Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

David Honzátko

GPU Acceleration of Advanced Image Denoising

Department of Software Engineering

Supervisor of the bachelor thesis:	RNDr. Martin Kruliš, Ph.D.
Study programme:	Computer Science
Specialization:	Obecná informatika

Prague 2015

I would like to dedicate this thesis to my family, my girlfriend, and my friends, without whose support this would have never been possible. I would like to specially thank my supervisor Martin Kruliš for an immeasurable patience he had reading and correcting the preliminary versions of this thesis.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In .... date .... signature of the author

Název práce: GPU Acceleration of Advanced Image Denoising

Autor: David Honzátko

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Martin Kruliš, Ph.D.

Abstrakt: BM3D (Block-Matching and 3D Filtering) je jedna z nejlepších metod na odšumování obrázků. Efektivní implementace této metody jsou existují, nicméně jsou časově náročné. Na běžných stolních počítačích může odšumění obrázků s vysokým rozlišením trvat i několik minut. Hlavním cílem této práce je navrhnout implementaci metody BM3D, která bude využívat surové výpočetní síly GPU. GPU nabízí mnohem více výpočetních jader než CPU, nicméně, kvůli specifickému výpočetnímu a paměťovému modelu, algoritmy pro GPU se od algoritmů pro CPU velmi liší. Proto tato práce prezentuje jak základní aspekty programování pro GPU, tak BM3D metodu jako takovou. Navržená implementace je pak experimentálně zhodnocena oproti těm současným.

Klíčová slova: odstraňování obrazového šumu, BM3D, paralelní, GPGPU, CUDA

Title: GPU Acceleration of Advanced Image Denoising

Author: David Honzátko

Department: Department of Software Engineering

Supervisor: RNDr. Martin Kruliš, Ph.D.

Abstract: BM3D (Block-Matching and 3D Filtering) is one of the state-of-art image denoising methods. Efficient implementations of this method exist for the CPU; however, these implementations are time demanding. On common desktop computers, denoising of high-resolution images can reach several minutes. The main objective of this thesis is to design an implementation of the BM3D method that utilize raw computational power of the GPU. GPU offers significantly more computational cores than the CPU; however, due to the specific execution and memory model, algorithms for the GPU are very different from algorithms for the CPU. Therefore, this thesis presents both: the basic aspects of the GPU computing and the BM3D method itself. Last but not least, the final implementation is empirically evaluated against the existing implementations by a set of performance tests.

Keywords: image denoising, BM3D, parallel, GPGPU, CUDA

Contents

1. Introduction

Photographers are struggling with noise in pictures since the beginning of a digital photography itself. Nowadays, when the size of image capturing devices is decreasing at the expense of noise corruption, fast and effective image denoising algorithms are necessary for preserving the image quality.

Human eye can see noise in an image as random speckles on an otherwise smooth surface. These speckles are not commonly desirable and can significantly degrade the image quality. More precisely, we can define image noise as undesired random information added to the expected image.

Noise in photographs generally originates in circuits of image sensors. It is mostly caused by the omnipresent thermal noise [1] that is unavoidable at common temperatures. Even though the level of noise generated by circuits of image sensors is constant on average, the level of noise in the produced image differs based on light conditions. At low illumination, digital cameras are forced to set the sensor to be more sensitive in order to produce as bright pictures as at normal light. Image sensor is then more sensitive to the collected light as well as to the noise, which is therefore more visible in the image. Moreover, the level of noise inversely depends on the physical size of a pixel of the sensor [2]. Because consumer electronics tends to be smaller or tighter, image sensors in these devices have to follow that trend too. As a consequence, employment of some kind of noise reduction is necessary.

Basic methods of image denoising are based on suppression of large differences in adjacent pixels. However, this approach encounters problems with edges. The edges are considered as large differences and they are suppressed which results in a blurry output. The same applies to small details. Unfortunately, a human eye considers a blurry image often less pleasant than the sharp noisy original. Although there are simple filters that are able to detect edges and preserve them, they commonly fail in preserving details and often produce artefacts. The main objective of advanced denoising methods is the noise reduction that preserves edges and detail.

One of the best image denoising methods is the **Block-Matching and 3D Filtering** (BM3D) [3] [4]. It effectively combines two main denoising approaches. The first is based on the fact, that noiseless image has much more sparse representation after some decorrelating transform as Fourier, Cosine, or Bior transform. The noise is then easier to detect and attenuate. The second approach is based on the block-matching concept that is commonly used for motion estimation in video compression algorithms. Each pixel is estimated from image fragments that are found to be similar to the neighbourhood of this pixel.

Briefly, BM3D finds the similar image fragments and then use this similarity to enhance the denoising in transform domain. This whole process is both memory and computationally intensive. It could be one of the reasons why it is not massively employed in photo editing tools.

1.1 Parallelization

Algorithms are traditionally constructed to be executed in a serial manner. The processing time of these algorithms, especially the computationally intensive ones, is often limited by clock-rate of the core. Unfortunately, the growth of clock-rate rapidly slowed down at the very beginning of the 21st century and since then, the growth of CPU computational power has been maintained mainly by adding new cores. This resulted in quick development in the area of parallel computing, which is focused on algorithms that utilize more compute units.

Parallel algorithms are based on the fact that the mutually independent parts of a problem can be processed at the same time. The number of simultaneously processed parts is limited by the number of cores in the system.

Although the number of cores in CPUs is growing, even the high-end processors contain only up to tens of them. It is mainly due to the complexity, related power consumption and heat production of a CPU core. The complexity of each core is determined by the ability to run an operating system. However, for many computing tasks, complex and independent cores are not necessary. One group of these tasks are data parallel tasks, where the same routine is performed over many data elements. A straightforward example of these data parallel tasks are image processing methods including image denoising where each pixel or set of pixels can be processed independently.

Data parallelism is so common that current CPU cores contain SIMD (Single Instruction Multiple Data) instructions that can process several data elements simultaneously. Still, even with the use of these instructions, the total number of concurrently processed data is very limited. Many data parallel problems offers numerous times higher parallelization than can be achieved on a single CPU. For such cases, we may use specialized parallel accelerators such as Xeon Phi cards and GPUs (Graphic processing units) that support GPGPU (General-purpose computing on GPU).

Unlike the Xeon Phi card, the GPU is available in almost every computer including notebooks, tablets, and mobile phones. For small tasks performed by end users (as image denoising) it is far more logical to adjust the software to the available hardware than vice versa; hence we will focus on the GPGPU platform.

GPU cores are not as complex, independent, nor large as their CPU counterparts; therefore, the current desktop GPUs can contain thousands of them. This does not imply that we will achieve thousands of times faster implementations compared to the serial version. There are several issues the GPU implementations have to deal with.

- **Data transfers:** The GPU often resides on graphic card that is connected with the host computer via a PCIe bus. Since this bus has limited bandwidth and since the data we want to process are often in the computer memory, we have to take into account the time to transfer the data to the GPU and back.

- **Memory hierarchy:** There are several types of memory on a GPU. They differs according to the purpose, latency, bandwidth, size, and locality. Their performance is often influenced by access patterns, we use. Therefore, we have to take some care to optimally utilize all these memories.

- **SIMT execution model:** Unlike a CPU, threads of a GPU are executed according to the SIMT (Single Instruction Multiple Thread) model. This restricts the tasks that can be efficiently solved on a GPU mainly to those that are massively data-parallel.

GPU cores are few times slower than current high-end CPU cores. Nevertheless, crucial for GPUs is the quantity. Therefore, the GPU can be a suitable platform for many data parallel algorithms, especially the compute intensive ones. Image processing algorithms are not an exception.

1.2 Related Work

Image denoising is nowadays a very common task. Since it is by definition data-parallel, various denoising algorithms has been accelerated using GPUs. Simple smoothing kernels, where each pixel is computed as a weighted average of its direct neighbours, are almost model examples of problems suited for GPUs.

Among the advanced denoising methods, the mostly studied in terms of GPU acceleration are those based on non-local similarities as *Non Local Means* (NL-means) [5]. Due to their high denosing effectivity and related high computational costs, several algorithms were developed [6] [7] [8]. GPU-accelerated NL-means based algorithms are nowadays widely used and they are contained in libraries as OpenCV.

Other GPU-accelerated denoising algorithms are based on sliding-window filters [9] or wavelet tranforms [10]. There are also algorithms that were created for specific (mostly medicine) purposes with specific noise models [11] [12]. Intensive research has been also made in fast or real-time denoising [12] [13]. These algorithms mostly do not offer the best denoising performance, however, due to their short processing times, they are often employed in video denoising.

Although some of these algorithms provide good denoising performance, in general neither of them offer better denoising qualities than BM3D. For more accurate comparison of denoising algorithms we refer to the work of Marc Lebrun [14].

1.3 Objectives

The main objective of this thesis is to design a GPU accelerated version of the BM3D method. We focus on the defining parts of this method - the block-matching and the 3D filtering. Transformation algorithms used by the method are not covered by this thesis. A prototype implementation which addresses the fundamental basics of this method is developed and compared with the existing implementations.

GPU accelerated BM3D algorithm might be a solution for users that do not have high performance computers, that are demanded by current implementations of this method. Furthermore, with the growing GPU performance, such algorithm might be used even in small electronic devices as mobile phones or tablets for immediate denoising of captured pictures.

The BM3D method itself is revised in Chapter 2. The basic ideas of this method are described and clarified as well as all the theoretical basis necessary for understanding it. Chapter 3 presents the GPU architecture fundamentals with all the benefits and restrictions. It describes GPUs memory model and problems arising from the low bus bandwidth between the GPU and CPU. It also briefly introduces the programming platform. Chapter 4 discusses the design of a GPU accelerated version of the BM3D algorithm. The reasons leading to the selected design are discussed for each significant part of the algorithm. Finally, Chapter 5 empirically evaluates the prototype implementation against the existing implementations comparing their performance, efficiency, and denoising effectiveness.

2. Block-Matching and 3D Filtering

BM3D is one of the most advanced algorithms for image denoising. Despite the fact, that BM3D has been published in 2006, according to the work of Marc Lebrun [15] it still has comparable results to other contemporary denoising methods. It effectively combines two successful denoising approaches **non-local filtering** and **transform based filtering**. To properly understand any of these approaches we first present the theory of denoising in general and the model of the noise we are working with. Later on, we sumarize both of the above mentioned approaches and finally we reveal the BM3D algorithm itself.

2.1 Noise

In this thesis we assume pictures that are captured by image sensors. Noise that appears in these pictures has several properties that are essential for the field of image denoising.

Despite the fact, that we are talking about the noise produced by image sensors, in order to simplify a notation we will use the notation that illustrates the simulation of noise corruption. First we have an image, that we denote *original image*, we corrupt it by noise with similar properties to the noise produced by an image sensor. Hence, we get a *noisy image.* Applying the denoising algorithm on it we obtain an *image estimate.* By comparing this estimate with the original image we can measure quality of denoising algorithms.

Before we present a mathematical model of the noise, we first define our conception of an image. We consider only grayscale images represented as matrix of pixels intensities. Nevertheless, an extension to the colour images is often very simple.

The properties of the noise produced by image sensor are essential for its elimination or simulation. Observations proves that this noise appears to be an **Additive White Gaussian Noise** (AWGN), which is defined by the following properties.

- **Additive** means that the noise is added to the original image. Hence, by subtracting the noise n from a noisy image g we can get the desired original picture f. The following formula is then valid:

  $$g = f + n$$

  where + is element-wise addition operator.

- **White** stands for the fact, that the level of the noise on each pixel is independent of the others and the mean value is zero.

- **Gaussian** denotes, that the values the noise n can take follow Normal distribution.

There are more noise models, which we would have to take into account for some specific images (e.g., very low intensity images). This and other models are well described in the Hnadbook of Image and Video Processing [16]. In general, the AWGN model well approximates most of the noise that appears in common images; hence, we shall consider this model only.

Because the noise n is additive, knowing the n we could reconstruct the original image f from the noisy image g by this formula:

$$f = g - n$$

Unfortunately, due to the randomness of the noise, for general images, the only thing we know is the noisy image g and the fact that n is AWGN.

Consequently, not all images we are even able to denoise. For example if the original image exhibit the same properties as the noise, we would not distinguish what is the noise and what the original. Therefore, regardless the denoising algorithm, the original image has to expose some level of order to be successfully denoised. For natural or any human-readable images, some level of order is guaranteed, because these images are generally made of finite number of geometric shapes. In this thesis, we shall consider these images only.

2.2 Non-Local Means

Generally, if we shoot more photographs of the exactly same scene and make an average of them, we shall get the best result in terms of noise attenuation and detail preservation. This is true due to the randomness of the noise. While the situation we want to shoot remains the same, the noise is changing with each picture we take. Consequently, the noise can be simply attenuated by averaging these pictures. The more pictures we take the less noticeable the noise will be. This approach is called *time-averaging*.

However, the situation not always allows us to take more pictures of a scene and when it does, it is often better to take one picture with a longer shutter speed and lesser sensitivity. Henceforth, we consider only the situation, where we have one noisy image.

NL-means [5] method is also employing the idea of *time-averaging*, but it operates only with one noisy image. It uses the fact, that images are usually made of geometric shapes and therefore they contain lots of repetitive areas. When it finds all these areas it uses a concept similar to *time-averaging*. More precisely, for each reference pixel and its neighbourhood NL-means searches the pixels with a similar neighbourhood and the pixel of image estimate is computed as an average of the found pixels.

