Problem of the Traveling Salesman with GPU*

Juan Diego Suarez Vargas[1]

University Technology of Pereira, COLOMBIA
suarez_2929@utp.edu.co

Abstract. This document presents the work carried out in the course of HPC of the Technology University of Pereira. Is this evidenced the potential and advantage of parallel programming using CUDA technology to solve the problem of optimization as is the Sales Traveler. Performance will be analyzed with different inputs for the problem. In turn, this document aims to be an example of how to solve a complex problem through the technology that gives us the GPU. Finally this document shows a comparative study of different versions of the parallel algorithm that show the advantages in performance of the GPU.

Keywords: GPU · CUDA · Salesman Travelling · TSP.

1 Introduction

This document presents a different application of parallel programming using a GPU and CUDA [1] technology to solve the problem of the Traveling Salesman, besides being a practice for the subject HPC It is a practical project of investigative nature that can be perfectly used in the teaching field for the better understanding of this new way of parallel computing.
In the various subjects that are taught in the first years of the career as can be Computer Architecture or Data Structure are studied the different degrees of parallelism such as the parallelism at the level of ILP instruction (Instruction Level Paralelism) and DLP data level parallelism (Data Level Paralelism). The most commonly used programming tools for shared memory applications is OpenMP.
Other subjects such as Client/server and Distributed Systems taught by the Department of Systems show parallel programming through the use of message passing systems such as MPI, OpenMPI and variables shared.
This project focuses on the study of the exploitation of technology CUDA taught in the subject Computering of High Performance of the Systems Engineering to strengthen important concepts such as is the mapping of the threads in CUDA applied to the problem of the Traveling Salesman.

2 State of Art

In the year 2006 the graphics processor company NVIDIA launches the first GPU capable of rendering 3D graphics and that also includes the possibility

* Supported by University Technology of Pereira.

to execute programs written in the C language using the programming model CUDA
Since then NVIDIA offers the capacity of its graphic cards for large data centers and other institutions, both scientific and governmental, being energy efficient for the processing they are able to perform. further small versions are used to enhance the applications of both phonesmobile phones, laptops and tablets among others.
The numerous cores that have a graphic card can not only be use to draw graphics but also allow to run various programs, giving rise to the boom in our days of the so-called GPGPU computing.

3 Architecture and organization of the GPU

Next we will explain the architecture and the logical organization of a GPU with CUDA as you can see in Fig. 1

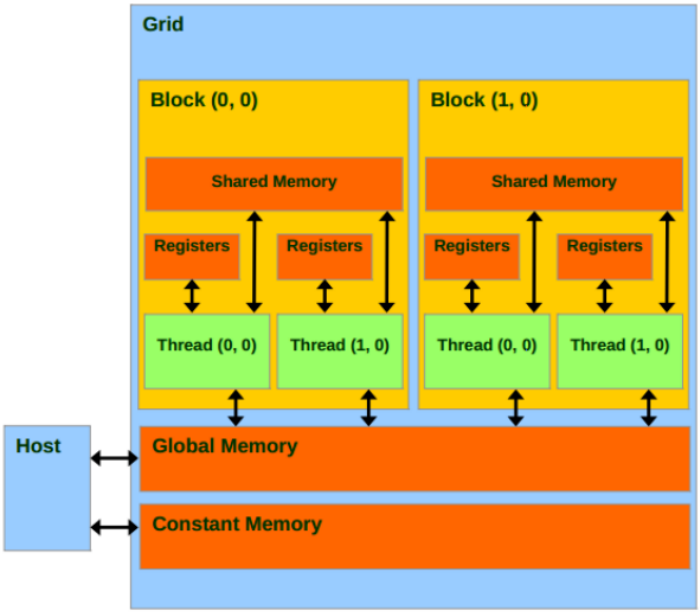

Fig. 1. Example of the CUDA memory architecture [2] .

