



OULUN YLIOPISTO
UNIVERSITY of OULU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Sampsa Sarjanoja

**BM3D IMAGE DENOISING USING HETEROGENEOUS
COMPUTING PLATFORMS**

Master's Thesis
Degree Programme in Computer Science and Engineering
March 2015

Sarjanoja S. (2015) BM3D image denoising using heterogeneous computing platforms. University of Oulu, Department of Computer Science and Engineering. Master's Thesis, 49 p.

ABSTRACT

Noise reduction is one of the most fundamental digital image processing problems, and is often designed to be solved at an early stage of the image processing path. Noise appears on the images in many different ways, and it is inevitable. In general, various image processing algorithms perform better if their input is as error-free as possible. In order to keep the processing delays small in different computing platforms, it is important that the noise reduction is performed swiftly.

The recent progress in the entertainment industry has led to major improvements in the computing capabilities of graphics cards. Today, graphics circuits consist of several hundreds or even thousands of computing units. Using these computing units for general-purpose computation is possible with OpenCL and CUDA programming interfaces. In applications where the processed data is relatively independent, using parallel computing units may increase the performance significantly. Graphics chips enabled with general-purpose computation capabilities are becoming more common also in mobile devices. In addition, photography has never been as popular as it is nowadays by using mobile devices.

This thesis aims to implement the calculation of the state-of-the-art technology used in noise reduction, block-matching and three-dimensional filtering (BM3D), to be executed in heterogeneous computing environments. This study evaluates the performance of the presented implementations by making comparisons with existing implementations. The presented implementations achieve significant benefits from the use of parallel computing devices. At the same time the comparisons illustrate general problems in the utilization of using massively parallel processing for the calculation of complex imaging algorithms.

Keywords: image restoration, GPU computing, Wiener filtering

Sarjanoja S. (2015) BM3D kohinanpoistoalgoritmi heterogeenisiä laskentalustoja käyttäen. Oulun yliopisto, tietotekniikan osasto. Diplomityö, 49 s.

TIIVISTELMÄ

Kohinanpoisto on yksi keskeisimmistä digitaaliseen kuvankäsittelyyn liittyvistä ongelmista, joka useimmiten pyritään ratkaisemaan jo signaalinkäsittelyvuon varhaisessa vaiheessa. Kohinaa ilmestyy kuviin monella eri tavalla ja sen esiintyminen on väistämätöntä. Useat kuvankäsittelyalgoritmit toimivat paremmin, jos niiden syöte on valmiiksi mahdollisimman virheetöntä käsiteltäväksi. Jotta kuvankäsittelyviiveet pysyisivät pieninä eri laskentalustoilla, on tärkeää että myös kohinanpoisto suoritetaan nopeasti.

Viihdeteollisuuden kehityksen myötä näytönohjaimien laskentateho on moninkertaistunut. Nykyisin näytönohjainpiirit koostuvat useista sadoista tai jopa tuhansista laskentayksiköistä. Näiden laskentayksiköiden käyttäminen yleiskäyttöiseen laskentaan on mahdollista OpenCL- ja CUDA-ohjelmointirajapinnoilla. Rinnakkaislaskenta usealla laskentayksiköllä mahdollistaa suuria suorituskyvyn parannuksia käyttökohteissa, joissa käsiteltävä tieto on toisistaan riippumatonta tai löyhästi riippuvaista. Näytönohjainpiirien käyttö yleisessä laskennassa on yleistymässä myös mobiililaitteissa. Lisäksi valokuvaaminen on nykypäivänä suosituinta juuri mobiililaitteilla.

Tämä diplomityö pyrkii selvittämään viimeisimmän kohinanpoistoon käytettävän tekniikan, lohkonsovitusta ja kolmiulotteinen suodatus (block-matching and three-dimensional filtering, BM3D), laskennan toteuttamista heterogeenisissä laskentaympäristöissä. Työssä arvioidaan esiteltyjen toteutusten suorituskykyä tekemällä vertailuja jo olemassa oleviin toteutuksiin. Esitellyt toteutukset saavuttavat merkittäviä hyötyjä rinnakkaislaskennan käyttämisestä. Samalla vertailuissa havainnollistetaan yleisiä ongelmakohtia näytönohjainlaskennan hyödyntämisessä monimutkaisten kuvankäsittelyalgoritmien laskentaan.

Avainsanat: kuvanparannus, näytönohjainlaskenta, Wiener-suodatus

TABLE OF CONTENTS

ABSTRACT	
TIIVISTELMÄ	
TABLE OF CONTENTS	
FOREWORD	
ABBREVIATIONS	
1. INTRODUCTION.....	8
2. HETEROGENEOUS COMPUTING WITH OPENCL AND CUDA.....	10
2.1. Background and motivation	10
2.2. GPGPU computing.....	11
2.2.1. GPU memory types	11
2.2.2. Accessing GPU memories.....	13
2.2.3. Occupancy	14
2.2.4. Branch divergences	15
2.3. OpenCL standard.....	15
2.4. CUDA.....	16
3. DENOISING ALGORITHMS	18
3.1. Spatial-domain filtering.....	18
3.1.1. Total variation	19
3.1.2. Neighborhood filters	19
3.1.3. Non-local means.....	20
3.2. Transform-domain filtering.....	20
3.2.1. Wiener filter	21
3.2.2. BLS-GSM.....	22
3.3. K-SVD.....	22
3.4. BM3D algorithm	23
3.4.1. Structure	23
3.4.2. Block-matching	24
3.4.3. Collaborative filtering	25
3.4.4. Aggregation.....	27
3.4.5. Wiener filtering	27
3.5. Summary	28
4. BM3D FILTER IMPLEMENTATION.....	30
4.1. Reference implementations	30
4.2. OpenCL implementation	30
4.2.1. Parallelization.....	32
4.2.2. CUDA implementations.....	34
4.2.3. Problems occurred.....	35
5. RESULTS.....	36
5.1. Test environment.....	37
5.2. Test case 1: 256×256 image with simulated noise.....	37
5.3. Test case 2: FHD image with simulated noise	38
5.4. Test case 3: UHD image with simulated noise.....	39
5.5. Test case 4: Real image with natural noise	40
5.6. Overall performance.....	41
5.7. Comparison to reference implementations.....	42
6. DISCUSSION	44

6.1.	Future development.....	45
7.	CONCLUSION	46
8.	REFERENCES.....	47

FOREWORD

This thesis was conducted at the Center for Machine Vision Research in the University of Oulu. The thesis was supervised by the Professors Jari Hannuksela and Jani Boutellier.

First of all I want to thank both my supervisors for the thesis subject and the invaluable opportunity to be a part of the CMV research group. This offered position in the University of Oulu has been a great help to me to finish my studies. Secondly, I would like to sincerely thank the staff of Department of Computer Science and Engineering for their knowledge and all the co-workers that have assisted me with some of the problems occurred in the work. Finally, I would also like to thank my family for all the support they have provided over the years.

Oulu, March 5th, 2015

Sampsa Sarjanoja

ABBREVIATIONS

1D	One-dimensional
2D	Two-dimensional
3D	Three-dimensional
ALU	Arithmetic logic unit
ANR	Application not responding
API	Application programming interface
AWGN	Additive white Gaussian noise
BLS	Bayes least squares
BM	Block-matching
BM3D	Block-matching and three-dimensional filtering
CPU	Central processing unit
CUDA	Compute unified device architecture
DC	Direct current
DWT	Discrete wavelet transform
FFT	Fast Fourier transform
FHD	Full high-definition
GFLOPS	Giga-floating point operations per second
GPGPU	General-purpose graphics processing unit
GPU	Graphics processing unit
GSM	Gaussian scale mixture
HD	High-definition
HSA	Heterogeneous system architecture
ILP	Instruction-level parallelism
LRU	Least recently used
MSE	Mean squared error
ND	N-dimensional
NL	Non-local
NP	Nondeterministic polynomial time
OMP	Orthogonal matching pursuit
OpenCL	Open computing language
OpenGL	Open graphics library
OpenMP	Open multi-processing
OS	Operating system
PC	Personal computer
PCA	Principal component analysis
PCIe	Peripheral component interconnect express
PSF	Point spread function
PSNR	Peak signal-to-noise ratio
RAM	Random-access memory
SA	Shape-adaptive
SD	Standard deviation
SIMD	Single instruction, multiple data
SPMD	Single program, multiple data
SNR	Signal-to-noise ratio
SAD	Sum of absolute differences
SDK	Software development kit
SSD	Sum of squared differences

SSIM	Structural similarity
SVD	Singular value decomposition
TLP	Thread-level parallelism
TV	Total variation
TVD	Total variation denoising
UHD	Ultra high-definition
σ	Standard deviation
σ^2	Variance

1. INTRODUCTION

Noise reduction a.k.a. image denoising is an old problem in the field of image processing. Albeit it has been studied for decades it is still a valid challenge and the theoretical limits of restoration are unknown. The state-of-the-art method BM3D invented by Dabov et al. [1] provides very good results in terms of visual image quality, but the complexity and the requirements of the algorithm add several limitations to its usage. Yet image denoising is a required step in many image processing applications and is often placed near the raw Bayer output in the processing pipeline to provide a cleaner signal for further processing steps to work with. However camera systems in different platforms nowadays are required to be quickly operated and including a modern image denoising implementation in the pipeline might add up too much latency. Therefore computational speed of denoising algorithms needs to be considered.

Digital images that are captured with analog sensors always contain digital noise of some magnitude. Especially pictures taken in low-light conditions have usually much of clearly visual noise that might ruin the image scenery. The noise is primarily caused by the thermal changes in the electronic circuits combined with the amplification of the sensor cells. Additionally, there is photon shot noise where randomly distributed photons are not shared equally to pixels in the receiving sensor. Therefore the exposure time relates inversely to the amount of noise; more light, less noise. [2]

Generally noisy images are modeled as

$$z(x) = y(x) + \eta(x), \quad x \in X \quad (1)$$

where X is the area of the image, $z(x)$ is the denoised signal, $y(x)$ is the original noiseless signal and $\eta(x)$ is the average white Gaussian noise affecting the original signal. Although there exist various types of noise, AWGN is the one that is typically modeled in denoising algorithms. Traditionally the quality of a denoising filter has been measured with the peak signal-to-noise ratio value. The PSNR is defined as the ratio between maximum possible pixel value and the mean squared error in the image presented usually in logarithmic decibel scale. [2]

A naïve solution to reduce AWGN in an image would be to filter it with a Gaussian blur point spread function. This type of filtering can be applied fast and can actually work well with high resolution images where the noise is small in size compared to the preservable details in the image. However in most cases the result is a blurred image with no sharp edges and small details visible, but still with blended noise artifacts.

A better solution is to exploit the characteristics of natural images. Most pictures taken are relatively sparse and mainly consist of smooth surfaces with sharp edges. Thus most of the information is contained in the lower frequencies and the data in higher frequencies denote the edges and small details. Natural images contain also a lot of local self-similarities, where similar patterns in image subsets can be found and matched together. The idea is to compare two or more similar patches of data and collaboratively filter the differing noise out.

The recent achievements in the field of heterogeneous computing opens up many opportunities in image processing. The current GPGPU platforms such as CUDA and OpenCL make it possible to use the massive concurrent processing power of GPUs for general-purpose computing. A top level view illustrating the parallel capacity in GPUs

is shown in Figure 1. This is very appealing for image processing since most algorithms or at least parts of them can be designed and implemented in parallel format.

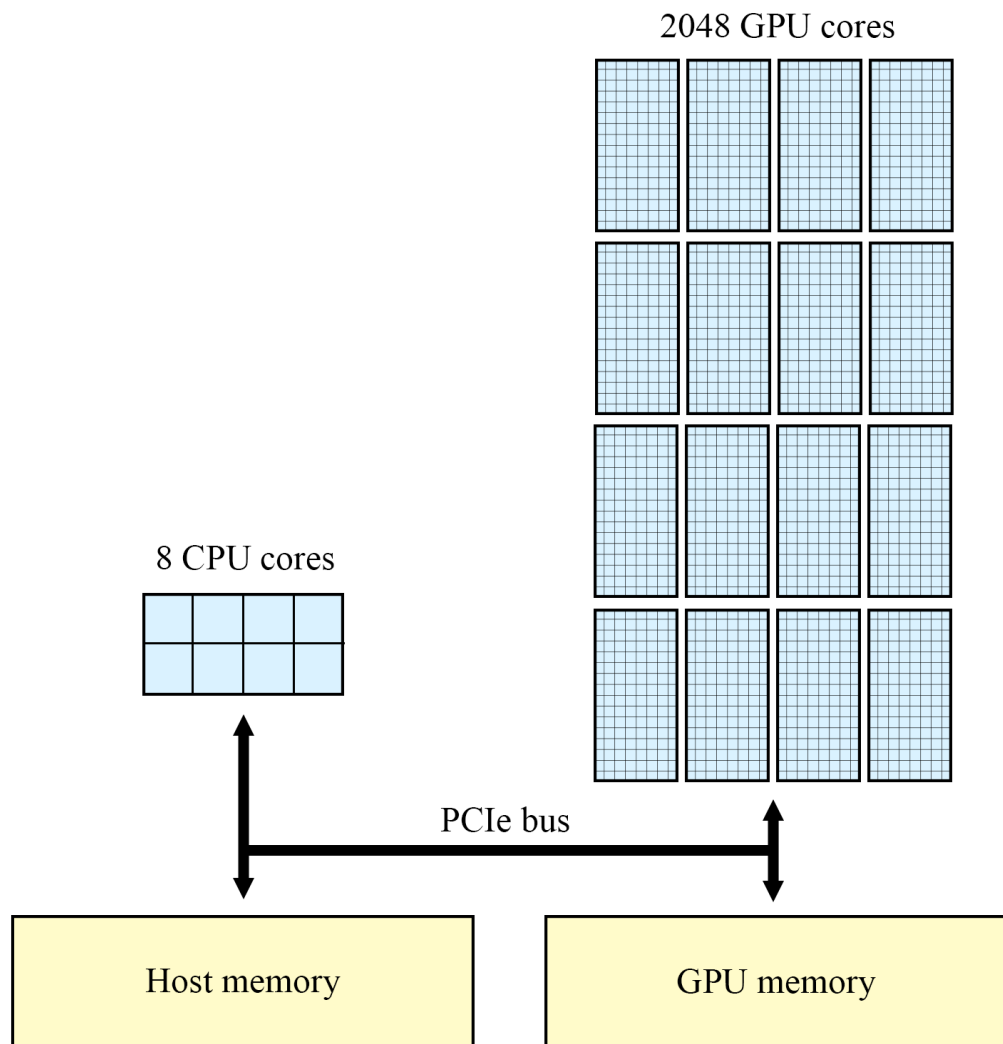


Figure 1. Top level hardware view of a modern PC's computational resources.

