BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

ACCELERATION OF 2D WAVELET TRANSFORM ON PARALLEL ARCHITECTURES

AKCELERACE 2D VLNKOVÉ TRANSFORMACE NA PARALELNÍCH ARCHITEKTURÁCH

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR — **Ing. MICHAL KULA**
AUTOR PRÁCE

SUPERVISOR — **prof. Dr. Ing. PAVEL ZEMČÍK**
ŠKOLITEL

BRNO 2021

Abstract

Although a 2D discrete wavelet transform has been widely studied during the last two decades, some directions were not examined from all points of view. One of these directions is a technique for calculating such transform that has various balanced barriers, arithmetic operations, and memory usage focused on various architectures. This thesis shows several new methods of calculation of such transform with variously balanced operations. These methods are widely described and their behaviour is evaluated on several graphics adapters using GPGPU, graphics pipeline, and multicore CPU architectures using OpenMP.

Abstrakt

I přesto, že byla 2D diskrétní vlnková transformace předmětem řady rozsáhlých studií, některé aspekty této problematiky byly doposud opomíjeny. Mezi takové aspekty lze zařadit techniky pro výpočet této transformace se zaměřením na vyvažování synchronizací, aritmetických instrukcí a využití paměti pro různé architektury. Tato práce ukazuje několik nových metod výpočtu této transformace s různě nastaveným vybalancováním těchto operací. Tyto metody jsou detailně popsány a jejich chování je vyhodnoceno na několika grafických adaptérech za použití GPGPU, zpracování pomocí grafické pipeliny a vícejádrových procesorů pomocí OpenMP.

Keywords

Discrete wavelet transform, seamless transform, 2D scheme, GPU, GPGPU, CPU, parallelization, optimization

Klíčová slova

Diskrétní vlnková transformace, bezešvá transformace, 2D schémata, GPU, GPGPU, CPU, paralelizace, optimalizace

Reference

KULA, Michal. *Acceleration of 2D Wavelet transform on parallel architectures.* Brno, 2021. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Dr. Ing. Pavel Zemčík

Acceleration of 2D Wavelet transform on parallel architectures

Declaration

I hereby declare that this thesis was prepared as an original work by the author under the supervision of prof. Dr. Ing. Pavel Zemčík. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Michal Kula
October 24, 2021

Acknowledgements

Firstly, I would like to thank my supervisor Pavel Zemčík for valuable advices during my studies, my collegues David Bařina, Michal Matýšek, Pavel Najman, Petr Klepárník, and Michal Kučiš for cooperation on papers, other colleagues, namely Petr Musil, Martin Musil, Tomáš Milet, Josef Kobrtek, Tomáš Chlubna and Tomáš Starka for valuable criticism of my ideas and their good company and my girlfriend Marcela for being support for me even in tough times.

Contents

Chapter 1

Introduction

Over the last two decades, complex performance demanding algorithms in computer graphics and multimedia areas became mainstream. Whether it is photorealistic rendering, image filtering, or compression algorithms, many of these algorithms require real-time or offline massive data processing. Additionally, many of them are built on top of a 2D DWT (two-dimensional discrete wavelet transform). Consequently, these algorithms can potentially benefit from an acceleration of an underlying discrete wavelet transform. The acceleration of such a transform is the main domain of this thesis.

The current trend of technologies and architectures shows several ways to accelerate such algorithms.

The first of them is the acceleration of sequential algorithms by algorithmic changes. One direction for these changes is optimal balancing between arithmetic operations and memory usage. However, for highly arithmetic demanding algorithms by their nature, this way appears to be insufficient.

The evolution of processors over the last decades showed that increasing the frequency and instruction throughput of processors is not power efficient, and as a solution, multi-core processors became mainstream. These processors opened a new way for accelerating algorithms using threads and processes. As a result, effective utilization of such processors brings new challenges that include synchronizations of threads, communication between threads, data hazards and many others.

Another direction in parallel processing starts to be GPUs. The GPUs became increasingly powerful in terms of memory and arithmetics throughput during the last two decades. Initially, the limiting factor for their general purpose usage was the requirement to use a graphics pipeline processing paradigm that uses specialized vertex and fragment processors with many restrictions. Several authors tried to accelerate general-purpose algorithms using these GPUs, and several of them succeeded.

The real breakthrough in the GPU processing area started with the invention of unified architecture. That unveiled the new path for accelerating algorithms using GPUs. In this architecture, the main change was a unification of computation units (vertex/fragment cores) used to calculate programable parts of the pipeline and corresponding scheduler changes. Many architectures based on unified architectures appeared during last decades. Even the most low-power processors used by modern smartphones and tablets integrate GPUs based on unified architecture (e.g. ARM Mali, Qualcomm Adreno). However, the exceptions still exist on the market and they should not be omitted.

Availability, high performance, and relatively low price of GPUs make them one of the best options for accelerating the time-consuming algorithms with CPUs retained free for

other tasks. Moreover, the GPU is integrated into many processors sold these days. This integration can be beneficial, especially when CPU and GPU processings are interleaved due to using the same memory so copying the data through a relatively slow PCI-E bus is not necessary unlike in the dedicated GPUs. Over the last few years, the importance of these GPUs in the GPGPU field has grown, and all indicate that it will also rise over the time.

The scope of this thesis includes all three acceleration directions: multicore CPUs, nonunified GPUs and modern GPUs with unified architectures. The main focus of interest is aimed at GPUs with unified architectures.

The thesis consists of the following chapters. The first of them, GPU Processing, describes GPU architectures and challenges related to its efficient usage. The next chapter, Discrete Wavelet Transform, is focused on DWT basics and state-of-the-art methods for its calculation. The formulations of the thesis goals and a list of published papers are parts of the chapter Goals of the Thesis. The framework used for evaluating 2D DWT calculation methods is presented in the chapter Wavelet Transform Framework. The published papers with proposed methods for acceleration of 2D DWT on various platforms are presented in the following chapters: Article: 2-D Discrete Wavelet Transform Using GPU, Article: Block-based Approach to 2-D Wavelet Transform on GPUs, Article: Parallel Wavelet Schemes for Images, Article: Accelerating Discrete Wavelet Transforms on Parallel Architectures and Article: The Parallel Algorithm for the 2-D Discrete Wavelet Transform. The chapter Summary, Applications and Future Work presents proof of the hypothesis, schemes overview, possible application and future research topics. The final chapter, Conclusion, contains the formulation of the conclusion of the thesis.

Chapter 2

GPU Processing

This chapter is focused on explanation of common concepts used in modern GPUs (the section 2.2), functions of their components (the section 2.3), descriptions of various types of memories and their applications (the section 2.4), disclose synchronization techniques (the section 2.5), and reveals various performance considerations (the section 2.6).

2.1 Recent GPUs

At the time of writing this thesis, the most recent GPUs consist of up to 10752 unified processors[1], and their memory can transfer up to 1.2 TB/s[2]. Such GPUs provide the power of 39TFlops using single precision data type, or even more on less precision data types using special execution units called tensor cores. Additionally, these GPUs are capable of processing more than 100 thousand threads on the fly.

The utilization of such powerful devices brings several challenges that are described later in this chapter.

2.2 Execution Model

Programs capable of executing on GPU are called kernels (GPGPU frameworks notation) or shaders (Graphics pipeline frameworks notation). It is desirable to execute kernel instances containing lots of threads for sufficient utilization of commonly used GPUs with a high amount of cores. For intuitive mapping of those threads to algorithms, the kernel instance can issue a block of threads with up to 3-dimensional shape, so-called thread grid. Threads in this grid are combined into equally-sized blocks of threads, so-called work-groups. The size of these work-groups is user-defined, and their maximum size is limited by platform resources (more information in the section 2.6.5). Grid size has to be aligned to multiple work-group size in each of its dimensions (more information in the section 2.6.1).

These work-groups are divided into warps that commonly correspond to HW-locked blocks of threads that are issued by multiprocessors schedulers[3]. When the work-group size is not aligned to multiply of warp size, the rest of the threads from the last subgroup is

[1] specification of RTX A6000 card, only half of processors are capable of processing integer data types [11]

[2] specification of AMD Instinct MI100 [8], accelerators without rendering support has up to 2TB/s (Nvidia A100 80GB SMX) [10]

[3] HW-locked block of threads is denoted as warp on Nvidia GPUs and as wavefront on AMD GPUs

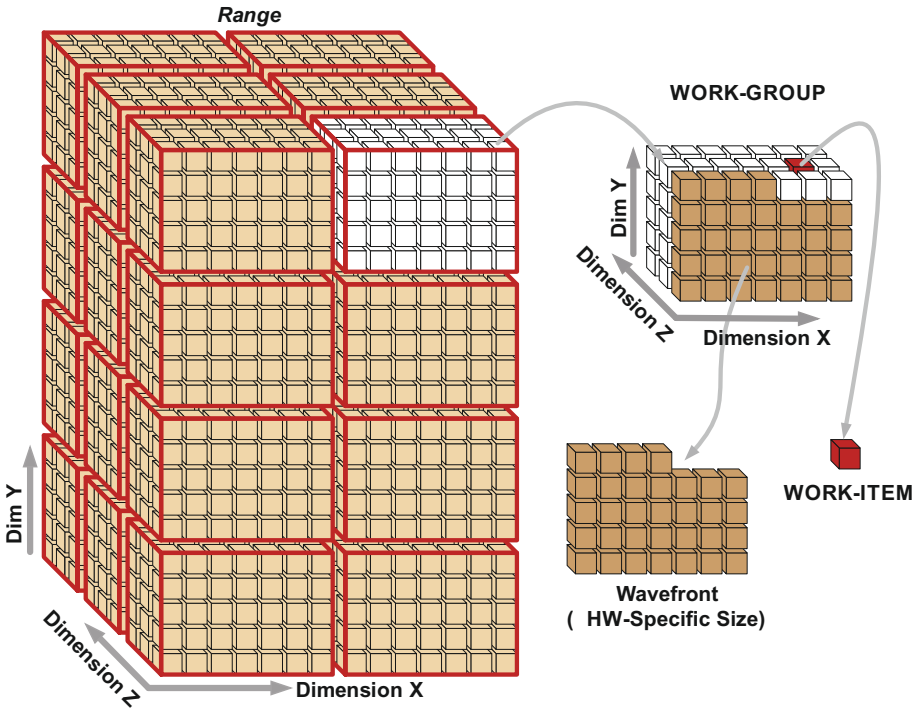


Figure 2.1: Partitioning of thread grid. Thread is denoted as WORK-ITEM. Warp is denoted as Wavefront. [2]

masked-out from processing. It leads to thread divergence that is described in the section 2.6.2. Partitioning of the grid is visualized in Figure 2.1.

2.3 GPU Components

An example of GPU structure is visualized in Figure 2.2. The typical unified GPU architecture consists of the main scheduler, multiprocessors, global memory controllers and L2 cache. Multiprocessors can be further subdivided into warp schedulers, dispatch units, execution blocks, local memory, L1 cache, constant cache and possibly other caches (such as Nvidia Kepler read-only cache [1]).

Global memory is an off-chip memory with high latency and relatively slow throughput. Dedicated global memory is commonly placed on the GPU board in GDDR5/6/6X form or on the silicon imposter in the HBM memory form. The section 2.4 provides more information on global memory.

L2 cache is used for caching data origins in global memory (global memory, textures, data for specialized read-only caches).

Main scheduler [4] distributes work-groups from kernel instances among multiprocessors as long as they have enough resources (registers, local memory, etc.). The rest of the work-groups wait in a queue until sufficient resources of some multiprocessors are

[4]Nvidia denotes Main scheduler as GigaThread scheduler, AMD as Command processor

Figure 2.2: Block diagram of Maxwell GM204 GPU platform which can be found in Nvidia GeForce GTX 980 GPU. The GPU consists of: Main scheduler denoted as GigaThread Engine, four Memory Controllers located on the sides of block diagram each connected to a GDDR5 global memory chip by 64bit bus, 2MB of L2 cache for caching global memory load/store operations, and 16 Multiprocessors denoted as SMM (Streaming Maxwell Multiprocessor) grouped into 4 GPC (Graphics Processing Clusters) [3].

available. The resources of each work-group are allocated to the multiprocessor unless all threads within the work-group finish their executions.

Multiprocessors (Figure 2.3), in the next step, decompose these work-groups into fixed-size bunch of threads, so-called warps (or wavefront on AMD). The size of these warps depends on either a platform or on a combination of platform and kernel properties (Intel GPUs [5]). These warps are allocated statically into warp schedulers, that are reponsible for issuing all of their instructions.

Warp schedulers choose warp from their pool of statically assigned warps that have prepared operands of the next instruction and pass it to dispatch unit. Note that the ratio between the number of warp schedulers and dispatch units is not always $1:1$. On platforms with ratio $N:1$ where $N > 1$, the utilization of all dispatch units is possible only when ILP[5] is used. This operation can be performed by creating a block of N independent instructions (see the section 2.6.3 for more information).

Dispatch units execute instructions on execution blocks that are owned by/attributed to/belong to underlying warp schedulers. Some of these blocks are shared with other warp schedulers across the multiprocessor (e.g.texture units blocks on almost all architectures, all blocks on Nvidia Kepler architecture [1] etc.).

Execution blocks can be categorized into three common classes: arithmetics blocks, load/store blocks, and texture units blocks.

Arithmetics blocks are either blocks of Core (streaming processor) ALUs that are suitable for calculating standard arithmetic instructions on 4B operands (except for transcendental instructions), blocks of SFU5 ALUs for calculating transcendental instructions (sin cos, etc.), blocks of DP blocks suitable for execution with double precision instructions (typically not presented on consumer GPUs) or other platform-specific arithmetics blocks (like Ray-Tracer cores, Tensor cores, Scalar units, VLIW[6] units, etc.). The availability of arithmetics blocks can vary between platforms. For instance, AMD GPUs, since GCN architecture, have additional scalar unit used for calculation instructions with similar operands across the warp (like calculation of indices, operations on constants, branch masks operations, etc.). VLIW architectures have a block of VLIW processors containing 4 or 5 ALUs (e.g. AMD VLIW 4, AMD VLIW 5 or ARM Mali Midgard). On these architectures, the independent common instructions are packed into VLIW instructions in a maximum ratio of $N:1$ where N is the number of ALUs in VLIW cores. The section 2.6.4 provides more information on instruction dependency and its impact on performance.

Load/store units block are responsible for loading and storing data to global or local memory. Properties of these memories are discussed later in the section 2.4.

Texture units blocks are responsible for loading data from global memory represented as nD textures. The section 2.4 gives detailed information on texture memory.

[5] ILP - Instruction level parallelism
[6] VLIW - Very long instruction word

Figure 2.3: Block diagram of Maxwell GM204 GPU platform multiprocessor denoted as SMM (Streaming Maxwell Multiprocessor). The multiprocessor consists of: PolyMorph engine for scheduling threads during graphics pipeline stages, 4 independent Compute blocks, 96kB of local memory denoted as Shared Memory, 24kB of combined Texture/L1 cache, and 8 texture units denoted as Tex. Each Compute block can be further divided into: 16k of 4B registers, 1 Warp Scheduler connected to 2 Dispatch Units capable of executing 2 independent instructions from the same warp on 2 blocks of units, block of 32 Core units, block of 8 LD/ST (Load/Store) units, and block of 8 SFU (Special Function Units) [3].

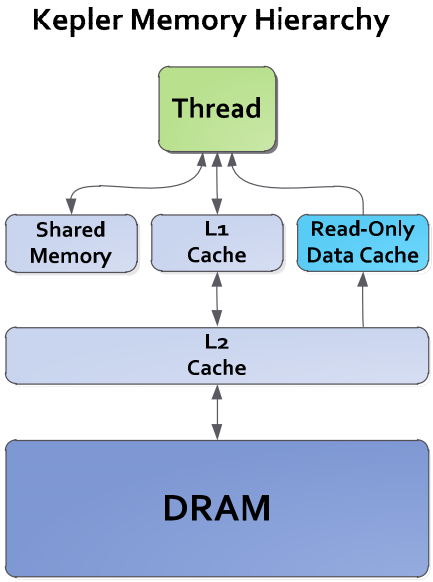


Figure 2.4: Memory hierarchy on Nvidia Kepler architecture. [1]

Local memory is programmable memory allocated for a particular work-group and existing only within the lifetime of the work-group. Its typical size per multiprocessor is 64-128kB on modern architectures. The section 2.4 provides detailed information on local memory.

Registers are the fastest memory available in GPU. They are allocated per thread. The most modern architectures have 56-64k of 4B registers. More information on registers can be found in the section 2.4.

L1 cache differs in use according to the platform. Commonly, it is used for caching textures data that originates in global memory. On some architectures, the global memory is cached as well. On some other architectures, the behavior can be set by load instruction qualifier (like `.ca` qualifier of `ld` instruction on Nvidia architectures [12]) or by kernel configuration.

2.4 Memory Model

Memory can be categorized into two classes: on-chip memory and off-chip memory. Almost all memories used in kernels, such as global memory, constant memory and texture memory, belong to the off-chip memory. The difference between these memories lies in caching behaviour. In contrast to the off-chip memory, the on-chip memories lie directly on the GPU chip. The common memories of the on-chip type are local memories that exist only in the scope of work-group lifetime and registers. The caches like L1, L2, constant cache, instruction cache, and other caches on some platforms, such as read-only cache on Kepler platform [1], eDRAM cache on Intel Iris platform [4], Infinity cache on AMD RDNA2 platform, etc., are also classified as the on-chip memories. The memory hierarchy is visualized in Figure 2.4. Memory properties are denoted in Table 2.1.

type	location	cache	size	size per	allocation	Host available	GPU readable	GPU writeable
global	off-chip	L2**	1-80GB	GPU	CPU	yes	yes	yes
texture	off-chip	L2/L1	1-80GB	GPU	CPU	yes	yes	yes
constant	off-chip	Constant	64kB	buffer	CPU	yes	yes	no
local	on-chip*	None	64-128kB	multiproc.	group	no	group	group
register	on-chip	None	224-256kB	multiproc.	thread	no	thread	thread

Table 2.1: Memory types. *On some plaforms emulated by global memory (ARM Mali architectures). **On some platform cached in L1 as well.

2.4.1 Texture Memory

Textures provide the ability to:

- read out-of-range data with several mods (mirror-repeat, repeat, clamp, etc.),
- filter the data using various filtering methods (bilinear, nearest-neighbour, trilinear),
- enable normalized coordinates (stretch an image to range 0-1 in both axes regardless of its original size).

Reading of such textures is typically cached to multiprocessors' L1 cache with the optimized spatial caching pattern. The texture units located in multiprocessors are used for reading these textures. The main advantage of these textures is that the previously described operations are processed without the assistance of arithmetic Cores that retain these Cores free for other tasks. The disadvantage of texture usage lies in limited throughput in comparison with L1 cache/local memory.

In 2D DWT or similar algorithms, the textures can be used for branchless reading of out-of-range data with mirror-repeat out-of-range mode.

2.4.2 Global Memory

Global memory is common memory lying outside of chip, fillable and readable from host and kernels. In most cases, the memory is located in dedicated GPU memory (DDRx or HBM memory) or system RAM when the GPU is integrated. However, when the GPU memory is full, the system can allocate part of global memory to the system RAM, causing significant degradation of performance.

Even the global memory is cached to L2 cache during loading of data on modern architectures, it is desirable to reduce the usage of this memory to avoid overloading of the cache or in worse case reloading the data from global memory when the size of L2 cache is not sufficient for on-the-fly data.

When a global memory load instruction is executed by warp, data needed by threads is loaded firstly from global memory to L2 cache in the form of cache lines (when it is not lying in L2 already). Afterwards, the cache lines are loaded to multiprocessors registers (these cache lines are also loaded to L1 cache on some architectures). The size of cache lines is platform-dependent, and their typical size is 32-128B. This fact implies that reading from various locations by warp threads instruction (uncoalesced access) leads to loading multiple cache lines from global memory. As a consequence, the latency of such load instruction rises and L2 cache are overused. The worst case of uncoalesced access is loading of small values from random locations. In contrast to random access is coalesced access where adjacent

threads in warps load adjacent locations in memory. Completely coalesced access is aligned to the size of the memory block that is loaded by the warp. Loading values from the same location to all or several threads in the warp is on current platforms without any penalization because current platforms support broadcast and multicast loading (in contrast to old Nvidia Tesla architecture where nonserialized access is conditioned by complete coalesced access). Supported data size loadable by single thread is one of the sizes of 1B, 2B, 4B, 8B or 16B assuming that data start is aligned to that data size. In other words, the maximum loadable size per warp is 512B when warp size is 32. Note that loadable/storable data size in global memory differs from local memory, which is limited to 4B. Loading/storing larger blocks from/to local memory leads to splitting memory operation to separate 4B operations with bank conflicts.

2.4.3 Constant Memory

Constant memory data is lying in global memory and global memory data but it is marked as read-only and it is cached often by special small constant caches that lie in multiprocessors. The maximum size of the constant memory buffer is often limited to 64kB, and the cache size in the multiprocessor is typically much smaller (8kB). Constant memory is primarily optimized for broadcasting values to the whole warp. As a result, load instruction issued by warp (where the location of threads is variable) possibly leads to serialization of reading and performance degradation (depending on the specific architecture implementation).

2.4.4 Local Memory

Local memory[7] is a user-controlled fast memory cache, typically located in multiprocessor (all AMD [6] and Nvidia GPUs [1, 3], Intel GPUs since 11th generation [5]). An exception is an architecture where local memory is emulated and the data lies physically in global memory (ARM Mali architectures) or lied in L2 cache that causes lower throughput (Intel architectures preceding 11th generation of Intel CPUs [4]). Local memory is allocated to a particular work-group and exists only within its lifetime. It is not prefillable from a host or any other work-group. Usually, the memory consists of banks where each bank can load data from one location within warp load/store instruction. The number of banks is platform-dependent but the common size is 32 or 16 banks. Those banks are interleaved across local memory by 4B segments (some architectures like Nvidia Kepler can be configured to 8B segments [1]). Load/store instruction that loads/stores data from N different locations lying in the same bank causes N-way bank conflict. N-way bank conflict is serialized to N separated loads/stores, leading to overuse of load/store units and lowering their effective bandwidth to 1/N. For this reason, bank conflicts should be avoided wherever it is possible. The issue with bank conflict can also happen when 8B or 16B data type values are loaded by coalesced access from global memory followed by storing them to local memory. For instance, when the 16B data type values are loaded by warp, its store is split into four separate 4B store instructions where each of them stores data with the stride of 4. Consequently, every 4th bank is used four times in one store instruction that causes a 4-way bank conflict. Note that bank conflict free access to local memory is not conditioned by coalesced access. Strided access by prime number higher than 2 (3, 5, 7, etc.) leads to bank conflict-free access. Additional information on local memory and coalescing can be found in Cuda C Programming Guide [7].

