

HPC Paper

by Ethan Randell

FILE	4154_ETHAN_RANDELL_HPC_PAPER_107698_490900296.DOCX (29.08K)		
TIME SUBMITTED	15-MAY-2019 12:13PM (UTC+0100)	WORD COUNT	2880
SUBMISSION ID	107285151	CHARACTER COUNT	14779

High Performance and Computing for Games& Visualisation Assignment 2

Student name: Ethan Randell

Student number: P150496

Module number: M6X01099

Contents

Introduction	2
Literature Review	2
Dijkstra's algorithm	2
A* algorithm.....	3
Pythagoras	3
MY ALGORITHM	3
IMPLEMENTATION OF THE ALGORITHM FOR THE CPU	4
IMPLEMENTATION OF PARALLELIZATION OF THE ALGORITHM	4
RESULTS AND EVALUATION	7
CONCLUSIONS AND FUTURE WORK	8
References	8
Appendix	9

Introduction

This report is examining which piece of hardware can run a complex program faster; the CPU, or GPU. Firstly, the literature that is associated with the program will be reviewed to give an overview of the background information relating to the algorithm being used. This will be followed by an implementation of the algorithm, using C++ for the CPU version and CUDA for the version running on the GPU. After this, the results will be compared and contrasted to each other so as to evaluate which piece of hardware was able to execute the algorithm the fastest. The report will be tied up with a conclusion and the approach to future work will be discussed.

Literature Review

The algorithm that will be used for this experiment will be shortest pathfinding algorithm but more looking into the distance checking between two points function of this algorithm. Firstly, the two main shortest pathfinding algorithms will be reviewed then will be followed by a review of distance checking.

Dijkstra's algorithm

The Dijkstra algorithm is it a shortest path algorithm it was invented by the Dutch computer scientist Edsger Dijkstra in 1956 and was then distributed three years after, the algorithm itself is a graph search algorithm which figures out the shortest path for a graph which has non-negative edge path costs.

So for each given vertex (node) in the graph the algorithm is trying to find the path with the lowest cost (this is the shortest path) between one vertex and every other vertex in the graph [1].

There are many disadvantages when using Dijkstra's algorithm, the biggest problem with it is that the algorithm looks through the entire graph non-selectively to find the shortest path, its useful if trying

to find the shortest path to every node in the graph which Dijkstra's algorithm was mainly designed for but it's not useful in point to point pathfinding [2].

A* algorithm

The A* algorithm is another path finding algorithm and like Dijkstra's algorithm it is also used in graph travel as well. The algorithm is very easy to implement and is more efficient than the Dijkstra algorithm, most pathfinding systems used now use some modified form of the A* as the main algorithm, the A* algorithm is far superior to Dijkstra's algorithm because it's designed for point to point pathfinding [3].

The algorithm mainly works the same as Dijkstra, instead of going to every open node with a low cost instead it chooses the node which is most likely to lead to the shortest path. The A* algorithm repeats the same process over and over to look for the shortest path [3]. The A* algorithm is the same as Dijkstra's algorithm but in A* adds the heuristic value [4] to the cost to the nodes so far then the algorithm will choose which node to go to base on this cost.

Pythagoras

The Pythagoras theorem is a mathematical equation; the equation can be used for multiple different problems mainly for the relationship between triangles.

$$c = \sqrt{a^2 + b^2}$$

The equation above is the Pythagoras theorem, examples of the Pythagoras theory being used is for example is to find the length of a hypotenuse of a right triangle, another use for the theorem is to find the diagonal distance across a square but the most relevant use of Pythagoras for this assignment is to find out the distance between two points.

MY ALGORITHM

The algorithm that has been chosen for this assignment is an implementation of the Pythagoras distance check equation. This algorithm was chosen because most pathfinding algorithms use distance checking to check the distance between the current node and the node it's navigating too.

The distance equation is:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

For example using the equation above, if $x_1 = 1$ and $x_2 = 5$ you would minus x_1 from x_2 this would then produce a single value that would be squared. The same process would be applied for the y values so if $y_1 = 2$ and $y_2 = 9$. Once you have the overall x and y values you would square root them and that would then produce the distance between the two points.

