LABORATORY 3

Parallel Jacobi Method with CUDA C & OpenMP

Juan José Betancourt
Nicolás Delgado
Camila Paladines

Parallel Programming

Professor: Roger Alfonso Gómez Nieto

June 4, 2021

Contents

0 Methods and Materials

In numerical linear algebra, the *Jacobi Method* is an iterative algorithm for determining the solutions of a strictly diagonally dominant system of linear equations. Each diagonal element is solved for, and an approximate value is plugged in. The process is then iterated until it converges. This algorithm is a stripped-down version of the *Jacobi transformation method of a matrix diagonalization.*

Let $A\mathbf{x} = \mathbf{b}$ be a square system of n linear equations, where:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

The A can be decomposed into a diagonal component D, a lower triangular part L and an upper triangular part U:

$$A = D + L + U \ \ \textbf{where} \ \ D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

$$\textbf{and} \ L + U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}$$

The solution is then obtained iteratively via $\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - (L + U)\mathbf{x}^{(k)})$, where $\mathbf{x}^{(k)}$ is the $k-th$ approximation or iteration of $\mathbf{x}$ and $\mathbf{x}^{(k+1)}$ is the next or $(k+1)$ iteration of $\mathbf{x}$. The element-based formula is thus:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right), \quad i = 1, 2, \ldots, n.$$

The computation of $x_i^{(k+1)}$ requires each element in $\mathbf{x}^{(k)}$ except itself.

Part I
Parallel Design

1 The Code to Parallelize

1.1 Algorithm

The Jacobi Method algorithm proposed to be parallelized is as follows:

```
int run(double *A, double *b, double *x, double *xtmp)
{
  int itr;
  int row, col;
  double dot;
  double diff;
  double sqdiff;
  double *ptrtmp;

  // Loop until converged or maximum iterations reached
  itr = 0;
  do
  {
    // Perfom Jacobi iteration
    for (row = 0; row < N; row++)
    {
      dot = 0.0;
      for (col = 0; col < N; col++)
      {
        if (row != col)
          dot += A[row + col * N] * x[col];
      }
      xtmp[row] = (b[row] - dot) / A[row + row * N];
    }

    // Swap pointers
    ptrtmp = x;
    x = xtmp;
    xtmp = ptrtmp;

    // Check for convergence
    sqdiff = 0.0;
    for (row = 0; row < N; row++)
    {
      diff = xtmp[row] - x[row];
      sqdiff += diff * diff;
    }

    itr++;
  } while ((itr < MAX_ITERATIONS) && (sqrt(sqdiff) >
     CONVERGENCE_THRESHOLD));

  return itr;
}
```

Input:

- ***A***: matrix of coefficients of the system of n linear equations
- ***b***: vector of solution values of systems of linear equations.
- ***x***: vector of components.
- ***xtmp***: a temporal vector of components.

The algorithm to calculate the solutions of a strictly diagonally dominant system of linear equations is the following:

Algorithm 1: Pseudocode of Jacobi Method

```
k = 0
while convergence not reached do
    for i ← 0 to n − 1 do
        $\sigma = 0$
        for j ← 0 to n − 1 do
            if $j \neq i$ then
                $\sigma = \sigma + a_{ij} x_j^{(k)}$
            end
        end
        $x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sigma)$
    end
    k = k + 1
end
```

The standard convergence condition is when the spectral radius of the iteration matrix is less than a CONVERGENCE_THRESHOLD:

$$\rho(D^{-1}(L+U)) < \text{CONVERGENCE\_THRESHOLD}$$

A sufficient (but not necessary) condition for the method to converge is that the matrix ***A*** is strictly or irreducibly diagonally dominant. Strict row diagonal dominance means that for each row, the absolute value of the diagonal term is greater than the sum of absolute values of other terms:

$$\mid a_{ii} \mid \; > \sum_{j \neq i} \mid a_{ij} \mid .$$

The Jacobi method sometimes converges even if these conditions are not satisfied. Note that the Jacobi method does not converge for every symmetric positive-definite matrix.