[7] Nvidia denotes local memory as shared memory

2.4.5 Registers

Registers are the fastest memory available in GPU. Common variables and arrays in kernels are interpreted as registers. While compiling kernel where too many registers are needed, the compiler can decide to spill some registers to global memory. As a consequence, it possibly degrades the throughput of the kernel. Information about register spilling can be obtained from the compilation log. On some architectures, registers can be exchanged between threads within the same warp. For more information on exchanging registers, see next section 2.5.

2.5 Synchronization

Three mechanisms exist for communication between threads in the kernel: memory synchronizations, atomic operations and in-warp communications.

Barriers ensure that all work-group threads load/store operations on desired memory preceding barrier are finished before starting any following memory operations of the same work-group threads. The impact of barriers on thread calculation is described in the section 2.6.

Atomic instructions are one-way communication instruments between threads in the whole kernel. It provides a way to load data from a particular place from global/local memory, calculate one of the predefined operations, and store the result back to memory without interfering with other threads. The supported operations for integer numbers are: bitwise operations, addition and subtraction operations, exchange value operation and compare value, and exchange operations. Modern platforms are capable of processing float add atomic operations as well, but the support is limited to languages that enable these operations within their extensions.

In-warp instructions can be used for data exchanges between threads in warp. They take advantage of the fact that the instruction for the whole warp is performed atomically[8]. The most common instructions from this class are shuffle instructions. The shuffle instructions can exchange register values between threads within the warp with: higher or lower thread IDs (`shuffle_up`, `shuffle_down`), threads IDs changed by mask (`shuffle_xor`), exact IDs of threads within the warp (`shuffle`). These instructions are platform-dependent because the warp size varies between platforms; therefore, they are not a core part of platform-independent parallel languages. Implementation of these instructions varies between platforms as well. Not all variants are operating on whole warps in cases of some platforms (like AMD data-parallel primitives operating either on blocks of 16 threads using `shuffle_up`, `shuffle_down` modifier or on a block of 4 threads using `shuffle` modifier). On the contrary, shuffles are not supported on some other platforms (Nvidia Fermi and older Nvidia platforms, ARM Mali platforms, AMD VLIW platforms). Shuffle instructions are optionally used in the Wavelet transform framework, described in the chapter 5. Note that the shuffle operation, presented in OpenCL library [13], shuffles the registers within the same thread in contrast to warp shuffle operations presented in this section.

[8] exception is Nvidia Volta and newer architectures where synchronization between threads in warp is not implicit and has to be forced.

2.6 Platform Targeting

Available GPU platforms have various types of caches, execution units, amount of resources and other properties that should be considered while targeting algorithms on a specified or wide range of platforms. One of those properties are instruction latencies. On GPUs, the processing of instructions is pipelined that results in such latencies. These latencies can be hidden either by processing independent instruction (see the section 2.6.4 for more information) or multithreading (see the section 2.6.5 for detailed information).

2.6.1 Grid Alignment

Grid size has to be aligned to multiple work-group size in each of its dimensions. Common strategies for meeting this rule are to align the grid size and exclude overflowing threads by a condition or to align the grid size and underlying data and execute all threads even if some of them are processed out of valid data range.

The first strategy encompasses the execution of conditional branch by all threads that implies splitting the code into separate code blocks (before the branch, after the branch and after the jump). Thus, the number of instructions that the compiler can shuffle to avoid instructions dependencies possibly declines (more information on this phenomenon is presented in the section 2.6.4).

The second strategy requires resizing input and/or output buffers or remapping threads pointing to out-of-range data to the valid data areas. For instance, the correct calculation of 2D DWT described in the chapter 3 requires a symmetric extension of borders. The symmetrically extended borders can be loaded by remapping load positions of out-of-range pointing threads to their symmetric counterparts. This technique is used in the implementation on the Wavelet transform framework described in the chapter 5.

2.6.2 Thread Divergence

Thread divergence is a state when all threads within a work-group are not going through the same execution path. It is caused by conditional jumps or the execution of work-groups not aligned to multiple warp sizes. Thread divergence can be categorized into in-warp divergence and divergence of whole warps.

In-warp divergence is caused by conditional jumps, where some of the threads within a warp follow another path than others. In that case, the execution of both paths is serialized, and inactive threads within the path are masked-out from processing. Moreover, warp instructions are processed in lock-step form, so instruction issuance time is not dependent on thread mask. Consequently, it leads to a reduction of GPU utilization. Thus, it is highly recommended to avoid in-warp divergence as much as possible.

In contrast to the in-warp thread divergence, thread divergence of whole warps does not necessarily lead to lower utilization of GPU, but the performance impact depends on a balance of various instruction types across warps in work-group, number of work-groups allocated to multiprocessor in parallel and warp scheduling strategy.

For instance, a platform with 32 threads-sized warps, four warp schedulers per multiprocessor (such as Maxwell architecture - see Figure 2.2) and strategy of assigning warps by modulo arithmetics is used for simple Gaussian filtering of size 3×3 by 32×8 threads-sized work-groups. Faster access to repeatedly loaded data is provided by caching the data to the local memory. Synchronization between work-groups during their calculations is either not

possible or it is costly[9] so that work-group threads can be mapped either onto input regions or onto output regions of an image. In this case, each work-group is mapped onto the input region of 32×8 pixels. Borders of Gaussian filter cause that output region produced by work-group will be 30×6. Firstly, the data from the input region is copied into local memory by all threads. The rest of the calculation is proceeded only by 30 from 32 threads in a row, so the rest two threads can be possibly masked out. Moreover, the first and the last warp representing the first and the last row can be masked out, which seems beneficial. However, in this particular algorithm and strategy of mapping warps to the schedulers, where the warp schedulers are mapped to the warps: 0:{0,4};1:{1,5},2:{2,6};3:{3,7} for all given work-groups, the warps 0 and 7 are inactive during the calculation that leads to an imbalanced number of instructions issued by warp schedulers 0,3 and 1,2. Consequently, that can lead to up to 25% performance loss regardless of masking out these warps (in a case where global memory throughput is not the limiting factor). Moreover, the branch condition used for masking the inactive threads causes splitting the code into separate blocks that can cause performance degradation (see the section 2.6.4 for more information). Implementation of 2D DWT in Wavelet transform framework (described in the chapter 5) deals with border effect using branch-less calculation without masking-out the threads except for final storing operations.

Barrier commands have to be issued by either none or all of the threads within work-group. Consequently, all threads within a work-group have to converge to the same path before they issue any barrier command.

2.6.3 Balancing Operations

The maximum ratio between various instruction types for preserving maximum arithmetics throughput is related to the number of execution units across different execution blocks. Such a number can be found in platform-specific datasheets or programming guides. A good source of this information focused on Nvidia platforms can be found in [7].

In Nvidia Maxwell platform multiprocessor (visualized in Figure 2.3), 4 warp schedulers are present, each of them with 32 Cores usable for common 32bit float/int instructions, 8 load/store units usable for global/local memory load/store instructions and 8 SFU units capable of transcendental instructions. Additionally, 2 dispatch units are present with a capability to issue 2 independent instructions from the same warp. Consequently, for full cores utilization, the ratio of transcendental instructions to the common instructions can be 1:4 at maximum. Besides, the ratio of load/store operations to the common instructions is the same. However, only 2 dispatch units per warp scheduler are available, so it is not possible to issue the common instruction, transcendental instruction and load/store instruction simultaneously but only two of them at a time. Moreover, the instructions of various types must be interleaved, and at least one instruction from each pair of independent instructions must be the common one.

But not all platforms have two dispatch units; therefore, not all of them can issue load/store or transcendental instruction in parallel with common instructions for free, even if the instructions are interleaved accordingly. For instance, in the Nvidia Fermi platform, the multiprocessor has 2 warp schedulers, 2 dispatch units, 2 blocks of 16 Cores, 1 block of 16 load/store units, and 4 SFU units. Using SFU or load/store units in that architecture automatically leads to stalling the block of cores.

[9] Cuda framework provides this ability for global synchronization but it is highly performance demanding

Another example is the newest Nvidia Ampere platform [11] (at the time of writing the thesis) with 4 warp schedulers each connected to 1 dispatch unit, that can dispatch instruction to the whole warp despite that the block of cores is half-warp sized. Thus, it behaves similarly to the Maxwell platform [3]. Additionally, 2 blocks of 16 Cores, 8 load/store units and 4 SFU units are available. In that architecture, every issue of load/store or transcendental instruction results in a stall of one of the cores blocks.

On most AMD GCN/RDNA/CDNA platforms, the Load/Store instructions can be issued for free with a ratio of 1:2 to common instructions without the need for any ILP. Additionally, there are scalar units capable of calculation scalar instructions (usable for calculation of thread masks, work-group indices, etc.). The scalar instructions can be issued free and simultaneously with load/store instructions in a ratio of 1:1 to common instructions.

In other words, the best-chosen ratio between various instructions types for one platform does not necessarily mean that the ratio is best for the rest of them.

2.6.4 Instruction Dependency

A compiler is responsible for shuffling instructions to reduce dependence between adjacent instructions and providing a balance of instructions for various execution blocks. These features are especially essential on platforms, where ILP is needed to utilize the whole multiprocessor or VLIW architectures that pack independent instructions into VLIW instructions. Changes in algorithms could lead to the extension or reduction of the shuffleable instruction pools. For instance, complete or partial unrolling of the loops, increasing the number of inputs processed by a thread, and similar techniques typically extend that pool. Consequently, these techniques extending the number of registers allocated to the threads that can possibly cause a reduction of the number of work-groups allocable to multiprocessors and, as a consequence, lower the occupancy (see the section 2.6.5 for more information). Unwilling reduction of shuffleable instruction pool is often caused by issuing of instructions that lock ordering of other instructions:

- Branching instruction splits shuffleable instructions pool into 3 instruction pools separated by jump condition and jump location.
- Barrier instruction forces to issue load/store instructions before barrier before load/-store instructions after barrier.
- Memory labeled as `volatile` forces to order all of its load/store operations and forces to load data even if it is loaded repeatedly.

Avoiding those instructions could lead to better utilization of the platform and, consequently, better algorithm performance. Especially beneficial is to remove the branches focused on mask-out threads that only lead to in-warp thread divergence (for more information, see the section 2.6.2).

2.6.5 Multithreading

The degree of multithreading can be expressed as a number of warps that are allocated to one warp-scheduler in a multiprocessor. The destination platform caps the maximum degree of multithreading, and the degree is commonly expressed in % of maximum degree supported by the platform as GPU occupancy in literature. The occupancy is dependent

on resources available in multiprocessor and resources needed for execution of kernel by work-group.

The resources that limit the occupancy with commonly available amounts of the resources on platform multiprocessors in braces are:

- number of work-groups (8-32),
- number of threads (1536-2560),
- size of local memory (64-128kB),
- number of registers (56-64k of 4B registers).

The resources are typically allocated for whole work-groups so lowering the work-group size can lead to a better granularity of resource allocation and higher occupancy (unless local memory consumption is not bound to the number of threads in a work-group). The degree of multithreading for preserving maximum arithmetics throughput varies by platform and by kernel properties. Specifically, it depends on the instruction latency, the ratio between execution units in execution blocks (e.g. block of Cores, SFUs, Load/Store units, etc.), warp size, and the number of independent instructions in the kernel. For instance, Maxwell architecture [3] has 32 threads-sized warp, 4 warp schedulers, each warp scheduler paired with the block of 32 Cores, maximally 2048 threads per multiprocessors and 6 cycles arithmetics latency for common instructions. The following parameters can be calculated based on the mentioned information: the number of warps per multiprocessor as $2048/32 = 64$, the maximum number of warps per warp scheduler as $64/4 = 16$ and latency in instructions as $6 \cdot 32/32 = 6$. Two ways of preserving the maximum arithmetic throughput can occur: either every instruction has to precede 5 instructions that are independent of it, or there are at least 6 warps allocated to each warp scheduler. Additionally, 6 warps per scheduler result in occupancy $6/16 = 37.5\%$. Note that the occupancy calculated this way does not always lead to 100% utilization of Cores. Other circumstances like global data load latency, an unbalanced number of various instruction types (see more in the section 2.6.3), thread divergence (see more in the section 2.6.2), and barriers (see more in the section 2.6.6) can lower the utilization of Cores as well. The occupancy of Nvidia GPUs can be calculated using Cuda Occupancy Calculator [9].

2.6.6 Barriers

Barriers ensure that all load/store operations within the work-group that precede the barrier are finished before any following memory operations are started. When warps from a work-group reach the barrier command, they are inactivated until the rest of the warps within the same work-group reach the barrier command as well. Consequently, a reduction of active warps in warp-schedulers occurs that possibly lowers the ability to overcome instruction latencies by multithreading. In addition, the barriers lead to splitting the execution of the work-group instructions into two parts: the part before and after issuing the barrier. Consequently, the unbalanced instructions for various execution blocks can lead to stalling of these blocks (see more in the section 2.6.3). This can be overcome by parallel processing of work-groups on the same multiprocessor.

Chapter 3

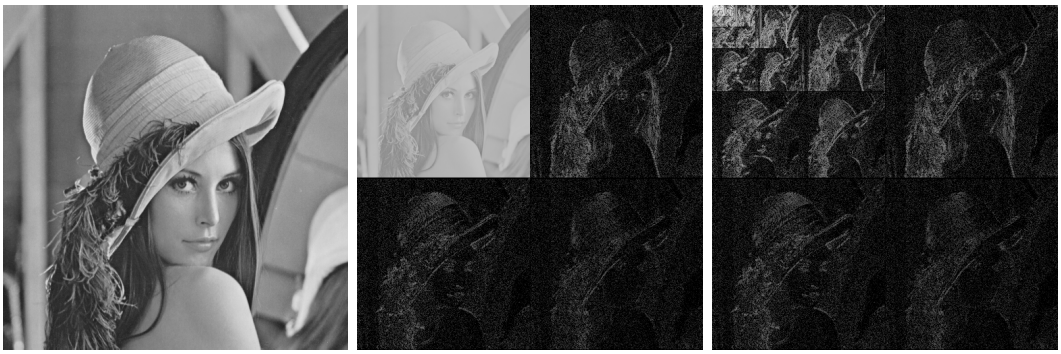
Discrete Wavelet Transform

Discrete wavelet transform (DWT) [56] is an integral transform capable of decomposing a signal to frequency and time/space. It is successfully used in many areas, such as encoding standards (JPEG2000 [73], Dirac codec), DjVu PDF reader, light field compression, signal filtering, noise reduction algorithms [63], as a part of convolutional networks [68, 54], and many other applications. Many types of wavelet are used in such applications but the most commonly used wavelets are Cohen–Daubechies–Feauveau 5/3 and 9/7 wavelets [30] used for lossless and lossy JPEG2000 compression algorithm. Recursive application of 2D DWT on low-frequency outputs ensures decomposing signal to multiple frequencies for such algorithms. Visualization of decomposition of 2D DWT using CDF97 wavelet is shown in Figure 3.1. During last two decades the acceleration of DWT have been widely studied using:

- pixel shaders in graphics pipeline paradigm [78, 74, 75, 76],
- blending in graphics pipeline paradigm [41, 42],
- GPGPU paradigm by separable row-column approach [36, 37, 15, 39, 40, 77, 62, 63, 67, 70],
- GPGPU paradigm by separable row-column approach for 3D decomposition [25, 38],
- GPGPU paradigm by separable pipeline approach [52, 53, 44],
- GPGPU paradigm by block-based tile approach [57, 15],
- GPGPU paradigm by block-based seamless approach [14, 34, 69, 60, 61],
- parallel CPU approach [21, 29, 37, 38],
- SIMD single-thread CPU approach [66, 20, 24, 28, 27, 51, 26, 58],
- and FPGA[1] [23, 79, 65, 32, 33].

The rest of the chapter provides information on: most commonly used schemes for 2D DWT calculation (in the section 3.1), the approaches for mapping these schemes to the platforms (in the section 3.3), and the variants for mapping the work-groups in these approaches using GPGPU paradigm onto the image regions (in the section 3.2).

[1] FPGA - Field Programmable Gate Array



(a) Original image (b) 1^{st} level of decomposition (c) 6^{th} level of decomposition

Figure 3.1: 2D DWT visualization

3.1 Wavelet Transform Schemes

One decomposition step of 2D DWT can be calculated using a separable application of 1D DWT in horizontal and vertical axis by Convolution Scheme or Lifting Scheme or spatial 2D schemes such as Implosion Scheme. In the rest of this chapter, the DWT schemes are represented using dataflow diagrams. The section 8.2 provides information on the schemes of matrix and block diagrams.

3.1.1 Convolution Scheme

The most common way to calculate the DWT is using sets of convolution filters. For each wavelet type, two mirror filters are applied interleaved, a low-pass and a high-pass one (Figure 3.4(a)). The 2D DWT can be calculated using a non-separable convolution scheme formed by fusion of vertical and horizontal high-pass and low-pass filters (Figure 3.3) or by separable convolution scheme that applies 1D convolution filters sequentially on a horizontal and vertical axis (Figure 3.2). In the separable convolution scheme, the low-pass and high-pass filters are applied in the horizontal axis, forming horizontal low-pass (L) and horizontal high-pass (H) outputs. The outputs are further processed by vertical low-pass and high-pass filters that form 2×2 shaped quadruple of spatially interleaved outputs: a low-low (LL), a low-high (LH), a high-low (HL) and a high-high one (HH). In the convolution scheme, the calculation of each element is independent of others. As a consequence, thread communication or synchronization is not required during calculation. However, the calculation consists of many operations (especially in a fused 2D version), and in-place calculation of DWT requires caching of a part of an image. The section 11.2 presents a list of the schemes and their properties.

3.1.2 Lifting Scheme

In contrast to the convolution scheme, the lifting scheme introduced by W. Sweldens [72] significantly reduces the number of operations for DWT calculation. W. Sweldens formed a method for factorizing DWT convolution filters to a sequence of K pairs of predict-update steps. Later in 1998 I. Daubechies [31] factorized her CDF 9/7 wavelet to a sequence of 2 pairs of predict-update steps $(\alpha, \beta, \gamma, \delta)$. In contrast to the convolution scheme, the lifting scheme reduces the number of operations on CDF9/7 by 50% and makes in-place DWT

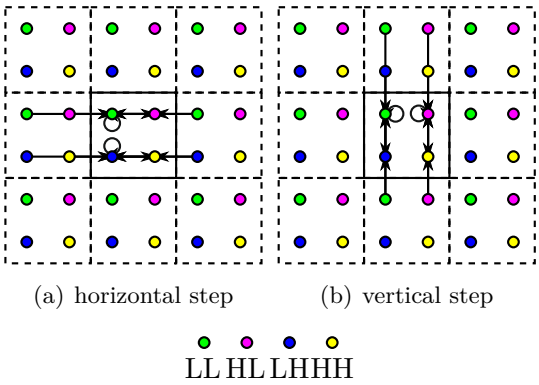


Figure 3.2: 2D dataflow diagram representing calculation of separable convolution scheme from the view of single thread. The thread is mapped onto quadruple of pixel elements (dots in solid box). The arrows indicate operations issued by the thread. The straight arrows indicate fused multiply-add operations, and the self-directed circle arrows indicate multiplication operations.

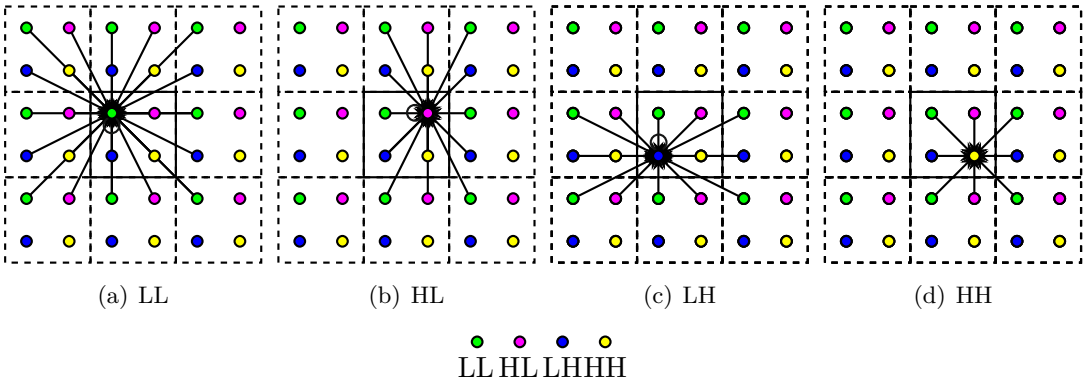
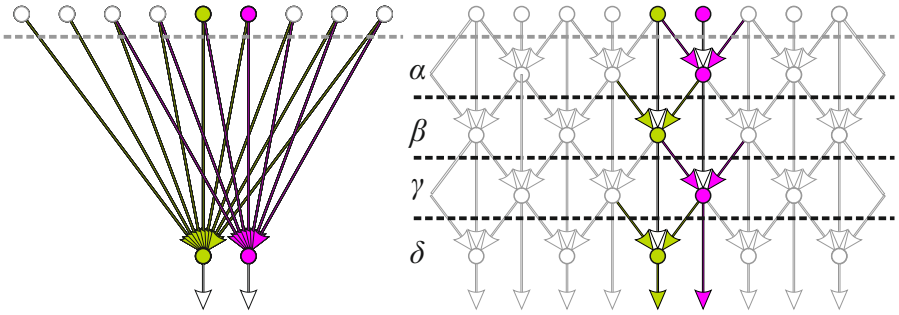


Figure 3.3: 2D dataflow diagram representing calculation of non-separable convolution scheme from the view of a single thread. The thread is mapped onto a quadruple of pixel elements (dots in solid box). The arrows indicate operations issued by the thread. The straight arrows indicate fused multiply-add operations, and the self-directed circle arrows indicate multiplication operations.



