CHAPTER 5
IMPLEMENTATION AND TESTING

5.1 Implementation

The data structure in this project uses 2D arrays, 2D arrays have advantages such as easy access, can store many elements and low memory usage. to divide the work into 2 here used extends Thread in Java programming, Thread makes it easy for programmers to process data with constructor data will be directly processed by Threads.

```
10.   public class Perkalian extends Thread{
11.         int Matrix1[][],Matrix2[][],MatrixHasil[][];
12.         int nomer;
13.         int size;
14.       int cols;
15.         public  Perkalian(int  Matrix1[][],int  Matrix2[][],int
   MatrixHasil[][],int nomer) {
16.               System.out.println("THREAD NOMER "+nomer);
17.           this.Matrix1= Matrix1;
18.           this.Matrix2= Matrix2;
19.           this.MatrixHasil= MatrixHasil;
20.           this.nomer= nomer;
21.       }
```

Here is the source code to complete the matrix multiplication in the CPU. The run function will run every time the thread is in the call, the calculation starts from making the loop i as the line, the line will increase according to the number of threads (2 threads), loop j as column and loop k as help. each calculation needs to be saved to a temp (temp) variable and terminated by replacing the index calculation with temporary variable (temp).

```
1. public void run() {
2.
3.        for(int i=nomer;i<Matrix1.length; i=i+2)
4.        {
5.          for(int j = 0;j<Matrix2[0].length;j++)
6.          {
7.                int temp=0;
8.                for(int k=0;k<Matrix1[0].length;k++)
9.               {
10.                temp = temp+(Matrix1[j][k] * Matrix2[k[i]);
```

```
11.                 }
12.           MatrixHasil[j][i]=temp;
13.                       }
14.            }
15.         }
```

The following is the source code to complete the additional matrix on the CPU. Start by creating a run function that will run every time a thread is called. in line 2 is a loop as a line that will increase as many threads as it is in line 4 ie loop as column and at line 6 is a calculation where the result of matrix index is generated from index matrix calculation 1 summed with matrix 2.

```
1.     public void run() {
2.         for(int i=nomer;i<Matrix1.length; i=i+2)
3.         {
4.           for(int j = 0;j<Matrix1[0].length;j++)
5.           {
6.                 MatrixHasil[i][j]=Matrix1[i][j]+Matrix2[i][j];
7.         }
8.     }
```

To horizontal flipping which horizontally requires a temporary variable to store the matrix value for swapping. This algorithm performs a loop until all the data in the line and each iterate will increase with the number of threads in this experiment the author uses 2 threads then the second loop needs to help restrict the data to swap if not using this swap it will cause the data Failure will return. to the old place Temp will store the point of the matrix row and the column for the swap then the row and column matrices will change to row matrices (i) and columns to max column length then minus indexes in j and - 1. (minus one because array starts from zero not t one) the last step change the row matrix (i) and column to max column length then minus index at j and - 1. to temp (temp already save first array).

```
1. public void run() {
2.         int temp;
3.         for(int i=nomer;i<Matrix1.length; i=i+2)
4.         {
5.           for(int j = 0;j<Math.floor(Matrix1[0].length/2);j++)
6.           {
7.                 temp = Matrix1[i][j];
8.              Matrix1[i][j] = Matrix1[i][Matrix1[0].length-j-1];
9.                 Matrix1[i][Matrix1[0].length-j-1]=temp;
10.            }
```

```
11.
12.        }
```

In a vertical transform it needs a temporary variable to store the value matrix for swapping. This algorithm loops from the number of threads to the row divided by 2 and each iterate will add the number of threads, the second loop will help the swapping column. Temp will store row matrix and iterate columns then row matrix and column iterate will swap to matrix row length minus index i and -1 (-1 because array start from zero) and column j and last step matrix row length is reduced by index i and - 1 and column j will be swapped to temp.

```
1.  public void run() {
2.         int temp;
3.         for(int i=nomer;i<Math.floor(Matrix1.length/2); i=i+2)
4.         {
5.             for(int j = 0;j<Matrix1[0].length;j++)
6.           {
7.                 temp = Matrix1[i][j];
8.                 Matrix1[i][j] = Matrix1[Matrix1.length-i-1][j];
9.                 Matrix1[Matrix1.length-i-1][j]=temp;
10.          }
11.
12.        }
13.   }
```

Below is the GPU CUDA programming the source code to calculate the matrix multiplication by using 2 threads. In line 1 and 2 is the preparation of blocks to complete the matrix multiplication. In the first line set the number of threads this source code works with array 1d (x), 2d (x, y) and 3d (x, y, z). In this project using 2d arrays then the number of blocks will be calculated by the number of N (N is the length of the array) divided by the threadPerblock index x and the y index and the number of blocks and threads used for kernel lauch in CUDA on line 3.

