CMU 15-418/618: Exam 1 Practice Exercises

Problem 1: Miscellaneous Short Answer Questions

A. Today directory-based cache coherence is widely adopted because snooping-based implementations often scale poorly. What is the main reason for this lack of scalability, and how do directory-based approaches avoid this problem?

B. On your first day of work at Intel, you sit in a design meeting for the company's next quad-core processor, the Intel Core i-15418. Your boss immediately announces that like previous chips, this processor will use directory-based cache coherence using a full bit vector scheme. An engineer slams his notebook down on the table, and yells "What!?! We talked about several better ways to reduce the overhead of directories in 15-418! We should implement one of those!" The room goes silent, and the next day he is transferred to another group. Why was the boss unhappy with this suggestion. (Intel processors have 64-byte cache lines.)

C. Give one reason why a processor architect might decide to adopt a relaxed memory consistency model. (Hint: I'd like to see the term *latency* in your answer.)

D. You and your friend think the OpenMPI library is a bit crufty and decide to design a new message passing API for C++ to replace MPI. You've already designed and implemented new versions of SEND and RECEIVE to your API and now your friend suggests that you also add support for LOCKS in your library. Do you agree with this suggestion or do you argue that it is unnecessary mechanism in the message passing programming model? Why?

E. Recall that in the flat, *cache-based*, sparse directory scheme, the list of sharing processors is maintained in the caches as a doubly-linked list. If this list was maintained as a *singly-linked* (rather than doubly-linked) list, would you expect any significant impact on the latency of the following operations (please explain your answers):

Read misses:

Write misses:

Replacements (the line is evicted to make room for other data):

F. Imagine you are asked to implement ISPC, and your system must run a program that launches 1000 ISPC tasks. Give one reason why it is very likely more efficient to use a fixed-size pool of worker threads rather than create a pthread per task. Also specify how many pthreads you'd use in your worker pool when running on a quad-core, hyper-threaded Intel processor. Why?

G. Your friend suspects that his program is suffering from high communication overhead, so to overlap the sending of multiple messages, he tries to change his code to use asynchronous, non-blocking sends instead of synchronous, blocking sends. The result is this code (assume it's run by thread 1 in two-thread program).

```
float mydata[ARRAY_SIZE];
int dst_thread = 2;

update_data_1(mydata); // updates contents of mydata
async_send(dst_thread, mydata, sizeof(float) * ARRAY_SIZE);

update_data_2(mydata); // updates contents of mydata
async_send(dst_thread, mydata, sizeof(float) * ARRAY_SIZE);
```

Your friend runs to you to say "my program no longer gives the correct results." What is his bug?

H. Complete the ISPC code below to write an if-than-else statement that causes an 8-wide SIMD processor to run at nearly 1/8th its peak rate. (Assume the ISPC gang size is 8. Pseudocode for an answer is fine.)

```
void my_ispc_func() {

   int i = programIndex;









}
```

I. Assume you want to efficiently run a program with very high temporal locality. If you could only choose one, would you add a data cache to your processor or add a significant amount of hardware multi-threading? Why?