The first logical element that we find in CUDA is the thread of execution or thread. In second place we find the concept of block and the third logical element is grid. This form of organization is related in how Resources are managed within the device.
Both the number of blocks per grid and the number of threads per block is limited

according to the physical characteristics of the device and / or the generation thereof. The so-called compute capability tells us the limits of the device.
As we can see in Figure 2, the maximum number of threads per block is of 8192 for all versions of compute capability. Both the blocks and the Threads can be organized in a one-dimensional, two-dimensional and / or three-dimensional way.The grids can be organized in a one-dimensional and/or two-dimensional way.
This plasticity of CUDA technology allows us to adapt the logical structure of the grid to the data structure of the problem that concerns us at that moment. For example If we are going to carry out a filter on 2D images, it would be appropriate to organize the blocks and the threads of two-dimensional form since the correspondence would be 1:1. To this process is called mapping or mapping.

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

Fig. 2. Table of specifications for each version of computing capacity [3] .

When we launch a set of threads in CUDA the minimum execution unit physical denominated warp executes 32 threads. That is, although our program only throw a single thread actually 32 simultaneous threads are thrown where the execution is the same for all of them because they share the same program counter.
The program that executes each thread is called a kernel. This program is executed in the same way for each thread. Before launching a kernel we need to have previously assigned the memory in the device by copying in said memory area the data that we need. That is, the data is copied from the host to the device.
Once the execution of our kernel is finished, the reverse operation is performed, Copy the results from the device to the host.

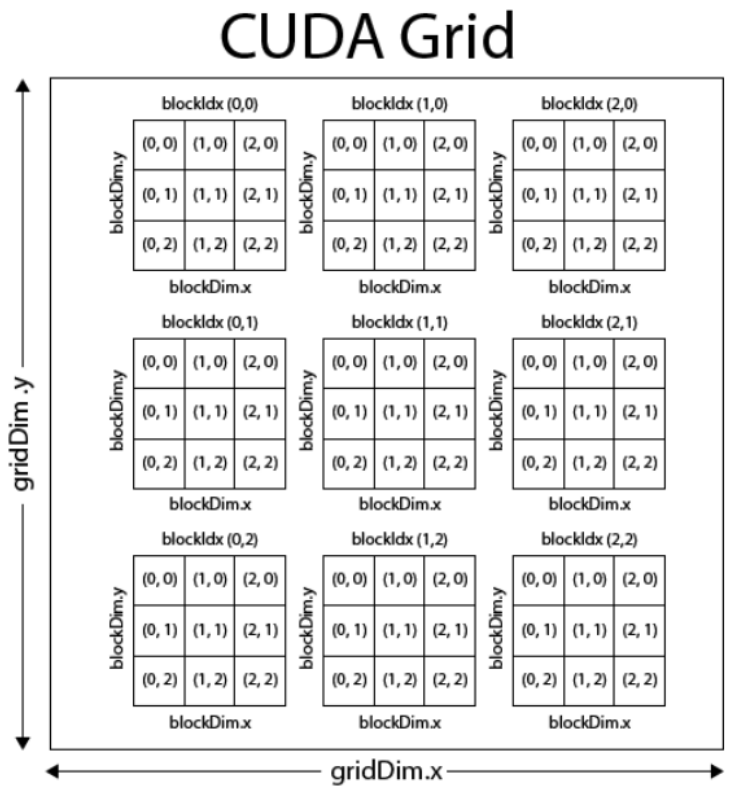

Fig. 3. Example of two-dimensional organization in CUDA [4] .

4 Project: optimization of the TSP through the GPU

This project uses GPU computing to solve the traveling salesman problem trade hereinafter referred to by its acronym (TSP) [5]. The TSP is a problem that is in the NP-Complete category since it is not they know algorithms that are capable of generating the optimal solution in time polynomial [6].
It is a combinatorial problem that is based on: several cities, finding the shortest path that travels all once only returning to the origin city. He extensive study of the TSP within computer science make it a emblematic problem that has repercussions and applications for the resolution of real life problems.
The TSP applications are multiple: planning, logistics, manufacturing of integrated circuits, etc. For example, one can think of a robotic loot industrial that makes welding points. The cities in this case each the soldering points and the resolution of the problem would give us the sequence of points which means that the robotic arm to minimize the total time of this work.

