ISSN 2464-4617(print) ISSN 2464-4625(CD) CSRN 2702 Computer Science Research Notes
http://www.WSCG.eu

CUDA-based SeqSLAM for Real-Time Place Recognition

Safa Ouerghi
Carthage Univ,
Sup'Com, GRESCOM,
2083 El Ghazela,
Tunisia
safa.ouerghi@supcom.tn

Remi Boutteau
Normandie Univ,
UNIROUEN
ESIGELEC, IRSEEM
76000 Rouen, France
Remi.Boutteau@esigelec.fr

Fethi Tlili
Carthage Univ,
Sup'Com, GRESCOM,
2083 El Ghazela,
Tunisia
fethi.tlili@supcom.tn

Xavier Savatier
Normandie Univ,
UNIROUEN
ESIGELEC, IRSEEM
76000 Rouen, France
Xavier.Savatier@esigelec.fr

ABSTRACT

Vehicle localization is a fundamental issue in autonomous navigation that has been extensively studied by the Robotics community. An important paradigm for vehicle localization is based on visual place recognition which relies on learning a database, then consecutively trying to find matchings between this database and the actual visual input. An increasing interest has been directed to visual place recognition in varying conditions like day and night cycles and seasonal changes. A major approach dealing with such challenges is Sequence SLAM (SeqSLAM) based on matching a sequence of images to the database instead of a single image. This algorithm allows global pose recovery at the expense of a higher computational time. To solve this problem with a certain amount of speedup, we propose in this work, a CUDA-based solution for real-time place recognition with SeqSLAM. We design a mapping of SeqSLAM to CUDA architecture and we describe, in detail, our hardware-specific implementation considerations as well as the parallelization methods. Performance analysis against existing CPU implementation is also given, showing a speedup to six times faster than the CPU for common sized databases. More speedup could be obtained when dealing with bigger databases.

Keywords

Place recognition, global localization, SeqSLAM, robotics, CUDA, GPU.

1 INTRODUCTION

Visual place recognition systems have become increasingly important for a variety of mobile robotic platforms and applications. A typical setup of a visual recognition system contains a given database of images (the map) and the robot consecutively tries to find matchings between this database and the actual visual input (query images). The query is either a single image, which is most common, or an organized set of images *i.e.* a sequence. Recently, there has been an increasing focus on visual place recognition in varying conditions which promoted the design of vision-based systems that can work across common real-world perceptual changes such as varying weather conditions and day and night cycles [4, 16, 10, 11].

A well-known successful approach is SeqSLAM (Sequence SLAM) for visual place recognition introduced by Milford and Weyth in [10] and also described in [9]. This work puts forward the concept of camera-based GPS, which means performing global localization with ensuring to never lose again and aims at matching image sequences under strong seasonal changes. SeqSLAM is based on the computation of image-by-image dissimilarity scores between all query and database images, stored in a so-called difference matrix, then, computing a straight-line path through the full matching matrix to finally select the path with the smallest sum of dissimilarity scores. Despite the quite remarkable results achieved by SeqSLAM, building a full matching matrix introduces substantial computational complexity and enhancing it then computing scores for all straight lines presents an additional computational bottleneck.

However, many problems are being solved using programmable graphics hardware to achieve a certain amount of speedup and the usage of GPU computation has become a popular topic in the community. In fact, the Compute Unified Device Architecture (CUDA) has enabled graphics processors to be explicitly programmed as general purpose shared-memory multi-core processors with a high level of parallelism [6].

In this paper, we focus on an efficient parallelization of the state-of-the-art visual place recognition SeqSLAM to achieve a certain amout of speedup. We consider a single GPU implementation and we describe the strategies to efficiently map the problem components to CUDA architecture.

The paper proceeds as follows: Section II presents some related work. Section III summarises the SeqSLAM algorithm, and brievly reviews GPU architecture and performance considerations. Implementation details are, subsequently, presented in Section IV. Finally, Section IV, presents the results, discusses the outcomes of this paper and presents suggestions for future work.