Problem 2: Buying a New Computer

You write a bit of ISPC code that modifies a $32 \times$ `height` grayscale image based on 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;
       }
     }
}
```

(question continued on next page)

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

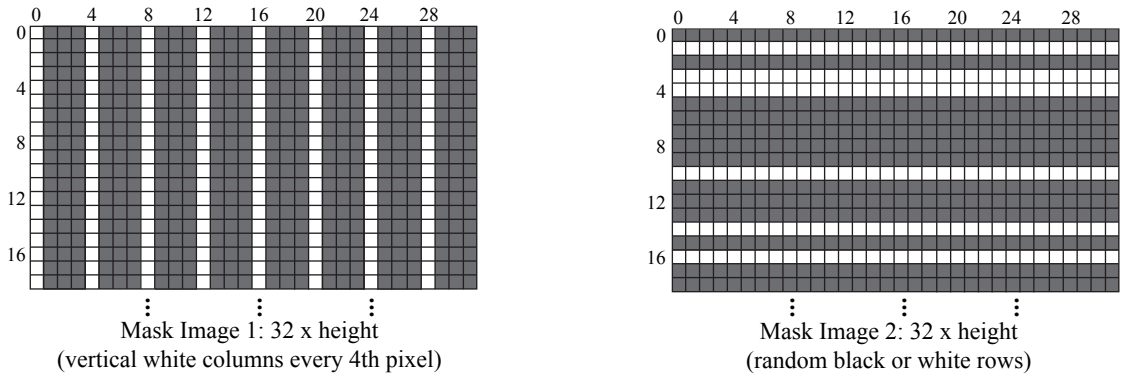


Figure 1: Image masks used to govern image manipulation by `brighten_image`

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. 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.)

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.

Problem 3: 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:

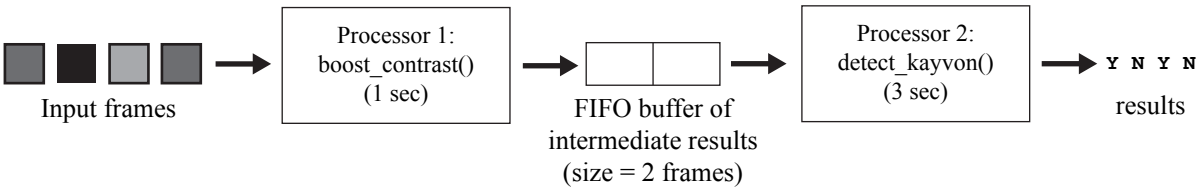
- 4 GHz single core CPU capable of performing one floating point addition per clock (no parallelism)
- 1 GHz quad-core CPU capable of performing one 4-wide SIMD floating point addition per clock
- 1 Ghz dual-core CPU capable of performing one 16-wide SIMD floating point addition per clock

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).

Problem 4: Parallelizing a Video Processing Pipeline

Consider a video processing pipeline that is parallelized using two processors that communicate through shared memory. Processor 1 accepts as input a new video frame, and runs the function `boost_contrast` to increase the frame's contrast. It then puts the modified frame in a FIFO buffer large enough to hold 2 frames. Processor 2 then retrieves the modified frame from the buffer and performs face detection using the function `detect_kayvon`. It outputs a bit indicating whether Professor Kayvon is in the frame. This process is repeated for all video frames in a pipelined fashion (that is, after Processor 1 completes contrast detection on frame n and places its result in the intermediate buffer, it immediately begins working on frame $n + 1$. The buffer is of finite size, so processor 1 blocks if the buffer becomes full. *All frames in the video are independent, so the results of processing a frame do not influence processing of any other frames.*



A. If contrast enhancement takes 1 second and face detection takes 3 seconds, what is the throughput of the pipeline in frames-per-second?

   (Assume that costs to store/retreive data from the buffer, or to synchronize buffer access between the two processors are negligible. Also, assume the input video is very long, containing millions of frames.)

B. You ask your friend to improve the performance of the video processing pipeline by changing the program's implementation. She smiles and says "Oh, that's easy! We just need to allocate a larger buffer to hold the intermediate results between processor 1 and 2." Do you agree with your friend? Why? If yes, what throughput do you expect to see as a result of her optimization?

Problem 5: Optimizing a Multi-Threaded Program

Your friend writes the following multi-threaded C++ program that combines two images `image1` and `image2`. The implementation uses three threads, and each thread is responsible for processing a single channel (red, green, or blue) of the image. Notice that this processing requires the thread to loop over the image data `MAX_ITERS` times.

```
struct Pixel {
   float r, g, b;
};

#define MAX_ITERS   1000
#define IMAGE_SIZE  64 * 64
float expensive_function(float, float);
Pixel *image1, *image2;

void workerR() {
   for (int iters=0; iters<MAX_ITERS; iters++)
      for (int i=0; i<IMAGE_SIZE; i++)
         result[i].r = expensive_funct(image1[i].r, image2[i].r);
}

void workerG() {
   for (int iters=0; iters<MAX_ITERS; iters++)
      for (int i=0; i<IMAGE_SIZE; i++)
         result[i].g = expensive_funct(image1[i].g, image2[i].g);
}

void workerB() {
   for (int iters=0; iters<MAX_ITERS; iters++)
      for (int i=0; i<IMAGE_SIZE; i++)
         result[i].b = expensive_funct(image1[i].b, image2[i].b);
}

int main() {
  image1 = new Pixel[IMAGE_SIZE];
  image2 = new Pixel[IMAGE_SIZE];
  result = new Pixel[IMAGE_SIZE];

  // ... initialize image1, image2 here ...

  pthread_t t0, t1;
  pthread_create(&t0, NULL, workerR, NULL);
  pthread_create(&t1, NULL, workerG, NULL);
  workerB();
  pthread_join(t0, NULL);
  pthread_join(t1, NULL);

  // ... use 'result' image here ...
}
```

A. Your friend runs this program on the cache coherent quad-core Intel processor. Given that the problem is embarrassingly parallel (there is no communication among the threads) and exhibits a high compute to memory access ratio (let's assume the images are small enough that all three images can fit in the private L2 cache of each core), your friend expects near perfect ($3\times$) speedup. She is shocked when she doesn't obtain it, but you are not surprised. What is the cause of the slowdown?

B. Modify the program to improve its performance. You are allowed to modify the data structures used in any way but you are not allowed to change what computations are performed by each thread. That is, `workerR` must still process the red channel of the image, `workerG` must still process the green channel, etc. You only need to describe your solution in pseudocode (compilable C++ is not required).

Problem 6: Be An ISPC Compiler

A. Please study the ISPC function `my_ispc_function` given below. The function multiplies all elements of the array 'input' by two.