This thesis presents an implementation of the BM3D image denoising algorithm by utilizing modern parallel processing technologies provided by recent GPGPU devices and the OpenCL framework. The developed solution outperforms the available implementations in terms of processing speed and is competitive in denoising quality accuracy. The aim is to explore the possibilities of harnessing the parallel computing power of GPU devices to image processing advantages.

2. HETEROGENEOUS COMPUTING WITH OPENCL AND CUDA

In this chapter graphics processing units and their interfaces to general-purpose computing are discussed. In Section 2.1 the evolution and general information of heterogeneous system architectures are presented, and in Section 2.2 GPU computing and the common issues in developing a GPGPU program are discussed. The Sections 2.3 and 2.4 present brief details about commonly used GPGPU frameworks OpenCL and CUDA.

2.1. Background and motivation

Traditionally to improve computational performances in computers the clock rates have been increased. However nowadays this strategy has faced its physical limits and is not feasible anymore. Other solutions to gain more performance are needed to speed up the computers. One of these solutions has been the development of superscalar processors that exploit various techniques of improving instruction-level parallelism within a single computational unit. On the hardware level this is known as dynamic parallelism, where the processing unit can, for example, analyze and execute the instructions out-of-order when applicable. On the software level, ILP is known as static parallelism, i.e. where the compilers pipeline the instructions in more efficient order. Another solution is to utilize more parallelization with data parallelism, where independent data items can be processed separately in different computational units. This can be achieved with code vectorization, namely with SIMD (single instruction, multiple data) instructions, where multiple data items are processed with the same operation simultaneously in parallel. The downside compared to instruction-level parallelism is that this cannot be done transparently; to run code in parallel processing units the code usually needs to be rewritten in parallel format.

In the 90s and early 2000s graphics accelerators were evolving mainly for gaming and multimedia purposes. But after the coming of programmable shaders researchers saw the opportunity of harnessing the computing power of all the GPU cores available for general-purpose computation. However the use of graphics libraries, such as OpenGL and DirectX, for GPGPU computation was tricky and hard to adopt in most of the software. Fortunately several parallel computing solutions emerged later from the need of cheap performance alongside easy access to the compute capabilities on external parallel compute units. NVIDIA's CUDA platform and the Khronos Group's open standard OpenCL are the two most well-known solutions to create parallel computing applications. Other competing frameworks exist also, namely RenderScript developed by Google and Microsoft's DirectCompute.

According to various online sources mobile games market is on the rise internationally and the graphics quality requirements are more demanding along. In addition, customer feedbacks on mobile specifications show that having good camera features on a smartphone is one of the most valued selling characteristics. Therefore performance boost enablers such as GPGPU frameworks are highly welcomed also on mobile platforms. Luckily most of the existing heterogeneous computing platforms used in both desktop and server computing are applicable for mobile usage. Most concerning difference is the power consumption, which is highly limited on mobile platforms.

2.2. GPGPU computing

Graphics processing units are massively parallel devices, which enable many opportunities to gain significant performance improvements in most of computationally expensive programs. Originally GPUs were developed solely to accelerate graphics rendering in games and 3D processing applications but these days they are considered to be more generalized parallel processing units. In high level a GPU nowadays consists of numerous streaming multiprocessors, thread execution controllers and memory partitions [3].

The execution of multiple parallel threads on a GPU are designed to be run in groups. These groups can vary in size, but there is a fixed size limit in the streaming multiprocessors how many threads can be executed simultaneously in a group. Therefore it is beneficial to use thread group sizes that are divisible by this limit. NVIDIA and AMD use the terms *warp* and *wavefront* respectively to describe this limit of parallel thread execution in a processor.

Typically programs are either limited arithmetically or by memory bandwidth. Arithmetically limited programs use more time computing the output of an algorithm compared to moving data between different memories during calculations, and bandwidth-limited is the opposite. This is described with the term arithmetic intensity, i.e. number of operations per memory-word transferred. Nowadays the processing hardware is mostly memory bandwidth-limited, because the development on memory access latencies has been slower than the development on arithmetic processing unit speeds [4]. Therefore optimizing the memory accesses should be the primary concern in algorithm design, especially in GPU computing since there is usually a lot of data exchange between host and GPU device memories.

2.2.1. GPU memory types

To hide high memory access latencies multiple arithmetic operations can be scheduled during the access time, where the data is independent of the queried memory data. However when there are data dependencies the arithmetic units will be stalled. To compensate this and improve arithmetic intensity there are various types of memory included on GPU architectures.

Registers

Registers are banks of memory dynamically partitioned in a register file, which is an on-chip memory that has minimum amount of access latency on a GPU. Each multiprocessor has one register file with a fixed size, e.g. 256 KB per multiprocessor on NVIDIA's Kepler architecture. Registers are allocated and assigned to threads and each register memory slot is accessible by only a single thread at a time. Because amount of registers available is limited they are mainly used for storing local variables in functions. Local variables that cannot fit in the registers are spilled to larger but slower memories. [5]

Shared memory and L1 cache

Another type of fast on-chip memory available on a GPU is the shared memory. This memory has two purposes in the Kepler architecture and is divided into two partitions:

L1 cache and shared memory. These partition sizes are configurable in CUDA. The L1 cache partition might not be available on all modern GPU platforms, but the shared memory is most likely. The main objective of this memory is to reduce the amount of memory access transactions between on- and off-chip memories. It is also shared amongst the thread groups executing in the same streaming multiprocessor, and therefore the memory can be used for inter-thread communication within a thread group. Typically data that is required by multiple threads simultaneously is cached in shared memory, e.g. data with dependency on neighborhood or sequential reads of independent data. The L1 cache behaves as an LRU cache and is designed for spatial reuse of data, not for temporal [6].

L2 cache

As with the L1 cache, L2 works also as an automatic LRU cache. This memory caches accesses to the global memory and is shared amongst all the streaming multiprocessors. The main objective of the cache is to avoid having the bottleneck on global memory bandwidth.

Constant memory

The read-only constant memory is for storing constant variables used by multiple threads. This memory is usually located off-chip but is cached on-chip to reduce memory transactions.

Global memory

The slowest memory type on a GPU is the global memory. The off-chip global memory has high access latency compared to on-chip memories but offers a lot more capacity, which is nowadays in gigabyte magnitude. The access latency is several hundreds of clock cycles and is therefore about ten times slower than on-chip memories.

Sometimes parts of global memory are addressed as local memory. This memory is local in scope of a thread, i.e. only accessible by a single thread, and the purpose is to extend threads' available local memory. It should not to be confused with OpenCL terminology of local memory, which means the same as the shared memory described earlier. More of these differences in terminology can be seen in Table 1 below.

Table 1. Terminology differences between OpenCL and CUDA.

OpenCL	CUDA
Compute unit	Multiprocessor
Processing element	CUDA core
Work-item	Thread
Work-group	Block
NDRange	Grid
Private memory	Local memory
Local memory	Shared memory
Local ID	Thread ID
Global ID	Block ID * Block size + Thread ID

Texture memory

Texture memory is an off-chip memory which has a special purpose of storing texture images. Likewise to constant memory, texture memory is also cached on-chip. Images are saved into this memory by taking advantage of spatial locality. Also several hardware benefits are often designed for this memory such as automatic normalization and image boundary handling.

2.2.2. Accessing GPU memories

Generally GPU architectures are structured to access several memory banks simultaneously. To take advantage of the full potential of a GPU unit's memory bandwidths, all memory accesses should be done in a coalesced and aligned manner since memory locations are divided into fixed-size segments. This allows multiple memory addresses in a segment to be bundled and available via a single load or store transaction. Examples of aligned sequential and non-sequential accesses are shown in Figure 2.

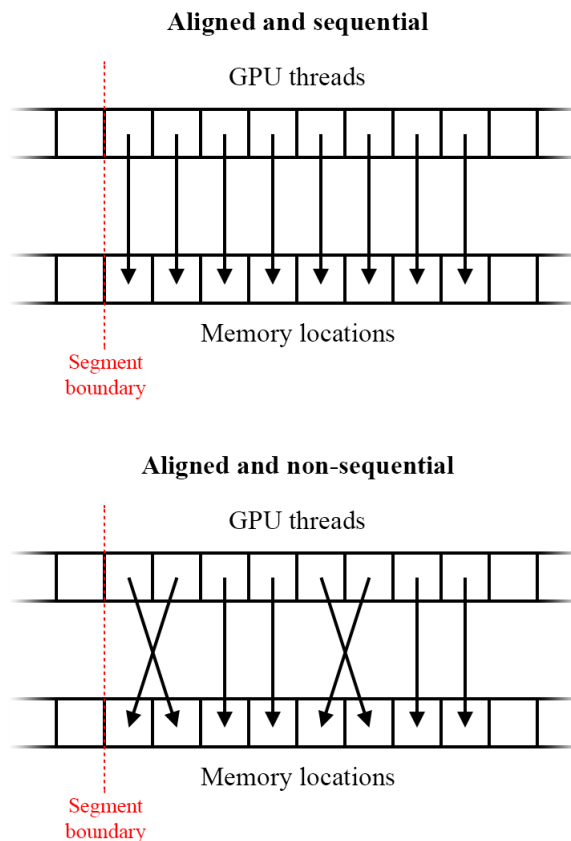


Figure 2. Efficient memory access patterns.

Non-sequential access patterns inside a segment have no degrading impact on performance, since the banks accessed reside in the same memory transaction as in sequential. However access patterns with offset or strides will most likely result in multiple transactions needed to perform the data exchanges. These inefficient

behaviors are illustrated in Figure 3, where more than one transaction are needed to transfer the data. [7]

The shared memory is also divided into memory banks which are equally sized areas in the memory that can be accessed simultaneously. The rules for simultaneous accesses and efficient bandwidth usage vary between different devices. Generally if any of the banks have multiple threads accessing to it at the same time, a conflict will occur and the requests will be serialized. However NVIDIA's recent technology allows concurrent access from multiple threads to different words in the same shared memory bank [8].

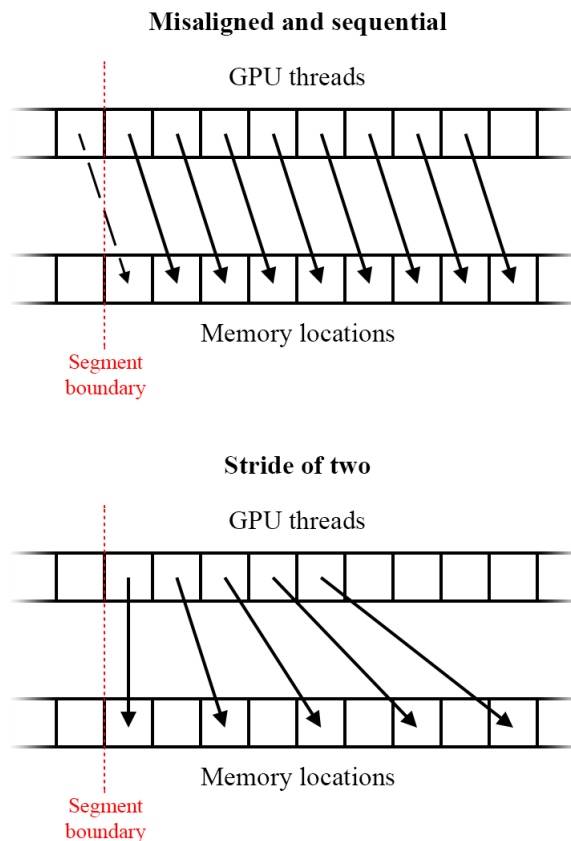


Figure 3. Performance degrading memory access patterns.

2.2.3. Occupancy

Occupancy is defined as the amount of parallel threads divided by the maximum amount of parallel threads in a GPU. Therefore occupancy is a measure of thread-level parallelism. Another contributing factor to computational performance is instruction-level parallelism which is a measure of how many operations a multiprocessor can perform simultaneously.

A common pitfall in hiding memory access latencies is to try raising the occupancy by increasing the amount of threads and thread blocks in parallel execution, but according to Volkov [9] it might be more useful to lower occupancy to gain better throughput with instruction-level parallelism. This is because when less threads are used, more registers are possibly available for a single thread to use and the data in

process will not spill to slower memories. Commonly the on-chip shared memory is considered to be as fast as the registers but this is a misconception because the maximum bandwidth available in register transfers is about six times greater than in shared memory.

2.2.4. Branch divergences

Since GPUs use mainly data parallelism, efficient use of GPU resources require that all concurrent threads follow the same execution paths. Conditional clauses in the code do not conform to this requirement, and using them should be kept to minimum. When there is a condition in the code, part of the threads need to stall while other threads continue executing the branch. This may result in significant performance losses. [7] One possible strategy to minimize the performance penalties of branch divergences is to use branch predication, where the use of jump instructions is replaced with predicate logic. The output of both branches are computed, and only after that a decision is made which output to use. This technique may increase the amount of instructions in the code but can be more effective in pipelined execution. Example code of branch predication is shown in Listing 1.

Listing 1. Code sample of branch predication.

<pre>// Without predication if (condition) x = y; else x = z;</pre>	<pre>// With branch predication pred = -(condition); x = (pred & (y)) ((~pred) & z);</pre>
---	--

2.3. OpenCL standard

OpenCL aims to bring parallelization to another level by mixing different computational units together under same standard to be usable with a single application programming interface. Originally developed by Apple Inc., OpenCL is currently overseen by Khronos Group, a nonprofit industry consortium for creating open standards for graphics, rich media and parallel computation [10]. The OpenCL platform model is shown below in Figure 4.