The process of searching for the areas that are similar is long known in the field of video compression under the term **block-matching**. It is used for coding the displacements of objects between consecutive frames and it is a key part of algorithms as MPEG 1,2, and 4 as well as many others.

Even though we are using the term block-matching in this thesis, we will denote the areas of a picture that are subject of matching as *patches* instead of *blocks*. By this notation, we avoid the ambiguity between *blocks* as we would have known them from this chapter and *thread blocks* which are presented in GPU chapter.

2.3 Transform Based Denoising

Transform based denoising methods are very popular. They first transform the noisy image by some decorrelating unitary transform and then they filter the coefficients of this transformed image. Finally, they transform the image back to obtain the image estimate. The point of this approach is that while an original image after this transform has much more sparse representation, the noise does not and therefore it can be more easily attenuated.

There are several suitable transforms that can be used for this task. There are for example Fourier, Cosine, Bior and Walsh–Hadamard transform. We will not cover all these transforms in this thesis, but we will focus on the first one of them, the Fourier transform (FT).

2.3.1 Fourier Transform

Form mathematical point of view, the picture can be taken as a discretely sampled continuous 2D signal. The 2D FT applied to this signal decomposes it into the spatial frequencies that make it up. Although the continuous FT is easier to understand, for computers it is faster to work with traditional discrete images and transform them using the 2D Discrete Fourier Transform (DFT).

Here, we present only the basis of the DFT, More about the FT, DFT and their application in image denoising can be found in the book 'Digital Image Processing' [17].

2D DFT decomposes an image f of size $M \cdot N$ into a finite set of discretely sampled spatial frequencies. The result is a matrix of complex coefficients F, which has the same size as the image we transform. Coefficient located at $F_{u,v}$ defines the phase and amplitude of discretely sampled spatial frequency $q_{u,v}$:

$$q_{u,v}(x,y) = e^{\frac{2\pi iuj}{M}} e^{\frac{2\pi ivk}{N}} = cos\left(\frac{2\pi uj}{M} + \frac{2\pi v}{N}\right) + isin\left(\frac{2\pi uk}{M} + \frac{2\pi v}{N}\right)$$

The domain of these complex sinusoids is of the same size as the image. Values u, v practically give number of cycles per image in each dimension. Figure 2.1 visualize one of these discretely sampled complex sinusoids.

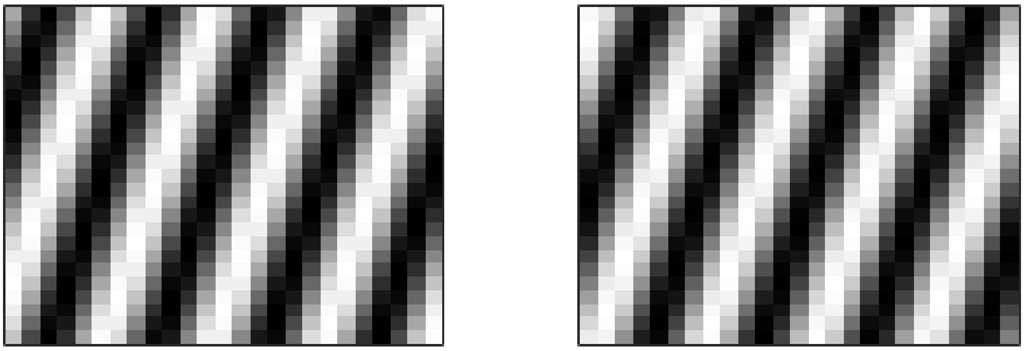

Figure 2.1: Sine and Cosine component of a sinusoid $q_{4,1}$

Every coefficient is a complex value. The phase and amplitude match the phase and absolute value of this complex number in their geometric representation.

The DFT transforming the image f of width M and height N into the matrix of coefficients F is defined as follows:

$$\forall_{u\in\{0..N-1\},v\in\{0..M-1\}} F_{u,v} = \frac{1}{\sqrt{MN}} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f_{x,y} e^{-2\pi i(\frac{ux}{M}+\frac{vy}{N})}$$

The DFT is invertible, which is necessary for transform based denoising. The *Inverse Discrete Fourier Transform* (IDFT) is defined as follows:

$$\forall_{x\in\{0..N-1\},y\in\{0..M-1\}} f_{x,y} = \frac{1}{\sqrt{MN}} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F_{u,v} e^{2\pi i(\frac{ux}{M}+\frac{vy}{N})}$$

An image, as we have defined it before, is a real-valued matrix. If such a matrix is transformed using the FT, the result satisfies the Hermitian symmetry, defined as $F_{u,v} = F^*_{M-u,N-v}$, where the star denotes complex conjunction. Therefore, it is not necessary to store all the coefficients. The same applies to the IDFT. If we pass it any complex matrix that satisfies Hermitian symmetry, the resulted matrix is always real. Consequently, we have to make symmetric changes to the transformed image in order to get valid image after the IDFT.

2.3.2 Power Spectrum and Noise Reduction

We use the FT for the noise reduction because the AWGN has some important characteristics in frequency basis. To understand and exploit these characteristics we define a *power spectrum* of an image as $|F_{u,v}|^2$. This is also called an *energy.*

The most important characteristic of AWGN is that its power spectrum is constantly equal to its variance σ^2. This also explains why this noise is called white. It is due to the analogy with white light, which has constant energy in the whole spectrum.

On the other hand, the original image often consists of low frequencies with higher amplitudes than σ^2. Figure 2.2 shows an original and noisy image and their respective power spectra.

Consequently, we can attenuate the noise simply by a low-pass filter (cancelling the coefficients of high frequencies) or by hard-thresholding (cancelling the coefficients below some threshold). The low-pass filter assumes that natural images have most of the information in low-frequencies, while the hard-thresholding presumes that these images consists of a small set of frequencies which high energy. Experimental results as well as figure 2.2 shows that the second presumption is more accurate.

The main problem of both of these approaches is that the information about an original image that was contained in the cancelled coefficients is now lost. In other words, an image looses detail and, because sharp edges are composed from many sinusoids with declining energy, after the IDFT the *ringing effect* can appear around edges.

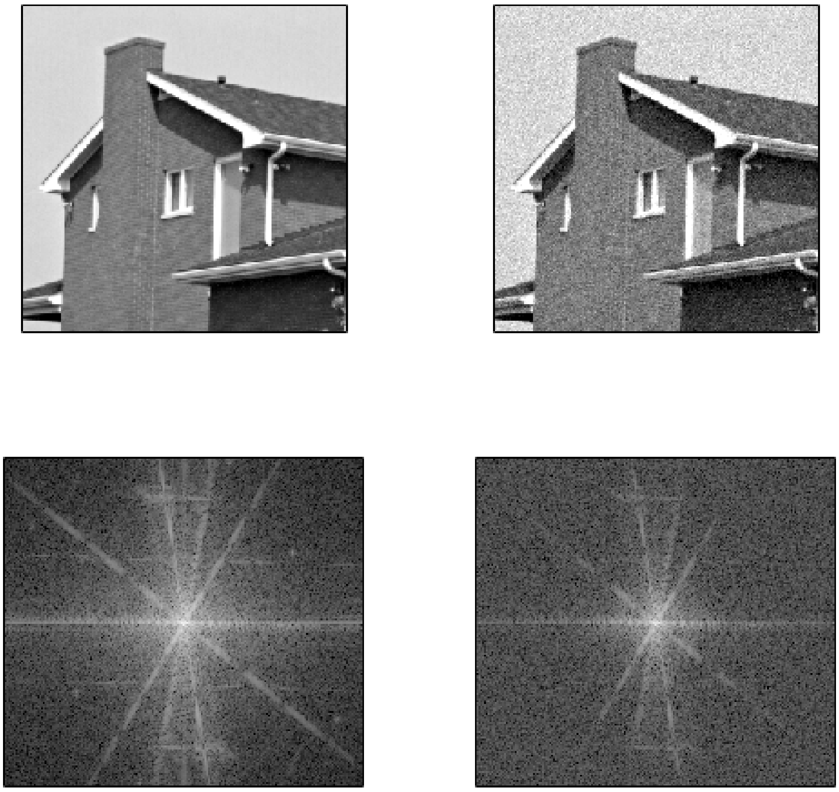

Figure 2.2: Original and noisy image and their respective power spectra, the zero coordinates are in the middle and the intensity is logarithmically scaled

2.4 BM3D Algorithm

The BM3D algorithm consists of three essential consecutive parts **Grouping**, **Collaborative Filtering**, and **Aggregation**.

- **Grouping** uses the block-matching concept to find similar patches for each reference patch. It groups the reference patch with the similar patches into a 3D array.
- **Collaborative filtering** is the main denoising procedure of the algorithm. It employ the fact, that each fragment has sparse representation in 2D transform basis. It exploits the similarity between stacked fragments by applying a 3D transform on the whole array which results in even sparser representation of the original image. Transformed image is then filtered by hard-thresholding. After an inverse transform the fragment estimates are returned to their original locations.
- **Aggregation** part computes the denoised image as a weighted average of all overlapping fragments.

We present only the simplified version of BM3D. The original BM3D method consists of two steps. First based on hard-thresholding and second based on *wiener filtering*. In this thesis we present only the first step of the algorithm; however, most of the developed processes can be reused in the second step as well. Although BM3D can use various transforms, we employ only the above mentioned Fourier transform. To simplify the reading, the notation used in this thesis is similar to the notation used in the paper by Marc Lebrun [14].

2.4.1 Grouping

For the first, grouping, part of the BM3D method we employ the above mentioned concept of block-matching. We define a *patch* as any square area of $k \cdot k$ pixels in image. Patches defined in this way can overlap.

For each reference patch P of a noisy image, the noisy image is searched in a $n \cdot n$ neighbourhood of this patch for patches that are similar to the reference one. An example result of block-matching on a single image is shown in figure 2.3.

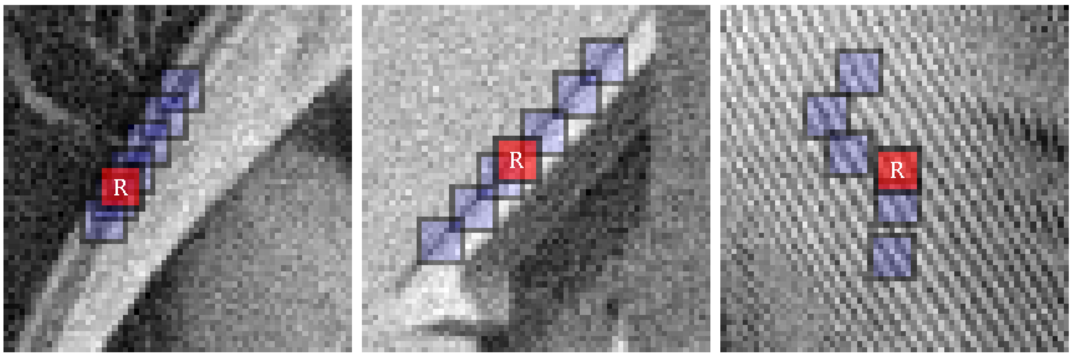

Figure 2.3: Block-matching with overlapping patches, Source:[3]

To define a similarity between patches, first we have to define a distance d between two arbitrary patches P and Q. We use the normalized squared L_2 distance:

$$d(P, Q) = \frac{\|P - Q\|_2^2}{(k)^2}$$

where $| \cdot |_2$ is L_2 norm. Patches P and Q are considered similar if the distance between them is lower than threshold τ.

$$d(P, Q) \leq \tau$$

Because the image is affected by the presence of the noise, for images where this noise is expected to have the variance $\sigma > 40$, the original paper extends the distance definition. It compares coarsely denoised patches instead of the original noisy ones. To do so, it employs simple hard thresholding in transform basis on every patch.

Patches similar to the reference patch P are grouped into a three dimensional array $\mathcal{P}(P)$:

$$\mathcal{P}(P) = \{Q : d(P, Q) \leq \tau\}$$

We can notice that one patch of the image can be in more than one array.

Due to the performance issues, only the N most similar patches to the reference patch are kept in the array. However, if we restrict the number of patches in 3D group, we have to ensure, that each 3D group contains at least the reference patch. This is important for the Aggregation part.

2.4.2 Collaborative Filtering

Collaborative filtering is a special procedure developed for filtering of the 3D groups, that were obtained from the grouping step. Standard transform based image filtering employs 2D transforms. Collaborative filtering, on the other hand, employs 3D transform and filters the coefficients of the whole 3D array at once. Due to the similarity between the patches, the 3D transform results in even sparser representation of the original patches than the 2D transforms whereas the noise still has a constant power spectrum. Hence, more details of the original image are preserved.

Colaborative filtering consists of the following steps.

1. Apply the 3D transform τ_{3D} to the 3D group.

2. Filter the coefficients by hard-thresholding to attenuate the noise.

3. Apply the inverse 3D transform τ_{3D}^{-1} to obtain estimates for all patches in the group.

This procedure is expressed by the following formula:

$$\mathbb{P}(P) = \tau_{3D}^{-1}(\gamma(\tau_{3D}(\mathcal{P}(P))))$$

where γ is a hard thresholding operator with threshold $\lambda_{3D}\sigma$:

$$\gamma(x) = \begin{cases} 0 & \text{if } x \leq \lambda_{3D}\sigma \\ x & \text{otherwise} \end{cases}$$

2.4.3 Aggregation

Once the collaborative filtering produces all the patch estimates, the Aggregation procedure returns the denoised patches to their original locations and compute an average of the overlapping pixels.

