DBSCAN

CUDA Parallelization

Ashish Choudhary
CED18I061

Hardware Configuration:

CPU NAME : Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
Number of Sockets: : 1
Cores per Socket : 4
Threads per core : 2
L1 cache size : 256 KiB
L2 Cache size : 1 MiB
L3 Cache size(Shared): 8 MiB
RAM : 16 GiB

Parallel Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
#include <chrono>

double ep;
double pts[1000][50];
int clusters[1000][1000];
int minpts, dim, num_pts;
__global__
void initialization(int *a, int *b, int N)
{

   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   int stride = gridDim.x * blockDim.x;
   for (int i = idx; i < N; i += stride)
   {
       a[i] = 0;
       b[i] = 0;
   }
}
__global__
void noise(int *a, int N)
{

   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   int stride = gridDim.x * blockDim.x;
   for (int i = idx; i < N; i += stride)
   {
       if (a[i] != 1)
           printf("%d ", i + 1);
   }
```

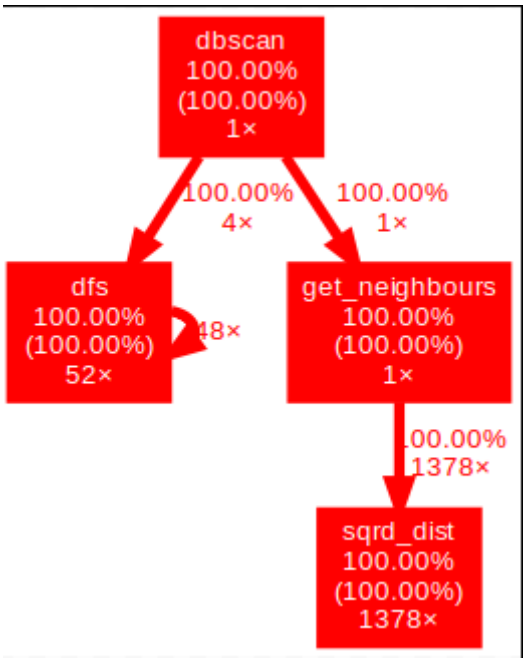
```
}
double sqrd_dist(int i, int j)
{
    double sum = 0;
    for (int k = 0; k < dim ; k++)
    {
        sum += pow(pts[i][k] - pts[j][k], 2);
    }
    return sqrt(sum);
}

void dfs(int i, int *siz, int* vis)
{
    vis[i] = 1;
    printf("%d ", i + 1);
    for (int a  = 0; a < siz[i] ; a++)
    {
        if (vis[clusters[i][a]] != 1)
            dfs(clusters[i][a], siz, vis);
    }
}



int main()
{
    //printf("Enter the ep distance:");
    //scanf("%lf", &ep);
    ep = 1;
    if (ep < 0)
    {
        printf("INVALID EPSILON DISTANCE");
        return 0;
    }

    //printf("Enter the minimum points:");
```

```
    //scanf("%d", &minpts);
    minpts = 2;
    if (minpts < 1)
    {
        printf("INVALID MIN PTS");
        return 0;
    }
    //printf("Enter the dimesions of the points:");
    //scanf("%d", &dim);
    dim = 3;
    if (dim < 1)
    {
        printf("INVALID DIMENSIONS");
        return 0;
    }
    //printf("Enter the number of points:");
    //scanf("%d", &num_pts);
    num_pts = 53;
    if (num_pts < 1)
    {
        printf("INVALID NUMBER OF PTS");
        return 0;
    }
    //printf("Enter points:");
    for (int i = 0 ; i < num_pts; i++)
    {
        for (int j = 0; j < dim; j++)
        {
            //scanf("%lf", &pts[i][j]);
            pts[i][j] = rand() % 10;
            //printf("%lf ", pts[i][j]);
        }
        //printf("\n");
    }
    int block_thread[9][2] = {{1, 1}, {1, 10}, {1, 20}, {1,
30}, {1, 40},
```

```
        {10, 10}, {20, 10}, {1, num_pts}, {num_pts / 8,
num_pts}
    };
    for (int thread = 0; thread < 9; thread++)
    {
        int *siz, *vis;
        size_t size = num_pts * sizeof(int);
        cudaMallocManaged(&siz, size);
        size = num_pts * sizeof(int);
        cudaMallocManaged(&vis, size);
        auto start =
std::chrono::high_resolution_clock::now();
        initialization <<< block_thread[thread][0],
block_thread[thread][1]>>>(siz, vis, num_pts);
        cudaDeviceSynchronize();
        printf("\n");
        for (int i = 0; i < num_pts - 1; i++)
        {
            for (int j = i + 1; j < num_pts; j++)
            {
                if (i == j)
                    continue;
                if (sqrd_dist(i, j) <= ep)
                {
                    clusters[i][siz[i]] = j;
                    clusters[j][siz[j]] = i;
                    siz[i]++;
                    siz[j]++;
                }
            }
        }

        int cnt = 0;
        for (int i = 0; i < num_pts; i++)
        {
            if (vis[i] != 1 && siz[i] >= minpts)
```

