SIMD Report

Name: Akshay Gujjar
Student ID: 027960788

Find 8 Shortest Distances Between N Points Code Explanation:

- In this assignment, the functionality being enhanced is finding the 8 shortest distances between points in a 3D space like a cluster in ML.
- I was able to achieve around 150 X faster speeds using SIMD.
- The variable MAX_LEN stores the number of points in the 3D space.
- We compute the distance between a point and all the other points except itself. We repeat the process for all the other points.
- We then find the 8 shortest distances. We update the result array of size 8 with the 8 shortest distance every time.
- Initialize with a big value using SIMD 4 values in parallel.
- Load the values to SIMD registers:

  ```
  x_simd1 = _mm256_loadu_si256((__m256i *)&x[j]);
  y_simd1 = _mm256_loadu_si256((__m256i *)&y[j]);
  z_simd1 = _mm256_loadu_si256((__m256i *)&z[j]);
  ```

- Get the differences on the X, Y, and Z axes.

  ```
  x_simd0 = _mm256_sub_epi32(x_simd0, x_simd1);
  y_simd0 = _mm256_sub_epi32(y_simd0, y_simd1);
  z_simd0 = _mm256_sub_epi32(z_simd0, z_simd1);
  ```

- Square the values.

  ```
  x_simd1 = _mm256_mul_epi32(x_simd0, x_simd0);
  y_simd1 = _mm256_mul_epi32(y_simd0, y_simd0);
  z_simd1 = _mm256_mul_epi32(z_simd0, z_simd0);
  ```

  - Sum the values.

  ```
  sum_simd = _mm256_add_epi32(x_simd1, y_simd1);
  sum_simd = _mm256_add_epi32(sum_simd, z_simd1);
  ```

- Update the result array in SIMD style.

  ```
  result_simd_i = _mm256_min_epu32(result_simd_i, sum_simd);
  ```

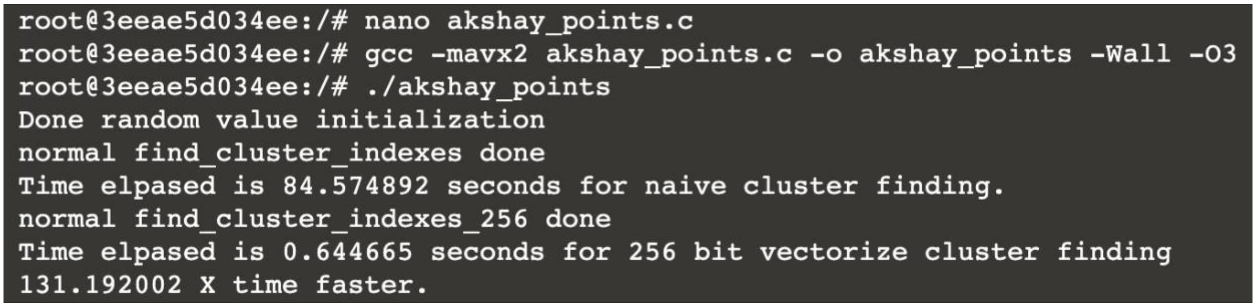
- Now, get the result back in 8 different integers to add them.

  ```
  _mm256_storeu_si256((__m256i *)result, result_simd);
  ```

Compilation and Execution Steps:

1. Create a file with the code at the end of this document.
2. In Ubuntu Terminal, enter the following command: nano akshay_points.c 3. Compile the code using the following command: gcc -mavx2 akshay_points.c -o akshay_points -Wall -O3
4. Print the output using the following command: ./akshay_points

Output:



```
root@3eeae5d034ee:/# nano akshay_points.c
root@3eeae5d034ee:/# gcc -mavx2 akshay_points.c -o akshay_points -Wall -O3
root@3eeae5d034ee:/# ./akshay_points
Done random value initialization
normal find_cluster_indexes done
Time elpased is 84.574892 seconds for naive cluster finding.
normal find_cluster_indexes_256 done
Time elpased is 0.644665 seconds for 256 bit vectorize cluster finding
131.192002 X time faster.
```

Issues:

1. Segmentation Fault Error

- Initially, I was facing a "segmentation fault" error because of the following line of code: `result_simd = _mm256_min_epu32(result_simd_i, result_simd);`
- I tried to fix the issue using the GDB debugger but was unable to do so.
- After researching online, I figured that the segmentation fault was occurring because I used aligned load instructions in the optimized code like "`_mm256_load_si256`".
- When using aligned instructions, the compiler is expected to generate memory addresses that are 64-byte aligned.
- GCC does not generate the program addresses that are 64-byte aligned.
- However, compilers like Clang and Intel's own compiler support this.
- I was able to resolve the segmentation fault issue by using unaligned load and store instructions when loading values to SIMD registers:

  ```
  x_simd0 = _mm256_loadu_si256((__m256i *)&x[i]);
  y_simd0 = _mm256_loadu_si256((__m256i *)&y[i]);
  z_simd0 = _mm256_loadu_si256((__m256i *)&z[i]);
  x_simd1 = _mm256_loadu_si256((__m256i *)&x[j]);
  y_simd1 = _mm256_loadu_si256((__m256i *)&y[j]);
  z_simd1 = _mm256_loadu_si256((__m256i *)&z[j]);
  ```

2. Output Discrepancy

- The other issue in the submitted code is the discrepancy between the naïve code output and the optimized code output.
- I am inclined to believe that the issue is caused because of my incorrect interpretation of the instruction used to calculate the SIMD result i.e. “_mm256_min_epu32”.
- result_simd_i = _mm256_min_epu32(result_simd_i, sum_simd);
- Using the above line of code I intended to update the SIMD result array.
- Due to this, there is a discrepancy between the naïve and optimized outputs.
- Also, there is an inaccurate speed-up of around 100x.
- It was interesting to learn that the maximum expected speed-up from SIMD is the number of operations per command, which can be up to 256 / 64 = 4 times according to my code.
- This inaccurate speed-up is because the SIMD result is incorrect.