2 RELATED WORK

Visual place recognition is a well-defined but very broad problem. State of the art techniques for visual place recognition based on features, fall generally into two categories: those that selectively extract parts of the image and those that use the whole image without a selection phase. Examples of the first category are local feature descriptors such as scale-invariant feature transforms (SIFT) [7] and speeded-up robust features (SURF) [1]. In the second category, we find global descriptors such as HOG [3] and GIST [13] that use to process the whole image regardless of its content. In fact, a considerable emphasis was placed on accelerating image descriptors for visual place recognition using CUDA GPUs that many problems are recently being solved with. In [20], Scale-invariant feature transform (SIFT) has been accelerated using CUDA GPU. Speeded-up robust features (SURF) has also been ported to CUDA to achieve a certain amount of speedup [19]. In addition, GIST and HOG global descriptors have been GPU accelerated in [18] and [15] respectively. We find, as well, another broad category of methods for visual place recognition based on a pure image retrieval. This process can be accelerated using inverted indices where image ID numbers are stored against words that appear in the image. FabMAP, a well known technique for visual place recognition along a 1000-km path [2] has relied on an inverted index with a bag-of-words model. Inverted index technique has been recently implemented on GPU demonstrating its efficiency [21]. The bag-of-words technique has also been GPU-implemented [17]. Furthermore, image retrieval systems can be enhanced by adding topological information as both FabMap and SeqSLAM do. Topological maps have in turn been improved by incorporating metric odometry information such as SMART [14] and CAT-SLAM [8]. The metric information within the topological place node can be stored as a sparse landmark map or as a dense occupancy grid map. However, dense spatial modeling has only become feasible in the past few years with the advent of GPU technology [12].

3 BACKGOUND

3.1 SeqSLAM

The SeqSLAM algorithm demonstrated impressive place recognition performance across significant condition variance such as seasonal changes and in case of low quality imagery (low resolution, low depth, and image blur). In order to increase the discriminative nature of the observation and to avoid the problem of false-positives, location is represented in SeqSLAM as a sequence of images, rather than a single image from one pose. Images are, beforehand, resolution-reduced and patch-normalised to enhance contrast. Then, SeqSLAM builds a difference matrix holding SAD (Sum of Absolute Differences) scores between all query and database image sequences. A key processing step is to normalize the image difference values within their (spatially) local image neighborhoods. Subsequently, a search for diagonals of low difference values is performed over the defined sequence length. However, SeqSLAM assumes similar speeds in repeated route traversals and negligible accelerations, limiting its performance in certain application contexts where these criteria are not met.

3.2 GPU architecture summary and performance considerations

NVIDIA GPUs comprise a set of SMs (Streaming Multiprocessors). Each SM consists of many SPs (Streaming Processors) and SFUs (Special Function Units), which are simple but energy-efficient processing units that execute instructions in a SIMD fashion (Single Instruction Multiple Data). The main memory of GPUs, called global memory, can be accessed from all the SPs in every SM. Furthermore, each SM has a set of registers and a small memory (shared memory). This shared memory is much faster than global memory. Each kernel is invoked by a set of threads that are grouped into thread blocks. The blocks are distributed by the hardware among the available SMs. Depending on the amount of required resources, each SM may be able to simultaneously execute several blocks. Each block has assigned an amount of shared memory that allows the exchanging of data among threads of the same block. A more detailed description of NVIDIA's GPU architecture can be found in [5]. The main important features influencing GPU performance are synchronization barriers, the trade-off between available threading and shared resources, coalescence issues in global memory access and shared memory bank conflicts.

4 CUDA-BASED SEQSLAM IMPLEMENTATION

In this section, we present the mapping strategies of SeqSLAM to the CUDA architecture. In total, we deal with three different kernels, each having a specific launch configuration that optimizes the use of the hardware. The first kernel is dedicated to the difference matrix computation, the second to contrast enhancement of the obtained difference matrix and the last one performs route searching in order to find the best match.

4.1 Difference Matrix Computation

The SeqSLAM algorithm uses Sum of Absolute Differences (SAD) to compare each query image I_q from the query sequence $\mathscr{Q} = (I_1, ..., I_Q)$ with $Q = |\mathscr{Q}|$ to each database image I_D from the database Sequence

$\mathscr{D} = (I_1, ..., I_D)$ with $D = |\mathscr{D}|$ and calculates the difference score:

$$d = \frac{1}{R_x R_y} |I_Q - I_D| \quad (1)$$

where d is the difference score and R_x and R_y are the horizontal and vertical image dimensions, respectively. The difference scores are assembled into the so-called difference matrix M of size $D \times Q$.

We assume that the database and query sequences reside in the GPU's global memory as unsigned char types. In fact, database images are cropped, converted to grayscale and resolution-reduced during the preprocessing step done offline and could therefore easily fit into GPU's global memory. On another note, it has been reported in [16, 10] that, the longer the query sequences are, the more difficult is to localize the precise match of a specific frame. This stems from the fact that although longer sequences are more distinguishable, the weight of each frame decreases with the sequence length. These considerations show that both the database and the query sequences could be stored in global memory, the first at the initialization of the GPU after an offline preprocessing, and the second, online acquired, is transferred from host to GPU after preprocessing as well.

