Stereo Disparity through Cost Aggregation with Guided Filter

- CUDA GPGPU programming -

2017-2018

Project Report

Beyens Ziad
Nougba Hamza

ULB

Contents

Introduction

The goal of the project consists to implement a Cost-Volume Stereo Matching[1] with a guided filter in CUDA and to compare the performance with the C++ implementation.

Materials

Depending on GPU performance, materials details must firstly be specified:

- Graphic Card : Geforce GTX 1080
- Processor: Intel i7-7700K 4.2 GHZ (can boost to 4.5 GHZ and quad core.)
- 16GB RAM

Requirements

- C++ code.
- CUDA code.
- The acceleration should be at least one order of magnitude compared to CPU.
- The occupancy is larger than 70%.
- The occupancy and memory access rates should be given with the Nsight snapshots.

Structure

The program was divided in 4 steps:

1. Compute the Cost Volume during which the value of $p(i, d)$ for every disparity will be retrieved at once.

[1] TM14.

2. Compute the image integral to make the conception of guided filter easier and improve performance.

3. Filter the cost volume.

4. Detect and fill the occlusions.

The following images coming from the Middlebury datasets[2] are used, with a resolution of 3052x1968:

(a) Left **(b)** Right

Figure 1 – Color images

To verify that the code is correct tsukuba images are also used:

(a) Left **(b)** Right

Figure 2 – Color images - Tsukuba images

Source code

The source code of the program written in C++ can be found on GitHub.[3]

[2] `http://vision.middlebury.edu/stereo/data/`

[3] `https://github.com/hamza1030/stereo_matching_cuda`

Chapter 1

CUDA Optimization

In this section, several optimization on CUDA will be discussed.[1]

1.1 Independent parallelism

We want to maximize independent parallelism. For instance, each thread can process one pixel that can not used by another thread at the same time. Otherwise, a synchronization of the threads should be done, losing efficiency.

1.2 Data transfers

We want to minimize the data transfers because it costs a lot of processing time. Most of the computation should be done on the GPU. However, the processing time of the data transfers will not be taken into account for the benchmark.

1.3 Coalescing

We want memory coalescing. A memory access is coalesced if all the threads of a half-warp access the global memory at the same time. A global memory access consists of accessing 32, 64 or 128-bit words in one transaction. Thus, the consecutive threads should access consecutive memory address. In an array, the elements should be read and written row by row, from left to right. An example of non-coalescing is when the consecutive threads access the memory of an array column because the elements of a column are not contiguous in the memory. That is called as strided memory access. This problem can be avoided using shared memory.

[1] NVi09.

1.4 Shared memory

The shared memory is two order of magnitude faster than the global memory. It caches the data to minimize the global memory accesses. Also, the threads of a same block can access the same shared memory. However, the size of the shared memory is limited (about 64KB). Thus, an image can be processed block by block where each block has the size of the shared memory. Nevertheless, the global memory should be used if the threads does not need to access the shared memory of another block.

1.5 Bank conflicts

The shared memory architecture is divided into banks where the successive 32-bit words are stored in successive banks. The memory allows only one access of a bank per cycle. Thus, if multiple accesses to the same bank are done at the same time, there is a bank conflict: there is no more parallelism, slowing the processing. For instance, storing an array in a shared memory can be a bad idea if the elements of a column are in the same bank: strided memory accesses can lead to bank conflict.

Chapter 2

Grayscale Image

2.1 Implementation

Before everything else, the two input RGB images are transformed into grayscale images using the kernel and the global memory. This method is also used multiple times to warm-up the GPU.

The operation consists only in a weighted sum of each colored pixel component with its coefficient: 64 threads per block with a total of $nBlocksX = (n + blockDim.x - 1)/blockDim.x$ blocks, where $n = width * height$.

2.2 Results

The results are:

(a) Left **(b)** Right

Figure 2.1 – Grayscale images

(a) Left **(b)** Right

Figure 2.2 – Grayscale images - Tsukuba images

Chapter 3

Cost Volume

3.1 Implementation

As mentioned in the paper, cost volume is obtained with (horizontal) gradients of both images. Instead of doing it in two steps, the gradients are directly implemented in the kernel computing the cost volume. For each pixel, its gradient is computed in a *device* function. It remains to achieve a weighted sum as for computing the grayscale image. Additionally, each block has two dimensions because the cost volume is computed for each disparity: the first dimension each pixel, the second dimension for each disparity

3.2 Results and performance

As performance, the program does the cost volume process in less than 65 μs for an image of resolution 3052x1968 (about 6 millions of pixels) as you can see in the figure 3.1.

