1. Introduction
Using a large number of cores in GPU for enhancing the speedup and performance, GPGPU has become the trend of computation acceleration. CUDA is one approach of GPGPU, it is a parallel computing architecture developed by NVIDIA. CUDA is specifically designed to use multiple cores in GPU to improve the performance via parallel computation. CUDA is also a “programming language”. For instance, CUDA can be programmed in C/C++ with a few extensions, because in CUDA C/C++, GPU is treated as a compute device which could execute a large number of parallel computations, and operating as a co-processor to the CPU.

2. Example of CUDA
By comparing the calculation time of minimum spanning tree (MST), this example shows the improvement between CUDA C and sequential C.

The running environment shown below:

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel i7 M640 @2.8 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphics</td>
<td>NVS 3100m w/ 16 CUDA cores</td>
</tr>
<tr>
<td>Memory</td>
<td>8G DDR3</td>
</tr>
</tbody>
</table>

2.1 Minimum Spanning Tree & Prim’s Algorithm

It’s a set of vertices (A, B, C, D and E) and the edges that interconnect them, each edge
attributes different weights. A minimum spanning tree is a subset of the edges of the graph, so there’s a path from any node to any other node and that the sum of the weights of the edges is minimum.

Prim’s Algorithm found by Robert C. Prim, is one of the classic algorithms for MST. This algorithm builds the minimal spanning tree by iteratively adding nodes into a working tree:

Step 1: Start with a tree which contains only one node.
Step 2: Identify a node (outside the tree) which is closest to the tree and add the minimum weight edge from that node to some node in the tree and incorporate the additional node as a part of the tree.
Step 3: If there is less then \((n - 1)\) edges in the tree, repeat Step 2.

2.2 CUDA Implementation

```c
#include <stdio.h>
#include <string.h>

__shared__ int n;
// The number of nodes in the graph
__shared__ int dist[100][100];
// the distance between node i and node j
__shared__ char inclde[100];
// inclde[i] is 1 if the node i is already in the minimum spanning tree; 0 otherwise
__shared__ int d[100];
// the distance between node i and the minimum spanning tree
__shared__ int w[100];

__global__ void MST(int target) {
    int i;
    for (i = 0; i < n; ++i)
        if ((dist[target][i] != 0) && (d[i] > dist[target][i]))
            
```
\[ d[i] = \text{dist}[\text{target}[i]]; \]
\[ w[i] = \text{target}; \]

\} \}

\textbf{int main(int argc, char *argv[])}
\{
    FILE *f = fopen("dist.txt", "r");
    fscanf(f, "%d", &n);
    int i, j;
    for (i = 0; i < n; ++i)
        for (j = 0; j < n; ++j)
            fscanf(f, "%d", &dist[i][j]);
    fclose(f);

    // Initialise d with infinity
    for (i = 0; i < n; ++i)
        d[i] = 100000;

    // Mark all nodes as NOT being in the minimum spanning tree
    for (i = 0; i < n; ++i)
        included[i] = 0;

    // Add the first node to the tree
    printf("Adding node \%c\n", 0 + 'A');
    included[0] = 1;

    int total = 0;
    int s; // tree size
    for (s = 1; s < n; ++s) {
        // Find the node with the smallest distance to the tree
        int min = -1;

        for(i=0;i<n/p;++i)
        {
            c[\*dist[i][i]+j]=0;
            visited[\*dist[i][i]+j]=0;
        }
        if(i==0)
        {
            visited[0]=1;
__syncthreads();

__shared__ int flag;
for(i=1;i<n;++i)
{
    d[i]=MAX;
    nd[i].d=d[i];
    flag=0;

    for(k=0;k<n/p;++j)
    {
        if(visited[*dist[i]*j+k]!=1)
        {
            if(d[i]>w[*dist[i]*j+k][c[*dist[i]*j+k]])
            {
                d[i]=w[*dist[i]*j+k][c[*dist[i]*j+k]]; 
                i=k; 
                flag=1; 
            }
        }
    }
}

if(flag==1)
{
    nd[i].d=w[*dist[i]*j+l][c[*dist[i]*j+l]]; 
    nd[i].v1=dist[i]*j+l; 
    nd[i].v2=c[*dist[i]*j+l];
}
__shared__ node nd1;
__shared__ int m;
__shared__ int l;

if(i==0)
{
    m=3;
    l=(int)(log((double)(m-1))/log((double)2));
}
int h;
__syncthreads();
for(h=0;h<l;h++)
{
    if((int)pow((double)2,h+1)==0 && h<m && (int)pow((double)2,h)<m)
    {
        if(nd[i+(int)pow((double)2,h)],d<nd[i].d)
            nd[i]=nd[i+(int)pow((double)2,h)];
    }

    printf("Adding edge %c-%cWn", w[min] + 'A', min + 'A');
    total += d[min];
    __syncthreads();
}

if(i==0)
{
    nd1=nd[0];
    tree[j-1].v1=nd1.v2;
    tree[j-1].v2=nd1.v1;
    visited[nd1.v1]=1;
}

__syncthreads();

for(k=0;k<n/p;++]
{
    if(visited[i*dist[i]][k+1]!=1)
    {
        if(w[i*dist[i]][k+1]<w[i*dist[i]][k+1][c[i*dist[i]][k]])
        {
            c[i*dist[i]][k]=nd1.v1;
        }
    }

    printf("Adding edge %c-%cWn", w[min] + 'A' + nd1.v1, min + 'A' + nd1.v1);
    total += d[min];
    __syncthreads();
}

printf("Adding edge %c-%cWn", w[min] + 'A' + nd1.v2, min + 'A' + nd1.v2);
    total += d[min];
{ }
    printf("Total distance: %d\n", total);
    return 0;
}

2.3 Simulation Results

The 10 nodes sample input file looks as below, other larger sample file is created by random number generator.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>0</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>13</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
<td>5</td>
<td>15</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>10</td>
<td>13</td>
<td>5</td>
<td>0</td>
<td>20</td>
<td>5</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>15</td>
<td>10</td>
<td>20</td>
<td>0</td>
<td>7</td>
<td>15</td>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>18</td>
<td>5</td>
<td>7</td>
<td>0</td>
<td>12</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>15</td>
<td>12</td>
<td>0</td>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>17</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>J</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>18</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The sheet of running time:

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>CUDA (ms)</th>
<th>C (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.186</td>
<td>0.198</td>
</tr>
<tr>
<td>100</td>
<td>0.479</td>
<td>1.61</td>
</tr>
<tr>
<td>200</td>
<td>0.658</td>
<td>2.87</td>
</tr>
<tr>
<td>500</td>
<td>1.11</td>
<td>4.21</td>
</tr>
</tbody>
</table>
3. Conclusion
The MST begins from one node, so there is no obvious speedup between CUDA and sequential C in the small size sample. However, CUDA can calculate the different node by parallel computation, as shown in the results, the larger size with the sample, the more improvement CUDA has. Consequentially, CUDA provide higher performance than CPU in parallel computation. Besides, CUDA shows high possibility and potential of general purpose computing. Including CUDA, GPGPU will become a trend of high computational throughput on parallel computation problems in the future.