Our parallelization strategy is based on a common strategy called tiling. This strategy consists in the partition of the data into subsets called tiles, such that each tile fits into the shared memory of a block. We apply this technique to the difference matrix computation where each tile of the difference matrix is handled by a 2-dimensional block. The tiling process requires for each block to load $Tile_Q$ query images into shared memory and $Tile_D$ database images as described in Algorithm 1. Due to shared memory size limitations, we use the unsigned char type for the allocated buffers in shared memory. Although the use of 1-byte unsigned char type is not recommended in shared memory as bank conflicts occur when threads in the same half warp access the same 4 or 8 bytes wide memory banks, the limited size of shared memory imposes its use. In our implementation, we use the buit-in type *uchar1* denoting an unsigned char. We empirically adjust the pair *(Tile$_D$, Tile$_Q$)* through a *time* vs *(Tile$_D$, Tile$_Q$)* evaluation. The best performance, overall, was achieved by *(Tile$_D$, Tile$_Q$)*$= (4,4)$.

input : $\mathscr{D}, \mathscr{Q}$
output: Difference Matrix M

```
__shared__ uchar1 Buf_D[Tile_D × R_x × R_y];
__shared__ uchar1 Buf_Q[Tile_Q × R_x × R_y];
__shared__ int temp[R_x × R_y/2], M_t[Tile_D × Tile_Q];
blockIdx.x ← B_x, blockIdx.y ← B_y;
s_d ← min(Tile_D, D − B_x × Tile_D);
s_q ← min(Tile_Q, D − B_y × Tile_Q);
threadIdx.x ← Idx, Dim ← R_x × R_y;
for i ← Idx to s_d × Dim do
    Buf_D[i] ← 𝒟[i + B_x × TileD × Dim];
    i ← i + 8 × warpSize;
end
synchronise the threads ;
for i ← Idx to s_q × Dim do
    Buf_Q[i] ← 𝒬[i + B_y × TileD × Dim];
    i ← i + 8 × warpSize;
end
synchronise the threads ;
for i ← 0 to s_d do
    for j ← 0 to s_q do
        Initialize temp ;
        for k ← Idx to Dim do
            id ← mod(k, 8 × WarpiSize);
            temp[id] ← temp[id] + |Buf_D[i × Dim +
            k] − Buf_Q[j × Dim + k]|;
            i ← i + 8 × warpSize;
        end
        synchronise the threads ;
        M_t[i × s_q + j] ← Reduction(temp);
    end
end
synchronise the threads ;
Copy M_t to global memory;
```

Algorithm 1: Parallel difference matrix computation

4.2 Difference Matrix Contrast Enhancement

After creating a difference matrix comparing all previously seen locations to each other using SAD scores, SeqSLAM employs local neighbourhood normalisation to remove biases from lighting variations between route traversals. Each element M_i in the difference matrix M is normalised within a fixed range l by subtracting the local mean and dividing by the local standard deviation to enhance the local matching contrast. This process gives the new difference matrix $\hat{M}$:

$$\hat{M}_i = \frac{M_i - \bar{M}_l}{\sigma_l} \quad (2)$$

where $\bar{M}_l$ is the local mean and σ_l is the local standard deviation, in a range l templates around template i.

We apply, as well, the tiling technique to enhance the previously calculated difference matrix, stored in global memory in a row major order. The tile size is fixed to $Tile \times Q$. The threads within each block handle the computation of a tile and are designed to be two-dimensional. In our design, each block of index *0<blockIdx.x<gridDim.x-1* will load $Tile+l$ of M. The blocks handling the border tiles, *i.e.* $block_0$ and $block_{gridDim.x-1}$ only load $Tile+\frac{1}{2}l$.

The cuda kernel begins with loading a tile of M into a preallocated shared memory of size $(Tile+l) \times Q$.

Next, the computation of Q means for *Tile* rows of the enhanced $\hat{M}$ is handled by $Q \times Tile$ threads where *threadIdx.x*<*Tile* and *threadIdx.y*<Q. Each thread sums $(1+l)$ values of M in a column where the y-coordinate is *threadIdx.y* and the x-coordinate starts from *threadIdx.x* to finally divide the sum by the number of values summed up *i.e.*, $(1+l)$. After the kernel has finished computing the means, threads are synchronised to subsequently perform the computation of standard deviations the same way they did for the means computation thus described using the formula of standard deviation and the means already calculated. However, border tiles handled by *blockIdx.x=0* and *blockIdx.x=gridDim.x-1* are special cases. In this way, for the first block, if *threadIdx.x*$\leq \frac{1}{2}l$ the local neighbourhood would be below $(1+l)$ and have thus to adapt its range with each row of $\hat{M}$, *i.e.* each *threadIdx.x*. The same principle applies to the last block. If *threaIdx.x*$\geq \frac{1}{2}l$ the range has to be adapted as well. In our implementation, we compute the range for each thread using thread registers. Finally, each block writes in column major the tile of $\hat{M}$ to global memory. Implementation details thus described are presented in Algorithm 2. However, we only describe the *means* computation given that the same principle applies to standard deviations computation. The final straightforward step is also not presented and it consists in substracting the *mean* and dividing by the *standard deviation* each element M_i to obtain $\hat{M}_i$.