	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)	Context ID	Signature	Process ID
1	sumArraysOnGPU	14	5	3	0.01	2,667.291	741.277	889.097	1,184.161	1	sumArraysOnGPU(unsigned char*, unsigned char*, int, int)	29616
2	costVolumOnGPU2	17	19	2	0.44	126,231.788	63,103.933	63,115.894	63,127.855	1	costVolumOnGPU2(unsigned char*, unsigned char*, float*, int, int, int, int, int, int)	29616
3	chToFlOnGPU	15	2	2	0.00	642.458	320.877	321.229	321.581	1	chToFlOnGPU(unsigned char*, float*, int)	29616
4	rowSum	12	5	132	3.19	913,146.521	6,901.261	6,917.777	6,929.550	1	rowSum(float*, float*, int, int)	29616
5	colSum	12	1	132	1.71	490,744.238	3,716.784	3,717.759	3,719.089	1	colSum(float*, float*, int, int)	29616
6	computeBoxFilter	15	21	132	1.18	337,894.384	2,527.938	2,559.806	2,617.958	1	computeBoxFilter(float*, float*, float*, int, int)	29616
7	flToChOnGPU	15	13	2	0.00	678.363	339.022	339.182	339.341	1	flToChOnGPU(float*, unsigned char*, int)	29616
8	pixelMultOnGPU	15	6	36	0.07	18,605.300	493.523	516.814	521.556	1	pixelMultOnGPU(float*, float*, float*, int)	29616
9	pixelSousOnGPU	15	17	2	0.00	820.927	410.127	410.464	410.800	1	pixelSousOnGPU(float*, float*, float*, int)	29616
10	copyFromBigToLittleOnGPU	15	9	32	0.05	13,919.712	433.681	434.991	436.145	1	copyFromBigToLittleOnGPU(float*, float*, int, int)	29616
11	compute_ak	15	1	32	0.11	32,620.821	1,017.127	1,019.401	1,021.992	1	compute_ak(float*, float*, float*, float*, float*, int)	29616
12	compute_bk	15	8	32	0.07	20,262.811	622.425	633.213	727.805	1	compute_bk(float*, float*, float*, float*, int)	29616
13	compute_q	15	19	32	0.07	19,938.568	619.672	623.080	625.977	1	compute_q(float*, float*, float*, float*, int)	29616
14	copyFromLittleToBigOnGPU	15	20	32	0.05	13,759.414	429.073	429.982	430.800	1	copyFromLittleToBigOnGPU(float*, float*, int, int)	29616
15	selectionOnGpu	17	7	2	0.05	13,277.893	6,633.315	6,638.947	6,644.578	1	selectionOnGpu(float*, float*, float*, int, int, int)	29616
16	flToCh2OnGPU	19	11	2	0.00	0.000	0.000	0.000	0.000	1	flToCh2OnGPU(float*, unsigned char*, int, int, int)	29616
17	detect_occlusionOnGPU	19	2	1	0.00	0.000	0.000	0.000	0.000	1	detect_occlusionOnGPU(float*, float*, float, int, int)	29616

Figure 3.1 – Cost volume on GPU v1

Time is lost during this phase because the derivative is computed for each disparity. Then the x-derivative is done before the cost. In figure 3.2, it can be seen

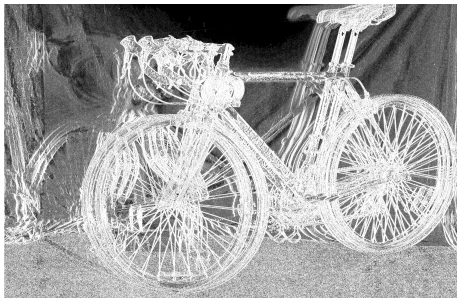
a processing time for the cost volume at 40 μs. The x-derivative has a negligible processing time.