Cuda has 3 functions that can be used like device, host and global function but in this project function in CUDA will be written globally on line 5. this function will be executed N times in parallel by different CUDA N thread.

For indexing in CUDA using int i = blockIdx.x * blockDim.x + threadIdx.x; This command will count all the blocks multiplied by dim with threads. Then create a temporary variable in line 11. on line 14 to calculate the multiplication requires the help of the loop because the kernel executes the command as much as N time kernel and can not use double loop as in CPU usage. line 16 temp will be added by loop helper. And on line 18 save the result to the dev c array to send back to the host.

```
1. dim3 threadPerBlock(2,2);
2. dim3 numBlocks(N/threadPerBlock.x,N/threadPerBlock.y);
3. Matrix_Multi<<<numBlocks,threadPerBlock>>>(dev_a,dev_b,dev_c);
   // kernel program
4.
5. __global__  void  Matrix_Multi  (  int  dev_a[][N]  ,  int  dev_b[]
   [N] , int dev_c[][N])
6. {
7.       //Get the id of thread within a block
8.    int i = blockIdx.x * blockDim.x + threadIdx.x;
9.    int j = blockIdx.y * blockDim.y + threadIdx.y;
10.
11.            int hasil=0;
12.            int e =0;
13.            if(i<N && j <N)
14.            {     for(e=0;e<N;e++)
15.              {
16.                hasil=hasil+ (dev_a[i][e]*dev_b[e][j]);
17.              }
18.              dev_c[i][j]=hasil;
19.             }
20.    }
```

The following is the source code in CUDA to do the addition matrix. To do the addition does not require a loop because CUDA will run the kernel as much as n time in parallel by different CUDA threads. so we only need to limit the execution of N times (N is the length of the array) by using if on line 5 and line 6 the calculation by using indexing i and j (details above) and the result of addition is saved to dev c to be returned to host (CPU).

```
1. __global__ void Matrix_Add ( int dev_a[][N] , int dev_b[][N] ,
   int dev_c[][N])
2. {
3.     int i = blockIdx.x * blockDim.x + threadIdx.x;
4.       int j = blockIdx.y * blockDim.y + threadIdx.y;
5.      if (i < N  && j <N)
6.          dev_c[i][j]=dev_a[i][j] + dev_b[j][j];
```

```
7. }
```

This source code is a vertical flip matrix using CUDA programming, the CUDA indexing details exist in the multiplication of the matrix source code. the program will take the function parameter when called and stored in line 6. Then on line 7 is the temporary variable to save swapping, at line 10 temp variable will be set to save the array value of dev_a row and column, then line 11 line of array dev_a and column will changes to the dev_a array of the max-i-1 index row (minus 1 because the array starts from zero) and the last column j on line 12 of the dev array of the max-i-1 row index (minus 1 because the array starts from zero) and the column j changed to temp variable.

```
1. __global__ void Matrix_Flip ( int dev_a[][N] , int size)
2. {
3.       //Get the id of thread within a block
4.     int i = blockIdx.x * blockDim.x + threadIdx.x;
5.       int j = blockIdx.y * blockDim.y + threadIdx.y;
6.     int total=size;
7.       int hasil=0;
8.       if(i<N && j <N)
9.       {
10.            hasil=dev_a[i][j];
11.            dev_a[i][j] = dev_a[(total-i-1)][j];
12.            dev_a[(total-i-1)][j]=hasil;
13.            }
14.    }
```

