Parallel implementation of a SDM using a GPU for vision-based robot navigation

André Rodrigues¹, André Brandão¹, Mateus Mendes¹², A. Paulo Coimbra¹, Fernando Barros¹ and Manuel Crisóstomo¹.

¹Institute of Systems and Robotics – University of Coimbra
Electrical and Computer Engineering Department, Pólo 2, 3030-290 Coimbra, Portugal
andre_cbr_87@hotmail.com, andre_alexandre_sb@hotmail.com, acoimbra@deec.uc.pt and mcris@isr.uc.pt

²Polytechnic Institute of Coimbra – ESTGOH
Rua General Santos Costa, 3400-124 Oliveira do Hospital, Portugal
mmendes@estgoh.ipc.pt

³Department of Informatics Engineering – University of Coimbra
Pólo 2, 3030-290 Coimbra, Portugal
barros@dei.uc.pt

Abstract- Mobile robotics navigation is an area that has been subject to constant innovation. Possible forms of autonomous navigation may involve visual information and/or odometry. Unlike conventional odometry-based navigation where information from wheel encoders is used to track motion, vision-based navigation performs robot localization using images captured by digital cameras. In the current approach, images are stored in a type of associative memory - Sparse Distributed Memory (SDM), proposed by Pentti Kanerva as a model of the long-term human memory. To speed up processing in real-time, the SDM is implemented in a GPU, taking advantage of GPU parallel processing capabilities. This paper describes the implementation and the performance of a SDM in a GPU, using CUDA programming. The results obtained in the CPU and GPU show that parallelization can improve the performance of the robot.

Keywords– Parallel Programming, Sparse Distributed Memory, SDM, GPU, CUDA, Vision-based Navigation.

1. Introduction

Sparse Distributed Memory (SDM) is a type of associative memory, similar to a random access memory of large capacity, which is adapted to work with high-dimensional vectors and it exhibits properties similar to the human cerebellum [1]. This memory is based on the idea that the distance between concepts in the human memory is linked to distances between points in a high-dimensional boolean space: a point of interest is, in general, quite distant from other points. It consists of pairs of address and data vectors, where each pair of vectors is stored in a physical location. An address activates a set of hard locations (physical locations) in a given radius. Reading/writing from/to a vector is done in those active physical locations. This memory can be used to store different types of information [2].

Parallel computing in GPUs is a technology that can achieve great computational performance when compared with the CPU [3, 4]. This technology is ideal for real-time systems, for it exploits the speed of massive parallel computing to make the system more efficient. There are different architectures that use the GPUs multiprocessors. CUDA [3] stands out for its programming simplicity, since programmers familiar with C or Fortran will quickly program in CUDA.

The goal of this work is to implement a SDM in a GPU, based on a previous implementation in a CPU [5]. This SDM is used by an autonomous robot in its navigation. Images are stored in the memory and used for robot localization. The implementation of the SDM in a GPU increases the performance of the system by speeding up the robot localization process.

Section 2 introduces the SDM. Section 3 describes the history of GPU programming, compares the CPU with the GPU and explains CUDA technology. Section 4 describes the methods used to implement a SDM in a GPU. It also describes the algorithm implemented to locate the robot, in a given space. Section 5 shows the experimental results obtained computing the differences between images in the SDM, both in the CPU and GPU. Results are discussed in Section 6, and conclusions are presented in Section 7.

2. Sparse Distributed Memory

Sparse Distributed Memory (SDM) was proposed by Pentti Kanerva, in the 80’s, as a model of memory that exhibits characteristics similar to the long-term human memory. This memory was presented on a vector is done in those active physical locations. This memory can be used to store different types of information [2].

Parallel computing in GPUs is a technology that can achieve great computational performance when compared with the CPU [3, 4]. This technology is ideal for real-time systems, for it exploits the speed of massive parallel computing to make the system more efficient. There are different architectures that use the GPUs multiprocessors. CUDA [3] stands out for its programming simplicity, since programmers familiar with C or Fortran will quickly program in CUDA.