1.1.1 Flowchart

A better way to understand an algorithm is through a flowchart. For the proposed algorithm, the flowchart is as follows:

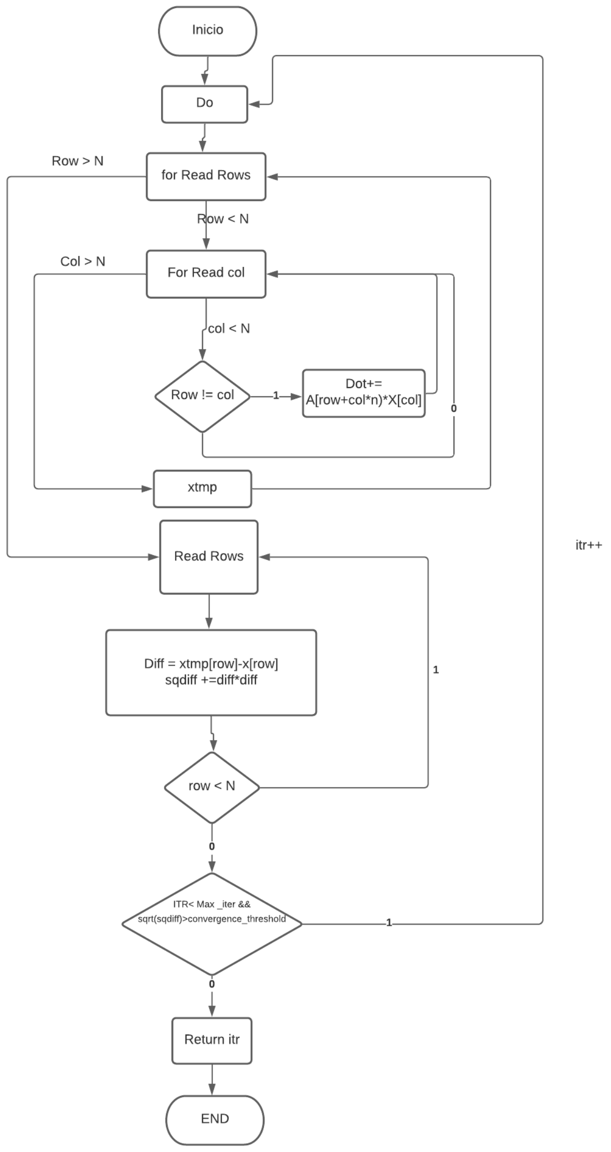


Figure 1: Flowchart of the algorithm

1.2 Complexity

Jacobi's method is an iterative but not exact method, i.e., it is an approximation (like most things in numerical computing). Then, to calculate the complexity of the algorithm in C shown above, we must consider the stop condition of the do - while statement, which, by Hoare triples, we can see that it is when the maximum number of iterations is reached or when it is satisfied that the spectral radius of the iteration matrix is less than the CONVERGENCE_THRESHOLD.

Therefore, to calculate the complexity of the method, it is necessary to take the worst case, which for us will be the maximum number of iterations. Therefore, the complexity of the method, when working with a square matrix (in $\mathbb{R}^{n\times n}$), is $\mathcal{O}(n^2)$, because we must go through the whole matrix. But this when repeated a total of times equal to the maximum number of iterations gives as a result that the complexity of the algorithm, in the worst case, is given by
$\mathcal{O}(\mathsf{MAX\_ITER} \times n^2)$.