This source code is a Horizontal flip matrix using CUDA programming, CUDA indexing details exist in the multiplication of the matrix source code. the program will take the function parameter when called and stored in line 6. Then on line 7 is temporary variable to save swapping, at line 10 temp variable will be set to save the array value of dev_a row and column, then at line 11 the array dev_a row and column will switch to the dev array of an index row i and the max-j-1 column (minus 1 because the array starts from zero) then on row 12 the array dev i index row and max-j-1 column (minus 1 because the array start from zero) swap to the temp variable.

```
1. __global__ void Matrix_Flip ( int dev_a[][N] , int size)
2. {
3.       //Get the id of thread within a block
4.       int i = blockIdx.x * blockDim.x + threadIdx.x;
5.      int j = blockIdx.y * blockDim.y + threadIdx.y;
```

```
6.        int total=size;
7.        int hasil=0;
8.        if(i<N && j <N)
9.        {
10.             hasil=dev_a[i][j];
11.             dev_a[i][j] = dev_a[i][(total-j-1)];
12.             dev_a[i][(total-j-1)]=hasil;
13.       }
14.   }
```

5.2 Testing

The result of matrix dimension effect with time calculation on CPU and GPU (on CPU will be tested on different processing unit).

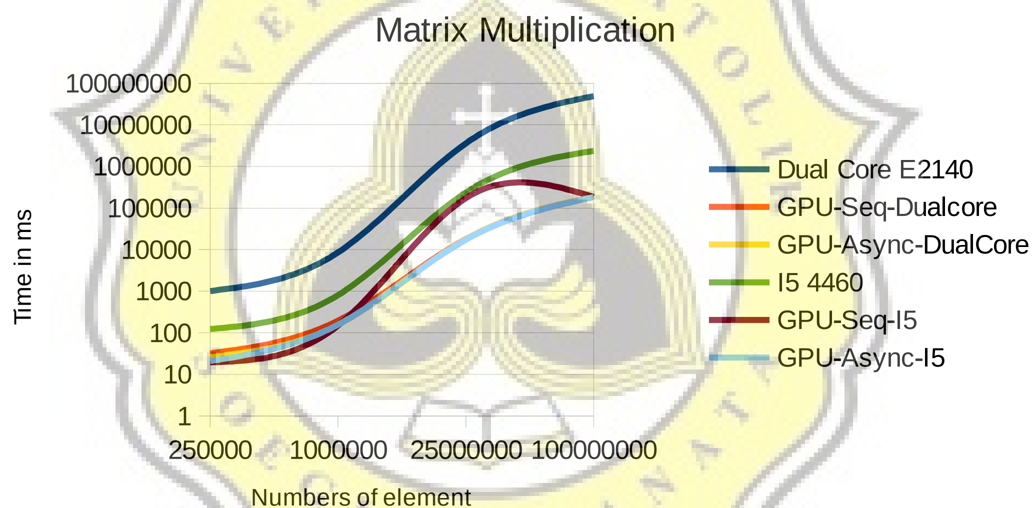

Illustration 5.1: Time Computation Matrix Multiplication

From the matrix multiplication graph above, computation time has increased significantly when data elements of more than 20 million of CPU Dual Core E2140. This makes the Core i5 4460 CPU better than the Dual Core E2140 CPU because the Dual Core CPU has a lower clock than the Core i5 4460 so the graph looks like it's linear.

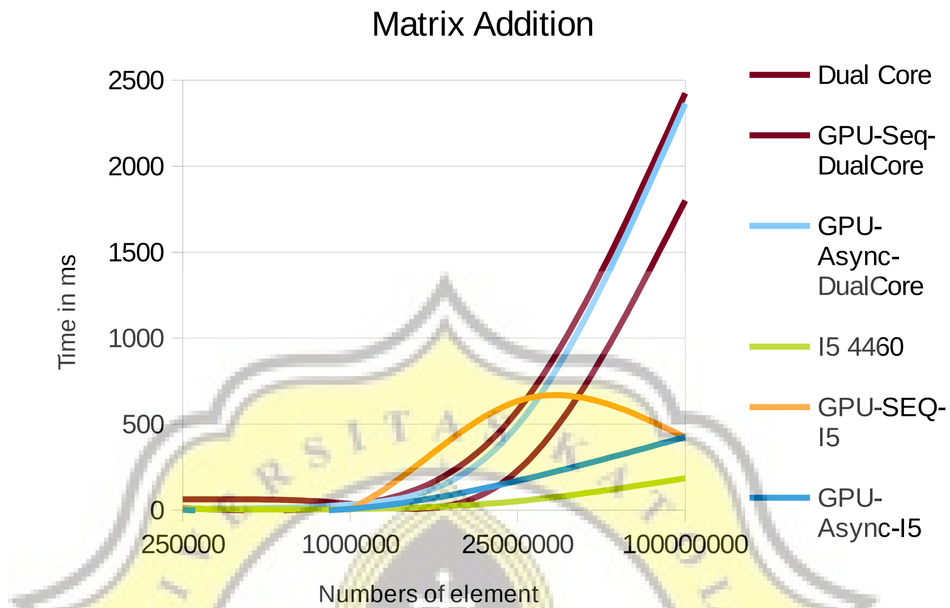

Illustration 5.2: Time Computation Matrix Addition

According to the above graphic addition matrix has a computational time lower on the amount of data below 1 million and increased in the number 10 million - 100 million data but no significant increase for the Core i5 4460 processor.