The goal of this work is to implement a SDM in a GPU, based on a previous implementation in a CPU [5]. This SDM is used by an autonomous robot in its navigation. Images are stored in the memory and used for robot localization. The implementation of the SDM in a GPU increases the performance of the system by speeding up the robot localization process.

Section 2 introduces the SDM. Section 3 describes the history of GPU programming, compares the CPU with the GPU and explains CUDA technology. Section 4 describes the methods used to implement a SDM in a GPU. It also describes the algorithm implemented to locate the robot, in a given space. Section 5 shows the experimental results obtained computing the differences between images in the SDM, both in the CPU and GPU. Results are discussed in Section 6, and conclusions are presented in Section 7.

2. Sparse Distributed Memory

Sparse Distributed Memory (SDM) was proposed by Pentti Kanerva, in the 80’s, as a model of memory that exhibits characteristics similar to the long-term human memory. This memory was presented on a vector is done in those active physical locations. This memory can be used to store different types of information [2].

Parallel computing in GPUs is a technology that can achieve great computational performance when compared with the CPU [3, 4]. This technology is ideal for real-time systems, for it exploits the speed of massive parallel computing to make the system more efficient. There are different architectures that use the GPUs multiprocessors. CUDA [3] stands out for its programming simplicity, since programmers familiar with C or Fortran will quickly program in CUDA.

The goal of this work is to implement a SDM in a GPU, based on a previous implementation in a CPU [5]. This SDM is used by an autonomous robot in its navigation. Images are stored in the memory and used for robot localization. The implementation of the SDM in a GPU increases the performance of the system by speeding up the robot localization process.

Section 2 introduces the SDM. Section 3 describes the history of GPU programming, compares the CPU with the GPU and explains CUDA technology. Section 4 describes the methods used to implement a SDM in a GPU. It also describes the algorithm implemented to locate the robot, in a given space. Section 5 shows the experimental results obtained computing the differences between images in the SDM, both in the CPU and GPU. Results are discussed in Section 6, and conclusions are presented in Section 7.

2. Sparse Distributed Memory

Sparse Distributed Memory (SDM) was proposed by Pentti Kanerva, in the 80’s, as a model of memory that exhibits characteristics similar to the long-term human memory. This memory was presented on a vector is done in those active physical locations. This memory can be used to store different types of information [2].

Parallel computing in GPUs is a technology that can achieve great computational performance when compared with the CPU [3, 4]. This technology is ideal for real-time systems, for it exploits the speed of massive parallel computing to make the system more efficient. There are different architectures that use the GPUs multiprocessors. CUDA [3] stands out for its programming simplicity, since programmers familiar with C or Fortran will quickly program in CUDA.

The goal of this work is to implement a SDM in a GPU, based on a previous implementation in a CPU [5]. This SDM is used by an autonomous robot in its navigation. Images are stored in the memory and used for robot localization. The implementation of the SDM in a GPU increases the performance of the system by speeding up the robot localization process.

Section 2 introduces the SDM. Section 3 describes the history of GPU programming, compares the CPU with the GPU and explains CUDA technology. Section 4 describes the methods used to implement a SDM in a GPU. It also describes the algorithm implemented to locate the robot, in a given space. Section 5 shows the experimental results obtained computing the differences between images in the SDM, both in the CPU and GPU. Results are discussed in Section 6, and conclusions are presented in Section 7.
Kanerva considers that the long-term human memory can be interpreted as a storage system that quickly associates sensory entries with actions that are appropriated for the situation. In the SDM, those sensory entries are represented by high-dimensional vectors which are distributed by a small group of physical locations, termed by hard locations. Each vector written to the SDM is stored in a group of hard locations and it is recovered by, for example, computing the average of the vectors stored in a given radius. This does not guarantee that data retrieved is exactly the original vector stored in the SDM, but most of the times it will be.

