A real time implementation with C++ and CUDA of Incremental Principal Component Pursuit for Video Background Modelling

Djeefther Souza Albuquerque

BEng Computer Engineering

Ph.D., Veronica Teichrieb

Ph.D., João Marcelo Teixeira

Centro de Informática

Universidade Federal de Pernambuco

10/02/2017

Summary

Background Subtraction aims to separate what in a video is foreground (moving objects) from the background. So this project aims to extend the IncPCP from (Rodriguez and Wohlberg 2015) that already is implemented in MATLAB with implementations in C++ and C++/CUDA versions speeding this application up and becoming possibly in real time without deterring the accuracy of the current method.

This project is going to evaluate and compare the original version of IncPCP and ours two versions regarding accuracy (F-measurement) and speed (FPS (Frames per Second)). For this purpose this project uses datasets with ground truth.

In the path to accomplish this goal are expected 6 deliverables: a CudaGemm function, a CudaSVD function, an IncSVD function in GPU and in CPU and an IncPCP algorithm in GPU and in CPU.

From these 6 deliverables, all of them but IncPCP using CUDA were delivered with good quality. All the other 5 deliverables are working with the precision expected and faster than the MATLAB version. The IncPCP in C++ is 152% faster than MATLAB and even without being the goal of this project is 17% more accurate than MATLAB. The IncSVD is also faster, something between 2x and 3x, when comparing GPU and CPU versions. The IncPCP using CUDA itself is not faster as expected, but with good precisions, in fact, being 50% slower.

Acknowledgements

My both supervisors Veronica and João Teixeira for the help, guidance and support.

My parents, Rildo dos Reis Albuquerque and Kdma Gomes e Souza for always encouraging me to study as much as possible and for always be with me.

All my friends from university, especially Guilherme Caminha, João Gabriel Machado that we had make so much projects together and flying around the world because of it. I own you so much because of that. Also friends of my college: Jefferson Ramos, Erick Lucena, Heitor Araujo, Ihago Henrique, Andrea Brandão, Mayara Cavalcanti, Newton Barbosa, Alice Zlocovikc, Saulo Albuquerque, Yago Gomes, Renato Albuquerque and a lot of others.

Teachers from my university, all of that that encourage me to continuing studying and learning, among them Edna Barros, Adriano Sarmento, Carlos Alexandre, Suruagi, Abel Guilhermino and all others.

Pyetro Ferreira who I had great pleasure to call boss and start in the world of image processing and computer vision as all other friends from LINCS/CETENE.

My both teachers at University of Kent: Kostas and Sanaul without this work would not be possible, thank you for all patience and advice. Yang Hu, that even not being a formal supervisor helped me as one, explained to me the basis of complex background that I should know to understand the papers, thank you Yang.

When outside home I have to thank for Mariana Lucena, Renata Maia, Leticia Rosique and Bruno Brugger for being almost my family outside home. I hope we are going to see each other more often around world.

Summary

Acknowledgements

1. Introduction

2. System Description

2.1 SVD

2.2 CUDA SVD

2.3 GEMM Wrapper

2.4 IncSVD

2.4.1 Rank1IncSVD

2.4.2 Rank1DwnSVD

2.4.3 Rank1IncSVD

2.4.4 IncSVD class

2.5 CUDA IncSVD

2.6 IncPCP and CUDA IncPCP

2.6.1 Flags in Original Version

2.6.2 Flags in this Project

2.6.3 Algorithm

2.7 Internal Modules

2.7.1 Find Files

2.7.2 Images From Folder

2.7.3 Accumulators

2.7.4 Tictoc

2.7.5 Binary Confusion Matrix

2.7.6 Tests

2.7.7 Scripts

3. System Implementation

4. Results and Discussion

4.1 Method for computational performance tests

4.2 Method for numerical precision tests

4.3 CUDA GEMM

4.4 SVD

4.5 Datasets

4.6 IncPCP qualitative view

4.7 CUDA SVD

4.8 IncSVD

4.9 CUDA IncSVD

4.10 IncSVD Time comparison

4.11 IncPCP Threshold and F-measurement analyses

4.12 IncPCP timing

5. Conclusion and Future Work

6. References

1. Introduction

Background Subtraction detection aims to separate what in a video is foreground (moving objects) from the background. In Figure 1 is shown a general view of how it works. A proper background subtraction should only highlight relevant movement; in the case in Figure 1 the water movement is not relevant. For accomplishing that the background model should avoid all not desired changes in the foreground, including movements, light changes, jitter or other kinds of noise.

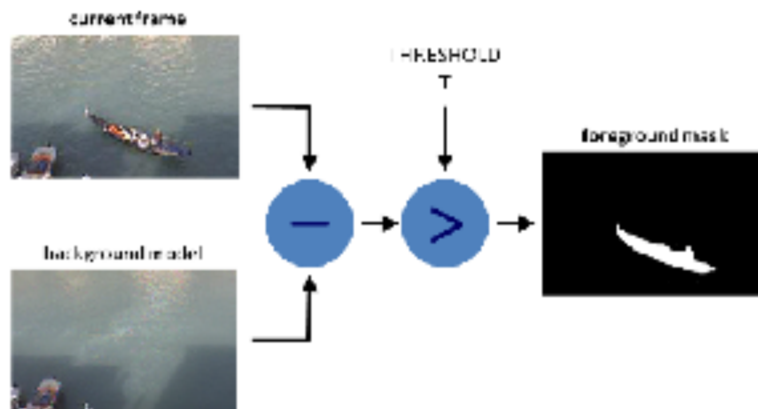


Figure 1: General scheme of a background subtraction. Image obtained from (Maghsoumi et el 2015).

It is a valuable tool for many kinds of applications, like vehicles navigation and surveillance, and automatic tools in graphics software. Those applications require a real time solution. As an example, in Figure 2 it is possible to see an automatic traffic analyser that internally uses a background subtraction algorithm to detect cars movement.

Figure 2: Example of a background subtraction algorithm for automatic traffic analysis. Image obtained from (Ferreira 2016).

1. Goals

This work intends to extend a previous work on background subtraction detection made by Rodriguez and Wohlberg (2016) called IncPCP (Incremental Principal Component Pursuit), translating the algorithm already implemented in MATLAB® (code could be download from Rodriguez and Wohlberg (2015a)) to C++, and after adding CUDA (Nvidia, 2016), to this using the GPU as a coprocessor. This way, we intend this software to become faster and possibly real time, without any loss of accuracy in regards of the original algorithm.

C++ is a language that combines both low-level and high-level programming features and allows us to explore more efficiently the CPU making the software possibly faster. CUDA is a framework that permits the use of GPUs for general purpose computation. Thus, it became possible to use those massive parallel hardware with a larger throughput than a CPU with the same price.

2. Outline

In next section 1.3 the reader are going to be introduced about the general state of the art of background subtraction and what is the position of this algorithm. In Section 2 the system itself produced by us, the original system in MATLAB and the algorithm are explained. The flags explained in Section 2 are important as parameters to algorithm that is going to be explaend at the end in Section 2.6.3. In Section 3 we see a fee about the system organization. Finally in Section 4 the results, about correctness and timing of all the system and subparts.

3. General Description

PCP (Principal Component Pursuit) is considered the state-of-the-art for video background modelling (Rodriguez and Wohlberg 2016). It is an algorithm that models the background of a video as a low-rank matrix as , where (number of rows, columns and depth or channels, respectively) and is the number of frames analysed. The PCP finds the background and foreground solving the optimization problem in Equation (1). and are the video input and foreground, respectively, and they have the same dimensions as , so

(1)

PCP algorithms are used as online and offline methods. The offline methods process an entire video in batch, analysing all the frames at once. An online method processes each new frame based on a background model and so can accept video streams. The online method can have a static or adaptive background model. Our implementation is an online algorithm what fits best to those applications this monograph has mentioned before.

PCP algorithm requires a partial SVD (Singular Value Decomposition) (HARTLEY, R., ZISSERMAN, A. 2004) evaluation, as can be seen in Equations (2) and (3). This significant mathematical operation makes this algorithm's timing and memory consuming, since from an input , where m is the number of pixels of image and n the timing parameter, it has to keep in memory three matrices that together consumes as output from a full SVD (2) and just after reduces to a partial SVD (3). In an online version of the PCP, this full SVD operations are done for each new frame computing the last frames (as being a memory parameter), without any data reutilization.

Where for full-size SVD:

- is the complex or real matrix input,
- is an unitary matrix,
- is an diagonal matrix with non-negative real number on the diagonal, and
- is an unitary matrix, the conjugate transpose of .

Moreover, a fixed rank factorization is a partial SVD with rank equal to and it is going to be:

(3)

Where for r rank SVD:

- *M* is the complex or real m × n matrix input,
- *r* is an integer input for the rank that should be less than or equal to *)*,
- is an unitary matrix,
- is an diagonal matrix with non-negative real number on the diagonal, and
- is an unitary matrix, the conjugate transpose of .

The multiplication from the output from Equation (2) by the full SVD always give the input itself, when the multiplication of output terms from Equation (3) always return an *r*-rank

matrix with the most principals components from M. What depending on the chosen and the level of redundancy from input could be a good approximation or a rough simplification.

