Name: SOLUTIONS	UID: CS 33

1. Cache that Struct!

Consider the following two scenarios. Which has a higher cache miss rate? Why might we choose to do either implementation? Assume that N $\ggg$ cache size and we have 64-byte cache lines.

a.
```
struct airplane {
    int id_number;
    int num_passengers;
    float airport_x, airport_y;
    float coord_x, coord_y;
    float dir_x, dir_y;
    float vel_x, vel_y;
};

struct airplane airplanes[N];
float avg_dist_from_port = 0;
for (int i = 0; i < N; i++) {
    float airport_x = airplanes[i].airport_x;
    float airport_y = airplanes[i].airport_y;
    avg_dist_from_port += sqrt((airport_x)**2 + (airport_y)**2);
}
avg_dist /= N;
```

Miss Rate = Misses/Requests = (Struct Size/Cache Size)/Requests

All of the excess data that we are not using is pulled into the cache when we access each airplane's position and nearest airport position. Therefore, we take the entire size of the struct (40 bytes) and the size of our cache line (64-byte) and plug the values into our formula. Additionally, we know we make 2 requests per iteration.

Miss Rate = Misses/Requests = $(40 * N/64)/(2N) = 0.3125$

This method has a higher cache miss rate, but is much more intuitive to use. Therefore, it is likely that we will choose this implementation if our N is small and impact on performance is negligible.

b.
```
struct pair{
    float x,y;
};
int id_numbers[N];
int num_passengers[N];
struct pair airport_coords[N];
struct pair coords[N];
struct pair dir[N];
struct pair vel[N];

float avg_dist_from_port = 0;
for (int i = 0; i < N; i++) {
    float airport_x = airport_coords[i].x;
    float airport_y = airport_coords[i].y;
    avg_dist_from_port += sqrt((airport_x)**2 + (airport_y)**2);
}
avg_dist /= N;
```

By simplifying our struct into JUST the x and y float values to act as a sort of "parent" struct, we can create an individual struct array for each value type we want to store. This simplifies our operations: each float is 4 bytes, so we can fit 16 into our cache. Each iteration accesses 2 bytes, which fits evenly into our cache. This means we only miss the first access of every 16 floats.
Miss Rate = Misses/Requests = 1/16 = 0.0625
While this is more optimized, it may be less intuitive to work with and is therefore better to use if N is extremely large.

2. Stencils

In the performance lab, you will be optimizing some code for 3D stencil computation. In this question, let's take a look at 1D and 2D stencil computations. (A side note: in the performance lab and in this question, we simplified the notation of the stencil computation (for handling edge cases), so it may differ slightly from other sources you might find.)

1. **1D Stencil**

   Assume OUT_LEN is N $\ggg$ 10000, $KERN\_LEN$ is 2, and IN_LEN is N+2. Assume the cache has 32B/line.

   a. What is the miss rate of *vanilla_1D_stencil*?

```
void vanilla_1D_stencil(double In[IN_LEN], double Out[OUT_LEN],
   double Kern[KERN_LEN]) {
    for (int i = 0; i < OUT_LEN; i++) {
        for (int x = 0; x < KERN_LEN; x++) {
            Out[i] += In[i+x] * Kern[x]
        }
    }
}
```

   1/16

   - Each cache line holds 4 doubles (32 bytes)
   - 2N iterations in total, from the nested loops
   - Kern is brought into cache once, and reused in each iteration
     - Approximately 0 miss per iteration
   - For the array Out, 1 miss every time the outer loops counter i increments by 4
     - Approximately 1/8 miss per iteration
   - Similarly for the array In
     - Approximately 1/8 miss per iteration
   - Per iteration, $(0 + 1/8 + 1/8) = 1/4$ miss
   - 4 memory accesses per iteration (3 reads and 1 write)
   - Missrate = #cachemiss/#memoryaccess = $(1/4)/4 = 1/16$

   b. Given the initialization below, what does the resulting *Out* array look after calling *vanilla_1D_stencil*?

```
double* In = malloc(IN_LEN * sizeof(double));
for (int i = 0; i < IN_LEN; i++) { In[i] = i + 1;}

double* Kern = malloc(KERN_LEN * sizeof(double));
for (int i = 0; i < KERN_LEN; i++) { Kern[i] = 1.0 / KERN_LEN; }

double* Out = malloc(OUT_LEN * sizeof(double));
for (int i = 0; i < OUT_LEN; i++) {Out[i] = 0;}
```

The resulting Out array is $[1.5, 2.5, 3.5, ...]$

- Array In is initialized as [1, 2, 3, 4,...]
- Kern is [0.5, 0.5]
  - This stencil/kernel just averages two neighboring elements
- Note: this question has little to do with memory organization or performance optimization. The aim is to use a toy example to hopefully provide some intuition about the stencil computation in the performance lab.

Bonus: Assume that $KERN\_LEN$ was changed to be 1000, how could we change the code to reduce the miss rate?

We could apply blocking.

With the kern size of 1000, it's unlikely that the full kernel array could fit within the L1 cache. To gain reuse, we could block portions of this array such that it would fit within the cache.