During the block-matching one patch could have gotten into more than one 3D group; therefore, the collaborative filtering can produce different estimates of one patch. Moreover, patches can overlap. Consequently, there can be many various pixel estimates for each pixel. In order to obtain a final image estimate, we have to aggregate all those patches into a single 2D image.

Aggregation will result in continuous image if and only if we obtain estimate for each pixel. This is ensured, because each 3D group contains at least the reference patch.

The final image estimate is computed as an weighted average of all overlapping pixels. Weight of each pixel is defined by homogenity of its patch. Homogeneous patches are prioritized over the patches containing edges and corners. The main objective is to avoid the ringing effect around the edges, which is typical for transform based denoising methods.

Homogeneous patches have much more sparse representation after the transform than the patches with edges; thus, homogeneous patches have more coefficients that do not pass the threshold in the filtering step. Therefore, we can use the number of coefficients that pass the threshold as a metric of homogeneity.

Consequently, aggregation uses an inverse number of retained (non-zero) coefficients of a 3D group after hard thresholding as a weight to each pixel. If we define the number of non-zero coefficients of 3D group $\mathbb{P}(P)$ as N_P, we use the following weight:

$$w_P \begin{cases} (N_P)^{-1} & \text{if } N_P \geq 1 \\ 1 & \text{otherwise} \end{cases}$$

To reveal the aggregation equation, we define indicator function X, which indicates if pixel x is contained in patch Q.

$$X_Q(x) \begin{cases} 1 & \text{if } x \in Q \\ 0 & \text{otherwise} \end{cases}$$

Also we define $u_{Q,P}(x)$ as the value of the pixel x belonging to the patch Q belonging to the denoised stack $\mathbb{P}(P)$

Each pixel x of the final image estimate u is then computed as:

$$u(x) = \frac{\sum\limits_{P} w_p \sum\limits_{Q \in \mathbb{P}(P)} X_Q(x) u_{Q,P}(x)}{\sum\limits_{P} w_p \sum\limits_{Q \in \mathbb{P}(P)} X_Q(x)}$$

2.5 Optimal Parameters for BM3D

The denoising performance of BM3D is controlled by several parameters, which were mentioned in the description of the algorithm. We are referring to the optimal values as they were measured in analysis of BM3D algorithm provided by Marc Lebrun [14].

- k defines the width and height of a patch. The optimal value slightly depends on the variance of the noise in the image. Nevertheless, the best results are generally obtained if we fix this number to 8.

- n specifies width and height of area around reference patch in which similar patches are searched. This parameter has a little influence on the final result if we choose reasonable values for it. The authors has chosen as optimal value of 39.

- N stands for a maximal number of patches in 3D groups. It more affects the processing-time of an algorithm rather than the denoising effectivity. Reasonable values of N has almost no influence on the result. The optimal value in the terms of efficiency and effectivity is 16.

- τ is a parameter specifying the threshold under which two blocks are assumed similar. Its optimal value is highly dependant on the noise variance σ^2. For $\sigma < 40$ the optimal value is 2500. For larger values of σ the algorithm compares coarsely denoised patches and therefore the threshold is higher. It is proven that optimal value of τ for this case is 5000

- λ_{3D} is one of the most important parameters in terms of denoising performance. It defines the threshold under which the coefficients of image in transform basis are set to zero. Its optimal value is 2.7 and even small variations of it have strong influence on the result.

Although in this thesis we are concerned only with DFT as τ_{3D} transforms, we should mention that it not the most optimal choice. More optimal would be DCT (Discrete Cosine Transform), which can be computed using DFT on symmetric data.

The 3D transforms we mentioned are separable, they can be computed as 2D transforms followed by 1D transforms in the remaining dimension. We can also combine different transforms for each dimension. Moreover, studies have proven that the most optimal transform combination for BM3D is 2D Bior1.5 transform followed by 1D Hadamard transform.

Both the original paper [3] and the analysis by Marc Lebrun [14] suggest that it is not necessary to use every patch in the image as a reference patch. For that purpose they present a parameter p, that denotes the step with which the reference patches are chosen. Its optimal value is 3. This parameter has a great influence on memory consumption and computational complexity.

Even with the employment of the p parameter, the BM3D algorithm remains computationally intensive. In block-matching, each reference patch still has to be compared with all the patches in its neighbourhood. However, the complexity of block-matching can be reduced by reusing already computed distances. We present this approach among with the implementation in the next chapter.

3. Graphics Card Architecture and Programming Model

Graphics cards are designed for compute-intensive, highly parallel computations. For that purpose they contains thousands of cores and high bandwidth memories. This design arises from the original purpose of the graphic card – graphic rendering. Graphic cards are well-suited for problems that can be expressed as data-parallel computations, where a same routine (usually denoted as *kernel*) is executed on many data elements in parallel.

To be able to design any high performance algorithm it is necessary to know at least basic aspects of the target platform. Therefore, in this chapter we will present the architecture of the current graphics cards. Then, we shall present the programming model which we use for GPU-accelerated implementation of the BM3D method.

3.1 Graphics Card Architecture

There are two main producers of graphics card designs – NVIDIA and AMD. Because we have only NVIDIA cards at disposal, all this chapter is concerned with the NVIDIA cards. We will cover only the basic aspects that are essential for developing the GPU-accelerated BM3D algorithm.

Even though all graphics cards are similar in nature, each new generation have some more or less important differences in the architecture. For better orientation in these architecture differences, NVIDIA present a concept of *compute capabilities*. It is a number which define features that are supported by a particular graphic card as well as the technical specifications such as memory sizes, number of registers, system of caching, etc. If we refer to the contemporary GPUs in this thesis, we mean the GPUs with Maxwell architecture [18] (compute capability 5.x).

3.1.1 CPU and GPU Communication

Since the graphic card is a separate device commonly connected with the host system by the PCIe bus, we will refer to it as the *device*. The host offloads certain computing tasks to the device. The offloading covers both, the data transfers and the computation itself.

The device has its own dedicated memory which is not directly accessible by the host CPU. However, the access to this memory is granted through a certain set of instructions. These instructions offers the host to allocate a space for data in this memory as well as transfer the data from the host to this memory and vice-versa. Nevertheless, due to the limited bandwidth of the bus between the host and the device, it is common to minimize these transfers, or overlap them with computations. A typical dedication of work to a device then looks like this:

1. Allocate the device memory for all the data (input, output and intermediate)

2. Transfer the input data from the host to the device
3. Launch a kernel on the device and wait until it finishes.
4. Transfer the output data to the host

The above mentioned practice is not always the most optimal one. For large input data, it is often better to partition this data, transfer each part to the device separately and launch kernel on it. Consequently, the kernel execution on one part can overlap with the data transfers of other parts, which can significantly speed up the algorithm. For the kernels with relatively short execution time and larger data, this approach may even triple the speed of the algorithm. However, for compute intensive kernels and small data, the time savings could be negligible.

Once the host launches the kernel on the device it can do some other work or it can enforce synchronization and wait for the device to complete the kernel. However, host can not interfere with the ongoing computation on the device.

3.1.2 GPU Architecture

GPU contains several independent processors called streaming multiprocessors (SM). Each SM contains many simple cores which share registers, L1 cache, and *shared memory.* Figure 3.1 shows the design of the current NVIDIA SM. Among the simple cores there are Load/Store units for memory access and special function units (SFUs) for calculation of functions as sin, cos, or log.

Threads that execute the kernel are launched in blocks. Each thread block is dynamically, but non-preemptively, assigned to one of the SMs and executed. Since each SM is independent of others, there is no way how to safely synchronize the computation among blocks. On the other hand, threads inside one block can be synchronized, which is important ability for many cooperative parallel algorithms.

There could be more than one block assigned to one SM, but that number is strongly limited by available registers and shared memory of this SM. Moreover, different SMs can process blocks of different kernels at the same time.

SM partitions the threads of the assigned blocks into *warps.* So far each warp has consisted of 32 threads. Threads of one warp work in a *lock-step* mode. That means that they always executes the same instruction at one time. This execution model is called a SIMT (Single Instruction Multiple Threads) model.

The SIMT model is very sensitive to branching. The proper execution path is selected based on the result of a condition that is evaluated for each thread separately. Each thread of a warp can select different execution path, but because the threads of a warp still have to execute the same instructions, SIMT model presents a concept of instruction masking. All threads of a warp serially process all the paths that were selected by at least one of these threads. The threads that did not select the currently processed execution path are masked.

Inappropriate branching in a kernel may lead to poor utilization of the GPU cores. Hence, there should be minimal differences in lengths of *for* or *while* loops between threads of one warp. Also, in the case of the *if/else* statements, we should be aware, that if the threads of a warp diverge on the execution path, both of these paths will be consecutively processed by all the threads.

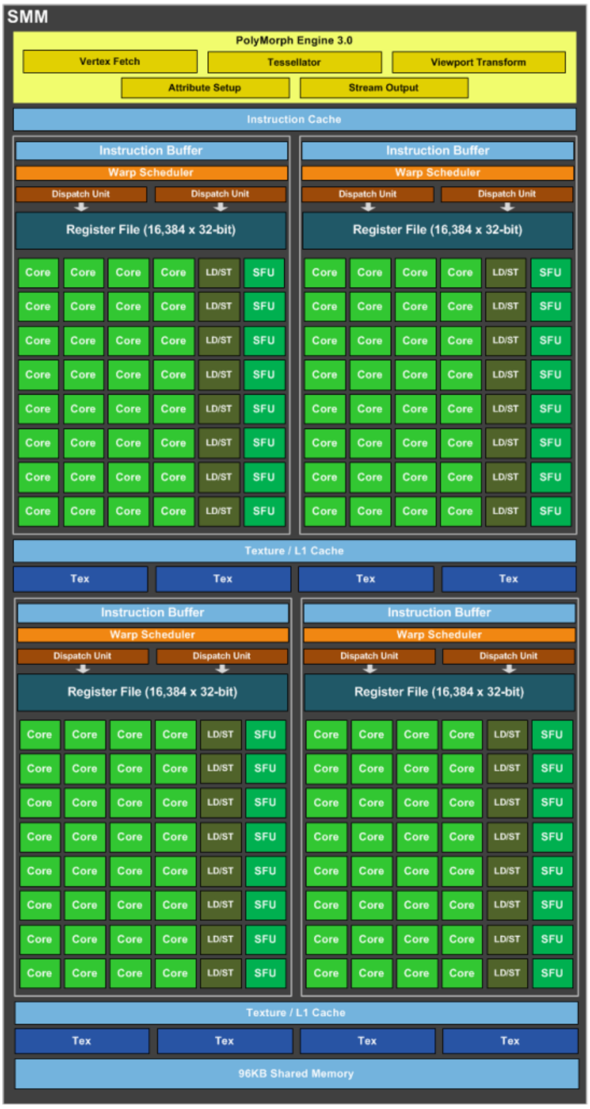

Figure 3.1: Streaming multiprocessor of NVIDIA Maxwell architecture, Source: NVIDIA

When an SM is given a thread block to execute, the SM partitions the block into warps and allocates registers for each warp. Subsequently, the warps get scheduled by *warp-schedulers* for execution. Warp-scheduler selects the warps whose threads are ready to be executed (which are not waiting for any data) and dispatches the instruction which is going to be executed to the physical cores of an SM.

The execution context of each warp is maintained on the SM; therefore, *context switch* has no cost. Consequently, thread block should consist of many threads in order to efficiently overcome data-dependency stalls (memory reads and writes) or stalls caused by other time-intensive operations.

Since the programmer is responsible for defining the sizes of the thread blocks, it is necessary to know the approximate number of cores, SMs, etc. We can see the numbers for last two generations of NVIDIA GPUs in the table 3.1

3.1.3 Memory Hierarchy

There are several types of memory on a device.

GPU	GeForce GTX 680 (Kepler GK104)	GeForce GTX 980 (Maxwell GM204)
CUDA Cores	1536	2048
Base Clock	1006 MHz	1126 MHz
Compute Capability	3.0	5.2
SMs	8	16
Shared Memory / SM	48KB	96KB
Register File Size / SM	256KB	256KB
Active Blocks / SM	16	32
Memory	2048MB	4096MB
Memory Bandwidth	192.3 GB/sec	224.3 GB/sec
L2 Cache Size	512KB	2048KB

Table 3.1: Hardware specifications of Kepler and Maxwell GPU architectures

- **Global memory** resides on the device outside of the GPU chip. It is designed to operate with huge data and therefore a great emphasis is placed on the bandwidth rather than on latency. The bandwidth of the current global memories is about five times higher than on high-end host RAMs while the latency can reach 400-600 cycles.

- **Registers** are used for operands of instructions. They also store thread-local variables. Registers have almost no latency, but their number is limited. Contemporary SMs contains 64K of 32-bit registers.

  When a block requires more registers than the SM offers, the registers are spilled into much slower *local memory*. Therefore, the blocks should not generally require more registers than SM offers.

- **Shared memory** is located on each SM and it is directly managed by the threads running on that SM. Shared memory is almost as fast as the registers and its size is also very limited. Contemporary SMs have 96KB of shared memory.

  Because shared memory resides on an SM it is designed for concurrent access by threads of a warp. For that purpose, it is divided into 32 equally sized memory modules called banks that can be accessed simultaneously. Each bank has a bandwidth of 32 bits per clock cycle. Every shared memory address is mapped to a particular bank in such manner that consecutive 32-bit words are mapped to the successive banks.