```
// Recall ISPC's built-in variables:
//   'programCount' is the ISPC gang size
//   'programIndex' is a per-instance identifier between 0 and programCount-1;


void my_ispc_function(uniform int N, uniform float* input, uniform float* output) {

   // do not assume programCount divides N
   uniform int chunkSize = (N+programCount-1) / programCount;

   int start = programIndex * chunkSize;
   int end = start + chunkSize;

   if (end > N)
     end = N;

   for (int i=start; i<end; i++) {
      output[i] = 2 * input[i];
   }
}
```

Imagine you are implementing an ISPC compiler that translates ISPC programs into C programs that use vector intrinsics. To help, we have provided you a library of vector intrinsics similar to the library you used in Assignment 1. The library's methods are given on the next page. NOTE: YOU DO NOT need to study these functions in detail, but note that:

(a) Although not listed assume that all arithmetic instructions (add, subtract, multiply, divide, etc. are present in the library for your use) and version are present for both vectors of floats and vectors of ints).

(b) Assume that all binary operations on masks are present: and, or, equal

(c) Just like Assignment 1, all operations can take an optional mask (1's in the mask mean the lane is ENABLED)

(d) There are two types of vector load and store methods: **packed loads and stores** (loading consecutive elements) and **scatters and gathers** (loading non-consecutive elements).

On the next page, please translate this ISPC program into its vector equivalent `my_vector_function`. *Your implementation can be pseudocode, but it should produce the same mapping of work to vector lanes as the real ISPC compiler implementation.*

```
// Assume intVec and floatVect are vectors of ints and floats of size PROGRAM_COUNT
// Assume maskVec is a vector of bools:  {111...111} = all lanes enabled

// ARITHMETIC EXAMPLES ////////////////////////////////////////////////////////////////

intVec  initVec(int value, maskVec mask);              // set all elements of vector to ‘value‘
maskVec initMaskVec(bool value, maskVec mask);         // set mask to all ones or zeros
intVec  copyVec(intVec a, maskVec mask);               // result = a;

intVec  addVec(intVec a, intVec b, maskVec mask);      // add vectors (same for: ’mul’, ’sub’, ’div’)

maskVec lessThanVec(intVec a, intVec b, maskVec mask); // result[i] = a[i] < b[i]
maskVec andVec(maskVec a, maskVec b, maskVec mask);    // a && b (same for: ’equal’, ’or’)
maskVec notVec(maskVec a, maskVec mask);               // !a

int     countOnesVec(maskVec v, maskVec mask);         // returns number of 1’s in v

// LOAD/STORE, GATHER/SCATTER EXAMPLES ////////////////////////////////////////////////

// load A[0], A[1], A[2], ..., A[PROGRAM_COUNT-1] into vector
intVec loadVec(int* A, maskVec mask);

// store v into A[0], A[1], A[2], ..., A[PROGRAM_COUNT-1]
void   storeVec(int* A, intVec v, maskVec mask);

// load A[indices[0]], A[indices[1]], ..., A[indices[PROGRAM_COUNT-1]] into vector
intVec gatherVec(int* A, intVec indices, maskVec mask);

// store elements of v into A[indices[0]], A[indices[1]], ..., A[indices[PROGRAM_COUNT-1]]
void   scatterVec(int* A, intVec indices, intVec v, maskVec mask);