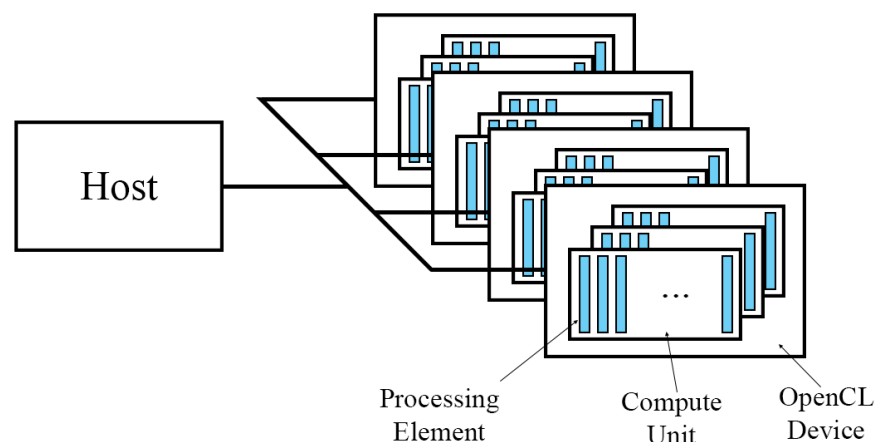


Figure 4. Platform model of OpenCL framework.

The execution model of the OpenCL requires that the problem to solve with the framework should have some dimensionality, which may be up to three dimensions. The division of the workload can be seen in Figure 5. Each work-item is uniquely indexed globally and intended to be processed by one kernel invocation, and multiple work-items are grouped and processed simultaneously in *warp* or *wavefront* -sized batches inside compute units. A work-item is an abstract concept and can be defined according to the algorithm and data requirements in question. OpenCL kernels are executed in SPMD style so that many kernel instances of a single kernel process the varying data of work-items in parallel in multiple processing elements. Different kernel runs are queued and run sequentially. Before running the kernels they are compiled in runtime. [11]

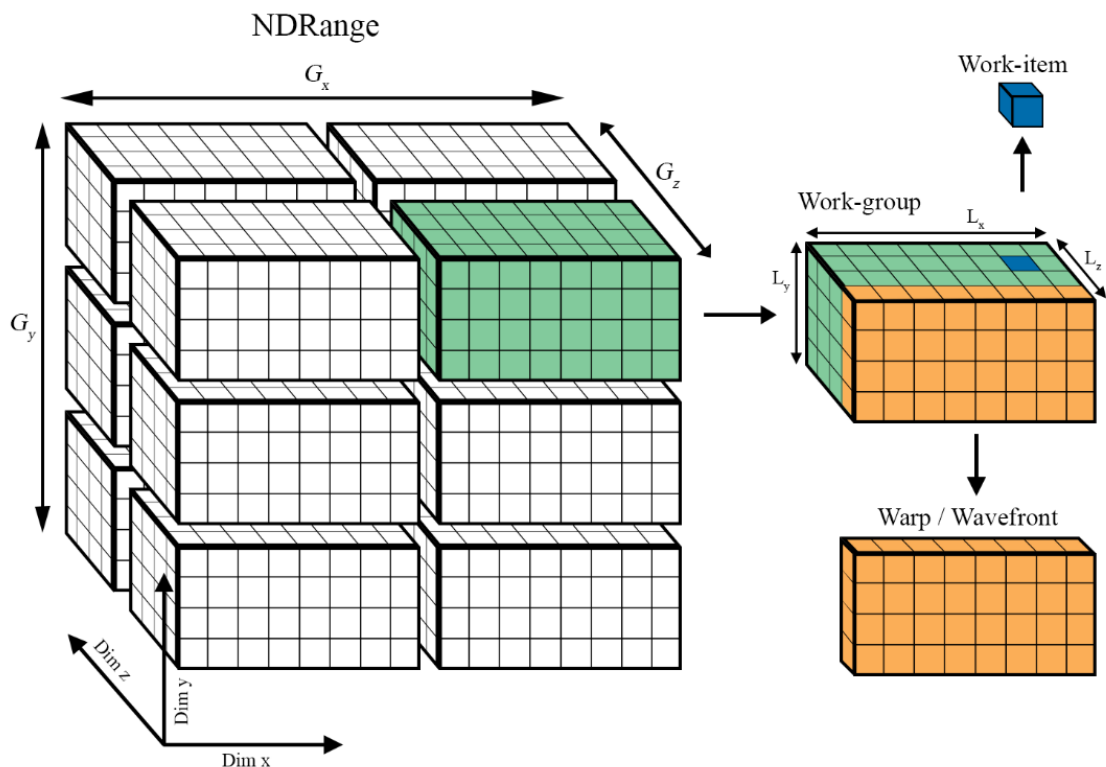


Figure 5. N-dimensional presentation of work-groups and -items.

The memory model in OpenCL follows the available memories in GPUs closely. There are some differences in terms used to describe memory regions, e.g. local memory equals shared memory. A brief illustration of the memory model can be seen in Figure 6. [11]

2.4. CUDA

NVIDIA introduced CUDA to the industry in 2006, and it was the first available GPGPU framework [12]. It defines a new programming language for NVIDIA's GPUs which extends the C language standard with data-parallel constructs. Currently CUDA

is not compliant for overall heterogeneous computing, because it is designed to work only on NVIDIA's GPU devices.

The CUDA and OpenCL bear many similarities to each other. Both are implemented and used with a variation of C language, threads are processed in groups and the memory models are very similar. However fundamental differences exist in the target platform design, where OpenCL is truly open heterogeneous solution for general-purpose parallel programming across various devices and CUDA is mainly designed for proprietary NVIDIA devices.

The CUDA is designed as a scalar architecture, and because NVIDIA's OpenCL implementation depends on the CUDA architecture, the vector data types specified in the OpenCL specification are mostly not useful in terms of performance on NVIDIA GPUs. However using the vector data types can add more portability and convenience to the code and their usage may therefore be justified when developing using NVIDIA cards.

At the time of writing the tools are arguably more mature in CUDA SDKs. Albeit NVIDIA GPUs do support OpenCL, the company does not provide principal debugging tools for the framework.

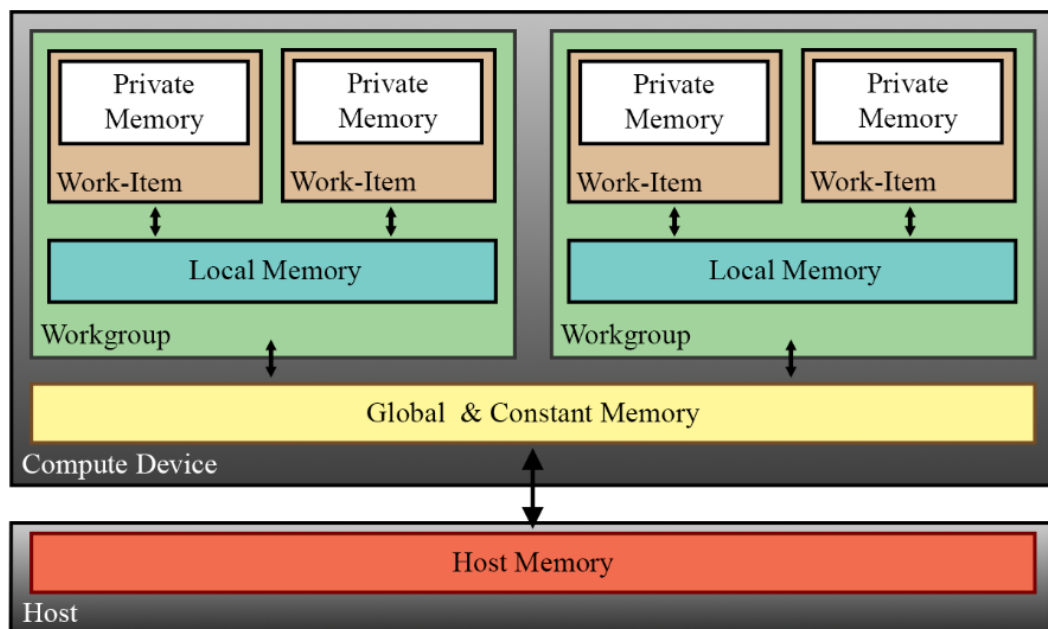


Figure 6. Memory model used in OpenCL.

3. DENOISING ALGORITHMS

In this chapter the most common existing image denoising algorithms are discussed. The algorithms are split into two categories: spatial- and transform-domain filters. In Section 3.1 spatial-domain denoising filtering is studied, and in Section 3.2 transform-domain denoising filters are examined. Section 3.3 describes briefly the K-SVD dictionary learning denoising method, and Section 3.4 is reserved for the main topic BM3D algorithm solely. A summary of the shown algorithms is shortly presented in Section 3.5.

3.1. Spatial-domain filtering

Direct operations on the pixels of an image is described with the term spatial-domain filtering. Usually a convolution operation is used on the original image in denoising algorithms. The convolution between the original image f and the filter impulse response or “mask” w can be presented as

$$f'(x, y) = (w * f)(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b w(s, t) f(x - s, y - t), \quad (2)$$

where f' is the filtered image, x and y image coordinates, a and b the horizontal and vertical sizes of the filter mask and s and t the indices of the filter mask. This however results in a horizontally and vertically mirrored output image. The mirroring can be corrected by using correlation instead, where the coefficients are mirrored pre-emptively as follows

$$g(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b w(s, t) f(x + s, y + t). \quad (3)$$

In general denoising operations can be considered as a blurring or smoothing operations where the highly dense AWGN is averaged out. The challenge is to preserve the small details and sharp object edges in the noise reduction process. One example of a smoothing operation is to calculate the moving average for the image. This can be done with the convolution or correlation Equations 2 or 3 above as

$$g(x, y) = \frac{\sum_{s=-a}^a \sum_{t=-b}^b w(s, t) f(x + s, y + t)}{\sum_{s=-a}^a \sum_{t=-b}^b w(s, t)} \quad (4)$$

with the coefficients being weighted as, for example,

$$w = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}. \quad (5)$$

Since the coefficients are symmetrical in the averaging filter, either convolution or correlation can be used. A small averaging filter mask is very simple to implement and

can do decent denoising on very large images where the noise is relatively small-sized compared to the details on the image. [13]

3.1.1. Total variation

In 1992, Rudin et al. presented the original concept of total variation based noise removal algorithms [14]. The main objective is to restore the signal by minimizing the total variation norm of the estimated solution. This can be described as a differential equation optimization problem as follows:

$$\min_x \left\{ F(x) = \frac{1}{2} \sum_{n=0}^{N-1} (y(n) - x(n))^2 + \lambda \sum_{n=1}^{N-1} |x(n) - x(n-1)| \right\}, \quad (6)$$

where y is the noisy input signal, x is the estimated denoised signal, N is the amount of samples, and the second term defines the actual total variation. λ is a regularization parameter which alters the amount of total variation in the result. With $\lambda = 0$ the output remains unchanged compared to the input. The first term is a measure of distance as a sum of squared differences to deduce the sample's closeness after TV reduction. Likewise to other denoising methods the closeness is used to keep the edges sharp in the noise reduction.

Since the concept defines an optimization problem which requires solving a differential equation, multiple algorithms have been developed to overcome this. One of the more modern solutions for 2D image denoising is Chambolle's algorithm [15]. TVD algorithms have been surpassed by newer methods like NL-means and BM3D, but are still included in this thesis for more complete comparison.

3.1.2. Neighborhood filters

All image denoising filters that are designed to restore a pixel by averaging its neighboring similar gray level pixels are considered as neighborhood filters. These filters are generally considered to be of the form

$$\hat{f} = \frac{\sum_{j \in \Omega} w_{i,j} y_j}{\sum_{k \in \Omega} w_{i,k}}, \quad (7)$$

where Ω is the area of the input image, and the estimated image \hat{f} is a weighted average of the noisy image y and the weights w may depend on values of y .

Buades et al. present Yaroslavsky, SUSAN and Bilateral filters briefly in their research on developing the NL-means algorithm [16, 17]. According to these published papers these filters do not differ significantly in practice. Typically these filters do not blur the edges due to the fact that pixels are averaged based on the reference pixel's region. However since the pixel gray levels are individually compared the results are very vulnerable to noise in single pixels. Also these methods may generate artificial shocks in the output images.

3.1.3. Non-local means

Non-local means is one of the most successful denoising methods available in spatial-domain filters. In contrast to neighborhood filters, the algorithm takes into account not only the neighboring pixels of a reference pixel but compares non-local patches of pixels to each other. Patches are fixed-size, e.g. 3x3, 5x5, 7x7, etc. Non-locality of the algorithm comes from the fact that the patches can in theory locate anywhere in the image. In practice, the patch locations are limited inside a fixed-size search window to reduce computation. Euclidean distances between patches are measured and each output pixel is a sum of weighted averages, where the weights come from the patch distances in the neighborhood of the reference patch. Larger weights are given to pixels with a similar intensity neighborhood, i.e. patches that are similar to the reference patch. [16]

The discrete form of the NL-means algorithm to estimate a pixel value $NL[v](i)$ at location i is

$$NL[v](i) = \frac{1}{Z(i)} \sum_{j \in \Omega} w(i, j) v(j), \quad (8)$$

where Ω is the area of the original noisy image, i and j two pixel locations in the image and $v(i)$ is the unfiltered pixel value at i . The normalizing factor $Z(i)$ is given by:

$$Z(i) = \sum_{j \in \Omega} w(i, j), \quad (9)$$

and $w(i, j)$ is the weighting function which can vary in applications. Usually a Gaussian weighting function is used:

$$w(i, j) = e^{-\frac{F * |v(\mathcal{N}_i) - v(\mathcal{N}_j)|^2}{h^2}}, \quad (10)$$

where \mathcal{N}_k denotes a fixed-size square neighborhood centered at pixel k and h is a degree of filtering parameter. The function essentially calculates the Euclidean distances between patches by taking sums of squared differences of them and weighting the distances with Gaussian distribution. Symbol F denotes a weight function for SSD calculation which usually is a box function or another Gaussian function. With F being a Gaussian function it is possible to weight the distances to be focused on the pixels near center of a patch, while the box function weights simply 1 for all pixels inside a patch and 0 for the rest. [16]

3.2. Transform-domain filtering

In Section 3.1 we looked at typical denoising methods in spatial-domain, how they access and change data based on the actual pixel values. But some information can be more easily and efficiently manipulated in the frequency domain, where the image data is decomposed into its frequency components, e.g. to a sum of cosine functions. There are several options to choose the transform method from, and different methods vary

in computational complexities and output characteristics. Generally these transform methods are split into two categories: wavelet and Fourier transform based methods.