4.3 Route Searching

The inputs to the localization algorithm are the query sequence locally acquired and the database Sequence that serves as the visual memory of the robot. The algorithm attempts to find a subsequence of equal length in the database that is the most similar to all regarded subsequences. This is performed by building the image difference matrix as stated above, contrast enhancing it and then searching for the minimizing sub-route referring to connected regions of low difference in the difference matrix resembling to lines shapes. This process is presented in Figure 1 where, for each starting search position in left column (marked by a green dot), a range of slopes is traversed (highlighted by the semi-transparent green area) modelizing possible other sub-routes that will be traversed for other slopes. For each matrix entry that is traversed during a sub-route traversal, its difference value is added to an accumulative value that is initialized with zero in the beginning. This value is called sub-route score. Since there is a range of possible slopes for each starting point there is an equal number of scores for each sub-route starting point. When all scores are calculated for one starting point the minimal score of them is selected. Finally, the sub-route with the smallest score and the second smallest score are used to compute the best database matching to the input query sequence.

input : Difference Matrix M, l
output: Enhanced Difference Matrix $\hat{M}$

$\_\_shared\_\_\ \hat{M}_{sh}[Tile][Q], M_{sh}[l+Tile][Q];$
$\_\_shared\_\_\ Mean[Tile][Q];$
Initialize M_{sh} *to* 0;
$Idx \leftarrow threadIdx.x, Idy \leftarrow threadIdx.y,$
$B_{id} \leftarrow blockIdx.x, grid \leftarrow gridDim.x;$
$a \leftarrow \max(0, Tile \times Bl_{id} - \frac{1}{2}l);$
$b \leftarrow \min(D, Tile \times Bl_{id} + (Tile-1) + \frac{1}{2}l);$
if $Idx \leq b-a$ **and** $Idy < Q$ **then**
  $M_{sh}[Idx][Idy] \leftarrow M[a+Idx][Idy];$
end
synchronise the threads ;
if $(Idx \leq Q)$ **and** $(Idy < Tile)$ **and** $(B_{id} = 0)$ **then**
  $a \leftarrow \max(0, Idx - \frac{1}{2}l);$
  $b \leftarrow \min(1+l, 1+Idx+\frac{1}{2}l);$
  $Mean[Idx][Idy] \leftarrow \frac{1}{b} \cdot \sum_{i=0}^{b} M_{sh}[a+i][Idy];$
end
synchronise the threads ;
if $(Idx \leq Q)$ **and** $(Idy < Tile)$ **and** $(0 < B_{id} < grid)$ **then**
  $Mean[Idx][Idy] \leftarrow \frac{1}{1+l} \cdot \sum_{i=0}^{1+l} M_{sh}[Idx+i][Idy];$
end
synchronise the threads ;
if $(Idx \leq Q)$ **and** $(Idy < Tile)$ **and** $(B_{id} = grid-1)$ **then**
  $b \leftarrow (1+\frac{1}{2}l) + \min(D - B_{id} \times Tile - 1 - Idx, \frac{1}{2}l);$
  $Mean[Idx][Idy] \leftarrow \frac{1}{b} \cdot \sum_{i=0}^{1+l} M_{sh}[Idx+i][Idy];$
end
synchronise the threads ;

Algorithm 2: Parallel mean computation for Difference Matrix contrast enhancement

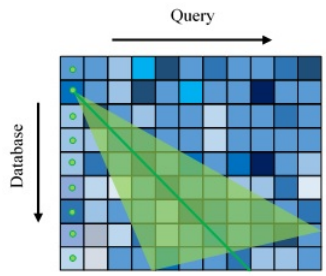


Figure 1: Minimal slope search in the difference matrix.

One of the main important features influencing GPU performance is coalescence issues in global memory access. However, subroute scores computation require multiple non coalesced global memory accesses (to sum up values in diagonals). We hence aim to design a novel subroute score computation that enables coalesced accesses.