Drag a column header and drop it here to group by that column

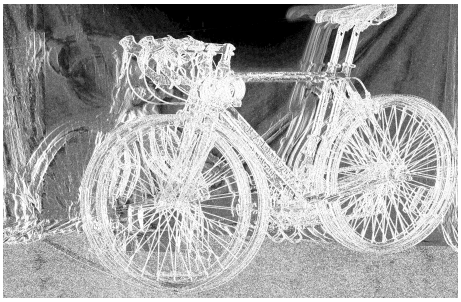
	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)	Context ID	Signature	Process ID
1	sumArraysOnGPU	11	5	9	0.05	9,733.211	741.117	1,081.468	1,184.473	1	sumArraysOnGPU(unsigned char*, unsigned char*, int, int)	31168
2	x_derivativeOnGPU	12	21	4	0.02	4,640.663	1,154.414	1,160.166	1,166.158	1	x_derivativeOnGPU(unsigned char*, float*, int, int)	31168
3	costVolumOnGPU2	12	1	2	0.42	80,820.894	40,408.382	40,410.447	40,412.512	1	costVolumOnGPU2(unsigned char*, unsigned char*, float*, float*, float*, int, int, int, int, int, int)	31168
4	chToFlOnGPU	16	2	2	0.00	642.873	321.356	321.437	321.517	1	chToFlOnGPU(unsigned char*, float*, int)	31168
5	rowSum	17	5	132	4.72	913,466.228	6,908.528	6,920.199	6,935.634	1	rowSum(float*, float*, int, int)	31168
6	colSum	17	1	132	2.54	490,735.783	3,716.786	3,717.695	3,718.739	1	colSum(float*, float*, int, int)	31168
7	computeBoxFilter	16	21	132	1.77	342,439.375	2,560.773	2,594.238	2,708.587	1	computeBoxFilter(float*, float*, float*, int, int)	31168
8	flToChOnGPU	16	13	2	0.00	678.459	339.150	339.230	339.309	1	flToChOnGPU(float*, unsigned char*, int)	31168
9	pixelMultOnGPU	16	6	36	0.10	18,570.971	493.523	515.860	519.924	1	pixelMultOnGPU(float*, float*, float*, int)	31168
10	pixelSousOnGPU	16	17	2	0.00	821.057	410.321	410.529	410.736	1	pixelSousOnGPU(float*, float*, float*, int)	31168
11	copyFromBigToLittleOnGPU	16	9	32	0.07	13,919.302	434.065	434.978	436.049	1	copyFromBigToLittleOnGPU(float*, float*, int, int)	31168
12	compute_ak_and_bk	16	12	32	0.22	42,160.252	1,314.451	1,317.508	1,320.468	1	compute_ak_and_bk(float*, float*, float*, float*, float*, float*, int)	31168
13	compute_q	16	19	32	0.10	19,940.469	620.024	623.140	626.329	1	compute_q(float*, float*, float*, float*, int)	31168
14	dispSelectOnGPU	16	10	32	0.10	20,076.123	613.624	627.379	652.634	1	dispSelectOnGPU(float*, float*, float*, int, int)	31168
15	flToCh2OnGPU	15	12	2	0.01	1,328.596	664.090	664.298	664.506	1	flToCh2OnGPU(float*, unsigned char*, int, int, int, int)	31168
16	detect_occlusionOnGPU	15	7	1	0.01	1,208.112	1,208.112	1,208.112	1,208.112	1	detect_occlusionOnGPU(float*, float*, int, int, int)	31168
17	fill_occlusionOnGPU1	15	17	1	0.36	69,827.384	69,827.384	69,827.384	69,827.384	1	fill_occlusionOnGPU1(float*, int, int, float)	31168

Figure 3.2 – Cost volume on GPU v2 - Factor speed of 1.5

Finally, the cost volume with slice $d = -15$ is displayed.

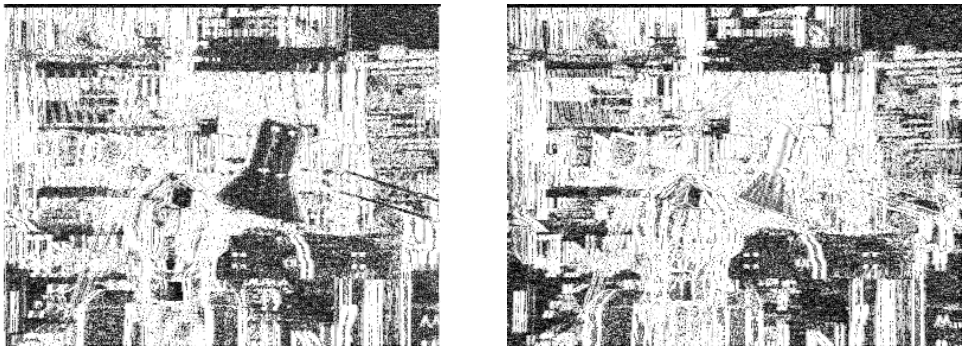


(a) Left



(b) Right

Figure 3.3 – Cost volume of slice $d = -15$



(a) Left **(b)** Right

Figure 3.4 – Cost volume of slice $d = -15$ - Tsukuba images

Chapter 4

Image integral

4.1 Implementation

It will be seen at the end that it is the most important function to achieve good performance. This method is used many times to compute Box Filters in the program. Firstly, the method to implement box filter was to use the shared memory and simply doing a convolution of kernel full of 1. But this method is slow and limited (in our individual case) for the radius. Instead, image integral is computed by summing each row (each pixel in a row corresponds to the sum of previous pixels). And after that, a column value is done.

Afterwards, a method inspired by the MIT[1] was implemented and slightly modified to have no zero in the first row and the first column.

4.2 Result and Performance

This step has the most costly operations of all because it is the most used but also because it is not optimized. This function is decomposed into two sub functions which are $rowsum$, $colsum$. Both processes are exceeding 1 second.

Then, we wanted to optimize the integral because, as seen in chapter one, the $colsum$ leads to non-coalesced memory access. A trick for that is to apply consecutively a $rowsum$, a transpose, a $rowsum$ again and finally a transpose. But, as seen in the course, the transpose also has strided memory accesses. The trick for that is to do the transpose in the shared memory with a size of 32x33. Why 33? 32x32 for the image processing and the last padding column to avoid the bank conflicts.