According to the Fourier theorem [13] the convolution operation used in spatial-domain is equal to single multiplication operation in transform-domain. This means that convolution in Equation 2 can be rewritten as

$$f' = w * f = W \cdot F, \quad (11)$$

where W and F are the transform-domain representations of the filter mask and original image. The multiplication in transform-domain is computationally significantly less complex than the convolution operation in spatial-domain, which often leads to faster algorithm implementations, although the transformation back and forth spatial to transform-domain adds overhead to the processes. Therefore a lot of research has been done in the field of frequency-domain transformations. There are multiple algorithms for both wavelet and Fourier transforms available. Currently some of the fastest DWT methods are Haar [18] and Walsh-Hadamard [19] transforms. Some of the fastest DCT methods are BinDCT [20], AAN DCT [21] and Loeffler's DCT [22].

3.2.1. Wiener filter

One of the earliest known image restoration strategies is to use a Wiener filter. The filter produces an output which minimizes the statistical error based on a cost function. Usually mean squared error is used as a cost function:

$$e^2 = E \{(f - \hat{f})^2\}, \quad (12)$$

where f is the original signal and \hat{f} is the estimate of the original signal after filtering, and E denotes the expectation. The task is to find the coefficients that provide the output up to the expectation. The filter can be presented for images in transform-domain as

$$\hat{F}(u, v) = \left[\frac{1}{H(u, v)} \frac{|H(u, v)|^2}{|H(u, v)|^2 + S_\eta(u, v) / S_f(u, v)} \right] G(u, v), \quad (13)$$

in which $\hat{F}(u, v)$ is the estimated original signal and $G(u, v)$ is the observed noisy signal in transform-domain, $H(u, v)$ is the PSF (point spread function) used, and S_k is the power spectral density of a signal. The ratio of power spectral densities is the inverse of the signal-to-noise ratio, and is most likely unknown. The ratio can be replaced with some constant value R , which can be evaluated empirically in the filtering process. [23]

Because the Wiener filter is about minimizing the effect of a cost function and is not very complex to implement, it is also widely used within other image restoration algorithms. Some examples of these algorithms can be seen in the next sections.

3.2.2. BLS-GSM

BLS-GSM, proposed by Portilla et al. in 2003, is a modern image denoising method that utilizes wavelets. Its basic idea is to model the original image properties of each neighborhood in a noisy image in wavelet domain using a mixture of scaled Gaussians (GSM), and then estimate the reference image coefficients by computing the Bayes least squares (BLS) estimates. The final solution is a weighted sum of local Wiener estimates of each neighborhood produced from covariance matrices of known noise and observed neighborhoods. [24, 25]

The Gaussian scale mixture is used to model the wavelet pyramid coefficients, and is defined as

$$x = \sqrt{z}\mathbf{u}, \quad (14)$$

where z is an independent positive scalar random variable and \mathbf{u} is a zero-mean Gaussian vector [26]. The square root of z is used to simplify the expressions in further equations. The model can be applied to the noisy image model as

$$y = x + \eta = \sqrt{z}u + \eta. \quad (15)$$

The BLS estimation is calculated as

$$E\{x|\mathbf{y}\} = \int_0^\infty p(z|\mathbf{y})E\{x|\mathbf{y}, z\}dz, \quad (16)$$

which essentially is a $p(z|\mathbf{y})$ weighted sum of $E\{x|\mathbf{y}, z\}$ local Wiener estimates, where

$$E\{x|\mathbf{y}, z\} = \frac{z\mathbf{C}_u}{z\mathbf{C}_u + \mathbf{C}_\eta} \mathbf{y}. \quad (17)$$

The variables \mathbf{C}_u and \mathbf{C}_η denote the covariance matrices computed from the observed and known noise neighborhoods. [24, 25]

3.3. K-SVD

Some denoising methods are based on machine learning. K-SVD algorithm is one of the existing dictionary learning algorithms that has been applied successfully to image denoising. The goal is to be able to reconstruct a signal by using a sparse representation of the signal over an overcomplete set of basis (the dictionary), and to learn from the provided data and add the learnings to the dictionary. To have results the dictionary needs to be initialized with some basis to begin with, e.g. basis of DCT or a training set from a database. [27]

K-SVD algorithm is essentially a generalization of k -means [28] clustering algorithm. The denoising algorithm has two stages: sparse coding stage and dictionary update stage. In the sparse coding stage an approximated solution for the sparse representation of the signal is solved from the NP-hard problem

$$\min_{\alpha} \|\alpha\|_0^0 \text{ s. t. } \|\mathbf{D}\alpha - y\|_2^2 \leq (C\sigma)^2, \quad (18)$$

by using a pursuit algorithm, e.g. OMP [29]. The symbol α denotes the sparse representation vector, \mathbf{D} the dictionary, y the noisy image and C the noise gain.

In the dictionary update stage the contents of the atoms in the dictionary are re-evaluated. For each atom that is active, a set of patches from α is collected and used to compute the error in representation:

$$\mathbf{e}_{ij}^l = y_{ij} - \sum_{m \neq l} \mathbf{d}_m \alpha_{ij}(m), \quad (19)$$

where \mathbf{E}_l is the error matrix with columns \mathbf{e}_{ij}^l and \mathbf{d}_m an atom in the dictionary. Singular value decomposition (SVD) is then applied to the error matrix \mathbf{E}_l and the atom values are updated with the resulting coefficient values. [27]

3.4. BM3D algorithm

In this section the main topic of this thesis, BM3D algorithm [1], is studied. The BM3D algorithm combines the best of both spatial- and transform-domain methods. Likewise to NL-means algorithm, BM3D utilizes also patch distances to gather similar pixel groups together to reveal self-similarities of the patches under the noise. This patch comparison is called block-matching, where the name of the algorithm also refers to. The “3D” part of the name comes from stacking the matching blocks into a 3-dimensional image to be able to use collaborative filtering on them. This collaborative filtering reveals the finest details shared by the blocks while keeping the unique features of individual blocks mostly untouched. [1, 2]

There are multiple variations and extensions to the algorithm for different applications. The basic algorithm named BM3D is mainly for processing grayscale still images. Other algorithms are listed below:

- C-BM3D – a color extension to the basic algorithm,
- V-BM3D – an extension for grayscale video processing,
- BM3D-SH2D and BM3D-SH3D – a variation to sharpen an image using BM3D filter,
- BM3D for deblurring – a variation to use BM3D for deblurring an image,
- SA-BM3D – a variation with shape-adaptive grouping via block-matching,
- BM3D-SAPCA – a variation using shape-adaptive principal component analysis. [2]

3.4.1. Structure

The algorithm is divided into two major steps as shown in Figure 7. The first step focuses on producing an image which has significantly less noise than the noisy image. This image is a basic estimate of the original noiseless image. In the second step the basic estimate is used as a block-matching base for empirical Wiener filtering. According to Dabov, this second step has been empirically confirmed to improve the image quality compared to the first step output [2].

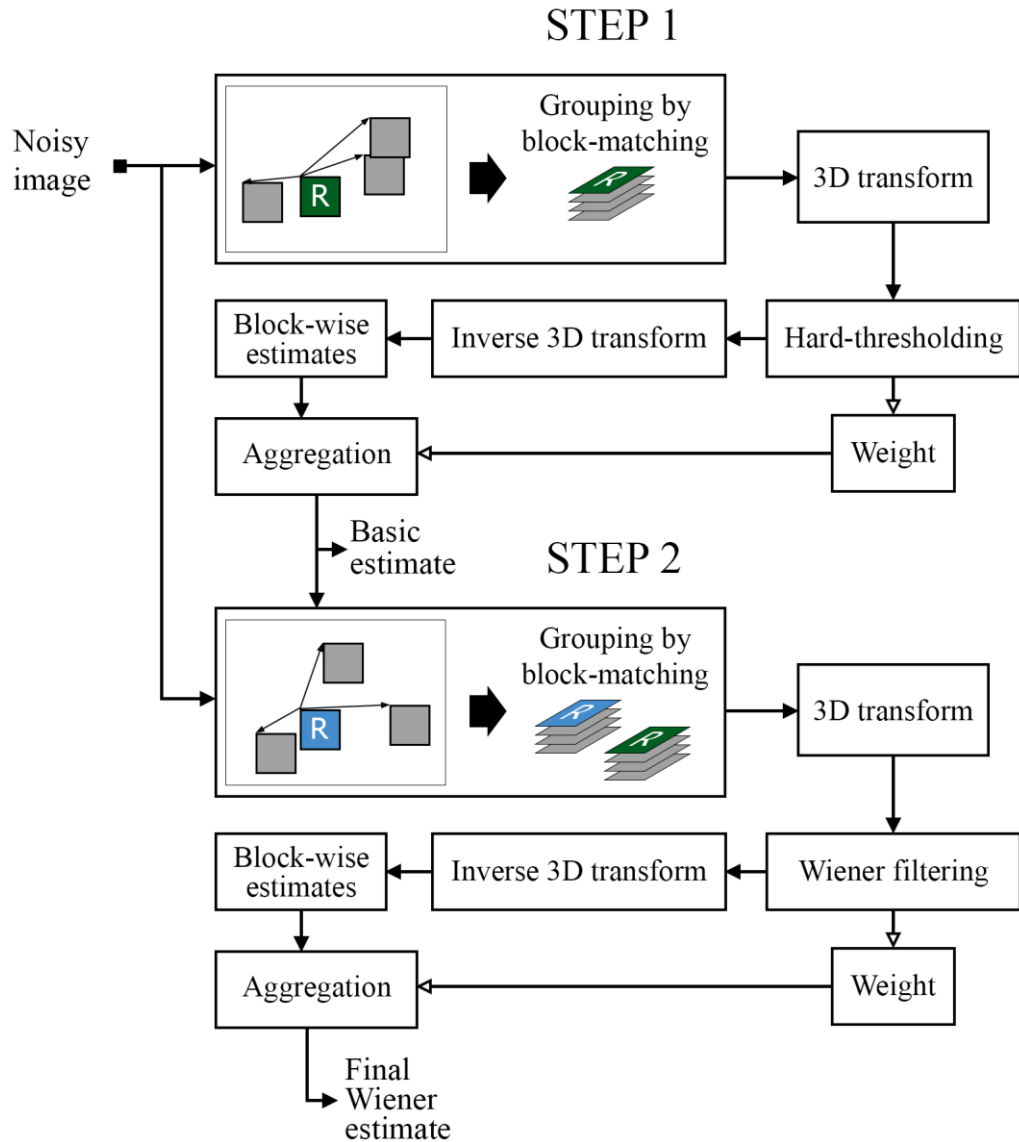


Figure 7. High-level flowchart of the BM3D algorithm.

3.4.2. Block-matching

The grouping of the blocks is realized by block-matching. Blocks that have high similarity with the reference block are considered for a group. The similarity is measured with a distance function

$$d^{\text{noisy}}(Z_{x_R}, Z_x) = \frac{\|Z_{x_R} - Z_x\|_2^2}{(N_1^{\text{ht}})^2}, \quad (20)$$

where x is a 2D coordinate in the noisy image, x_R the coordinate of the reference block, N_1^{ht} the length of a side of a square block and Z_k a noisy fixed-size block located at

coordinate k . The distance is effectively measured as sums of squared differences between blocks.

According to Dabov et al. [1] the distance function presented in Equation 20 produces good results when the standard deviation σ is low and the size N_1^{ht} is not too small, but with high σ or small N_1^{ht} the probability densities of distances can overlap heavily. Thus to avoid this problem the block-matching can also be done in transform-domain with thresholded coefficients:

$$d(Z_{x_R}, Z_x) = \frac{\left\| \Upsilon' \left(\mathcal{T}_{2\text{D}}^{\text{ht}}(Z_{x_R}) \right) - \Upsilon' \left(\mathcal{T}_{2\text{D}}^{\text{ht}}(Z_x) \right) \right\|_2^2}{(N_1^{\text{ht}})^2}, \quad (21)$$

where $\mathcal{T}_{2\text{D}}^{\text{ht}}$ denotes a 2-dimensional linear transform and Υ' a hard-thresholding operator.

The measured distances are composed into a set of coordinates $S_{x_R}^{\text{ht}}$, where only the nearest distances are taken into account by thresholding them with a threshold-value $\tau_{\text{match}}^{\text{ht}}$:

$$S_{x_R}^{\text{ht}} = \{x \in X : d(Z_{x_R}, Z_x) \leq \tau_{\text{match}}^{\text{ht}}\}. \quad (22)$$

3.4.3. Collaborative filtering

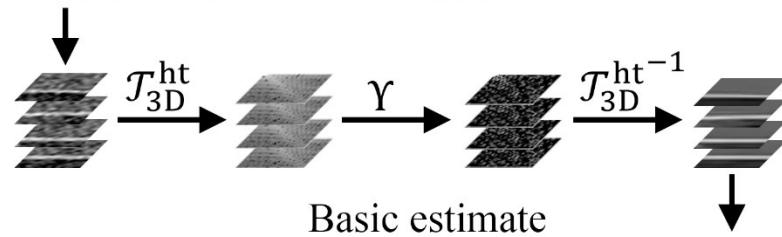
Now the nearest blocks for a reference block are known and can be stacked together to form a group of blocks for further processing. Linear transformation is applied to the group $Z_{S_{x_R}^{\text{ht}}}$. This transformation reveals the shared features between blocks efficiently. In the transformed data the shared features have large coefficients while the noise is mostly presented with small coefficients. By thresholding these small coefficients to zero it is possible to reduce noise significantly and keep the unique details in place. The collaborative filtering can be presented as

$$\hat{Y}_{S_{x_R}^{\text{ht}}}^{\text{ht}} = \mathcal{T}_{3\text{D}}^{\text{ht}^{-1}}(\Upsilon(\mathcal{T}_{3\text{D}}^{\text{ht}}(Z_{S_{x_R}^{\text{ht}}}))\}, \quad (23)$$

where $\hat{Y}_{S_{x_R}^{\text{ht}}}^{\text{ht}}$ is the resulting filtered group of blocks in spatial domain and Υ the hard-thresholding operator. The 3D linear transform $\mathcal{T}_{3\text{D}}^{\text{ht}}$ is usually implemented as separate 2D and 1D linear transformations. If σ is high, the 2D linear transformed blocks from distance calculation can be used here to avoid redundant transformations. The filtering process is illustrated in Figure 8 and the 3D transform can be seen in Figure 9.

Different types of linear transformations can be used for the filtering and the output quality may vary based on the selected transformation type combined with hard thresholding. Also each transform has their own computational complexities. In general, Haar wavelet transformation and Walsh-Hadamard transformation are the fastest options, because these transformations can be calculated with only a few addition operations.

Noisy image



Basic estimate



Figure 8. Processing of one block with collaborative filtering. The reference block is shown with blue borders and the nearest matching blocks with red borders. The filtered blocks are repositioned and aggregated to the output image.