```
            {
                cnt++;
                printf("cluster %d : ", cnt);
                dfs(i, siz, vis);
                printf("\n");
            }

        }
        printf("NOISE :");
        noise <<< block_thread[thread][0],
block_thread[thread][1]>>>(vis , num_pts);
        cudaDeviceSynchronize();
        printf("\n");
        auto end =
std::chrono::high_resolution_clock::now();
        auto duration =
std::chrono::duration_cast<std::chrono::nanoseconds>(end -
start);
        printf("Exec time : %ld \n", duration.count());
        cudaFree(vis);
        cudaFree(siz);
    }
    return 0;
}
```

CRITICAL PART AND METHODOLOGY



sqrt_dist is the most invoked function. It is invoked inside the closure function. Initial attempt was made to parallelize the sqrd_dist function itself.

Significant improvement was observed by parallelizing the noise function. Furthermore, some more parallelization like the one below was tried on the noise function and initialization was done.

```
__global__
void initialization(int *a, int *b, int N)
{
```

```
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = gridDim.x * blockDim.x;
    for (int i = idx; i < N; i += stride)
    {
        a[i] = 0;
        b[i] = 0;
    }
}
__global__
void noise(int *a, int N)
{

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = gridDim.x * blockDim.x;
    for (int i = idx; i < N; i += stride)
    {
        if (a[i] != 1)
            printf("%d ", i + 1);
    }
}
```

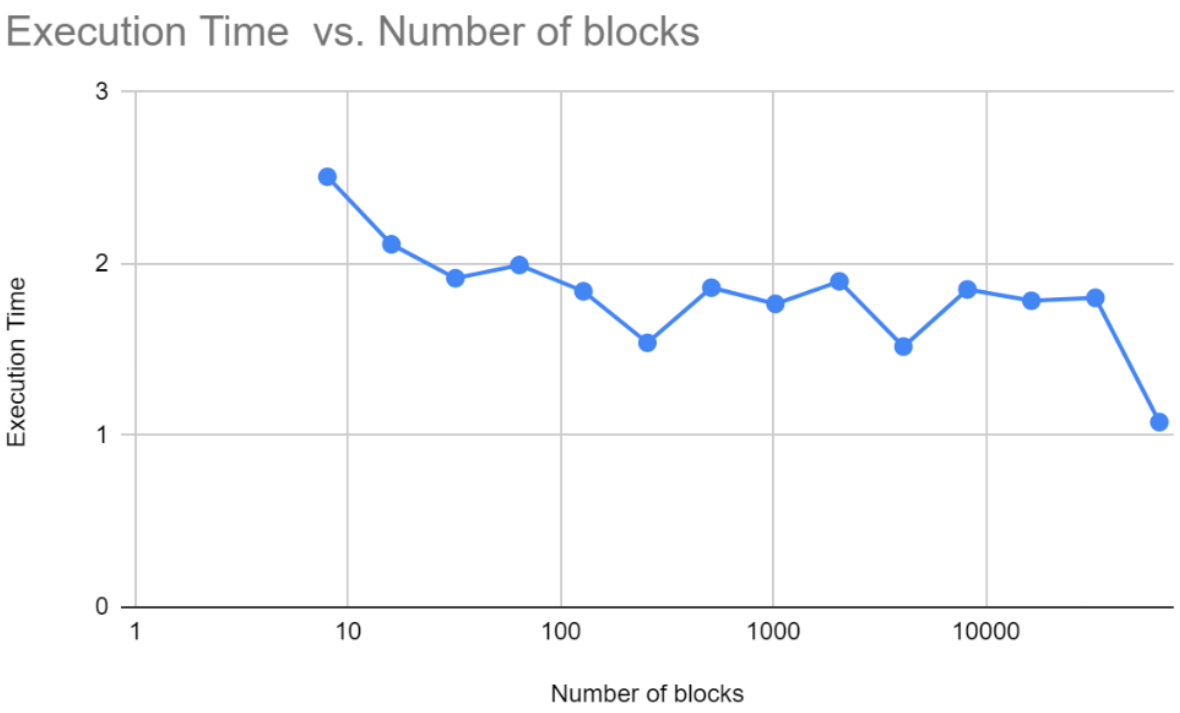
Output:

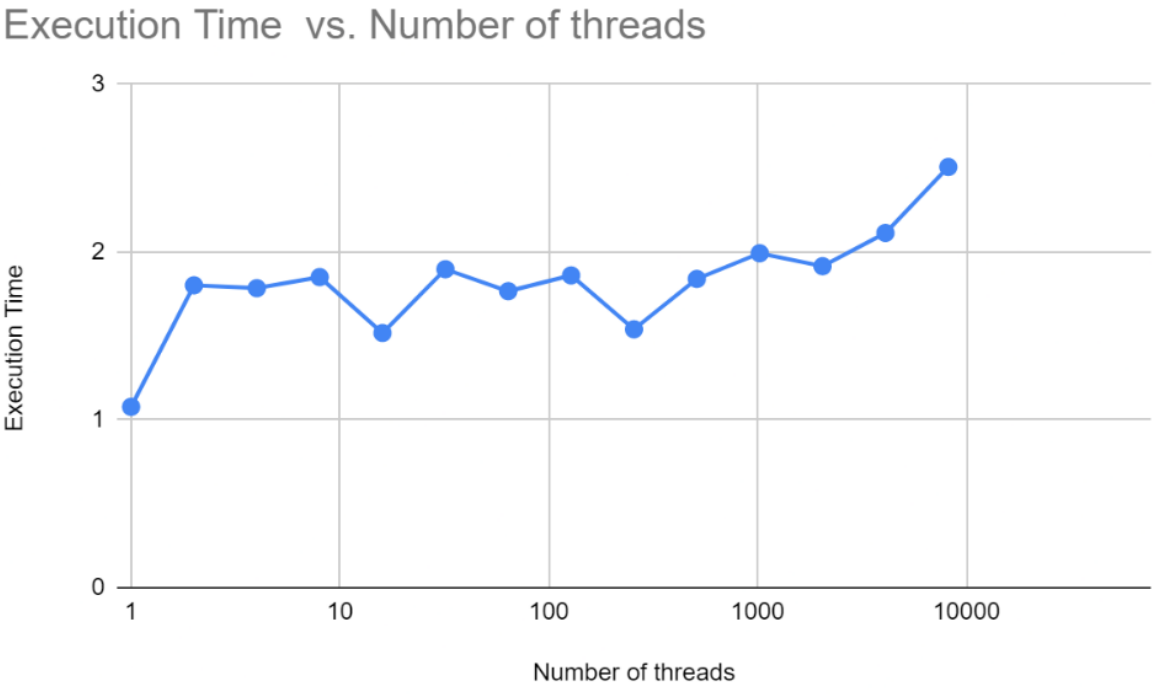
```
In [18]: !nvcc -arch=sm_70 -o matrix-multiply-2d 08-matrix-multiply/01-matrix-multiply-2d.cu -run -std=c++11
```