Result can be seen in figure 4.2. Nevertheless, the full image integral process takes more than 2 seconds.

[1] BHM10.

	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)	Context ID	Signature	Process ID
1	sumArraysOnGPU	11	5	9	0.01	9,717.668	732.285	1,079.741	1,184.320	1	sumArraysOnGPU(unsigned char*, unsigned char*, int, int)	23104
2	x_derivativeOnGPU	14	1	4	0.01	4,576.596	1,139.373	1,144.149	1,149.197	1	x_derivativeOnGPU(unsigned char*, float*, int, int)	23104
3	costVolumOnGPU2	14	16	2	0.08	69,880.596	34,938.206	34,940.298	34,942.390	1	costVolumOnGPU2(unsigned char*, unsigned char*, float*, float*, float*, int, int, int, int, int, int)	23104
4	chToFlOnGPU	18	2	2	0.00	634.136	316.780	317.068	317.356	1	chToFlOnGPU(unsigned char*, float*, int)	23104
5	rowSum	16	2	132	1.04	902,178.763	6,825.034	6,834.688	6,850.156	1	rowSum(float*, float*, int, int)	23104
6	colSum	16	3	132	0.57	490,706.638	3,715.345	3,717.475	3,718.545	1	colSum(float*, float*, int, int)	23104
7	computeBoxFilter	18	21	132	0.38	329,530.171	2,457.472	2,496.441	2,640.391	1	computeBoxFilter(float*, float*, float*, int, int)	23104
8	flToChOnGPU	18	13	2	0.00	669.690	334.829	334.845	334.861	1	flToChOnGPU(float*, unsigned char*, int)	23104
9	pixelMultOnGPU	18	6	36	0.02	18,314.476	487.251	508.735	512.692	1	pixelMultOnGPU(float*, float*, float*, int)	23104
10	pixelSousOnGPU	18	17	2	0.00	812.831	406.223	406.416	406.608	1	pixelSousOnGPU(float*, float*, float*, int)	23104
11	copyFromBigToLittleOnGPU	18	9	32	0.02	13,745.499	428.017	429.547	430.192	1	copyFromBigToLittleOnGPU(float*, float*, int, int)	23104
12	compute_ak_and_bk	18	12	32	0.05	41,742.782	1,302.834	1,304.462	1,307.027	1	compute_ak_and_bk(float*, float*, float*, float*, float*, float*, int)	23104
13	compute_q	18	19	32	0.02	19,716.513	614.168	616.141	617.912	1	compute_q(float*, float*, float*, float*, int)	23104
14	dispSelectOnGPU	18	10	32	0.02	20,465.953	624.152	639.561	698.491	1	dispSelectOnGPU(float*, float*, float*, int, int)	23104
15	flToCh2OnGPU	17	12	2	0.00	1,386.134	691.355	693.067	694.779	1	flToCh2OnGPU(float*, unsigned char*, int, int, int, int)	23104
16	detect_occlusionOnGPU	17	7	1	0.00	1,185.871	1,185.871	1,185.871	1,185.871	1	detect_occlusionOnGPU(float*, float*, int, int, int)	23104
17	fill_occlusionOnGPU1	17	17	1	0.07	62,665.493	62,665.493	62,665.493	62,665.493	1	fill_occlusionOnGPU1(float*, int, int, float)	23104