Rodriguez and Wohlberg (2016) work made possible to compute an SVD based on a previous SVD from the last frame keeping or increasing the rank in this operation, without the need for recomputing all SVD and after reducing the rank. This allows us to save memory (keeping a low rank all time) and time (reutilizing already computed data). The input memory usage is just floating points from the current frame, and the output memory usage is going to be just floating points considering a 1-rank evaluation is) floating points, which delivers an incredible economy of memory and processing. They called this mathematical operator IncSVD (Incremental Singular Value Decomposition). The IncSVD is what makes possible the IncPCP to exist and be faster than other algorithms. In section 4.3 we discuss more about IncSVD.

Since we will implement two versions of IncPCP, one in C++ and another in C++ with CUDA, we also need to develop an IncSVD with and without GPU support. To accomplish that we need an SVD and a GEMM (General Matrix Multiply) from BLAS (Basic Linear Algebra Subprograms) standard (Netlib, 2016). Since those functions are not part of CUDA module in OpenCV, this project is going to wrap those functions as OpenCV-like functions with input and output as OpenCV CUDA Matrices. Those functions are implemented in cuSOLVER and cuBLAS libraries respectively, which are low-level libraries. Summing up all the deliverables this project had produced that have an external relevant usage are:

- IncPCP
- CUDA IncPCP
- IncSVD
- CUDA IncSVD
- CUDA SVD (Wrapper)
- CUDA GEMM (Wrapper)

Rodriguez and Wohlberg MATLAB version outputs a matrix with floating points that indicates how much each pixel is likely to pertain to foreground. This project is going to introduce a threshold operator as Hu et al. (2015) have done to produce a binary mask as output and analyse the F-measurement (also called F1-score, is a metric that combines precision and recall) (Wikipedia, 2016a) comparing with a binary ground truth that are

included in dataset. To best fit the threshold, a subprogram is going to test all datasets varying threshold and keep the one with maximum F-measurement.

The accuracy and speed of Rodriguez's and Wohlberg's algorithm was shown by themselves and other research such as Hu et al. (2015). In Hu and colleagues comparison using nine datasets with five online methods and two offline methods, all of them based on PCP or the PCP itself, IncPCP was the faster, and being as much accurate as the original PCP and a less accurate in regard to other three methods.

This project is going to evaluate and compare the original version of IncPCP and our two versions regarding accuracy (F-measurement) and speed (FPS (Frames per Second)). For this purpose this project uses datasets in (Li et al. 2013) and (Unknown 1).

2. System Description

Since programming in CUDA directly without a CPU version is a venturesome approach, we had firstly developed an IncPCP using just C++ and OpenCV (Open Computer Vision) and deliver a C++ stable version without any optimization in GPU. We expect in that process that even this C++ only version is going to be faster than Rodriguez's and Wohlberg's (2016) MATLAB version. After this, CUDA optimization started to bebe included in the C++ code to allow the software to use the GPU as a coprocessor, becoming possibly faster than the previous version. CUDA is used in three ways, OpenCV functions in CUDA module, CUDA low-level functions in some library (CuSolver and cuBLAS) and CUDA kernels totally programmable in this project. This way, this project delivered two working versions of this Background Subtraction Algorithm, one in CPU and another in GPU using CUDA functions.

As internal components to produce IncPCP algorithm, we had produce the IncSVD mathematical operator introduced by Rodriguez and Wohlberg (2015a) and implemented by themselves in MATLAB. This module is a useful tool itself to many other applications. So as their module is a valuable tool to MATLAB environment, this implementation in pure C++ and C++ with CUDA functions is also relevant to those environments (C++ and CUDA) or even more useful if faster than their MATLAB version.

Moreover, since IncSVD needs to call an SVD as part of their operations, it is required an SVD version for CPU and GPU in C++. OpenCV includes a CPU version of SVD, but does not include this function in their CUDA module, so this project is going to wrap this feature directly from CuSolver. The same situation happens with GEMM (General Matrix Multiply) in BLAS standard (Netlib, 2016) NVidia, 2016b) and so this project produces its own wrappers from cuBLAS Library. Both wrappers are going to be OpenCV-like and it will be possible to work with OpenCV CUDA Matrices in a similar way that this library usually implements.

Summing up all the deliverables this project is going to produce that has some application outside the own project is:

- IncPCP
- CUDA IncPCP
- IncSVD
- CUDA IncSVD
- CUDA SVD
- CUDA GEMM

This project includes furthermore code to internal purposes, as small operations, testing, integration, controlling, timing and checking accuracy that is not something interesting outside this project. A short list of the main relevant internal modules are:

- Find Files – Find files and folders in the File System warping the internal Windows Functions
- Images From Folder
- Cuda IncPCP Functions
- Accumulators
- Tictoc
- Binary Confusion Matrix
- Tests
- Scripts
- Utils
- Cuda Utils

2.1.SVD

The SVD mathematical operator as defined in (2) and (3) has implementations in many languages, like C++, Java, MATLAB, Python and others. The OpenCV CPU and MATLAB operation have an economic size parameter, which if true returns a smaller output in some situations with the same accuracy as the full version, as can be seen in Equation (4).

Where for full-size SVD:

- is the complex or real matrix input,
- is an unitary matrix,
- is an diagonal matrix with non-negative real number on the diagonal, and
- is an unitary matrix, the conjugate transpose of .

This reduces memory usage and rather than the rank-r version does not change the resulting multiplication, just removing some leading columns from *U* and *V* that will never affect the

output since is a diagonal matrix, and so just the first *min (m,n)* columns from *w* have values different than zero. In the version we are going to use in C++ the output is not given as a matrix but as a *w* vector that is a column vector with the diagonal values from . MATLAB version always returns a dense matrix, even if just the diagonal has values different than zero.

A characteristic from SVD is that the eigenvalues are in the *w* diagonal in the descending order, so to produce a low-rank matrix we have to force to zero some of the last eigenvalues as Equation (3) in the partial SVD.

2.2. CUDA SVD

Since CUDA module in OpenCV does not have an SVD function, this project is going to provide a wrapping from cuSOLVER functions cusolverDnSgesvd and cusolverDnDgesvd, single precision and double precision versions respectively.
This module is a class called CudaSVD in `cuda_svd.h` and `cuda_svd.cpp` files. Moreover, it was implemented as a class also to manage the cuSolver handle and the buffers that need to operate the SVD.

cuSOLVER works as a column major library, expecting that a column is closer in memory than the rows, what is different from what is expected from a C++ program and also the OpenCV library that has CPU and GPU matrices in row major. This difference between who writes the matrix and this operator makes the appearance that this library reads the input as they are transposed and writes the output as transposed matrices as well. To keep this module compatible without the need of manually transposing input and output, in the wrapper the *U* and *V* outputs are passed to cuSolver functions swapped. Working as a transposed multiplication applied to SVD formula in Equation (2) lead to Equation (5). Because is a diagonal matrix and because cuSolver views the matrices as they were transposed we could conclude Equation (6).

= (5)

Where

(6)

Where is the column major version from some data or operation. = and both just renaming matrices in the internal operations.

2.3. GEMM Wrapper

CUDA module in OpenCV includes a GEMM function in its headers and documentation, but an OpenCV build version with cuBLAS is difficult to find, and due to the lack of time in this project, it was preferable to implement the wrapper rather than compiling the entire OpenCV.

This wrapper is also a class and not just a function to manipulate the lifetime of the cuBLAS handle.

cuBLAS, similarly to cuSolver, works in column major order and applies the same strategy. As happens with CUDA SVD module, we also called the internal functions with swapped operators aiming to work outside the wrapper as a row-major multiplication.

2.4. IncSVD

IncSVD functions that could be downloaded from Rodriguez (2015) were implemented in MATLAB in the "incSVD" subfolder. These operations are explained in Rodriguez and Wohlberg (2015) paper, which contains a full algebraic proof.

This module has three operations in three files, `rank1IncSVD.m`, `rank1DwnSVD.m` and `rand1RepSVD.m`. In our implementation, we kept these three operations as three methods from the IncSVD class in `inc_svd.h` and `inc_svd.cpp` files.

2.4.1. Rank1IncSVD

In MATLAB this function's signature is:

```
function[U S V] = rank1IncSVD(Uo, So, Vo, curFrame, flag)
```

Where Uo, So and Vo are the old SVD that will increase one column, curFrame is the new column and flag is a Boolean that indicates if the rank of the output SVD should be the same

as the input or higher. The return value is U, S, and V that correspond to the new SVD with one more column than the previous one.

Internally this function calls some matrix manipulators with Uo, So, Vo and curFrame as multiplication, sum and a smaller SVD with size just as $r \times r$ where r is the rank from the previous SVD (the rank is the size of the diagonal from So). This way, even if this SVD in Uo, So, Vo represents a large matrix, in case this is a low-rank matrix with a small r these operations are fast. Moreover, it always reuses data from the previous SVD to make the new one. As an example in the IncPCP algorithm, the fixed rank parameter is usually one, and just sometimes a small value like two or three. This method always returns a reduced in memory usage SVD.

In their implementation, the first column added in the SVD has an special treatment and it is done with the QR factorization.

2.4.2. Rank1DwnSVD

In MATLAB this function`s signature is:

```
function[U S V thresh] = rank1DwnSVD(Uo, So, Vo, k)
```

Where Uo, So, Vo are the previous SVD computed, k is the column this function is going to remove. U, S, and V are the new SVD without that column and thresh is a floating point that indicates the error introduced by removing the column. These functions in MATLAB never alters the rank.

