Lab Report

Przemysław Ryś

November 2022

1 Introduction

This report is about naive matrix multiplication and heat conduction kernels analysis using CUDA C/C++ language. It is an architecture developed by NVIDIA that uses the computing power of multiple GPU (Compute Unified Device Architecture) cores to offload CPU usage and thus increase device performance. The design of the CUDA architecture assumes full scalability of programs, it means an executable program written today is to run without any changes on increasingly powerful graphics processors in the future. When programming with CUDA technology, the GPU is seen as a computational device capable of executing a large number of threads in parallel. The GPU then works as a coprocessor for the main CPU.

In other words, the program is divided into separate applications that each thread executes at the same time. The program executable by these coprocessors is called the kernel. the following examples will present the use of this vases architecture with an analysis of how this process works.

2 Experimental Part

2.1 Matrix Multiplication

Let's start with an example of matrix multiplication. We call this object a table of numbers, in the sciences it is convenient to use this representation, because it simplifies the model for us, and causes less effort in various calculations.

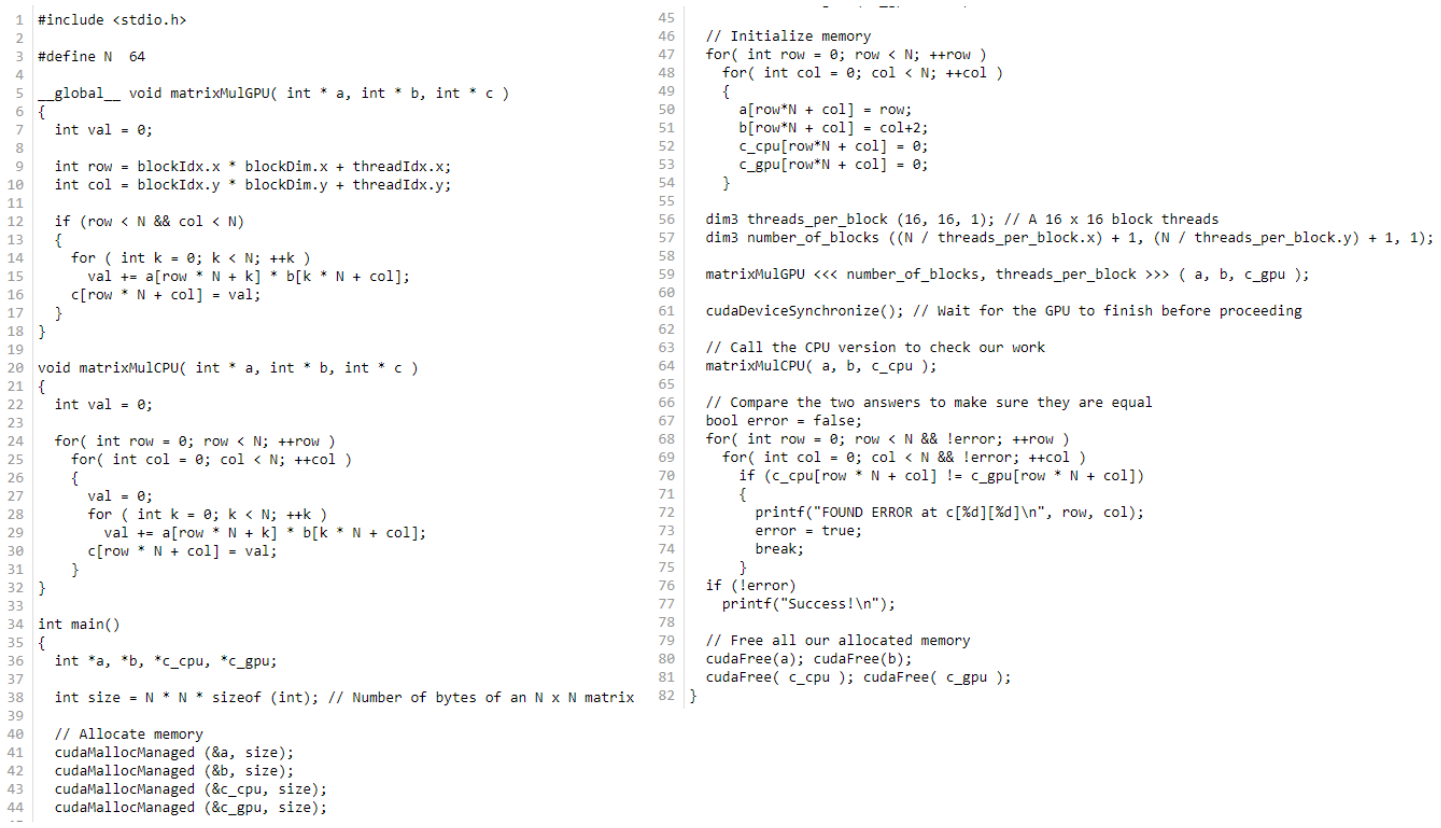
$$\underbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1k} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2k} \\ & \vdots & & \ddots & \vdots \\ a_{(n-1)1} & \cdots & \cdots & \cdots & a_{(n-1)k} \\ a_{n1} & a_{n2} & \cdots & \cdots & a_{nk} \end{bmatrix}}_{A_{nk}(t)} \cdot \underbrace{\begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots & b_{1m} \\ b_{21} & b_{22} & b_{23} & \cdots & b_{2m} \\ & \vdots & & \ddots & \vdots \\ b_{(k-1)1} & \cdots & \cdots & \cdots & b_{(k-1)m} \\ b_{k1} & b_{k2} & \cdots & \cdots & b_{km} \end{bmatrix}}_{B_{km}(t)} = \underbrace{\begin{bmatrix} c_{11} & c_{12} & c_{13} & \cdots & c_{1m} \\ c_{21} & c_{22} & c_{23} & \cdots & c_{2m} \\ & \vdots & & \ddots & \vdots \\ c_{(n-1)1} & \cdots & \cdots & \cdots & c_{(n-1)m} \\ c_{n1} & c_{n2} & \cdots & \cdots & c_{nm} \end{bmatrix}}_{C_{nm}(t)}$$

where each component is given a formula:

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}.$$

The number of multiplications of each such term is k, and these k terms must be added to each element of this new matrix. Taking into account the resulting size of the $n \times m$ matrix, these operations are quite a lot for one core, let alone for someone who would like to count them manually on a piece of paper... It just takes time to do this, as you can rightly see, unnecessarily, because each element is counted independently of the others. And this is where NVIDIA comes in with a brilliant approach to breaking this process down into some kind of co-processes called kernels. Our task is divided into parts performed by individual threads.