  Whenever multiple threads want to access different memory addresses that fall into the same bank at the same time, *bank conflict* occurs and the access has to be serialized. The only exception occurs when threads of one warp access the same address. In the case of read access the value is broadcasted and in the case of write access only one of the threads writes the value. In both cases no serialization occurs.

  Shared memory combined with a barrier synchronization allows the threads of a block quite efficiently cooperate on the computation. Nevertheless, it is good to use access patterns that minimize bank conflicts.

- **Local memory** physically resides in the thread-private area in the global memory. Therefore, it has the same bandwidth and latency as the global memory. It is used for the register spilling or for large thread-local structures and arrays.

- **Caches** speeds up the access to the global memory. The most important caches of the contemporary (Maxwell) GPUs are:
  - **Unified L1/texture cache** is in each SM. It has 64KB and it is used to cache the data in global memory, that are read-only during the entire lifetime of a kernel.
  - **L2 cache** is shared by all SMs of a device and it is used for caching the access to the global memory (including the local memory). It is about 2MB large.

3.2 Programming Model

In this section we will briefly present how to write, compile, and launch a program accelerated with a graphics card. For more information about the GPU programming we refer to the 'Programming Massively Parallel Processors' [19].

First of all, we have to choose the convenient GPGPU platform. The three most popular are: Direct Compute, OpenCL [20], and CUDA [21].

- **Direct Compute** is a part of Microsoft DirectX 11 API and it is mainly used for acceleration of non-graphic computations in graphic applications that uses the DirectX. It uses a language that is similar to HLSL, which is used for programming of shaders. Unfortunately, the usage of Direct Compute is limited to the Windows operating systems only.

- **OpenCL** is cross-platform and very complex framework. OpenCL code can run on many parallel devices including the graphic cards. Because the OpenCL standard is open, implementations for other devices can follow. Even though the code can run on many GPU architectures, if we tune the performance, we shall need a slightly different code for different architectures. Moreover, due to the OpenCL universality, programmer often can not use architecture-specific technologies.

- **CUDA** (Compute Unified Device Architecture) is a product of NVIDIA and it is bound to their graphics cards only. The absence of diversity among producers allows us to use architecture-specific features. Nevertheless, even the graphic cards from the same vendor may differ and a code tuned for one NVIDIA device might not be the best one for an another NIVIDA device.

We have choosen the CUDA which allows us to use architecture-specific abilities of the recent NVIDIA graphics cards. CUDA has also very intuitive syntax from which we can benefit when we will present our prototype solution. Nevertheless, the transformation from CUDA to OpenCL is not generally difficult.

The source files for CUDA applications are mixture of conventional C/C++ host codes and CUDA C/C++ device codes (kernels). The host code uses CUDA runtime library to select the device, to launch kernels on the device, to allocate the global memory, to transfer data between the host and device, etc.

During the compilation, the host code is compiled to a binary code and the device code is pre-compiled into an intermediate code common for many graphic card architectures or into a binary code for one particular architecture. This code is then embedded into the host binary code.

3.2.1 Kernel

Kernel is a routine which code is executed on the GPU by many threads in parallel. CUDA kernels are defined as normal C/C++ functions without a return value. They are introduced by `__global__`keyword.

Due to the architecture of the GPU, kernels are launched in equally sized blocks of threads, which can be executed in any order or even simultaneously. These blocks form a structure we call a *grid*. The size of the grid as well as the size of the blocks is specified when the kernel is launched; however, the kernel should be written regardless the concrete sizes.

Each thread that executes the kernel can access its thread and block index through the built-in `threadIdx` and `blockIdx` variables. Threads in a block as well as blocks in a grid can be organized in up to three dimensions. The sizes of these dimensions are available through the `blockDim` and `gridDim` variables. These sizes and indexes are essential for data parallelism. Each thread is able to get the unique data element from the memory based on this unique set of indexes. Figure 3.2 shows an example of how the threads and blocks can be organized in the case of 2 dimensions.

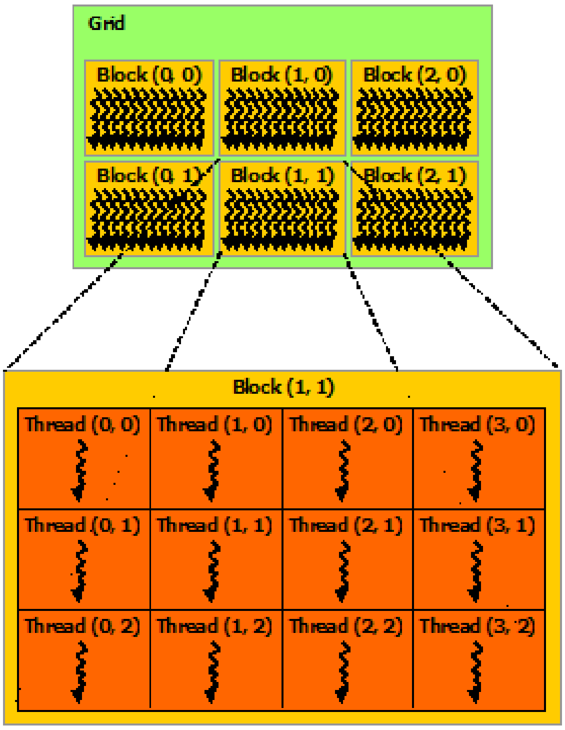

Figure 3.2: 2D Grid of 2D blocks of threads, Source: NVIDIA

In most cases the division into blocks and threads is based on the data structure. The possibility of having up to three dimensional grids and blocks provides a natural way to invoke computations across the elements of vector, matrix or volume.

When we are dividing the problem into threads and blocks, we have to take into account some limitations arising from the hardware architecture. Maximal number of threads in a block on current devices is 1024 regardless of the dimensionality and the number of blocks in grid is limited by 65535 in each dimension.

To launch a kernel CUDA uses special syntax which allows us to set size of a grid and the size of a block:

```
kernel_name<<<grid size, block size, shared memory size
   >>>(arguments)
```

The last parameter is optional and it defines the number of bytes of shared memory each block want to use. We will say more about the shared memory in 3.2.5.

Figure 3.2.1 presents an example of definition and launch of a kernel doing vector addition.

```
__global__ void VectorAdd(float *a,
                          float *b,
                          float *result,
                          int size) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < size) { // friendly branching
    result[i] = a[i] * b[i];
  }
}

int main() {
  ...
  VectorAdd<<<(N+255)/256,256>>>(a,b,result,N);
  ...
}
```

Figure 3.3: Example of definition and launch of the kernel doing vector addition

On operating systems started with GUI (graphic user interface), the graphics driver may restrict the maximal execution time of a kernel. Kernels that run longer end with error. Therefore, we are often forced to partition the work into multiple shorter kernels.

3.2.2 Warps

Threads in a block are divided into warps by their linear id which is computed as:

```
threadIdx.z * (blockSize.x*blockSize.y) + threadIdx.y *
    blockSize.y + threadIdx.x;
```

Each successive `warpSize` threads form one warp. The `warpSize` is an architecture-specific built-in variable; nevertheless, so far, a warp has been always formed by 32 threads. We know that the threads in a warp executes code in the lock-step mode and that we should avoid code divergence mentioned in 3.1.2.

3.2.3 Function Calls Inside a Kernel

A kernel can call other functions. These function has to be marked with `__device__` keyword and no other special syntax is needed. However, because the standard function call would be an expensive operation for the GPU, these function calls are generally inlined whenever possible.

Beside the standard functions, a kernel can call many intrinsic function. Among many arithmetic operations, these functions includes synchronization primitives, warp shuffle functions, and many others.

In the implementation of BM3D we shall use the following functions:

- **Atomic operations** handle writing conflicts in shared or global memory by serializing the access. If multiple threads want to simultaneously read, modify and write a value at the same memory address, they are serialized and no conflict occurs. Atomic operations allows us to use random access patterns for accessing both those memories. On the other hand, these operations are slower than their non-atomic counterparts and we should use them wisely.

- **Thread synchronization** across all threads of a block can be enforced by kernel through a lightweight function `__syncthreads()`. It is a barrier that can be passed by a thread only after it is reached by all threads of a block. This barrier combined with the shared memory allows the threads of a block to cooperate on computation.

- **Warp shuffle functions** are used for direct data exchange between threads of a warp. These functions are faster then the access to the shared memory, but they are available only since the compute capability 3.0.

3.2.4 Host Device Communication

We have already described how to launch a kernel in 3.2.1. Each kernel is launched with special parameters as block and grid size as well as with standard C/C++ arguments. These arguments are often device pointers to arrays in global memory and sizes of these arrays.

Generally, host communicates with the device using the CUDA runtime. Even the kernel calls are during the compilation transformed into the runtime calls. Beside that, CUDA runtime offers basic functions for utilization and management of the device. Primarily:

- **Device properties and device selection:** In order to get any information about hardware architecture of the device at runtime, we should call a `cudaGetDeviceProperties` function. Among the properties there are for example: shared memory size, warp size, maximal block size, etc. These properties may be useful for run-time adjustments of kernels.

There can be multiple devices available on the host. However, since most of end-user computers have only one GPU we will not discuss the selection of the most suitable device nor the problems associated with multi-GPU programming. The CUDA runtime automatically selects the first available device which is then used for all runtime calls.

- **Memory allocation:** Generally, the data is placed in global memory. In order to place them there, first, we have to allocate a space for them and obtain a pointer to this space.

  To allocate a space in global memory CUDA offers `cudaMalloc` function which has the same semantics as standard C `malloc`. The only difference is, that the allocated space resides in the global memory of the selected device and the returned pointer targets the global memory address space. This pointer looks as any other pointer; however, it can be used only in kernels or in several CUDA Runtime functions. Any dereference of it on the host would led to an undefined behaviour. It is common to mark variables containing these device pointers with some specific prefix. In the prototype implementation we use `d_` prefix.

  Any memory allocated by `cudaMalloc` has to be properly deallocated using a `cudaFree` function.

- **Data copying:** To copy data from host memory to device memory or vice-versa CUDA runtime offers a `cudaMemcpy` function. In order to use this function we have to provide the following arguments: a pointer to the device memory, a pointer to the host memory, size of data, and direction of the copying (device to host or host to device).

3.2.5 Utilization of Device Memories

The device has several different memories. There is a global memory, shared memory, registers and several caches. While caches and registers are managed by compiler or by hardware itself, the global and shared memory is managed by a programmer.

The global memory is designed for massive data movement rather than for fast access; therefore, it is generally used for large arrays. When dealing with arrays stored in global memory, we may rely on much faster L2 cache, but as with all caches its benefit depends on the access patterns that our program uses. In order to use even faster unified L1/texture cache for global memory reads, the array we want to access has to be read-only for the entire lifetime of a kernel. However, the compiler might not be able to detect, that the array is read-only. To increase the likelihood that the compiler detects that, we mark pointers with both `const` and `__restrict__` qualifiers. `__restrict__` keyword makes a promise to the compiler, that the pointed array will not be accessed through any other pointer.

Another type of memory which is exposed to a programmer is *shared memory*. It is a block-local low-latency memory. A variable of a kernel is shared among the threads of a block when it is marked with `__shared__` qualifier. Its value is then stored in the shared memory. To allocate an array in the shared memory, we can use either static or dynamic allocation. If we know the size of the array at compile time, we can use traditional static C allocation; otherwise, we have to use the dynamic allocation. Array which is the subject of dynamic allocation is marked with both `extern` and `__shared__` qualifiers and its size is defined by the third kernel launch parameter. Unfortunately this approach allows only one dynamically allocated array per kernel; hence, for multiple arrays we have to use pointer arithmetic and casting functionality to divide that one array into multiple different arrays.

The shared memory can be used for data-exchange between the threads of a block, as a programmer-managed cache, etc. When dealing with the shared memory, we have to avoid race conditions. To do so, we can use either the atomic operations or the `__syncthreads()` barrier. Moreover, we have to be aware of the limited size of the shared memory.

Finally, data stored in standard thread-local variables are likely to be stored in registers. In the case of lack of registers we can encounter the *register spilling*, where the data in registers are moved to the *local memory* (thread-local area in the global memory). In order to avoid the register spilling, we should limit the number of variables the kernel uses with respect to the target architecture.

3.2.6 Compilation

To compile a CUDA code we do not use a standard C/C++ compiler. To compile the code containing kernel calls or kernel definitions we employ the CUDA compiler `nvcc`.The compilation itself works as follows. The CUDA code is preprocessed and separated into a device code (kernels) and a host code. The device code is compiled/assembled for target platforms and the result is later embedded into the final binary code. In the host code, all the kernel calls are replaced by more complicated C/C++ CUDA runtime stubs. Since the host code then does not contain any CUDA specific syntax it is compiled by the default C/C++ compiler (eg. gcc or msvc).

Because nvcc does not support all C/C++ compilers, the common practice is to separate a CUDA code from the rest of an application as much as possible. That often results in a scheme, where all kernel calls are wrapped in `extern C` functions. The CUDA code then contains only kernel definitions and kernel calls and therefore the host code can be compiled by the desired compiler independently of the `nvcc`.

Since the device code could use various compute capability dependent functions, we have to provide the `nvcc` compiler an information about the compute capabilities of the GPUs which should be able to run our code. For example, the compilation of code that contains warp shuffle functions should fail for devices with compute capability 2.0, because these functions are defined only for compute capability 3.0 and higher.