IMPLEMENTATION OF THE ALGORITHM FOR THE CPU

So the program begun by generating 5 arrays, four of them held a thousand values these four arrays corresponded with the x and y values that was shown in the section above, the fifth array which was labelled P (Pythagoras) and was generated to hold the thousand results that the algorithm should produce.

```
int px1[1000];
int py1[1000];
float p[1000];
int px2[1000];
int py2[1000];
```

The arrays are called px and py due to the fact visual studio already has some x and y values already stored in background code and whenever the algorithm tried to run errors occurred.

Next the main would contain five for loops four of them were generated to populate the four main arrays with values.

```
for (int i = 0; i < 1000; i++)
{
    px1[i] = i;
}
```

Which were then used for the fifth for loop which contained the Pythagoras algorithm these values were given to the Pythagoras algorithm to calculate and the results of the calculation was then put into the P array. Once the P array had received the results it would then display these results by using a print statement for the user to look at the results.

```
for (int i = 0; i < 1000; i++)
{
    temp = ((px1[i] - px2[i])
            * (px1[i] - px2[i]) + (py1[i] - py2[i])
            * (py1[i] - py2[i]));
    p[i] = sqrt(temp);

    //

    printf("%f Distances", p[i]);
}
```

IMPLEMENTATION OF PARALLELIZATION OF THE ALGORITHM

For the CUDA version of the algorithm, first a global device is created this device will be called the add kernel the device will have the Pythagoras algorithm stored inside it.

```
global__ void addKernel(float *p, float *px1, float *py1, float *px2, float *py2)
```

```

{
    int i = threadIdx.x;
    p[i] = sqrt((px1[i] - px2[i]) * (px1[i] - px2[i]) + (py1[i] - py2[i]) * (py1[i] - py2[i]));
}

```

Then there's the main this will store the arrays for the x, y coordinates which have a thousand values and the p values.

```

const int arraySize = 10000;
float px1[arraySize]; // = { 1, 2, 3, 4, 5 };
float py1[arraySize]; // = { 10, 20, 30, 40, 50 };
float px2[arraySize];
float py2[arraySize];
float p[arraySize]; // = { 0 };

```

Underneath the arrays there are the for loops which are the same as the ones in the CPU version.

```

for (int i = 0; i < 10000; i++)
{
    px1[i] = i;
}

```

Underneath these for loops there will be the print function to print out the final distances.

```

for (int i = 0; i < 1000; i++)
{
    printf("%f Distances", p[i]);
}

```

After the main there will be the actual kernel which will take the add kernel device and will calculate all 1000 distances simultaneously. That is what parallelization is it does multiple calculations at the same time unlike CPU code which does the calculations one at a time.

This piece of code chooses which graphics card it wants to run the CUDA code on this can be change on a multiple GPU system.

```

cudaStatus = cudaSetDevice(0);
if (cudaStatus != cudaSuccess) {

```

```

    fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");
    goto Error;
}

```

This piece of code allocates GPU buffers for the five arrays (four input, one output).

```

cudaStatus = cudaMalloc((void**)&dev_p, arraySize * sizeof(float));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

```

This piece of code copies the values inside the arrays from the host memory into the GPU buffers this code repeated four times for each of the input arrays

```

    cudaStatus = cudaMemcpy(dev_px1, px1, arraySize * sizeof(float),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

```

This piece of code is launched on the GPU and each element is allocated its own individual thread.

```

    int bCount = arraySize / 500;
    addKernel<<<bCount, 500>>>(dev_p, dev_px1, dev_py1, dev_px2, dev_py2);

    // Check for any errors launching the kernel
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
        goto Error;
    }

```

This piece of code waits for the kernel to finish and will return any errors that it has encountered during the launch.

```

    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching
addKernel!\n", cudaStatus);
        goto Error;
    }

```

This piece of code copies out the P array vectors from the GPU buffer to the host memory.

```

    cudaStatus = cudaMemcpy(p, dev_p, arraySize * sizeof(float), cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

```

RESULTS AND EVALUATION

CPU RESULTS

The processor that has been used for these tests is the Intel Core I7 processor. The results that're in the table are measured in microseconds (I couldn't find out how to change the results into miliseconds) and the results is measuring how long the main function will take to calculate a number of distances between points and will find out the average.