The following graph shows the different sizes of the integer array and the corresponding execution time:

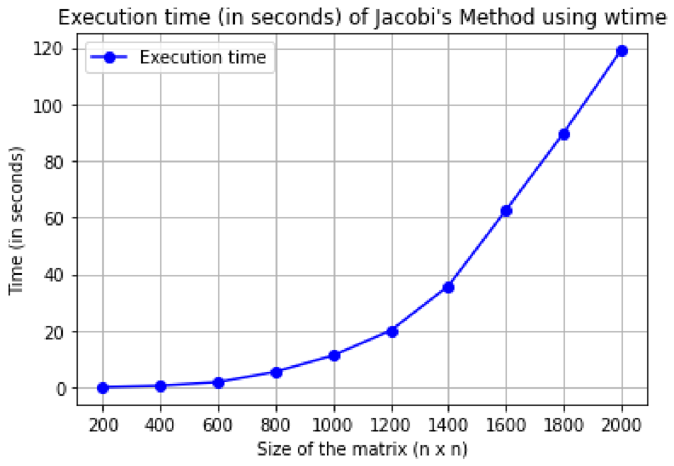


Figure 2: Execution time (in seconds) of Jacobi's Method using `wtime`

The graph shows a quadratic behavior with respect to the size of the input. Data that were extracted once the algorithm was run 5 times per test case, with different matrix sizes. The data collected can best be seen in the following table (with MAX_ITERATIONS = 20000 and CONVERGENCE_THRESHOLD = 0.0001).

Size of matrix	**Execution time (s)**
200×200	0.065090
400×400	0.566833
600×600	1.832976
800×800	5.483748
1000×1000	11.284553
1200×1200	20.040845
1400×1400	35.737575
1600×1600	62.506036
1800×1800	89.733197
2000×2000	119.020606

Table 1: Execution time (average in seconds) of Jacobi's Method using `wtime`

1.3 Time Analysis

The code was executed in a personal computer. The experimental environment where the tests were taken are as follows:

Configuration items	**Item value**
System version	Ubuntu 18.04.5
Compiler	NVCC 9.1.85
Server model	Unknown
CPU	Intel Core i5-7200U 2.50GHz x4
GPU	NVIDIA GTX 920mx
HDD capacity	64 GB (Ubuntu)
RAM size	8GB
OMP version	OpenMP 4.5
CUDA Version	CUDA 11.3.1

Table 2: General system information

Configuration (CPU)	**Value**
Model Name	Intel(R) Core(TM) i5-7200U @ 2.50GHz
Architecture	x86_64
CPU(s)	4
Thread(s) per Core	2
Core(s) per Socket	2
Socket(s)	1
CPU Max MHz	3100,0000
CPU Min MHz	400,0000
BogoMIPS	5399.81
Virtualization	VT-x
L1d Cache	32K
L1i Cache	32K
L2 Cache	256K
L3 Cache	3072K

Table 3: Detailed CPU information

Configuration (RAM)	**Value**
Total Width	64 bits
Data Width	64 bits
Size	8GB
Type	DDR4
Type Detail	Sync. Unbuffered
Speed	2133 MT/s
Manufacturer	Samsung
Configured Clock Speed	2133 MT/s

Table 4: Detailed RAM information

Configuration (GPU)	**Value**
Model Name	GeForce 920MX
Manufacturer	NVIDIA
Capacity	2 GB
NVIDIA CUDA Cores	256
Total Dedicated Memory	2004MB
Memory Interface.	64-bit
Maximum PCIe Link Width	x4
Maximum PCIe Link Speed	8.0 GT/s
TDP	16W
Interconnect GEN3	Express x4 PCIe
Bandwidth	14.40GB/s
Memory Type	DDR3
Architecture	$Maxwell$

Table 5: Detailed GPU information

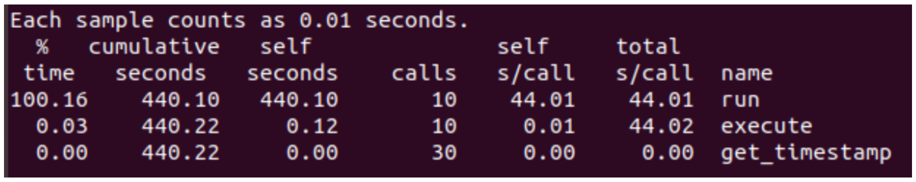
When the worst-case execution time (matrix in $\mathbb{R}^{n\times n}$, where $n = 10^4$, STACK-SIZE = 8G) was measured with the `time` command, the following results were obtained (with MAX_ITERATIONS = 100 and CONVERGENCE_THRESHOLD = 0.001, it takes more than 1.5hr with MAX_ITERATIONS $\geq$ 1000 per case!):

Type	**Average (sec)**	**Standard deviation (sec)**
real	131.9712	1.6360
user	131.6782	1.6509
sys	0.2824	0.0247

If better performance is desired, without problems with the stack size, then obviously the size of the RAM memory must be increased. The good thing is that the arrays in C/C++ have a size of $2^{1024} - 1$, which in a laboratory is difficult to achieve.