2.4.3. Rank1RepSVD

This function`s signature in MATLAB is:

```
function[U S V ] = rank1RepSVD(Uo, So, Vo, k, curFrame, newFrame)
```

Where Uo, So, Vo are the old SVD already computed, k is the column that this function replaces. CurFrame is a not used parameter and remove will increase readability of this peace of code. The newFrame input is the column to be placed in the kth column. The return value is the new SVD.

2.4.4. IncSVD class

Our C++ implementation is a class and its most important methods are:

```
class IncSVD {
	//...
public:
	enum RankPolicy {
		increment_rank, adaptively_rank, keep_rank
	};

	IncSVD();
	~IncSVD();

	void Inc(const cv::Mat& current_column, RankPolicy rank_policy =
adaptively_rank, double threshold = 1);
	double Dwn(int col);
	void Rep(int col, const cv::Mat& new_column);

	cv::Mat& u();
	cv::Mat& w();
	cv::Mat& vt();
	//...

};
```

The SVD class from OpenCV keeps internally the three output values and they are slightly different from what MATLAB returns, *u* (same as U in MATLAB), *w* (just the diagonal from S) and *vt* (transposed V). This way, to reconstruct the original matrix, firstly a proper size *s* matrix should be allocated and put *w* as the diagonal matrix and then $u \times s \times vt$ multiplied . This IncSVD follows the same approach, with these three variables as internal elements. The methods *u*, *w*, *vt* access those internal values. Each one of these operations Inc, Dwn and Rep changes the internal SVD (represented by those 3 OpenCV matrices).

The Inc method works as the rank1IncSVD and equally has the option of always increase (increment_rank), always keep the same rank as it is (keep_rank) and an additional behaviour made called adaptively_rank. When the input has the adaptively flag the function just

increases if the rank has a new singular value that represents perceptually in the final matrix at least threshold. The contribution value is the ratio between new eigenvalue and the sum of all old eigenvalues. So it became possible to use this class as incremented one column in the internal SVD but also to increase rank if more than 1% of magnitude is introduced for this new value, or any other percentage.

The Dwn method does the same as rank1DwnSVD, and just has one col parameter since Uo, So and Vo are internal. This method has a few differences in comparison to the MATLAB implementation, which could reduce the rank in this operation. This only happens in a specific context when the removed column changes the value of (the same as to less than the current rank. IncPCP usage in MATLAB never reduces the rank since this function are in a less general use, where the rank is always one or a small value, as two or three. The rank1RepSVD done by authors from IncPCP method similarly to the previous function. The only difference is the absence of the "curFrame" (current frame) parameter because even in MATLAB this parameter appears in the signature but it is not used any place inside the function.

One final small difference is the MATLAB version initialization of the IncSVD with a QR factorization making it faster and could be used in the particular case of a single column. Since OpenCV does not have this mathematical operator and because of the lack of time in the project I did not install another library to use this operator. I have used a reduced in memory usage SVD version, which produces the same result, but slower. This does not compromise so much our implementation since this just runs once at each video in IncPCP, but a future work is to add QR factorization to it.

2.5. CUDA IncSVD

Our implementation comprises the class CudaIncSVD in the files `cuda_inc_svd.h` and `cuda_inc_svd.cpp`. The most relevant methods for this class are shown as follows:

```
class CudaIncSVD {
	//...
public:
	enum RankPolicy {
		increment_rank, adaptively_rank, keep_rank
	};

	CudaIncSVD() {
	}
	~CudaIncSVD() {
	}

	void Inc(const cv::cuda::GpuMat& current_column, RankPolicy rank_policy =
adaptively_rank, double threshold = 1);
	double Dwn(int col);
	void Rep(int col, const cv::cuda::GpuMat& new_column);

	cv::cuda::GpuMat& u();
	cv::cuda::GpuMat& w();
	cv::cuda::GpuMat& vt();
	//...

};
```

It shows a similar behaviour compared to IncSVD. Internally, all functions have the same operations, but in GPU, some modifications are needed. This class has also used the CUDA module in OpenCV, the CUDA SVD Wrapper, and CUDA GEMM Wrapper.

A specific behaviour of this class is that the initialization is done in CPU in the common SVD class in OpenCV. This is done since CUDA SVD Wrapper does not compute a simplified operator, and since the first column to be added is probably very long and calculating a full size is a non optimal solution, the initial data is downloaded to CPU again, computed in CPU and then uploaded back to GPU (all three matrices). CudaIncPCP does all other operations in GPU, and since initialization is done just once in an IncPCP video, this does not affect so much our timing. But as the IncSVD it is a future work introduce QR factorization in this operator, and cuBLAS library has this operator.

2.6. IncPCP and CUDA IncPCP

This is the highest hierarchy module in the project hierarchy that can be found in `inc_pcp.h` and `inc_pcp.cpp` as a class called IncPCP. This class includes the CUDA

and non-CUDA versions of the algorithm, and a single flag decides which implementation runs.

The original MATLAB version had been downloaded from the professional web page from one of the creators at the beginning of this project at October 2015 (Rodriguez 2015) and is a function with the following signature:

```
function[D, L, S, stats] = incrementalPCP(basedir, rank, innerLoops,
winFrames, myFlags)
```

Our proposed version comprises a class with its main functions listed as follows, with basically one constructor, one destructor and the main method that is similar to incrementalPCP in MATLAB.

```
class IncPCP : public IncPCP_Flags {
//...
public:
        typedef IncPCP_Flags Flags;
        struct Stats {
                //... optional output
        };

        IncPCP(Preset preset = _default);
        ~IncPCP();
        void operator()(const std::string& input_directory,
                const std::string& output_directory = std::string(),
                unsigned rank = 1,
                unsigned inner_loops = 3,
                unsigned win_frames = 30,
                Stats * stats = nullptr);
//...
};
```

In both versions we have a path to the input (“basedir” or “input_directory”), a rank that is the low-rank background model, the number of inner loops that the algorithm tries to converge the foreground and the windows frames (winFrames, or the memory parameter), the number of frames to be kept in background model, and both of them have a structure with flags input (in C++ as an Inherence from IncPCP_Flags, and in MATLAB by parameter).

Since this algorithm has several configurations and different configurations to run to we are going to firstly discuss the flags from the original version in comparison to the proposed version in 2.7.1 and 2.72 sections.

2.6.1. Flags in Original Version

The “myFlags” parameter should be initialized with one of four pre-set configurations provided by the incAMFastPCPinputPars function; they are “default”, “default_cuda”, “ti_standard”, “ti_search” and “none”. The “ti_standard” and “ti_search” versions are implementation of an algorithm called IncAMFastPCP that is translational and rotational jitter (Rodriguez, P., Wohlberg 2015 2) and is not the focus of this project to reproduce this part of the algorithm. The “default_cuda” is the same as “default” but uses MATLAB CUDA toolkit to optimize some functions, and also was developed by the authors. The “none” pre-set permits a fully manual configuration. In each of the configurations, values can be changed by the user.

Observing the "default" configuration output below, we could understand what configurations are available using this algorithm.

```
         showFlag: 1
         grayFlag: 0
         saveFlag: 0
           lambda: 0.0375
         fullInit: 0
         stepInit: 5
         sparseGT: []
               L0: []
               S0: []
 backgroundThresh: 10
 backgroundStable: 10
              url: 0
        frameNoff: 0
               TI: 0
             TImu: []
       TIcsiLoops: []
    TIadjustDiff: []
        baseAlpha: []
         baseTras: []
      alphaThresh: []
         cudaFlag: 0
          vecFlag: 1
        adaptShow: 1
```

- `showFlag`: a Boolean value that controls if output is shown or not. The images shown are three, input, foreground and background.
- `grayFlag`: a Boolean value that if true reads the images in grayscale, if not reads as colourful.
- `saveFlag`: a Boolean value that if true saves the same output from `showFlag` in the file system, but the foreground is stored twice as image and as a matrix.
- `Lambda`: An internal value to algorithm behaviour, this will be explained after.
- `fullInit`: a Boolean value that if true initializes the algorithm completely before finding a foreground, what means that the first "winFrames" will not produce any foreground. The rank should only be greater than one if fullInit is true.
- `stepInit`: if fullInit is true they ignore stepInit – 1 images between each image. For example, with stepInit = 5, and winFrames = 30, the algorithm is going to initialize with the images 1,6,11,16,…151. Otherwise, if fullInit is false it does not affect the program.
- `sparseGT`: Sparse Ground Truth, in floating points, was used by the authors to show that IncPCP is closed to PCP itself, do not is relevant for this project.
- L0 and S0: Optional initial foreground and background.
- `backgroundThresh`: Threshold floating point that indicates if two follow foregrounds are so different that could be considered unstable.
- `backgroundStable`: Integer number of frames that the algorithm expects the foreground to be stable.
- `url`: if true indicated that baseDir is a URL path and not a folder with images.
- `frameNoff`: an Integer offset to ground truth in the authors experiments, and is not so relevant in this project since is used to prove equivalency between IncPCP and PCP what we already consider truth.
- From `TI` until `alphaThresh` are parameters related to TI versions that are not the focus of this project.
- `cudaFlag`: if true runs in CUDA MATLAB toolkit.