(a) convolution (b) lifting scheme

Figure 3.4: Comparison of 1D DWT convolution scheme (a) and lifting scheme (b) on CDF 9/7 wavelet ($K = 2$). The dataflow diagram represents the calculation of 1D DWT from the view of a single thread. The thread is mapped onto the pair of pixel elements and calculates L (green dot) and H (purple dot) outputs. Green and purple arrows indicate fused multiply-add operations issued by the thread. The horizontal black lines are memory synchronization points, and the horizontal gray line is implicit memory synchronization used in cases where inputs are copied to temporary memory.

without caching of image possible. However, such factorizing introduces data dependency between each pair of these $2K$ steps (Figure 3.4(b)). Consequently, synchronization with adjacent threads is required after each step when the threads are mapped onto the part of an image. Moreover, the communication between work-groups (presented in the section 2.5) is either impossible or costly, so the work-groups have to correctly calculate borders themselves. As a consequence, the lifting scheme is not well-performing on platforms or paradigms with costly barriers. Calculation of 2D DWT using lifting scheme is formed either as a separable lifting$^+$ scheme by interleaving predict and update lifting steps for each direction (Figure 3.5) or as a separable lifting scheme by calculation all of the horizontal lifting steps followed by the vertical ones (Figure 3.6). The section 8.2.5 provides more information on the separable lifting$^+$ scheme whereas the lifting one is further described in the section 7.3.1. Note that the alternative notation of the separable lifting$^+$ scheme used in the section 8.2.5 is Sweldens. The separable lifting scheme is implemented by Tenllado et al. [75], Laan et al. [53], and Wang et al. [77]. Moreover, comparison of separable lifting scheme and separable convolution scheme is examined by Tellando et al. [75] and Laan et al. [53].

3.1.3 Implosion Scheme

Iwahashi et al. [46, 45, 47] formed a new spatial scheme denoted as the implosion scheme. In contrast to the separable lifting scheme variants, the implosion scheme reduced the number of calculation steps by 25% and increased the number of arithmetical operations by 50%. The scheme consists of 3 spatial steps (Figure 3.7). The first of them calculates HH output by predict filter, the second one calculates HL and LH outputs by the combination of predict and update filters. Finally, the last one calculates LL output by update filters. The section 8.2.6 presents detailed information on the implosion scheme. Note that the alternative notation used in the section 8.2.6 is Iwahashi.

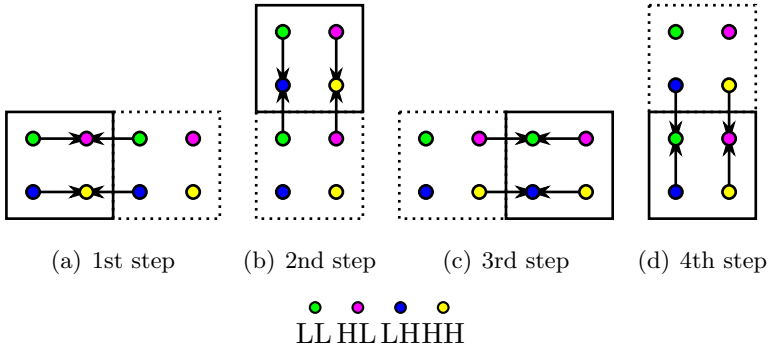


Figure 3.5: 2D dataflow diagram representing calculation of separable lifting$^+$ scheme from the view of a single thread. The thread is mapped onto a quadruple of pixel elements (dots in solid box). The arrows indicate fused multiply-add operations issued by the thread.

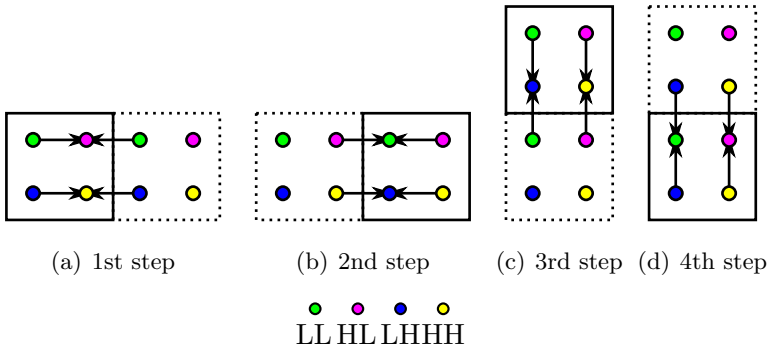


Figure 3.6: 2D dataflow diagram representing calculation of separable lifting scheme from the view of a single thread. The thread is mapped onto a quadruple of pixel elements (dots in solid box). The arrows indicate fused multiply-add operations issued by the thread.

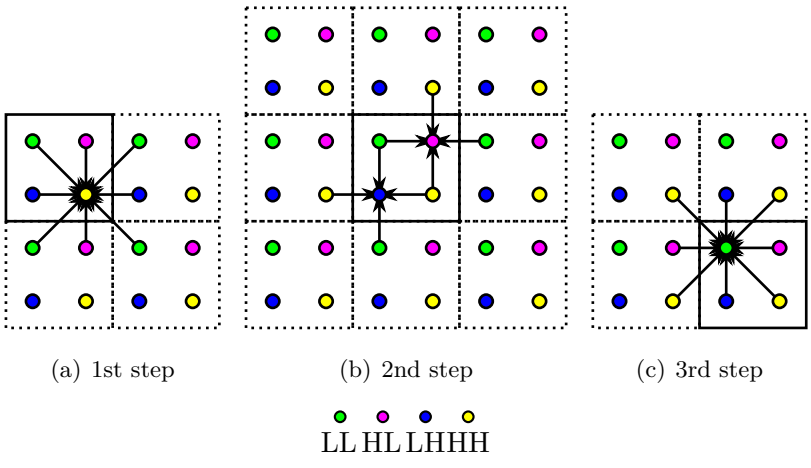


Figure 3.7: 2D dataflow diagram representing calculation of implosion 2D DWT scheme from the view of a single thread. The thread is mapped onto a quadruple of pixel elements (dots in solid box). The arrows indicate fused multiply-add operations issued by the thread.

3.2 Schemes to Platform Mapping Strategies

Two known strategies are known for the calculation of 2D discrete wavelet transform. The first of them, the Separable strategy, calculates such transform by horizontal and vertical passes. In contrast to the Separable strategy is the Block-based strategy that processes the 2D DWT by a single pass.

3.2.1 Separable

Separable strategy can be further subdivided into a pipelined variant incorporating a sliding-window approach and a row-column variant that splits an image into tiles.

Pipeline variant

The work-groups are mapped onto horizontal/vertical strips in horizontal/vertical transform pass in the pipeline variant. The algorithm of the sliding window ensures seamless transform. The sliding window approach combined with separable lifting scheme is used by Laan et al. [52, 53].

Row-column variant

In the row-column variant, the image is decomposed into tiles. Each work-group is mapped onto one of the tiles and calculates horizontal or vertical transform on that tile. These tiles have to be overlapped in the transform axis to ensure a seamless transform. When using a graphics pipeline, the communication between threads during kernel runtime is impossible, and local memory is unavailable. In that case, each thread is mapped onto the output region and processed independently to other ones. Additionally, multiple steps can be processed by multiple kernel calls. The row-column variant was widely studied using graphics pipeline paradigm [74, 75, 78, 76] and using GPGPU paradigm [36, 37, 15, 39, 40, 77, 62, 63].

3.2.2 Block-based

The 2D DWT is calculated using a single kernel that incorporates a sequence of 1D filters or 2D filters in a Block-based strategy. Additionally, two possible 2D DWT outputs variants exist: a tiled one and a seamless one.

Tiled output

An image is split into non-overlapping regions, so-called tiles, in the tiled variant. These tiles are processed separately, each one of them by one work-group. The advantage of the tiled version is that the region needed for the tile calculation has the same size as the tile, so it leads to coalesced access to global memory with the possibility of alignment of these accesses to cache lines (see the section 2.4.2 for more information on global memory). Aligned memory load and store operations lower the L2 cache usage and potentially lead to better performance. Especially, it is beneficial on platforms without support for global memory load/store operations caching (such as Nvidia Tesla architecture). Consequently, the tile-based algorithms have to deal with replacing pixel elements from adjacent tiles that are missing. Commonly, the symmetric extension of tile is used as a replacement. Creating such extension can be calculated either by extending the tiles in local memory by the

necessary size for correct calculation and writing the symmetric extension on extended areas or by remapping the pointers to out-of-range pixel elements to their symmetric counterparts.

The extending variant requires an extension of the work-group local memory over the tile by: $2K$ on the top and left side of the tile, $2K - 1$ on the bottom and right side of the tile where K is a number of predict-update pairs. Such extension is filled by symmetrically mirrored pixels from the tile block.

The remapping variant requires a calculation of offsets to the adjacent positions during kernel calculation instead of using constant offsets that are usable by the extended local memory variant. Moreover, the horizontal offsets differ on border threads within warps so at least the horizontal part of offsets cannot be calculated using scalar units on AMD GPUs. However, on some platforms, integer units block and float units block can be used in parallel using ILP (e.g. Nvidia Turing platform) by the same warp scheduler (see the section 2.6.3 for more information). For such platforms, the remapping variant should be beneficial. The advantage of the remapping variant lies in the direct mapping of work-groups to tiles and calculation without the need for thread divergence (further described in the section 2.6.2).

Although the symmetric extension is used, the tile output variant introduces block artefacts on the borders of tiles.

The tile output variant of block-based strategy combined with separable lifting scheme is used by Matela et al. [57] and Blazewicz et al. [15].

Seamless output

In contrast to the tiled output approach, the seamless one ensures the absence of such block artefacts. The image is split into non-overlapped regions similarly to the tiled version. Similarly, each work-group calculates the output of its given tile. In contrast to the tiled version, the seamless one forces the work-groups to load tiles enlarged to the same size as the local memory size in the extended local memory variant of the tiled output variant. After the load stage, the rest of the calculation is the same as in the extended local memory variant of the tiled output version.

Seamless output variant combined with separable lifting scheme is implemented by Arguello et al. [14], Song et al. [69], Enfedaque et al. [34], and Quan et al. [60, 61].

3.3 Work-group to Image Mapping

Two possible work-group mapping variants can occur using a seamless output or an extended local memory variant of the tiled output version: mapping to local memory and mapping to output tile.

Region Mapping

The mapping of the work-group to the local memory is similar to the example presented in the section 2.6.2. In that variant, the threads load the data to the local memory, and the symmetric extension is created either only on the image borders on the seamless version or on every tile border in the tiled version. Global memory of modern GPUs is capable of multicast and broadcast of values; thus, the symmetric extension can be created by loading mirrored pixels directly from global memory and storing them to local memory on thread position. Loading of quadruple of pixels per thread can be processed by 4 separate loads where each of the loaded pixels is saved to a separate block in local memory (used

for ensuring bank conflict free load and store operations described in the section 2.4.4). After that, the calculation of scheme steps of 2D DWT starts. During the calculation of the steps, some of the border threads can be masked-out because they no longer produce valid intermediate results/outputs. All threads mapped onto the extended area can be masked-out for the last step of the scheme calculation. Note that masking-out the threads brings conditional jumps to the algorithm that possibly leads to performance degradation (for more information, see the section 2.6.2). The mapping combined with switching the threads IDs from row-major to column-major order is used by Song et al. [69], Enfedaque et al. [34], and Quan et al. [61]. Additionally, the mapping without switching the thread IDs is used by Wavelet transform framework presented in the chapter 5.

Tile Mapping

In contrast to the mapping of work-groups to the local memory is the mapping to the output tiles. The tiles mapped to work-groups are smaller than the local memory regions so some of the threads need to be reused for a load to local memory purposes. Additionally, calculation of all scheme steps except for the last one requires the threads to be reused for intermediate results processing. Reusing some of the threads requires issuing additional instructions for the calculation of their secondary positions. Moreover, it can lead to an unbalanced number of processed elements by work-group warps and possibly to an unbalanced number of issued instructions by warps schedulers as well (depending on scheme, warp scheduler strategy and a number of work-groups per multiprocessors). Consequences of unbalanced number of processed elements by work-group warps are described in the section 2.6.2. The mapping is used by Matela et al. [57] and Blazewicz et al. [15].

Chapter 4

Goals of the Thesis

The scientific goal of the thesis was to prove the hypothesis: *"It is possible to achieve speedup of 2D DWT comparing to the state-of-the-art using well balancing of arithmetic operations, barriers, and local memory usage on various parallel computing architectures."*

The proof of the hypothesis is experimental. The experiments executed in order to prove the hypothesis are pesented in papers located in the chapters 6, 7, 8, 9 and 10. A summary of the results of these experiments is presented in the section 11.1.

For the purpose of evaluation, testing, visualization, and transforming the video, another suplementary goal was set to create a Wavelet transform framework capable of evaluating various approaches for 2D DWT calculation that can be used with various wavelets as well. Such Wavelet transform framework was successfully created and described in the chapter 5.

Core Papers and Contributions

The core papers contain experimental proofs of the above defined hypothesis and their contents are presented in the thesis as follows:

Kucis, M., Barina, D., Kula, M. and Zemcik, P. 2-D Discrete Wavelet Transform Using GPU. In: *International Symposium on Computer Architecture and High Performance Computing Workshop*. IEEE, October 2014, p. 1–6. DOI: 10.1109/SBAC-PADW.2014.13. ISBN 978-1-4799-7014-8

Contribution: 30% Reformated paper: in the chapter 6 Original paper: [48]

Kula, M., Bařina, D. and Zemčík, P. Block-based Approach to 2-D Wavelet Transform on GPUs. In: *Information Technology: New Generations*. Springer International Publishing, 2016, vol. 448, p. 643–653. Advances in Intelligent Systems and Computing. DOI: 10.1007/978-3-319-32467-8_56. ISBN 978-3-319-32467-8

Contribution: 50% Reformated paper: in the chapter 7 Original paper: [50]

Bařina, D., Kula, M. and Zemčík, P. Parallel wavelet schemes for images: How to make the wavelet transform friendly to parallel architectures. *Journal of Real-Time Image Processing*. Springer Science and Business Media LLC. 2019, vol. 16, no. 5, p. 1365–1381. DOI: 10.1007/s11554-016-0646-3. ISSN 1861-8200

Contribution: 40% Reformated paper: in the chapter 8 Original paper: [22]

Barina, D., Kula, M., Matysek, M. and Zemcik, P. Accelerating Discrete Wavelet Transforms on Parallel Architectures. *Journal of WSCG*. Union Agency. 2017, vol. 25, no. 2, p. 77–85. ISSN 1213-6972

Contribution: 25% Reformated paper: in the chapter 9 Original paper: [17]

Barina, D., Najman, P., Kleparnik, P., Kula, M. and Zemcik, P. The Parallel Algorithm for the 2D Discrete Wavelet Transform. In: *Ninth International Conference on Graphic and Image Processing (ICGIP 2017)*. SPIE - the international society for optics and photonics, 2017, vol. 10615, no. 4, p. 1–6. DOI: 10.1117/12.2302881. ISBN 978-1-5106-1741-4

Contribution: 20% Reformated paper: in the chapter 10 Original paper: [19]

Additional Papers and Contributions

The additional papers are related to DWT similarly to the core papers but they do not directly contribute to proving of the hypothesis, or their contributions are contained in the above core papers as well.

Kula, M., Barina, D. and Zemcik, P. New Non-Separable Lifting Scheme for Images. In: *IEEE International Conference on Signal and Image Processing*. IEEE, 2016, p. 292–295. DOI: 10.1109/SIPROCESS.2016.7888270. ISBN 978-1-5090-2375-2

Contribution: 50% Original paper: [49]

Barina, D., Kula, M. and Zemcik, P. Simple Signal Extension Method for Discrete Wavelet Transform. In: *Proceedings of 2016 IEEE International Conference on Signal and Image Processing*. IEEE, 2016, p. 534–538. DOI: 10.1109/SIPROCESS.2016.7888319. ISBN 978-1-5090-2375-2

Contribution: 10% Original paper: [18]

Barina, D., Kula, M., Matysek, M. and Zemcik, P. Accelerating Discrete Wavelet Transforms on GPUs. In: *International Conference on Image Processing (ICIP)*. IEEE Signal Processing Society, 2017, p. 2707–2710. DOI: 10.1109/ICIP.2017.8296774. ISBN 978-1-5090-2175-8

Contribution: 20% Original paper: [16]

Chapter 5

Wavelet Transform Framework

Wavelet transform framework is designed for execution and evaluation of various approaches for seamless 2D DWT calculation. The framework is capable of evaluation of various schemes:

- separable lifting, lifting$^+$, separable convolution, and implosion schemes described in the section 3.1,
- explosion and non-separable lifting schemes introduced in the chapter 8,
- non-separable polyconvolution and non-separable convolution schemes introduced in the chapter 9,
- separable polyconvolution described in the chapter 9,
- and separable polyconvolutionX and polyconvolitionY schemes introduced in the chapter 11.4.

The complete list of the implemented schemes is presented in the section 11.2. The framework is not limited to wavelets evaluated in the following papers (CDF 5/3, CDF 9/7, and DD 13/7 wavelets). It is useful for wavelets of any degree and any number of predict-update steps as well. Moreover, it supports texture and global memory utilization, several in-warp optimization techniques, various regions to thread mapping, single or double buffering and the optimization approach introduced in the section 8.4. The framework was used for the evaluation of 2D DWT calculation methods using OpenCL framework [13] in the chapters 7,8, and 9 and in the papers [49] and [16].

The rest of the chapter encompasses: description of the framework structure (in the section 5.1), explanation of the properties of the framework‘s kernel generator module (in the section 5.2), supported variants of DWT on the platforms and work-group to image region mappings (in the section 5.3) that are described in the previous chapter 3, visualization and description of the structure of the kernel generated by the kernel generator module (in the section 5.4), partitioning of the local memory in the kernel (in section 5.5), and the supported warp optimization techniques (in the section 5.6).

5.1 Framework Structure

Wavelet transform framework structure is visualised in Figure 5.1. The framework is structured into two separate parts that correspond to two processing phases: the resources preparation phase and the image processing loop phase.

The preparation phase of the framework is capable of creating of objects that are required for wavelet framework to run based on user-defined command line arguments. Such objects are:

- Video Reader that is capable of read video frames in a loop. Video Reader can be configured for either software decoding using OpenCV/FFMpeg or HW decoding of video using FFMpeg. The reader is enabled only when the input device memory type is set to texture or global memory (memory-less computation is disabled).
- Video Writer that is capable of writing the 2D DWT output frames to output video file. Video Writer uses OpenCV as underlying library. Writing of video frames can be enabled by command line argument only when using of texture or global memory is enabled.
- Output Renderer that is capable of rendering the 2D DWT output video frames onto the SDL/OpenCV GUI window configured by user. Rendering such frames can be enabled by command line argument only when using of texture or global memory is enabled.
- OpenCL DWT library that is capable of multiscale decomposition of image by 2D DWT transform using user-defined parameters. The most important part of the library is Kernel generator that is usable for generation of OpenCL C kernel body [13] by user-defined parameters. The kernel generator is described in the next section.

5.2 Kernel Generator

Kernel generator module generates arithmetic part of block-based 2D DWT GPU kernel. It takes wavelet properties (like lifting step coefficients) and desired scheme as inputs. An output of the generator is the whole kernel body except for loading and storing data from/to input/output memory. Flowchart diagram of output generated by kernel generator is visualised in Figure 5.2.

The generator was dramatically evolving during past years. The first version that was used in the paper introduced in the chapter 7 is implemented directly by OpenCL C language macros [13]. Nevertheless, this version was difficult to maintain and hardly extensible to other schemes. Some of the language macros had to be hand-written (like non-separable convolution scheme, non-separable polyconvolution scheme, etc.), and a compilation of complex schemes never ended on some architectures (AMD VLIW architectures). To cope with these issues, we came up with the idea of rewriting the generator as a library in C++ used further in papers introduced in the chapters 8 and 9. The current version is not limited only to CDF 5/7, CDF 9/7 and DD 13/7 (used in the papers presented in the chapters 8 and 9), but it can also be applied on wavelets with any degree and any number of predict-update pairs. The current version takes wavelet definition (wavelet coefficients, number of predict-update pairs, degree of wavelet), thread mapping definition (number of quads in both axes per thread, work-group size), scheme selection (desired scheme from Table 11.1,

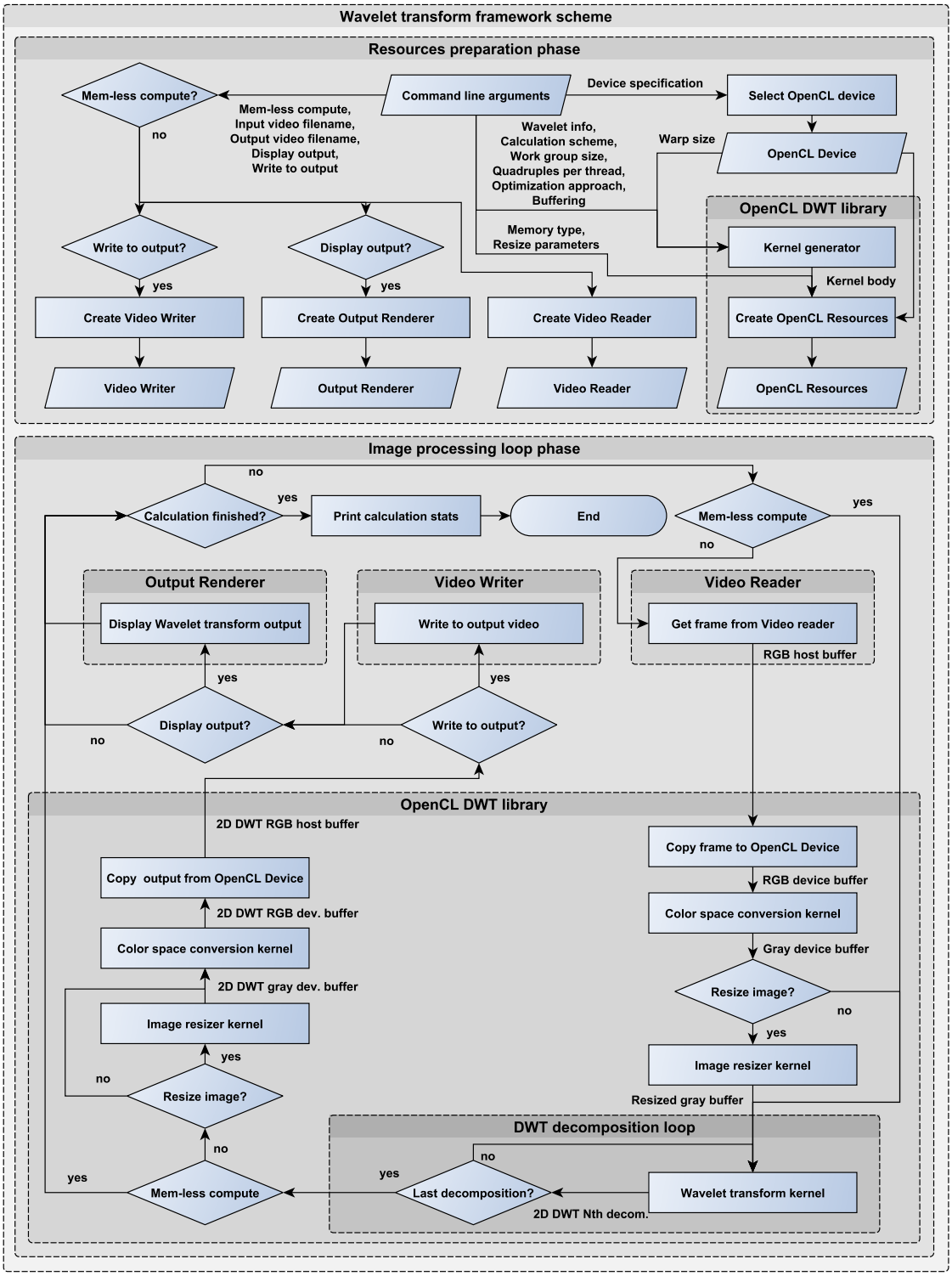


Figure 5.1: Flowchart diagram representing structure of wavelet transform framework.