One of the problems that have to be solved in an implementation of an SDM is the dimension of space. If the size of the vectors (n) is large, N (Kanerva’s notation to represent the boolean space) will be extremely large too. This is a problem because if, for example, a robot uses the SDM to store images, it will have to deal with state vectors which are very large, so that the hardware must have good computational performance to operate in real time. For a large N, it is almost impossible to implement the memory [1].

To overcome this problem, Kanerva proposes that the memory is sparse. Its size must be only N', a small group of hard locations, much less than N, that manages to represent, by sampling, N.

This memory can be implemented using linked lists or neural networks. Figure 1 shows an example of a SDM model suitable for being implemented using linked lists. This memory comprises an array of addresses that may or may not be activated by the reference address, depending on the distance between the reference address and the addresses distributed across the memory. In this example, the goal is to store the data input vector in the input reference address. In fact, this vector is stored in the two addresses that are at a distance less than or equal to the access radius. To write the data in the vectors, bit counters are used (like in the original model of Kanerva). Those bit counters will be incremented to store 1, and decremented to store 0. To recover the data vector of the reference address, the average of the elements of the vectors of the active locations is calculated column-wise and a threshold is applied.

A simplified version of the memory may use just 1 bit per bit, instead of bit counters. For many practical applications, the result is not significantly affected [5]. The bits may also be grouped as integers, and the absolute sum of the differences may be used to compute the distance between the input address and memory addresses, instead of the Hamming distance, leading to the simplified model shown in Figure 2. This simplified model was the version used in the present work, for it was the one that showed better computational results in previous work [5].

3. GPUs programming

Before the creation of the first GPU, graphics cards were just used for graphic computing applications [6]. After NVIDIA GeForce 256 was created in 1999 [7], the concept of GPGPU (General Purpose Graphics Processing Unit) arose [8]. The GPU could be used for applications different from the traditional graphic rendering like, for example, numerical calculation, image processing or LDPC decoding [9]. GPGPU can have good performance, it has numerous Arithmetic Logic Units and several multiprocessors, to make calculations that require large computational effort. CPUs have a larger amount of memory than the GPU, an efficient cache but a reduced number of cores. Figure 3 illustrates these architectural differences.

There are different architectures and programming languages for GPUs. Among them, NVIDIA CUDA [3] is outstanding, for it can be used with small extensions of C, C++ or Fortran. CUDA is suitable only for NVIDIA processors. In those processors, parallel computing is implemented according to three key ideas: a hierarchy of threads, shared memory and barrier synchronization [10].
A task is divided in several sub-tasks that are executed, in parallel, on thread blocks. Each thread of each block is launched to solve a part of the sub-task. Therefore, threads are grouped into blocks and blocks are grouped in a grid, as shown in Figure 4.

![Figure 4 – Grid of thread blocks.](image)

CUDA cores are divided into Streaming Multiprocessors (SMs). In one SM, several blocks containing a fixed number of threads can be executed, concurrently. Once the blocks are finished, new blocks are launched on the free multiprocessors. Thus, the execution time of a CUDA program, with multiple blocks, in a GPU that has more SMs than another, will be smaller than the execution time of the same program on a GPU with less SMs.

A multiprocessor is designed to run hundreds of threads in parallel by creating, managing, scaling and processing them in groups of 32 threads called warps. An instruction is executed by every thread in a warp in each clock cycle, before moving to the next instruction. This type of architecture is called SIMT (Single-Instruction, Multiple-Thread). If a multiprocessor receives a block with more than 32 threads to process, it divides that block into warps, gathering the threads according to their thread ID. Each multiprocessor has a set of registers, which is divided among the warps, and a shared memory which is divided among the thread blocks [11].

CUDA threads can access data by several memory spaces, represented in Figure 5, namely: registers, local memory, shared memory, global memory, constant memory and texture memory.