Figure 4.1 – RowSum and colSum are too high

	Function Name	Module ID	Function ID	Count	Device %	Device Time (µs)	Min (µs)	Avg (µs)	Max (µs)	Context ID	Signature	Process ID
1	sumArraysOnGPU	12	5	9	0.01	9,725.458	731.933	1,080.606	1,244.887	1	sumArraysOnGPU(unsigned char*, unsigned char*, int, int)	20352
2	x_derivativeOnGPU	16	1	4	0.01	4,584.118	1,141.101	1,146.030	1,150.990	1	x_derivativeOnGPU(unsigned char*, float*, int, int)	20352
3	costVolumOnGPU2	16	16	2	0.09	69,884.625	34,934.536	34,942.313	34,950.089	1	costVolumOnGPU2(unsigned char*, unsigned char*, float*, float*, float*, int, int, int, int, int, int)	20352
4	chToFlOnGPU	11	9	2	0.00	635.546	317.453	317.773	318.093	1	chToFlOnGPU(unsigned char*, float*, int)	20352
5	rowSum	19	2	264	1.93	1,485,291.977	4,374.604	5,626.106	6,846.094	1	rowSum(float*, float*, int, int)	20352
6	transpose	19	4	264	1.18	907,980.257	3,436.391	3,439.319	3,538.731	1	transpose(float*, float*, int, int)	20352
7	computeBoxFilterOnGPU	11	10	132	0.45	347,782.137	2,535.620	2,634.713	3,215.551	1	computeBoxFilterOnGPU(float*, float*, float*, int, int)	20352
8	flToChOnGPU	11	20	2	0.00	669.882	334.925	334.941	334.957	1	flToChOnGPU(float*, unsigned char*, int)	20352
9	pixelMultOnGPU	11	1	36	0.02	18,359.731	487.763	509.993	514.068	1	pixelMultOnGPU(float*, float*, float*, int)	20352
10	pixelSousOnGPU	11	12	2	0.00	812.256	405.296	406.128	406.960	1	pixelSousOnGPU(float*, float*, float*, int)	20352
11	copyFromBigToLittleOnGPU	11	3	32	0.02	13,742.689	428.881	429.459	430.257	1	copyFromBigToLittleOnGPU(float*, float*, int, int)	20352
12	compute_ak_and_bk	11	2	32	0.06	46,451.429	1,450.009	1,451.607	1,454.234	1	compute_ak_and_bk(float*, float*, float*, float*, float*, float*, int)	20352
13	compute_q	11	6	32	0.03	19,794.088	616.792	618.565	623.929	1	compute_q(float*, float*, float*, float*, int)	20352
14	dispSelectOnGPU	11	17	32	0.03	20,569.385	641.017	642.793	645.946	1	dispSelectOnGPU(float*, float*, float*, int, int)	20352
15	flToCh2OnGPU	15	12	2	0.00	1,036.905	518.196	518.453	518.709	1	flToCh2OnGPU(float*, unsigned char*, int, int, int, int)	20352
16	detect_occlusionOnGPU	15	7	1	0.00	1,078.603	1,078.603	1,078.603	1,078.603	1	detect_occlusionOnGPU(float*, float*, int, int, int)	20352
17	fill_occlusionOnGPU1	15	17	1	0.00	468.274	468.274	468.274	468.274	1	fill_occlusionOnGPU1(float*, int, int, float)	20352

Figure 4.2 – Results using transpose trick

Chapter 5

Guided Filter

5.1 Implementation

This step can be summarized as a series of the execution of the previous step. The box filter is used many times during the process to finally obtain the filtered cost. To help in computing co-variance, average filter, and other matrices, global functions doing basic operations like subtraction and scalar product on vectors were implemented in a loop of n iterations where n is the size of the disparity range.

5.2 Result and performance

We applied a Box Filter using image integral techniques

(a) Left **(b)** Right

Figure 5.1 – Average Filter of radius 9

(a) Left

(b) Right

Figure 5.2 – Average Filter of radius 9 - Tsukuba images

(a) Left

(b) Right

Figure 5.3 – Variance with radius 9

(a) Left

(b) Right

Figure 5.4 – Variance with radius 9 - Tsukuba images

The filtered cost seems to be wrong but it is just due to wrong parameters.

(a) Left

(b) Right

Figure 5.5 – Filtered Cost with $r = 9$ and $\epsilon = 255^2.10^{-4}$ (bad parameters for these stereo images)

(a) Left

(b) Right

Figure 5.6 – Filtered Cost with $r = 9$ and $\epsilon = 255^2.10^{-4}$ - Tsukuba images

Chapter 6

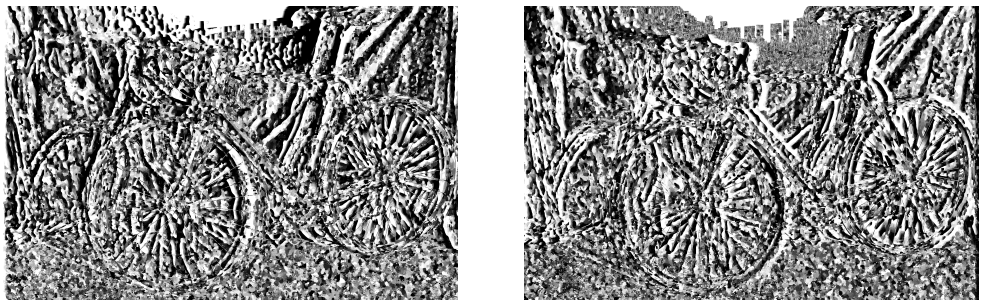
Disparity Map Filling

6.1 Implementation

In this step, each occluded pixel is marked and simply filled. The shared memory could not be used because one thread could need a data from another block. Thus, 1024 threads per block with $nBlocksX$ are used.

6.2 Results

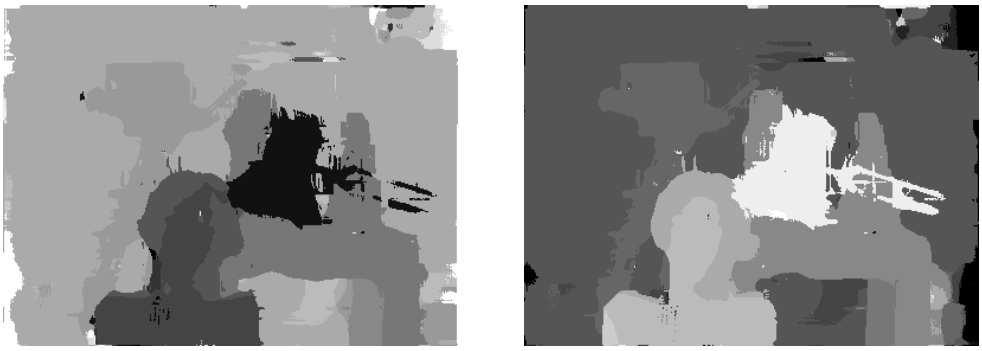
The parameters being not optimal for stereo images representing the bike, the following results might be strange. but it is normal, the CPU-version returns something