The `nvcc` distinguishes two ways of compilation of a kernel. It can be translated into an architecture-specific CUDA binary (cubin) code or into an intermediate PTX code. While the cubin code can be immediately executed by the proper GPU, the PTX code is first compiled by the CUDA runtime into the cubin whenever it is going to be launched and only then it is executed.

To select the proper type of compilation of the device code, the nvcc offers two options `-arch` and `-code`. Option `-arch` defines the target virtual architecture, which only specifies the compute capability that the target GPU has to support. Values for the virtual architectures start with `compute_` followed by the number identifying the target compute capability. Code is then translated into the PTX code and due to the backward compatibility of compute capabilities it can be run on all GPUs with this or higher compute capability.

Through the `-code` option we may specify the real architectures for which the cubin code should be generated directly during compilation. Values for the real architectures start with `sm_` followed by the number identifying the architecture of the GPU. Code is then compiled directly to the cubin code of the specified architectures. We may specify more of them, but we can not select a real architecture that do not support the compute capability defined in `-arch`.

4. Implementation

4.1 Analysis

The BM3D algorithm, presented it in the section 2.4, consists of three parts: *grouping*, *collaborative filtering*, and *aggregation*. For each reference patch of an input image, the grouping finds similar patches and stacks them into a 3D group. The collaborative filtering denoise these groups in a transform domain by hard-filtering. The patch estimates in the filtered groups are then aggregated into the final image estimate at their former locations. The whole process is shown in Figure 4.1.

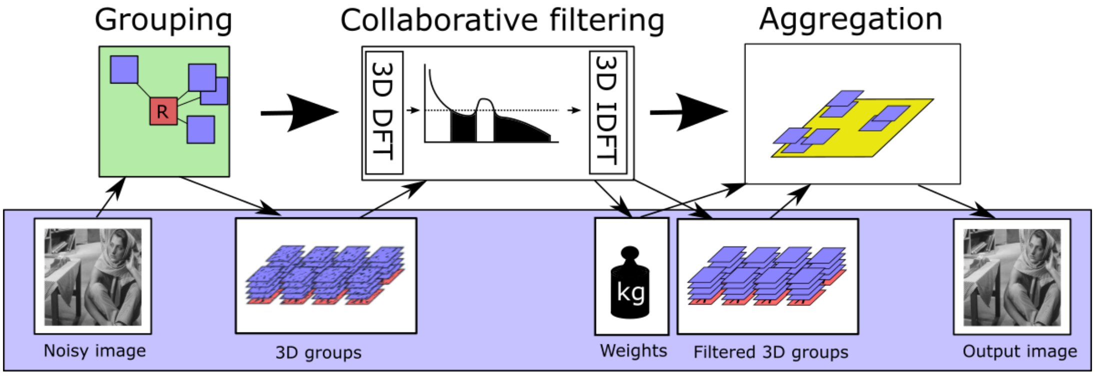

Figure 4.1: BM3D method

Before we will proceed to the grouping part, first, we have to define which patches are the reference patches. This depends on the p parameter, which defines the space (in pixels) between two adjacent reference patches in horizontal or vertical direction. If $p = 1$, all patches of an image are marked as reference patches.

We have to ensure, that whole image is covered by reference patches. If the parameter p is set to a value larger than 1, it may happen that some pixels near the borders of the image would not be covered by any patch at all. This would resulted in black spots in the image estimate. To solve this problem we mark all patches on the border of the image as reference patches, regardless the value of p.

Grouping uses block-matching algorithm to find for each reference patch N the most similar patches that pass the threshold. The search area is limited by $n \cdot n$ window centred at the location of this patch. This is an opportunity for data-parallelism, because each reference patch could be processed independently of others. Because we have to compute L_2 distances between each reference patch and all patches in its search window, block-matching is computationally very intensive.

For future extensions of the algorithm, the grouping will not return assembled 3D groups but only coordinates of the patches in these groups. The collaborative filtering, at beginning, assemble the 3D groups according to these coordinates. In the case of extending the algorithm to process color images, it allows us to find the similar patches according to luminance channel only and then build the 3D groups based on this matching for all channels.

Collaborative filtering assembles the 3D groups according to the coordinates and transform them using 3D (or 2D followed by 1D) DFT or other decorrelating transform. Transformed patches are then filtered using hard-thresholding. During this part we shall count the number of retained (non-zero) coefficients for each group. From this value we compute the weight of the 3D group, which is used later in the aggregation part. Finally, the filtered groups are transformed back using the inverse transforms.

The collaborative filtering exposes many opportunities for data-parallelism. The 3D groups can be processed independently. Beside that, the DFT and IDFT can be computed using Fast Fourier Transform (FFT) algorithm which is well-suited for the GPU platform. Moreover, the transformed coefficients in a 3D group can be also filtered independently.

Aggregation returns patch estimates to their original locations and computes weighted averages of all overlapping pixels of all overlapping patches. Each pixel of a patch estimate is aggregated with the weight of the 3D group from which it was obtained. In order to process patches in any order we divide the aggregation into two steps. First, we fill two buffers: the *numerator* $\boldsymbol{v}$ and the *denominator* $\boldsymbol{\delta}$. Each of them is used to store the appropriate part of the former aggregation fraction:

$$\forall Q \in \mathbb{P}(P), \forall x \in Q \begin{cases} \boldsymbol{v}(x) = \boldsymbol{v}(x) + w_P u_{Q,P}(x) \\ \boldsymbol{\delta}(x) = \boldsymbol{\delta}(x) + w_P \end{cases}$$

Second, when all the patches are aggregated into v and δ buffers, we obtain the image estimate as a simple element-wise division of these two buffers. Consequently, if we handle the writing conflicts when aggregating overlapping patches, we can process every patch independently.

Computing each part of the BM3D method on the entire image before continuing to the next successive part demands extreme amount of memory. For example, the collaborative filtering can produce up to N patch estimates for each reference patch. Assuming the optimal parameters as they were presented in 2.5 and that we can have almost as many reference patches as pixels in image, solely on storing all these patch estimates, we would have to allocate about thousand times more bytes of memory than we use for storing the image.

It is true that the number of concurrently kept 3D groups is significantly reduced by the parameter p. Unfortunately, the step p has to be significantly lower than size of a patch and therefore this reduction may not be sufficient for larger images.

The entire algorithm (with except of the final division in the aggregation part) can be done for each reference patch and its 3D group separately. Hence, we can reduce the memory consumption by iterative computation, where in each iteration only a reasonable number of reference patches are processed.

In summary, the BM3D method exposes many opportunities for data-parallelism. The reference patches can be processed concurrently and the individual operations done on each reference patch or the 3D group (such as computing the distances, 3D transforms, etc.) can be computed in parallel. However, the parallel algorithm have to deal with writing conflicts in aggregation part and with the memory intensity.

Because the BM3D method offers data-parallelism on various levels, there are many ways how to design the algorithm and divide its parts into blocks and threads of a GPU. We have empirically tested several approaches and selected a suitable algorithm for current GPU.

4.2 Overall Architecture

We present a GPU accelerated implementation of BM3D. Due to the limited bus bandwidth between a host and device, we propose an implementation, where all the data remain in the device memory throughout the computation. Therefore, only the input and output images are transferred through the bus.

This implementation is focused on common high-resolution grayscale images. These images have up to 20Mpx (mega pixels); therefore, in traditional byte representation, they does not occupy more than 20MB. Because the size of global memory of current devices is in the order of GBs, we can afford to keep the whole input and output image in global memory throughout the computation.

Because the auxiliary arrays needed for storing intermediate results demand lot of memory and because of the computational complexity of the block-matching (see the section 4.1), we propose to process the image in batches. The image is divided into virtual rectangles of reasonable size. The reference patches inside one rectangle form one batch. The space needed for the intermediate results of each batch is then limited by the size of the batches. Also it reduces processing-time of kernels, that would be otherwise applied to the entire image.

The batches are processed serially; however, the reference patches or their 3D groups in a single batch are processed in parallel. We have empirically chosen the batch size to be $256 \cdot 128$. Batches of this size can still utilize all GPU cores and the intermediate results of the batch can fit the device memory. Nevertheless, we can change the batch size if we find out the proposed values insufficient in the future.

Because the batches are compact, there is higher possibility that during the computation of a batch, the appropriate part of the image and other data will remain in caches. For each reference patch of a batch the similar patches are found, assembled into 3D groups, collaboratively filtered and aggregated.

The algorithm needs several auxiliary arrays that are used to store the intermediate results:

- Coordinates of patches in 3D groups
- Sizes of 3D groups
- Assembled 3D groups
- Weights of 3D groups
- Numerator
- Denominator

All the auxiliary buffers have necessary size for one batch and they are reused by all the batches. The only exception are the numerator and denominator buffers which are of the same dimensions as the image estimate and which are iteratively filled by the aggregation part of every batch.

We may notice that 3D groups of a batch can consist of patches originating outside the virtual rectangle of this batch. However, accessing patches located outside the rectangle is possible because the entire noisy image is stored in the global memory throughout the computation.

We use the following scheme of processing:

1. Noisy image is copied to the device memory
2. Reference patches are divided into batches
3. For each batch the 3 parts of the BM3D method are performed:
    - **Block-matching** produces an array of the coordinates of the patches similar to each reference patch of a batch. These arrays are stored in the coordinates buffer.
    - **Collaborative filtering** assembles 3D groups according to the arrays of coordinates and filters them. Collaborative filtering produces filtered 3D groups and weights of these groups.
    - **Aggregation** takes the patch estimates in the filtered 3D groups and aggregates them to the numerator and denominator buffers at their original locations according to the arrays of coordinates and weights of the 3D groups.
4. The numerator buffer is divided element by element by denominator buffer.
5. The result (image estimate) is copied back to the host.

A detailed overview of buffers used in parts of BM3D is shown in Figure 4.2

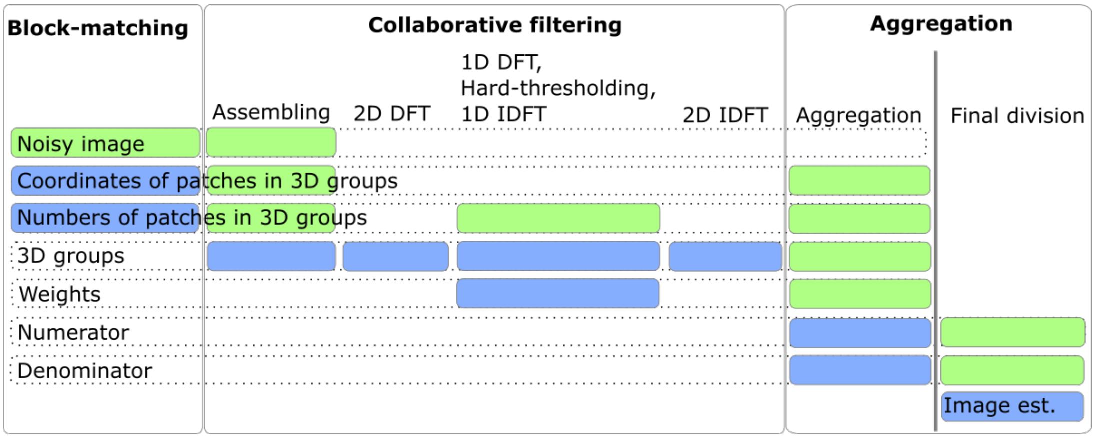

Figure 4.2: Buffers used in different parts of BM3D and its kernels. Green color marks that buffer is read-only in the particular kernel.

We have implemented the block-matching in a single kernel. It is computationally intensive; nevertheless, by batch processing, we have prevented the problem of terminating long-running kernels.

While the block-matching could be written in one kernel, for collaborative filtering we propose a more complicated scheme. Mainly because we want to employ an existing efficient CUDA implementation of FFT (Fast Fourier Transform) which is contained in CUDA toolkit, the CUFFT [22]. Its functions can be launched from host only; therefore, we have to divide collaborative filtering into two kernels. One for assembling 3D groups and one for filtering the transformed patches.

4.3 Block-Matching

For each reference patch of a noisy image the block-matching algorithm finds similar patches located in the $n \cdot n$ search window centred around the reference patch. Similarity is measured using the normalized quadratic L_2 distance. Similar patches are those which has lower distance than threshold τ. As a result, the block-matching produces N the most similar patches for each reference patch.

We have implemented the block-matching algorithm in a single kernel. Each block processes `warpSize` reference patches successive in horizontal direction. Thus, we need roughly 32 times less blocks then reference patches in a batch. Each thread operates on one reference patch; however one reference patch is processed by more threads which cooperate on the result. Threads of a single block which have the same `laneId` (index of a thread within a warp) and different `warpId` (index of a warp within a block) process the same reference patch.

Each thread searches portion of the $n \cdot n$ search window of its reference patch and calculates the quadratic L_2 distance to the patches in this portion. Threads that handle the same reference patch alternates when traversing the search window. If a thread with `laneId` $= 0$ and `warpId` $= 0$ calculates a distance to the patch displaced from the reference patch by $(-5, -5)$, thread with `laneId` $= 0$ and `warpId` $= 1$ would calculate the distance to the patch displaced by $(-4, -5)$ and so on. Thus, the more is the warps in a group, the less distances each thread has to compute.

When a thread computes the distance to some patch it checks if it pass the threshold. If it does, it stores the coordinates of this patch and the computed distance to the array that we denote `stack`. This array can store up to N values for each reference patch; therefore, the insertion of a new value to this array is a non-trivial operation. Consequently, in order to avoid writing conflicts, each thread has a local copy of this array. Due to its size, we can store all the copies in the shared memory. Consequently, when all distances in the search window are computed, we have to merge the `stack` arrays of the threads which processed the same reference patch. This merged array is then saved to the global memory as a result of the block-matching part.