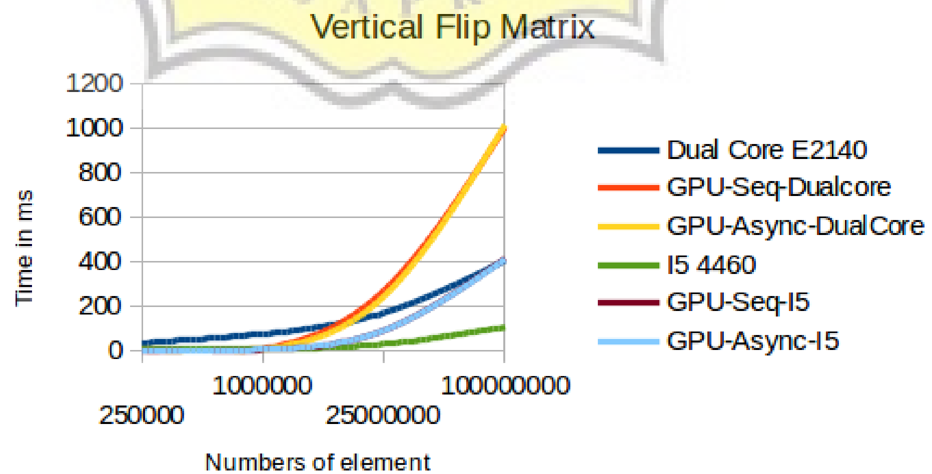

Illustration 5.3: Time Computation Vertical Flip Matrix

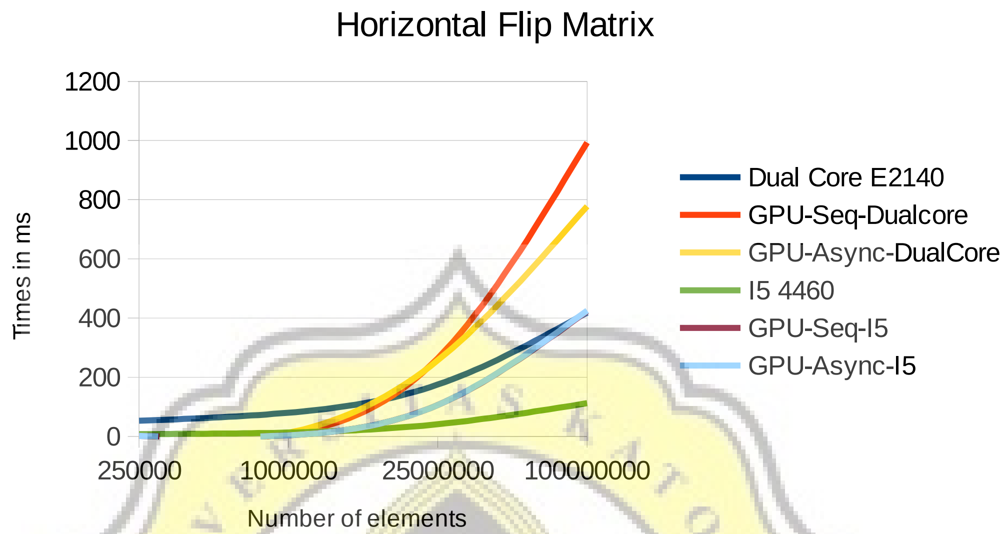

Illustration 5.4: Time Computation Horizonatal Flip Matrix

Based on the graph above Horizontal flip matrix and Vertical flip matrix increase the calculation time straight upward to form linear graphic.