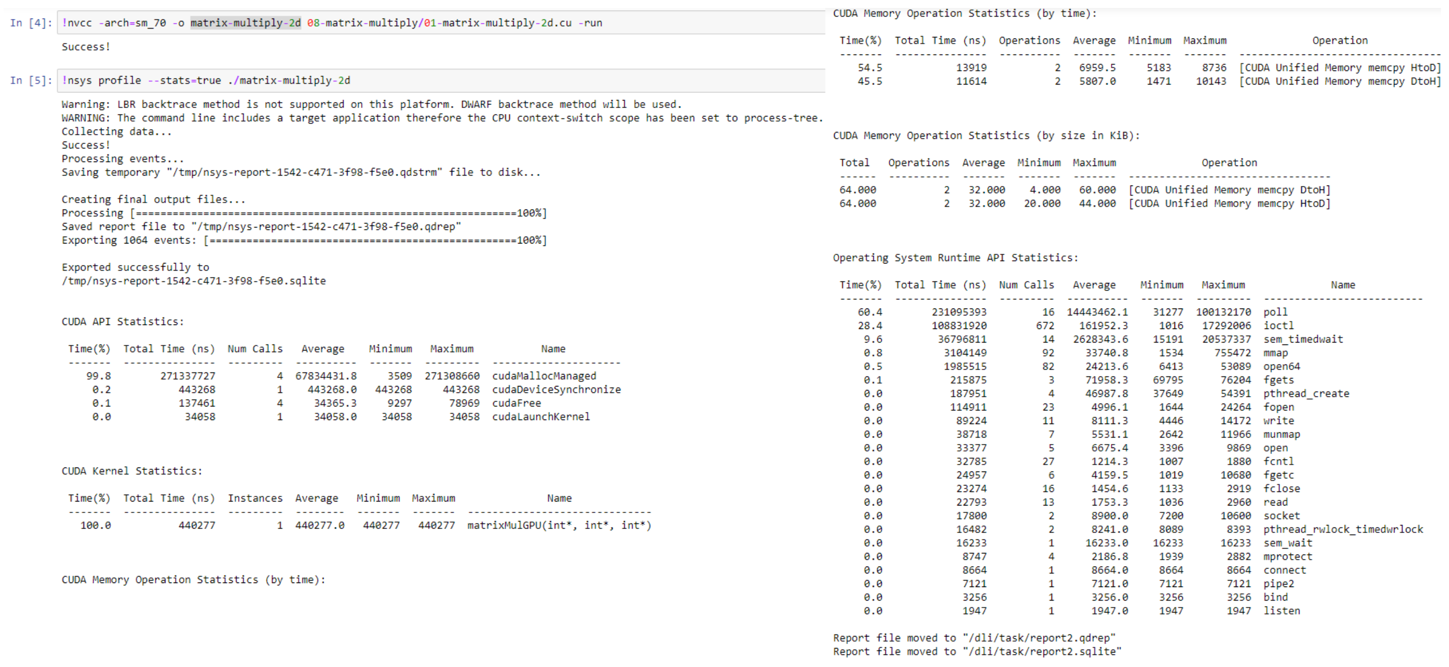
Let's look at the code of matrix multiplication in the CUDA architecture, where we will divide the multiplication of the elements of each row and column into individual threads.



```
 1  #include <stdio.h>
 2
 3  #define N  64
 4
 5  __global__ void matrixMulGPU( int * a, int * b, int * c )
 6  {
 7    int val = 0;
 8
 9    int row = blockIdx.x * blockDim.x + threadIdx.x;
10    int col = blockIdx.y * blockDim.y + threadIdx.y;
11
12    if (row < N && col < N)
13    {
14      for ( int k = 0; k < N; ++k )
15        val += a[row * N + k] * b[k * N + col];
16      c[row * N + col] = val;
17    }
18  }
19
20  void matrixMulCPU( int * a, int * b, int * c )
21  {
22    int val = 0;
23
24    for( int row = 0; row < N; ++row )
25      for( int col = 0; col < N; ++col )
26      {
27        val = 0;
28        for ( int k = 0; k < N; ++k )
29          val += a[row * N + k] * b[k * N + col];
30        c[row * N + col] = val;
31      }
32  }
33
34  int main()
35  {
36    int *a, *b, *c_cpu, *c_gpu;
37
38    int size = N * N * sizeof (int); // Number of bytes of an N x N matrix
39
40    // Allocate memory
41    cudaMallocManaged (&a, size);
42    cudaMallocManaged (&b, size);
43    cudaMallocManaged (&c_cpu, size);
44    cudaMallocManaged (&c_gpu, size);
45
46    // Initialize memory
47    for( int row = 0; row < N; ++row )
48      for( int col = 0; col < N; ++col )
49      {
50        a[row*N + col] = row;
51        b[row*N + col] = col+2;
52        c_cpu[row*N + col] = 0;
53        c_gpu[row*N + col] = 0;
54      }
55
56    dim3 threads_per_block (16, 16, 1); // A 16 x 16 block threads
57    dim3 number_of_blocks ((N / threads_per_block.x) + 1, (N / threads_per_block.y) + 1, 1);
58
59    matrixMulGPU <<< number_of_blocks, threads_per_block >>> ( a, b, c_gpu );
60
61    cudaDeviceSynchronize(); // Wait for the GPU to finish before proceeding
62
63    // Call the CPU version to check our work
64    matrixMulCPU( a, b, c_cpu );
65
66    // Compare the two answers to make sure they are equal
67    bool error = false;
68    for( int row = 0; row < N && !error; ++row )
69      for( int col = 0; col < N && !error; ++col )
70        if (c_cpu[row * N + col] != c_gpu[row * N + col])
71        {
72          printf("FOUND ERROR at c[%d][%d]\n", row, col);
73          error = true;
74          break;
75        }
76    if (!error)
77      printf("Success!\n");
78
79    // Free all our allocated memory
80    cudaFree(a); cudaFree(b);
81    cudaFree( c_cpu ); cudaFree( c_gpu );
82  }
```

Fig. 1: CUDA C/C++ code for matrix multiplication.