Code:

```
#include <stdio.h>
#include <smmintrin.h>
#include <immintrin.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>

//#define MAX_LEN 2147483648
//#define MAX_LEN 1873741824
#define MAX_LEN 120000
#define RESULT_LEN 8

// trivial implementation of function which calculates distances
between points
// and then find 8 closest points like
a cluster like in ML int *
find_cluster_indexes(int *x, int *y,
int *z)
{
    int
*result;
int l = 0;
    int dist;
    result = (int *)malloc(sizeof(int)
* RESULT_LEN);     if(result == NULL) {
        printf("Out of Memory error
for result\n");         exit(1);
    }
    // Initialize with a big value
    for(int i = 0; i < RESULT_LEN; i++) result[i] = (1<<30) - 1;

    for(int i=0; i < MAX_LEN; i++)
    {
        for(int j=0; j < MAX_LEN; j++)
        {
            // Don't compute the
distance of point with it self
if(i != j)
            {
                dist = (x[i]
- x[j]) * (x[i] - x[j])
+                     (y[i] -
y[j]) * (y[i] - y[j]) +
                    (z[i] - z[j]) * (z[i] - z[j]) ;

} else {
dist = 0;
            }
            // Update result array now
            for(int k = 0; k < RESULT_LEN; k++)
            {
                if(dist < result[l])
                {

result[l] = dist;
l = (l+1) % RESULT_LEN;
                    break;
                }
                l = (l+1) % RESULT_LEN;
            }
        }
    }

    return result;
}

int * find_cluster_indexes_256(int *x, int *y, int *z)
```

```
{
int
*result;
    __m256i x_simd0; // get AVIX 256 bit vector
    __m256i y_simd0;
    __m256i z_simd0;
    __m256i x_simd1; // get AVIX 256 bit vector
    __m256i y_simd1;
    __m256i z_simd1;
    __m256i result_simd;
    __m256i sum_simd;

    result = (int *)_mm_malloc(sizeof(int) * RESULT_LEN, 64);
    if(result == NULL) {
        printf("Out of Memory error for result\n");
        exit(1);
    }
    // Initialize with a big value using SIMD
4 values in parallel     for(int i = 0; i <
RESULT_LEN; i++)
    {
        result[i] = ((1<<30) -1);
    }

    result_simd = _mm256_set1_epi32(result[0]);

    for(int i=0; i < MAX_LEN; i+=8)
    {
        __m256i result_simd_i = _mm256_set1_epi32(result[0]);
        // Load the values to SIMD
registers         x_simd0 =
_mm256_loadu_si256((__m256i *)&x[i]);
y_simd0 = _mm256_loadu_si256((__m256i
*)&y[i]);         z_simd0 =
_mm256_loadu_si256((__m256i *)&z[i]);

        for(int j=0; j < MAX_LEN; j+=8)
        {
            // Don't compute the
distance of point with it self
if(i != j)
            {
                x_simd1 =
_mm256_loadu_si256((__m256i *)&x[j]);
y_simd1 = _mm256_loadu_si256((__m256i *)&y[j]);
z_simd1 = _mm256_loadu_si256((__m256i *)&z[j]);
                // Get differences
                x_simd0 =
_mm256_sub_epi32(x_simd0, x_simd1);
y_simd0 = _mm256_sub_epi32(y_simd0, y_simd1);
                z_simd0 = _mm256_sub_epi32(z_simd0, z_simd1);
                // Square the values
                x_simd1 =
_mm256_mul_epi32(x_simd0, x_simd0);
y_simd1 = _mm256_mul_epi32(y_simd0, y_simd0);
z_simd1 = _mm256_mul_epi32(z_simd0, z_simd0);
                // Sum the values
                sum_simd =
_mm256_add_epi32(x_simd1, y_simd1);
sum_simd = _mm256_add_epi32(sum_simd, z_simd1);

                // Update result array SIMD style
                result_simd_i = _mm256_min_epu32(result_simd_i,
sum_simd);
            }
        }
        result_simd = _mm256_min_epu32(result_simd_i,
result_simd); // Causes segmentation Fault
    }
    // Now get the result back in 8
different integers and add them
_mm256_storeu_si256((__m256i *)result,
result_simd);     return result;
}

long long int sum_vectorized_256(int *arr)
{
    __m256i v_avix256i; // get AIVX 256 bit vector
    __m256i sum = _mm256_setzero_si256(); // Zero out the
sum vector     unsigned int decoupled_min[8]; // We are
adding 8 32 bit signed integers in parallel     long long
int final_sum = 0;
    // add 16 integers from input in 1 go with
AIVX 256 extensions     for (int i = 0; i <
MAX_LEN; i += 8)
    {
        // Get 256 bit vector from array
pointers         v_avix256i =
_mm256_load_si256((__m256i *)&arr[i]);
// Add the result into existing temp sum
across 8 lanes         sum =
_mm256_add_epi32(sum, v_avix256i);
    }
    // Now get the result back in 8 different integers and add
them
    _mm256_storeu_si256((__m256i *)decoupled_min, sum);

    for(int i = 0; i < 2; i++)
    {
        final_sum +=
decoupled_min[(i*4)] +
decoupled_min[(i*4) + 1] +
decoupled_min[(i*4) + 2] +
decoupled_min[(i*4) + 3];
    }
    //printf("final_sum=%0lld, decoupled_min[0]=%0d\n",
final_sum, decoupled_min[0]);     return final_sum;
}


void print_result(int *arr)
{
    for(int i=0; i < RESULT_LEN; i++)
    {
        printf("%u ", arr[i]);
    }
    printf("\n");
}

int main()
{
    int *x, *y, *z;
    int *result_normal, *result_simd;

x    = (int *)
malloc(sizeof(int) * MAX_LEN);
if(x == NULL) {
        printf("Out of memory in Malloc for x");
        return 1;
    }
```

```
y     = (int *)
malloc(sizeof(int) * MAX_LEN);
if(y == NULL) {
        printf("Out of memory in Malloc for y");
        return 1;
    }

z     = (int *)
malloc(sizeof(int) * MAX_LEN);
if(z == NULL) {
        printf("Out of memory in Malloc for z");
        return 1;
    }
    // Fill some
random data     for
(int i = 0; i <
MAX_LEN; i++){
x[i] = rand() %
(1<<16);         y[i]
= rand() % (1<<16);
        z[i] = rand() % (1<<16);
    }

    printf("Done random value initialization\n");

    double time_spent1 =
0.0;     double
time_spent2 = 0.0;

    /////////////
clock_t begin =
clock();

    result_normal = find_cluster_indexes(x, y, z);

    clock_t end = clock();

    printf("normal
find_cluster_indexes done\n");
//print_result(result_normal);
free(result_normal);

    time_spent1 += (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Time elpased is %f seconds for naive cluster
finding.\n", time_spent1);


///////////////
begin =
clock();

    result_simd = find_cluster_indexes_256(x, y, z);

    end = clock();

    printf("normal find_cluster_indexes_256 done\n");
    //print_result(result_normal);
    _mm_free(result_simd);

    time_spent2 += (double)(end - begin) / CLOCKS_PER_SEC;

    printf("Time elpased is %f seconds for 256 bit vectorize
cluster finding \n", time_spent2);     printf("%f X time
faster.\n", time_spent1/time_spent2);


    // Free the memory

free(x);
free(y);
free(z);

    //printf("TEST=%0lld\n", sum_vectorized_256(x));

    return 0;
}
```