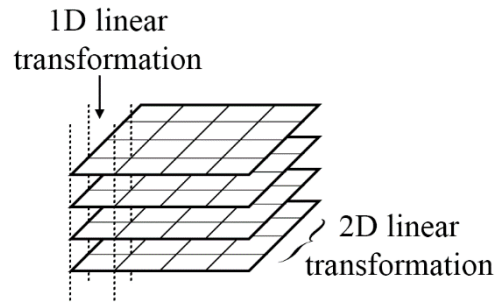


Figure 9. 3D linear transformation as separate 2D and 1D transformations.

3.4.4. Aggregation

After filtering the group of blocks, they need to be put back in place to the result image. Each group may however have different amount of noise to begin with and this should be compensated. Dabov et al. propose that instead of taking into account all the effects of differencing variances and biasing within individual pixels, it is enough to give more weight to blocks with less noise and less weight to those with much noise. This is done by calculating the amount of non-zero coefficients in a group $N_{\text{har}}^{x_R}$ after thresholding and creating the weight $w_{x_R}^{\text{ht}}$ based on that as

$$w_{x_R}^{\text{ht}} = \begin{cases} \frac{1}{\sigma^2 N_{\text{har}}^{x_R}}, & \text{if } N_{\text{har}}^{x_R} \geq 1 \\ 1, & \text{otherwise} \end{cases}. \quad (24)$$

With the weight $w_{x_R}^{\text{ht}}$ the group can be aggregated into the result image as a weighted average. Because the blocks inside a group may overlap, the result of summing the weighted averages needs to be normalized by dividing the result with the sum of the weights. The result image \hat{y}^{basic} becomes then

$$\hat{y}^{\text{basic}}(x) = \frac{\sum_{x_R \in X} \sum_{x_m \in S_{x_R}^{\text{ht}}} w_{x_R}^{\text{ht}} \hat{Y}_{x_m}^{\text{ht}, x_R(x)}(x)}{\sum_{x_R \in X} \sum_{x_m \in S_{x_R}^{\text{ht}}} w_{x_R}^{\text{ht}} \chi_{x_m}(x)}, \quad \forall x \in X \quad (25)$$

where χ_{x_m} is an operator which is 1 in the area of a block and 0 outside, and X is the area of the input image.

3.4.5. Wiener filtering

To further improve the quality of the BM3D algorithm, in step 2, Wiener filter is used in co-operation with the basic estimate from step 1. The SSD distance calculation is performed again but with the basic estimate \hat{Y}^{basic} as the base image:

$$S_{x_R}^{\text{wie}} = \left\{ x \in X : \frac{\|\hat{Y}_{x_R}^{\text{basic}} - \hat{Y}_x^{\text{basic}}\|_2^2}{(N_1^{\text{wie}})^2} \leq \tau_{\text{match}}^{\text{wie}} \right\}. \quad (26)$$

$S_{x_R}^{\text{wie}}$ denotes the set of nearest block coordinates for a reference block positioned at x_R in the basic image. The empirical Wiener filter shrinkage coefficients for collaborative filtering in the second step are defined as

$$\mathbf{W}_{S_{x_R}^{\text{wie}}} = \frac{|\mathcal{T}_{3D}^{\text{wie}}(\hat{Y}_{S_{x_R}^{\text{wie}}}^{\text{basic}})|^2}{|\mathcal{T}_{3D}^{\text{wie}}(\hat{Y}_{S_{x_R}^{\text{wie}}}^{\text{basic}})|^2 + \sigma^2}. \quad (27)$$

The stack of blocks $Z_{S_{x_R}^{\text{wie}}}$, which are the corresponding blocks at coordinates $S_{x_R}^{\text{wie}}$ in the noisy image Z , is 3D linear transformed and filtered by using the Wiener shrinkage coefficients $\mathbf{W}_{S_{x_R}^{\text{wie}}}$. Finally the filtered stack is inverse transformed back to spatial domain. This process is as follows:

$$\hat{Y}_{S_{x_R}^{\text{wie}}}^{\text{wie}} = \mathcal{T}_{3D}^{\text{wie}^{-1}}(\mathbf{W}_{S_{x_R}^{\text{wie}}} \mathcal{T}_{3D}^{\text{wie}}(Z_{S_{x_R}^{\text{wie}})}), \quad (28)$$

where the output $\hat{Y}_{S_{x_R}^{\text{wie}}}^{\text{wie}}$ is the final denoised image estimate from step 2. This image still needs to be aggregated and normalized as in step 1 (Equation 25) with weights

$$w_{x_R}^{\text{wie}} = \sigma^{-2} \|\mathbf{W}_{S_{x_R}^{\text{wie}}}\|_2^{-2}, \quad (29)$$

where the weights are derived from the l^2 -norm of the filter coefficients.

3.5. Summary

The existing denoising methods were briefly described in the previous sections. The denoising methods are generally split into three major categories:

- spatial-domain filters,
- transform-domain filters and
- machine learning algorithms.

In general, the denoising methods are designed to filter out simulated AWGN. The current state-of-the-art for denoising is the BM3D algorithm, which provides superior PSNR results in simulated AWGN filtering comparisons most of the time. BM3D has evolved from the earlier available methods and combines the best of them by utilizing non-locality with block-matching, transform-domain benefits with collaborative filtering and optimal minimum MSE output with the final Wiener filter.

When considering the computational complexities of the algorithms and their suitability for GPGPU computing, the algorithms that require less data to process one sample, produce a singular output item and have less complexity are in general more

suitable. The NL-means algorithm has been proven to be well suited to GPGPU computation. Several implementations of NL-means using GPGPU have emerged in the field of biomedical image processing, where the algorithm has been extended to support also multidimensional denoising of magnetic resonance images or video sequences [30, 31]. With a small search window the computational complexity is low and it is possible to implement the NL-means filter by using solely deterministic calculation. Other GPU implementations of wavelet-based [32], adaptive bilateral filtering [33] and Gaussian blur [12] image denoising algorithms exist also.

4. BM3D FILTER IMPLEMENTATION

The main objective of this thesis was to design an implementation of the BM3D image denoising algorithm presented in Chapter 3 by using the parallel processing methods provided by the OpenCL framework presented in Chapter 2. In this chapter the implementation and its design challenges are discussed. In Section 4.1 the reference implementations are shown, and Section 4.2 describes the new solution in more detail.

4.1. Reference implementations

At the time of writing only one implementation of the BM3D algorithm with open source code exists online. This program, created by Lebrun [34], is done using standard C++ and is compilable on various operating systems with minor changes. The program executes only on host machine CPU, but can be compiled to use multi-threading with OpenMP library [35]. It has also some features, such as color channel processing, integral images and standard deviation estimation, which are missing from the presented implementations.

Another implementation for reference use is the application created by the original author of BM3D algorithm [1]. However the source code of the application is not available in public and therefore design choices cannot be fully compared with the implementations presented in this paper. The performance can still be measured and will be compared in Chapter 5.

4.2. OpenCL implementation

The work was started by designing and building a prototype MATLAB model of the algorithm for validating the functioning of the upcoming OpenCL implementation version. This prototype model had no parallelism included and was very inefficient in terms of performance. However the benefit was to have better knowledge of the algorithm beforehand.

The actual OpenCL implementation was done using ANSI C code in the host application and OpenCL version 1.1 in the kernel code. An OS independent host application using Qt framework was also created for testing the OpenCL program in different operating systems, e.g. in Android.

The OpenCL kernel code consists of three kernels: `calc_distances`, `bm3d_basic_filter` and `bm3d_wiener_filter`. A data flow diagram of the kernels is shown in Figure 10. Both steps of the BM3D algorithm use the `calc_distances` kernel first and then the respective kernel. The `calc_distances` kernel does the block-matching step of the algorithm described in Section 3.4.2. It saves the positions of N_1 nearest blocks for each reference block into a memory area in global memory in a cached manner. These positions are used in multiple kernels simultaneously.

As the naming suggests, the `bm3d_basic_filter` kernel does the step of creating the basic estimate image described in Sections 3.4.3 and 3.4.4. Each kernel instance produces a singular portion of the output image. The size of this portion is fully configurable and has effect on the performance. The parallelization of the algorithm is discussed in more detail in the following Section 4.2.1. Lastly the

`bm3d_wiener_filter` kernel is responsible for applying the Wiener filtering step, described in Section 3.4.5, between the noisy original and the basic estimate images. The kernel functions essentially in the same way than the `bm3d_basic_filter` kernel, but requires slightly more register memory because two 3D similar block stacks are needed to be kept in memory instead of only one.

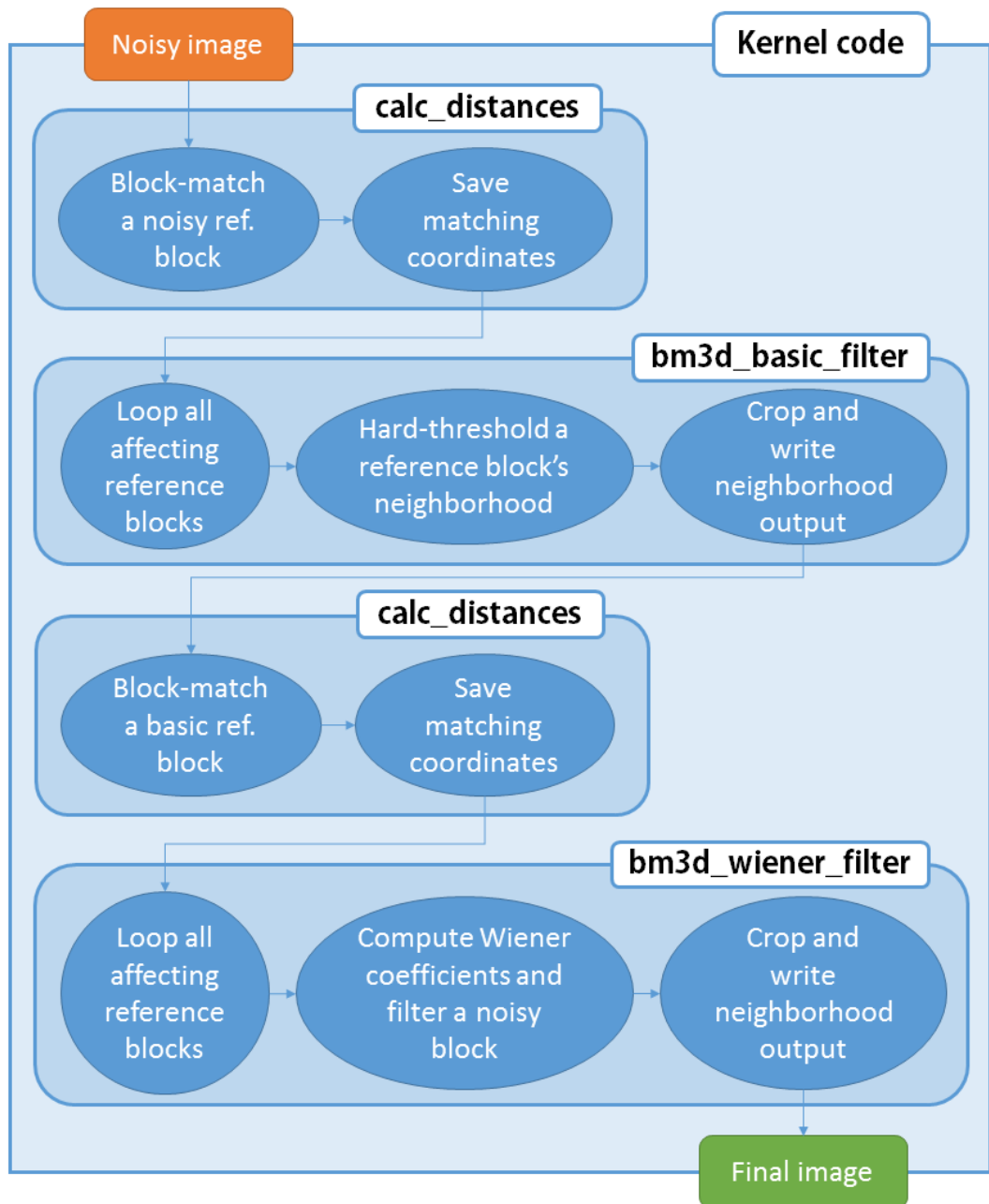


Figure 10. Data flow in the OpenCL and CUDA kernel codes.

For the use of collaborative filtering, the 3D linear transformation functions were implemented using the Loeffler's DCT method [22] for fast computation. As illustrated in Figure 9, the 3D transform is easily separable into multiple transforms. Additionally, an option to use 1D Haar wavelet transform method was provided with

the implementation. Choosing different transform methods may have some effect on the denoising quality and performance when filtering an image.

The kernels were coded to use mostly local variables residing primarily in the register memory area of the execution device. Therefore a single kernel instance requires much space from the register file on a streaming multiprocessor. This is because the register memory is the fastest memory to use on a GPU and local variables are also more convenient to maintain in the code. However using much register memory forces to have less parallel threads running simultaneously to avoid register spillage to slow global memory. Hence the occupancy of a GPU is kept rather low in the implementation.

Using the shared memory efficiently turned out to be problematic in the implementation. The shared memory is usually relatively small in size; for example on NVIDIA GeForce GTX 650 GPU there are only 48 kilobytes available to be used in a work-group. Since one work-group consists of multiple work-items and the data to be processed is a large two-dimensional area, it was not feasible to fit in all the needed common readable data. The option in NVIDIA GPUs to use shared memory as a L1 cache was a good compromise to gain some performance improvement.

Similarly to the reference implementation by Lebrun, an option to multiply the filtered blocks with a Kaiser window was added to the implementation. As originally proposed by Dabov et al. [1], the window may reduce the border effects visible on the final images. The window coefficients are illustrated in the Figure 11 below.