- `vecFlag`: If true runs each frame as a vector in the background model, this model just focus on this flag true all the time.
- `adaptShow`: If true the shown images are normalized based on all the past, what could be more visualise understandable, if not they are normalized with fixed values.

2.6.2. Flags in this Project

The flags in this project can be found in inc_pcp_flags.h and inc_pcp_flags.cpp files in the IncPCP_Flags structure. These files have mainly two sections: the first one is the original flags version from `imshow` to `adapt_imshow`. Some of the parameters were not translated to C++ and are commented in the code snippet above representing not implemented features from the MATLAB version. Two flags from the original version have been slightly modified: `imshow` and `imsave`. Moreover, other new flags were introduced in this project, from `cv_bit_depth` to `fprint_threshold_vs_measuarement`, to provide new features. The new features introduced are controlling the floating-point precision of operations and the binary foreground production and measurement.

```
struct IncPCP_Flags {
	//Original flags from Rodriguez's:
	int imshow;
	bool gray;
	int imsave;
	double lambda;
	bool full_init;
	unsigned step_init;
	//sparseGT:[]
	//L0 : []
	//S0 : []
	double background_thresh;
	unsigned background_stable;
	//u-1: 0
	//frameNoff: 0
	//TI: 0
	//TImu: []
	//TIcsrLoops: []
	//TIadjustDiff: []
	//baseAlpha: []
	//baseTras: []
	//alphaThresh: []
	bool cuda;
	//vecFlag : []
	bool adapt_imshow;

	//My Aditional Flags:

	//Floating Precision:
	int cv_bit_depth; // CV_64F (double precision) or CV_32F (single precision)

	//Binary Foreground:
	bool make_binary_foreground;
	double threshold_foreground;
	bool auto_threshold_foreground;
	bool analize_every_ground_truth; //DEBUG
	std::string directory_ground_truth;
	bool compute_f_mesuarement;
	bool fprint_threshold_vs_mesuarement;//DEBUG
};
```

The following list just comments the flags that had a different behaviour or are introduced in this project, so explaining the introduced new features that this project has when comparing with MATLAB one:

- `imshow`: It is quiet similar to `showFlag` but is not a Boolean, but an integer where each bit means a different image, and so allows us to control exactly what images to show. Rather than three images in the original version this version hade 5 possible outputs in this mode.
- `imsave`: Similar to saveFlag and equally to imshow is not Boolean, but an integer with each bit control a particular image. The images that could be saved are not exactly the same then shown ones. Rather than four images the original version have, this version have six different imagens and eight possible controls, two extra controls allows just to save a picture in some context, as an example, `foreground_mat_when_has_ground_truth,` that as the

name indicates just save the foreground as matrix (not as a picture) when this frame has a ground truth.

- `cv_bit_depth`: Controls with single or double precision is going to be used, what is a new feature from this version.
- `make_binary_foreground`: A Boolean that if true produces the binary foreground applying a threshold.
- `threshold_foreground`: The threshold manually given to produces the binary foreground.
- `auto_threshold_foreground`: If true find automatically a foreground based on the first frame find in ground truth, choose a threshold to maximize the F-measurement.
- `analize_every_ground_truth`: If true and `auto_threshold_foreground` is also true automatically adjust the F-measurement for each ground truth found.
- `std::string directory_ground_truth`: Possible path to ground truth.
- `compute_f_mesuarement`: Computa, the F-measuarement for each ground truth found, allowing the user to now the precision of the system while running.
- `fprint_threshold_vs_mesuarement`: A Boolean that if true when trying to find the best threshold to optimize the F-measurement (just if `auto_threshold_foreground` is true) print the threshold values and correspond F-measurement in a file. Could be used for debugging or understanding the relation between threshold and F-measurement.

And one more comment about the features that this project has and MATLAB does not is the `fullInit` flag. Even existing in both of them the MATLAB version fails if this flag is true, what is probably a software degeneration between versions that introduces some small mistakes in this feature. However, since this code is still understandable, we had implemented it in the C++ and C++ with CUDA versions.

2.6.3. Algorithm

The aim of this algorithm is the optimization of the functions in Equation (1). The solution is to optimize the problem with two parameters. One numeric solution is in permutating the optimized equations (5) and (6) in loop, for innerLoops (parameter for the algorithm) times.

(5)

(6)

Where L is the current background estimation, S is the current foreground estimation and D is the current frame as a column.

The first step of the algorithm is to read the frames in order, reshaping each one to become a column matrix with size m × 1 where (number of rows, columns and depth or channels, respectively).

If full initialization is active, the algorithm reads winFrames jumping stepInit between each frame until completely winFrames. And so for each frame read the IncSVD function is called, if the current rank is smaller than the parameter rank they increase the rank in this operations if not they keep the rank as it is. At the end of this initialization, each element from V matrix receives the average from its column, so mixing every column to become its own mean, as shown in Equation (7).

(7)

Where m is the number of columns.

Otherwise, if full initialization is not active, the algorithm just initializes with an SVD from the first frame as a column, or in MATLAB version they initialize with an optimization version that produces the same result. Computing a QR-decomposition as in Equation (8) and seting U as Q, S as R and V as a 1×1 matrix with just a number one. This is possible because an orthogonal matrix is the real equivalent of a unitary matrix in complex numbers (Wikipedia 2016b) and an upper triangular matrix with size 1×1 is also a diagonal matrix and in this case this fits the SVD requirement in Equation (2).

(8)

After initialization, we are always going to have three matrices representing the background, they are, where m is the number of values in a frame, is the fixed rank adopted and n is the number of frames inserted. If full initialization is done, n is equal to winFrames; if not, n is equal to one. The background is descripted by Equation (9) and the current background (what means just the last frame in the background) is described as Equation (10) and is called L.

Σ (9)

Σ (10)

Now, for each new frame, the algorithm calls first IncSVD with the current frame as a column and does not increase the rank.

After that for innerLoops times, the algorithm tries to optimize Equations (6) and (7) estimating L and S in the permutation. So in each iterator first estimate L (current background) using (10). After the shrink operator is as in Equation (11) and   is a parameter in Flags. After estimate L the algorithm uses RepSVD to replace the rank-th column with D – S, and so repeat another iteration.

(11)

Where A is a matrix and is a double. Sing is the operator that returns minus one, zero or one depending on the signal. The used functions are all of them element wise.

After innerLoops iterations, the algorithm checks the stability of the background calculating the average from the absolute difference from this L and the L from the last frame and setting this value as local distance. So current local distance divided by the previous local distance is set to Lfrac. After `backgroundStable` frames if the Lfrac is greater than `backgroundStable` the background is reinitialized with just the current frame.

Finally, for this frame if the number of columns in the background (n or number of rows in V) is greater than or equal to winFrames the last frame is removed from the background with the DwnSVD function removing the oldest frame. And so repeat the procedure after initialization.

2.7.Internal Modules

2.7.1. Find Files

Using Windows API from file system gives some facilities to looking for files in a folder recursively or not with some filters.

2.7.2. Images From Folder

Using the previous module provides an interactive way to look just in pictures in a folder.

2.7.3. Tictoc

Provides the same interface as *tic* and *toc* MATLAB functions. Warping some Windows timing functions they calculate the timing from a tic and a toc with milliseconds of precision t*toc*.

2.7.4. Binary Confusion Matrix

Analyse the output of an expected result (Ground Truth) and the algorithm result and given true positive, true negative, false positive and false negative. Based on these four numbers also provide recall, specificity, and f-measurement. Can be found in `binary_confusion_matrix.h` and `binary_confusion_matrix.cpp`.

2.7.5. Tests

In `tests.h` and `tests.cpp` is a set of developing tests. Usually, they test a function or class isolated and show the result or if they pass the test. Usually black box testing, some random input in the module and after comparing the output if the expected one. For CUDA functions often computing in CPU the same computations and check if both are equal. They also have timing testing of some modules.

2.7.6. Scripts

Scripts module is present in `scripts.h` and `scripts.cpp.` Those functions that act as scripts with a configuration to run the IncPCP or other feature from this project with a real input, as a dataset, or a set of inputs, as a folder full of the dataset. Could be used for testing but at a high level and does not test module by module but an entire feature with a real input.

3. System Implementation

All this project including the system implementation, research papers, literature review and presentation has been made available in a git repository that can be found in https://bitbucket.org/Djeefther/el600. The following structure is used:

- CudaProject – The project itself
  - CudaProject
    - hdr – C++ Headers
    - src – C++ Source
- incPCP – Rodriguez`s and Wohlberg`s implementation in MATLAB, additionally some testing files made by this project
  - incSVD – Incremental SVD functions in MATLAB
- Results – Experimental results
- Other folders

In the current version the code has more than 9000 lines summing up .cpp, .h .cu and .cuh only.

The development environment chosen is a computer with Windows 7 or higher, Visual Studio 2013, CUDA 7.5 Toolkit and OpenCV 3.0 installed. The Visual Studio 2013 is the newest version compatible with CUDA, since CUDA Toolkit has a compiler inside that needs to be compatible with the IDE. The machine to run the project should have a CPU middle-end or above (as an i5-3470) and an NVIDIA GPU with CUDA capability 2.0 or greater and with fair or good computation power as a GTX 690.

The OpenCV download was a build by third party with CUDA 7.0 at (Saharsh, B 2015).