One common solution to coalescence issues is the use of shared memory to load data from global memory in a coalesced fashion, then, have contiguous threads stride through it. Unlike global memory, there is no penalty for strided access in shared memory especially when there is no bank conflict. We assume that the difference

matrix sizes are bigger than the shared memory of a single block, which would be the case for the majority of real applications. It is also worth noting that depending on the database size is not a limiting factor in our implementation as the whole process of SeqSLAM is based on a learning phase where several learnable parameters have to be tuned.

We present our design using the mapping vector technique that assigns CUDA ressources including blocks, threads and registers to SeqSLAM data. We will first begin with presenting the global memory mapping vector followed by the SM mapping vector. The main structure of SeqSLAM; the difference Matrix M of size $D \times Q$, is stored in global memory in a column major order according to the following mapping vector:

$$[\underbrace{M_0 M_1 \dots M_{D-1}}_{\text{col 0}} \underbrace{M_D M_{D+1} \dots M_{2D-1}}_{\text{col 1}} \dots \underbrace{M_{D(Q-1)-1} \dots M_{DQ-1}}_{\text{col Q-1}}] \tag{3}$$

This means that columns of size D are contiguously stored in global memory. Hence $M(0,0)$ starts at location 0, $M(0,1)$ at location D and $M(i,j)$ at location $j \times D + i$.

The proposed mapping vector of data on the *SM* ressources is the following:

$$[\underbrace{M_0, \dots, M_{2D-1}}_{\text{block 0}}, \underbrace{M_{2D}, \dots, M_{4D-1}}_{\text{block 1}}, \dots, \underbrace{M_{2\times i\times D}, \dots, M_{2\times(i+1)\times D-1}}_{\text{block i}},$$
$$\dots, \underbrace{M_{D(Q-2+MOD(Q,2))-1}, \dots, M_{DQ-1}}_{\text{block ceil(Q/2)}}] \tag{4}$$

This means that each block i performs a coalesced memory access to global memory in order to load data from location $2 \times i \times D$ to location $2 \times (i+1) \times D - 1$ into its shared memory which is equivalent to 2 columns of the difference Matrix M. The last block of index $blockIdx.x = ceil(Q/2)$ loads the remaining columns which would be either two if $MOD(Q,2) = 0$ or only one column if not. Thus, a total number of $ceil(Q/2) + 1$ one-dimensional blocks is used in the launch configuration of the kernel and 4 warps of threads per block.

As shown in Algorithm 3, the function of each block is, therefore, to compute a subscore for each sub-route where the used values are only strided by D+*offset* with *offset*$\in \{0,1,2\}$. Within a block, the subscore of a given sub-route for all slope possibilities is handled by a thread. We note that simultaneous accesses of threads within a block to shared memory are done to consecutive values, *i.e.* a column of the matrix M to minimize bank conflicts. As only 4 *warps* are used per block, $128 \times k$ subscores are computed in parallel by the active threads, where k designs the number of slope possibilities. The process is, hence, repeated $(1 + floor(D/128))$ times to account for the D possible subroutes. The $128 \times k$ subscores are then written to global memory via an *atomicAdd* between the active blocks. We note that the *atomicAdd* is done inside the loop due to the limited size of shared memory and only $128 \times k$ floats are allocated for *Scores* vector. The final Score for each sub-route would be equal to the sum of the calculated subscores for the slope possibility k.

5 RESULTS

In this section, we describe the conducted experiments to evaluate the performance of porting SeqSLAM to CUDA GPU. We will first begin with exposing the experimental setup, then, discuss the performance expressed in terms of timing of our implementation versus exsisting CPU implementation.

5.1 Experimental Setup

For the experiments described in the following, we extracted still frames from the original video of the Nordland dataset[1], downsampled them to 64×32 and converted them into grayscale. It is worth noting that in related litterature, the downscaling was 64×32 for not only the Nordland dataset, but also the Alderley[2]. The downscaling was even greater for other datasets such as the Nurburgring dataset, equal to 32×24 and the approach of SeqSLAM is fundamentally based on important downscaling ratios.