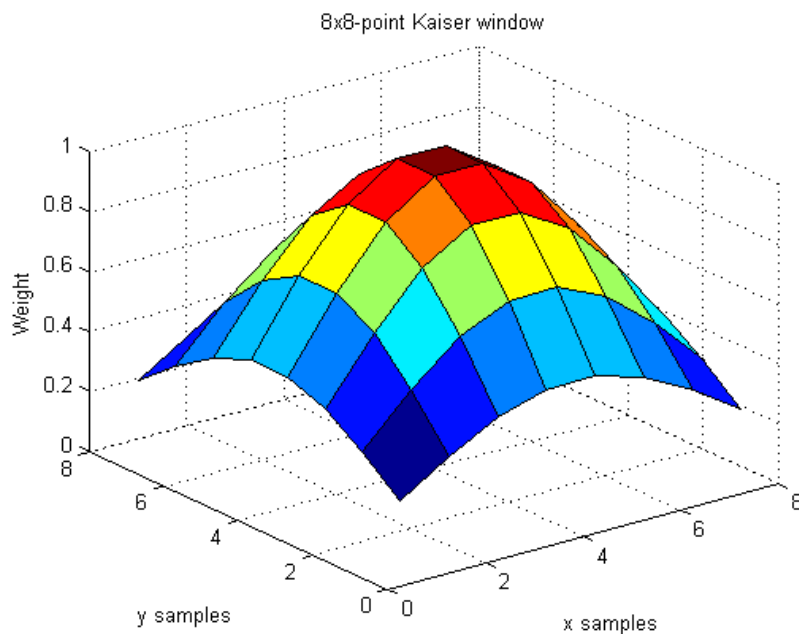


Figure 11. 8x8-point Kaiser window with parameter value $\beta = 2$.

4.2.1. Parallelization

To gain performance benefits from using the GPUs for computing, the BM3D algorithm had to be designed to be run in parallel. The algorithm showed to be difficult to parallelize efficiently due to its nature. Algorithms can have several kinds of data

dependency patterns between input and output memories. Typically these patterns are defined as

- one-to-one,
- one-to-many,
- many-to-one,
- many-to-many,

where the left-hand side refers to the memory blocks in the input data array and the right-hand side refers to the output array. Many-to-N patterns are also known as *gather* operations and N-to-many as *scatter* operations [5]. The BM3D algorithm is the type of many-to-many pattern, since for one reference block in the image, the neighboring pixels need to be read widely and the filtering result will be written based on the same neighboring area, the search window, around the reference block. The *scatter* patterns are the most unpreferred type, because they introduce writing race conditions to the algorithm that must be satisfied.

Two methods for parallelizing the BM3D algorithm were tried when developing the implementation. First was to map reference blocks into work-items directly and process each block in kernel instances separately. The output was written in layered areas in the global memory. Once the work-group had all work-items processed, one work-item was assigned to merge the intermediate results in the layers into group results. Then after all groups had processed their items, the host application merged the group results into the final image. Since the algorithm requires writing the weight map separately, the memory requirements for the output writings are doubled.

The memory requirements raised an issue with the first method in terms of scalability. The intermediate results required much of memory space to store and processing images with greater resolution turned out soon to be impossible. Also removing the layering by serializing the write operations was not an option when seeking a true parallel solution.

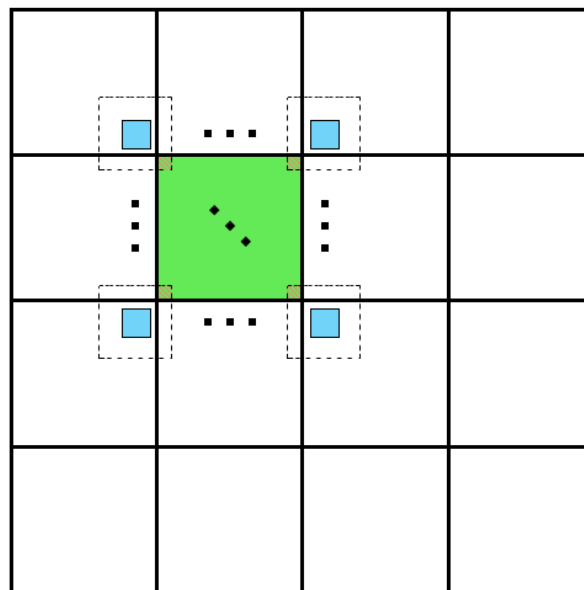


Figure 12. Neighborhoods to be processed by a thread.

The second method is similar to the parallelization used in the reference implementation by Lebrun [34], and can be seen in Figure 12. In the method, the memory dependency pattern of many-to-many in the algorithm is redundantly reduced to be type of many-to-one, which eliminates the need of writing intermediate results thoroughly. This is achieved by dividing the workload equally between threads by assigning fixed-size portions of the output image for each thread to compute. Each portion can be calculated by a single thread by computing and adding also the results of the neighboring reference blocks which have an effect on the portion. The process is illustrated in Figure 13. As said, this introduces a lot of redundant calculation in kernel instances, since the same results are being computed multiple times concurrently. This method adds a lot of latency in computing a single work-item, but reduces the overall memory requirements and adds scalability in terms of resolution. It was noted that the size of a portion should be a multiple of N_{step} , because then the results of the block-matching can be cached and reused more efficiently in all filter threads.

The amount of redundancy depends solely on the size of a portion and the size of a search window, i.e. the ratio between them. In the implementation of Lebrun, the overhead is much smaller since the parallel processing is done using OpenMP and the amount of parallel threads is small, e.g. 4-8 threads processed on a CPU. Therefore the output image is divided into same amount of portions than there are threads, and redundant calculation appears only in the border areas of a portion. Then each thread is responsible for computing a wide area of payload, in contrast to small payload on massively parallel computation.

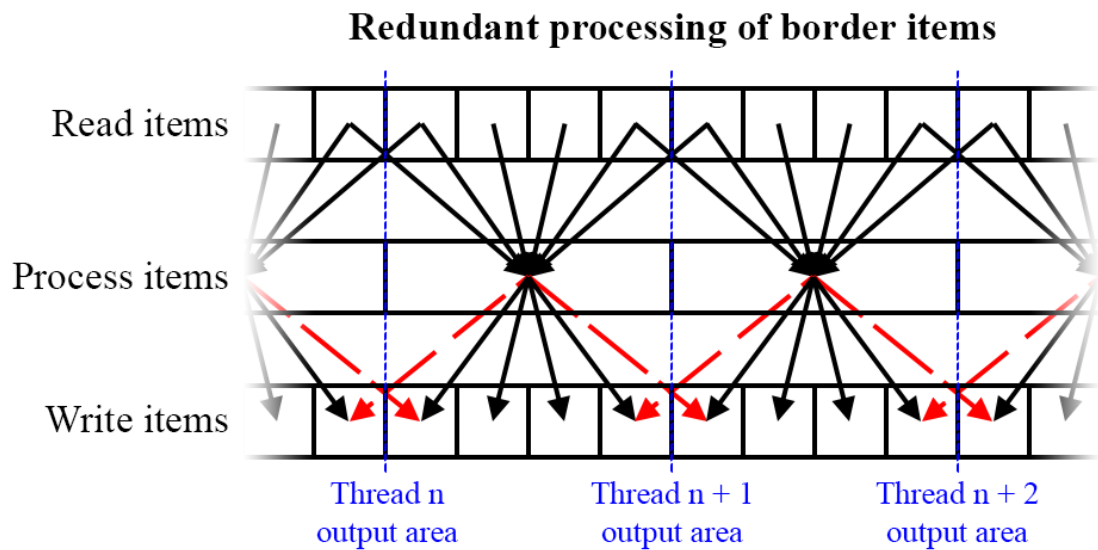


Figure 13. One-dimensional illustration of dividing the work-items in equally sized portions. The results indicated with red lines are discarded as the neighboring threads are responsible for their computation.

4.2.2. CUDA implementations

The OpenCL kernel code was also ported for the CUDA platform to be able to run the filter on a NVIDIA SHIELD mobile device. Porting the code was straightforward and

did not introduce major challenges to have the filter running on a PC. To be able to run the CUDA code on the mobile device, a small Android host application was developed. Fortunately, NVIDIA provided a sample code for loading CUDA applications and shared libraries on SHIELD devices, and modifying it for BM3D usage was not a big task.

4.2.3. Problems occurred

Although the OpenCL framework drives forward the idea of consistency in code, this does not necessarily apply in practice. The BM3D implementation showed that using the private memory excessively in a complex kernel does not function properly on all platforms. For example, running the algorithm with a GPU from ATI generated false results, which can be seen in Figure 14. Also it was not possible to even compile the code on Qualcomm's DragonBoard platform without further modifications, because the compiler crashed always on invocation. It was suspected that the excessive register usage did not spill to global memory properly in both cases. The code executed without problems on NVIDIA cards. This ultimately means that despite the code being in a valid format for OpenCL usage, it usually needs to be modified and optimized for each platform separately.



Figure 14. Computational errors while processing an image with ATI Radeon HD 5450 GPU: the basic BM3D estimate (left) and the final result (right).

5. RESULTS

This chapter presents the results of the designed BM3D algorithm implementation from Chapter 4. Section 5.1 describes the used testing environment in all test cases in Sections 5.2-5.5. In Section 5.6 the overall performance based on the results is evaluated, and in Section 5.7 the results are compared to the reference implementation results.

The result comparisons focus more on comparing the performances of the implementations while the statistical image quality (PSNR) is less considered. The overall performance results are visible in Figure 19 and Figure 20.

The test cases consist of four test images with varying sizes. Images with different sizes were used because the image size is usually proportional with the execution time of a filter. The test cases included only grayscale images with no color information and all the photographs were captured digitally using mobile phone cameras. Three of the images had different levels of synthetic noise added into them and one image had significant amount of natural noise. The synthetic and natural noise differ in their characteristics and therefore the visual quality after filtering and the filter parameter requirements can also vary between these two. The synthetic noise is only AWGN added to an image, but the natural noise can consist of various noise types, such as

- photon shot noise,
- speckle noise and
- thermal noise.

The origins of these noise types were discussed in more detail in Chapter 1.

Two different filter parameter profiles were used, and these profiles are described in detail in Table 2. Five different implementations were compared with each other having the same filter profile in use in each test case. These five implementations are referred as *Lebrun* [34], *Original* by Dabov et al. [1], *OpenCL*, *CUDA* and *Mobile CUDA*. The test execution times were averaged from five sequential test runs.

Table 2. Filter parameter profiles used in test cases.

Parameter	Description	Original profile	Modified profile
N_1	Block size	8	8
N_2^{ht}	1. step similar block count	16	8
N_2^{wie}	2. step similar block count	32	8
N_{step}	Reference block step size	3	7
N_S	Search window size	39	21
$N_{S,step}^{ht}$	1. step search window step size	1	1
$N_{S,step}^{wie}$	2. step search window step size	1	1
τ_{match}^{ht}	1. step similarity threshold	2500	2500
τ_{match}^{wie}	2. step similarity threshold	400	400
λ_{2D}	2D hard-threshold value	0	0
λ_{1D}	1D hard-threshold value	2.7	2.7
β	Kaiser window parameter	2.0	0
\mathcal{T}_{2D}	2D linear transform method	2D-DCT	2D-DCT
\mathcal{T}_{1D}	1D linear transform method	1D-Haar	1D-Haar ¹

¹The implementation by Lebrun uses 1D Walsh-Hadamard transform instead.

5.1. Test environment

The test cases described in the following sections were run on a computer which had an Intel i5-4570 processor as a CPU, 8 gigabytes of RAM and NVIDIA GeForce GTX 650 GPU. The processor has four physical processing cores and no support for Hyper-Threading Technology, and the GPU has compute capability version 3.0 and 384 CUDA cores. All tests were run on 64-bit Windows 7 OS. Additionally, NVIDIA SHIELD which is a mobile Android tablet device with the CUDA platform and a Tegra K1 192 CUDA core GPU was also used for further comparison in the tests.

5.2. Test case 1: 256×256 image with simulated noise

In the first test case a small 256×256 sized image is processed. The image is a close-up shot of a cup with detailed texturing. Synthetic noise is added to the image with standard deviation of 25. The processed images can be seen in Figure 15 and the measurement results in Table 3 with the best result in bold type.



Figure 15. Noisy image with $\sigma = 25$ (left, PSNR 20.39 dB) and the OpenCL BM3D denoised image using the original profile (right, PSNR 28.16 dB).

The results show that when small images are processed, the GPGPU computation does not add any value in the presented implementations. In fact, the performance is significantly better when only a CPU is used. The overhead of using a GPGPU solution is not compensated in the case of small images. A CPU can compute the small image more efficiently because processing a single step in the BM3D algorithm is generally faster with a CPU core than with a GPU core, and when the amount of reference blocks is low the CPU can sequentially process all the blocks faster.

The quality of the output images between the implementations does not vary notably, except the CUDA implementations suffered moderately from having small errors in the right and bottom border areas. The implementation by Lebrun uses Walsh-Hadamard 1D linear transformation in the 3D filtering and the coefficient thresholding produces slightly different results.

Table 3. Test results of test case 1 (256×256 , $\sigma = 25$).

Profile	Implementation	1. step PSNR (dB)	2. step PSNR (dB)	1. step time (ms)	2. step time (ms)	Total time (ms)	Speedup
Original	Lebrun (4x CPU)	27.58	28.28	616.6	670.8	1287.4	0.71x
	Original	27.86	28.31	537.1	375.6	912.7	1.00x
	OpenCL	27.72	28.16	666.5	2299.7	2966.2	0.31x
	CUDA	27.72	27.77	630.1	2245.6	2875.7	0.32x
	Mobile CUDA	27.72	27.77	1339.7	5330.7	6670.4	0.14x
Modified	Lebrun (4x CPU)	27.11	27.63	115.2	139.2	254.4	0.66x
	Original	26.99	27.48	124.5	42.4	166.9	1.00x
	OpenCL	27.19	27.66	216.4	330.4	546.8	0.31x
	CUDA	27.19	27.55	244.1	378.2	622.3	0.27x
	Mobile CUDA	27.19	27.55	339.1	507.6	846.7	0.20x

5.3. Test case 2: FHD image with simulated noise

The image in this test case presents several objects with different levels of focus. The image resolution is 1920×1080 pixels and the noise is generated with SD value of 35. The image and the results can be seen in Figure 16 and Table 4.