(a) Left **(b)** Right

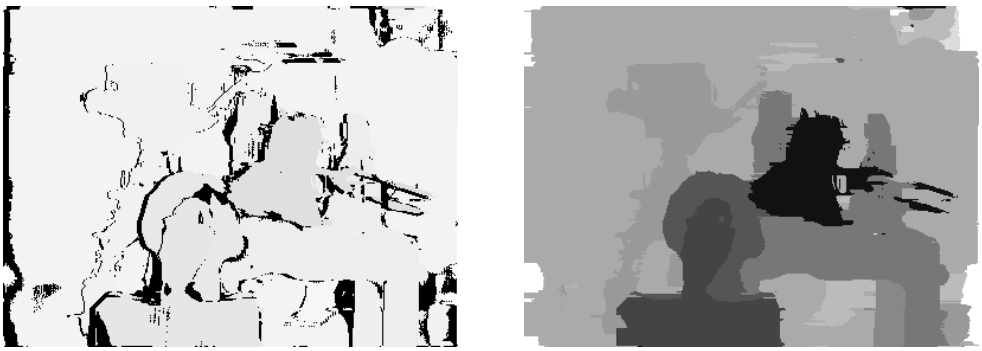
Figure 6.1 – Disparity map with r=9, $d_{LR} = 0$ and $\epsilon = 255^2.10^{-4}$

like this. To verify if the code is correct, then tsukuba images will be more useful.



(a) Left **(b)** Right

Figure 6.2 – Disparity map with r=9, $d_{LR} = 0$ and $\epsilon = 255^2.10^{-4}$ - Tsukuba images



(a) Left **(b)** Right

Figure 6.3 – Occlusions detection (left) and filling (right) of the guidance image with r=9, $d_{LR} = 0$ and $\epsilon = 255^2.10^{-4}$ - Tsukuba images

Chapter 7

Performance overview

7.1 Occupancy

The occupancy is always above 70 percent. Only the cost volume function has an occupancy of about 75 percent, otherwise it is averaged to 100 percent.

	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy
1	costVolumOnGPU2	{6912, 1, 1}	{16, 16, 1}	129,897.824	773.448	75.00%
2	costVolumOnGPU2	{6912, 1, 1}	{16, 16, 1}	135,514.309	771.911	75.00%
3	sumArraysOnGPU	{1728, 1, 1}	{64, 1, 1}	120,111.313	18.817	100.00%
4	sumArraysOnGPU	{1728, 1, 1}	{64, 1, 1}	121,370.259	18.209	100.00%
5	x_derivativeOnGPU	{108, 1, 1}	{1024, 1, 1}	129,602.129	36.834	100.00%
6	x_derivativeOnGPU	{108, 1, 1}	{1024, 1, 1}	129,640.147	35.073	100.00%
7	x_derivativeOnGPU	{108, 1, 1}	{1024, 1, 1}	135,185.588	38.146	100.00%
8	x_derivativeOnGPU	{108, 1, 1}	{1024, 1, 1}	135,225.238	35.266	100.00%
9	chToFlOnGPU	{864, 1, 1}	{128, 1, 1}	149,016.722	10.177	100.00%
10	rowSum	{1, 1, 1}	{1024, 1, 1}	149,899.776	505.754	100.00%
11	colSum	{1, 1, 1}	{1024, 1, 1}	150,407.226	369.971	100.00%
12	computeBoxFilterOnGPU	{24, 18, 1}	{16, 16, 1}	152,972.886	49.955	100.00%
13	flToChOnGPU	{864, 1, 1}	{128, 1, 1}	153,024.473	9.825	100.00%

Figure 7.1 – Occupancy of device function sorted in increasing order

The attached files "*cuda_launches_tsukuba.csv*" and "*cuda_launches_bike.csv*" contain more details.

7.2 Time and Gain

The CPU-version was used to estimate the gain without counting the process of smoothing the disparity map. For the tsukuba images, the program takes **0.63** seondes and the bike images, it takes about **35** secondes (fig 7.2).