4.3.1 Cooperative Calucation of Distances

The reference patches overlaps; therefore, we can share some portions of the computation of distances to the patches displaced form these reference patches by the same value. We can notice, that the area covered by these reference patches and the area covered by the displaced patches is exactly of the same size. We share the computation for several reference patches successive in the horizontal direction.

To compute the quadratic L_2 distances, we first compute the quadratic L_2 distances between corresponding pixels of the area covered by the reference patches and the area covered by the displaced patches. Then we sum the pixel distances in columns to obtain the column distances. Finally, we obtain each distance by summing the column distances of the columns corresponding to the reference patch. A visualisation of this computation is presented in Figure 4.3

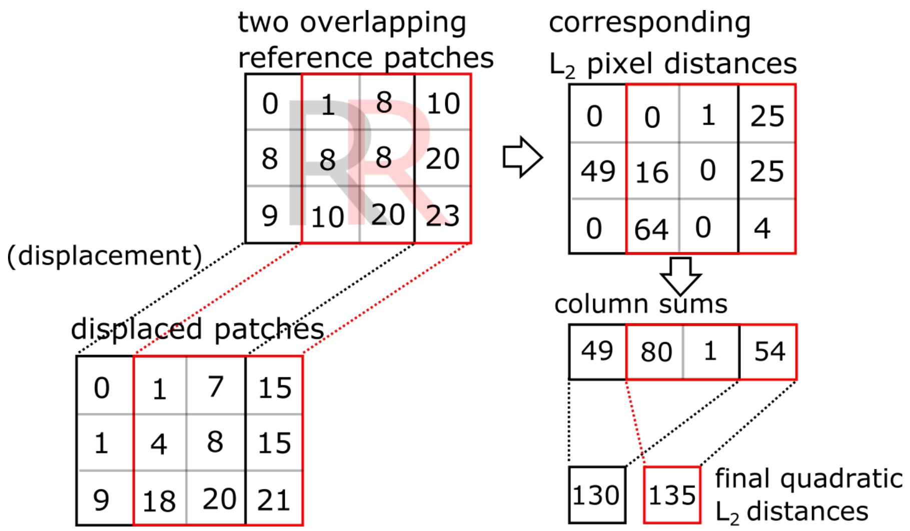

Figure 4.3: Cooperative calculation of distances between two overlapping reference patches and patches displaced by the same value

The threads of one warp always compute the distance to the patches that are displaced by the same value from their reference patches. Because the threads of a warp process reference patches successive in horizontal direction, these patches share many columns. Therefore, to compute the distances, we use the above mentioned approach.

The reference patches that are processed by a single warp covers an area in the image which we denote `P_area`. The patches with displacement that is currently computed by this warp defines `Q_area`. Both these areas have the same dimensions. Width is equal to $(\texttt{warpSize} - 1) \cdot p + k$, where k denotes the width of a patch and p the step between two reference patches. Height of these areas is k.

Each thread of a warp computes the quadratic L_2 distance between a column of pixels of the `P_area` and the corresponding column of the `Q_area`. Because the number of columns in these areas is greater than the size of a warp, each thread processes multiple columns. To avoid bank conflicts, a warp always computes a compact block of `warpSize` columns. The column distances are saved into the `diff` array, which resides in shared memory. In order to obtain the final distances from the `diff` array, each thread sums the values corresponding to the columns of its reference patch. This sum could be handled cooperatively in a warp (e.g., by reduction); nevertheless, because the k is relatively small, the sequential sum of these values is sufficient. Figure 4.4 illustrates the computation for multiple warps when the parameter p is set to 1.

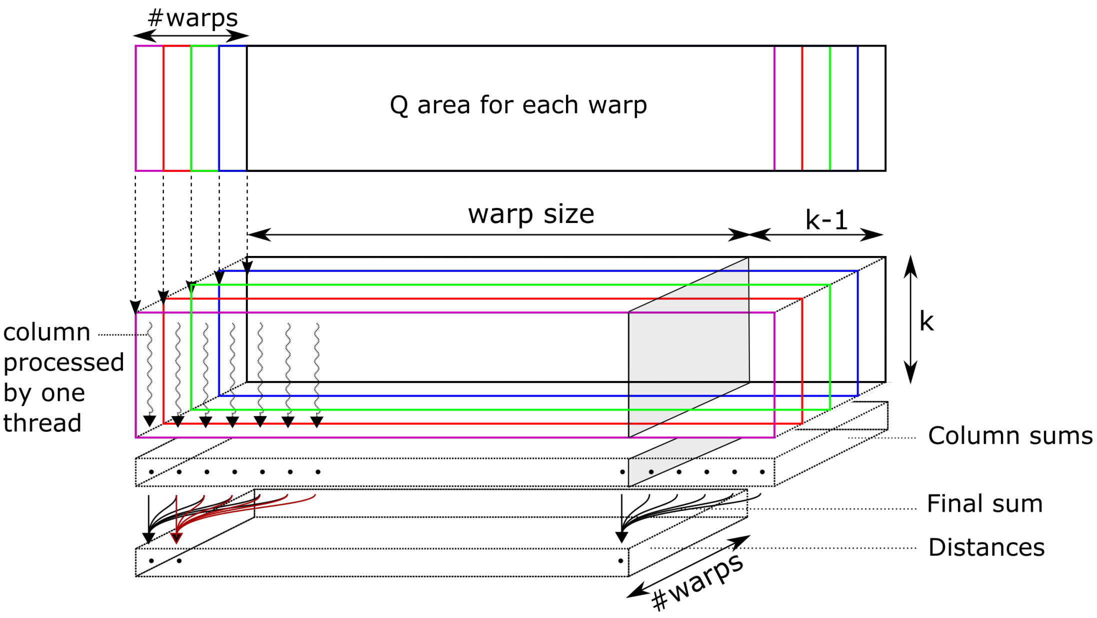

Figure 4.4: 3D visualisation of work in a block when $p = 1$. Different colors show different displacements processed by different warps

After each warp processes its actual `Q_area` and saves the results to the `stack` arrays, all the warps slides by `num_warps` to the right to their next `Q_area` and repeat the same process again. The initial displacement between `P_area` and `Q_area` is $(-\frac{n-1}{2} + i, -\frac{n-1}{2})$, where i denotes the `warpId`. Each warp is sliding horizontally until its horizontal displacement becomes grater than $\frac{n-1}{2}$. Then it continues with the initial horizontal displacement and increased vertical displacement by one. There is no need for any synchronization in this process, therefore, it may happen that two warps are processing completely different displacements.

Because the `P_area` does not change throughout the computation of a block and because it is shared among all warps of a block, we first pre-load it to the shared memory. We can notice, that the `Q_area` overlaps; therefore, we could also pre-load the union of these areas to shared memory. However, the shared memory is very exposed by the rest of the algorithm and any additional array would limit the number of warps in a block; consequently, we rely on L1 and L2 caches to hold `Q_area` values.

4.3.2 Add Patch Coordinates to Thread-Local Array

When a thread obtains the quadratic L_2 distance between its reference patch and some displaced patch, it checks if this distance fits the threshold τ.[1] If the distance is lower than the threshold the thread adds the coordinates of the displaced patch to its local copy of the `stack` array.

We could also have one `stack` array shared by all threads in a block with the same `laneId` and synchronize the insertion of new values. Unfortunately, the adding is an non-trivial operation; therefore, any synchronization would be expensive.

[1] In order to speed up the algorithm, we do not normalize the distance by k^2, but we multiply the threshold by this normalization factor.

The stack array has space for up to N coordinates and distances. It is sorted so that the coordinates of the most similar patch are stored at the end. Sorting the values in this manner allows us to efficiently insert a new value to the stack.

We classify two ways of adding.

- num_patches_in_stack $< N$: The new value is added to the array using the insert-sort algorithm.
- num_patches_in_stack $= N$: If the distance is lower than the largest distance in the array, the new value is inserted using the insert-sort and the value with the largest distance is removed from the array.

Because of the SIMT model of execution, there is a possibility, that a warp would have to pass both the ways. However, because the images we denoise are natural, such situations do not happen very often. Moreover, merging those two branches of code into one would lead to more complex code, which would resulted in a longer execution time.

4.3.3 Merging the Arrays of Coordinates

Finally, when we have explored the entire search window of every reference patch processed by one block, we have total num_warps copies of a stack array for each reference patch. Each of them contains up to N patch coordinates and their distances. For each reference patch we have to choose only N values with the lowest distances from all the copies of stack array and save them to the global memory, where they will be available for collaborative filtering and aggregation.

We use only the first warp for this job and we employ the fact, that every stack array is sorted. Each thread handles only its reference patch and it iterates over the minimal values of the corresponding copies of the stack array to find the patch coordinates with the minimal distance. Once the thread finds such value, it removes it from its local array and insert it to the global stack array of their respective reference patch. We can notice, that the distance itself is not necessary for any future part of the method, therefore only the patch coordinates are written to the global memory. This process is repeated N times or until all the stack copies are empty. For the future processing, it is also necessary to save the number of patch coordinates in each global memory stack array.

4.4 Collaborative Filtering

In the collaborative filtering part the 3D groups are assembled according to the arrays of coordinates produced by block-matching. These groups are then transformed using 3D DFT and filtered by hard-thresholding. During the filtering, the weight of each 3D group is computed. Finally, the 3D IDFT is applied producing filtered 3D groups that consist of patch estimates.

We want to compute the DFT and IDFT transforms using the existing and efficient algorithms of CUFFT library. The CUFFT transforms are done in two steps: creating the transform plan and executing the transform plan. The creation of the plan is an expensive operation; however, one plan can be executed many times. Unfortunately, the transforms in one plan has to be of the same size. Because the 3D groups can contain different number of patches, we have to separate the 3D DFT into 2D and 1D DFT and use the CUFFT only for the 2D DFT. The remaining 1D transform is then handled in a separate kernel.

We create the plan for 2D transforms so that it always transforms N patches in each 3D group regardless the size of the group. Because the number of patches in 3D group can be lower than N, some transforms are needless. Moreover, some of patches could appear in more than one 3D group and they are transformed multiple times with the same result. Although some optimizations may be possible, the FFT is very fast and therefore they would have negligible influence on the total processing time of the method. Moreover, this approach allows us to use simple data layout of the assembled 3D groups array.

CUFFT employs the fact, that DFT of real-valued data input satisfies Hermitian symmetry which is described at the end of the section 2.3.1. In case of real to complex transform of an float array of size $M \cdot N$, CUFFT produces only $M \cdot (\lfloor \frac{N}{2} \rfloor + 1)$ complex coefficients. Some coefficients are still redundant, but now we are allowed to do in-place transforms with minimal additional space requirements. More precisely, if we store complex value as two floats, we need the input array to be padded to $M \cdot (N + 2)$ floats in order to be able to store the transformed data.

Due to the above mentioned problems, each 3D group is filtered using the following scheme. Each of the following items represents a separate kernel or a CUFFT call.

1. Assemble the 3D group according to the coordinates that were obtained during block-matching

2. Apply the 2D transform to every patch of the group. (using the CUFFT)

3. Apply the 1D transform to the remaining dimension of the group, filter the coefficients, count the retained ones, and apply the inverse 1D transform.

4. Apply the inverse 2D transform to every patch of the group (using the CUFFT)

Buffers that are needed for each of these kernels and transforms is shown in Figure 4.2.

4.4.1 Assembling the 3D Groups

Block-matching produces global `stack` buffer that contains patch coordinates for each reference patch of the current batch. According to these coordinates, this kernel assembles the 3D groups and saves them to the `assembled_groups` array in global memory.

Because the CUFFT expects the real-valued padded input and because we expect the number of patches in a group to be N in most cases, for each group there is a space for $k \cdot (k+2) \cdot N$ floats in the `assembled_groups` array regardless the size of a group.

This kernel is launched in the blocks of k^2 threads, where each block is responsible for assembling one 3D group. Each thread sequentially loads one pixel of every patch of the 3D group and stores it in the `assembled_groups` array as a float value.

This kernel does not need any shared memory; therefore, multiple blocks can be processed on each SM of a GPU. Moreover, because the optimal value of k is 8, the blocks of size k^2 (64) are processed by exactly two warps, without any idle threads. Consequently, blocks of this size can efficiently utilize SMs of contemporary GPUs.

4.4.2 2D Transform

The 2D DFT transform is performed using the CUFFT library. This library exposes the same API as the well known FFTW library. Launching the transformation consists of two steps. Creating the transform plan and using that plan to perform the transform itself.

We are creating the plan only once at the beginning and then we use it for all the batches. The plan creation has the following syntax:

```
cufftPlanMany(cufftHandle *plan, int rank, int *n,
  int *inembed, int istride, int idist,
  int *onembed, int ostride, int odist,
  cufftType type, int batch);
```

- **handle** is an output parameter. It is a reference to the plan, and we use it later for executing and destroying the plan.

- **rank** sets the dimensionality of the transform. In our case it is equal to the dimensionality of a patch which is 2.

- ***n** is an array containing the size of the transform in each dimension. We transform patches, therefore the array contains two values: $\{k, k\}$

- **type** defines a type of the transform. We use `CUFFT_R2C`, which denotes real to complex transform. This choice defines that input data will be stored as floats and that output data will consist of only $k(\lfloor \frac{k}{2} \rfloor + 1)$ complex coefficients.