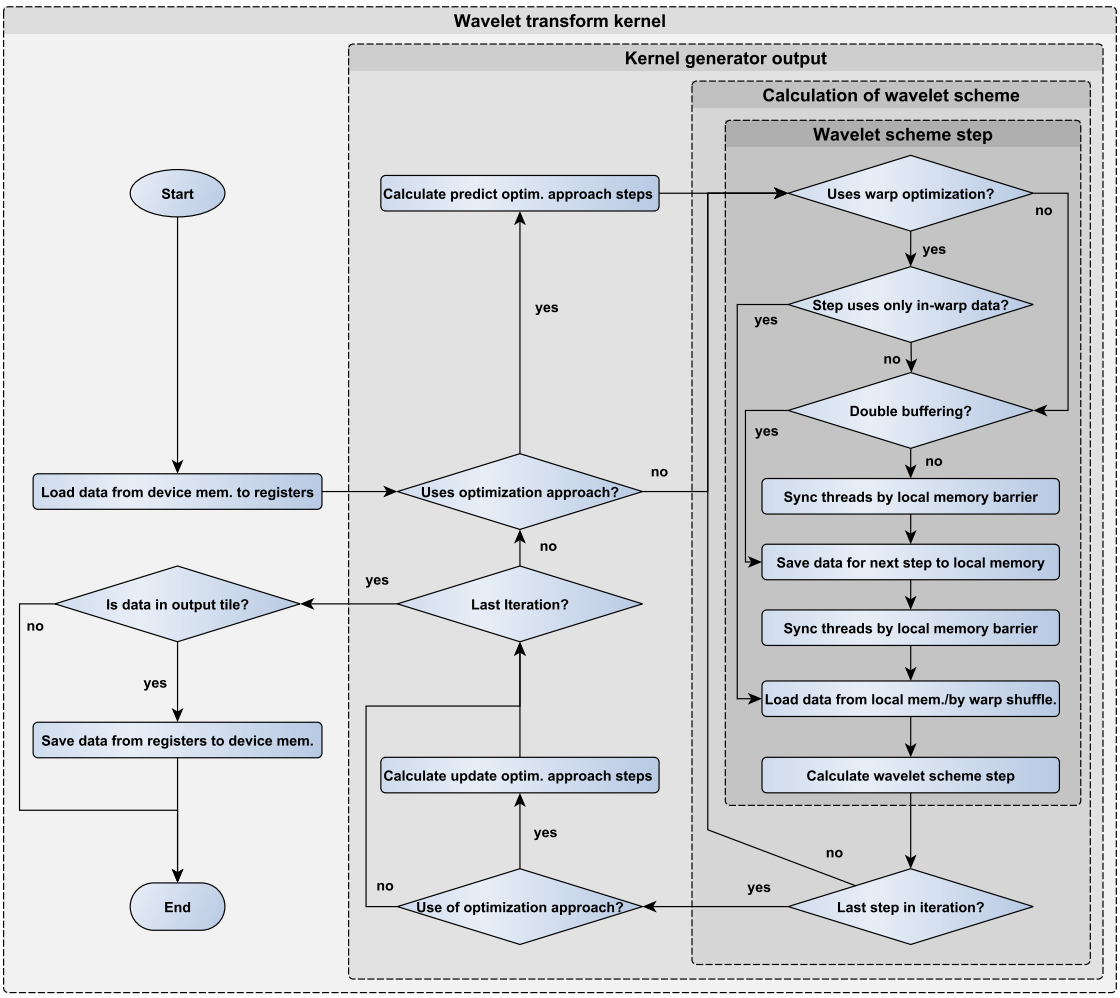


Figure 5.2: Flowchart diagram representing structure of 2D DWT GPU kernel. In one iteration of wavelet scheme calculation are calculated either all steps in separable/non-separable convolution schemes or one predict-update lifting step (K) in all other schemes. All decisions described in the kernel generator output part of the diagram is evaluated in kernel generation process.

optimization approach introduced in the section 8.4, warp optimizations), and warp size of used device as inputs and it generates OpenCL kernel body that consists of multiplication/fused multiply-add instructions, local memory load/store instructions, in-warp shuffles instructions and barriers.

5.3 Mapping Threads

The framework is capable of the calculation of seamless 2D DWT. The implementation uses the block-based strategy with seamless output and Region Mapping with all advantages and disadvantages that configuration entails (see the sections 3.2.2 and 3.3 for more information).

Work-group Mapping Properties

According to information from the section 3.3, work-groups are mapped onto overlapped image regions that need to be loaded to local memory for the correct calculation of given tiles. Threads in work-groups of size (G_x, G_y) are mapped onto thread regions that contain a user-defined number of quads of desired shape (Q_x, Q_y). The size of the thread region (R_x, R_y) can be calculated as follows:

$$(R_x^t, R_y^t) = (2Q_x, 2Q_y) \tag{5.1}$$

and, similarly, the work-group region (R_x^g, R_y^g) as:

$$(R_x^g, R_y^g) = (G_x R_x^t, G_y R_y^t). \tag{5.2}$$

Note that the required size of the work-group region is narrower by 1 pixel on the right and the bottom part but omitting that reading leads to additional branching. Moreover, the framework uses a branch-less calculation of the whole 2D DWT. The only branch is used to mask-out out-of-tile threads while writing outputs to the global memory (see the sections 2.6.2 and 2.6.2 for more information on branching consequences). Experiments conducted during research show that the described configuration is the best performing one in most combinations of wavelet and schemes.

Size of the tile given to the work-group (T_x, T_y) that is mapped onto the region (R_x^g, R_y^g) depends on a number of predict-update steps (K) and wavelet degree (D) and can be calculated as:

$$(T_x, T_y) = (R_x^g - 4DK, R_y^g - 4DK). \tag{5.3}$$

Extending work-group size (G_x, G_y) and a number of quadruples per thread (Q_x, Q_y) lead to lower work-group to region mapping overhead N that can be expressed as:

$$N = 100(1 - \frac{T_x T_y}{R_x^g R_y^g}). \tag{5.4}$$

The overhead of the region to work-group mapping for a various combinations of work-group sizes, number of quadruples per thread and wavelets types is presented in Table 5.1.

work-group	quadrup. per thread	CDF 5/3	CDF 9/7	DD 13/7	local memory	threads per group	pixels per thread	pixels per group
32×32	1×1	13.78%	30.61%	51.48%	4kB	1024	4	4096
32×16	1×1	21.90%	52.38%	96.92%	2kB	512	4	2048
32×8	1×1	42.22%	128.57%	392.31%	1kB	256	4	1024
16×16	1×1	30.61%	77.78%	156.00%	1kB	256	4	1024
16×8	1×1	52.38%	166.67%	540.00%	0.5kB	128	4	512
32×32	1×2	10.11%	21.90%	35.81%	8kB	1024	8	8192
32×16	1×2	13.78%	30.61%	51.48%	4kB	512	8	4096
32×8	1×2	21.90%	52.38%	96.92%	2kB	256	8	2048
16×16	1×2	21.90%	52.38%	96.92%	2kB	256	8	2048
16×8	1×2	30.61%	77.78%	156.00%	1kB	128	8	1024
32×32	1×4	8.36%	17.97%	29.13%	16kB	1024	16	16384
32×16	1×4	10.11%	21.90%	35.81%	8kB	512	16	8192
32×8	1×4	13.78%	30.61%	51.48%	4kB	256	16	4096
16×16	1×4	17.97%	42.22%	76.55%	4kB	256	16	4096
16×8	1×4	21.90%	52.38%	96.92%	2kB	128	16	2048
32×32	2×2	6.56%	13.78%	21.76%	16kB	1024	16	16384
32×16	2×2	10.11%	21.90%	35.81%	8kB	512	16	8192
32×8	2×2	17.97%	42.22%	76.55%	4kB	256	16	4096
16×16	2×2	13.78%	30.61%	51.48%	4kB	256	16	4096
16×8	2×2	21.90%	52.38%	96.92%	2kB	128	16	2048

Table 5.1: Overhead of the work-group to region mapping approach (N) used by the framework on CDF 5/3, CDF 9/7, and DD 13/7 wavelets. Local memory size is presented per one location (subband). A number of memory locations needed for various schemes are presented in Table 11.1. A number of pixels per work-group is connected to a minimal number of registers needed to store the region mapped to the work-group. The real number of registers depends on scheme, platform and configuration and typically is around twice as much. In most cases, the factor limiting the occupancy is the local memory size.

Best Performing Settings

Experiments show that the best performing work-group size for modern platforms is typically 32×16. However, some platforms (like Intel GPUs or ARM Mali GPUs) cannot create a work-group with more than 256 threads. The performing size of 32×8 threads is optimal on such platforms, even if it is less efficient than 16×16. Note that extending the work-group size can lead to lower occupancy of the GPU (more information in the section 2.6.5) and worse handling of barriers (see the section 2.6.6 for details).

Calculation of more quadruples per thread is more efficient but consumes more local memory and registers per thread that lower the occupancy of GPU (for more information on occupancy, see the section 2.6.5). Experiments show that on most platforms, the best performing amount of quadruples per thread lies between 2 or 4 (exception is ARM Mali Midgard architecture that is best performing using 1 quadruple per thread). The peak location depends on the GPU occupancy that depends on the number of local memory locations per quadruple (the comparison is in Table 11.1). However, the schemes behave similarly to the ones with 1 quadruple per thread until the peak for the schemes is reached. Note that published researches presented in the next chapters use only 1 quadruple per thread.

5.4 Kernel Structure

The kernel instance execution process is visualized in Figure 5.3. The flowchart diagram of the process is presented in Figure 5.2. Firstly, the assigned regions (R_x^t, R_y^t) for threads are copied into thread registers. Then, the core of the 2D DWT calculation, generated by the Kernel Generator, starts to execute. When the optimization approach (introduced in the section 8.4) is enabled, the registers are updated using the horizontal and the vertical predict in-quadruple operations. After that, the data needed by other threads in the next step is saved to local memory. Then the calculation of wavelet scheme step is started consisting of:

- loading necessary data produced by adjacent threads using warp shuffle instruction when possible if enabled;
- loading rest of necessary data from local memory;
- moving data loaded from local memory to temporary variable when it is used multiple times (for prevention of multiple issuing of load instruction when using volatile memory);
- moving register data to temporary registers if needed (used when some registers are updated and used in the same step);
- update threads registers by calculation steps.

If the step was the last one in the scheme and the optimization approach is enabled, the in-quadruple update steps are calculated; otherwise, the updated data, needed by other threads in the following steps, is saved to local memory. If the single buffering is enabled, the barrier is placed before and after such saving; otherwise, this barrier is placed after the saving procedure. Then processing of the next step begins.

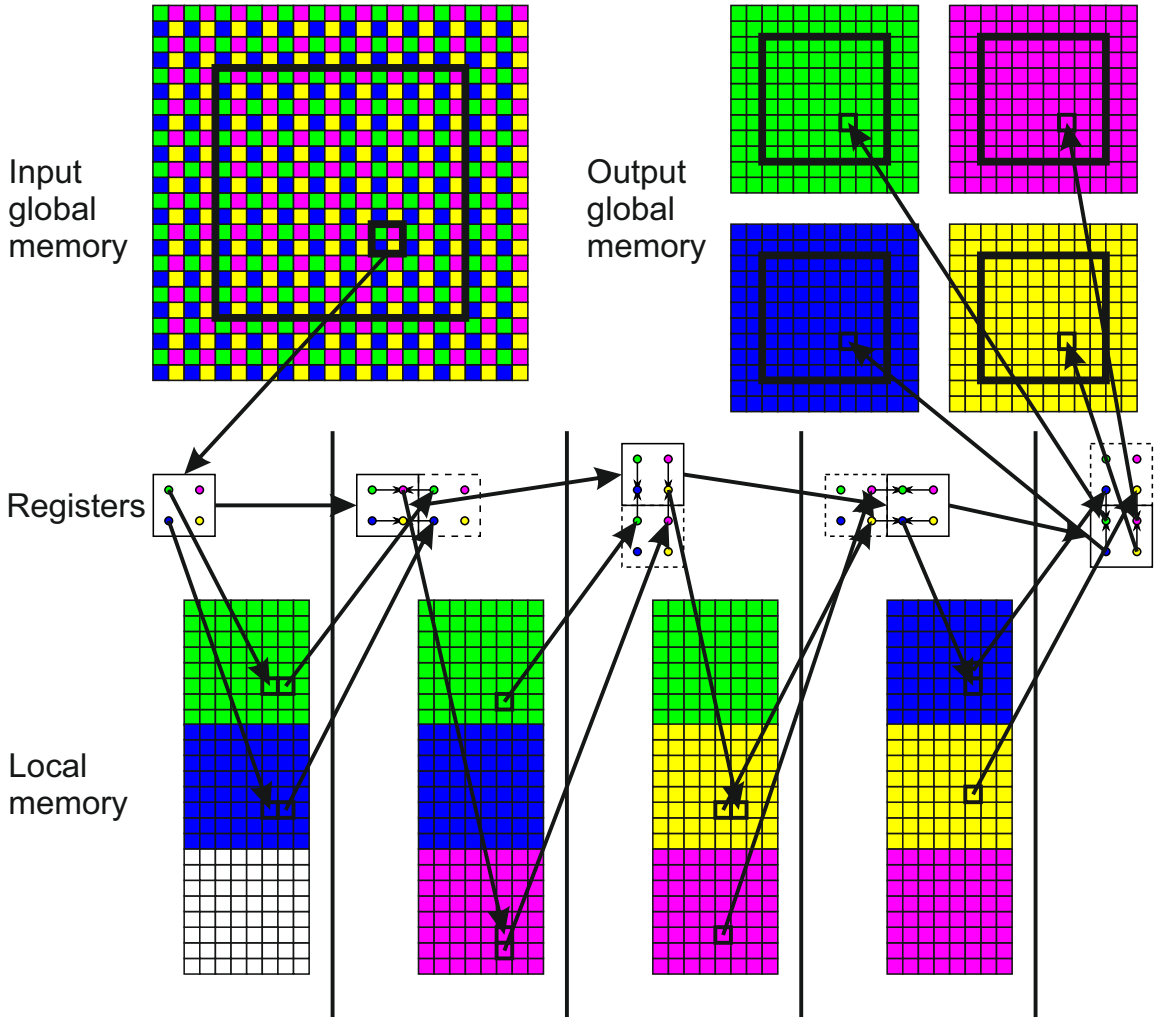


Figure 5.3: Kernel processing diagram representing calculation of separable lifting$^+$ scheme from the view of a single thread. The work-group size is set to 8×8. The thread is mapped onto a quadruple of pixel elements (register). The arrows in the register boxes represent the fused multiply-add operations. The vertical lines represent barriers. The local memory blocks are depicted in the bottom part of the image.

When the last step and optionally final optimization approach is calculated (the part generated by Kernel generator), the data of the tile (T_x, T_y) is saved to global/texture memory.

5.5 Local Memory Usage

In the wavelet transform kernel, the local memory is partitioned into a preface, local memory blocks, and postface. Preface and postface are not filled with any values, but they are needed when the threads on the border of the work-group try to load out-of-region values. This happens because of the usage of Region Mapping (see more in the section 3.3) combined with branch-less execution that does not mask-out any thread during the calculation. Each local memory block is the same sized as an appropriate work-group. Each thread within the appropriate work-group stores one value to that local memory block and this value is one subband from one of the given quadruples. Only subbands, loaded in the following calculation by other threads within the work-group, are stored. Local memory block structure combined with using 4B float point datatype leads to bank conflict free access to local memory (see the section 2.4 for more information on bank conflicts). The minimum number of possible used local memory blocks for double buffering can be defined as a number of positions read from the region threads of any adjacent threads summed with a number of regions that need to be written. Similarly, the minimum used local memory blocks for single buffering can be defined as a maximum from those two parameters. To further minimize the writes number, the values from thread region positions that are not updated stay in the same local memory blocks until they are recalculated or are not needed anymore. This behaviour leads to rotating the thread region positions in memory during the calculations steps. Additionally, the preface and postface parts of local memory are minimized as well.

5.6 Warp Optimizations

Warp optimization techniques enable barrier-less data exchanges between threads in the warp. This functionality is platform-dependent (see the section 2.5 for information about warp optimization). Warp optimization types supported by Kernel generator are:

Local optimization - When threads between calculation step change data on in-warp basis, barriers are removed. A `volatile` keyword is needed for ensuring load/store operations from the same memory location, even if a barrier is not presented. This leads to a lock of memory load/store ordering. A generator loads values to a temporary register when the data is used more than once to prevent unnecessary loads.

Shuffle optimization - Barriers are removed in the same way as within the local optimization approach but data is exchanged using shuffle instructions. No `volatile` keyword is needed, so load/store operations can be twisted freely by a compiler. Supported platforms are: Intel GPUs using OpenCL extension, AMD GCN GPUs using intrinsics that are exposed by ROCm Linux driver compiler, and Nvidia GPUs since Kepler architecture using inline assembler [7]. A warp optimization approach for all schemes and their properties has not been published yet.

5.7 Framework usage

The framework was primarily created for evaluation and testing purposes of the specific wavelets and the schemes. Over the time, it was evolved in a universal tool that is: able to generate GPU kernel implementations of user-defined wavelets without fixation to predefined number of predict-update steps (K) nor degree of wavelet (D), capable of use of global/texture memory for real data processing or enable of constant memory for obtaining raw throughput of DWT by memory-less processing, and capable of multiscale decomposition using interleaved or non-interleaved data mapping. Additionally, the framework is able to adapt the schemes to warp optimization techniques described in the section 5.6, optimization technique introduced in the section 8.4 and user-defined number of quadruples per thread. Moreover, the framework is capable of processing the video frames decoded by FFMpeg or OpenCV using SW or HW acceleration decoder, render the outputs by SDL or OpenCV frameworks, and writing the DWT output to video file.

Chapter 6

Article: 2-D Discrete Wavelet Transform Using GPU

This paper proposes a novel approach for the calculation of the vertical pass of the separable pipeline method (more information about separable pipeline method can be found in the section 3.2.1). It examines the impact of balance between barriers, arithmetic operations and local memory usage for such passes. Such a vertical pass approach overcame the throughput of the previous state-of-the-art one by 30%.

The approach reduces the barriers on two pairs of predict-update wavelet CDF 9/7 to the implicit one used for synchronization after global to local memory transfer for each shift of sliding window. Consequently, output for pixel element pairs given to GPU thread p_p needed to be calculated without sharing intermediate results from lifting steps with adjacent threads, leading to redundant calculations W_r around borders. Quadratic dependency of redundant operations W_r to predict-update pairs of the used wavelet K arises from

$$W_r = 2K(2K - 1). \tag{6.1}$$

Non-redundant operations W_{nr} can be calculated as

$$W_{nr} = 4Kp_p \tag{6.2}$$

and number of operations per pixel element pair W_p as

$$W_p = \frac{W_r}{p_p} + 4K \tag{6.3}$$

which shows that increase of number of pairs per thread lowers the impact of redundant operations.

For the given work-group size (G_x, G_y) and number of predict-update pairs of wavelet K the amount of local memory per work-group L_g can be calculated as

$$L_g = G_x(2p_pG_y + 2K - 1), \tag{6.4}$$

similarly the amount of local memory per thread L_t can be calculated as

$$L_t = 2p_p + \frac{2K - 1}{G_y} \tag{6.5}$$

which clearly shows that amount of local memory per thread depends linearly on the number of pairs per thread given to the thread. Thus, with an increasing number of such pairs,

algorithm	pairs per thread	operations per pair	local memory	barriers	occupancy	time/pixel 580GTX	time/pixel 4200M
Proposed	1	20	2944B	1	100%	114ps	1154ps
Proposed	2	14	4992B	1	100%	79ps	743ps
Proposed	4	11	9088B	1	83%	63ps	735ps
Proposed	8	9.5	17280B	1	33%	66ps	753ps
lifting	1	8	2560B	4	100%	101ps	949ps
Blazewicz	1	11	9088B	1	33%	167ps	2165ps

Table 6.1: Comparison of vertical pass algorithms. Time/pixel is measured in picoseconds.

the occupancy of multiprocessor linearly declines. The experiments performed on wavelet CDF 9/7 ($K = 2$) show that a combination of work-group size $(Gx, Gy) = (32, 8)$ and 4 pixel elements pairs per thread is the best performing combination on the tested platforms for vertical pipeline approach. Comparison of several algorithms combined with various number of the pixel pairs is presented in Table 6.1.