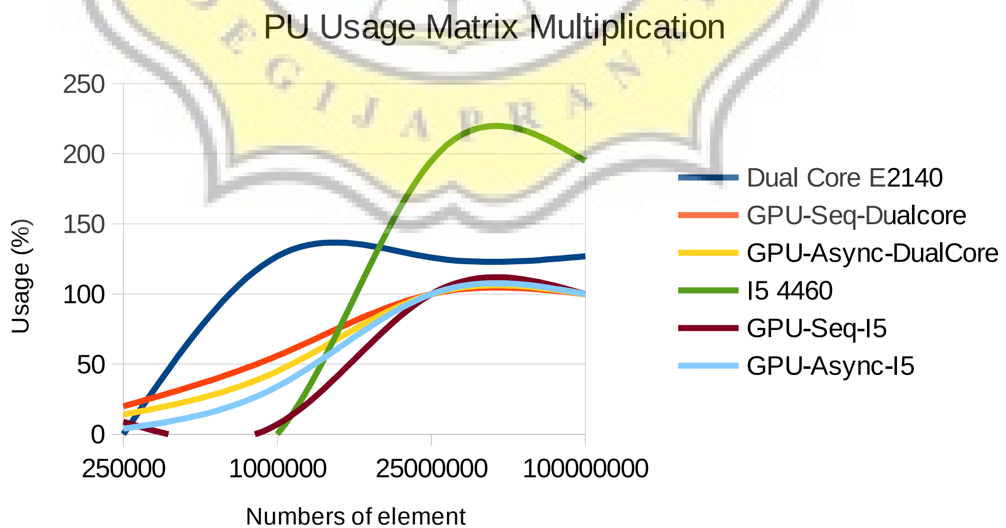

Illustration 5.5: Matrix Multiplication PU usage

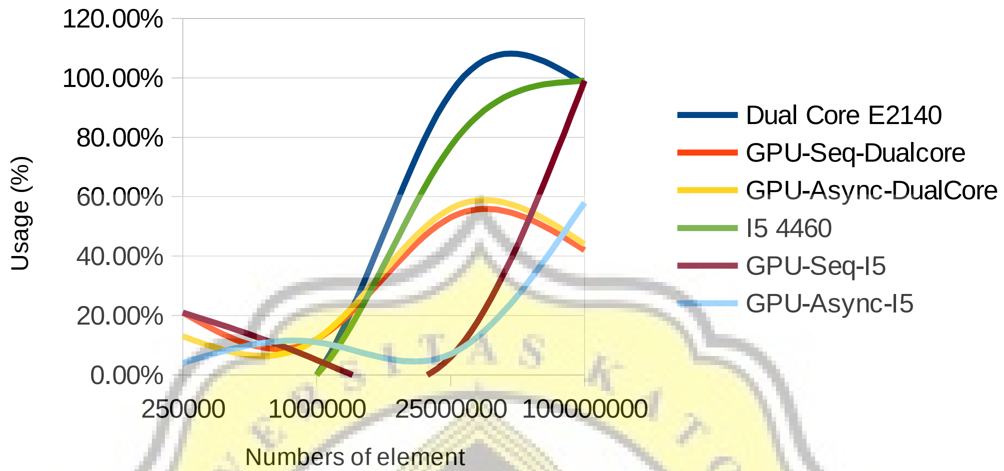

Illustration 5.6: Matrix Addition PU usage

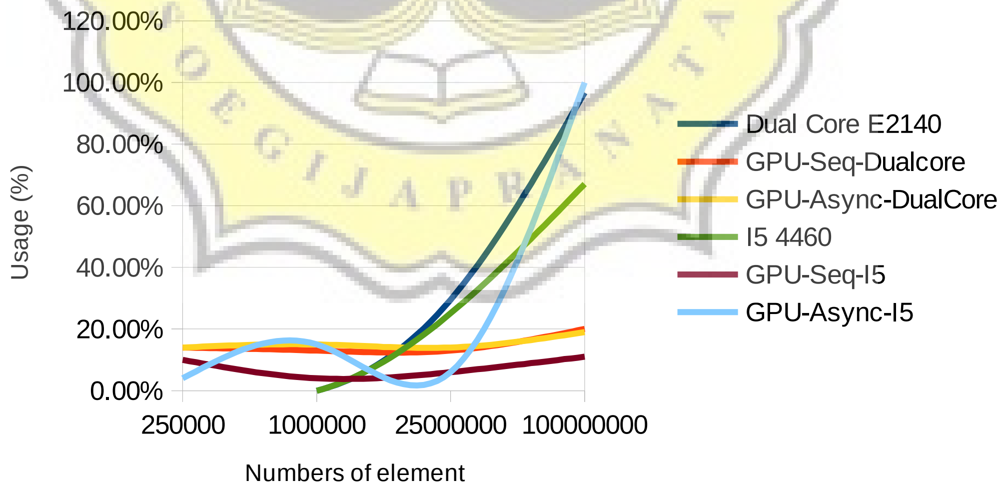

Illustration 5.7: Horizontal Flip Matrix PU usage

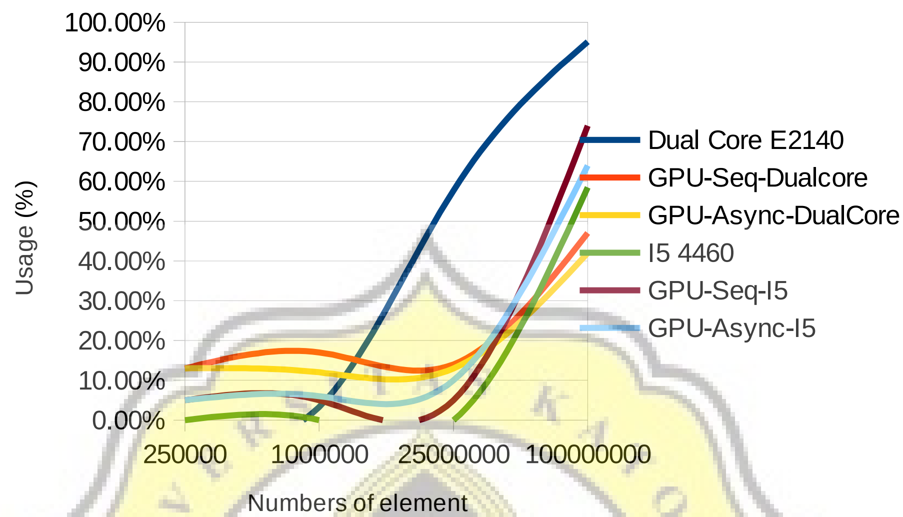

Illustration 5.8: Vertical Flip Matrix PU usage

Effect of matrix dimension by using processing unit on CPU and GPU. (in %) on all matrix operations in which the use of PU (Processing Unit) is tended to experience ups and downs are almost the same so as to make graphics look curved / exponential.