The above code is written in the syntax typical for CUDA, it means it contains some kind of idioms characteristic for this language, it means expressions not found in C/C++ itself, although we support a version based on these languages. It is characteristic here, among other things, to define the functions themselves as global (we declare them with the __global__ prefix), so that they are visible to the GPU. Undoubtedly, the biggest advantage, not complete, but partial replacement of loops in the code by spreading the work to other machine units (GPU threads), thus freeing you from the information overload, and thus their processing by the CPU. however, we must remember about the appropriate Malloc of memory for the variables processed by the GPU and the cudaDeviceSynchronize() function, which will ensure that the GPU will finish its work before giving us the results of the operation. We also need to determine how much computing power we engage in the execution of the process, saving the number of threads, the number of functions performed on the program at the same time and blocks (We will determine them by dividing the data by the number of our functions, read threads, so that the task is distributed as much as possible even way). It is obvious that by increasing the computing power we reduce the time needed to complete the task, but as we will see, we cannot reduce the effective time of the process to zero. As we can see in line 9, we omit the instance related to loops iterating over the matrix at the expense of the definition of another quantity that will help us navigate through the objects of the matrix. Because it so happens that an object such as a matrix does not exist in the computer's memory as we see it, the first thread of the second row is not at all close to the first thread, but in the first row, it is away from it exactly by the number of threads in the row in which it is located. In this approach, iterating over the matrix becomes simple, because if we are interested in a specific thread, it is enough to check in which block it is located, and then add the differences between the general position and the number of the block times its size. Finally, of course, we also need to clean up that busy memory to free ourselves from memory leaks.
I base the analysis of programs on changing parameters, such as the number of executable threads, the program execution time changes. I'm seeing this as a result of using "!nsys profile –stats=true ./[filename]". Each time the program is called, it happens at a different time, so if I notice a gross error involving a drastic increase in execution time compared to a program using fewer threads, I repeat the call. In this way, I receive a series of measurements for the case of a problem with matrices and heat conduction , which I adjust to the process duration curve depending on the number of threads.



```
In [4]: !nvcc -arch=sm_70 -o matrix-multiply-2d 08-matrix-multiply/01-matrix-multiply-2d.cu -run

        Success!

In [5]: !nsys profile --stats=true ./matrix-multiply-2d

        Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
        WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-tree.
        Collecting data...
        Success!
        Processing events...
        Saving temporary "/tmp/nsys-report-1542-c471-3f98-f5e0.qdstrm" file to disk...

        Creating final output files...
        Processing [==============================================================100%]
        Saved report file to "/tmp/nsys-report-1542-c471-3f98-f5e0.qdrep"
        Exporting 1064 events: [==================================================100%]

        Exported successfully to
        /tmp/nsys-report-1542-c471-3f98-f5e0.sqlite


        CUDA API Statistics:

         Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
         -------  ---------------  ---------  ----------  -------  ---------  ---------------------
            99.8        271337727          4  67834431.8     3509  271308660  cudaMallocManaged
             0.2           443268          1    443268.0   443268     443268  cudaDeviceSynchronize
             0.1           137461          4     34365.3     9297      78969  cudaFree
             0.0            34058          1     34058.0    34058      34058  cudaLaunchKernel



        CUDA Kernel Statistics:

         Time(%)  Total Time (ns)  Instances  Average   Minimum  Maximum               Name
         -------  ---------------  ---------  --------  -------  -------  ------------------------------
           100.0           440277          1  440277.0   440277   440277  matrixMulGPU(int*, int*, int*)



        CUDA Memory Operation Statistics (by time):

         Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum              Operation
         -------  ---------------  ----------  -------  -------  -------  ---------------------------------
            54.5            13919           2   6959.5     5183     8736  [CUDA Unified Memory memcpy HtoD]
            45.5            11614           2   5807.0     1471    10143  [CUDA Unified Memory memcpy DtoH]


        CUDA Memory Operation Statistics (by size in KiB):

         Total   Operations  Average  Minimum  Maximum              Operation
         ------  ----------  -------  -------  -------  ---------------------------------
         64.000           2   32.000    4.000   60.000  [CUDA Unified Memory memcpy DtoH]
         64.000           2   32.000   20.000   44.000  [CUDA Unified Memory memcpy HtoD]



        Operating System Runtime API Statistics:

         Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum             Name
         -------  ---------------  ---------  ----------  -------  ---------  --------------------------
            60.4        231095393         16  14443462.1    31277  100132170  poll
            28.4        108831920        672    161952.3     1016   17292006  ioctl
             9.6         36796811         14   2628343.6    15191   20537337  sem_timedwait
             0.8          3104149         92     33740.8     1534     755472  mmap
             0.5          1985515         82     24213.6     6413      53089  open64
             0.1           215875          3     71958.3    69795      76204  fgets
             0.0           187951          4     46987.8    37649      54391  pthread_create
             0.0           114911         23      4996.1     1644      24264  fopen
             0.0            89224         11      8111.3     4446      14172  write
             0.0            38718          7      5531.1     2642      11966  munmap
             0.0            33377          5      6675.4     3396       9869  open
             0.0            32785         27      1214.3     1007       1880  fcntl
             0.0            24957          6      4159.5     1019      10680  fgetc
             0.0            23274         16      1454.6     1133       2919  fclose
             0.0            22793         13      1753.3     1036       2960  read
             0.0            17800          2      8900.0     7200      10600  socket
             0.0            16482          2      8241.0     8089       8393  pthread_rwlock_timedwrlock
             0.0            16233          1     16233.0    16233      16233  sem_wait
             0.0             8747          4      2186.8     1939       2882  mprotect
             0.0             8664          1      8664.0     8664       8664  connect
             0.0             7121          1      7121.0     7121       7121  pipe2
             0.0             3256          1      3256.0     3256       3256  bind
             0.0             1947          1      1947.0     1947       1947  listen

        Report file moved to "/dli/task/report2.qdrep"
        Report file moved to "/dli/task/report2.sqlite"
```

Fig. 2: Program statistics from Figure 1.