Furthermore, all the data reside in the GPU device memory at the beginning of each test, so there are no data transfers to CPU during the benchmarks to prevent interactions with other factors in the study. The performance of the experiments for Parallel SeqSLAM is measured in time t in $milliseconds(ms)$. The system on which our implementation was evaluated is equipped with an i7 CPU running at up to 3.5 GHz, the intel i7 CORE. The CUDA device is an NVIDIA GeForce GTX 850M running at 876 MHz with 4096 MB of GDDR5 device memory. The evaluation has been performed with CUDA version 7.5 integrated with VisualStudio 2012. At the first execution of SeqSLAM, memory allocations have to be performed. This is required only once and takes about $10ms$. All the experiments were run for 5 database sequence lengths as presented in Table 1. Morever, 3 query sequences were used with 3 different lengths as presented in Table 2. As stated before, database sequences were obtained using the open-Source Code OpenseqSLAM[3] by varying the parameter *imageSkip*. However, the CPU-based code used

[1] http://nrkbeta.no/2013/01/15/nordlandsbanen-minute-by-minute-season-by-season/

[2] https://wiki.qut.edu.au/display/cyphy/Michael+Milford+Datasets+and+Downloads

[3] https://openslam.org/openseqslam.html

input : $\hat{M}$, mov_{min}, mov_{max}
output: $Match_{index}$, $Match_{score}$

$slopes \leftarrow mov_{max} - mov_{min} + 1$;
$Idx \leftarrow threadIdx.x$;
$B_{id} \leftarrow blockIdx.x$;
$\_\_shared\_\_$ $Icrem_{indices}[slopes][Q], Score[slopes][Q]$;
$\_\_shared\_\_$ $Buffer[2 \times D], Score[D][128]$;
if $Idx = 0$ **then**
  $Slopes \leftarrow FindSlopePossibilities(mov_{min}, mov_{max})$;
  $Icrem_{indices} \leftarrow SlopeIndices(Slopes)$;
end
synchronise the threads ;
for $i \leftarrow Idx + 2 \times D \times B_{id}$ **to** $2 \times D \times (1 + B_{id})$ **do**
  $Buffer[i - 2 \times D \times B_{id}] \leftarrow \hat{M}[i]$;
  $i \leftarrow i + 4 \times warpSize$;
end
synchronise the threads ;
for $s \leftarrow Idx$ **to** D **do**
  $intindices[slopes][Q]$;
  for $i \leftarrow 0$ **to** $slopes$ **do**
    for $j \leftarrow 0$ **to** Q **do**
      $indices[i][j] \leftarrow j \times D + s +$
      $\min(Icrem_{indices}[i \times Q + j], D - s)$;
    end
  end
  synchronise the threads ;
  for $i \leftarrow 0$ **to** $slopes$ **do**
    $sum \leftarrow 0$; **for** $j \leftarrow 0$ **to** 2 **do**
      $index \leftarrow indices[i][j + 2 \times B_{id}]$;
      $sum \leftarrow sum + Buffer[index - 2 \times D \times B_{id}]$;
    end
    $Subscore[i][mod(s, 128)] \leftarrow sum$;
  end
  synchronise the threads ;
  atomicAdd of Subscore in Temp of size $D \times slopes$
  in global memory;
  $s \leftarrow s + 4 \times warpSize$;
end
$Idx \leftarrow threadIdx.x \times blockDim.x + blockIdx.x$;
for $k \leftarrow Idx$ **to** D **do**
  $Score[k] \leftarrow$ `MinOverSlopes`$(Temp)$;
  $k \leftarrow k + blockDim.x \times gridDim.x$;
end
synchronise the threads ;
$min1 \leftarrow$ `FindMinWithShuffle`$(Score)$;
$index_1 \leftarrow$ `FindIndexOfMin`(min_1);
Set values in Score around min_1 *of radius* R_{window} *to max machine value*;
$min2 \leftarrow$ `FindMinWithShuffle`$(Score)$;
$Match_{index} \leftarrow index_1$;
$Match_{score} \leftarrow min_1 / min_2$;

Algorithm 3: Parallel Route Searching

in benchmarking is a C++/OpenCV port of OpenSeqSLAM [4].

Database	Database Seq Length	imageSkip
D1	714	50
D2	1428	25
D3	2747	13
D4	3570	10
D5	5100	7

Table 1: Database Sequences

Query	Query Seq Length
query 1	11
query 2	20
query 3	32

Table 2: Query Sequences

5.2 CUDA based implementation vs CPU based implementation

5.2.1 *Difference Matrix Computation timing*

In the first experiment, we measured the execution time of difference matrix computation kernel using the query and the database sequences presented in Table 1 and Table 2 respectively. We selected the pair $(Tile_D, Tile_Q) = (4,4)$. The evaluation is depicted in Figure 2 for $query = 32$ showing an increasingly important speedup of CUDA-based parallel SeqSLAM with the database sequence length. The speedup is averaged at $4\times$.