An example of this blocking is given below:

```
void vanilla_2D_stencil(double In[IN_LEN][IN_LEN], double Out[
   OUT_LEN][OUT_LEN], double Kern[KERN_LEN][KERN_LEN]) {
    int B = 64; // 64 is just a choice, just needs to fit within
       L1 cache
    for (int x = 0; x < KERN_LEN; x++) {
        double k = Kern[x];
        for (int bi = 0; bi < OUT_LEN; bi += B) {
            int end = min(bi + B, OUT_LEN);
            for (int i = bi; i < end; i++) {
                Out[i] += In[i + x] * k;
            }
        }
    }
}
```

2. **2D Stencil**

Assume OUT_LEN is N $\ggg$ 10000, $KERN\_LEN$ is 4, and IN_LEN is N+4. Assume the cache has 32B/line and the total cache size is smaller than 8N bytes.

a. What is the miss rate of `vanilla_2D_stencil`?

```
void vanilla_2D_stencil(double In[IN_LEN][IN_LEN], double Out[
    OUT_LEN][OUT_LEN], double Kern[KERN_LEN][KERN_LEN]) {
     for (int i = 0; i < OUT_LEN; i++) {
         for (int j = 0; j < OUT_LEN; j++) {
             for (int x = 0; x < KERN_LEN; x++) {
                 for (int y = 0; y < KERN_LEN; y++) {
                     Out[i][j] += In[i+x][j+y] * Kern[x][y];
                 }
             }
         }
     }
}
```

5/256

- Each cache line holds 4 doubles (32 bytes)
- 16 (N^2) iterations in total
  - from the four nested loops NxNx4x4
- Kern is brought into cache once, and reused in each iteration
  - Approximately 0 miss per iteration
- For the array Out, 1 miss every time the loops counter j (the second outmost loop) increments by 4
  - Approximately 1/64 miss per iteration
- For the array In, 4 miss on average when the loops counter j (the second outmost loop) increments by 4
  - brings in 4 cache lines that respectively start with the addresses: In[i+1][j], In[i+2][j], and In[i+3][j]
  - approximately 1/16 misses per iteration
- Per iteration, $(0 + 1/16 + 1/64) = 5/64$ miss
- 4 memory accesses per iteration
- miss rate = # cache miss / # memory access = $(5/64)/4 = 5/256$

b. How can we reduce the miss rate?

- First of all, before considering how, why reducing miss rates?
  - Compared to cache hit, cache miss is expensive (order of magnitude more clock cycles)
- Use tiling/blocking to improve spatial locality
  - The total cache size is less than 8N, so it is not large enough to even hold a full row of data (of either the input or the output array)

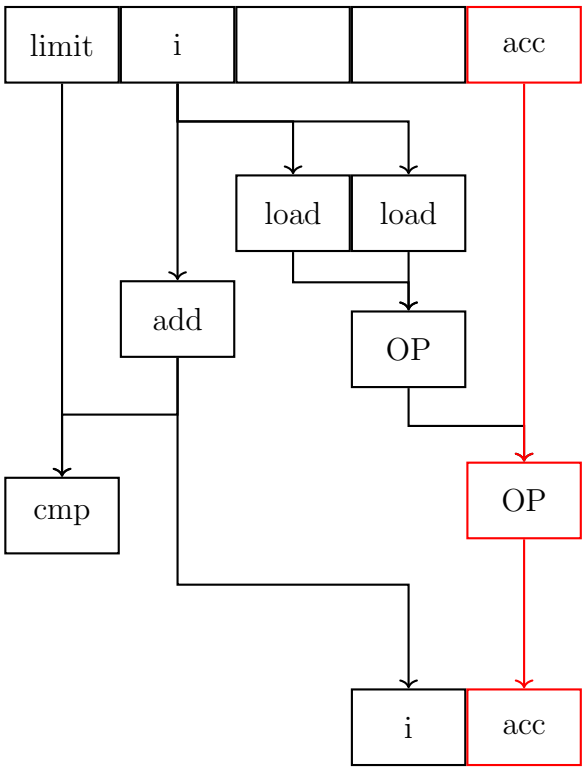
Optional: MisS1oN CRit1KaL

Examine the code below that stores data into an accumulator from operations on elements of the vector v. Assume that there are no compiler optimizations applied when compiling this code. Assume that OP has a higher latency than the add operation.

```
void combine7(vec_ptr v, data_t *dest) {
    long length = vec_lenth(v);
    long limit = length - 1;
    data_t *data = get_vec_start(v);
    data_t acc = 0;

    for (long i = 0; i < limit; i+= 2) {
        acc = acc OP (data[i] OP data[i + 1]);
    }
    for(; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

a. For one iteration of the loop, draw the data-flow graph, highlighting the critical path.



b. The function incorporates 2x1 loop unrolling, but what other ways can we modify the function to reduce the latency bound by half?

We have incorporated a 2x1 loop unrolling, and we are solely dependent upon acc waiting on acc from previous loops. Unrolling it again to be a 4x1 loop unrolling, we would half our latency bound again.

You could also create a second accumulator, creating a 2x2 unrolling, or even a 4x2 unrolling, which would shrink the latency bound as well.