OpenCV was chosen as library because it allows to open, show and save images, process numeric computations and to easily manipulate matrices in C++ with a good performance. OpenCV was firstly made in C++ to CPU only applications but since the 2.0 version includes some CUDA wrappings from CUDA libraries (as cuBLAS, cuSPARSE, and CuSolver) that allow an easy way to use that CUDA functions without low-level functions, memory allocations, and deallocations. This set of features is called as "CUDA" in 3.0 version and

inside the code is the "opencv::cuda" namespace. This module provides several operations that a CPU matrix in OpenCV (called just as Mat) has to their GPU matrix (called GpuMat) with the similar or identical interface. Sometimes these operations are just CUDA already developed libraries, sometimes as OpenCV own CUDA kernels implementations. However, some operations that this project needs are implemented in CUDA libraries but are not in CUDA OpenCV module. This is the case of SVD and GEMM (are part of the library but not build in my version) operation and because of that we developed our wrappers.

4. Results and Discussion

4.1.Method for computational performance tests

All the timing tests were done using the fastest versions of MATLAB, C++ and CUDA codes, running as release and if a C++ code detached from Visual Studio. Show or save flags were deactivated when they existed. The target computer was used only for the algorithms being tested. In MATLAB the time was computed using the tic and toc functions and in C++ our Tictoc class with similar behaviour was used. When comparing GPU and CPU, when possible, speedup with and without the upload and download time are given. In timing tests in GPU CudaDeviceSynchronize (NVidia) is always called after the operation and before the toc command to ensure the timing is correct. All timing tests were done at least three times to check if they are consistent values. In almost all operations CPU and GPU versions usually have initializing time, so when this was noticed before the real timing test the same function was called without timing to initialize possible internal handles. In GPU usually this handles are explicit and also this timing is taking as outside comparison. Speedup greater than one means GPU is faster than CPU or CPU is faster than MATLAB versions. The speedup comparison with upload and download timing, is not always a fair comparison because it is not always relevant since in some context it is not necessary one upload and one download for each operations, rather are done many followed operations for one upload and one download.

The computer used for the results computation is a desktop with Windows 10 64 bits with an i7-processor @ 3.50 GHz with 4 cores and 8 threads (by Hyper-Threading), 64.0 GB of memory and three GTX 780 Ti graphic card (what by design our CUDA code just uses one graphic card as accelerator).

4.2.Method for numerical precision tests

Numeric precision comparison of a matrix is usually done using the compare2mat function available in the utils module. This function is configurable to show different comparisons

from Absolute Difference from two matrices, Absolute Relative Difference from two matrices and Square Difference from two matrices. For each one of these evaluations are shown maximum, minimum and average values. Sometimes Relative Difference is cruel when the expected value is so close to zero, but is useful with big values that a difference of some units means perceptually almost nothing.

4.3. CUDA GEMM

In the tests module in `test.h` and `tests.cpp` files there is a function to

```
void test_CudaGemm(int size_a = 1000, int size_b = 1000, int size_c = 1000);
```

This function tests the CudaGemm multiplying a matrix with size_a × size_b with a matrix with size_b × size_c, both randomly initialized with values from 0 to 1. This function supports test using single and double precision. In GPU the same operation is called with GEMM from OpenCV in CPU and two outputs are comparable using a `compare2Mat` function developed in this project. After operations in GPU and before toc command this function calls CudaDeviceSynchronize to ensure that the timing is correct.

The output test is shown in Figure 3, that firstly shows the Absolute and Relative Difference from CPU and GPU, and they are small enough to be considered numerically equal. After that there are two timings about initialization in CPU and GPU and after the timing of each one follows the speed up greater than one meaning the GPU version is faster. speed_up_total_time considers the upload and download timing.

GEMM is an operator that allows us to transpose the inputs, so manually this function was tested varying the four combination from transposed and non-transposed in both input no timing or output change had been notice.

```
Absolute Difference:
max = 1.7053e-013
min = 0
avg = 1.32995e-014

Relative Difference:
max = 6.57661e-016
min = 0
avg = 5.31599e-017

first cpu gemm time = 0.638
create_cuda_gemm_time = 0.252

cpu_gemm_time = 0.483
upload_time = 0.005
gpu_gemm_time = 0.019
download_time = 0.003
gpu_total_time = 0.027

speed_up = 25.4211
speed up total time = 17.8889
```

(a)

```
Absolute Difference:
max = 0.000579831
min = 0
avg = [illegible]

Relative Difference:
max = 2.23037e-006
min = 0
avg = 3.34999e-007

first_cpu_gemm_time = 0.987
create_cuda_gemm_time = 0.232

cpu_gemm_time = 0.836
upload time = 0.006
gpu gemm time = 0.002
download time = 0.001
gpu_total_time = 0.009

speed_up = 418
speed_up_total_time = 92.8889
```

(b)

Figure 3: CudaGemm test with double precision (a) and single precision (b) both of them with matrix sizes equal to 1000 in all three parameters.

4.4. SVD

This test is a simple testing using SVD already developed in OpenCV, the function `test_cv_svd` in tests module. This function initializes a matrix with M × N call SVD and reconstruct the original input multiplying all the outputs and compare the two matrix with compare2mat function. This functions was used just to understanding the usage of this operator and was not really needed since was an already approval function from OpenCV committee.

After the initial output with a full size SVD result be printed a for starts that after each enter the program reduce the rank of the output and shows the new comparison and eigenvalues.

The input choose used was various, from random values to some matrix depending on index formulas as (11). In the case of random input to a good reconstruct the rank should min(N,M)

and so just a full SVD good representing the input. To some index dependency formulas as low-rank matrix good represent the input, since the input is full of redundancy that random values do not have. As an example, rank equal to 3 yet represent with good precision (bellow absolute error) the input with size equal to 10 × 11.

, where A is the matrix input in the SVD (11)

4.5. Datasets

This project comparison between the original version of IncPCP and ours two versions about accuracy (F-measurement) and speed (FPS (Frames per Second)). To provide that a right input is not random values or formula based, but real datasets videos. For this purpose this project use the same datasets as (Hu et al. 2015) and (Rodriguez and Wohlberg 2015) used, they are (Li et. al. 2013) and (Unknown 1) respectively. The first one used nine datasets all of them with binary mask ground truth and resolution small or medium size between (160 × 120) and (320 × 256). The second paper we found three datasets without ground truth but with median to high between (320 × 256) and (1920 × 1088). This two packages of datasets share one dataset, and another one was not used in this test so become ten datasets, in each we have 8 of them with ground truth.

4.5.1. Bootstrap

120 × 160 images with 3057 frames, crowd scene with people walking and or stopped. Contains ground truth. In figure 7 some examples frames.

Figure 7: Frames from Bootstrap

4.5.2. Campus

128 × 160 images with 439 frames, waving trees. Contains ground truth

.

4.5.3. Curtain

128 × 160 images with 523 frames, fountain water, after sometime two persons come in front of it. Contains ground truth. In figure 8 some examples frames.

Figure 8: Frames from Curtain

4.5.4. Escalator

128 × 160 images with 3417 frames, moving escalator with some periods with many people in and others not. Contains ground truth.

4.5.5. Fountain

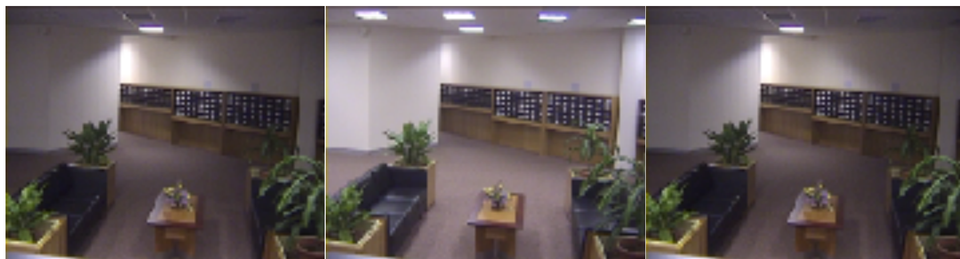
128 × 160 images with 523 frames. Fountain water. Contains ground truth.

4.5.6. Hall

144 × 176 images with 1546 frames. Crowd scene. Contains ground truth.

4.5.7. Lobby

128 × 160 images with 1546 frames. Few people are walking and after switching light.



4.5.8. Shopping Mall

256 × 320 frames with 1286 frames, crownd scene.

4.5.9. WaterSurface

160x128 images with 633 frames of water and a person walking towards.

4.5.10.Lank3

640 × 480 images with 400 images. Superior visualization from a large street with some cars. Do not contains ground truth.

4.5.11.Neovision3

1920 × 1088 images from 900 items. Superior or view of one square with a fountain inside. The street the around the plaza have people and biking walking. Do not contains ground truth.

4.6.IncPCP qualitative view

It is possible to view an output from our implementation in Figure 9 by the single precision C++ version, however all our three versions in C++ and the MATLAB version will return an indistinguible visual result.