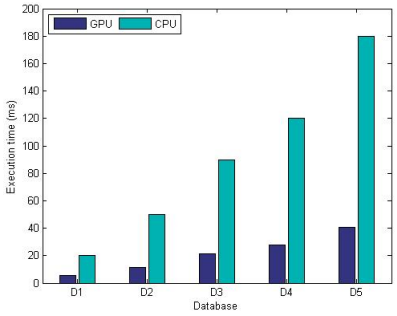


Figure 2: Difference matrix computaion.

5.2.2 *Difference Matrix Enhancement timing*

We subsequently evaluated the performance of parallel SeqSLAM contrast enhancement over Sequential contrast enhancement. The optimal tile selected is $Tile = 20$ and the performance for both sequential and parallel implementations is presented in Figure 3 for $query = 32$. The obtained speedup exceeds $16\times$ for D1 and almost $13\times$ for D5. Overall, a good speedup was shown with parallel seqSLAM averaged at $14\times$.

[4] https://github.com/subokita/OpenSeqSLAM

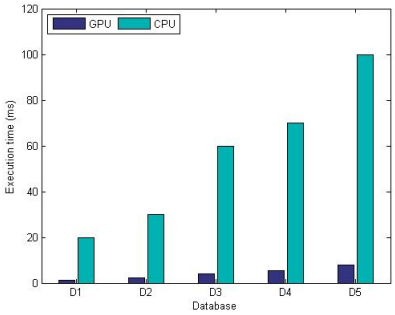


Figure 3: Difference matrix enhancement.

5.2.3 *Route searching timing*

The third experiment was dedicated to the comparison of the Route searching execution time for both CPU-based and CUDA GPU- based implementations. The speedup of GPU over CPU is clearly visible averaged at almost 6× for the datasets used in the evaluation and is depicted in Figure 4 for $query = 32$. An interesting point is that the route searching is not a botteleneck for CPU when attempting to localize a single image using a short query sequence as recommended. However, we aim to design a GPU only solution and a good speedup was though shown.

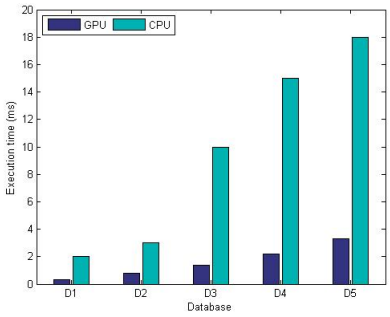


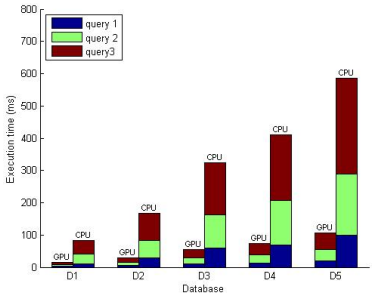
Figure 4: Route searching.

5.2.4 *Performance anlalysis of CUDA based SeqSLAM*

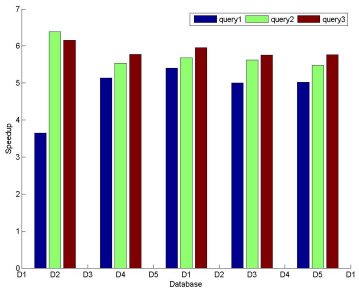
In Figure 5, we show the performance results of parallel SeqSLAM using a CUDA GPU. Firstly, in Figure 5a, we compare the mean computation time of CPU and GPU implementations, for the different database and query sequences used in this evaluation. We show a computation time even more important for CPU reaching 298ms for $D5$ and $query3$ against 51ms for GPU. In figure 5b we demonstrate that the speedup is about 6× compared to the CPU implementation.

5.3 Discussion

A special feature of the Nordland dataset is that the viewpoint for the database and query sequences is exactly the same. The camera has only one degree of freedom along its track and corresponding images from two traverses in different conditions overlap almost perfectly. This condition is only met for certain applications where the camera has only one degree of freedom and SeqSLAM remains a major approach from which many recent approaches are being built upon. Hence, since our aim is to enhance timing and not localization precision, we have relied on the Nordland dataset images as input for different database and query sequences sizes. Furthermore, we aimed to design a GPU only solution as in a real application for a robotic platform, several modules are required and we can put some tasks that do not depend on the results of the kernel on the CPU while the GPU is executing a different task in order to achieve a better performance and reduce the hardware complexity. In fact, this is possible thanks to the asynchronous execution feature between the CPU and the GPU of CUDA systems meaning that the control returns to the CPU immediately after the kernel is launched.



(a) GPU vs CPU time



(b) Speedup

Figure 5: Performance of CUDA based SeqSLAM.