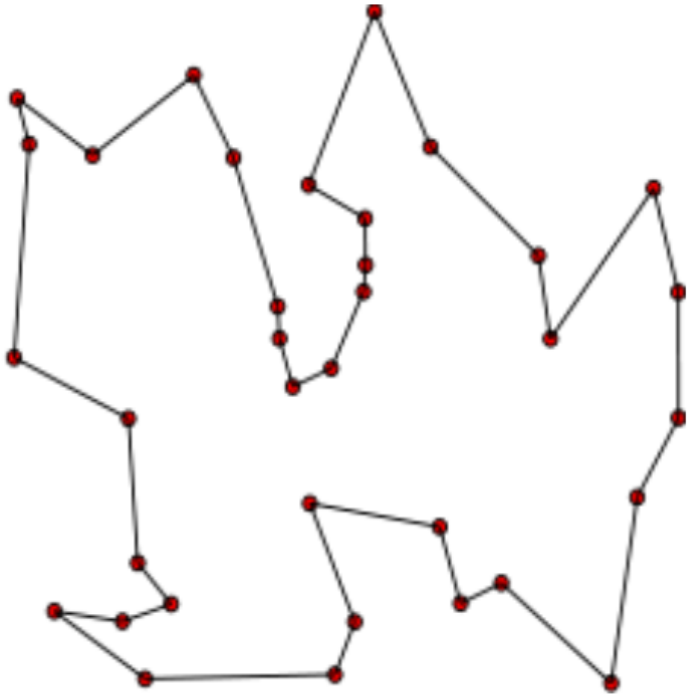

Fig. 4. Sample circuit obtained for the TSP [7] .

Given the nature of the TSP problem, an algorithm has been developed that making use of a simple heuristic, manages to generate sub-optimal solutions,

close to the real optimum, in an acceptable execution time.
The algorithm described below is as follows:

```
for () // 8192 solutions
      for () // n iterations
           for () // Calculate solution distance
           {
                // Distance matrix
                exchange items
                if (improvement) {
                     distance = new_distance
                }
                else {
                     restore elements
                }
           }
           update solution
```

The data structure of the problem is a one-dimensional vector that presents the Following way:

[e0, e1,, eN-1, c0, eN+N+2,, eN+N+N+1, cN-1]

Where e represented the element of the solution and c the cost associated with said solution.

5 Parallelization of the problem

The parallelization of the problem is achieved by having each thread apply the algorithm and generate a sub-optimal solution. Therefore as we generate 8192 solutions will be launched 8192 threads, each of them generating 1 sub-optimal solution.
This number of threads (8192) is not trivial. This number has been chosen since If we were to launch more threads depending on the size of the problem, we would also have that occupy more VRAM memory of the graphics card and reservation times, copied and access to the global memory of the results would be greater. In turn we avoid divergence control to be power of two.
Proceed then parallelizing the first loop, since if we tried a supposed parallelisation of the second loop the cost of synchronizing the result of the Threads in each iteration through shared memory would severely damage the performance.

The proposed organization is as follows, one-dimensional for the blocks and one-dimensional for the threads. We would not get any difference making another type of distribution since this distribution is perfectly adapted to the nature of the problem
Therefore, the mapping scheme for the 8192 threads is as follows:

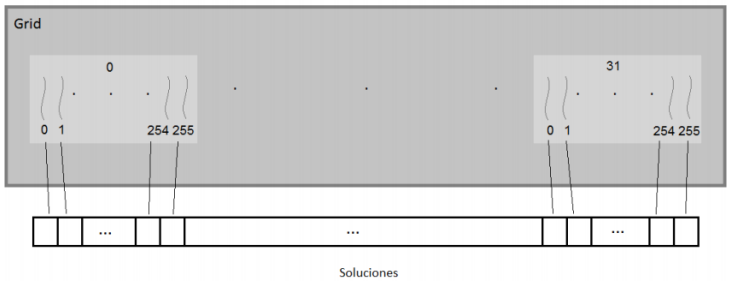

Fig. 5. Mapping of work by thread.