2-D Discrete Wavelet Transform Using GPU

Kucis, M., Barina, D., Kula, M. and Zemcik, P. 2-D Discrete Wavelet Transform Using GPU. In: *International Symposium on Computer Architecture and High Performance Computing Workshop*. IEEE, October 2014, p. 1–6. DOI: 10.1109/SBAC-PADW.2014.13. ISBN 978-1-4799-7014-8

Author contribution: 30%

Abstract

With the wide spread of the discrete wavelet transform, the need for its efficient implementation becomes increasingly important. This work presents an improved version of an algorithm suitable to compute the 2-D discrete wavelet transform on GPU. Depending on the GPU platform, it is suitable to split the 2-D transform computation into separated horizontal and vertical passes. Considering the horizontal passes, we have examined and chosen the best performing method among the already known ones. Furthermore, we have adapted this method for an existing algorithm computing the vertical transform pass. This step helps to reduce several synchronizations and arithmetic operations in the utilized computation scheme. For large data, the proposed vertical method achieves speed-up about 30 % compared to the current state of the art methods. In contrast to previously published works, the presented approach is built on the OpenCL parallel programming framework.

6.1 Introduction

The discrete wavelet transform (DWT) is a mathematical tool which is suitable to decompose discrete signal into several frequency components. It is frequently used as a basis of sophisticated compression algorithms. This paper focuses on the CDF 9/7 wavelet which is often used for image compression (e.g., JPEG 2000 standard). Responses of this wavelet can be computed by a convolution with two FIR filters, one with 7 and the other with 9 taps. In case of two-dimensional transform, the transform can be realized using a separable decomposition scheme. In this paper, we present several algorithms for 2-D transform computation suitable for modern GPUs.

In present personal computers, programmable graphics cards are almost always found. OpenCL is a framework for programming of heterogeneous computer systems, e.g. modern graphics processing units (GPU) found in personal computers, servers or mobile devices. When compared to CUDA framework, CUDA is limited to Nvidia hardware while OpenCL

is not platform dependent. The performance analysis [35] remarks that OpenCL offers similar performance to CUDA in general when compared fairly.

Several algorithms for the 2-D DWT computation using GPU have been published in the last decade. Some of them used the pixel shader through the Cg programming language. These were able to take advantage of SIMD operations offered by shader units. Other algorithms were built over the CUDA framework. We are not aware of any approach that use the OpenCL framework.

In this paper, we present several algorithms for 2-D DWT computation focusing on the parallel capabilities of programmable GPUs. Our implementation is based on the OpenCL framework. All the methods presented in this paper are evaluated using Nvidia GeForce GTX 580 graphics card equipped with 3072 MiB RAM and 512 streaming processors and Nvidia Quadro NVS 4200M graphics card equipped with 1024 MiB RAM and 48 streaming processors. Only the forward transform is evaluated since the inverse one has a symmetric nature and performs almost identically. We have also evaluated only one level of the DWT decomposition as the others again perform almost identically. In the chosen memory layout, the sub-bands are interlaced.

The rest of the paper is organised as follows. Related Work section summarizes the state of the art, especially existing GPU implementations. Proposed Approach section reviews significant algorithms and presents the proposed method. Finally, Conclusion section summarizes the paper and outlines the future work.

6.2 Related Work

The discrete wavelet transform [56] (DWT) is a mathematical tool suitable to decompose a signal into low-pass and high-pass frequency components. Such a decomposition can be performed at several scales resulting in a multi-scale signal representation. It is often used as a basis for sophisticated compression algorithms. A basis of such a transform consists of dilated and shifted wavelets. The Cohen-Daubechies-Feauveau [30] (CDF) 9/7 wavelet is a popular one as used, e.g., in JPEG 2000 image compression standard. One level of the discrete wavelet transform can be computed using the convolution with two mirror filters (a high-pass and a low-pass one). According to the total number of arithmetic operations, the more efficient computational scheme – the lifting – introduced by W. Sweldens in [72] exists. Using this scheme, the whole signal can be transformed in-place. In [31], I. Daubechies and W. Sweldens factored CDF 9/7 wavelet into four successive lifting steps employing short symmetric two-taps FIR filters.

For understanding of the following text, it may be important to understand the lifting scheme in more detail. Any discrete wavelet transform with finite filters can be factored into a finite sequence of N pairs of predict and update convolution operators P_n and U_n. Each predict operator P_n corresponds to a filter $p_i^{(n)}$ and each update operator U_n to a filter $u_i^{(n)}$. These operators alternately modify even and odd signal coefficients.

$$P_n(z) = \sum_{i=-l_n}^{g_n} p_i^{(n)} z^{-i} \tag{6.6}$$

$$U_n(z) = \sum_{i=-m_n}^{f_n} u_i^{(n)} z^{-i} \tag{6.7}$$

The discrete wavelet transform was also extended [55] to two (and more) dimensions. Specifically, the classical 2-D DWT is separable to series of 1-D transforms performed successively on rows and columns (or vice versa). For various requirements, different strategies of 2-D DWT implementation were developed. For example, the simplest row-column methods transform the whole image at once. Furthermore, the block-based methods transform the image using smaller blocks utilizing the row-column method inside. Finally, the pipelined methods such as [26] transform the image using column strips while employing the sliding window on them. Inside this window, the row and column transforms are combined together in a way that a vertical transform is interleaved on multiple columns. This concept was also extended to whole image resulting into the single-loop approach [51].

Implementation of 2-D DWT was also studied on modern programmable graphics cards. In this scenario, the input image have to be initially transferred from main memory into memory on the graphics card. Similarly, the resulting coefficients have to be transferred back.

OpenCL is a framework for general-purpose parallel programming across multiple device types (like GPUs, CPUs, etc.) and platforms. In this framework, a platform independent executable program is called the kernel. The kernel is executed on required number of threads (work-items) that identify their data and control flow by their N-dimensional indices. These threads are organized into work-groups with identical user-defined number of threads. The threads in such a group can cooperate with each other through local memory and barriers. Threads executing a kernel have access to: global memory – device memory that is accessible to all threads (like main GPU memory); local memory – small memory region that is shared by threads in work-group; constant memory – small memory that remains constant during the kernel execution; private memory – the private thread memory. Optionally, a device can support additional functionalities like textures, double type operations, etc.

In recent GPU architectures, the GPU contains the thread scheduler, multiprocessors, L2 cache and a memory controller. The thread scheduler allocates as much work-groups to multiprocessors as their resources allow. Thus, the resources like local memory size should be minimized. The multiprocessor contains blocks of processors, warp schedulers, local memory, load store units, etc. The allocated work-groups created by OpenCL framework is then divided into warps (hardware blocks with 32 threads). Execution instructions of these warps on blocks of processors are provided using warp schedulers dynamically. Due to fact that each instruction is executed on whole warp (half-warp on some architectures) at once, recommendations for ensuring good performance of memory operations exist. Global memory indices in warp should be coalesced. Otherwise, addition memory operations are executed. The local memory is organized into banks. Access to same banks from warp causes serialization. The serialization of local memory operations and uncoalesced global memory access can cause a performance degradation.

In [74] and [75], Ch. Tenllado *et al.* adapted the discrete wavelet transform on pixel (fragment) shaders of GPU. They used the Cg programming language and mapped the input image into textures. The authors compared convolution-based and lifting scheme implementations of CDF 9/7 discrete wavelet transform. The pixel processors support SIMD operations (4-element wide in this case). Using the convolution, the authors used a rearrangement step in order to allow to filter two image rows/columns in parallel. The results of this comparison speaks slightly in favor of convolution scheme. Moreover, the authors compared these results with corresponding CPU implementation using the CDF 9/7 wavelet. Ignoring CPU-GPU data transfer times, the GPU version significantly outperforms

the CPU counterpart. Finally, the authors state that the data transfers between the CPU and the GPU are the major bottleneck. However, these works are now obsolete as an instruction set of the modern GPUs does not contain the real SIMD instructions.

The other authors attempted to take advantage of the GPU using the CUDA programming model in [36], [57] or [15]. In [36], the convolution scheme is applied on each row. Then, the image matrix is transposed and the convolutions are applied on each column. Finally, the image is transposed back. The authors point out that important reductions of execution time are obtained for the CUDA version even when they take into account the time needed to copy data and results to and from the GPU memory. However, their CPU implementation seems to be naive compared to the state of the art methods, e.g. [51]. The latter two papers are focused on CDF wavelets and the lifting scheme. Their implementations splits the image into small tiles and performs several independent transforms on each of them. Moreover, in [15], another implementation performs the horizontal transform on the whole image. The horizontal transform is followed by transposition and by vertical filtering. The authors proposed omission of mutual thread synchronization at the cost of loading of more input pixels per each thread. Furthermore, the author of [57] consider the coalesced memory accesses to be crucial for a transform performance.

In [39] and [40], V. Galiano *et al.* compared several CUDA implementations of DWT. They used the CDF 9/7 wavelet and convolution-based algorithm on entire rows/columns. Their fastest implementation uses the coalesced memory access.

In [52], W. J. Laan *et al.* accelerated the Dirac video codec using the CUDA platform. Their DWT implementation is based on the lifting scheme. The authors highlight the coalesced memory access. In the vertical filtering, they divided the image into vertical strips and used a sliding window technique within each strip. However, this paper does not discuss the implementation of DWT in detail. In [53], W. J. Laan *et al.* provided a detailed analysis of the DWT implementation using the lifting scheme on the CUDA platform. They focused on several wavelets (including CDF 9/7) and used a sliding window approach within strips. Their design is a hybrid method between the row-column and block-based methods. Moreover, they implemented the methods for 2-D and 3-D data and compared to optimized CPU counterparts. Also here, the authors point out the importance of coalesced memory accesses.

As it can be seen, the problem of efficient 2-D discrete wavelet transform implementation on GPU was widely studied. Despite this fact, we see a gap in existing implementations. In the section below, we propose several improvements that lead to additional speedups.

6.3 Proposed Approach

In this section, several existing algorithms for the discrete wavelet transform computation are analysed. Initially, algorithms for horizontal pass of the transform are presented. For further experiments, the best performing algorithm for this pass is adopted. Furthermore, a vertical pass of the transform is discussed. Here, we have proposed several improvements over the state-of-the-art algorithm yielding to an additional speed-up.

In all of the algorithms below, two separated passes needed to compute the 2-D transform are considered. These are referred here to as a horizontal and a vertical pass. In general, both of these passes can share intermediate results between threads that access adjacent data. However, this sharing introduces some requirements for their mutual synchronization. Another approach might be to not share the intermediate results at all for

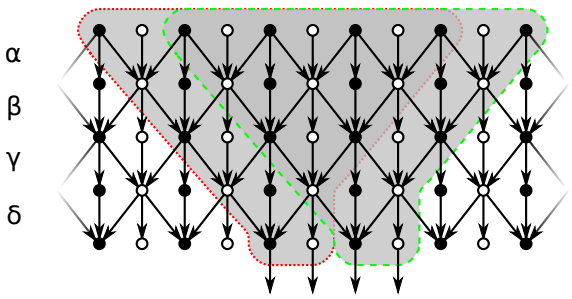


Figure 6.1: Lifting scheme of CDF 9/7 wavelet showing the calculation performed by a single thread (dotted and dashed). No intermediate results are shared between threads. No synchronization is required.

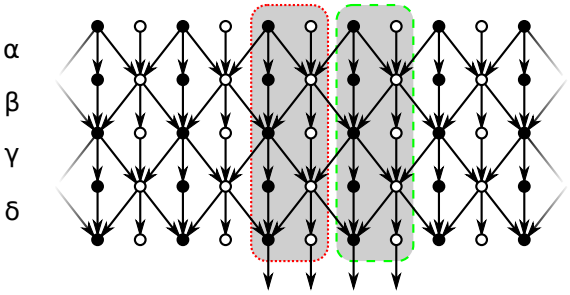


Figure 6.2: Lifting scheme of CDF 9/7 wavelet showing the calculation performed by a single thread (dotted and dashed). Intermediate results are shared between neighbouring threads. Synchronization is required.

the price that some calculations become redundant. In all cases, the coalesced memory access was used wherever it was possible.

6.3.1 Horizontal Pass

At the beginning, we have focused on an algorithm for computation of the horizontal pass. We have implemented and compared plenty of existing algorithms. The most prominent of these algorithms are presented below. All of the implemented algorithms use up to 256 threads for each work group.

Considering the horizontal pass, it can generally consist of the following steps. Firstly, transfer a data row from the global memory to the local memory. Secondly, perform the horizontal transform using data in the shared memory. Thirdly, transfer the computed row of result from the shared memory to the global one. Note that the local memory is shared for the group of threads. An access to this local memory is much faster, but it is limited by relatively small size and it is shared just along one single work group.

Considering the first relevant algorithm, each thread in the work group computes a pair of the output coefficients by the convolution scheme. Each thread loads nine coefficients from the local memory and computes a corresponding pair of resulting coefficients (responses to the FIR filters). The implementation was earlier described in detail in [39, 40]. We refer to as *Horiz-Galiano2011* in this paper.

The second algorithm uses lifting scheme instead of convolution. In this case, every thread loads nine coefficients from local memory and computes all the required computation by itself. No intermediate results are shared between threads. This implementation was described in [15]. We will further denote it as *Horiz-Blazewicz2012-1*. The data-flow graph for a single thread is shown in Fig. 6.1.

The other algorithm computes 4 pairs of the output coefficients instead of one by each thread. This algorithm employs the lifting scheme which was described in [15]. It is further denoted as *Horiz-Blazewicz2012-4*. The algorithm does not share the intermediate results between threads and does not require synchronization barriers. The implementation requires to load 15 coefficients from the local memory. A group of threads loads and process a single row of the input image.

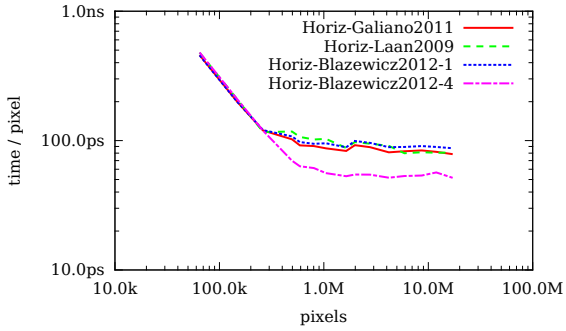


Figure 6.3: GeForce GTX 580. Horizontal pass algorithms.

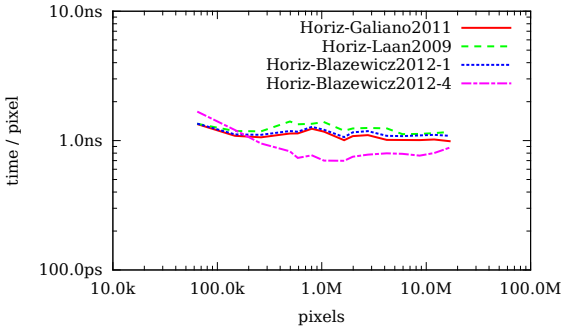


Figure 6.4: NVS 4200M. Horizontal pass algorithms.

algorithm	1 Mpel	10 Mpel
Horiz-Galiano2011	87.1	81.9
Horiz-Laan2009	103.0	80.6
Horiz-Blazewicz2012-1	95.4	89.3
Horiz-Blazewicz2012-4	**56.0**	**56.7**

Table 6.2: GTX 580. Horizontal pass. Picoseconds per pixel.

algorithm	1 Mpel	10 Mpel
Horiz-Galiano2011	1173.8	1020.2
Horiz-Laan2009	1389.9	1142.2
Horiz-Blazewicz2012-1	1212.2	1108.4
Horiz-Blazewicz2012-4	**700.7**	**800.8**

Table 6.3: NVS 4200M. Horiz. pass. Picoseconds per pixel.

The last of the implemented algorithms uses the lifting scheme. The algorithm was described in [52, 53] and it is referred to as *Horiz-Laan2009* here. There is no redundant computation considering different threads. However, this implementation requires additional synchronizations in the lifting steps. The algorithm works in the following way. Each of the threads loads three input coefficients from the local memory and performs elementary lifting step. Then, neighbouring threads exchange the intermediate results. These two steps are repeated further. The data-flow graph for a single thread is shown in Fig. 6.2.

The previously described algorithms are limited to a certain resolution of the input image. One can compute the transform of the input image up to 512 pixels wide if the maximum number of threads in the work group is 256 and if two output coefficients are computed by a single thread. To overcome this limitation, the algorithms are extended in the following way. A group of threads processes the left-most coefficients in a row from the global memory, computes horizontal transform and then saves results into the global memory. The same group processes a following block of coefficients, where previously loaded and processed coefficients are reused. This approach reduces a global memory access and some of the redundancy.

We have implemented and evaluated all of the algorithms described above. The results of the comparison are plotted in Fig. 6.3 and 6.4. Several measurements are listed in the Table 6.2 and 6.3. The *Blazewicz2012-4* algorithm proved to be the fastest one across a whole range of image sizes. This result is caused mainly by maximizing a number of arithmetic operations by using each thread as pointed in [15].

6.3.2 Vertical Pass

In this part, we focus on the vertical pass of the transform. The simplest approach is to use same algorithms that are used in the horizontal pass but transform columns instead of

rows. However, such an approach does not consider coalescent access to the global memory that consequently causes a performance degradation. W. J. Laan *et al.* [53] states that this approach is 10× slower than more complex solution, that will be described later.

A more complex approach was presented in [15]. Initially, this approach transposes input data. After that, the unchanged horizontal pass algorithm is performed. Finally, the resulting data are transposed again. We have implemented this algorithm in the following way. The horizontal pass *Blazewicz2012-4* algorithm is used in the heart of the algorithm. We refer this approach to as *Vert-Blazewicz2012*. This process uses fast coalescent access to the global memory and creates lot of working groups that help to utilize computing resources. On the other hand, every transposition requires a separate kernel run. This causes access to global memory for 6−3× for reading and 3× for writing per every element of the data.

Furthermore, we have adopted a vertical transform algorithm presented in [52]. This original algorithm is denoted as *Vert-Laan2009*. The algorithm divides the input image to multiple vertical strips. A width of the strip is based on the size of the successive bytes defined by coalescent memory access. We use width of the strip of 32 coefficients. Every strip is processed by a single work-group of threads. Inside of such a strip, a sliding window approach is used. The width of the sliding window is same as the width of the strip (32 coefficients), the height of the window is 20 coefficients. The algorithm works as follows:

1. The window is placed on the top of the strip, 17 rows are copied from the global memory to the local one. The rows in the window are processed by lifting scheme, where result values are in first 13 rows and 4 rows contain intermediate results. The result values are copied into the global memory, intermediate results stay in the local one.

2. The sliding window is moved by 16 rows down. Missing rows (not in the local memory) are loaded from the global memory.

3. In the window, the lifting scheme is performed .

4. The results are moved to the global memory, the rows with intermediate results are still in the local memory.

5. This process is repeated until the window reach the border of the strip. The last remaining section is processed by a similar process like previous one.

The above described approach performs the lifting computation using barriers after each lifting step. This algorithm is similar to the horizontal pass *Horiz-Laan2009* described before. On the plus side, this approach reduces access to the global memory just for one read and one write per each element. On the negative side, the algorithm creates a small count of work-groups, which can be problematic at the modern GPU with many of multiprocessors.

Furthermore, we have experimented with a different adaptation of this algorithm. As a result, we have created a faster adaptation. The core of our proposed approach is the same as the previously described algorithm presented in [52]. As in the previous case, we have used the sliding window method to process entire tile by a single work-group. The *Horiz-Blazewicz2012-4* algorithm proved to be the fastest one considering the horizontal pass. Therefore, we have adapted this approach to perform the lifting scheme in the vertical direction. This approach requires extension of the sliding window height to process 8 coefficients by a single thread in one particular window position. Consequently, the approach

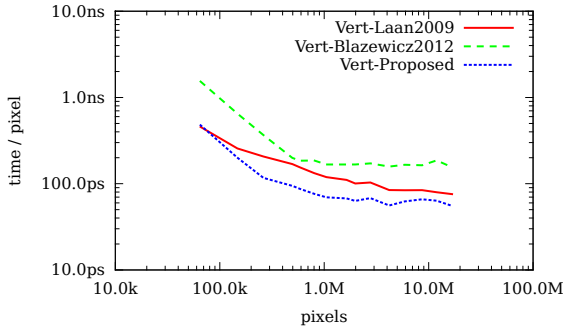


Figure 6.5: GeForce GTX 580. Vertical pass algorithms.

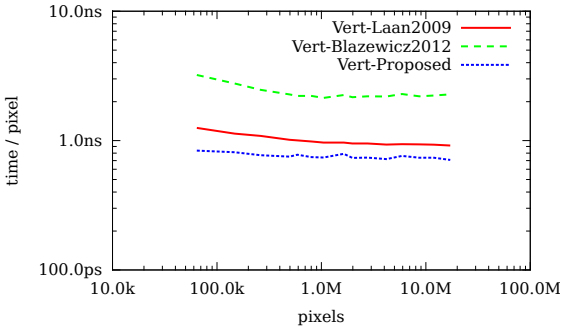


Figure 6.6: NVS 4200M. Vertical pass algorithms.

algorithm	1 Mpel	10 Mpel
Vert-Laan2009	119.3	79.6
Vert-Blazewicz2012	167.0	186.5
Vert-Proposed	**69.6**	**63.5**

Table 6.4: GTX 580. Vertical pass. Picoseconds per pixel.

algorithm	1 Mpel	10 Mpel
Vert-Laan2009	965.3	930.2
Vert-Blazewicz2012	2139.0	2231.7
Vert-Proposed	**739.4**	**737.9**

Table 6.5: NVS 4200M. Vert. pass. Picoseconds per pixel.

requires the sliding window of 71 rows height. No intermediate results are passed between lifting steps nor different window positions. In the first window position (on the top of strip), 64 rows are computed. These values are computed by the same scheme as the one used in *Horiz-Blazewicz2012-4*. It is required to load 67 rows from global memory to the local one (the window). After processing and moving the results to the global memory, the window is moved by 64 rows down to process additional 64 rows. It is required to have 71 rows in the window to perform the transform correctly. The most bottom part of the strip is processed by separate algorithm to process borders correctly. This approach eliminates part of the synchronisations at the cost of adding some redundant arithmetic operations and increasing the local memory consumption. We refer this adaptation such as *Vert-Proposed*.