- **batch** is the number of transforms that are done in one call. In our case each transform deals with one patch, each 3D group has maximally N patches and one batch as we have defined it in the section 4.2 contains *batch* 3D groups, where *batch* denotes the size of a batch.

- ***inembed, *onembed** parameters define the layout of the input and output data, respectively. We set both these values to `NULL`, which implies that the basic data layout is used. For in-place transforms that means that pixel at (x, y) of the b-th patch is stored in the input float array at location:

$$b \cdot (k \cdot (k + 2)) + (x \cdot (k + 2) + y)$$

In the output complex array the coefficient at (x, y) of the b-th patch is stored at location:

$$b \cdot (k \cdot (k/2 + 1)) + (x \cdot (k/2 + 1) + y)$$

> We employ the basic data layout because the patches after the in-place transform begins at the same memory address as the original ones.
>
> When the basic data layout is used for both input and output data, the `istride`, `idist`, `ostride` and `odist` are ignored.

The plan allocates $O(batch \cdot N \cdot k^2)$ bytes of space in the global memory, therefore the creation of the plan is an expensive operation and we should minimize the total number of transform plans.

To perform the real to complex single-precision transform according to a plan, we use the following function:

```
cufftExecR2C(cufftHandle plan, cufftReal *idata, cufftComplex *odata);
```

where `plan` is a handle of the plan we have created, `*idata` is an input array of floats, and `*odata` is an output array of complex numbers. Internally these complex numbers are pairs of floats. Because we perform the in-place transforms, `*idata` and `*odata` represents the same array (they point to the same memory address) and we only need to cast these arrays to the proper types.

To compute the 2D IDFT we use almost the same approach as for the 2D DFT. We create new CUFFT plan and execute it on our filtered complex data. For plan the only difference is that `type` is now `CUFFT_C2R`, which denotes complex to real transform. The transform itself is then performed using:

```
cufftExecC2R(cufftHandle plan, cufftComplex *idata, cufftReal *odata);
```

4.4.3 1D Transforms and Filtering

Once the patches are transformed using the CUFFT, we apply the 1D DFT transform in the remaining dimension of every 3D group of a batch. Each coefficient is then checked if its absolute value is higher than the threshold $\lambda_{3D}\sigma$. If it is not, this coefficient is set to zero. During the thresholding the number of retained (non-zero) coefficients for each group is counted. Finally, we apply the 1D IDFT and save the weight of every 3D group.

We use very similar division into the blocks and threads as with the kernel, which is responsible for assembling the 3D groups. Each block handles one 3D group and it has as many threads as the size of a transformed patch which is: $(\lfloor \frac{k}{2} \rfloor + 1)k$. Each thread process the same coefficient in every patch of the group and it serially computes the 1D DFT on these coefficients.

Because the number of patches in each group may vary, we are not able to apply the simple FFT algorithm for 1D DFTs. Fortunately the number of patches is not very high; therefore, we are allowed to compute the DFT exactly according to the DFT definition it in the section 2.3.1.

The DFT intensively access the memory, therefore we store the transformed coefficients in the shared memory. This data then remains in the shared memory until they are transformed by the inverse transformation back to the global memory.

The coefficients of the transformed patches are filtered using the hard-thresholding with the $\lambda_{3D}\sigma$ threshold. Unfortunately the data produced by CUFFT or our DFT are not normalized. To normalize the data we need to divide them by the square root of the transform size, but instead we multiply the threshold. Moreover, due to the performance issues, we do not compute the absolute value of each complex number, but only its quadratic sum. Therefore, we use the following threshold:

$$\lambda'_{3D} = (\lambda_{3D}\sigma)^2 \cdot k^2 \cdot g$$

where g denotes the number of patches in the 3D group.

Finally, the filtered coefficients are transformed using the 1D IDFT which is done in the same way as the 1D DFT.

Computing the Weight of a 3D Group

The weight of each 3D group is defined as an inverse number of retained (nonzero) coefficients. If there is no retained coefficient, the weight is 1.

The number of retained coefficients for a 3D group is computed during the filtering. Each thread counts how many of the processed values has passed the threshold. Because the CUFFT produces only $k \cdot (\lfloor \frac{k}{2} \rfloor + 1)$ complex coefficients, we have to count some coefficients twice in order to get correct result.

To sum the numbers counted by threads within a warp, we use the warp shuffle functions. The sum is performed using traditional parallel reduction sum algorithm. The result for each warp is then stored in the shared memory.

Because the maximal size of a block on contemporary GPUs is 1024, the number of warps in a block is not higher than the `warpSize` (32). Therefore, to obtain the total number of retained coefficients we employ only one warp in the same manner as before to sum the counts that the warps have stored in the shared memory. The final weight is then placed into the global memory, where it is available for the aggregation part.

4.5 Aggregation

Collaborative filtering produces filtered 3D groups as well as weights of these groups. Filtered groups contain patch estimates which has to be aggregated into numerator and denominator buffer at their original locations with the weight of their group. The values of both these buffers corresponds to the pixels of the image. The aggregation is basically an inverse operation to the assembling; therefore, we use the same division into blocks and threads as with the kernel which was responsible for the assembling of the 3D groups.

The patches could overlapped before they were gathered to the 3D groups. While in assembling kernel it led to optimal utilization of caches, in aggregation it complicates the writing to their original locations in numerator and denominator buffers. A write conflict happens every time different blocks want to write overlapping patches at the same time. Thus, to avoid conflicts, we use the atomic adding to write the patch values to both these buffers. Because the buffers are likely to be cached and the collisions are relatively rare, the employment of atomic operations does not significantly slow down the algorithm.

After all batches have been aggregated into the numerator and denominator buffers, we shall proceed to the final division. Final division is computed in a simple kernel. It divides the numerator buffer by the denominator buffer element by element and then it rounds the float result to fit the byte representation of an image. Simple rounding is not sufficient, because even the small inaccuracies could lead to overflow. For example value -0.0001 should be rounded to 0 and not 255.

The final division kernel is not computationally intensive and therefore the division to blocks and threads has negligible influence on the runtime of the algorithm. We define the kernel so that each thread process one pixel and the size of a block is set to 256.

4.6 Data Representation

4.6.1 Image Representation

The gray-scale image is represented as an array of pixels where each pixel defines the brightness at its location. It is often stored as a *byte* with the value from 0 to 255 or as a *float* with the value from 0.0 to 1.0. Due to the smoother quantization, float representation is more accurate. Nevertheless, because of the properties of displays and cameras, the byte representation is the most common and we use it in this thesis. In case of need, the change to float representation would be simple.

Because the block-matching, which is the most computationally and memory intensive part of the method, process the image primarily in horizontal direction, we store the image in a single array so that pixel $u_{x,y}$ is stored on the $y \cdot \texttt{width} + x$. Although the GPU natively works with the 32-bit values, storing the image as a compact array of byte values ensures that 4 times larger area in the image can be cached.

4.6.2 Global Auxiliary Buffers

The proposed algorithm uses several global memory buffers for storing auxiliary data:

- **Coordinates of patches in 3D groups** are stored in the array that we have earlier denoted as `stacks`. For each reference patch of a batch it contains up to N coordinates of similar patches.

  There is a space for N coordinates for each reference patch of a batch no matter of the real number of found similar patches. We store the coordinates as two unsigned integers; therefore, we need 8 bytes for each value. The total size of this array is then $b \cdot N \cdot 8$ bytes, where b denotes the total size of the batch.

  Beside that, for each reference patch we have to know how many patches it contains. If we store each of these counts as an unsigned integer, we need additional $b \cdot 4$ bytes large array to store them.

  When we use the optimal values of the parameters form the section 2.5 and default batch size $(256 \cdot 128)$, these buffers occupies 2MB and 128KB of global memory.

- **3D groups** are stored in `aseembled_groups` buffer. It contains b assembled 3D groups, where b is the total size of a batch. Each group consist of up to N

patches. Because we use CUFFT in-place transforms, there is a $k \cdot (k+2) \cdot 4$ bytes of space for each patch. The total size of this array is therefore:

$$k \cdot (k+2) \cdot N \cdot b \cdot 4$$

This array is the largest array of all auxiliary arrays. When we use the default values, this array occupies 80MB of global memory.

3D groups are stored in this array one after another. The 3D group built according to the i-th reference patch of the virtual rectangle (see the definition of a batch in the section 4.2) starts in this array on the $i \cdot (k \cdot (k+2) \cdot N \cdot 4)$ byte.

- **Weights** for each 3D group of a batch are stored as floats and therefore the size of the weights buffer is $b \cdot 4$ bytes. When we employ the default values, the size is 128KB. The weight of i-th 3D group of a batch is then stored as expected on the i-th position of the array.

- **Numerator and denominator:** These two buffers are of the same dimensions as the image, but each value is stored as a float. Therefore, each of these buffers consumes 4 times more space than the image in byte representation. The size of each of these buffers for a 16MPx image is 61MB.

The total memory consumption is defined mostly by the size of a batch and size of an image. The optimal batch size may vary. For small images it is good to have the smallest batch that covers all reference patches; however, for larger images we found optimal batch to be (256 x 128) patches large.

To copy image between device and host memory we employ `cudaMemcpy` function. We could partition the image and overlap the execution with copying, but due to the computation complexity of BM3D the saved time would be negligible. Consequently, we copy the entire image at once.

The proposed BM3D algorithm is optimized for natural images of the sizes that are common for contemporary cameras. For extremely large images, the allocated space could reach the size of the global memory it self, regardless the batch size. Nevertheless, this problem could be solved by the partitioning as well.

5. Experiments

Based on the information we have provided in the chapter 3 we have created a CUDA accelerated BM3D prototype implementation. In this section, we provide empirical results, that evaluates our implementation in terms of processing time and quality of the denoised pictures. The results are compared with the existing implementations. compared with the existing implementations.

5.1 Existing Implementations

There are two existing implementations that we employ in testing. First of them was created in MATLAB along with the BM3D method itself [4]. The second one was created together with the analysis of this method by Marc Lebrun [14]. It is written in C++, it uses OpenMP for parallelization and it is open-source.

5.1.1 Original MATLAB Implementations

This implementation is still maintained and updated. We use the latest version (2.0) from 2014. It is written in MATLAB; therefore, in order to run it, it is necessary to have MATLAB software installed. Also the processing time of this implementation is dependent on the MATLAB version installed. We use the MATLAB 2014a x64 for UNIX.

This implementation allows us to use various transforms; unfortunately, the DFT is not one of them. Consequently we employ Discrete Cosine Transform (DCT) which is similar to DFT in the terms of both memory and time complexity. DCT can be computed using Fast Cosine Transform (FCT) algorithms, which are based on the FFT and which have the same computation complexity as FFT. Unfortunately, because the authors do not provide source codes of this implementation, it would be difficult measure the time spent solely on transforms and we rely on the above mentioned facts.

The MATLAB implementation computes both steps of the BM3D method. Because we do provide only the first step, for testing purposes, we have slightly modified the MATLAB scripts to measure the first step only.

5.1.2 Open Source OpenMP Implementation

With the analysis of the BM3D method [14] Marc Lebrun has developed the C++ open-source parallel implementation based on OpenMP. Unfortunately, DFT is not listed among the available transforms as well. Therefore, as with the MATLAB version, we use the DCT. We have also slightly changed the code to measure the performance of the first step of BM3D only.

This implementation requires large amount of memory space. The larger the image is, the more memory it require. Unfortunately, on common desktop computers, the program fails to denoise any larger images. For example when denoising 9Mpx image on computer with 4GB of RAM, the process is killed before the denoising is finished.

5.2 Testing Hardware & Metholodogy

The performance test were conducted on the following computers:

- **Desktop 1:**
  GPU: NVIDIA GeForce GTX 660 (Kepler architecture)
  CPU: AMD Phenom(TM) II X2 550 @ 3.1GHz (2 cores)
  RAM: 4GB
  OS: Linux x86_64

- **Desktop 2:**
  GPU: 2 x NVIDIA GeForce GTX 980 (Maxwell architecture)
  CPU: Intel(R) Core(TM) i7 870 @ 2.93 (4 cores)
  RAM: 16GB
  OS: Linux x86_64

- **Server 1:**
  GPU: 2 x NVIDIA Tesla K20m (Kepler architecture)
  CPU: Intel(R) Xeon(R) E5-2630 @ 2.30GHz (2 x 6 cores)
  RAM: 128GB
  OS: Linux x86_64

The graphics cards of the server computer do not have any display output. Therefore, the long-running kernels are not terminated and all the computational power of a GPU can be used by the tested applications.

We measure the execution-time of the functions we are interested in (first step of BM3D and individual kernel calls). Because the execution-time depends on many external factors such as hardware, OS, etc., it is not an exact measurement unit. However, all the experiments were conducted in the environment where most of the computational power and RAM were available to the tested application and they were repeated at least 5 times. Therefore, we use this unit for basic overview of the complexity of the tested algorithms.

In the following sections, we always show only the arithmetic average and the maximal deviation of the measured values for each test.

5.3 Performance Tests

5.3.1 Comparison of Implementations

In order to compare the implementations we have measured the execution-time of the first step of BM3D on images of different sizes. Tables 5.1, 5.2 and 5.3 shows the results for Desktop 1, Desktop 2 and Server 1. On desktop computers, the denoising of images above 9MPx failed because of the lack of memory. Every performance test was conducted at least 5 times and we show the average of times and the maximal deviation we measured.

Unfortunately, we do not have MATLAB available on the Desktop 2 and Server 1 computers; therefore, on these computers we measured only the CUDA and OpenMP implementation.