Threads organized in blocks will be launched, that is, 32 blocks of 256 threads each giving a total of 8192 threads.
Each thread will write its result in its corresponding part of the solution vector. This is done by calculating the offset of each strand in the following way:

```
offset = ((blockIdx.x * blockDim.x) + threadId.x)*
         (dimension + 1)
```

Where dimension is the number of cities of the problem to solve. A unit is added to this variable since each dimension solution has its cost.

6 Proposed solutions

Once decided to parallelize, three versions of the same have been implemented algorithm whose differences are:

- The first version uses shared memory to store the matrix of distances and records for him vector solution. This version of algorithm has turned out to be the fastest.
- The second version algorithm uses records for the vector solution.
- The third version uses shared memory for the vector of random numbers and records for him vector solution. This algorithm version has turned out to be the slowest. It launches n / 85 times being n the number of cities.

The use of the first, second or third version of the algorithm depends the size of the problem, since there are problems, that due to their size can not be addressed with the first version, since the distance matrix takes up too much and it is not possible to store it in the shared memory of the device.

- Number of cities less than or equal to 85: First algorithm
- Number of cities between 85 and 89: Second algorithm
- Number of cities greater than 90: Third algorithm

Since we have a size of 64KB for the shared memory we can fit 64384 8-bit elements. With 89 cities we reached that limit to store the matrix of distances

since 89x89x8 = 63368 elements. For this reason the first algorithm is the most efficient at run-time to have the matrix of distances in memory shared and also the solution vector in registers. From a size greater than 85 cities and less than 89 cities we can no longer enter the matrix of distances in shared memory so we have used the second algorithm.
When the number of cities is greater than 90 we use the third algorithm released n / 89 times where n is the number of cities. In each execution the maximum size of the solution vector stored in registers is 89. In the shared memory it is store 4000 random 32-bit numbers The distance matrix remains in global memory due to the impossibility of storing it in shared memory. This algorithm produces worse solutions since the permutations are made on parts of the solutions vector and not on the complete solutions vector.
The pseudo-code of each version is shown below:

```
for () // 8192 solutions
   for () // n iterations
     for () // Calculate solution distance
     {
       // Matrix of distances in shared memory
       // Vector solution in registers
       exchange items
       if (improvement)
       {
         distance = new_distance
       }
       else
       {
         restore elements
       }
     }
     update solution
```

Table 1. First version of the algorithm, executed when number of cities [1, 85].

```
for () // 8192 solutions
     for () // n iterations
          for () // Calculate solution distance
          {

               // Vector solution in registers
               exchange items
               if (improvement)
               {
                    distance = new_distance
               }
               else
               {
                    restore elements
               }
          }
          update solution
```

Table 2. Second version of the algorithm, executed when number of cities (85, 89).

```
for () // 8192 solutions
  for () // n iterations
    for () // calculate solution distance
    {
      // Vector solution in registers
      // Random numbers in shared memory
      exchange items
      if (improvement)
      {
        distance = new_distance
      }
      else
      {
        restore elements
      }
    }
    add distance from the last city of the partial solution
    real with first city of the next stub
    update solution
```

Table 3. Third version of the algorithm, executed n / 45 times when number of cities (90, +].

7 Results parallelization I

The times obtained are shown in Table 4, the first column indicates the number of cities and the rest of the columns the time spent both in CPU and in CUDA with different number of iterations. All executions are performed on 8192 threads.

Table 4. Time (sec.) Different executions of the algorithm with different input sizes.