In Figure 2, apart from the general parameters of the executed program, we can see a section called "CUDA Kernel Statistics", this is where the information on how long the process took for a single thread is located

Tab. 1: Kernel time execution table depending on the number of threads for matrix multiplication.

Threads_x	Kernel Time [ms]						Mean [ms]	Mean Deviation [ms]
1	0,57	0,53	0,50	0,54	0,53	0,42	0,51	0,03
2	0,51	0,52	0,50	0,51	0,43	0,42	0,48	0,02
3	0,56	0,52	0,53	0,50	0,51	0,42	0,51	0,02
4	0,55	0,54	0,50	0,50	0,49	0,42	0,50	0,02
5	0,51	0,55	0,50	0,48	0,51	0,45	0,50	0,02
6	0,55	0,49	0,50	0,45	0,45	0,43	0,48	0,02
7	0,48	0,50	0,49	0,46	0,46	0,46	0,48	0,01
8	0,54	0,51	0,52	0,48	0,47	0,42	0,49	0,02
9	0,43	0,49	0,50	0,49	0,45	0,44	0,47	0,02
10	0,39	0,49	0,53	0,49	0,42	0,40	0,45	0,03
11	0,50	0,50	0,51	0,46	0,46	0,43	0,47	0,02
12	0,51	0,52	0,48	0,47	0,46	0,43	0,48	0,02
13	0,49	0,48	0,45	0,45	0,43	0,42	0,45	0,02
14	0,53	0,47	0,51	0,45	0,43	0,43	0,47	0,02
15	0,52	0,48	0,48	0,46	0,46	0,48	0,48	0,01
16	0,50	0,48	0,46	0,43	0,41	0,44	0,46	0,02
Threads_y	1	2	4	8	16	32		

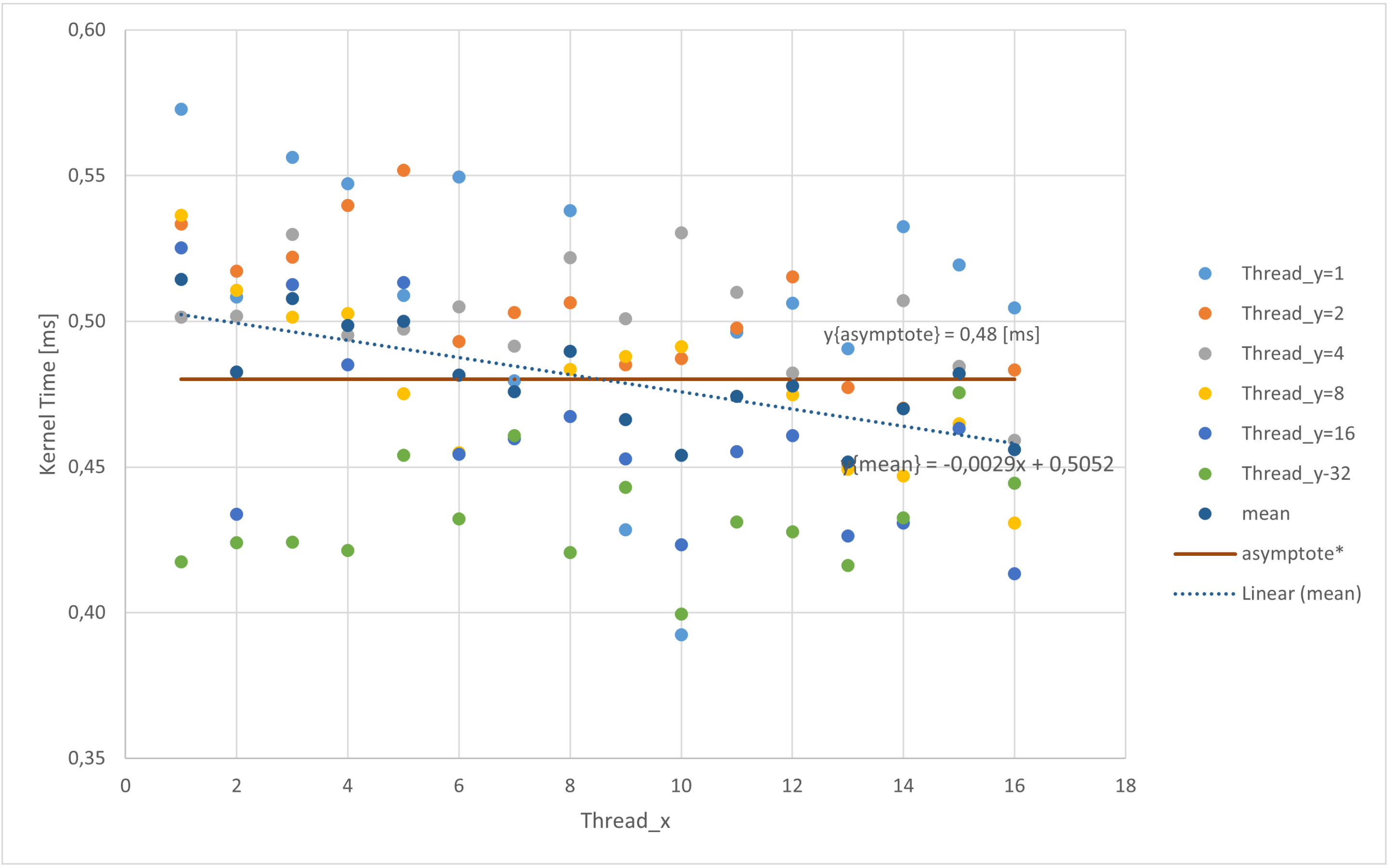


Fig. 3: Matrix plot for data from tab 1. *Actually it is not an asymptote as we know it, but a value around which the kernel execution time oscillates.