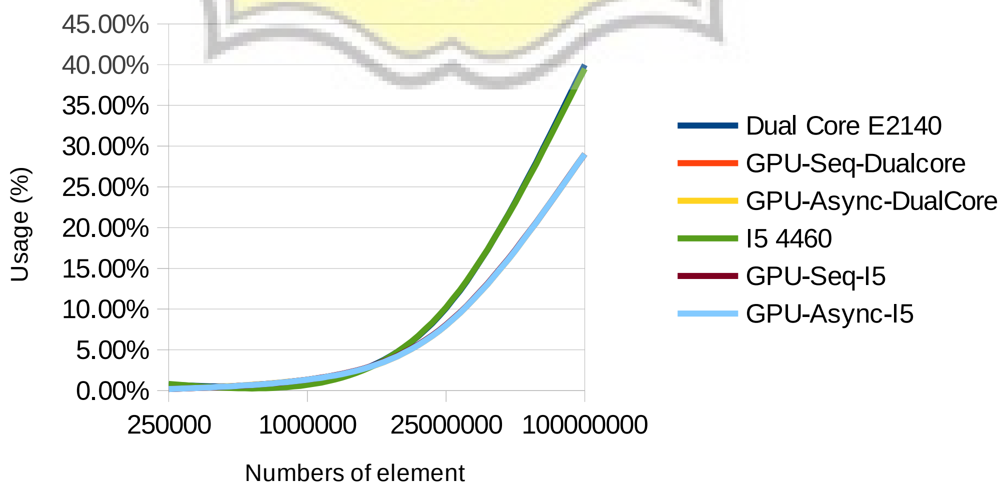

Illustration 5.9: Matrix Multiplication Memory Usage

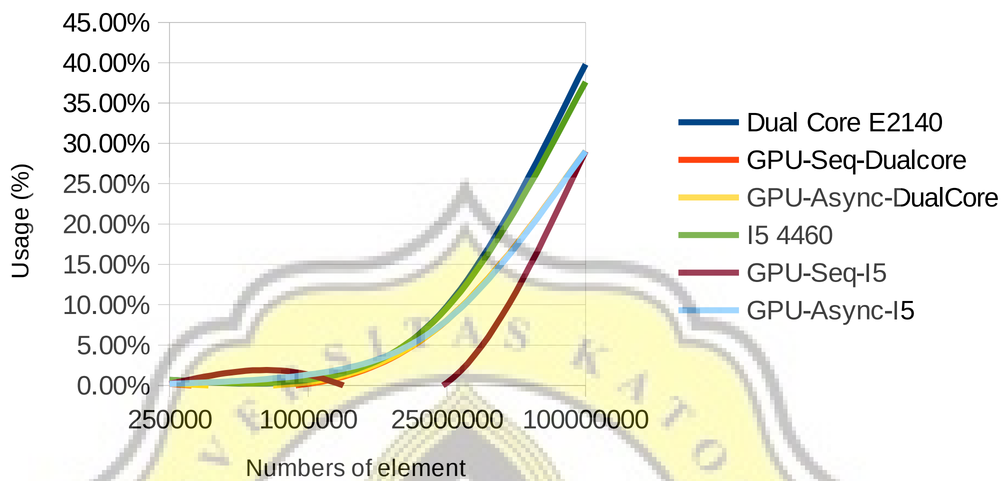

Illustration 5.10: Matrix Addition Memory Usage

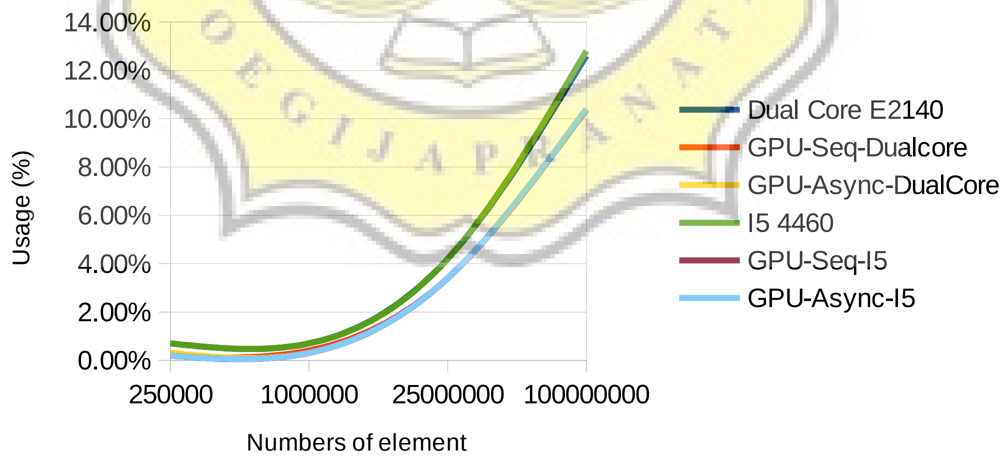

Illustration 5.11: Horizontal Flip Matrix Memory Usage

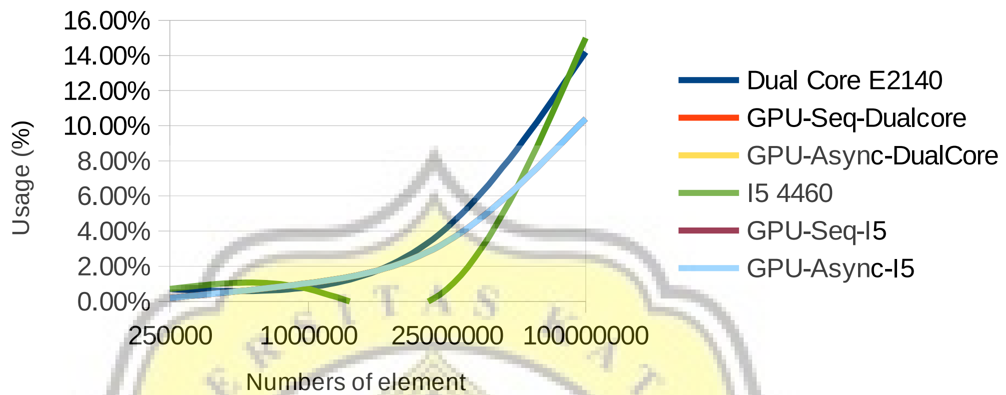

Illustration 5.12: Memory Usage Vertical Flip Matrix

Effect of matrix dimension with memory usage on CPU and GPU. (In%) of the graphics the results obtained above indicate that the process of increasing memory usage on CPU and GPU occurs in a linear fashion where the use of large data takes up more memory.