	CUDA	OpenMP	Matlab 2014a x64
lena 512x512	0.74s 46%	3.95s 1.55%	7.72s 1.55%
roads 1615x1026	3.49s 0.34%	279.18s 4.09%	57.54s 1.15%
bodies 9MPx 3714x2460	18.16s 1.93%		263.20s 1.94%
bodies 4714x3122	29.53s 0.44%		423.44s 0.89%

Table 5.1: BM3D processing time and maximal deviation on Desktop1

	CUDA	OpenMP
lena 512x512	0.21s 5.18%	1.57s 1.40%
roads 1615x1026	0.69s 1.91%	9.34s 0.95%
bodies 9MPx 3714x2460	3.13s 0.33%	
bodies 4714x3122	5.10s 2.23%	

Table 5.2: BM3D processing time and maximal deviation on Desktop2

	CUDA	OpenMP
lena 512x512	0.50s 12.40%	1.21s 3.22%
roads 1615x1026	2.42s 0.09%	5.19s 10.96%
bodies 9MPx 3714x2460	12.30s 0.01%	22.47s 5.59%
bodies 4714x3122	20.10s 0.01%	29.96s 6.24%

Table 5.3: BM3D processing time and maximal deviation on Server1

These experiments proves, that processing time of our CUDA-accelerated implementation is on server computers slightly faster than the OpenMP implementation. Nevertheless, on desktop computers our implementation can be more than ten times faster than the existing ones. Moreover, our prototype implementation is rapidly faster on the Maxwell GPU architecture than on the older Kepler architecture. We assume that it is mainly a consequence of the larger SM memories on the Maxwell architecture.

5.3.2 Kernel Execution Times

For future development of our implementation we have tested which parts of BM3D are the most computationally intensive. We have measured how much time the algorithm spent on each defining kernel of BM3D. The results are visualized in figure 5.1

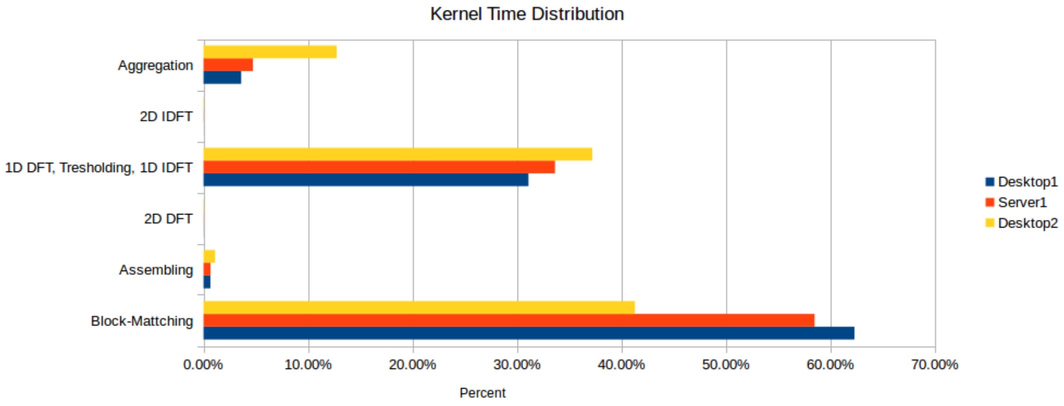

Figure 5.1: BM3D method

The graph demonstrates, that the processing time of 2D transforms is negligible in comparison to the rest of the algorithm. Also it shows, that the distribution of processing time is dependent on the graphics card architecture. On the Desktop2, which has GPUs with the Maxwell architecture, the processing time of the three most complex kernels is much more balanced than on the older Kepler architecture. This is probably also the result of the larger shared memory.

Although the most computationally intensive kernel is the block-matching, the processing time of the thresholding kernel is also very important. Nevertheless, this time can be significantly reduced by employing more sophisticated 1D DFT and IDFT algorithm, than we have used.

5.3.3 Denoising Performance

In order to evaluate our implementation in terms of denoising performance, we measure the PSNR (peak signal to noise ratio). To compute the PSNR, first we define the RMSE (root mean square error). RMSE between the original

	CUDA	OpenMP	Matlab 2014a x64
lena $\sigma = 20$	32.27	31.96	32.49
roads $\sigma = 25$	24.81	25.54	25.53
bodies 9MPx $\sigma = 25$	33.47	34.73	34.62
bodies $\sigma = 25$	33.25	35.17	35.01

Table 5.4: Denoising performance of different BM3D implementations measured on different images in PSNR

(reference) image u_R and denoised image u_D is computed as:

$$RMSE = \sqrt{\frac{\sum\limits_{x \in X} (u_R(x) - u_D(x))^2}{|X|}}$$

where X denotes set of all pixel coordinates in an image. PSNR is then defined as:

$$PSNR = 20 \cdot \log_{10} \left(\frac{MAX}{RMSE} \right)$$

where MAX denotes maximal pixel value. In the case of images in byte representation it is 255. PSNR is measured in decibels (db). The larger the PSNR, the better the denoising.

Because we have implemented only the simplified version of the BM3D method, there are some differences between our implementation and the existing implementations. We can only denoise images with noise variance below 40. Also we use DFT transforms instead of the DCT transforms. Therefore, the denoising performance is not the same. For basic overview, we conducted tests on several images affected by the noise with variance σ around 25.

The results in table 5.4 shows the PSNR values for various images. As expected, our implementation has slightly lower denoising performance than the existing implementations. We assume that it is mainly because of the used transforms. Replacement of the DFT by DCT, should improve the PSNR values, while the processing time should (due to the similar time complexity) remain the same.

This implementation is mainly focused on the total processing time of the method, which is significantly lower than with the existing implementations. Nevertheless, the measured values prove that even with the DFT, the denoising performance is still very good.

Conclusion

Analysis of the BM3D image denoising method and the basic information about the GPU architecture and GPU programming were used as a basis for implementation of simplified CUDA-accelerated BM3D method.

Compared to the full BM3D algorithm, the simplified version can denoise only images corrupted by noise with variance $\sigma < 40$. It employs only the 1st step of the method and it uses only the DFT for all the transforms. However, common photos are rarely corrupted by the noise with higher variance and the extension to the both steps of the BM3D as well as the replacement of DFT by DCT should be simple.

The merit of this thesis, has been proven empirically. On traditional desktop computers, the CUDA-accelerated implementation was more than 10 times faster than the existing CPU implementations. Moreover, even on server computers, that contains up tens of CPU cores, the achieved speed-up was still significant. The empirical results have also showed, that the processing-time of the algorithm on modern desktop GPUs, that are primarily designed for computer gaming, can be significantly lower than on one generation older professional server-oriented GPU.

Due to the simplifications, the proposed implementation has slightly lower denoising performance than the original method. Although, this implementation is focused on short processing time rather than on the maximal denoising performance, the empirical results shows, that the denoising quality is still very good.

Because the presented implementation has been created mainly for denoising of high-resolution photos on common desktop computers, where the speed-up was largest, we may proudly mark it as successful.

Future work

The prototype implementation created as a part of this thesis offers lot of space for improvement. These improvements can be divided into three groups:

- **Improvements of algorithms:** The processing times of the kernels shows, that the most compute-intensive parts are block-matching, 1D DFT transforms and filtering, and aggregation. On the latest GPUs the processing-times of these algorithms are very similar. However, for the 1D DFT and 1D IDFT, the current implementation uses rather ineffective algorithms. Consequently, by employing sophisticated FFT algorithm or by precomputing some values of the current DFT algorithm, we can lower the processing-time of this kernel.

  The block-matching also provides some space for improvement. For example, when processing images in byte representation, we can employ SIMD intrinsics, that can compute the distance between four pairs of pixels by one thread simultaneously.

- **Utilize multiple GPUs:** The proposed implementation can utilize only one device; however, a computer may have multiple devices. Because batches are generally smaller than images and because the search area around each reference patch is strongly limited, the overall architecture can be modified to process each batch on different GPU. Because the batches overlap only around their edges, there wont be many conflicts while merging the aggregation buffers. Therefore, the achieved speed-up should be almost equal to the number of available GPUs.

- **Extensions:** In order to improve the denoising performance of the implementation, we can extend it to process the second step of BM3D (Wiener filtering). Fortunately, the steps are very alike; therefore, we can use most of the developed algorithms; namely, block-matching, assembling of 3D groups, aggregation, and final division.

  The implementation could be also extended to use other transformation as DCT, Hadamard transform, or Bior1.5 transform. However, in order to extend the algorithm to denoise images corrupted by the noise with $sigma > 40$, more complex changes to the overall architecture would be necessary.

  The method itself can be also extended to process colour images. For that purpose only the minor changes to the overall architecture has to be made. The block-matching would be computed on the luminance channel only and the filtering and aggregation would be processed on each channel separately.

Bibliography

[1] GOW, Ryan D., et al. A comprehensive tool for modeling CMOS image-sensor-noise performance. *Electron Devices, IEEE Transactions on*, 2007, 54.6: 1321-1329.

[2] CHEN, Ting, et al. How small should pixel size be?. In: *Electronic Imaging.* International Society for Optics and Photonics, 2000. p. 451-459.

[3] DABOV, Kostadin, et al. Image denoising with block-matching and 3D filtering. In: *Electronic Imaging 2006.* International Society for Optics and Photonics, 2006. p. 606414-606414-12.

[4] DABOV, Kostadin, et al. Image denoising by sparse 3-D transform-domain collaborative filtering. *Image Processing, IEEE Transactions on*, 2007, 16.8: 2080-2095.

[5] BUADES, Antoni; COLL, Bartomeu; MOREL, Jean-Michel. A non-local algorithm for image denoising. In: *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on.* IEEE, 2005. p. 60-65.

[6] ZHENG, Ziyi; XU, Wei; MUELLER, Klaus. Performance tuning for CUDA-accelerated neighborhood denoising filters. In: *Workshop on high performance image reconstruction (HPIR).* 2011.

[7] HUANG, Kuidong; ZHANG, Dinghua; WANG, Kai. Non-local means denoising algorithm accelerated by GPU. In: *Sixth International Symposium on Multispectral Image Processing and Pattern Recognition.* International Society for Optics and Photonics, 2009. p. 749711-749711-8.

[8] MÁRQUES, Adrián; PARDO, Alvaro. Implementation of Non Local Means Filter in GPUs. In: *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications.* Springer Berlin Heidelberg, 2013. p. 407-414.

[9] DOLWITHAYAKUL, B.; CHANTRAPORNCHAI, Chantana; CHUMCHOB, N. GPU-based total variation image restoration using Sliding Window Gauss-Seidel algorithm. In: *Intelligent Signal Processing and Communications Systems (ISPACS), 2011 International Symposium on.* IEEE, 2011. p. 1-6.

[10] SU, Yang; XU, Zhijie. Parallel implementation of wavelet-based image denoising on programmable PC-grade graphics hardware. *Signal Processing*, 2010, 90.8: 2396-2411.

[11] YUANFENG, Lian; YAN, Zhao. Accelerating fuzzy adaptive anisotropic diffusion on GPU. In: *Electronic Measurement & Instruments (ICEMI), 2011 10th International Conference on.* IEEE, 2011. p. 175-180.

[12] DE FONTES, Fernanda Palhano Xavier, et al. Real time ultrasound image denoising. *Journal of real-time image processing*, 2011, 6.1: 15-22.

[13] PERROT, Gilles, et al. Fast GPU-based denoising filter using isoline levels. *Journal of Real-Time Image Processing*, 2013, 1-12.

[14] LEBRUN, Marc. An analysis and implementation of the BM3D image denoising method. *Image Processing On Line, 2012*, 2012.

[15] LEBRUN, Marc; BUADES, Antoni; MOREL, Jean-Michel. Implementation of the "Non-Local Bayes" (NL-Bayes) Image Denoising Algorithm. *Image Processing On Line, 2013*, 2013.

[16] BONCELET Charles. Image Noise Models. In: Alan C. Bovik. *Handbook of Image and Video Processing* Academic press, 2010.

[17] PRATT, William K. *Digital Image Processing*, Third Edition, John Wiley & Sons, Inc., 2001.

[18] NVIDIA. *Maxwell Tuning Guide.*
`http://docs.nvidia.com/cuda/maxwell-tuning-guide/`

[19] KIRK, David B.; WEN-MEI, W. Hwu. *Programming massively parallel processors: a hands-on approach.* Newnes, 2012.

[20] SCARPINO, Matthew. *Opencl in Action: How to Accelerate Graphics and Computation.* NY. 2012.

[21] NVIDIA. *CUDA C Programming Guide.*
`http://docs.nvidia.com/cuda/cuda-c-programming-guide/`

[22] NVIDIA. *CUFFT library documentation.*
`http://docs.nvidia.com/cuda/cufft/index.html`

List of Abbreviations

AWGN Additive White Gaussian Noise
BM3D Block-Matching and 3D Filtering
Bior Biorthogonal wavelet
CPU Central Processing Unit
CUDA Compute Unified Device Architecture
DCT Discrete Cosine Transform
DFT Discrete Fourier Transform
FT Fourier Transform
FFT Fast Fourier Transform
GPGPU General-Purpose Computing on GPU
GPU Graphic Processing Unit
IDFT Inverse Discrete Fourier Transform
NL-Means Non-Local Means
SFU Special Function Unit
SIMD Single Instruction Multiple Data
SIMT Single Instruction Multiple Thread
SM Streaming Multiprocessor