```
cluster 1 : 18 38 53
NOISE :1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 39 40 41
42 43 44 45 46 47 48 49 50 51 52
Exec time : 3172936

cluster 1 : 18 38 53
NOISE :1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 39 40 41
42 43 44 45 46 47 48 49 50 51 52
Exec time : 601674

cluster 1 : 18 38 53
NOISE :1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 39 40 41
42 43 44 45 46 47 48 49 50 51 52
Exec time : 440548

cluster 1 : 18 38 53
NOISE :1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 39 40 41
42 43 44 45 46 47 48 49 50 51 52
Exec time : 390846

cluster 1 : 18 38 53
NOISE :33 34 35 36 37 39 40 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 19 20 21 22 23 24 25 26 27 28 29 30 31 32 41
42 43 44 45 46 47 48 49 50 51 52
Exec time : 383486

cluster 1 : 18 38 53
NOISE :51 52 1 2 3 4 5 6 7 8 9 10 31 32 33 34 35 36 37 39 40 11 12 13 14 15 16 17 19 20 21 22 23 24 25 26 27 28 29
30 41 42 43 44 45 46 47 48 49 50
Exec time : 341834

cluster 1 : 18 38 53
NOISE :51 52 31 32 33 34 35 36 37 39 40 11 12 13 14 15 16 17 19 20 21 22 23 24 25 26 27 28 29 30 41 42 43 44 45 46
47 48 49 50 1 2 3 4 5 6 7 8 9 10
Exec time : 347871

cluster 1 : 18 38 53
NOISE :33 34 35 36 37 39 40 41 42 43 44 45 46 47 48 49 50 51 52 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 19 20 21
22 23 24 25 26 27 28 29 30 31 32
Exec time : 355263

cluster 1 : 18 38 53
NOISE :33 34 35 36 37 39 40 41 42 43 44 45 46 47 48 49 50 51 52 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 19 20 21
22 23 24 25 26 27 28 29 30 31 32
Exec time : 377924
```

Observations:

Number of blocks	Number of threads	Execution Time
65536	1	1.078
32768	2	1.802
16384	4	1.785
8192	8	1.851
4096	16	1.518
2048	32	1.898
1024	64	1.767
512	128	1.861
256	256	1.54
128	512	1.84
64	1024	1.993
32	2048	1.916
16	4096	2.114
8	8192	2.507


Execution Time vs. Number of blocks
Execution Time
3
2
1
0
1
10
100
1000
10000
Number of blocks


Execution Time vs. Number of threads
Execution Time
3
2
1
0
1
10
100
1000
10000
Number of threads

Inference:

- Execution time, graph and inference are based on hardware configuration.
- The performance of the program remains similar, even with varying block size and thread count.
- A block can have utmost 1024 threads.