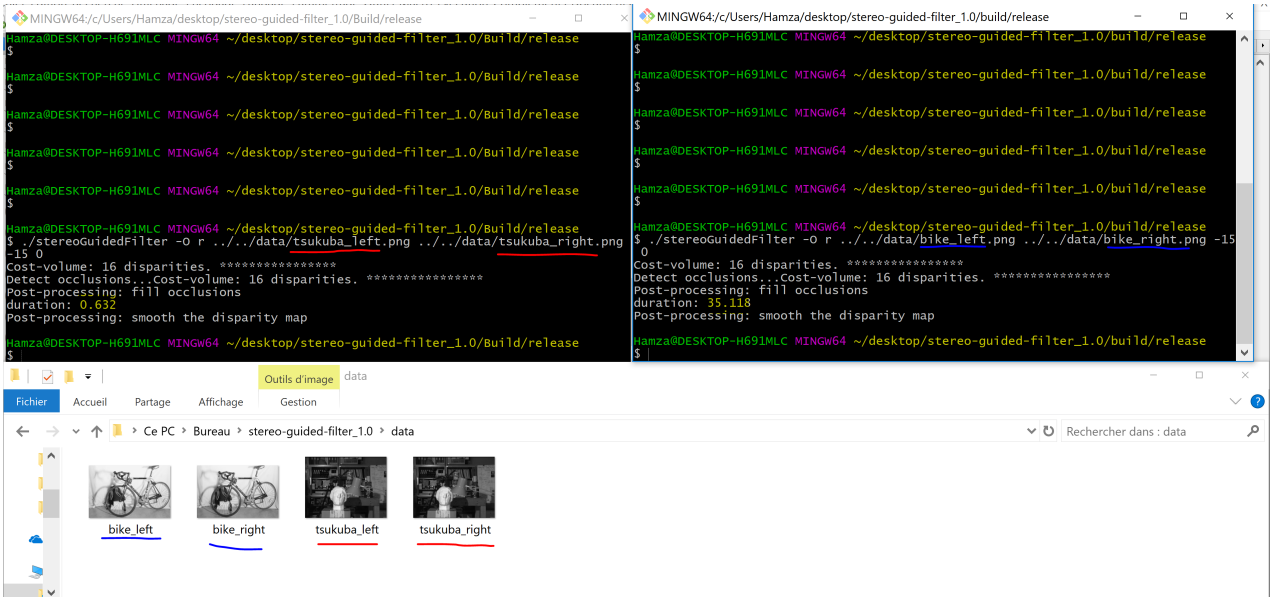


Figure 7.2 – Running Time for the CPU-version

Our GPU-version counting time of transfers between the host and the device, different allocations and others operations done in CPU (except writing images) takes **0.47** secondes for the tsukuba images and **11** secondes for the bike images (fig 7.3).

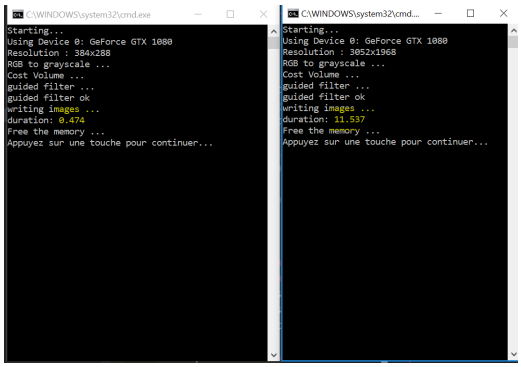


Figure 7.3 – Running Time for the GPU-version (with time of device $\Longleftrightarrow$ host transfer)

When only the time of functions in GPU are taken into account, it is done in **less**

than 0.13 secondes for the tsukuba images and in **less than 2 secondes** for the bike images (fig 7.4). It could reach a maximum gain of **17**.

Top Device Functions By Total Time Summary | All

	Name	Launches	Device %	Total (µs)	Min (µs)	Avg (µs)	Max (µs)
1	rowSum	132	7.14	65,340.634	492.475	495.005	505.945
2	colSum	132	5.23	47,845.695	361.105	362.467	370.099
3	computeBoxFilterOnGPU	132	0.71	6,502.921	47.970	49.265	54.147
4	costVolumOnGPU2	2	0.17	1,545.359	771.911	772.680	773.448
5	compute_ak_and_bk	32	0.11	1,025.741	31.298	32.054	32.706
6	compute_q	32	0.06	525.434	16.001	16.420	16.929
7	pixelMultOnGPU	36	0.05	494.971	12.193	13.749	14.433
8	dispSelectOnGPU	32	0.05	490.001	13.152	15.313	17.024
9	copyFromBigToLittleOnGPU	32	0.04	372.465	11.360	11.640	11.937
10	x_derivativeOnGPU	4	0.02	145.319	35.073	36.330	38.146