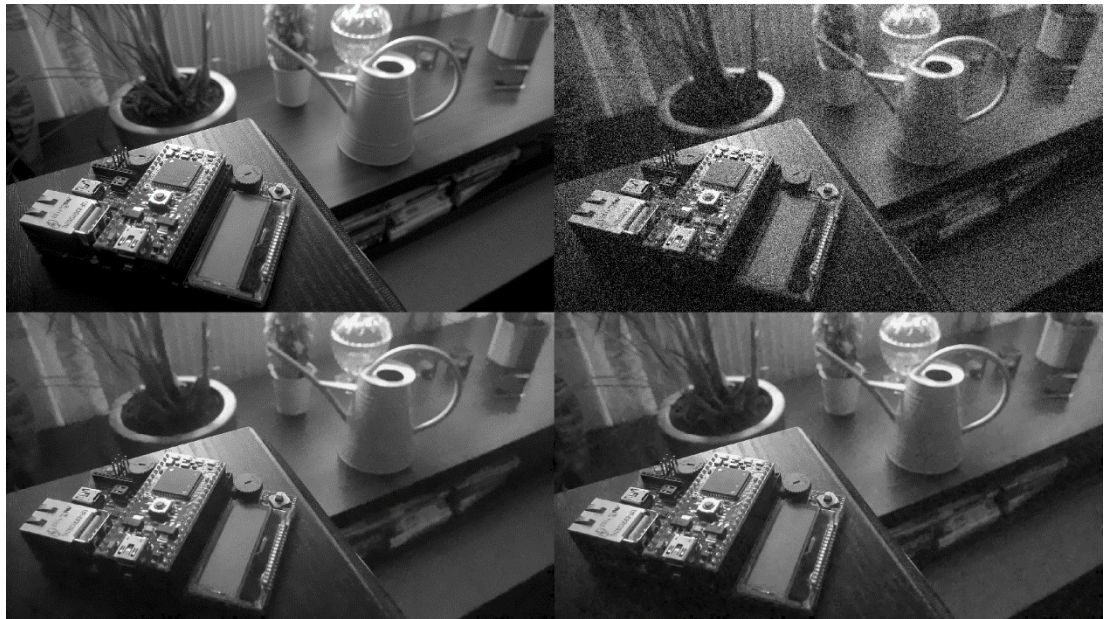


Figure 16. A FullHD image filtered with the modified profile using the CUDA implementation: original image (top-left), noisy image with $\sigma = 35$ (top-right, PSNR 18.10 dB), result after first step in BM3D filtering (bottom-left, PSNR 29.99 dB) and final result of BM3D filtering (bottom-right, PSNR 30.04 dB).

With FullHD images the GPGPU starts to add some value to the computational performance. With the modified profile the computation of the image is over four times faster using a GPU with OpenCL than a CPU, and the PSNR is even better with the

presented implementations. The implementation by Lebrun was not able to process FullHD images, and the reason for this is discussed in more detail in Section 5.6. It took too long for the mobile CUDA implementation to process the FullHD image with the original filter profile and an ANR halted the execution.

Table 4. Test results of test case 2 (1920×1080 , $\sigma = 35$).

Profile	Implementation	1. step PSNR (dB)	2. step PSNR (dB)	1. step time (ms)	2. step time (ms)	Total time (ms)	Speedup
Original	Original	29.83	29.97	22733.3	17422.7	40156.0	1.00x
	OpenCL	29.99	30.05	13461.0	47488.0	60949.0	0.66x
	CUDA	29.99	30.04	12324.0	42101.0	54425.0	0.74x
Modified	Original	28.95	29.28	3638.5	1789.2	5427.7	1.00x
	OpenCL	29.28	29.57	559.0	703.2	1262.2	4.30x
	CUDA	29.28	29.57	629.6	757.8	1387.4	3.91x
	Mobile CUDA	29.28	29.57	1867.0	1715.7	3582.7	1.51x

5.4. Test case 3: UHD image with simulated noise

The third image has a resolution of 3840×2160 pixels, which is four times larger than the FHD resolution. AWGN with large SD ($\sigma = 100$) was simulated into the image. The image and the filtering results are shown in Figure 17 and Table 5.



Figure 17. A 4K UHD image filtered with the OpenCL implementation: original image (top-left), noisy image with $\sigma = 100$ (top-right, PSNR 10.76 dB), result with modified profile after first step of BM3D filtering (bottom-left, PSNR 20.49 dB) and final result of BM3D filtering (bottom-right, PSNR 19.61 dB).

4K images are four times larger than the FullHD images. Therefore significant performance gains can be predicted when a more parallel solution is used. The results

confirm this prediction when the lower quality profile is used. Only the original implementation was able to process the test image with the original filter profile. The others timed out after computing for too long. The reason for this is that with the original profile a GPU thread's private memory usage exceeds its limits. Surprisingly the visual quality is slightly better with the modified profile.

Table 5. Test results of test case 3 (3840×2160 , $\sigma = 100$).

Profile	Implementation	1. step PSNR (dB)	2. step PSNR (dB)	1. step time (ms)	2. step time (ms)	Total time (ms)	Speedup
Original	Original	19.62	19.25	47095.4	58842.3	105937.7	1.00x
Modified	Original	19.05	19.18	11400.0	6499.3	17899.3	1.00x
	OpenCL	20.49	19.61	1070.8	1312.2	2383.0	7.51x
	CUDA	20.49	19.61	1340.8	1488.0	2828.8	6.33x
	Mobile CUDA	20.49	19.61	6563.4	6076.6	12640.0	1.42x

5.5. Test case 4: Real image with natural noise

The influence of the BM3D filter is based on the standard deviation parameter; i.e. larger SD implies less noise and possibly more filtering artifacts. The fourth test case has an image with natural noise instead of simulated, and therefore the SD needs to be estimated. The image has a resolution of 960×1280 , and the image was chosen for the test because of the excessive noise, which is visible in multiple forms. That is, not only AWGN equivalent noise is present. Filtering with SD value of 25 was selected manually and appeared to produce the best results in terms of subjective visual quality. The image and the results can be seen in Figure 18 and Table 6.



Figure 18. A photograph with excessive noise taken in low-light conditions (left) and the BM3D filtered version using modified profile and $\sigma = 25$ (right).

With this image it is impossible to compare the statistical image quality, because the ground truth is not known due to the presence of natural noise. However it was subjectively estimated that the original profile did not produce significantly better visual quality image output. With both profiles the AWGN was efficiently reduced in the image, but other noise types, such as speckle noise, still persist in the image. As the image is smaller than, for example, the FullHD image, the performance gain of parallelization is also smaller.

Table 6. Test results of test case 4 (960×1280 , $\sigma = 25$).

Profile	Implementation	1. step time (ms)	2. step time (ms)	Total time (ms)	Speedup
Original	Original	9007.0	8826.7	17834.0	1.00x
	OpenCL	7617.4	27808.0	35425.4	0.50x
	CUDA	7007.0	24933.0	31940.0	0.56x
Modified	Lebrun (4x CPU)	1647.4	1906.2	3553.6	0.79x
	Original	1888.9	918.9	2807.8	1.00x
	OpenCL	366.1	542.8	908.9	3.09x
	CUDA	418.9	570.3	989.2	2.84x
	Mobile CUDA	852.1	989.4	1841.5	1.52x

5.6. Overall performance

The original filter profile turned out to be problematic for the GPU implementations. This was mostly caused by having more workload and private memory requirements per thread. The larger search window N_S adds latency for each thread, and greater N_2 values cause more private memory to be used by a thread. The memory usage is spilled to slower global memory and then threads slow down due to memory latencies. Greater register usage also implies also that less threads can be run in parallel efficiently.

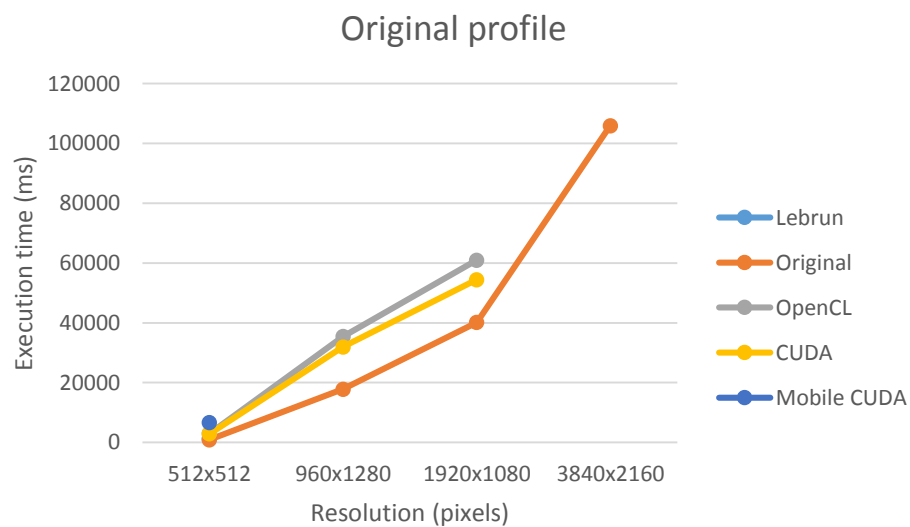


Figure 19. The execution times using the original filter profile.

The results that are missing from the table denotes a failure in the execution of the application in question. The failures in the implementation by Lebrun occurred because it uses integral images [36] to speed up the patch distance evaluation, and this uses memory excessively. Therefore it was not possible to run most of the larger resolution image tests at all due to running out of memory. In GPU applications the failures took place because of timeouts invoked by the GPU drivers in the test platforms. In other words, one kernel in the application took too much time to compute the required result. This could have been avoided by dividing the single invocation of a filtering kernel into multiple kernel invocations, e.g. by using the offset parameter in the `clEnqueueNDRangeKernel` call within the OpenCL implementation. Later this was tested and the real execution time was measured to be in the same magnitude or slightly slower than the original implementation.

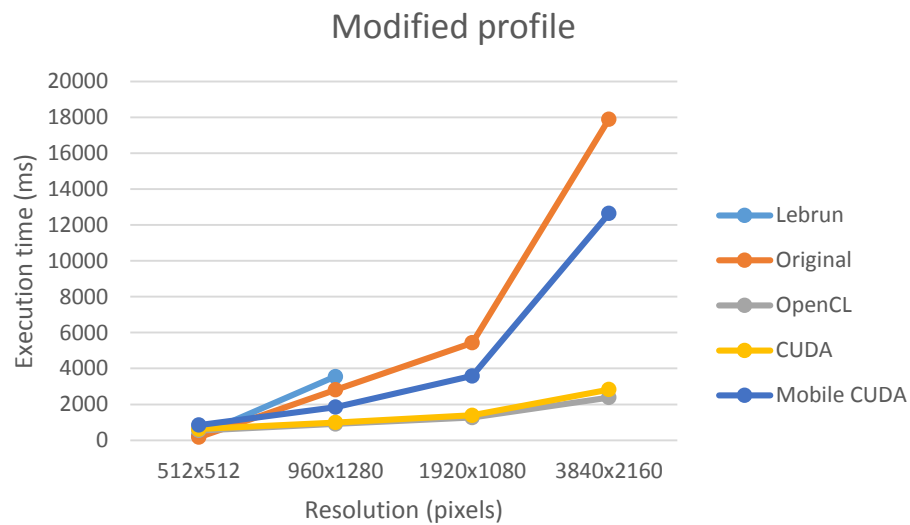


Figure 20. The execution times using the modified filter profile.

5.7. Comparison to reference implementations

A comparison collage of filtering the first test image with all implementations is shown in Figure 21. The collage shows that the images filtered with the original and OpenCL implementations are nearly equal. The proposed implementations used unnormalized for accessing the image data. Both of the GPGPU frameworks did not support texture mirroring in the border areas of the images when the unnormalized access method was used. Therefore the edge areas in the proposed results had some distortion compared to the reference implementations. Complex edge handling logic was left out in the GPGPU kernel codes with the cost of slightly worse PSNR values.

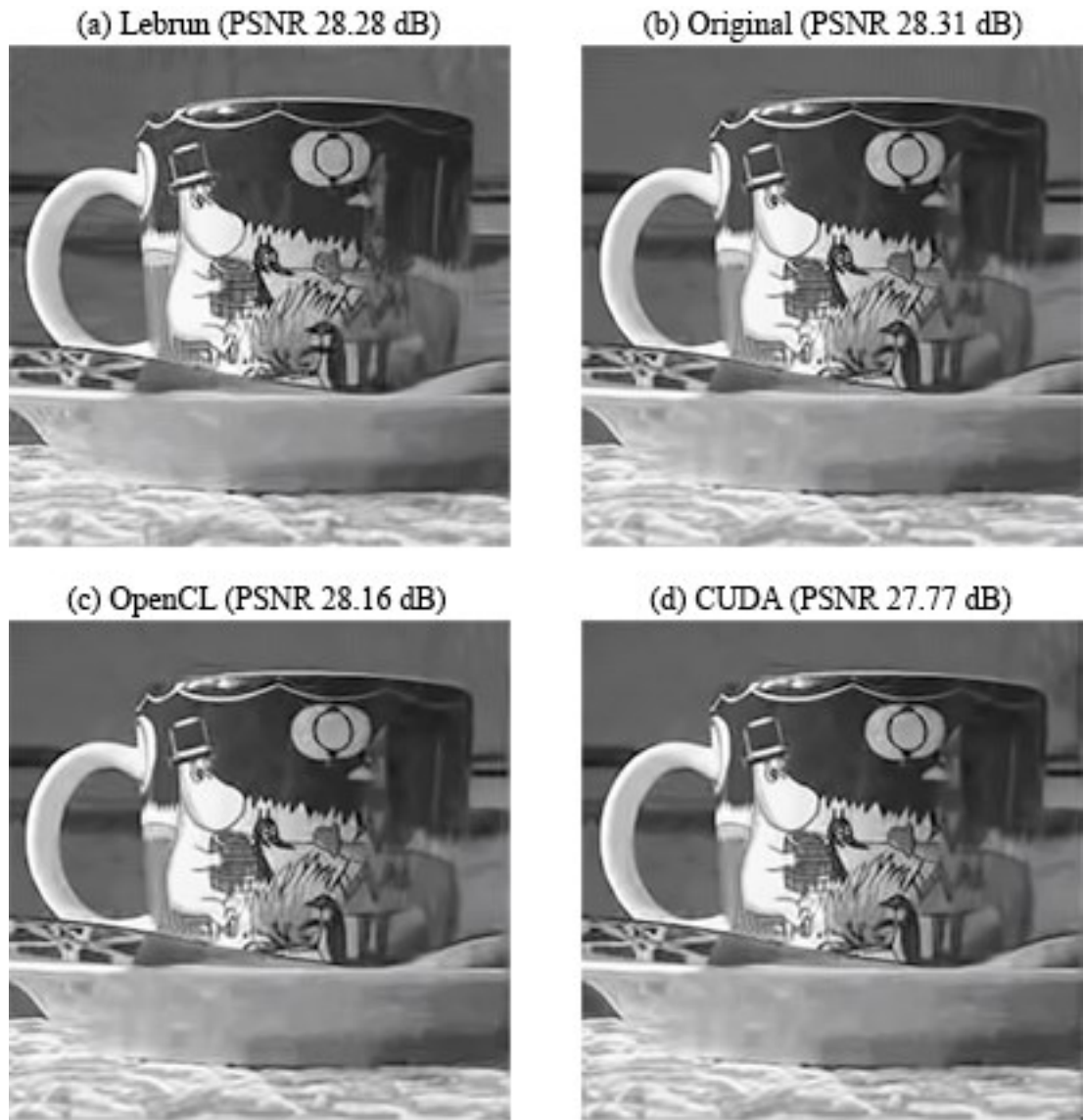


Figure 21. Filter implementation quality comparison using the original filter profile.