Based on the measurement data obtained with each program call, I determine the average time for a given Threads_x between different Thread_y (these are the values on the vertical line Thread_x = n; n = 1,2,...,16). fitting the line to the data, we get a decreasing function, but it changes slightly with the change in the number of threads. theoretically, we can determine its point of intersection with the x axis. We get this for Thread_x=173, inserting this value into the body of the code for any Thread_y (preferably to the one closest to the mean value, i.e. reading Thread_y=4 from the graph) we get about 0.47 [ms], so it is not a descending line, but a constant around which the data oscillates. if I take the average of the averages from the 1 table and use the uncertainty transfer law, I get the asymptote that the thread execution time tends to with a deviation $\sigma_y = 0,2$[ms]. This means that the time of the executing kernel will saturate for a certain value, regardless of the number of blocks or threads used.

2.2 Heat Conduction

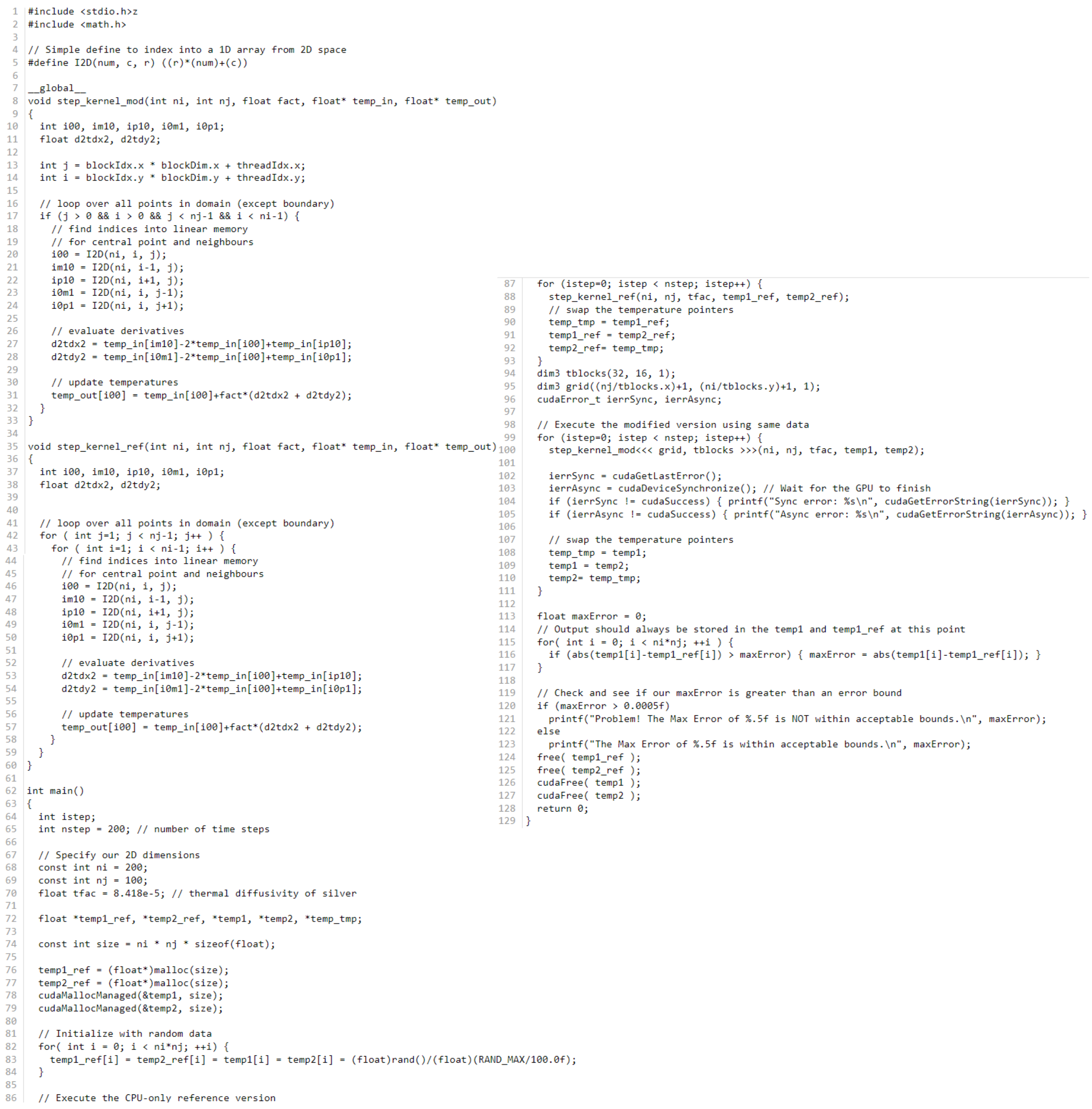
The formula to represent 2D heat conduction is given as follows:

$$\frac{\partial T}{\partial t} = \alpha(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}),$$

where T is temperature, t is time, α is the thermal diffusivity, and x and y are points in a grid. To solve this problem, one possible finite difference approximation is:

$$\frac{\Delta T}{\Delta t} = \alpha(\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}),$$