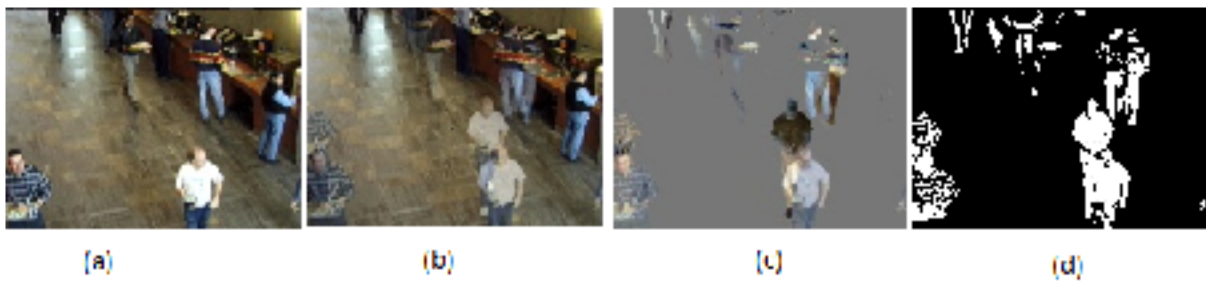


Figure 9: IncPCP bootstrap input and outputs. (a) the input, (b) the background, (c) the foreground (d) binary foreground (after threshold)

4.7.CUDA SVD

In tests module it is possible to find `test_CudaSVD and test_CudaSVD_timing,` testing the wrapper of the CUDA SVD function. The first function is similar to test_cv_svd and just outputs that the reconstructed input (by multiplying the outputs) and the numeric comparison with compare2mat function with a very similar result.

The second test shows the timing difference and as well reconstructing error difference between CPU and GPU in Figure 4.

```
CPU Reconstruct Error:
Absolute:
max   [illegible]
min = 0
avg = 1.06292e-009

Relative:
max = 0.0115330
min = 0
avg = 1.15340e 000
```

```
GPU Reconstruct Error:
Absolute:
max   [illegible]
min = 0
avg = 2.37366e-009

Relative:
max = 0.0023225
min = 0
avg = 6.23225e 000
```

```
Timing
time_gpu_alloc    0.004
time_gpu_init     28.768

time_gpu_svd_init = 1.73
time_gpu_alloc = 0.002
time_upload = 0.004
time_gpu_svd = 0.216
time download  = 0.011
time gpu total  = 0.231

time cpu svd init = 0
time_cpu_svd = 15.416

speedup   [illegible]
speedup total = 2.2374
```

Figure 4: Precision and time comparison between SVD and Cuda SVD, all times are in seconds, the input size was 1000 × 1000.

Varying the size of this test and also produce some MATLAB tests to the SVD function we produced the Figure 5 about timing in C++, CUDA and MATLAB comparison.

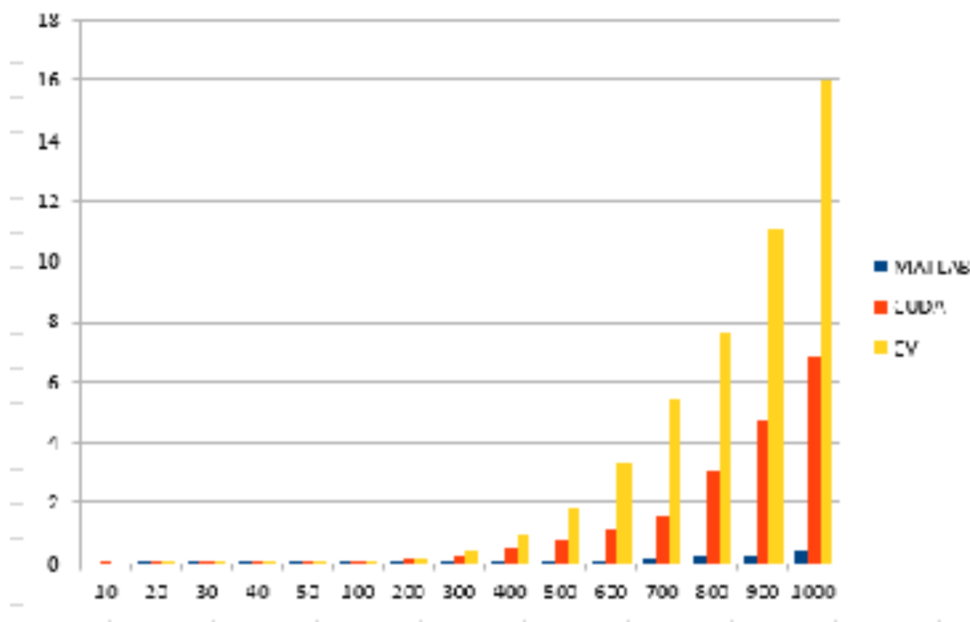


Figure 5: Timing tests of Full SVD function in MATLAB, C++ and CUDA. Smaller is best. The X-axis means the size from a square input, and the y-axis the time in seconds.

As expected CUDA version is faster than C++ version, but surprising MATLAB function is much faster, what contradicts the common timing comparison between MATLAB and C++. The algorithm used in SVD from MATLAB is proprietary and not easy to guess why is faster

than C++ and GPU, but is visible in analyses that they have a concededly greater numeric error, what possible reflects in our result in section of accuracy of our algorithm be 17% greater. Probably MATLAB algorithm approximate the factorization less iterations than other versions.

4.8. IncSVD

To test this class three test functions were introduced: `test_incremental_svd_inc_svd`, `test_incremental_svd_dwn_svd` and `test_incremental_svd_rep_svd`. All of them are available in tests.h and tests.cpp files and possible to test with double and single precision. Each one of them testing one of the methods from IncSVD class, showing the reconstruction error for the iterative version (IncSVD) and batch version (SVD) for the same operation.

The first test increments successively columns (random or not initialized) using Inc method from IncSVD and concatenates them in another matrix that in the end runs a batch SVD version after comparing both of them. One example of double precision with each column size equal to 10 and 10 colomns added using Equation in (11) in Table 1 to single precision test and Table 2 to double precision in the lines called “Inc”.

The second test initializes a matrix with an initial value and randomly starts to remove a column from this matrix and uses the Dwn method. For an initial matrix with 10×10 initialized with Equation (11) we have a result after 5 removed columns with double precision in Table 1 and single precision in Table 2, both in “Dwn” lines.

The third test to replace columns works as the test before initialized with an initial size and randomly choose one column to replace. For a 10×10 initial matrix initialized with random values after 5 replaces we have a result in Table 1 to double precision and Table 2 to single precision in “Rep” lines.

And so we could see that

Table 1: Iterative and Batch Error with double precision

Double precision

	Iterative Reconstruct Error					
	Absolute			Relative		
	max	min	avg	max	min	avg
Inc	2.021E-11	0	2.131E-12	1.776E-07	0	1.776E-09
Dwn	5.68E-13	0	1.44E-13	2.27E-06	0	4.54E-08
Rep	2.70E-13	0	8.81E-14	3.36E-14	0	3.09E-15
	Batch Reconstruct Error					
	Absolute			**Relative**		
	max	**min**	**avg**	**max**	**min**	**avg**
Inc	1.14E-13	0	1.85E-14	3.85E-07	0	1.85E-14
Dwn	8.53E-14	0	2.15E-14	1.26E-07	0	2.53E-09
Rep	1.14E-13	0	2.88E-14	7.57E-14	0	1.83E-15

Table 2 Iterative and Batch Error with single precision

Single Precision						
	Iterative Reconstruct Error					
	Absolute			**Relative**		
	max	**min**	**avg**	**max**	**min**	**avg**
Inc	1.526E-04	0	3.305E-05	6.414E+00	0	6.414E-02
Dwn	5.34E-05	0	1.36E-05	1.58E+00	0	3.16E-02
Rep	1.06E-04	0	3.05E-05	1.06E-04	0	2.35E-06
	Batch Reconstruct Error					
	Absolute			**Relative**		
	max	**min**	**avg**	**max**	**min**	**avg**
Inc	6.10E-05	0	1.14E-05	2.65E+00	0	2.65E-02
Dwn	3.81E-05	0	9.31E-06	1.54E+00	0	3.09E-02
Rep	4.58E-05	0	1.10E-05	4.53E-06	0	4.62E-07

4.9.CUDA IncSVD

The CUDA version has three similar tests with the same parameters and conditions, `test_incremental_svd_cuda_inc_svd`, `test_cuda_incremental_svd_rep_svd` and `test_cuda_incremental_svd_rep_svd`. The batch version compared is the same as IncSVD, the CPU one from OpenCV. In similar configurations about initialization and sizes could be found in Table 3 and 4 for double and single precision.