The threads of the same block have mechanisms for communication and synchronization. Communication between threads of the same block is done through shared memory. Synchronization is done using a barrier.

### 4. Implementation

In a hard location the pixels of an image will be stored in the address vector and additional information related to the image will be saved in the data vector. Because the SDM will assist the navigation of a mobile robot through visual information, the additional information to be stored is the image number, sequence number and a character that shows the movement the robot was performing when the image was captured. This specific information of each image is important, as it will help the process of motor control in real time navigation and also help localization in ambiguous situations.

To implement the SDM in the GPU, the method chosen was to transform the linked list implemented in the CPU [5] in a pair of global vectors: Address and Data. In a learning process, i.e., when storing images in the SDM, the linked list is updated in the CPU. At the end of the update process, the Address and Data arrays are processed and transferred to the GPU. Figure 6 illustrates this process with a SDM of three hard locations.

![Figure 6 – SDM models implemented in the CPU and in the GPU.](image)

The CPU is only used in the learning process. In the autonomous run mode, the GPU contains all relevant data and localization can be done in parallel computing. In autonomous navigation, the robot compares an image of its current view with all the images stored in the SDM, and, by pixel difference, it can determine what is the closest image in the SDM, and so predict its location.

To compute the difference between images, six kernels (functions executed on the GPU) were implemented. The next topics describe each one of them.

1) **Difference between the pixels of the current image and the stored images, using various blocks in parallel** → In this kernel, the partial (or full if one
block per image stored in the SDM is used) difference of images is calculated by pixel difference. Each thread makes \( x \) differences (depending on the number of blocks per image stored in the SDM), sum them and stores the result in a vector of the shared memory. When all threads have stored their results in the vector, all the elements of the vector are summed using a parallel reduction algorithm\(^2\) \cite{12} to obtain the partial difference of two images. The partial difference is stored in a vector resident in the global memory. Thus, the task of each block is to calculate a partial difference between two images. If one block per image stored in the SDM is used, the resulting value is considered the full difference of two images. 1024 threads per block were used.

2) **Computing the total differences of each image to the current image** \(\rightarrow\) The total difference between the current image and the images stored in the SDM is calculated by adding the partial differences, calculated in kernel 1, and storing the result in a vector of size equal to the number of images stored in the SDM. This difference is also used to check if the hard location that contains the image is within the range of the access radius of the current image, assigning 0 (if it’s outside) or 1 (if it’s inside) to the corresponding element of the image in a vector designated active. A block per image stored in the SDM was used.

3) **Finding the value of the smallest difference** \(\rightarrow\) In this kernel, the smallest difference between the current image and the images stored in the SDM is searched. That value is searched in the vector filled in kernel 2, by a parallel reduction algorithm.

4) **Finding the index of the closest image** \(\rightarrow\) Knowing the value of the smallest difference between the current image and all the images stored in the SDM, in this kernel, the index of image of the SDM that corresponds to that difference, in the two global vectors, is searched. To do this, the value obtained in kernel 3 was compared with all elements of the resulting vector in kernel 2, and the index of the image is found.

5) **Determining active locations** \(\rightarrow\) The elements of the active vector are added through a parallel reduction algorithm to find out the number of active locations.

6) **Presenting the data vector retrieved and the number of active locations** \(\rightarrow\) Data of the returned vector are determined averaging data elements of the active locations. If there is no active location, data vector of the closest image is returned. The number of active locations is also returned.

Using passport-type photos in PGM format, several functional tests were made to check the ability of the new SDM implemented in a GPU to calculate image differences. These tests also served to confirm the results obtained in the GPU, comparing them with the results obtained in the version of the SDM implemented in the CPU. The results were the same for both versions and the closest image displayed by the SDM was correct. Figure 7 shows an example of these tests.

![Image](315x497 to 547x636)

Figure 7 – Example of the first tests made to the SDM in the GPU, with successful retrieving of the closest image.