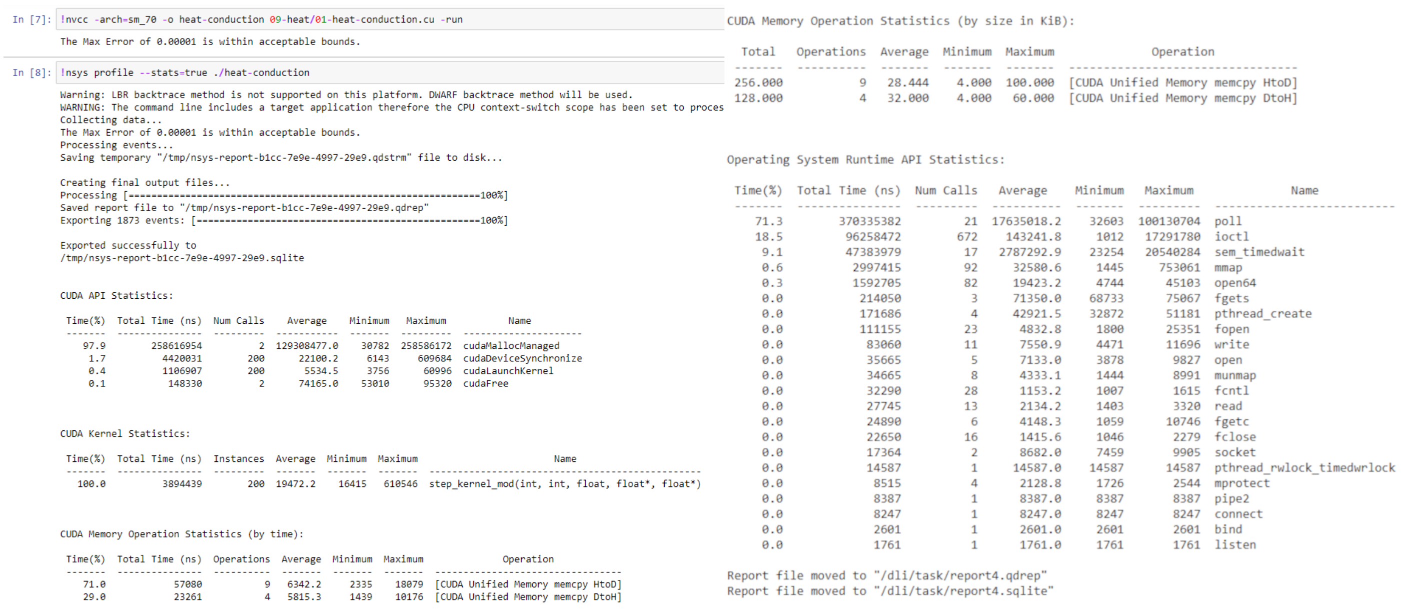
where Δ T is the temperature change over time Δ t and i,j are indices in a grid. The implementation of this problem into accelerated CUDA code is below.



```
#include <stdio.h>z
#include <math.h>

// Simple define to index into a 1D array from 2D space
#define I2D(num, c, r) ((r)*(num)+(c))

__global__
void step_kernel_mod(int ni, int nj, float fact, float* temp_in, float* temp_out)
{
  int i00, im10, ip10, i0m1, i0p1;
  float d2tdx2, d2tdy2;

  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int i = blockIdx.y * blockDim.y + threadIdx.y;

  // loop over all points in domain (except boundary)
  if (j > 0 && i > 0 && j < nj-1 && i < ni-1) {
    // find indices into linear memory
    // for central point and neighbours
    i00 = I2D(ni, i, j);
    im10 = I2D(ni, i-1, j);
    ip10 = I2D(ni, i+1, j);
    i0m1 = I2D(ni, i, j-1);
    i0p1 = I2D(ni, i, j+1);

    // evaluate derivatives
    d2tdx2 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
    d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];

    // update temperatures
    temp_out[i00] = temp_in[i00]+fact*(d2tdx2 + d2tdy2);
  }
}

void step_kernel_ref(int ni, int nj, float fact, float* temp_in, float* temp_out)
{
  int i00, im10, ip10, i0m1, i0p1;
  float d2tdx2, d2tdy2;


  // loop over all points in domain (except boundary)
  for ( int j=1; j < nj-1; j++ ) {
    for ( int i=1; i < ni-1; i++ ) {
      // find indices into linear memory
      // for central point and neighbours
      i00 = I2D(ni, i, j);
      im10 = I2D(ni, i-1, j);
      ip10 = I2D(ni, i+1, j);
      i0m1 = I2D(ni, i, j-1);
      i0p1 = I2D(ni, i, j+1);

      // evaluate derivatives
      d2tdx2 = temp_in[im10]-2*temp_in[i00]+temp_in[ip10];
      d2tdy2 = temp_in[i0m1]-2*temp_in[i00]+temp_in[i0p1];

      // update temperatures
      temp_out[i00] = temp_in[i00]+fact*(d2tdx2 + d2tdy2);
    }
  }
}

int main()
{
  int istep;
  int nstep = 200; // number of time steps

  // Specify our 2D dimensions
  const int ni = 200;
  const int nj = 100;
  float tfac = 8.418e-5; // thermal diffusivity of silver

  float *temp1_ref, *temp2_ref, *temp1, *temp2, *temp_tmp;

  const int size = ni * nj * sizeof(float);

  temp1_ref = (float*)malloc(size);
  temp2_ref = (float*)malloc(size);
  cudaMallocManaged(&temp1, size);
  cudaMallocManaged(&temp2, size);

  // Initialize with random data
  for( int i = 0; i < ni*nj; ++i) {
    temp1_ref[i] = temp2_ref[i] = temp1[i] = temp2[i] = (float)rand()/(float)(RAND_MAX/100.0f);
  }

  // Execute the CPU-only reference version
  for (istep=0; istep < nstep; istep++) {
    step_kernel_ref(ni, nj, tfac, temp1_ref, temp2_ref);
    // swap the temperature pointers
    temp_tmp = temp1_ref;
    temp1_ref = temp2_ref;
    temp2_ref= temp_tmp;
  }
  dim3 tblocks(32, 16, 1);
  dim3 grid((nj/tblocks.x)+1, (ni/tblocks.y)+1, 1);
  cudaError_t ierrSync, ierrAsync;

  // Execute the modified version using same data
  for (istep=0; istep < nstep; istep++) {
    step_kernel_mod<<< grid, tblocks >>>(ni, nj, tfac, temp1, temp2);

    ierrSync = cudaGetLastError();
    ierrAsync = cudaDeviceSynchronize(); // Wait for the GPU to finish
    if (ierrSync != cudaSuccess) { printf("Sync error: %s\n", cudaGetErrorString(ierrSync)); }
    if (ierrAsync != cudaSuccess) { printf("Async error: %s\n", cudaGetErrorString(ierrAsync)); }

    // swap the temperature pointers
    temp_tmp = temp1;
    temp1 = temp2;
    temp2= temp_tmp;
  }

  float maxError = 0;
  // Output should always be stored in the temp1 and temp1_ref at this point
  for( int i = 0; i < ni*nj; ++i ) {
    if (abs(temp1[i]-temp1_ref[i]) > maxError) { maxError = abs(temp1[i]-temp1_ref[i]); }
  }

  // Check and see if our maxError is greater than an error bound
  if (maxError > 0.0005f)
    printf("Problem! The Max Error of %.5f is NOT within acceptable bounds.\n", maxError);
  else
    printf("The Max Error of %.5f is within acceptable bounds.\n", maxError);
  free( temp1_ref );
  free( temp2_ref );
  cudaFree( temp1 );
  cudaFree( temp2 );
  return 0;
}
```