Table 3: Iterative and Batch Error with double precision from CudaIncSVD

Double precision						
	Iterative Reconstruct Error					
	Absolute			Relative		
	max	min	avg	max	min	avg
Inc	4.345E-11	0	4.581E-12	1.776E-07	0	1.776E-09
Dwn	1.330E-12	0	4.318E-13	2.27E-06	0	4.54E-08
Rep	2.970E-12	0	9.692E-13	3.36E-14	0	3.09E-15
	Batch Reconstruct Error					
	Absolute			Relative		
	max	min	avg	max	min	avg
Inc	4.661E-12	0	7.600E-13	3.855E-07	0	1.854E-14
Dwn	1.833E-13	0	9.878E-13	1.26E-07	0	2.53E-09
Rep	5.912E-12	0	1.499E-12	7.57E-14	0	1.83E-15

Table 4: Iterative and Batch Error with single precision from CudaIncSVD

Single Precision							
	Iterative Reconstruct Error						
	Absolute			Relative			
	max	min	avg	max	min	avg
Inc	3.281E-04	0	7.107E-05	6.414E+00	0	6.414E-02
Dwn	1.148E-04	0	2.915E-05	1.58E+00	0	3.16E-02
Rep	2.236E-04	0	6.405E-05	1.06E-04	0	2.35E-06
	Batch Reconstruct Error					
	Absolute			Relative		
	max	min	avg	max	min	avg
Inc	1.312E-04	0	2.442E-05	2.653E+00	0	2.653E-02
Dwn	1.183E-03	0	2.886E-04	1.54E+00	0	3.09E-02
Rep	2.289E-04	0	5.722E-05	4.53E-06	0	4.62E-07

4.10. IncSVD Time comparison

The `test_timing_cuda_and_nor_cuda_inc_svd` tests a timing comparison between them. The parameters are the number of rows in the IncSVD simulation, the number of columns inserted and number of columns removed, after the same number of columns inserted are going to be removed. To a simulation with 640 × 480×3 (VGA colourful) 30 columns be inserted and 30 columns replaced an output from this tests as Figure 6 is found in the end and double precision.

```
time_inc = 0.2372
time_rep = 0.563033
time_dwn = 0.168033

time_cuda_inc = 0.202333
time_cuda_rep = 0.160233
time_cuda_dwn = 0.0221333

speedup_inc = 1.17232
speedup_rep = 3.51383
speedup_dwn = 7.59187
speedup_one_of_each = 2.51694

speedup_inc_t = 1.08774
speedup_rep_t = 3.19966
speedup_dwn_t = 4.4375
speedup_one_of_each_t = 2.41805
```

Figure 6: Timing comparison with IncSVD and CUDAIncSVD

4.11. IncPCP Threshold and F-measurement analyses

All tests about accuracy and timing are going done with default configuration from MATLAB implementation and the corresponding parameters in C++.

The subprogram `find_best_threshold` open an output directory for any of the 3 implementations and analyse the best threshold for each data set and for all of them individually optimize the best F-measurement. This program looks for best threshold with two criteria analyse the first ground truth of each data set and the average of all ground truth.

Using this program from the output of IncPCP MATLAB, IncPCP with single and double precision and CudaIncPCP with double precision.

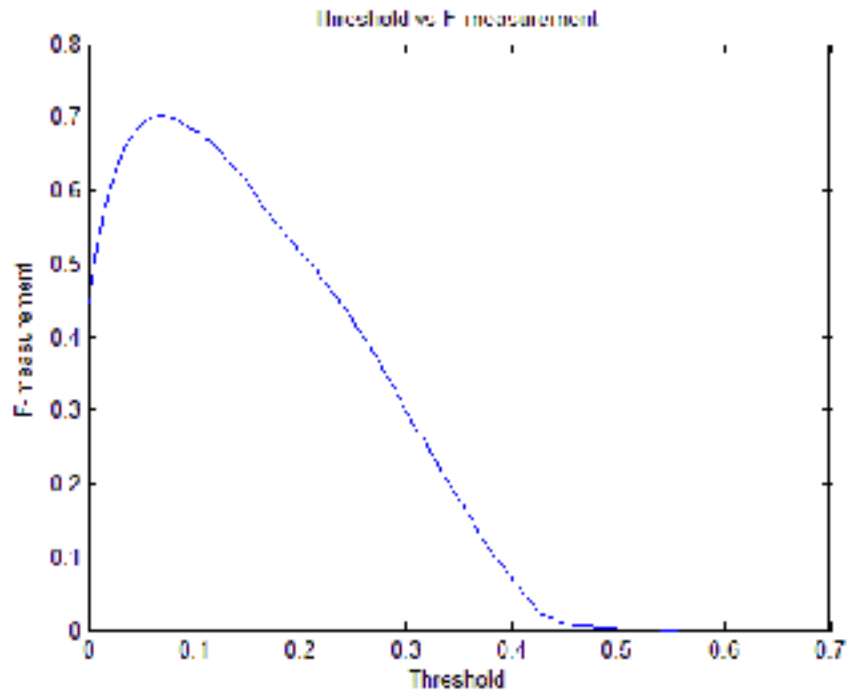


Figure 10: Threshold vs F-measurement for Shopping Mall data set frame number 433

Table 5: IncPCP C++ double Precision threshold and F-measurement for all datasets

Code		C++ (double precision)			
Threshold Description		Best avg All Frames		Best First Frame	
Threshold		0.075		0.078	
Avaliation		Avg All Frames	First Frame	Avg All Frames	First Frame
Bootstrap	160x120	58.94%	72.00%	59.09%	71.71%
Campus	160x128	27.19%	30.06%	27.86%	30.77%
Curtain	160x128	51.11%	85.46%	50.40%	84.98%
Escalator	160x130	37.12%	47.27%	37.66%	47.46%
Fountain	160x128	63.02%	56.63%	63.02%	56.87%
hall	176x144	46.39%	74.92%	46.53%	75.89%
Lobby	160x128	47.46%	70.45%	46.59%	69.43%
ShoppingMall	320x256	70.71%	68.80%	70.49%	68.79%
lank3-rgb	640x480				
neovision3	1920x1088				
Mean		**50.24%**	**63.20%**	**50.21%**	**63.24%**

One example of thresholding optimize from just one frame is given Figure 7, as is possible to see varying threshold initially all pixels are considering true and after becoming more restrict the F-measurement increase until a maximum value and become to decrease. This subprogram find the best F-measurement not for just one frame, but for all data sets. For C++ double precision version the Table 5.

Where in the columns bellow Best Avg All Frames we see the number 0.075 the best threshold to fit the average for all ground truth in all datasets, bellow that two sub columns the evaluators All ground truth and just the first. In the other field Best First Frame is the value of threshold 0.078 to fit the average of all first ground truth. The blanket columns are from datasets without a ground truth. The percentage numbers are F-measurement.

The Table 6 shows the same data from IncPCP with single Precision. Looks like identical values but analysing them more precision in between 6th and 8th decimal case they have minor difference about F-measurmen.

Table 6 : IncPCP C++ single Precision threshold and F-measurement for all datasets

Code		C++ (single precision)			
Threshold Description		Best avg All Frames		Best First Frame	
Threshold		0.075		0.078	
Avaliation		Avg All Frames	First Frame	Avg All Frames	First Frame
Bootstrap	160x120	58.94%	72.00%	59.09%	71.71%
Campus	160x128	27.19%	30.06%	27.86%	30.77%
Curtain	160x128	51.11%	85.46%	50.40%	84.98%
Escalator	160x130	37.12%	47.27%	37.66%	47.46%
Fountain	160x128	63.02%	56.63%	63.02%	56.87%
hall	176x144	46.39%	74.92%	46.53%	75.89%
Lobby	160x128	47.46%	70.45%	46.59%	69.43%
ShoppingMall	320x256	70.71%	68.80%	70.49%	68.79%
lank3-rgb	640x480				
neovision3	1920x1088				
Mean		**50.24%**	**63.20%**	**50.21%**	**63.24%**

In Table 7 we could see CudaIncPCP with double precision.

Table 7 Cuda IncPCP C++ double Precision threshold and F-measurement for all datasets

Code		CUDA C++ (double precision)			
Threshold Description		Best avg All Frames		Best First Frame	
Threshold		0.148		0.173	
Avaliation		Avg All Frames	First Frame	Avg All Frames	First Frame
Bootstrap	160x120	52.64%	52.40%	53.52%	52.40%
Campus	160x128	27.01%	32.62%	32.49%	32.62%
Curtain	160x128	69.70%	87.73%	62.74%	87.73%
Escalator	160x130	40.30%	52.14%	44.77%	52.15%
Fountain	160x128	71.76%	53.18%	70.55%	53.18%
hall	176x144	50.68%	78.08%	50.19%	78.08%
Lobby	160x128	48.50%	66.19%	43.25%	69.19%
ShoppingMall	320x256	69.75%	68.58%	67.20%	68.58%
lank3-rgb	640x480				
neovision3	1920x1088				
Mean		**53.79%**	**61.37%**	**53.09%**	**61.37%**

In table 8 is show the same output from MATLAB version. (Rodrigues and Wohlberg 2015b)

Table 8 MATLAB IncPCP threshold and F-measurement for all datasets

Code		MATLAB			
Threshold Description		Best avg All Frames		Best First Frame	
Threshold		0.063		0.065	
Avaliation		Avg All Frames	First Frame	Avg All Frames	First Frame
Bootstrap	160x120	48.25%	60.04%	48.23%	59.55%
Campus	160x128	21.81%	25.46%	22.19%	25.80%
Curtain	160x128	51.09%	81.79%	50.68%	81.68%
Escalator	160x130	33.58%	44.86%	33.94%	45.28%
Fountain	160x128	53.34%	39.79%	53.49%	40.26%
hall	176x144	38.37%	60.91%	38.42%	61.53%
Lobby	160x128	37.95%	52.61%	37.38%	52.24%
ShoppingMall	320x256	57.16%	51.76%	56.99%	51.73%
lank3-rgb	640x480				
neovision3	1920x1088				
Mean		**53.79%**	**61.37%**	**53.09%**	**61.37%**

4.12. IncPCP timing

The timing is generating from `compute_all_data_set_timing` subprogram that runs all data sets with the fastest possible configuration, without any show, save even generating the binary mask. In MATLAB a similar script was done to test Rodriguez and Wohlberg (2015a) MATLAB software, this script is at `incPCP/run_all_datasets_timing.m`. The time are initialization time in second and FPS. In table 9 is the timing from IncPCP in C++ with double precision, followed by in Table 10 the single precision. The Table 11 contains the timing from from Rodriguez and Wohlberg (2015a) software. And finally in Table 12 the timing from our CUDA version with double precision.