6 CONCLUSION

In this paper, we have proposed a parallel CUDA GPU based implementation of Sequence SLAM (SeqSLAM) for visual place recognition. SeqSLAM is a well-known successful approach for mobile localization and place recognition in varying conditions like seasonal changes and day and night cycles when certain conditions are met. Our parallelization method was based on the allocation of the three major steps of the approach to three GPU kernels, each of which with a specific launch

configuration promoting the best possible performance. Experimentations showed promising results thanks to the parallel exploitation of CUDA ressources that offered a good overall speedup averaged at 6 times better than its CPU counterpart. As future work, we plan to apply other algorithmic techniques instead of simple image difference to deal better with a reasonable viewpoint change. We plan also to use a multi-GPU system to scale with bigger databases and achieve a greater speedup.

7 REFERENCES

[1] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-up robust features (surf). *In. Elsevier Computer Vision Image Understanding*, 110:346–359, 2008.

[2] M. Cummins and P. Newman. Appearance-only slam at large scale with fab-map 2.0. *In. The International Journal of Robotics Research (IJRR)*, 30:1100–1123, 2011.

[3] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. *Proc. of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), Washington, DC, USA*, 110:886–893, 2005.

[4] E. Johns and G. Z. Yang. Feature co-occurrence maps: Appearance-based localisation throughout the day. pages 3212–3218, 2013.

[5] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: a Hands-on Approach*. 2010.

[6] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *In. IEEE Microwave Magazine (IEEE Micro)*, 28:39–55, 2008.

[7] D. G. Lowe. Object recognition from local scale-invariant features. *Proc. of the International Conference on Computer Vision (ICCV), Washington, DC, USA*, 2:1150–, 1999.

[8] W. Maddern, M. Milford, and G. Wyeth. Cat-slam: probabilistic localisation and mapping using a continuous appearance-based trajectory. *In. The International Journal of Robotics Research (IJRR)*, 31:429–451, 2012.

[9] M. Milford. Vision-based place recognition: how low can you go? *In. The International Journal of Robotics Research (IJRR)*, 32:766–789, 2013.

[10] M. Milford and G. Wyeth. Seqslam: Visual route-based navigation for sunny summer days and stormy winter nights. pages 1643–1649, 2012.

[11] T. Naseer, L. Spinello, W. Burgard, and C. Stachnis. Robust visual robot localization across seasons using network flows. pages 2564–2570, 2014.

[12] R. A. Newcombe, S. J. Lovegrove, and A. J. Davison. Dtam: Dense tracking and mapping in real-time. *Proc. of International Conference on Computer Vision (ICCV), Barcelona, Spain*, pages 2320–2327, 2011.

[13] A. Oliva and A. Torralba. Building the gist of a scene: the role of global image features in recognition. *In. Visual Perception, Progress in Brain Research*, 155, Part B:23 – 36, 2006.

[14] E. Pepperell, P. Corke, and M. Milford. All-environment visual place recognition with SMART. *Proc. of IEEE International Conference on Robotics and Automation (ICRA), Hong Kong, China*, pages 1612–1618, 2014.

[15] V. Prisacariu and I. Reid. fasthog - a real-time gpu implementation of hog. Technical Report 2310/09, Department of Engineering Science, Oxford University, 2009.

[16] N. Sunderhauf, P. Neubert, and P. Protzel. Predicting the change, a step towards life-long operation in everyday environments. 2013.

[17] K. E. A. van de Sande, T. Gevers, and C. G. M. Snoek. Empowering visual categorization with the gpu. *In. IEEE Transactions on Multimedia (IEEE Trans. Multimedia)*, 13:60–70, 2011.

[18] Y. Wang, Z. Feng, H. Guo, C. He, and Y. Yang. Scene recognition acceleration using cuda and openmp. *Proc. of First International Conference on Information Science and Engineering (ICISE), Nanjing, China*, pages 1422–1425, 2009.

[19] W. Yan, X. Shi, X. Yan, and L. Wang. Computing opensurf on opencl and general purpose gpu. *In. International Journal of Advanced Robotic Systems (IJARS)*, 10:375, 2013.

[20] Z. Yonglong, M. Kuizhi, J. Xiang, and D. Peixiang. Parallelization and optimization of sift on gpu using cuda. *Proc. of IEEE International Conference on High Performance Computing and Communications (HPCC), Zhangjiajie, China*, pages 1351–1358, 2013.

[21] J. Zhou, Q. Guo, H. V. Jagadish, W. Luan, A. K. H. Tung, Y. Yang, and Y. Zheng. Generic inverted index on the GPU. *In. Computing Research Repository (CoRR)*, abs/1603.08390, 2016.