Amount of distances	Average Elapsed Inclusive Time (measured in microseconds)	Average Elapsed Exclusive Time (measured in microseconds)
1000	Test 1 = 322,560 Test 2 = 331,778 Test 3 = 343,040 Average =332,459.3	Test 1 = 306,178 Test 2 = 297,986 Test 3 = 308,222 Average =304,128.6
10000	Test 1 = 3,251,204 Test 2 = 3,626,472 Test 3 = 3,269,640 Average =3,382,438	Test 1 = 3,217,412 Test 2 = 3,228,678 Test 3 = 3,217,414 Average =3,221,168
100000	Test 1 = 115,126,504 Test 2 = 115,126,504 Test 3 = 66,957,446 Average =99,070,158	Test 1 = 115,089,638 Test 2 = 115,089,638 Test 3 = 66,920,582 Average =99,033,286

GPU RESULTS

The graphics card that has been used for these tests is the NVIDIA 1080 GTX.

The results that're in the table are measured in milliseconds and the results is measuring how long the add with Cuda function will take to calculate a number of distances between points and will find out the average.

Amount of distances	Average Elapsed Inclusive Time (measured in milliseconds)	Average Elapsed Exclusive Time (measured in milliseconds)
1000	Test 1 = 490.21 Test 2 = 320.19 Test 3 = 318.39 Average =376.263	Test 1 = Unknown Test 2 = Unknown Test 3 = Unknown Average = Unknown
10000	Test 1 = 454.11 Test 2 = 397.09 Test 3 = 398.39 Average =416.53	Test 1 = Unknown Test 2 = Unknown Test 3 = Unknown Average = Unknown
100000	The program couldn't handle the amount of data being passed through	The program couldn't handle the amount of data being passed through

The results show that the GPU can handle small amounts of data very quickly unlike the CPU but the CPU can handle a large amount of data unlike the GPU which every time it tried to run one hundred thousand values the program would crash.

CONCLUSIONS AND FUTURE WORK

So in conclusion the graphics card is a very powerful piece of software it can calculate small amounts of data faster due to the parallelization of the program unlike the CPU, but when it comes to large amounts of calculations the CPU is more optimized for it because if you wanted to try and run a large amount of data you would have to change the amount of threads used in that program and can become a very tedious task when you want the calculations done as soon as possible, but the fact cannot be denied that the CPU is very slow when doing any amount of calculations, so in the final conclusion the graphics card is the better piece of hardware to choose for quick calculations.

For work in the future the CUDA program will be upgraded to try and run larger amounts of data and also both pieces of code will be upgraded to try and use either the Dijkstra algorithm or the A* algorithm and will try and have some sort of graphical output.

References

- [1] N. A. A. E. K. I. L. E. N. N. Nadira Jasika, "Dijkstra's shortest path algorithm serial and parallel execution performance analysis," Opatija, 2012.

- [2] J. F. Ian Millington, "Dijkstra," in *Artificial Intelligence for games*, Morgan Kaufmann publishers , 2009, pp. 204-215.
- [3] Ian Millington, "A*," in *Artificial intelligence for games* , Morgan Kaufmann, 2009, pp. 215-237.
- [4] A. K. m. David L. Poole, "Heuristic Search," unknown unknown unknown . [Online]. Available: https://artint.info/html/ArtInt_56.html. [Accessed 13 May 2019].

Appendix

CPU CODE

```
#include <stdio.h>

#include <math.h>


int px1[100000];
int py1[100000];
float p[100000];
int px2[100000];
int py2[100000];


int main(void)
{
    for (int i = 0; i < 100000; i++)
    {
        px1[i] = i;
    }

    for (int i = 0; i < 100000; i++)
    {
```

```
        py1[i] = i;
    }

    for (int i = 0; i < 100000; i++)
    {
        px2[i] = 21;
    }

    for (int i = 0; i < 100000; i++)
    {
        py2[i] = 56;
    }

    float temp;

    for (int i = 0; i < 100000; i++)
    {
        temp = ((px1[i] - px2[i]) * (px1[i] - px2[i]) + (py1[i] - py2[i]) * (py1[i] - py2[i]));
        p[i] = sqrt(temp);

        //printf("%f Distances", p[i]);
    }

    return 0;
}
```

GPU CODE

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <math.h>
#include <stdio.h>

cudaError_t addWithCuda(float* p, float* px1, float* py1, float* px2, float* py2, const int arraySize);

