the sum of each column. The new code runs faster. Explain how this could be.

Solution: Each thread in a warp will sum adjacent columns. On each step, they will be able to use a single block read to read in all of the array elements.

D. Imagine a hypothetical GPU with the following properties:

- It has one warp of 32 functional units operating in SIMD mode.
- A thread launch by the host requires 20 cycles.
- A memory read or write by the entire warp requires 1 cycle.
- Copying an array of size K from the device to the host and computing the sum of its elements serially on the host requires K cycles.
- All other operations require no cycles.

(a) What would be the cost incurred by reducing a vector of length $N = 1024$ using square reduction, for an optimal setting of parameter `NMIN`? **(By optimal we mean the stopping criterion that yields the highest performance.)** Show your work by listing a sequence of steps, describing what would happen with each step and how many cycles it would require.

Solution: The solution would proceed as follows:

i. Reduce from 1024 to 32 with one kernel launch. Requires 20 + 33 cycles.

ii. Serially reduce the rest. Requires 32 cycles

Total = 20 + 33 + 32 = 85 cycles

(b) What would be the cost incurred by reducing a vector of length $N = 1024$ using binary reduction, for an optimal setting of parameter `NMIN`? Show your work by listing a sequence of steps, describing what would happen with each step and how many cycles it would require.

Solution: The solution would proceed as follows:

i. *Reduce from 1024 to 512 with one kernel launch. Requires 20 + 16*3 cycles.*
ii. *Reduce from 512 to 256 with one kernel launch. Requires 20 + 8*3 cycles.*
iii. *Reduce from 256 to 128 with one kernel launch. Requires 20 + 4*3 cycles.*
iv. *Reduce from 128 to 64 with one kernel launch. Requires 20 + 2*3 cycles.*
v. *Reduce from 64 to 32 with one kernel launch. Requires 20 + 1*3 cycles.*
vi. *Serially reduce the rest. Requires 32 cycles*

*Total = 5*20 + 31*3 + 32 = 225 cycles.*