Ciudades	CPU 10000	CUDA 10000	CPU 100000	CUDA 100000
8	1,11	0,01	10,84	0,16
22	1,42	0,02	14,03	0,21
55	1,97	0,03	19,7	0,29
70	2,58	0,05	25,53	0,47
80	3,54	0,08	35,35	0,69
85	5,27	0,14	52,53	1,2
89	5,4	1,01	53,95	9,83
90	5,66	1,04	56,41	10,41
98	5,66	1,04	56,41	10,41
100	5,66	1,04	56,41	10,41
152	6,07	7,21	60,33	72,27
220	14,2	21,08	141,64	207,77

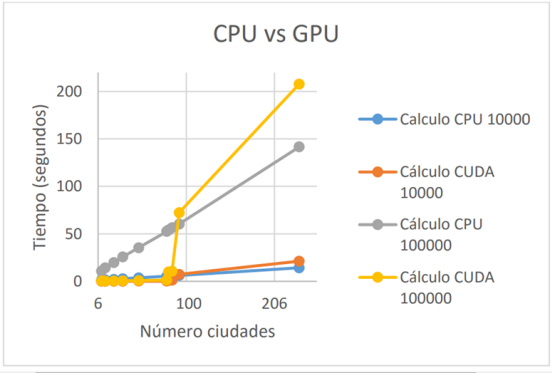

Fig. 6. CPU calculation time versus GPU. As we can see when the number of cities is less than 85 the calculation time in GPU is extremely small earning a significant profit. From 98 cities the CPU wins in calculation time to the GPU.

In Figure 6 we can see the difference between the times obtained with the CPU and with GPU. The problem can be seen clearly. When the number of cities is less than 85 the time difference is very clear not reaching the second in the GPU. This is because both the local solution vector of each thread fits in your records and the distance matrix is in shared memory that is approximately 10 times faster than the global memory. Now when the number of cities is greater than 90 both the vector solutions, and the vector of random numbers, as the vector of distances are in global memory. This produces a considerable bottleneck making practically memory accesses and the execution of the program are sequential, thus eliminating any gain. In this point the CPU exceeds the GPU in calculation time.

8 Results parallelization II

The times obtained are reflected in Table 5, the first column indicates the number of cities and the second column time. All executions are carried out over 8192 threads.

Table 5. Time (seconds) of the different executions of the algorithm with different sizes.

Ciudades	CPU 10000	CUDA 10000	CPU 100000	CUDA 100000
52	9,18	1,73	94,57	16,45
120	20,8	3,82	219,21	37,74
150	26,61	4,74	280,38	47,07
202	35,17	6,5	368,77	64,11
561	99,89	40,02	1053,34	399,39

In Figure 7 and Figure 8 we can see how the time of execution in the GPU when the number of cities is greater than 90. This has been achieved by launching the kernel several times in CUDA where in each execution they store in registers 85 cities, and in memory shared 4000 random numbers. It thus minimizes access to global memory and therefore decreases time of calculation at the cost of obtaining less optimal solutions since the permutations They are made in pieces of 89 elements.

9 Impact of the number of iterations

The number of iterations with which the program is called determines the quality of the solution obtained for a specific city entry. Below is a comparison of the execution.

As can be seen in Table 3, the execution time is less than 10 iterations By counterpart you get a better solution in the with 1000 iterations since the distance is smaller (72 versus 104).