Finally, we have evaluated all of the vertical pass algorithms described in this section. The results are plotted in Fig. 6.5 and 6.6. Several measurements are listed in the Table 6.4 and 6.5. In all cases, medians of ten measurements are used. The proposed *Vert-Proposed* algorithm has proved to be the fastest one. This algorithm achieved an average speed-up at least 30 % compared to the *Vert-Laan2009* algorithm which is considered to be the state-of-the-art method. The average speedup of *Vert-Proposed* relative to the *Vert-Laan2009* implementation are shown in Table 6.6.

graphics card	avg. speed-up
GeForce GTX 580	50 %
NVS 4200M	31 %

Table 6.6: Vertical pass. Average percentage speedups.

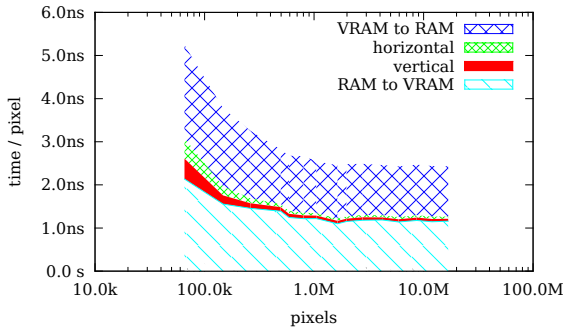


Figure 6.7: GeForce GTX 580. Entire transform.

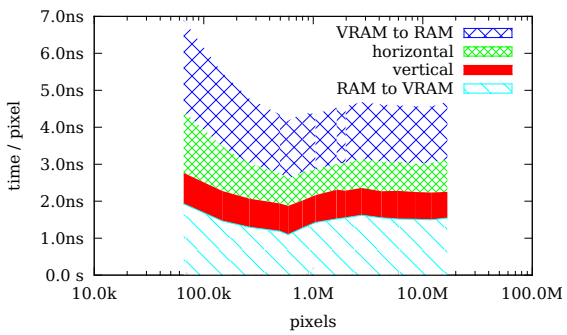


Figure 6.8: NVS 4200M. Entire transform.

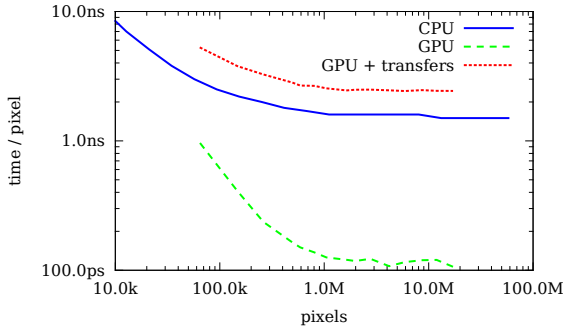


Figure 6.9: GeForce GTX 580. Entire transform compared to CPU.

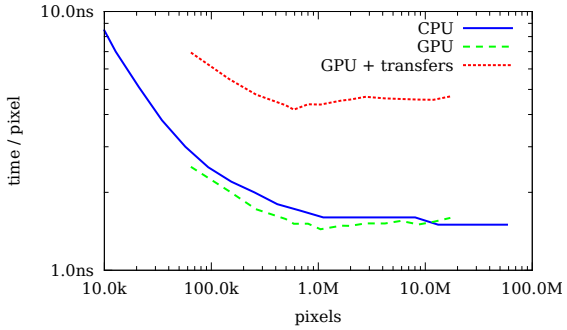


Figure 6.10: NVS 4200M. Entire transform compared to CPU.

6.3.3 Entire Transform

Finally, we decided to evaluate the full transform computation. It may generally consists of the two memory transfers (on-board memory ↔ video memory) and two transform passes (the horizontal and the vertical one). It would not be entirely fair to include the memory transfers in the final comparison. For this reason, the cumulative flow diagram is shown here separating the memory transfers and the two transform passes. The final comparison is plotted in Fig. 6.7 and 6.8.

It would be interesting to compare the GPU implementations described in this section with a tuned CPU counterpart. For this purpose, the state of the art CPU implementation [20] with fused vertical and horizontal passes was used. This implementation utilizes SIMD instructions and have 4 threads running simultaneously. The implementation was evaluated using mainstream PC with Intel x86 CPU. Specifically, Intel Core2 Quad Q9000 running at 2.0 GHz was used. This CPU has 32 kiB of level 1 data cache and 3 MiB of level 2 shared cache (two cores share one cache unit). The comparison is summarized in Fig. 6.9 and 6.10.

Note that both GPUs are equipped with an older version (2.0) of PCI-Express ×16 bus. Due to this fact newer GPUs with current version (3.0) of PCI-Express ×16 bus may have 2× faster VRAM to RAM and RAM to VRAM transfers.

6.4 Conclusion

We have presented a novel approach to 2-D wavelet transform using GPU reaching an average speedup at least 30 % on tested graphics cards. This approach is focused on utilization of parallel capabilities of modern GPUs. All the methods compared in this paper were evaluated using GeForce GTX 580 and NVS 4200M cards. In addition, we have compared these methods with state of the art implementation on CPU.

In more detail, the computation of single level of the transform is split into horizontal and vertical passes. Initially, we have adapted an existing algorithm to perform the horizontal pass without synchronizations. Furthermore, we have incorporated this algorithm into other existing technique performing the vertical pass using the sliding window. Additionally, we have extended this sliding window height to process 8 coefficients by a single thread in one particular window position. This step helps to reduce the computing redundancy. Finally, we have evaluated the performance of the entire transform and also compared it with transform performed using CPU.

Further work could focus on multi-level decompositions, improvement of the proposed vertical algorithms in order to utilize more work-groups or an exploration of fusion of the horizontal and vertical passes into a single one.

Acknowledgements This work has been supported by the IT4Innovations Centre of Excellence (no. CZ.1.05/1.1.00/02.0070) and the TACR project V3C (no. TE01020415).

Chapter 7

Article: Block-based Approach to 2-D Wavelet Transform on GPUs

In contrast to the previous paper, which focuses on the separable strategy for 2D DWT calculation, this one deals with the block-based strategy (see the section 3.2.2 for more information) and incorporates Region Mapping (see the section 3.3 for more information). This approach maps work-group threads to image regions where each thread is mapped onto a single 2×2 shaped pixel quadruple. For correct calculation of seamless 2D DWT, the regions mapped to work-groups have to have overlapping areas, causing redundant calculations as well as in the previous paper. The number of such calculations depends on the shape and size of the work-group, the number of predict-update lifting pairs (K) and the degree of calculated wavelet (D). Detailed information regarding the work-group to image region mapping is described in the section 5.3.

For the calculation of the seamless 2D DWT using the block-based strategy, the two schemes are proposed. One of them is the separable lifting scheme. The second one is a novel non-separable 2D scheme denoted as Kula2016 that reduces the number of barriers, including implicit one to $2K$ from $4K$ used on separable lifting scheme. As a consequence, the number of arithmetic operations increased from 16 to 20. All schemes are denoted in Table 11.1. Note that Table 11.1 incorporates only evolved versions of the non-separable scheme with the reduced number of operations from 20 to 18 so-called non-separable lifting* scheme. The evolved scheme is introduced in the following chapter 8.

Finally, the proposed schemes using block-based strategy are compared to each other and to the best performing separable strategy approach based on the pipeline approach that is introduced in the section 6.3. A comparison of the implemented seamless 2D DWT approach based on block-based strategy to the state-of-the-art one based on separable strategy clearly shows that approaches based on block-based strategy outperform the throughput of separable strategy approaches by 60-100% on the tested platforms. Additionally, the novel non-separable scheme outperforms the throughput of the separable lifting one on reasonable resolutions by 5% on average on the tested platforms. On resolutions $>$ 15MPix, a global memory throughput on the tested AMD GCN architecture became the limitation of the speed of 2D DWT that leads to the same performance of the separable lifting scheme.

The approaches are implemented using the Wavelet transform framework presented in the chapter 5 and use the macro-based version of the kernel generator described in the section 5.2.

Block-based Approach to 2-D Wavelet Transform on GPUs

Kula, M., Bařina, D. and Zemčík, P. Block-based Approach to 2-D Wavelet Transform on GPUs. In: *Information Technology: New Generations*. Springer International Publishing, 2016, vol. 448, p. 643–653. Advances in Intelligent Systems and Computing. DOI: 10.1007/978-3-319-32467-8_56. ISBN 978-3-319-32467-8

Author contribution: 50%

Abstract

This paper introduces a new approach to computation of 2-D discrete wavelet transform on modern GPUs. The proposed approach involves block-based processing enabling one seamless transform even for high resolution input data. Inside the blocks, two distinct methods can be used – either separable or non-separable 2-D lifting scheme. Furthermore, the paper presents a comparison of the proposed approach under different conditions to the best existing methods, whereas our approach consistently outperforms the other ones. Our methods are implemented using the OpenCL framework and tested on a wide range GPUs.

7.1 Introduction

The 2-D discrete wavelet transform (DWT) is the signal-processing transform suitable for decomposition of the analysed 2-D signal into several scales. On each scale, three directional subbands are formed. These are usually referred to as HL, LH, and HH subbands. The 2-D transform is defined as separable product of 1-D transforms performed sequentially on rows and columns (or vice versa). Each of these one-dimensional transforms can be computed through either the convolution or the lifting scheme. Different strategies of 2-D DWT implementation were developed for various computational platforms.

In this paper, we focus on implementation of DWT using modern graphics cards (GPU) capable of a general-purpose computing. In these architectures, the GPU contains thousands of stream processors that are clustered into blocks. All processors in each block execute the same instruction with different operands at one time. The blocks are grouped into multiprocessors which form the basic functional units of the GPUs. The thread scheduler allocates as many work groups to multiprocessors as their resources allow. The work groups are defined as a group of threads that can interoperate with each other using the local memory and memory barriers. Thus, the resources, such as the local memory size, should be minimized. The allocated work groups created by OpenCL framework is then di-

vided into warps (hardware blocks with 32 threads). Execution instructions of these warps on blocks of processors are provided using warp schedulers dynamically. Global memory accesses in warp should be coalesced. Otherwise, additional memory operations are executed. The local memory is organized into banks. Access to the same banks from warp causes serialization. This issue is referred to as a bank conflict. The serialization of local memory operations and uncoalesced global memory access can cause a performance degradation.

This paper is further focused on the OpenCL framework.[1] OpenCL is a framework for general-purpose parallel programming across multiple device types. In this framework, a platform independent executable program is called the kernel. The kernel is executed on required number of threads that identify their data and control flow by their N-dimensional indices. These threads are organized into work groups with identical user-defined number of threads. The threads in such a group can cooperate with each other through local memory and barriers.

Several methods for the 2-D DWT computation using GPU have been published in the last decade. For example, the simplest row–column methods transform the whole image at once. Usually, the transposition is needed between the horizontal and vertical part. Furthermore, the block-based methods transform the image using smaller blocks utilizing the row-column method inside. Unfortunately, such a method results in several independent transforms instead a single seamless one. Finally, the pipelined methods transform the image using column strips while employing the sliding window on them.

In this paper, we propose two novel block-based methods computing the seamless 2-D transform. The first of them employs the separable (row–column) lifting scheme inside the overlapping blocks. The second uses a non-separable lifting scheme recently proposed. Both of the proposed methods consistently outperform the existing methods.

The rest of the paper is organized as follows. The Related Work section summarizes the state of the art, especially existing GPU implementations. The heart of our work is presented in Block-Based Approach section. First, we propose the separable transform. Further in the text, the non-separable method is discussed. Finally, Conclusions section summarizes the paper and outlines the future work.

7.2 Related Work

This section takes a closer look at the discrete wavelet transform and revises the state of the art of its implementation on contemporary graphics cards.

The DWT can be understood as a transform suitable for decomposition of a signal into low-pass and high-pass frequency components. Usually, such a decomposition is performed at several scales resulting in a multi-scale signal representation. At this point, we are considering one-dimensional signals. The transform of 2-D signals is computed through the tensor product of these 1-D transforms. For various requirements, different strategies of 2-D transform computation emerged. Going back to 1-D transform, as the discrete wavelet transform is a linear one, the decomposition into the low-pass and high-pass components can be performed through a convolution scheme with two filters. However, the more efficient computational scheme according to the number of arithmetic operations exists. This scheme is referred to as the lifting scheme. Additionally, using this scheme, the whole signal can be transformed in-place. Specifically, any discrete wavelet transform can be factored into a finite sequence of lifting steps. These steps alternately update odd and even intermediate

[1] http://www.fit.vutbr.cz/research/prod/index.php?id=434.

results using short FIR (finite impulse response) filters. When evaluating this scheme, intermediate results can be appropriately shared between neighbouring coefficients.

The discrete wavelet transform is often used as a basis for sophisticated compression algorithms. This paper focuses on a popular CDF (Cohen-Daubechies-Feauveau) 9/7 wavelet. This wavelet is used, e.g., in JPEG 2000 image compression standard. In [31], Daubechies and Sweldens factored CDF 9/7 wavelet into four successive lifting steps, employing short symmetric two-taps FIR filters. A data-flow diagram of the factorization (without scaling) is depicted in Fig. 7.1, where, the $\alpha, \beta, \gamma, \delta$ are real constants specific to CDF 9/7 transform. Formally, the forward transform in Fig. 7.1 can be expressed by the dual polyphase matrix

$$\tilde{P}(z) = \begin{bmatrix} 1 & \alpha\left(1+z^{-1}\right) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \beta\left(1+z\right) & 1 \end{bmatrix} \begin{bmatrix} 1 & \gamma\left(1+z^{-1}\right) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \delta\left(1+z\right) & 1 \end{bmatrix}. \tag{7.1}$$

In this paper, we consider the lifting scheme only, as it is usually a better alternative. A detailed comparison of the convolution and lifting schemes on GPUs was addressed, e.g., in [74] and [75].

The implementation of this transform was comprehensively studied on various platforms including the modern GPUs. Considering this scenario, the input image has to be initially transferred from main memory into memory on the graphics card. Similarly, the resulting coefficients could be transferred back. Having the input 2-D image in the GPU global memory, different strategies of 2-D DWT implementation can be used. These strategies can be divided into three groups – row–column, block-based, and pipelined methods.

The row–column method applied on the entire 2-D image was used for instance in [74], [75], [36], [15], [39], [40]. In [36] and [15], data transposition was performed in between the horizontal and vertical series of 1-D transforms. In [74] and [75], Tenllado *et al.* adapted the discrete wavelet transform on GPU fragment shaders. As this paper is focused on the OpenCL framework, we will not discuss their paper in more details. The other cited papers are focused on the CUDA architecture. In [36], the convolution scheme is applied on each row. Then, the image matrix is transposed and the convolutions are applied on each column. Finally, the image is transposed back. In [39] and [40], V. Galiano *et al.* compared several CUDA implementations of DWT. They used the CDF 9/7 wavelet and convolution-based algorithm on entire rows/columns. Their fastest implementation uses the coalesced memory access.

In [15], the authors calculate the wavelet transform through 4 kernels. The first kernel performs an image transposition using work groups of size 16×16 threads, where the thread processes one image element. To ensure coalesced global memory access, the transposition in the shared memory is used rather than directly in the global memory. In the second kernel, the vertical wavelet transform is performed as follows. Each thread loads its elements from the global memory and stores them into the shared memory. Then, the adjacent elements, that are required for the computation of the output coefficient, are loaded from the shared memory into registers. The threads compute their output coefficients using 4 steps of the wavelet scheme independently to each other (with no synchronization). When the computation is finished, the output coefficients are written back to the global memory. The third and the fourth kernels calculate the image transposition and the horizontal wavelet transform in the same way as the first two kernels. The calculations that are performed by a single thread using the approach described can be seen in Fig. 7.1(b) and Fig. 7.1(c).

The pipelined approach was used in [52] and [53]. In [52], Laan *et al.* accelerated the Dirac video codec using the CUDA platform. In [53], the authors provided a detailed analysis of the DWT implementation using the lifting scheme on the CUDA platform. They

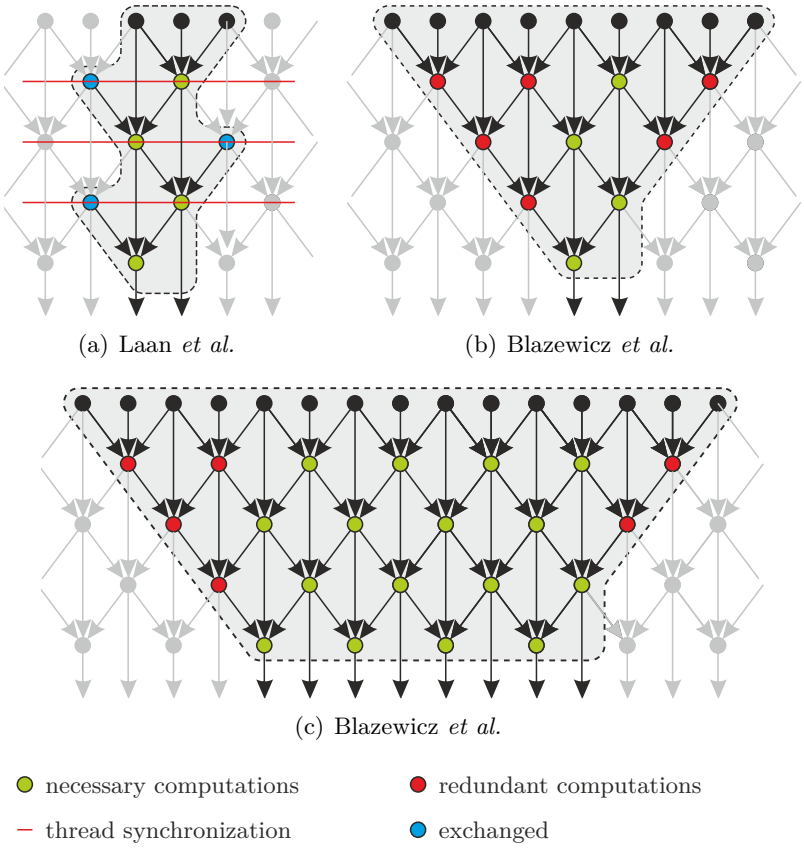


Figure 7.1: A portion of the data-flow graph attributable to individual threads. The method of (a) Laan and two methods used by Blazewicz – with (b) one, and (c) four pairs.

focused on 2-D and 3-D methods of DWT implementation using several wavelets including CDF 9/7. In the horizontal part of their transform, each work group is mapped to a single image row. Each thread computes one coefficient per a single step and shares it with other threads. Because of non-atomic instructions issued in whole group, memory barrier is needed in between each two steps. See Fig. 7.1(a). The vertical part of their transform maps each work group to multiple vertical strips with a width that ensures coalesced global memory accesses and bank-conflict-free shared memory transfers.

Another row-column approach was used in [48]. The horizontal transform is computed in the same way as Blazewicz *et al.* did. The vertical transform is computed using 32 coefficients wide strips per work group like in Laan's implementation. The difference between the Laan's and Kucis's vertical methods lies in processing assigned to a single thread. Kucis *et al.* used rotated Blazewicz's approach with 2 pairs per thread mapping. Moreover, Kucis *et al.* also demonstrated that their approach outperforms Laan's and Blazewicz's ones. The approach of Kucis is used as a reference approach and labeled as *Kucis2014*.

The approaches in [57] as well as [15] are focused on the lifting scheme. Their implementations split the image into small tiles and perform several independent transforms on each of them. Thus, they performed several independent transforms (introducing a block effect) which is different and much easier task comparing to what we are dealing with in this paper.

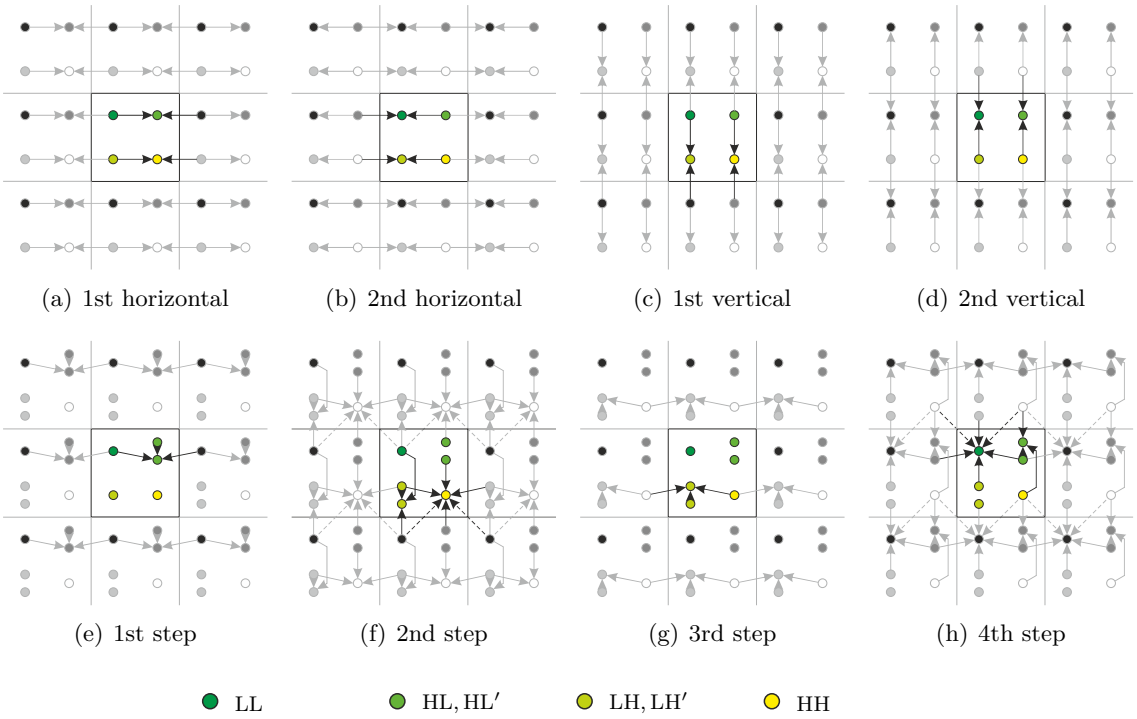


Figure 7.2: The separable (top) and non-separable block-based approach (bottom).

As it can be seen, the problem of the efficient 2-D discrete wavelet transform implementation on conventional GPUs was fairly well studied. However, we see several gaps which can allow for additional speedups. Specifically, only the separable 2-D schemes were examined so far. These schemes require to pass the results through the global memory, while causing unnecessary memory traffic.