Using the `wtime` tool to measure the execution time of the `run` function (Jacobi's method solver) on the worst-case, we obtained that the average was 129.865718s, with a standard deviation of 1.5410s.

Using the `gprof` tool, the following results were obtained:



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.16    440.10   440.10       10    44.01    44.01  run
  0.03    440.22     0.12       10     0.01    44.02  execute
  0.00    440.22     0.00       30     0.00     0.00  get_timestamp
```

2 How to Parallelize the Algorithm

2.1 Proposal for Parallelization

The portion of code proposed to be parallelized is between lines 18 and 21 of the C code shown in the first section. This section contains a for loop that runs through the columns of each row in the matrix. It is not proposed to parallelize the two cycles, the outer and the inner one (the one that goes through the rows and the columns, respectively). Therefore, only the inner cycle will be parallelized using OpenMP (`#pragma omp parallel for`). In addition, the accumulated sum will not be carried in `dot`, but each of the elements of the sum will be stored in an array, which will then be passed to the GPU to calculate the accumulated sum with a single instruction (SIMD).

The way in which this cycle will be parallelized is the following: the interval of $0, 1, \ldots, n$ (the total number of columns) is separated in a number x of threads, which will execute one by one the elements they have been assigned within the interval, without repeating any element of the interval by any thread, that is to say, each element of the interval (corresponding to the columns) will be processed only once.

In other words, it is proposed to divide the n iterations of the loop into ranges so that each thread takes care of a range. In each of the iterations it will be necessary to store in an array what would be added to `dot`, and then pass this array to the device (GPU) for processing.

2.2 Data Independence

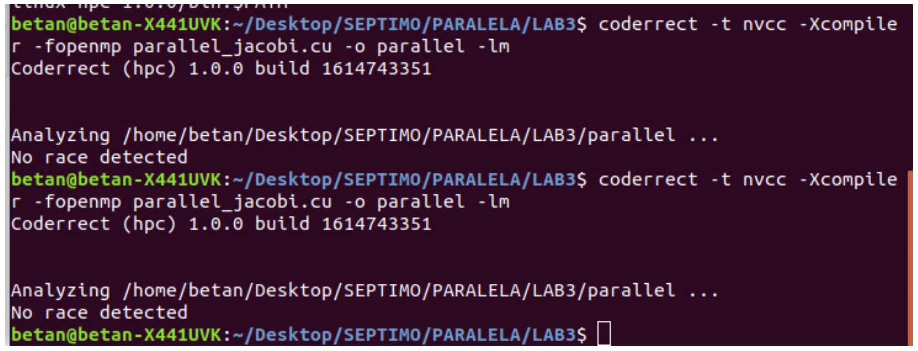
In the internal cycle proposed to be parallelized there is no data dependency, so parallelizing it is beneficial (or so we expect to see in the results of the final report). In addition, the proposed algorithm unifies the summations into one in such a way that for each case the corresponding element is added. Also each sum is independent of the others.

Part II
Parallel Algorithm

3 Verification and Analysis

3.1 Data Races

After performing the data race analysis with the Coderrect tool, the following results were obtained:



```
betan@betan-X441UVK:~/Desktop/SEPTIMO/PARALELA/LAB3$ coderrect -t nvcc -Xcompile
r -fopenmp parallel_jacobi.cu -o parallel -lm
Coderrect (hpc) 1.0.0 build 1614743351