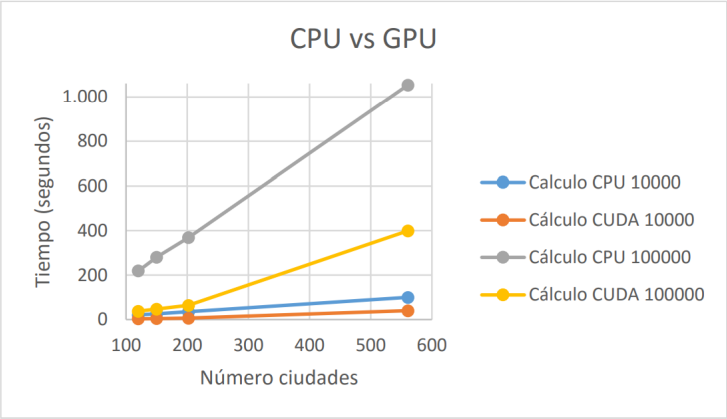

Fig. 7. CPU calculation time versus GPU. As we can appreciate now it they obtain significant gains with more than 90 cities.

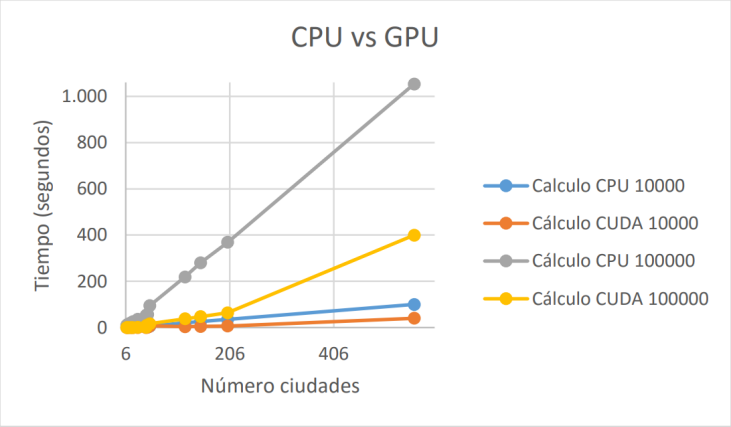

Fig. 8. CPU calculation time versus GPU. Union of both previous graphs.

Table 6. Distance and time of execution of the algorithm for the same input size and different number of iterations.

Ciudades	Iteraciones	Tiempo (segundos)	Distancia
22	10	0,02	104
22	1000	0,19	72

10 Conclusions

In this practice it has been verified how the GPU is able to obtain gains in problems that in appearance are not apt to be paralleled as the programs image processing algorithms that fit perfectly with the CUDA programming philosophy. Some of the results presented gains of 58x when we have a number less than 85 cities. The most remarkable thing about this practice is the importance of optimizing parallel algorithms so that they minimize access to global memory, being evident in section 7 and 8. Finally, this practice can be reused in the subject "high performance computing" taught in the specialty of Systems and Computer Engineering for students to observe and deduce from the different input sizes, when global memory begins to be used more intensively, observing the degradation of the gain. You can also examine the code and propose improvements or simply learn from it.

References

1. NVIDIA Corporation. (2016, May) http://www.nvidia.com/object/what-is-gpu-computing.html
2. Delbosc. Nicolas. 2016. Overview of the CUDA device memory model https://sites.google.com/site/computationvisualization/_/rsrc/1321741184041 /programming/cuda/article1/memory.png
3. StreamComputing BV. 2016. Compute Capability of Fermi and Kepler GPUs. http://streamcomputing.eu/wpcontent/uploads/2012/10/ComputeCapability-ofFermiandKeplerGPUs.png
4. Microway Inc. 2016. CUDA Grid Block Thread Structure. http://www.microway.com/wp-content/uploads/CUDA-GridBlockThreadStructure.png
5. Wikipedia Inc. (2016, May) wikipedia.org. https://en.wikipedia.org/wiki/Travelling_salesman_problem
6. Ellis Horowitz, Sartaj Sahni, Fundamentals of Computer Algorithms, Rockville, Maryland, U.S.A., 1978.
7. Wikipedia Inc. 2016. Solution of a travelling salesman problem. https://upload.wikimedia.org/wikipedia/commons/thumb/1/11/GLPK_solution_of_a_travelling_salesman_problem.svg/220pxGLPK_solution_of_a_travelling_salesman_problem.svg.png