6. DISCUSSION

The first phenomenon to notice from the results is that in the case of small images, the CPU computation is faster in general because of the overhead of parallel computing and the benefits of a modern processor such as out-of-order processing. However with bigger resolutions the overhead is compensated and the parallel processing starts to add improvements to the performance with the modified profile. But with the original profile the workload tends to be too data-intensive for a single thread as it was discussed earlier in Section 5.6.

In general, the visual quality was slightly degraded by the modifications in the filter profile. This was a compromise that had to be done to reduce the workload a single thread had to compute. Alternatively a different approach for parallelizing the algorithm as described in Section 4.2.1 could have been used, but it would have produced other practical problems such as serialization of memory accesses and memory requirement issues with intermediate result saving. But subjectively in some cases where the noise was relatively small the image quality was still nearly the same with the modified filter profile, e.g. in the fourth test case with natural noise.

Additionally, there were several occasions where the basic estimate had equal or greater quality in terms of PSNR than the final Wiener filtered image. For practical applications the need for the Wiener filtering step can therefore be questioned, since it adds a lot of computation to the algorithm. While the Wiener filtering step produced smoother and better images in most of the synthetic noise test cases, there were occasions where some of the visual artifacts on solid surfaces were amplified by the Wiener filter as shown in Figure 22. In these cases the basic filter estimate was more pleasing for a human eye. As the original profile suggests, using a bigger N_2 value in the second filter step reduces this effect. Since the N_2 values were the same in both filtering steps in the modified filter profile, the second filtering step produced visually worse results at times.

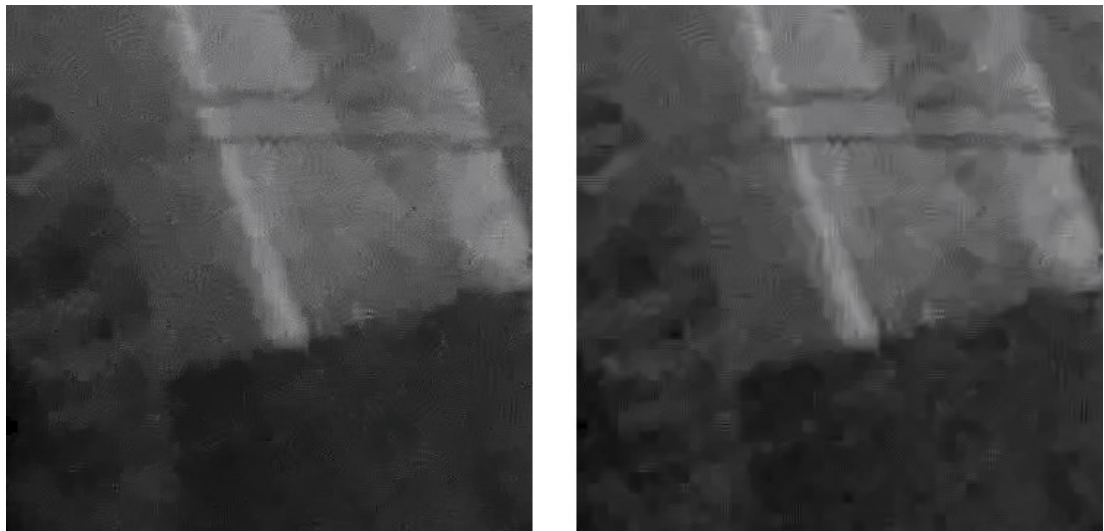


Figure 22. Close-up of the FHD image: basic estimate (left) and Wiener estimate (right). The Wiener filter adds blockiness on solid surfaces and makes the DCT basis artifacts more visible, but reduces small noise.

Especially in the 4K images in Figure 17 one can see a significant loss of contrast in the filtered image when compared to the original image. This happens because the added noise adds also energy to the image and a DC offset is added consequently. Both the objective and subjective visual quality could most likely be improved by altering the contrast or the dynamic range in the final image, for example by using histogram equalization or some corresponding method.

It should be noted that color processing was not included in the implementations. Adding two more color channels increases the computational requirements at least by a factor of two. Therefore real-time image processing with high-quality BM3D denoising is still not feasible with the processing units currently in use, especially on mobile platforms.

6.1. Future development

Since the kernel code ended up having significant amount of instructions per thread, dynamic parallelism might have been one good option worth trying to profile and optimize the various subroutines in the code. However the hardware used in the tests did not support such recursive programming model, where kernel launches can be called inside other kernels. NVIDIA introduced dynamic parallelism in CUDA version 5.0 for GPUs having compute capability version 3.5 or higher. The same concept was supported in OpenCL at version 2.0.

Mostly 32-bit floating-point units were used in the 3D transformation code. By having support for 16-bit half-precision floating-point units on the hardware and frameworks, the memory requirements could have been halved and to have greater occupancy. The loss of precision would most likely have only a minor effect on the final image quality, since the dynamic range in pixel values is not very wide. Also using fixed-point number formats might have resulted in very different performance measurements.

The presented implementations did not use the concept of unified memory. In unified memory the memory areas on the host and the devices are merged together into an area that is allocated and accessible on both systems. By using unified memory some parts of the code could have been simplified and possibly accelerated.

The implementation by Lebrun uses integral images for measuring the patch distances. In the integral images the pixel differences for each search window offset are calculated beforehand into buffers, and then the patch difference can be calculated with only four addition operations [36]. But because the integral images stored into buffers require much memory and also the buffer values are cumulating and possibly overflowing, using integral images is only feasible for small images where it could improve the performance.

The BM3D algorithm requires scattering of computed results and normalizing them with a weight map. By removing the scattering and writing only one result block in a thread would be much more suitable for GPGPU computing, because it would also eliminate the serialization of data write accesses.

7. CONCLUSION

The aim of this thesis was to study the field of image denoising and then implement a state-of-the-art solution to use parallel processing in heterogeneous computing platforms. In the study the basic theories of some of the well-known denoising algorithms were described. BM3D algorithm implementations for both desktop and mobile usage were created by using OpenCL and CUDA programming interfaces. The presented implementations were evaluated with several test images having natural or synthetic noise.

At the time of writing only two implementations of BM3D algorithm were publicly available, the original design by Dabov et al. [1] and an implementation created using C++ by Lebrun [34]. The source code for the original design was not available, and therefore the presented implementations could not be fully compared. The implementation by Lebrun uses OpenMP to parallelize the algorithm with CPU threads. The presented implementations use a similar parallelization strategy than the implementation by Lebrun but with GPGPU devices. However the parallelization of the BM3D algorithm efficiently using GPGPU turned out to be challenging, and multiple designs were considered. Main concerns were the serialization of data accesses and the memory usage as a whole.

The results clearly showed that there is potential in using recent GPGPU technologies in image denoising algorithms. Albeit the fact that the presented implementations did not provide real improvements in performance when high quality filtering parameters were used, there were notable speed-ups when using lower quality parameters. The study showed that it is possible to outperform a desktop CPU with a mobile GPU in performance of BM3D algorithm computation. For example, the filtering of FullHD images was 1.5 times faster on a mobile GPU when a lower quality filter profile was used. Also filtering 4K images in lower quality with a desktop GPU using OpenCL was 7.5 times faster than filtering with a desktop CPU.

Despite the manufacturer efforts on development GPGPU tools they still lack some key features and stability. For instance, several fatal crashes occurred while debugging or even analyzing the OpenCL kernel code with the CodeXL development tools. To the author's knowledge a traditional profiling tool, where each function call inside the kernel code is counted and the time spent is measured, was not available on any of the development platforms used. Such tool would make it significantly easier to detect the bottlenecks in algorithms and to optimize the code.

The current state-of-the-art denoising algorithm BM3D has hold its place well to this day in terms of image quality. But new algorithms are still being researched and developed and some implementations have already been shown to surpass BM3D in PSNR measurements in some cases [25, 37]. It is also likely that some of these new algorithms may be more suitable for GPGPU computing in terms of memory usage and parallelization.

8. REFERENCES

- [1] Dabov K., Foi A., Katkovnik V. & Egiazarian K. (2007) Image Denoising by Sparse 3-D Transform-Domain Collaborative Filtering. *IEEE Transactions on Image Processing*, vol. 16, no. 8, pp. 2080-2095.
- [2] Dabov K. (2010) Image and video restoration with nonlocal transform-domain filtering. Publication 909. Tampere University of Technology.
- [3] Owens J., Houston M., Luebke D., Green S., Stone J. & Phillips J. (2008) GPU Computing. *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879-899.
- [4] Carlos C. (2002) The Gap between Processor and Memory Speeds. *Proc. Internal Conference on Computer Architecture (ICCA)*.
- [5] Kirk D. B. & Hwu W.-m. W. (2010) *Programming Massively Parallel Processors*.
- [6] Farber R. (2011) *CUDA Application Design and Development*.
- [7] NVIDIA (2009) *NVIDIA OpenCL Best Practices Guide*.
- [8] Woolley C. (2013) *GPU Optimization Fundamentals*. NVIDIA.
- [9] Volkov V. (2010) Better performance at lower occupancy. URL: <http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf>.
- [10] Khronos Group — The Khronos Group Inc. URL: <https://www.khronos.org>.
- [11] Khronos OpenCL Working Group (2014) *The OpenCL Specification*.
- [12] NVIDIA — *Parallel Programming and Computing Platform*. URL: http://www.nvidia.com/object/cuda_home_new.html.
- [13] Gonzalez R. C. & Woods R. E. (2002) *Digital Image Processing (2nd Edition)*.
- [14] Rudin L. I., Osher S. & Fatemi E. (1992) Nonlinear Total Variation Based Noise Removal Algorithms. *Phys. D*, vol. 60, no. 1-4, pp. 259-268.
- [15] Chambolle A. (2004) An Algorithm for Total Variation Minimization and Applications. *Journal of Mathematical Imaging and Vision*, vol. 20, no. 1-2, pp. 89-97.
- [16] Buades A., Coll B. & Morel J.-M. (2005) A non-local algorithm for image denoising. *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition*.

- [17] Buades A., Coll B. & Morel J.-M. (2006) The staircasing effect in neighborhood filters and its solution. *IEEE Transactions on Image Processing*, vol. 15, no. 6, pp. 1499-1505.
- [18] Haar A. (1910) Zur Theorie der orthogonalen Funktionensysteme. *Mathematische Annalen*, vol. 71, no. 1, pp. 38-53.
- [19] Fino B.J. & Algazi V. (1976) Unified Matrix Treatment of the Fast Walsh-Hadamard Transform. *IEEE Transactions on Computers*, vol. C-25, no. 11, pp. 1142-1146.
- [20] Tran T. D. (2000) The BinDCT: Fast multiplierless approximation of the DCT. *IEEE Signal Processing Letters*, vol. 7, no. 6, pp. 141-144.
- [21] Arai Y., Agui T. & Nakajima M. (1988) A fast DCT-SQ scheme for images. *IEICE Transactions*.
- [22] Loeffler C., Ligtenberg A. & Moschytz G.S. (1989) Practical fast 1-D DCT algorithms with 11 multiplications. *Proc. International Conference on Acoustics, Speech, and Signal Processing*.
- [23] Gonzalez R. C. (2004) *Digital Image Processing Using MATLAB*.
- [24] Portilla J. & Simoncelli E. (2003) Image restoration using Gaussian scale mixtures in the wavelet domain. *Proc. International Conference on Image Processing*.
- [25] Rajaei B. (2014) An Analysis and Improvement of the BLS-GSM Denoising Method. *Image Processing On Line*, vol. 4, pp. 44-70.
- [26] Andrews D. F. & Mallows C. L. (1974) Scale Mixtures of Normal Distributions.
- [27] Elad M. & Aharon M. (2006) Image Denoising Via Sparse and Redundant Representations Over Learned Dictionaries. *IEEE Transactions on Image Processing*, vol. 15, no. 12, pp. 3736-3745.
- [28] MacQueen J. (1967) Some methods for classification and analysis of multivariate observations. *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, pp. 281-297.
- [29] Pati Y., Rezaifar R. & Krishnaprasad P. (1993) Orthogonal matching pursuit: recursive function approximation with applications to wavelet decomposition. *Proc. Conference Record of The Twenty-Seventh Asilomar Conference on Signals, Systems and Computers*.

- [30] Palma G., Comerci M., Alfano B., Cuomo S., De Michele P., Piccialli F. & Borrelli P. (2013) 3D Non-Local Means denoising via multi-GPU. Proc. Federated Conference on Computer Science and Information Systems (FedCSIS).
- [31] Goossens B., Luong H., Aelterman J., Pižurica A. & Philips W. (2010) A GPU-Accelerated Real-Time NLMeans Algorithm for Denoising Color Video Sequences. *Advanced Concepts for Intelligent Vision Systems*.
- [32] Su Y. & Xu Z. (2010) Parallel Implementation of Wavelet-based Image Denoising on Programmable PC-grade Graphics Hardware. *Signal Processing*, vol. 90, no. 8, pp. 2396-2411.
- [33] Sushchyk K. (2012) GPU-accelerated adaptive image denoising. Proc. International Conference on Modern Problems of Radio Engineering Telecommunications and Computer Science (TCSET).
- [34] Lebrun M. (2012) An Analysis and Implementation of the BM3D Image Denoising Method. *Image Processing On Line*.
- [35] OpenMP ARB — OpenMP. URL: <http://openmp.org>.
- [36] Facciolo G., Limare N. & Meinhardt-Llopis E. (2014) Integral Images for Block Matching. *Image Processing On Line*, vol. 4, pp. 344-369.
- [37] Lebrun M., Buades A. & Morel J.-M. (2013) Implementation of the "Non-Local Bayes" (NL-Bayes) Image Denoising Algorithm. *Image Processing On Line*, vol. 3, pp. 1-42.