Fig. 4: Code in CUDA C/C++ architecture for heat conduction .

As a result of using the nsys function, we receive program execution statistics, in particular, we will be interested in the program execution time.



```
In [7]: !nvcc -arch=sm_70 -o heat-conduction 09-heat/01-heat-conduction.cu -run

        The Max Error of 0.00001 is within acceptable bounds.

In [8]: !nsys profile --stats=true ./heat-conduction

        Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
        WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to proces
        Collecting data...
        The Max Error of 0.00001 is within acceptable bounds.
        Processing events...
        Saving temporary "/tmp/nsys-report-b1cc-7e9e-4997-29e9.qdstrm" file to disk...

        Creating final output files...
        Processing [==============================================================100%]
        Saved report file to "/tmp/nsys-report-b1cc-7e9e-4997-29e9.qdrep"
        Exporting 1873 events: [==================================================100%]

        Exported successfully to
        /tmp/nsys-report-b1cc-7e9e-4997-29e9.sqlite


        CUDA API Statistics:

         Time(%)  Total Time (ns)  Num Calls    Average    Minimum   Maximum           Name
         -------  ---------------  ---------  -----------  -------  ---------  ---------------------
            97.9        258616954          2  129308477.0    30782  258586172  cudaMallocManaged
             1.7          4420031        200      22100.2     6143     609684  cudaDeviceSynchronize
             0.4          1106907        200       5534.5     3756      60996  cudaLaunchKernel
             0.1           148330          2      74165.0    53010      95320  cudaFree



        CUDA Kernel Statistics:

         Time(%)  Total Time (ns)  Instances  Average  Minimum  Maximum                      Name
         -------  ---------------  ---------  -------  -------  -------  ------------------------------------------------
           100.0          3894439        200  19472.2    16415   610546  step_kernel_mod(int, int, float, float*, float*)



        CUDA Memory Operation Statistics (by time):

         Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum              Operation
         -------  ---------------  ----------  -------  -------  -------  ---------------------------------
            71.0            57080           9   6342.2     2335    18079  [CUDA Unified Memory memcpy HtoD]
            29.0            23261           4   5815.3     1439    10176  [CUDA Unified Memory memcpy DtoH]

CUDA Memory Operation Statistics (by size in KiB):

  Total   Operations  Average  Minimum  Maximum              Operation
 -------  ----------  -------  -------  -------  ---------------------------------
 256.000           9   28.444    4.000  100.000  [CUDA Unified Memory memcpy HtoD]
 128.000           4   32.000    4.000   60.000  [CUDA Unified Memory memcpy DtoH]



Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum             Name
 -------  ---------------  ---------  ----------  -------  ---------  --------------------------
    71.3        370335382         21  17635018.2    32603  100130704  poll
    18.5         96258472        672    143241.8     1012   17291780  ioctl
     9.1         47383979         17   2787292.9    23254   20540284  sem_timedwait
     0.6          2997415         92     32580.6     1445     753061  mmap
     0.3          1592705         82     19423.2     4744      45103  open64
     0.0           214050          3     71350.0    68733      75067  fgets
     0.0           171686          4     42921.5    32872      51181  pthread_create
     0.0           111155         23      4832.8     1800      25351  fopen
     0.0            83060         11      7550.9     4471      11696  write
     0.0            35665          5      7133.0     3878       9827  open
     0.0            34665          8      4333.1     1444       8991  munmap
     0.0            32290         28      1153.2     1007       1615  fcntl
     0.0            27745         13      2134.2     1403       3320  read
     0.0            24890          6      4148.3     1059      10746  fgetc
     0.0            22650         16      1415.6     1046       2279  fclose
     0.0            17364          2      8682.0     7459       9905  socket
     0.0            14587          1     14587.0    14587      14587  pthread_rwlock_timedwrlock
     0.0             8515          4      2128.8     1726       2544  mprotect
     0.0             8387          1      8387.0     8387       8387  pipe2
     0.0             8247          1      8247.0     8247       8247  connect
     0.0             2601          1      2601.0     2601       2601  bind
     0.0             1761          1      1761.0     1761       1761  listen

Report file moved to "/dli/task/report4.qdrep"
Report file moved to "/dli/task/report4.sqlite"
```

Fig. 5: Program statistics from Figure 4.

Tab. 2: Kernel time execution table depending on the number of threads for heat conduction.