Analyzing /home/betan/Desktop/SEPTIMO/PARALELA/LAB3/parallel ...
No race detected
betan@betan-X441UVK:~/Desktop/SEPTIMO/PARALELA/LAB3$ coderrect -t nvcc -Xcompile
r -fopenmp parallel_jacobi.cu -o parallel -lm
Coderrect (hpc) 1.0.0 build 1614743351


Analyzing /home/betan/Desktop/SEPTIMO/PARALELA/LAB3/parallel ...
No race detected
betan@betan-X441UVK:~/Desktop/SEPTIMO/PARALELA/LAB3$ 
```

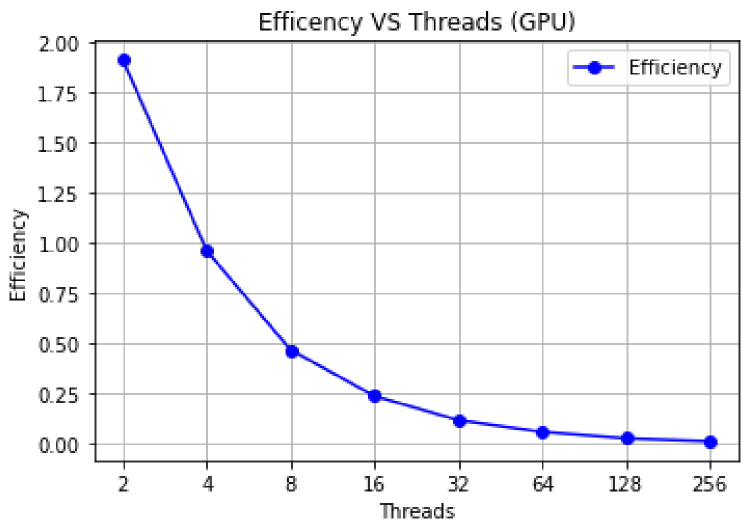
3.2 Strong and Weak Scaling

3.2.1 Strong Scaling

For the strong scaling analysis we took the problem size $n = 1000$, and calculated the efficiency for each number of cores with the given n and obtained the following results:

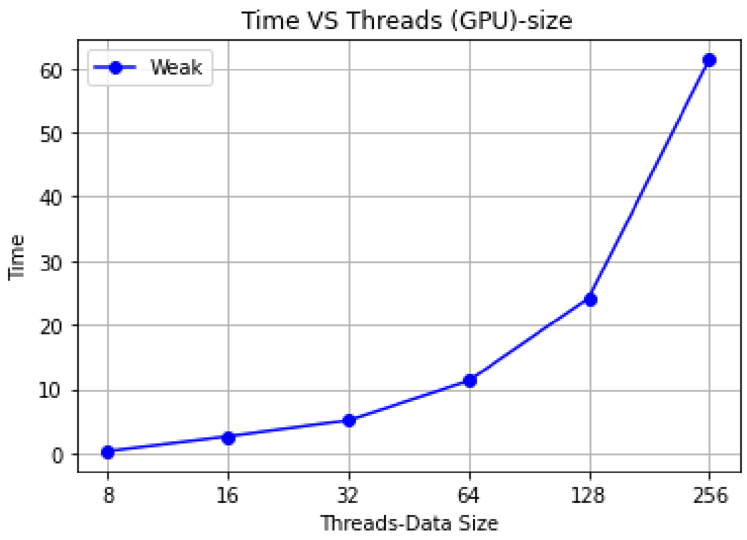
Threads	Efficency (Average)	Efficency (Standard Deviation)
2	1.90913	0.52517
4	0.96047	0.84965
8	0.46862	0.52080
16	0.23964	0.87109
32	0.11967	0.70687
64	0.06108	0.65065
128	0.02943	0.83082
256	0.01507	0.51826

Which can be seen better in the following graph:



3.2.2 Weak Scaling

For the weak scaling we took an amount of threads and N data and mesaure its time. We obtained the following results:



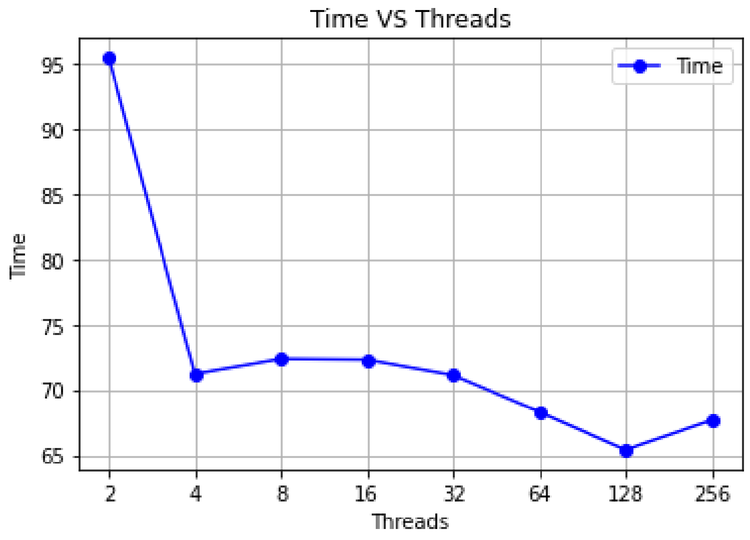
4 Results

4.1 Execution Time

To calculate the execution time, it was taken 5 times and the average and standard deviation were obtained as shown in the following table:

Threads	Time (Average)	Time (Standard Deviation)
2	95.40000	1.06947
4	71.20000	1.02049
8	72.34000	0.98356
16	72.28000	1.25201
32	71.10000	0.96498
64	68.30000	0.91949
128	65.40000	1.24624