(a) Left

Top Device Functions By Total Time Summary | All

	Name	Launches	Device %	Total (µs)	Min (µs)	Avg (µs)	Max (µs)
1	rowSum	132	3.60	902,368.627	6,820.880	6,836.126	6,856.464
2	colSum	132	1.96	490,694.031	3,714.836	3,717.379	3,718.771
3	computeBoxFilterOnGPU	132	1.33	334,015.588	2,491.138	2,530.421	2,623.464
4	costVolumOnGPU2	2	0.28	69,830.735	34,913.000	34,915.368	34,917.735
5	compute_ak_and_bk	32	0.16	39,267.315	1,225.265	1,227.104	1,229.105
6	dispSelectOnGPU	32	0.08	19,832.854	602.872	619.777	642.074
7	compute_q	32	0.08	19,782.606	615.385	618.206	621.625
8	fill_occlusionOnGPU1	1	0.08	19,013.936	19,013.936	19,013.936	19,013.936
9	pixelMultOnGPU	36	0.07	18,358.746	486.931	509.965	515.829
10	copyFromBigToLittleOnGPU	32	0.05	13,744.190	428.529	429.506	430.449

(b) Right

Figure 7.4 – Time of functions for tsukuba images (left) and bike images (right)

7.3 Memory

For the bike images, the total size of data transferred from host to device is above 31 GB which is 55 times higher than the tsukuba images but the last one have about 55 times less pixels. The number is normal knowing that the integral is performed 132 times and the cost volume two times. The number of transfers still remains too high and costs too much time.

CUDA Devices All

	Device ID	Device Name	Contexts	Device %	Host to Device (Bytes)	Device to Host (Bytes)
1	[0]	GPU 0 - GeForce GTX 1080	1	13.61%	574,857,216	136,912,896

CUDA Contexts

	Total	No Context	1
CUDA Device ID		-	0
Runtime API Calls Summary | All			
# Calls	7.034	2	7.032
# Errors	0	0	0
% Time	54,44	0,04	54,41
Driver API Calls Summary | All			
# Calls	95	95	0
# Errors	0	0	0
% Time	0,05	0,05	0,00
Launches Summary | All			
# Launches	576	0	576
% Device Time	13,61	0,00	13,61
Memory Copies All			
H to D # Copies	1.249	0	1.249
H to D # Bytes	574.857.216	0	574.857.216
H to D % Time	5,2	0,0	5,2
D to H # Copies	284	0	284
D to H # Bytes	136.912.896	0	136.912.896
D to H % Time	1,3	0,0	1,3
D to D # Copies	0	0	0
D to D # Bytes	0	0	0
D to D % Time	0,0	0,0	0,0

(a) Left

CUDA Devices All

	Device ID	Device Name	Contexts	Device %	Host to Device (Bytes)	Device to Host (Bytes)
1	[0]	GPU 0 - GeForce GTX 1080	1	7.75%	31,220,934,528	7,435,843,968

CUDA Contexts

	Total	No Context	1
CUDA Device ID		-	0
Runtime API Calls Summary | All			
# Calls	7.034	2	7.032
# Errors	0	0	0
% Time	35,28	0,00	35,27
Driver API Calls Summary | All			
# Calls	95	95	0
# Errors	0	0	0
% Time	0,00	0,00	0,00
Launches Summary | All			
# Launches	576	0	576
% Device Time	7,75	0,00	7,75
Memory Copies All			
H to D # Copies	1.249	0	1.249
H to D # Bytes	31.220.934.528	0	31.220.934.528
H to D % Time	18,5	0,0	18,5
D to H # Copies	284	0	284
D to H # Bytes	7.435.843.968	0	7.435.843.968
D to H % Time	4,6	0,0	4,6
D to D # Copies	0	0	0
D to D # Bytes	0	0	0
D to D % Time	0,0	0,0	0,0

(b) Right

Figure 7.5 – Summary of memory information for tsukuba images (left) and bike images (right)

The attached files "*cuda_memory_copies_tsukuba.csv*" and "*cuda_memory_copies_bike.csv*" contain more details about the memory copies.

Conclusion

As a conclusion, the results show that CUDA can drastically increase the performance of a program by using the parallelism of threads. Nevertheless, it needs a lot of trick to avoid non-coalesced memory access, bank conflicts and whether to use or not the shared memory. The processing time could be compared with success. The program could also be better in particular during the integral image where most of the processing time is and also by decreasing the number of transfers.

Bibliography

[BHM10] B Bilgic, B K P Horn, and I Masaki. "Efficient integral image computation on the GPU". In: IEEE, June 2010, pp. 528–533. ISBN: 978-1-4244-7866-8. DOI: 10.1109/IVS.2010.5548142. URL: http://ieeexplore.ieee.org/document/5548142/ (visited on 05/08/2018).

[NVi09] NVidia. "Advanced CUDA Webinar - Memory Optimizations". en. In: (2009).

[TM14] Pauline Tan and Pascal Monasse. "Stereo Disparity through Cost Aggregation with Guided Filter". In: *Image Processing On Line* 4 (Oct. 2014), pp. 252–275. ISSN: 2105-1232. DOI: 10.5201/ipol.2014.78. URL: http://www.ipol.im/pub/art/2014/78/?utm_source=doi (visited on 05/02/2018).

[YJ08] Eric Young and Frank Jargstorff. "Image Processing & Video Algorithms with CUDA". en. In: (2008), p. 60.