7.3 Block-Based Approach

The heart of our work is presented in this section. At the beginning, we propose the separable block-based method. Afterwards, we discuss the block-based method utilizing non-separable lifting scheme recently proposed. The performance comparison of the proposed methods is shown in Fig. 7.4. As it can be seen, the block-based methods perform consistently faster compared to the best of the existing methods. Our implementation is based on the OpenCL framework. All of the algorithms are evaluated using AMD R9 290X and NVIDIA TitanX graphics cards. The main benefit of the block-based methods is the reduction of memory access count as the data is read as well as written only once.

7.3.1 Separable Method

Except for the sliding window, our separable block-based approach uses the same scheme as the Laan's method. The threads in each work group are responsible for processing of 2×2 input coefficients. At the beginning, the thread loads their coefficients from the global memory and stores them into separate shared memory locations. The computation is briefly illustrated in Fig. 7.2. In the first step of the horizontal pass, each of the threads computes

the LH coefficient using two LL coefficients of the thread itself and the thread on the right. Additionally, the HH coefficient is computed using HL coefficients of the current thread and the thread on the right. In the second step, the computation of LL and HL coefficients is performed in the same way as the computation of the LH and HH coefficients. After that, these two steps are repeated with a substitution of α and β with γ and δ coefficients.

The vertical steps are performed in the same way as the horizontal steps except for a rotation of the scheme by 90 degrees. Unlike the horizontal pass, synchronization using the memory barrier is required between the steps. Horizontal steps are synchronization-free thanks to the atomicity of hardware instructions. Fig. 7.2 shows individual steps of the underlying data-flow graph.

7.3.2 Non-Separable Method

In [47], the authors derived a non-separable 2-D lifting scheme for CDF 5/3 and subsequently CDF 9/7 transforms. As initial step of CDF 5/3 transform, the input signal is split into quadruples $(\mathrm{LL}, \mathrm{HL}, \mathrm{LH}, \mathrm{HH})$. Then, lifting steps leading to the calculation HH coefficients are performed. This is followed by parallel computation of the HL and LH coefficients. In the third step, the LL coefficient is updated. Finally, the coefficients can be scaled. The scheme for CDF 9/7 comprises two these connected transforms.

Motivated by the work of Iwahashi *et al.* [47], we have reorganized the elementary lifting FIR filters in order to obtain a highly parallelizable scheme suitable for the modern GPUs. The main purpose of this modification is to minimize the number of memory barriers that slow down the calculation. As a result, we get several non-separable two-dimensional FIR filters. For their description, we employ the well known z-transform notation. The transfer function of the two-dimensional FIR filter $x(k_m, k_n)$ is defined as

$$X(z_m, z_n) = \sum_{k_m=-\infty}^{\infty} \sum_{k_n=-\infty}^{\infty} x(k_m, k_n)\, z_m^{-k_m} z_n^{-k_n}, \tag{7.2}$$

where m refers to the horizontal axis and n to the vertical one. Moreover, to keep consistency with [47], the $H^*(z_m, z_n) = H(z_n, z_m)$ denotes a filter transposed to the $H(z_m, z_n)$. Furthermore, the $\overline{H}(z_m, z_n) = H(z_n^{-1}, z_m^{-1})$ denotes a filter reversed along the m- as well as n-axis. Coupled together, the $\overline{H}^*(z_m, z_n)$ denotes a transposed and reversed filter to the original $H(z_m, z_n)$. The scheme we formed is composed of three elementary filters F, G, H given by

$$\begin{bmatrix} \mathrm{F_a} \\ \mathrm{G_a} \\ \mathrm{H_a} \end{bmatrix} = \begin{bmatrix} F_a(z_m, z_n) \\ G_a(z_m, z_n) \\ H_a(z_m, z_n) \end{bmatrix} = a \begin{bmatrix} 1 \\ z_n \\ 1 + z_m \end{bmatrix}, \tag{7.3}$$

where a denotes a filter parameter. The filters above are assembled into more complex operations. Our scheme consists of two halves between which a memory barrier is placed. The first half of the scheme uses the following filters. Similarly, the second half uses these filters in the reverse orientation. Due to the limited place, we have made a small abuse of notation. Instead of the full notation $H(z_m, z_n)$, we only use a shortened labeling, such as H.

$$\begin{bmatrix} \mathrm{F_a} \\ \mathrm{G_a} \\ \mathrm{H_a} \\ \mathrm{H_a^*} \\ \mathrm{G_aH_a} \end{bmatrix} = \begin{bmatrix} a \\ a\,z_n \\ a\,(1+z_m) \\ a\,(1+z_n) \\ a^2\,(z_n + z_m z_n) \end{bmatrix}, \qquad \begin{bmatrix} \overline{\mathrm{F}}_\mathrm{a} \\ \overline{\mathrm{G}}_\mathrm{a} \\ \overline{\mathrm{H}}_\mathrm{a} \\ \overline{\mathrm{H}}_\mathrm{a}^* \\ \overline{\mathrm{G}}_\mathrm{a}\overline{\mathrm{H}}_\mathrm{a} \end{bmatrix} = \begin{bmatrix} a \\ a\,z_n^{-1} \\ a\,(1+z_m^{-1}) \\ a\,(1+z_n^{-1}) \\ a^2\,(z_n^{-1} + z_m^{-1} z_n^{-1}) \end{bmatrix} \tag{7.4}$$

Finally, our scheme is composed of four steps referred to as S^1 to S^4. Between the second S^2 and the third S^3 step, the memory barrier must be inserted in order to properly exchange intermediate results. Additionally, our scheme requires the induction of two auxiliary variables per each quadruple of coefficients LL, HL, LH, and HH. These are denoted as $\mathrm{HL}', \mathrm{LH}'$. This is valid regardless of their initial as well as final values. The scheme

$$\mathbf{y} = S_\beta^4 S_\beta^3 S_\alpha^2 S_\alpha^1 \mathbf{x} \tag{7.5}$$

describes the relation between input $\mathbf{x}$ and output $\mathbf{y}$ vectors

$$\begin{bmatrix} \mathrm{LL} & \mathrm{HL} & \mathrm{LH} & \mathrm{HH} & \mathrm{HL}' & \mathrm{LH}' \end{bmatrix}^T. \tag{7.6}$$

Each single thread of the work group is responsible of one such a vector.

Regarding this notation, the individual steps are defined as follows. For better understanding, the signal-processing block diagram of this scheme is shown in Fig. 7.3. In addition, the operations are graphically illustrated in Fig. 7.2.

$$S_\alpha^1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \mathrm{H}_\alpha & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.7}$$

$$S_\alpha^2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \mathrm{G}_\alpha\mathrm{H}_\alpha & \mathrm{G}_\alpha & \mathrm{H}_\alpha & 1 & \mathrm{F}_\alpha & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ \mathrm{H}_\alpha^* & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{7.8}$$

$$S_\beta^3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \overline{\mathrm{H}}_\beta & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.9}$$

$$S_\beta^4 = \begin{bmatrix} 1 & 0 & \overline{\mathrm{F}}_\beta & \overline{\mathrm{G}}_\beta\overline{\mathrm{H}}_\beta & \overline{\mathrm{H}}_\beta & \overline{\mathrm{G}}_\beta \\ 0 & 0 & 0 & \overline{\mathrm{H}}_\beta^* & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.10}$$

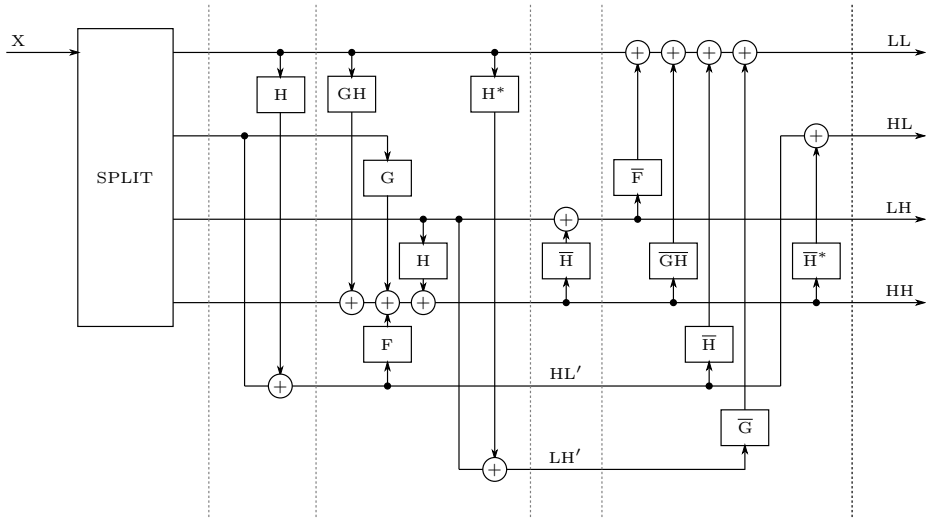
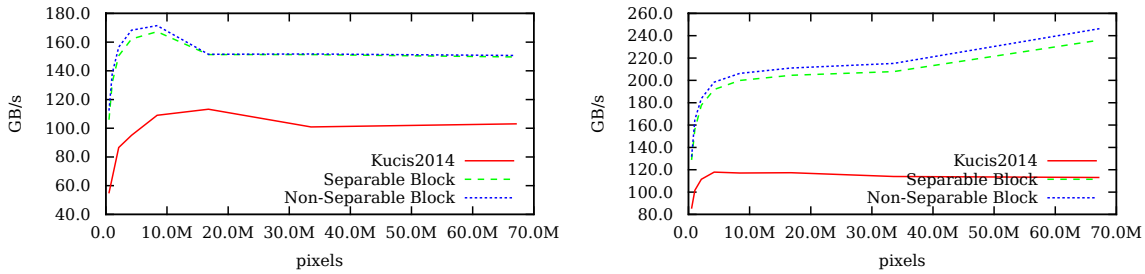


Figure 7.3: Block diagram of the proposed non-separable scheme.



(a) AMD R9 290X

(b) NVIDIA TitanX

Figure 7.4: Throughput performance. *Kucis2014* is the reference state-of-the-art method.

Compared to [47], the total number of arithmetic operations has been reduced form 24 to 20. The calculation of CDF 9/7 transform comprises two of these connected transforms (the first one with α, β, the second with γ, δ) between them another barrier is placed. In total, the calculation contains 3 memory barriers.

7.4 Conclusions

We have presented two novel block-based approaches to 2-D wavelet transform using modern GPUs. These approaches can handle high resolution images while producing the seamless transform. Both of the proposed methods consistently outperform the existing methods with all tested GPUs.

The first presented approach utilizes classical separable 2-D lifting scheme. Whereas the second approach employs a novel 2-D non-separable scheme. Considering the second one, we have minimized the number of memory barriers. Moreover, as compared to the existing non-separable scheme, the total number of arithmetic operations has been reduced form 24 to 20.

The future work includes behavior of the proposed methods under a multi-scale decomposition. Another direction of our research may include a connection with some practical application (e.g., JPEG 2000 scheme).

Acknowledgement

This work has been supported by the TACR Competence Centres project V3C – Visual Computing Competence Center (no. TE01020415).

Chapter 8

Article: Parallel Wavelet Schemes for Images

This paper extends the research from paper [49] that further improved the novel non-separable scheme (denoted as Kula2016) from the paper presented the previous chapter 7 by reducing the number of operations from 20 to 18 and compares it with the existing scheme invented by Iwahashi [47] (labeled in the thesis as implosion). In the current paper, this scheme is further changed by separating in-quadruple operations using optimization approach and is denoted as Monolithic* (later labeled as non-separable lifting* scheme) and compared with implosion scheme as well. For the sake of shortness, that paper is skipped.

This paper introduces a novel non-separable scheme called explosion (labeled in the paper as Explosive) that has the same number of barriers and operations as the implosion scheme. Its advantage lies in the reduction of local memory store operations from $6K$ to $4K$ and reduction of local memory consumption by 33% while using single buffering approach or 25% using double buffering approach (for more information about Region Mapping and buffering see the section 5.5). Mapping of the work-group and threads to tiles remains the same as in the previous paper.

Also a novel approach that separates in-quadruple predict/update operations before the first barrier/after the last step is introduced. This approach can be applied on all separable or non-separable schemes and can significantly reduce the number of operations (with the exception of the separable lifting$^+$ scheme that has the minimum number of operations already).

Finally, all existing schemes and their improved variants are compared and tested on CDF 5/3 ($K = 1$, $D = 1$), CDF 9/7 ($K = 2$, $D = 1$) and DD 13/7 ($K = 1$, $D = 3$) wavelets on various GPUs. All of the tested schemes and their properties are described in Table 11.1. The evaluation shows that the optimization approach increases the wavelet throughput by 80% on non-separable polyconvolution and by 5-20% on the rest of the schemes, except for the separable lifting$^+$ one, where it is not beneficial on the tested platforms. The novel non-separable lifting scheme, combined with the optimization approach, outperforms state-of-the-art schemes used by other authors (implosion and separable lifting$^+$) by 30% on the tested platforms. Our later experiments show that another novel scheme, the explosion scheme, combined with the optimization approach, is the best performing scheme for most Intel GPUs.

The implementation uses Wavelet Transform Framework and second version of kernel generator described in the section 5.2.

Parallel Wavelet Schemes for Images

Bařina, D., Kula, M. and Zemčík, P. Parallel wavelet schemes for images: How to make the wavelet transform friendly to parallel architectures. *Journal of Real-Time Image Processing*. Springer Science and Business Media LLC. 2019, vol. 16, no. 5, p. 1365–1381. DOI: 10.1007/s11554-016-0646-3. ISSN 1861-8200

Author contribution: 40%

Abstract

In this paper, we introduce several new schemes for calculation of discrete wavelet transforms of images. These schemes reduce the number of steps and, as a consequence, allow to reduce the number of synchronizations on parallel architectures. As an additional useful property, the proposed schemes can reduce also the number of arithmetic operations. The schemes are primarily demonstrated on CDF 5/3 and CDF 9/7 wavelets employed in JPEG 2000 image compression standard. However, the presented method is general, and it can be applied on any wavelet transform. As a result, our scheme requires only two memory barriers for 2-D CDF 5/3 transform compared to four barriers in the original separable form or three barriers in the non-separable scheme recently published. Our reasoning is supported by exhaustive experiments on high-end graphics cards.

8.1 Introduction

The two-dimensional discrete wavelet transform (DWT) is a signal-processing transform suitable as a basis for sophisticated compression algorithms. For example, JPEG 2000, an image coding system, is based on such compression technique. This paper focuses on the Cohen–Daubechies–Feauveau (CDF) 5/3 and 9/7 wavelets [30], which are often used for image compression. However, the methods are general, and they are not limited to any specific type of transform. Of course, plenty of other applications are built over the discrete wavelet transform.

The one-dimensional discrete wavelet transform has undergone a gradual development in the last few decades. Probably, the most important advance is the discovery of a factoring algorithm [31] referred to as the lifting scheme. In this context, the discrete wavelet transform or two-band subband filtering can be represented by a polyphase matrix. The lifting scheme algorithm decomposes any wavelet transform with finite filters into a finite sequence of lifting steps, while reducing the number of arithmetic operations. The de-

composition corresponds to a factorization of the polyphase matrix filters into elementary matrices. The resulting coefficients of 1-D transform are formed in two subbands. The subbands correspond to low-pass (L) and high-pass (H) filtered subsampled variants of the original signal.

In case of two-dimensional transform [55], one level of the transform can be realized using a separable decomposition scheme. In this scheme, the coefficients are evaluated by successive horizontal and vertical 1-D filtering, resulting in four disjoint groups (LL, HL, LH, and HH subbands). A naive algorithm of 2-D transform computation directly follows the horizontal and vertical filtering loops. As a consequence, the number of elementary polyphase matrices is doubled.

Unfortunately, this separable computation does not reflect the requirements of the parallel architectures where the scheme will need twice as many synchronizations. Such synchronizations often form a bottleneck of the overall calculation. State-of-the-art algorithms fuse the horizontal and vertical loops into a single one, which results in the single-loop approach. However, the number of the elementary polyphase matrices and thus the number of memory barriers remain unaffected.

To solve the outlined issue, we propose several novel spatial lifting structures computing the 2-D discrete wavelet transform with reduced number of memory barriers. These lifting structures are presented in the order in which they were gradually derived. The presented work is accompanied by exhaustive performance experiments.

A typical representative of parallel architectures is the graphics processing unit (GPU) capable of executing a general-purpose program. Actually, this is the architecture used to evaluate the performance of algorithms presented in this paper. We have employed OpenCL language for writing underlying implementations. These implementations were then subject of performance measurements on significant graphics cards of two biggest vendors.

In order to avoid misunderstandings, it should be noted that the schemes presented in this paper do not affect an image compression ratio nor quality. The schemes only affect the speed in which the compression is completed. Since practical applications require a multi-level discrete wavelet decomposition, the question of how to compute this multi-scale pyramid may arise. In this case, the schemes discussed in this paper can simply be applied in a sequence exchanging intermediate results through a off-chip memory (a global memory in the case of GPU). Another possibility is to apply this sequence on blocks exchanging the results using a fast on-chip memory (a local memory on GPU). The latter possibility was used, e.g., in [57, 14] employing the naive algorithm of 2-D transform computation.

The rest of the paper is organized as follows. Section Related Work presents the theory in the necessary level of detail. This theory includes the lifting scheme basics and the spatial lifting structures recently proposed. Subsequent Section Proposed Schemes derives the new spatial lifting structures. Additionally, Section Improvements presents a simple trick proposed in order to reduce the number of arithmetic operations. Section Evaluation and Section Performance offer a thorough performance evaluation. Finally, Section Conclusions summarizes the paper.

8.2 Related Work

In this paper, the well-known z-transform notation is employed for the description of FIR filters. The transfer function of the FIR filter h_k is a Laurent polynomial defined as

$$H(z) = \sum_{k=k_0}^{k_1-1} h_k \, z^{-k}, \tag{8.1}$$

where k_0 denotes the smallest and $k_1 - 1$ denotes the largest integer number k for which h_k is non-zero. The degree of a Laurent polynomial $H(z)$ is defined as $|H(z)| = k_1 - k_0 - 1$. Similarly, the transfer function of the two-dimensional FIR filter h_{k_m,k_n} is a bivariate Laurent polynomial defined as

$$H(z_m, z_n) = \sum_{k_m=k_{0,m}}^{k_{1,m}-1} \sum_{k_n=k_{0,n}}^{k_{1,n}-1} h_{k_m,k_n} \, z_m^{-k_m} z_n^{-k_n}, \tag{8.2}$$

where m refers to the horizontal axis and n to the vertical one. Moreover, to keep consistency with other papers, the $H^*(z_m, z_n) = H(z_n, z_m)$ denotes a filter transposed to the $H(z_m, z_n)$. For simplicity, we have made a small abuse of notation. Instead of the full notation $H(z_m, z_n)$, we only use a shortened labeling, such as H. Finally, we work with 2×2 and 4×4 matrices of Laurent polynomials. These are usually referred to as the polyphase matrices. The 2×2 matrices refers to the 1-D systems, whereas the 4×4 to the 2-D ones. For simplicity, a shortened labeling is used for matrices as well. The superscript T denotes the vector or matrix transposition.

8.2.1 Discrete Wavelet Transform

The discrete wavelet transform has undergone a gradual development [56] in the last few decades. First, S. Mallat [55] demonstrated the multi-scale wavelet decomposition computed with a pyramidal algorithm based on convolutions with quadrature mirror filters. In detail, the discrete wavelet transform splits the input signal x_k into two components L and H, each subsampled by a factor of 2. Both of these components can be computed by the discrete convolution with two FIR filters $G_0(z)$ and $G_1(z)$ followed by the subsampling. However, such computation is usually not the fastest one. The transform can also be represented by the polyphase matrix [71]. Using this representation, the input signal is initially split into the L, H components. No calculation is performed so far. After such splitting, the DWT

$$\mathbf{y} = \mathrm{M}\,\mathbf{x}. \tag{8.3}$$

is described by the 2×2 matrix M mapping the initial components

$$\mathbf{x} = \begin{bmatrix} \mathrm{L} & \mathrm{H} \end{bmatrix}^T \tag{8.4}$$

onto the resulting ones

$$\mathbf{y} = \begin{bmatrix} \mathrm{L} & \mathrm{H} \end{bmatrix}^T. \tag{8.5}$$

The polyphase matrix is initially assembled as a polynomial matrix

$$\mathrm{M} = \begin{bmatrix} \mathrm{G}_{1,o} & \mathrm{G}_{1,e} \\ \mathrm{G}_{0,o} & \mathrm{G}_{0,e} \end{bmatrix}, \tag{8.6}$$

where subscript e refers to the even coefficients, whereas o refers to the odd coefficients.

8.2.2 Lifting Scheme

As a next step, W. Sweldens [72, 31] showed how any discrete wavelet transform can be decomposed into a sequence of simple filtering steps. These steps are referred to as the lifting steps, and the scheme is known as the lifting scheme. The lifting scheme reduces the number of arithmetic operations by up to 50 %. The lifting steps occur in K pairs. The first step is referred to as the predict and the second one to as the update. It may happen that the very first step of the lifting scheme is missing and the sequence of steps starts with the update step. Usually, the very last step has a different form compared to all the others. This one is then called the scaling step.

$$\mathrm{M} = \begin{bmatrix} \zeta & 0 \\ 0 & 1/\zeta \end{bmatrix} \prod_{k=K-1}^{0} \begin{bmatrix} 1 & \mathrm{U}^{(k)} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \mathrm{P}^{(k)} & 1 \end{bmatrix}, \tag{8.7}$$

where ζ is a non-zero scaling factor, $\mathrm{P}^{(k)}$ is the kth predict convolution operator, and $\mathrm{U}^{(k)}$ is the kth update convolution operator. In this paper, we focus on a single pair of lifting steps. We thus omit the (k) superscript. We also omit the scaling step, as the application of this step is trivial.

In parallel environments [64], the processing of a single or several adjacent signal samples is mapped to independent processing units, commonly referred to as the threads. To avoid race conditions (the behavior where the output is dependent on the sequence or timing of other threads), the threads must use some type of synchronization method. In this paper, we will consider the use of memory barriers. When we return to the lifting scheme, these barriers are usually required before each of the individual lifting steps. However, certain form of the steps guarantees correctness of the calculation even without using the memory barrier between them. In this paper, the barriers are indicated by the $\big|$ symbol placed in between the steps. For example, $\mathrm{M}_2\big|\mathrm{M}_1$ denotes a sequence of two steps – the initial M_1 and the subsequent M_2 – separated by the barrier.