256	67.66000	1.21881

Which can be seen better in the following graphic:

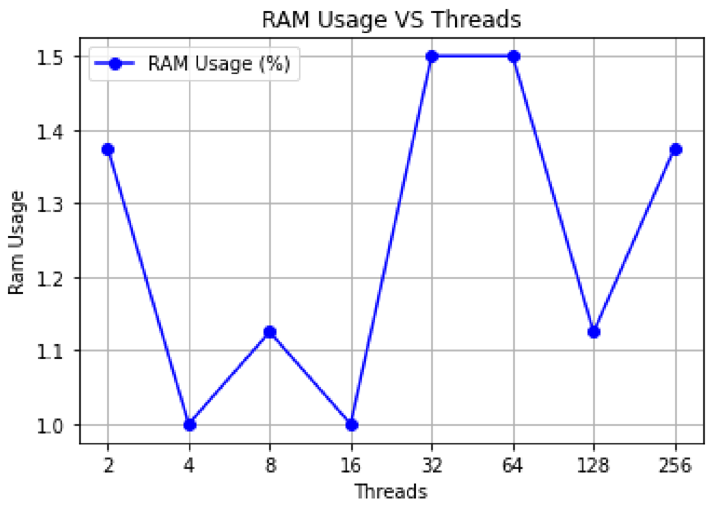


4.2 RAM Usage

The percentage of RAM used by the program during execution was measured for each of the thread number for N = 2000. The results can be seen in the following table:

Threads	**RAM Usage (Average)**	**RAM Usage (Average)**
2	1.37500	1.08169
4	1.00000	1.18132
8	1.12500	0.75328
16	1.00000	1.15071
32	1.50000	1.06891
64	1.50000	1.33136
128	1.12500	1.12235
256	1.37500	0.77587

Which can be seen better in the following graphic:

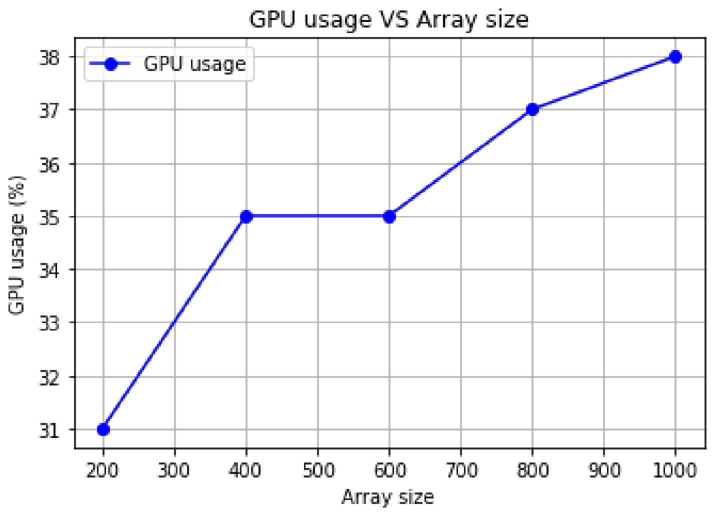


4.3 GPU Analysis

As the maximum number of threads that can be launched within the GPU that was worked on is 256, then metrics were taken of the percentage of GPU used depending on the input size. The results can be seen in the following table:

Threads	**GPU Usage (AVG)**	**GPU Usage (STD)**
200	31	0.990
400	35	1.373
600	35	1.378
800	37	1.078
1000	38	1.050

Which can be seen better in the following graphic:



5 How it Was Paralyzed

We defined a function `__global__`, which can be executed in the host or in the device, which receives as parameters two arrays, one input and one output, and the size of the input array. Then the elements of the input array ("`input`") are added up and partial sums are made, which will be stored in the thread with id `0` of each block, to be used later to calculate the total sum of the whole array. This can be best seen in the following portion of the code:

```
__global__ void total(float * input, float * output, int len) {
  __shared__ float partialSum[2 * BLOCK_SIZE];

  unsigned int t = threadIdx.x;
  unsigned int start = 2 * blockIdx.x * blockDim.x;

  partialSum[t] = (t < len) ? input[start + t] : 0;
  partialSum[blockDim.x + t] = ((blockDim.x + t) < len) ? input[start + blockDim.x + t]
     : 0;


  for(unsigned int stride = blockDim.x; stride >= 1; stride >>= 1) {
    __syncthreads();
    if(t < stride)
      partialSum[t] += partialSum[t + stride];
  }

  if(t == 0) {
    output[blockIdx.x + t] = partialSum[t];
  }
  __syncthreads();
}
```

In addition, the original function (presented in the `design` part) "`run`" was modified as follows:

```
int run(float *A, float *b, float *x, float *xtmp)
{
  int itr;
  int row, col;
  float dot;
  float diff;
  float sqdiff;
  float *ptrtmp;
  float *hostInput = (float*)malloc(N*sizeof(float));
  float * hostOutput; // The output list
  float * deviceInput;
  float * deviceOutput;
```

```
int numInputElements = N; // number of elements in the input list
int numOutputElements = numInputElements / (BLOCK_SIZE<<1);

 if (numInputElements % (BLOCK_SIZE<<1)) {
        numOutputElements++;
   }
hostOutput = (float*) malloc(numOutputElements * sizeof(float));
cudaMalloc((void **) &deviceInput, numInputElements * sizeof(float));
cudaMalloc((void **) &deviceOutput, numOutputElements * sizeof(float));

// Loop until converged or maximum iterations reached
 itr = 0;
do
{
  // Perfom Jacobi iteration
  for (row = 0; row < N; row++)
  {
    dot = 0.0;
    zeros(hostInput, N-1); //Filling hostInput with zeros ( parallel function)
    #pragma omp parallel for
    for (col = 0; col < N; col++)
    {
       if (row != col)
         hostInput[col] = A[row + col * N] * x[col];
    }
    #pragma omp barrier

    /*Begin CUDA*/
    cudaMemcpy(deviceInput, hostInput, numInputElements * sizeof(float),
   cudaMemcpyHostToDevice);
    dim3 DimGrid((numInputElements - 1)/BLOCK_SIZE + 1, 1, 1);
    dim3 DimBlock(BLOCK_SIZE, 1, 1);

    // Initialize the kernel with grid dimensions and block dimensions (#threads)
    total<<<DimGrid, DimBlock>>>(deviceInput, deviceOutput, numInputElements);

    cudaDeviceSynchronize();