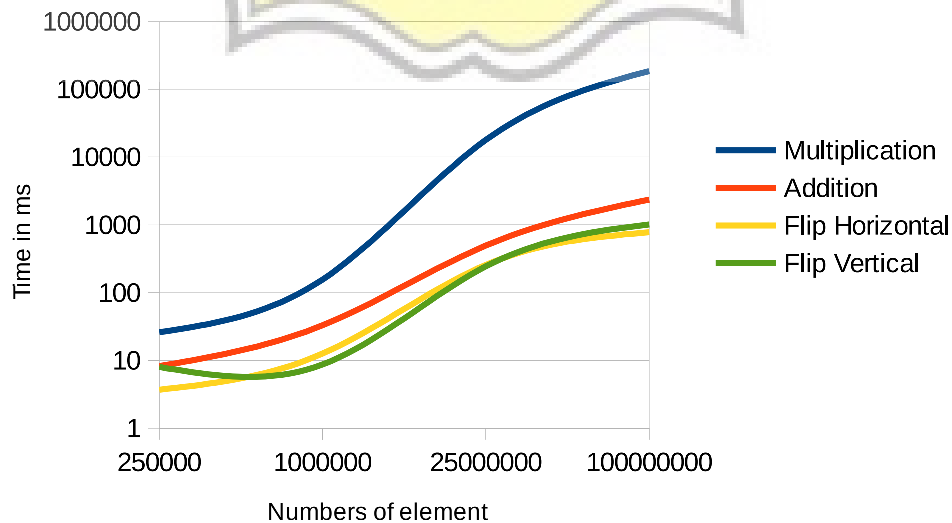

Illustration 5.13: Comparing Matrix Operation Dual Core

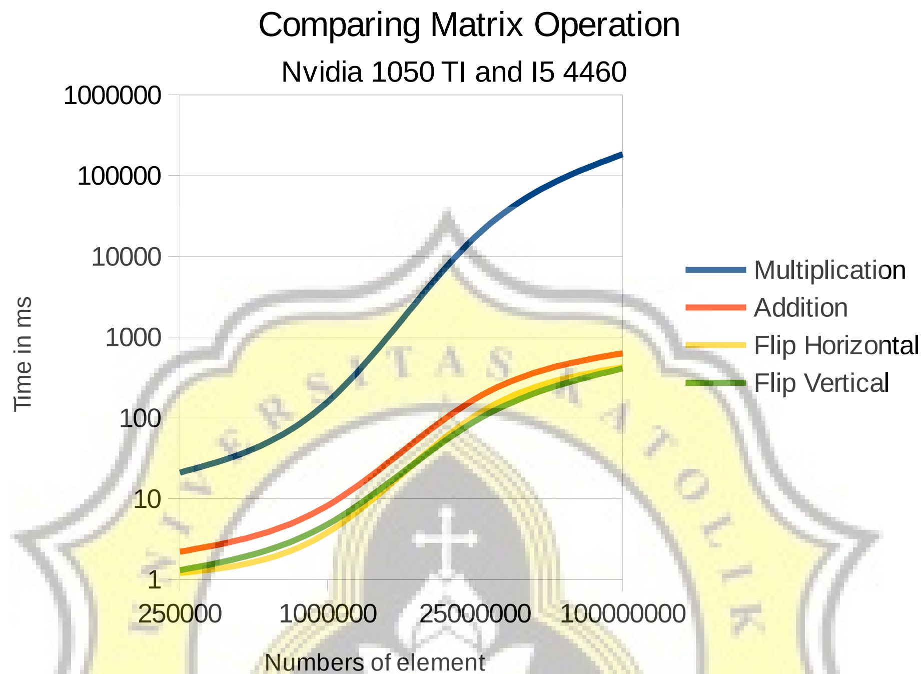

Illustration 5.14: Comparing Matrix Operation Core I5

Time effects on the number of matrix elements in performing different tasks on the GPU. From the above results the time used for Core i5 4460 is faster than Dual Core E2110.