### 5. Experimental Results

This section presents performance tests and obtained results. The tests consisted of using the SDM in the GPU to calculate the difference of an image to all the images stored in the SDM, identifying the closest image. The calculation time obtained in the GPU was measured and compared to the time obtained in the CPU. These tests were used to confirm the performance improvement provided by the GPU over the CPU. The tests were repeated with three GPUs. Table 1 summarizes the characteristics of the GPUs and computers used.

<table>
<thead>
<tr>
<th>Computer #</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Type</td>
<td>Laptop</td>
<td>Laptop</td>
<td>Desktop</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Core i7-2630QM @ 2.00 GHz, 8 Gb</td>
<td>Intel Core i7-3630QM @ 2.40 GHz, 6 Gb</td>
<td>AMD @ 2.00 GHz, 32 Gb</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA GeForce GT 540M, 2 Gb</td>
<td>NVIDIA GeForce GT 635M, 2 Gb</td>
<td>NVIDIA GeForce GTX 460, 1 Gb</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>96</td>
<td>96</td>
<td>336</td>
</tr>
<tr>
<td>GPU Processor Clock</td>
<td>1.34 GHz, 0.95 GHz, 1.56 GHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Streaming Multiprocessors</td>
<td>2 (48 CUDA Cores / SM)</td>
<td>2 (48 CUDA Cores / SM)</td>
<td>7 (48 CUDA Cores / SM)</td>
</tr>
<tr>
<td>GPU Memory Clock</td>
<td>0.9 GHz</td>
<td>0.9 GHz</td>
<td>2.0 GHz</td>
</tr>
<tr>
<td>GPU Memory Bandwidth</td>
<td>28.8 Gb/sec</td>
<td>43.2 Gb/sec</td>
<td>96.2 Gb/sec</td>
</tr>
</tbody>
</table>

The GPUs of computers 1 and 2 have very similar characteristics. The GPU processor of the computer 1 has

---

\(^1\) In section 5, the performance of the SDM on the GPU is tested with different number of blocks per image stored in the SDM to determine the optimal number of blocks per image.

\(^2\) These algorithms are often used in CUDA to optimize the performance of the program, taking advantage of communication between threads and the speed of shared memory.
a higher speed over the GPU of the computer 2, although lower memory bandwidth.

The GPU of computer 3 has characteristics far superior to the previous ones. The number of SMs is 3.5 times larger than the number of SMs of computers 1 and 2. It also has approximately 2-3 times more memory throughput. Therefore, it can be expected the results obtained on computer 3 will be 3.5-4 times better than those obtained for computers 1 and 2.

The number of images in the SDM ranged from 10 to 10000. 100 images with 176x144 resolution were captured by the robot X80Pro. To simulate a larger number of stored images, the number of hard locations per image was varied. For example, to have 10000 images, each image was distributed over 100 hard locations.

The next three tables show the obtained results in each GPU. The number of blocks stored in the SDM was varied on the kernel 1 from 1 to 25 to be able to find the most appropriate number of blocks per image. Tables 2 to 4 present some of the results. Compilation of the GPU code was done with the flag “-O3”, the highest level. In the tables, “-” means that it was not possible to perform the test because the maximum number of blocks was exceeded.

Table 2 – Results obtained with GeForce GT 540M for different blocks per image.

<table>
<thead>
<tr>
<th>Number of images</th>
<th>CPU time (ms)</th>
<th>GPU time (ms) for different number of blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>0.7</td>
<td>1.2</td>
</tr>
<tr>
<td>20</td>
<td>1.3</td>
<td>1.5</td>
</tr>
<tr>
<td>50</td>
<td>3.0</td>
<td>2.6</td>
</tr>
<tr>
<td>100</td>
<td>6.0</td>
<td>4.4</td>
</tr>
<tr>
<td>500</td>
<td>18.1</td>
<td>11.3</td>
</tr>
<tr>
<td>1000</td>
<td>29.9</td>
<td>19.2</td>
</tr>
<tr>
<td>2000</td>
<td>59.4</td>
<td>35.5</td>
</tr>
<tr>
<td>5000</td>
<td>116.9</td>
<td>70.2</td>
</tr>
<tr>
<td>10000</td>
<td>286.2</td>
<td>173.2</td>
</tr>
</tbody>
</table>

Table 3 – Results obtained with GeForce GT 635M for different blocks per image.