    //Copy the GPU memory back to the CPU here
    cudaMemcpy(hostOutput, deviceOutput, numOutputElements * sizeof(float),
   cudaMemcpyDeviceToHost);

    //Cumulative sum in hostOutput (as a master)
    for (int ii = 1; ii < numOutputElements; ii++) {
      hostOutput[0] += hostOutput[ii];
    }

    //Put the cumulative sum in dot-variable
    dot = hostOutput[0];

    /*End CUDA*/
    xtmp[row] = (b[row] - dot) / A[row + row * N];
  }


  // Swap pointers
  ptrtmp = x;
```

```
    x = xtmp;
    xtmp = ptrtmp;

    // Check for convergence
    sqdiff = 0.0;
    for (row = 0; row < N; row++)
    {
      diff = xtmp[row] - x[row];
      sqdiff += diff * diff;
    }

    itr++;
  } while ((itr < MAX_ITERATIONS) && (sqrt(sqdiff) >
     CONVERGENCE_THRESHOLD));

  //Free the pointers in GPU
  cudaFree(deviceInput);
  cudaFree(deviceOutput);

  //Free the pointers in CPU
  free(hostInput);
  free(hostOutput);

  return itr;
}
```

So, in a nutshell, what was done was to fill a "hostInput" array with the elements that should be added to the dot variable (to understand better, check the design part first). This array is then copied to the GPU, more specifically it is copied to the "deviceInput" array, which is hosted on the GPU. Then, inside the function "total" the partial sums of each block are calculated and stored in the array "deviceOutput" (inside the GPU) to, later, copy those elements in the array "hostOutput", which will collapse the partial sums in a total sum, in the position 0 of the same array, then this total sum is stored in the dot variable to continue with the logic of the serial algorithm. At the end the pointers are released, both those of the GPU and those of the CPU.

6 Problems with Parallelization

We identified that CUDA have a more difficult curve of learning comparing with OpenMP and MPI, It is needed to change more things in compare with the both mentioned before. We found that we implemented a less efficient algorithm despite of it use a $\approx$ 30% / 40% of the GPU.

We were able to see firsthand the mess of working with as many threads as are available in a GPU NVIDIA block. In addition to exploring the different ways in which you can work with data-level parallelism (SIMD), which should be treated with caution and always keeping in mind the way you are working with the GPU, because depending on the distribution with which the kernel is initialized, it changes the logic completely. Or, sometimes, it changes the result of the algorithm (in case you are not working properly with the GPU threads).

At first it was difficult to adapt to this way of thinking, to think as if you were a thread in a sea of threads that are sectioned by cities (or blocks). So if you ask us if we found it difficult to work with CUDA arrays our answer will be that at the beginning it was. However, we do not rule out that we only know a small portion of a whole world of possibilities that CUDA can (or has) waiting to be discovered by some college students who are just entering this world.

7 Algorithm Applications

Algorithms like the Jacobi method are used in different things, some of them are:

- Calculations in population dynamics,
- In the area of chemistry to know compositions of gases in a mixture,
- Calculations in statistics and biostatistics,
- In the area of mechanical physics calculating the equilibrium of objects,
- Calculating Galileo's transformation in the principle of relativity,
- ...

8 References

- Course material, available on BlackBoard
- Jacobi method - Wikipedia
- Algorithm's author: `James Price` (@jrprice on `github.com`). 2017. Taken from this Github repo: `intro-hpc-jacobi`
- CUDA Toolkit Developer
- Stdio Library
- Stdlib Library
- Scheduled Relaxation Jacobi method: Improvements and applications
- APLICACIONES DE LOS SISTEMAS MATRICIALES EN DINÁMICA DE POBLACIONES
- Formalismo matricial para la mecánica y la termodinámica. I. Translación
- CUDA C - Programming Guide - NVIDIA Inc.
- Hybrid OpenMP + CUDA programming model - Joint Institute for Nuclear Research