__global__ void addKernel(float *p, float *px1, float *py1, float *px2, float *py2)
{
    int i = threadIdx.x;

    p[i] = sqrt((px1[i] - px2[i]) * (px1[i] - px2[i]) + (py1[i] - py2[i]) * (py1[i] - py2[i]));
}

int main()
{
    const int arraySize = 10000;
    float px1[arraySize]; // = { 1, 2, 3, 4, 5 };
    float py1[arraySize]; // = { 10, 20, 30, 40, 50 };
    float px2[arraySize];
    float py2[arraySize];
    float p[arraySize]; // = { 0 };
    // initialize for loops grab them from cpu version

    for (int i = 0; i < 10000; i++)
    {
        px1[i] = i;
    }

    for (int i = 0; i < 10000; i++)
```

```

    {
        py1[i] = i;
    }

    for (int i = 0; i < 10000; i++)
    {
        px2[i] = 21;
    }

    for (int i = 0; i < 10000; i++)
    {
        py2[i] = 56;
    }

// Add vectors in parallel.
cudaError_t cudaStatus = addWithCuda(p, px1, py1, px2, py2, arraySize);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "addWithCuda failed!");
    return 1;
}

/*for (int i = 0; i < 1000; i++)
{
    printf("%f Distances", p[i]);
}*/

// cudaDeviceReset must be called before exiting in order for profiling and
// tracing tools such as Nsight and Visual Profiler to show complete traces.
cudaStatus = cudaDeviceReset();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaDeviceReset failed!");
}

```

```

        return 1;
    }

    return 0;
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(float* p, float* px1, float* py1, float* px2, float* py2, const int arraySize)
{
    float *dev_px1 = 0;
    float *dev_py1 = 0;
        float *dev_p = 0;
        float *dev_px2 = 0;
        float *dev_py2 = 0;

    cudaError_t cudaStatus;

    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");
        goto Error;
    }


    // Allocate GPU buffers for three vectors (two input, one output) .
    cudaStatus = cudaMalloc((void**)&dev_p, arraySize * sizeof(float));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

```

```

cudaStatus = cudaMalloc((void**)&dev_px1, arraySize * sizeof(float));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

cudaStatus = cudaMalloc((void**)&dev_py1, arraySize * sizeof(float));
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

    cudaStatus = cudaMalloc((void**)&dev_px2, arraySize * sizeof(float));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_py2, arraySize * sizeof(float));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

// Copy input vectors from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_px1, px1, arraySize * sizeof(float), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

```

```

cudaStatus = cudaMemcpy(dev_py1, py1, arraySize * sizeof(float), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

    cudaMemcpy(dev_px2, px2, arraySize * sizeof(float),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    cudaMemcpy(dev_py2, py2, arraySize * sizeof(float),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

    int bCount = arraySize / 500;
    // Launch a kernel on the GPU with one thread for each element.
    addKernel<<<bCount, 500>>>(dev_p, dev_px1, dev_py1, dev_px2, dev_py2);

    // Check for any errors launching the kernel
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
        goto Error;
    }

    // cudaDeviceSynchronize waits for the kernel to finish, and returns

```

```

// any errors encountered during the launch.
cudaStatus = cudaDeviceSynchronize();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n",
        cudaStatus);
    goto Error;
}

// Copy output vector from GPU buffer to host memory.
cudaStatus = cudaMemcpy(p, dev_p, arraySize * sizeof(float), cudaMemcpyDeviceToHost);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

Error:
    cudaFree(dev_p);
    cudaFree(dev_px1);
    cudaFree(dev_py1);
    cudaFree(dev_px2);
    cudaFree(dev_py2);

    return cudaStatus;
}

```

HPC Paper

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

41 / 100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

RESEARCH-INFORMED LITERATURE

 Refs very light. 2 of the refs are to the same text. (good text though).

3

KNOWLEDGE AND UNDERSTANDING

 Dijkstra and A* identified and discussed - this could be clearer. An emphasis was placed on the code description - it would have been better to focus on the literature more here. Suitable discussion of the distance heuristic and its suitability for parallelisation.

7

ANALYSIS

 Euclidian distance identified as the calculation that needs to be parallelised. Limited to 2 or 3 co-ordinate points which could also limit the extent of your timing tests. Suitable tools identified for carrying out these tests.

4.25

PRACTICAL APPLICATION AND DEP.

 Limited set of result. Make sure your data arrays are stored on the heap, otherwise a system stack

24

overflow might occur. Timing data collected for both CPU and GPU though the application could be more sophisticated and fully implement the algorithm.

SKILLS FOR PROFESSIONAL PRAC.



Report structured. Refs include (not many though).
No abstract.

2.5