<table>
<thead>
<tr>
<th>Number of images</th>
<th>CPU time (ms)</th>
<th>GPU time (ms) for different number of blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td>20</td>
<td>1.0</td>
<td>1.4</td>
</tr>
<tr>
<td>50</td>
<td>2.4</td>
<td>2.5</td>
</tr>
<tr>
<td>100</td>
<td>4.8</td>
<td>4.2</td>
</tr>
<tr>
<td>300</td>
<td>14.5</td>
<td>11.3</td>
</tr>
<tr>
<td>500</td>
<td>23.6</td>
<td>18.3</td>
</tr>
<tr>
<td>1000</td>
<td>46.9</td>
<td>35.8</td>
</tr>
<tr>
<td>2000</td>
<td>92.7</td>
<td>70.9</td>
</tr>
<tr>
<td>5000</td>
<td>229.8</td>
<td>175.4</td>
</tr>
<tr>
<td>10000</td>
<td>457.8</td>
<td>349.7</td>
</tr>
</tbody>
</table>

Table 4 – Results obtained with GeForce GTX 460 for different blocks per image.

<table>
<thead>
<tr>
<th>Number of images</th>
<th>CPU time (ms)</th>
<th>GPU time (ms) for different number of blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>0.8</td>
<td>1.6</td>
</tr>
<tr>
<td>20</td>
<td>1.5</td>
<td>1.8</td>
</tr>
<tr>
<td>50</td>
<td>3.6</td>
<td>1.7</td>
</tr>
<tr>
<td>100</td>
<td>7.4</td>
<td>2.0</td>
</tr>
<tr>
<td>500</td>
<td>37.5</td>
<td>6.0</td>
</tr>
<tr>
<td>1000</td>
<td>75.5</td>
<td>10.8</td>
</tr>
<tr>
<td>2000</td>
<td>151.1</td>
<td>21.5</td>
</tr>
<tr>
<td>5000</td>
<td>375.5</td>
<td>49.5</td>
</tr>
<tr>
<td>10000</td>
<td>749.4</td>
<td>97.2</td>
</tr>
</tbody>
</table>

From Tables 2, 3 and 4, it can be observed that the best number of blocks per image stored in the SDM is five for GeForce GT 540M and GeForce GT 635M, and four for GeForce GTX 460.

The computation times obtained on the GPU include the time of information transfer from the host (CPU) to the device (GPU), including the current image. Transfers between devices have a high cost in terms of time because the bandwidth between CPU and GPU is inferior to the memory bandwidth of the GPU [13]. Nevertheless, in this case, transfers have to be made because, when the robot uses the SDM on the GPU, it must pass its current image from the host to the device. The number of transfers between devices has been reduced to a minimum.

Table 5 shows the best speedups obtained in the GPUs, for the optimal number of blocks in kernel 1, compared with the computer’s CPUs.

Table 6 shows the best speedups obtained by GeForce GTX 460, relative to computers 1 and 2’s GPUs.