// YOUR IMPLEMENTATION GOES HERE (WE’VE STARTED IT FOR YOU) ///////////////////////////

void my_vector_function(uniform int N, uniform float* input, uniform float* output) {

   intVec programIndex;         /* assume programIndex = {0, 1, 2, 3, ..., PROGRAM_COUNT-1}; */
   intVec programCount = initVec(PROGRAM_COUNT);
   intVec chunkSize    = divVec(vecAdd(initVec(N), vecAdd(programCount, initVec(-1))), programCount);

   intVec start        = mulVec(programIndex, chunkSize);
   intVec end          = addVec(start, chunkSize);
```

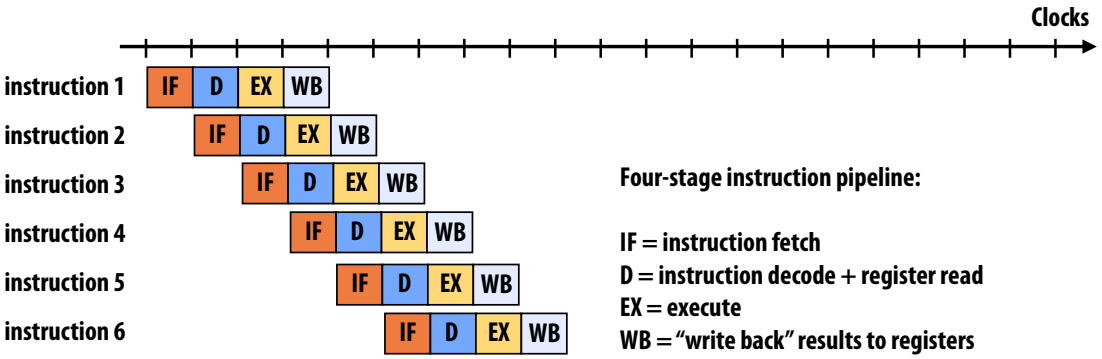
B. There is a performance problem with the current implementation of `my_ispc_function`. Please explain the problem and then re-write the original ISPC code to remove this problem. (Hint: it has to do with memory access.) For simplicity, your implementation can assume `programCount` divides `N` equally if you wish – though it would be cooler if it did not. (Note: if you've forgotten *exact* ISPC syntax it's okay, just write good pseudocode.)

C. How would the code you produced in part A change as a result of your ISPC program rewrite in part B? You do not need to provide the exact modified code here. A short explanation of the major difference is sufficient.

Problem 7. A Yinzer Processor Pipeline

Yinzer Processors builds a single core, single threaded processor that executes instructions using a simple four-stage pipeline. As shown in the figure below, each unit performs its work for an instruction **in one clock**. To keep things simple, assume this is the case for all instructions in the program, including loads and stores (memory is infinitely fast).

The figure shows the execution of a program with six **independent instructions** on this processor. *However, if instruction B depends on the results of instruction A, instruction B will not begin the IF phase of execution until the clock after WB completes for A.*



A. Assuming all instructions in a program are **independent** (yes, a bit unrealistic) what is the instruction throughput of the processor?

B. Assuming all instructions in a program are **dependent** on the previous instruction, what is the instruction throughput of the processor?

C. What is the latency of completing an instruction?

D. Imagine the EX stage is modified to improve its throughput to two instructions per clock. What is the new overall maximum instruction throughput of the processor?

E. Consider the following C program:

```
float A[100000];
float B[100000];
// assume A is initialized here

for (int i=0; i<100000; i++) {
    float x1 = A[i];
    float x2 = 2*x1;
    float x3 = 3 + x2;
    B[i] = x3;
}
```

Assuming that we consider only the four instructions in the loop body (for simplicity, disregard instructions for managing the loop), what is the average instruction throughput of this program? (Hint: You should probably consider a number of loop iterations worth of work).

F. Modify the program to achieve peak instruction throughput on the processor. Please give your answer in C-pseudocode.

G. Now assume the code is changed to the following:

```
#pragma omp parallel for
for (int i=0; i<100000; i++) {
  float x1 = A[i];
  float x2 = 2*x1;
  float x3 = 3 + x2;
  B[i] = x3;
}
```

Given this program, what feature (SIMD, hardware multi-threading, or a larger cache?) would you add to the processor to obtain peak instruction throughput on a *single core*? Please be specific. (You may not change how the instruction pipeline works or how it handles dependent instructions. You may not change the program.)

Problem 8. Memory Consistency

Consider the following program which has four threads of execution. In the figure below, the assignment to `x` and `y` should be considered stores to those memory addresses. Assignment to `r0` and `r1` are loads from memory into local processor registers. (The print statement does not involve a memory operation.)

Processor 0	**Processor 1**	**Processor 2**	**Processor 3**

`x = 1`	`y = 1`	`r0 = y`
		`r1 = x`
		`print (r0 & ~r1)`
			`r0 = x`
			`r1 = y`
			`print (r0 & ~r1)`

- Assume the contents of addresses `x` and `y` start out as 0.
- Hint: the expression `a & ~b` has the value 1 only when `a` is 1 and `b` is 0.

You run the program on a four-core system and observe that both processor 2 and processor 3 print the value 1. Is the system sequentially consistent? Explain why or why not?