Table 9: Timing for IncPCP C++ double precision

Code		C++ (double precision)	
Threshold Description		Time	
Threshold			
Avaliation		Init	FPS
Bootstrap	160x120	0.004	219,113
Campus	160x128	0.008	196.582
Curtain	160x128	0.007	194.780
Escalator	160x130	0.016	201.558
Fountain	160x128	0.002	191.489
hall	176x144	0.014	167.461
Lobby	160x128	0.004	194.830
ShoppingMall	320x256	0.006	33;764
WaterSurface	160x128	0,751	118,931
lank3-rgb	640x480	0.0751	118.931
neovision3	1920x1088	0.029	8,134
Mean		**0,087**	**138,893**

Table 10: Timing for IncPCP C++ single precision

Code		C++ (single precision)	
Threshold Description		Time	
Threshold			
Avaliation		Init	FPS
Bootstrap	160x120	0.002	213.432
Campus	160x128	0.009	196.046
Curtain	160x128	0.013	195.578
Escalator	160x130	0.007	180.769
Fountain	160x128	0.004	196.981
hall	176x144	0.013	164.034
Lobby	160x128	0.009	194.462
ShoppingMall	320x256	0.004	52.539
WaterSurface	160x128	0.005	184.096
lank3-rgb	640x480	0.023	10.381
neovision3	1920x1088	0.121	1.528
Mean		**0.019**	**144.531**

Table 11: Timing MATLAB double precision

Code		MATLAB	
Threshold Description		Time	
Threshold			
Avaliation		Init	FPS
Bootstrap	160x120	0.016	69.673
Campus	160x128	0.008	68.527
Curtain	160x128	0.009	68.437
Escalator	160x130	0.008	71.131
Fountain	160x128	0.008	70.062
hall	176x144	0.009	65.641
Lobby	160x128	0.009	68.122
ShoppingMall	320x256	0.010	26.699
WaterSurface	160x128	0.008	72.972
lank3-rgb	640x480	0.025	7.950
neovision3	1920x1088	0.109	1.209
Mean		**0.020**	**53.675**

Table 12: Timing CUDA C++ double precision

Code Threshold Description Threshold Avaliation		CUDA C++ (double precision) Time	
		Init	FPS
Bootstrap	160x120	0.009	34.122
Campus	160x128	0.010	33.093
Curtain	160x128	0.012	33.709
Escalator	160x130	0.016	35.327
Fountain	160x128	0.008	33.276
hall	176x144	0.014	28.199
Lobby	160x128	0.009	33.849
ShoppingMall	320x256	0.043	17.214
WaterSurface	160x128	0.007	31.559
lank3-rgb	640x480	0.087	6.043
neovision3	1920x1088	0.457	1.264
Mean		**0.061**	**26.150**

So we could see that our implementation in C++ in single and double precision are faster than MATLAB version from Rodrigues and Wohlberg, between 2.5x faster, or 167% faster. Our initialization time is bigger since as we had comment we initializated the algorithm with a SVD and not with a QR, what is a possible optimization.

However, for some reasons other CUDA version is slower than all of others, what is not expected at the begging of research. Maybe more debugging and perfling could found the timing wasted and maybe redesign some operations in GPU, or even do new ones that are wtill nowadays done in CPU, as a division of two matrices, check of background stability and so on. But, this is still a supposition because our principal operation in timing is the IncSVD

(75% of time in CPU) and it is faster in CUDA than C++ by 3x as shown before. More discuss could be found in chapter 5.

5. Conclusion and Future Work

It is visualized that the CUDA GEMM module, the wrapper have a numeric error (less than 1%) and it is faster than a multiplication in GPU, from 25x to 400x times faster.

The CudaSVD has also a good numeric error, what means do not affect visual specs of project, and is faster than SVD in C++ by 2x to 3x faster, but was not expected that the MATLAB version was even faster in just CPU. However this MATLAB is been faster than C++, the timing resulting about IncSVD and IncPCP in faster since this algorithm just requires really small SVD evaluation.

Also that the IncSVD in their three methods has a good numeric approximation also in CPU and in GPU. And that the GPU version is from 2x to 3x faster.

The qualitatively evaluation from the IncPCP algorithm in all the platforms are as expected a good foreground, with some datasets being difficult than others to be resistance to undesirable movement, as the curtain one, but we accomplished to be as good as an state-of-art-algortihm.

The quantitatively evaluation from ours IncPCP is always higher then MATLAB one, in an factor of 17.68% above even this was not intended to be achieved, but a collateral effect maybe for do not use QR factorization in initialization or maybe because SVD precision in C++ or C++/CUDA is better.

Another conclusions is that the accuracy from double and single precision are almost the same, but single precision is faster in all datasets, and so we had found a possible not expected optimization, the using of single precision. This is a valuable discover that could be even used in the MATLAB version in an easy way.

And even the C++ double and single precision been more accurate they are also faster than MATLAB, achieving our goal that keep the accuracy and become faster. The single precision is in average 107.38% faster in FPS and in all datasets are faster. The double precision is 43.85% faster than MATLAB, but is slower in some datasets, the bigger ones. Both of them 17% more accurate.

But MATLAB Original from Rodriguez and Wohlberg version has a good initialization time, in this aspect they are 96% faster than our double precision and 62% faster than the single precision, what probably could be fixed with QR factorization been applied to our version.

Unhappily the CudaIncSVD is slower than C++ and MATLAB, and do not work with single precision points, even been designed to do so. This is probably because of the lack of time of the project, that just finish this feature in the week 23 and it is yet a draft. The implementation has some bottlenecks that with more time could be solved, as so much copies from CPU and GPU and some conversions from double and single precision.

A desirable future work is include the QR factorization in both C++ and CUDA version to decrease this so long initialization time and analyse. After that analyse the CUDA code and why single precision is not working and remove the bottlenecks to be faster than C++ version, since all the submodules are faster than in CPU, is expected that this algorithm also were, but some integration problem avoid that.

Further that all developed methods (that were finished) have a decent numeric and qualitatively output.

Since IncPCP is an excellent in speed and good but not the best in speed (Hu et. al. 2015) in precision, and that Hu's algorithm use the IncPCP as a subroutine, a possible work after this code is faster than MATLAB in all specs (including initialization, and single precision CUDA).

6. References

Ferreira, Mauricio Azevedo Lage. Vigilância e Monitoramento em Tempo Real de Veículos em Rodovias com Câmeras Não-Calibradas. Diss. PUC-Rio, 2008. APA

Maghsoumi, H., Asemani, D. and Amirpour, H. 2015. "An efficient adaptive algorithm for motion detection," *Industrial Technology (ICIT), 2015 IEEE International Conference on*, Seville, 2015, pp. 1630-1634.
doi: 10.1109/ICIT.2015.7125330

Rodriguez, P., and Wohlberg, B. 2016a. Incremental Principal Component Pursuit for Video. J Math Imaging Vis (2016) 55:1–18 DOI 10.1007/s10851-015-0610-z

Rodriguez, P., and Wohlberg, B. 2015a. "Incremental Principal Component Pursuit" Available at https://sites.google.com/a/istec.net/prodrig/Home/en/pubs/incpcp [Accessed at 15/10/2016].

Rodriguez, P., Wohlberg B. 2015b."Translational and Rotational Jitter Invariant Incremental Principal Component Pursuit for Video Background Modeling", accepted IEEE International Conference on Image Processing (ICIP), (Quebec, Canada), September, 2015.

Y. Hu, K. Sirlantzis, G. Howells, N. Ragot and P. Rodríguez. 2015. "An online background subtraction algorithm using a contiguously weighted linear regression model," *Signal Processing Conference (EUSIPCO), 2015 23rd European*, Nice, 2015, pp. 1845-1849. doi: 10.1109/EUSIPCO.2015.7362703
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7362703&isnumber=7362087

Wikipedia, 2016a, "F1_score", https://en.wikipedia.org/wiki/F1_score access at 10/02/2017

Netlib, 2015. "BLAS (Basic Linear Algebra Subprograms)." http://www.netlib.org/blas/ [accessed 06/04/2016]

Li, L., Huang, W., Gu, I. and Tian, Q. 2003. "Foreground object detection from videos containing complex background," ACM International Conference on Multimedia.

Unknown (1), found in README file from (Rodriguez and Wohlberg 2015b) [Made available by authors at 06/02/2017].

Wikipedia. 2016b. "Orthogonal Matrix". https://en.wikipedia.org/wiki/Orthogonal_matrix [accessed at 07/04/2016]

Saharsh, B. 2015. “OpenCV CUDA Binaries”. https://www.nuget.org/packages/opencvdefault/ [accessed and download at 10/09/2016]

NVIDIA, 2016a, “CUDA Toolkit | NVIDIA Developer” https://developer.nvidia.com/cuda-toolkit [accessed on 06/02/2017].

HARTLEY, R., ZISSERMAN, A. (2004) “Multiple view geometry in computer vision”, 2nd edition, Cambridge University Press

NVIDIA, 2016b, “cuBLAS”,https://developer.nvidia.com/cublas [access on 06/02/2017]