The speedup indicates how much a parallel computing algorithm is faster compared to its sequential version, as in Equation (1).

\[
\text{speedup} = \frac{\text{total time in CPU}}{\text{total time in GPU}}
\]  

Table 5 – Best speedups obtained in the three GPUs compared to the computer’s CPUs.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.58x</td>
<td>0.56x</td>
<td>0.56x</td>
</tr>
<tr>
<td>20</td>
<td>1.00x</td>
<td>0.91x</td>
<td>0.84x</td>
</tr>
<tr>
<td>50</td>
<td>1.67x</td>
<td>1.50x</td>
<td>2.25x</td>
</tr>
<tr>
<td>100</td>
<td>2.14x</td>
<td>1.78x</td>
<td>3.89x</td>
</tr>
<tr>
<td>300</td>
<td>2.83x</td>
<td>2.27x</td>
<td>8.45x</td>
</tr>
<tr>
<td>500</td>
<td>3.02x</td>
<td>2.30x</td>
<td>10.14x</td>
</tr>
<tr>
<td>1000</td>
<td>3.16x</td>
<td>2.47x</td>
<td>13.48x</td>
</tr>
<tr>
<td>2000</td>
<td>3.17x</td>
<td>2.48x</td>
<td>14.12x</td>
</tr>
<tr>
<td>5000</td>
<td>3.17x</td>
<td>2.51x</td>
<td>15.92x</td>
</tr>
<tr>
<td>10000</td>
<td>3.20x</td>
<td>2.51x</td>
<td>16.77x</td>
</tr>
</tbody>
</table>

3 www.drrobot.com
Table 6 – Best speedups obtained in the GeForce GTX 460 related to the times obtained in the GeForce GT 540M and GeForce GT 635M.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.75x</td>
<td>0.56x</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.73x</td>
<td>0.62x</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>1.13x</td>
<td>1.00x</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1.47x</td>
<td>1.42x</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>2.37x</td>
<td>2.37x</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>2.68x</td>
<td>2.70x</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>3.66x</td>
<td>3.50x</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>3.45x</td>
<td>3.50x</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>3.88x</td>
<td>3.88x</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>4.02x</td>
<td>4.08x</td>
<td></td>
</tr>
</tbody>
</table>

6. Discussion

From the results, the speedups achieved with the GPUs of computers 1 and 2 are modest. The reasons for this is the source code (more improvements can be made) and the characteristics of the GPUs used, in particular the low number of CUDA cores, SMs and memory bandwidth. The results obtained with GeForce GTX 460, with a larger number of SMs and memory bandwidth, the speedups obtained are considerable. In all cases, the speedups increase as the number of images stored in the SDM grows. In autonomous navigation, the number of images stored in the SDM can be much higher than 10000, which results in a greater efficiency by the GPU.

The computation times obtained in the GeForce GT 540M and GeForce GT 635M are similar, because both have almost the same features, as seen in Table 5, differing, for example, in the processor speed.

The speedup obtained in the GeForce GT 635M is lower than the speedup obtained in the GeForce GT 540M, due to the higher CPU processor speed.

With 10 or 20 images stored in the SDM, computation times obtained in the GPU are larger than the calculation times obtained in the CPU. One reason for this is that the transfer of information from host to device occupies most of the computation time, since the number of images is very low.

In kernel 1, the computation times obtained in the GeForce GT 540M and in GeForce GT 635M worsen with the use of 6 blocks per image. As for the GeForce GTX 460, computation times worsen with 7 blocks per image. One possible reason for this is the time spent by the multiprocessors scaling the number of blocks.

Comparing the times obtained in the three GPUs, the GeForce GTX 460 obtained speedups of, approximately, 3.5-4x for a number of images equal to or greater than 1000. These results have confirmed our expectations.

With 10000 images stored in the SDM, with GeForce 540M, the robot identifies, approximately, 6 images per second. In the CPU, for the same search space, the robot does not manage to get 2 images per second. Thus, introducing parallelism contributed to increase the performance of robot navigation.

7. Conclusions

The implementation of a SDM in a GPU, using CUDA, showed good results since the process of searching the image stored in the SDM closer to a current image showed lower computation times than those obtained in the CPU. The GPU implementation of the SDM for controlling the robot X80Pro is a good choice, to increase the performance during autonomous navigation.

The use of parallel computing on GPUs, using CUDA architecture, is a good solution to problems such as image search. CUDA can be used in various applications that require high computational processing power in order to speedup computation.

References