Threads_x	Kernel Time [ms]							Mean [ms]	Mean Deviation [ms]
1	13,89	7,97	4,76	3,22	2,52	1,75	1,65	5,11	1,69
2	7,83	4,55	3,02	2,19	1,86	1,71	1,84	3,28	0,85
3	5,73	3,65	2,67	2,00	1,71	1,64	1,84	2,75	0,57
4	4,66	2,98	2,63	1,97	1,61	1,63	1,58	2,44	0,43
5	3,98	3,12	2,40	1,83	1,82	1,73	1,69	2,37	0,34
6	3,53	2,77	2,36	2,08	2,01	1,67	1,80	2,32	0,25
7	3,20	2,70	2,10	1,71	2,05	1,84	1,89	2,21	0,21
8	2,95	2,59	1,90	1,86	2,12	1,84	1,82	2,15	0,17
9	3,10	2,24	2,20	1,92	2,10	1,90	1,88	2,19	0,17
10	2,65	2,36	2,33	1,96	2,18	1,89	2,21	2,22	0,1
11	3,04	2,18	2,37	2,19	2,19	1,92	2,06	2,28	0,14
12	2,80	2,25	2,44	2,40	2,21	2,02	2,18	2,33	0,1
13	2,97	2,11	2,36	2,05	2,26	2,16	2,29	2,31	0,12
14	3,02	2,43	2,38	2,15	2,04	2,18	2,42	2,37	0,13
15	2,99	2,23	2,07	2,13	2,18	2,19	2,54	2,33	0,13
16	2,82	2,36	2,46	2,51	2,29	2,27	2,34	2,44	0,08
Threads_y	1	2	4	8	16	32	64		

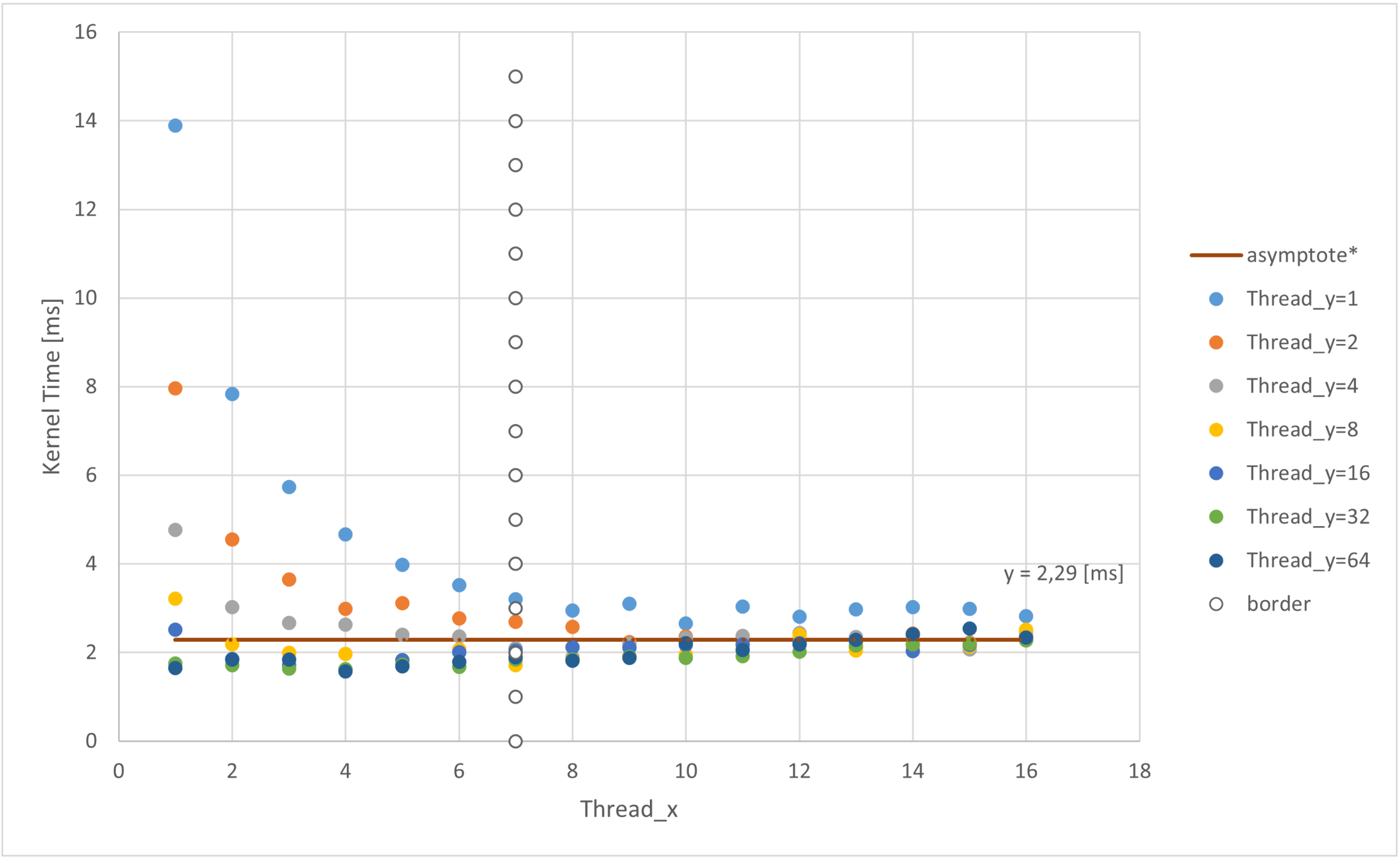


Fig. 6: Heat conduction plot for data from tab 2. *Actually it is not an asymptote as we know it, but a value around which the kernel execution time oscillates, in this case, for the average around which the data oscillates, I took the values from Thread number 7 because below 7 they diverge.

The case of heat conduction is so much simpler that you can see at first glance how the values of thread execution times for this process drastically decrease to a certain constant value. The graph of the dependence of these times on the number of threads becomes constant already for Thread_x = 7, so to determine this value, I will take the average values between the points, similarly as in the case of matrix multiplication, with the difference that I will omit the points on the Thread_x axis from 1 to 6. So, taking the average of the average distances between thread execution times for a given Thread_x, it obtains a value of 2.29 [ms], with an deviation of 0.2 [ms], also obtained by the uncertainty transfer law.

3 Summary

In this report, I presented the process implemented by the CUDA architecture on the examples of matrix multiplication with a very large dimension and heat conduction , approximating the given partial differential equation by very small temperature equally small spatial increments. in which this process will be resolved. As a result of a more detailed analysis of the graph of the process duration versus the number of threads, they come to the conclusion that the execution time reaches a certain constant value, which it does not fully achieve due to errors, but the measurement points remain close to the curve despite the increasing number of threads.
To sum up, although we will not be able to reduce the execution time of the program, you will underestimate any value of time, it is a fair price for the incredibly fast access to results compared to an analogous program, but operating in the architecture of only one process.