The schemes presented above can be extended into two dimensions. The most widely used 2-D extension is Mallat's [55] 2-D decomposition. The transform is defined as the tensor product of 1-D transforms. At each scale of such decomposition, we obtain a quadruple of wavelet coefficients (LL, HL, LH, HH).

8.2.3 Convolution and Polyphase Schemes

Similarly to the 1-D case, the transform can be computed using the convolution scheme. Considering this case, one needs to convolve the input signal with four 2-D FIR filters. This operation is followed by the subsampling in both dimensions. However, in practical implementations, the subsamplings are built into the convolutions in order to save arithmetic operations. This scheme will further be labeled as Convolution. In this scheme, no barrier is required at all.

Moreover, the 2-D transform can be described by the polyphase matrix as well. Using the polyphase representation, the input signal is initially split into the four polyphase components. No calculation is performed so far. Further, the 2-D DWT is described by the 4×4 matrix M mapping the input components

$$\mathbf{x} = \begin{bmatrix} \mathrm{LL} & \mathrm{HL} & \mathrm{LH} & \mathrm{HH} \end{bmatrix}^T \tag{8.8}$$

onto the final ones

$$\mathbf{y} = \begin{bmatrix} \mathrm{LL} & \mathrm{HL} & \mathrm{LH} & \mathrm{HH} \end{bmatrix}^T. \tag{8.9}$$

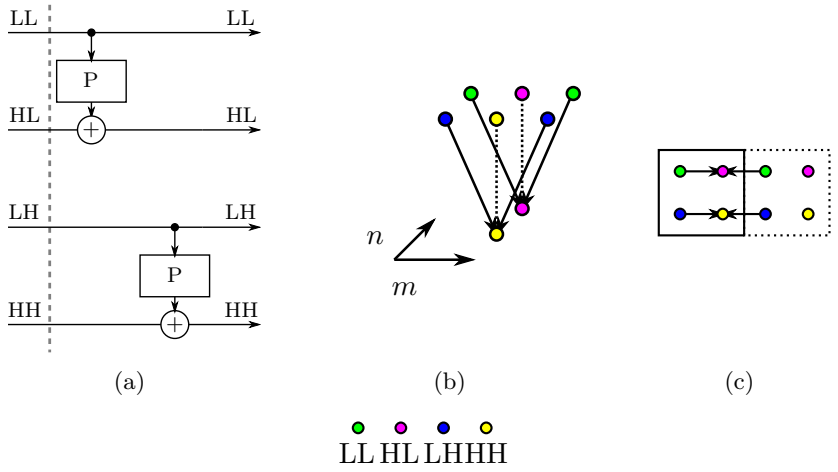


Figure 8.1: Different visual representations of the same polyphase matrix.

Similarly to the 1-D case, this can be written as

$$\mathbf{y} = \mathrm{N}_{\mathrm{P,U}} \,\big|\, \mathbf{x}, \tag{8.10}$$

where P, U are 1-D predict and update convolution operators. Please notice the included initial barrier. This scheme will further be called as Polyphase.

To define the 2-D polyphase matrices, the predict and update operators must first be migrated into two dimensions. Coupled together with filter transposition defined above, the two-dimensional counterparts of the operators are defined like follows.

$$\begin{bmatrix} \mathrm{P} \\ \mathrm{U} \\ \mathrm{P}^* \\ \mathrm{U}^* \end{bmatrix} = \begin{bmatrix} P(z_m, z_n) \\ U(z_m, z_n) \\ P^*(z_m, z_n) \\ U^*(z_m, z_n) \end{bmatrix} = \begin{bmatrix} P(z_m) \\ U(z_m) \\ P(z_n) \\ U(z_n) \end{bmatrix} \tag{8.11}$$

Roughly speaking, the P and U denote the filters oriented along the horizontal axes, whereas the P^* and U^* denote the filters oriented along the vertical one.

8.2.4 Notation

For readers not familiar with signal-processing notations, a relationship of the block and data-flow diagrams is explained in this section. In this paper, we work with 4×4 matrices of Laurent polynomials, usually referred to as the polyphase matrices, for example, this one

$$\mathrm{T}_{\mathrm{P}}^{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathrm{P} & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \mathrm{P} & 1 \end{bmatrix}. \tag{8.12}$$

Since these matrices define a linear mapping from vectors of form $\begin{bmatrix} \mathrm{LL} & \mathrm{HL} & \mathrm{LH} & \mathrm{HH} \end{bmatrix}^T$ to vectors of the same form, we can simply illustrate this mapping by the block diagram in Fig. 8.1(a).

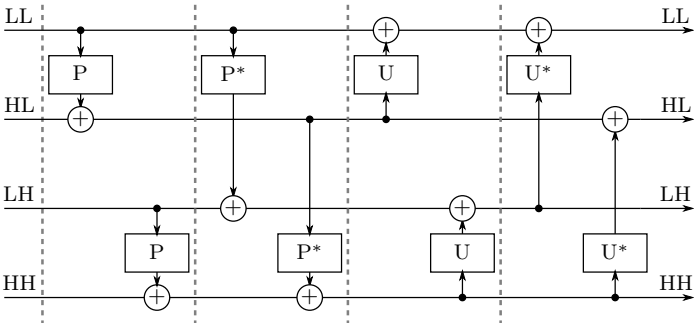


Figure 8.2: Block diagram of the Sweldens scheme. The dashed vertical lines indicate barriers. The left half corresponds to the spatial predict operator, whereas the right half to the update one.

Moreover, the matrices are composed of elementary lifting operators like

$$P(z) \ = -1/2(1 + z^{-1}). \tag{8.13}$$

If we substitute such particular polynomials into the matrix, the mapping gets a specific shape, as illustrated by the dataflow diagram in Fig. 8.1(b). The solid arrows correspond to multiplication by $-1/2$ along with subsequent summation. The dotted arrows similarly correspond to multiplication by factor of 1, since the matrix T_P^H contains ones on the main diagonal.

For reader's convenience, we use two-dimensional diagrams to illustrate the schemes with CDF 5/3 wavelets. For the example above, such a diagram is shown in Fig. 8.1(c), whereas the elementary quadruples of coefficients are highlighted by solid and dotted boxes.

8.2.5 Sweldens Scheme

Following the Mallat's scheme, the predict and update lifting steps are applied in both directions sequentially. This can be classified as a separable scheme. As the convolution is the linear operator, horizontal and vertical steps can be arbitrary interleaved. The baseline formulation of this scheme will be considered as follows. The predict steps are always preceding the update ones. Such separable scheme can be formally described by

$$\mathbf{y} = \mathrm{S}_\mathrm{U}^V \,|\, \mathrm{S}_\mathrm{U}^H \,|\, \mathrm{T}_\mathrm{P}^V \,|\, \mathrm{T}_\mathrm{P}^H \,|\, \mathbf{x}, \tag{8.14}$$

where the individual matrices are defined as follows. Let us mention a short comment on the matrix notation used. For example, the matrix T_P^H is parameterized by the P polynomial. Further in the text, the same matrix appears parameterized by different polynomials, which is completely valid. As it can be expected, the matrix T^H definition is not repeated for such case. For better understanding, the corresponding signal-processing block diagram is shown in Fig. 8.2. For the CDF 5/3 wavelet, these steps are also graphically illustrated in Fig. 8.3.

$$\mathrm{T}_\mathrm{P}^H \ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathrm{P} & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \mathrm{P} & 1 \end{bmatrix} \tag{8.15}$$

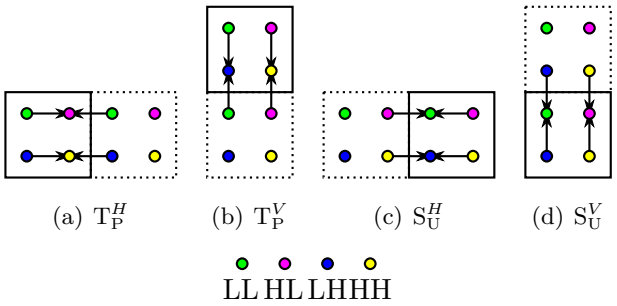


Figure 8.3: 2-D dataflow diagram, CDF 5/3 wavelet, Sweldens lifting scheme. The displayed part of the calculation results in the coefficients inside of the solid box. The dotted boxes refer to the surrounding threads.

$$T_P^V = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ P^* & 0 & 1 & 0 \\ 0 & P^* & 0 & 1 \end{bmatrix} \tag{8.16}$$

$$S_U^H = \begin{bmatrix} 1 & U & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & U \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.17}$$

$$S_U^V = \begin{bmatrix} 1 & 0 & U^* & 0 \\ 0 & 1 & 0 & U^* \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.18}$$

Please note the barriers in between each of the lifting steps. In total, four barriers are required for each pair of the original 1-D lifting steps. This scheme will further be labeled as Sweldens.

Contemporary approaches on parallel architectures most commonly reflect this separable Sweldens scheme. Exceptionally, the Convolution scheme is employed. Considering the independent horizontal and vertical filtering steps, several different strategies of 2-D DWT implementation can be used. These strategies can be divided into three groups – row–column, block-based, and pipelined methods. The row–column methods process all of the horizontal filtering steps prior to the vertical ones. The row–column method applied on the entire 2-D image was used for instance in [37, 74, 75, 36, 15, 39, 40]. In some papers, the transition between the horizontal and vertical stage is accompanied with data transposition. The pipelined methods was used, e.g., in [52] and [53]. These methods uses moving window for the vertical part of the transform. However, the horizontal and vertical parts remain separated. The block-based methods were used, e.g., in [57, 15, 14, 69]. The transform is tiled into blocks, in which the horizontal and vertical processing still remain separated. However, between these parts, the data remain loaded in the local memory (making them

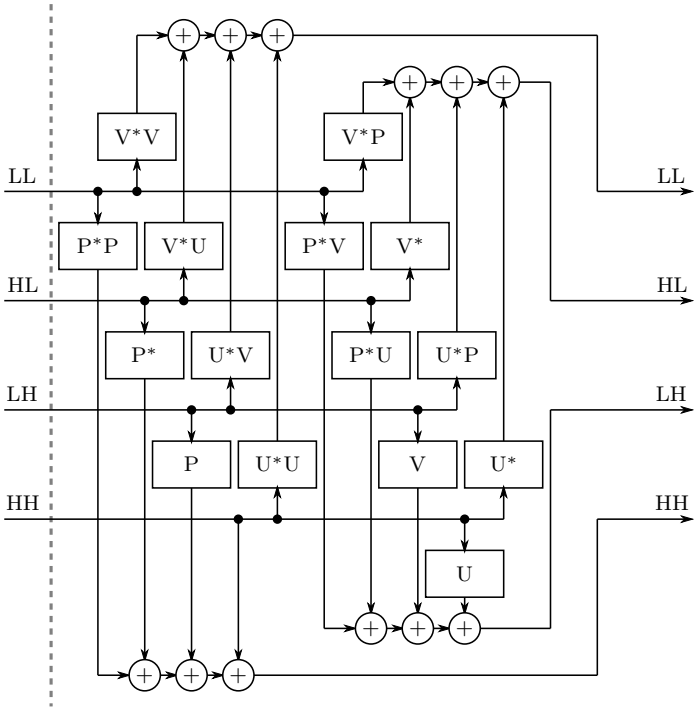
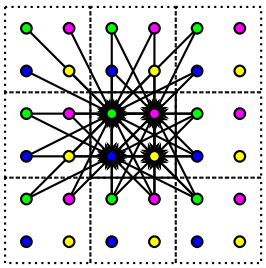


Figure 8.4: Block diagram of the Polyphase scheme. The dashed vertical lines indicate implicit barrier.



(a) $N_{P,U}$

Figure 8.5: 2-D dataflow diagram, CDF 5/3 wavelet, Polyphase scheme. The solid box in the middle corresponds to the output coefficients.

faster accessible). For the sake of completeness, some of the works compute an entire [57] or partial [14] multi-scale transform inside the blocks.

Going back to the Polyphase scheme, the polyphase matrix

$$\mathrm{N}_{\mathrm{P,U}} = \begin{bmatrix} \mathrm{V^*V} & \mathrm{V^*U} & \mathrm{U^*V} & \mathrm{U^*U} \\ \mathrm{V^*P} & \mathrm{V^*} & \mathrm{U^*P} & \mathrm{U^*} \\ \mathrm{P^*V} & \mathrm{P^*U} & \mathrm{V} & \mathrm{U} \\ \mathrm{P^*P} & \mathrm{P^*} & \mathrm{P} & 1 \end{bmatrix} \tag{8.19}$$

can expressed using the auxiliary polynomial $\mathrm{V} = \mathrm{PU} + 1$. The matrix can be obtained as the product of individual matrices of the Sweldens scheme. In this scheme, it is no longer possible to distinguish the vertical and horizontal filtering. Only an initial barrier is required for this scheme. Unfortunately, the number of arithmetic operations has grown in proportion to the square of filter sizes. The corresponding generic signal-processing diagram is shown in Fig. 8.4. For the CDF 5/3 wavelet, these operations are illustrated in Fig. 8.5.

8.2.6 Iwahashi Scheme

Recently, Iwahashi *et al.* [46, 45, 47] presented the non-separable lifting scheme, consisting of three spatial lifting steps. As in the previous case, it is not possible to distinguish the vertical and horizontal filtering. The three steps can be described as follows. Initially, a 2-D lifting step leading to the computation of the HH coefficient is performed. This step corresponds to a spatial predict convolution operator. This is followed by parallel computation of the HL and LH coefficients, using the original 1-D predict and update filters. In the third step, the LL coefficient is computed using another 2-D filter. The last step can be understood as a spatial update operator. In the matrix notation, the scheme can be defined as

$$\mathbf{y} = \mathrm{S}^I_{\mathrm{U}} \,|\, \mathrm{R}^I_{\mathrm{P,U}} \,|\, \mathrm{T}^I_{\mathrm{P}} \,|\, \mathbf{x}, \tag{8.20}$$

where the individual matrices are defined as follows. The signal-processing diagram is shown in Fig. 8.6. For the CDF 5/3 wavelet, the individual steps are illustrated in Fig. 8.7.

$$\mathrm{T}^I_{\mathrm{P}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \mathrm{PP^*} & \mathrm{P^*} & \mathrm{P} & 1 \end{bmatrix} \tag{8.21}$$

$$\mathrm{R}^I_{\mathrm{P,U}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathrm{P} & 1 & 0 & \mathrm{U^*} \\ \mathrm{P^*} & 0 & 1 & \mathrm{U} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.22}$$

$$\mathrm{S}^I_{\mathrm{U}} = \begin{bmatrix} 1 & \mathrm{U} & \mathrm{U^*} & -\mathrm{UU^*} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.23}$$

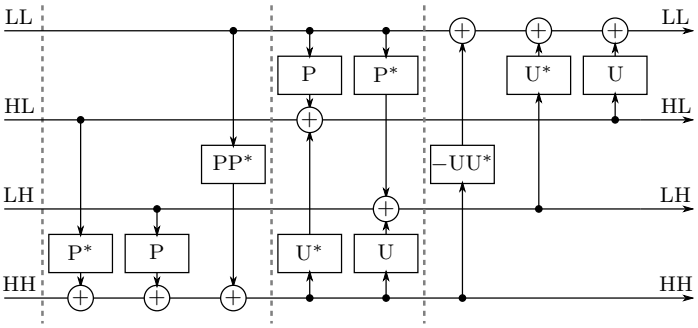
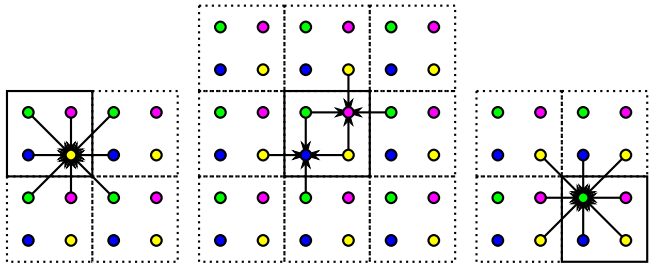


Figure 8.6: Block diagram of the Iwahashi scheme. The dashed vertical lines indicate barriers.



(a) $\mathrm{T}_{\mathrm{P}}^{I}$ (b) $\mathrm{R}_{\mathrm{P,U}}^{I}$ (c) $\mathrm{S}_{\mathrm{U}}^{I}$

Figure 8.7: 2-D dataflow diagram, CDF 5/3 wavelet, Iwahashi lifting scheme. The solid box corresponds to the output coefficients.

Three barriers are required in between these steps. As for the Polyphase scheme, the number of arithmetic operations increased proportionally to the square of filter sizes. However, the total number of operations is significantly lower. This scheme will further be labeled as Iwahashi.

When we compare the separable Sweldens and non-separable Iwahashi schemes, some findings becomes obvious at first glance. The number of operations tends to be considerably smaller for the separable case. On the other hand, the number of memory barriers in the non-separable scheme was reduced to 75 % (from four to three barriers). The Polyphase scheme stands apart from these two schemes. It needs only an initial memory barrier. Unfortunately, the number of arithmetic operations is unreasonably large. This is caused by the number of non-zero elements in the corresponding polyphase matrix as well as by the degree of the longest filter V. For clarification, the product of a Laurent polynomial of degree $|P(z)|$ and a Laurent polynomial of degree $|U(z)|$ is a Laurent polynomial of degree $|P(z)| + |U(z)|$. Finally, the Convolution scheme employing four 2-D filters is even worse in terms of the operations. Anyway, only an initial memory is required here as well. Detailed quantitative comparison is provided in Section Evaluation.

When we consider the linearity of the convolution and the dependencies between the individual lifting steps, several gaps can be inferred in the schemes described above. Recombining the operations into a new form could lead to the removal of unnecessary barriers. Actually, exactly this idea is investigated in the following section, in which several novel 2-D schemes are proposed.

Since this work is based on our previous work in [50], it should be explained what the difference between this work and [50] is. In [50], we presented a block-based method employing a scheme foregoing the schemes proposed in this paper. Unlike [50], the schemes presented in this paper are defined by general predict and update operators.

8.3 Proposed Schemes

In this section, the polyphase matrices, known so far, are reassembled in order to obtain the schemes suitable for parallel architectures. All of the schemes discussed here are general, and they can be used for any discrete wavelet transform. Please note that the contribution of this paper is presented in this section and the following one.

8.3.1 Explosive Scheme

When we take a detailed look at the original 1-D lifting scheme, a certain pattern can be identified in the predict and update steps. Particularly, the predicts transmit data from L into H samples, whereas the updates transmit data from H into L. The transmission can be viewed from two perspectives – the data flow out from a source component (similarly to an explosion); or the data flow in into a destination component (an implosion). As it can be expected, the Sweldens scheme exactly follows this pattern, since this scheme is a mere extension of 1-D lifting into two dimensions. Roles of source and destination samples properly turn during four lifting steps (horizontal and vertical, predict and update). This procedure can be also seen as a data transmission in direction from LL into HH component (using 1-D predicts), and a transmission from HH into LL one (using updates). The HL and LH components are not relevant in this view. The situation is clearly visible in Fig. 8.2. In contrast to this scheme, the Iwahashi scheme has a different structure. The leading step transmits data into HH component (using predicts), while the trailing one transmits them

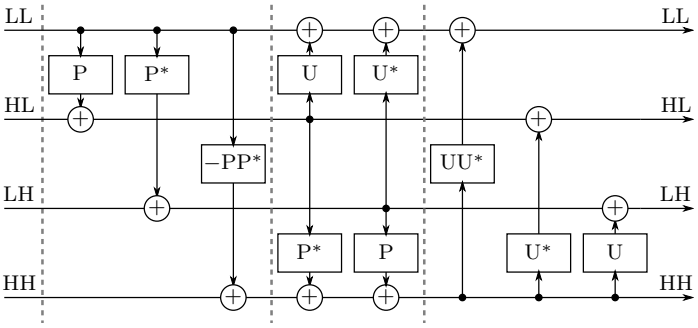
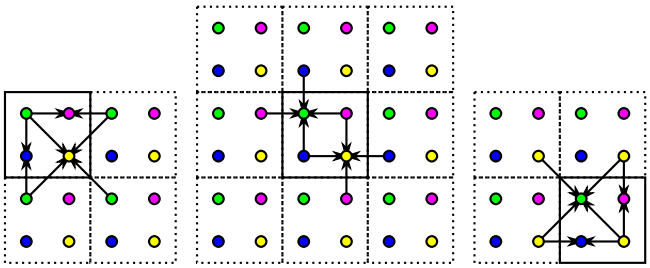


Figure 8.8: Block diagram of the Explosive scheme. The dashed vertical lines indicate barriers.



(a) $\mathrm{T}_{\mathrm{P}}^{E}$ (b) $\mathrm{R}_{\mathrm{P,U}}^{E}$ (c) $\mathrm{S}_{\mathrm{U}}^{E}$

Figure 8.9: 2-D dataflow diagram, CDF 5/3 wavelet, Explosive lifting scheme. The solid box corresponds to the output coefficients.

into LL one (updates). However, no exclusive source components can be identified in this case. The remaining step in the middle is not relevant. See the block diagram in Fig. 8.6. Regarding to the perspectives outlined above, the Iwahashi scheme can be classified as an implosive one. However, this is not the only three-step version (two-step scheme is discussed below in the text). Similar scheme can be formulated using data explosions instead of the implosions. Particularly, the LL component spreads the data into its neighborhood during the predict step, whereas the data flow out from the HH component in the update step. No exclusive destination components can be identified here as well. Again, the step in the middle is not relevant. For further purposes, this newly proposed scheme will be labeled as Explosive. The block diagram is shown in Fig. 8.8. The steps for the CDF 5/3 wavelet are also illustrated in Fig. 8.9. Formally, the scheme can be defined as

$$\mathbf{y} = \mathrm{S}_{\mathrm{U}}^{E} \,\big|\, \mathrm{R}_{\mathrm{P,U}}^{E} \,\big|\, \mathrm{T}_{\mathrm{P}}^{E} \,\big|\, \mathbf{x}, \tag{8.24}$$

where the individual matrices follows. Three barriers are required, as in the case of the Iwahashi scheme.

$$\mathrm{T}_{\mathrm{P}}^{E} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathrm{P} & 1 & 0 & 0 \\ \mathrm{P}^{*} & 0 & 1 & 0 \\ -\mathrm{PP}^{*} & 0 & 0 & 1 \end{bmatrix} \tag{8.25}$$

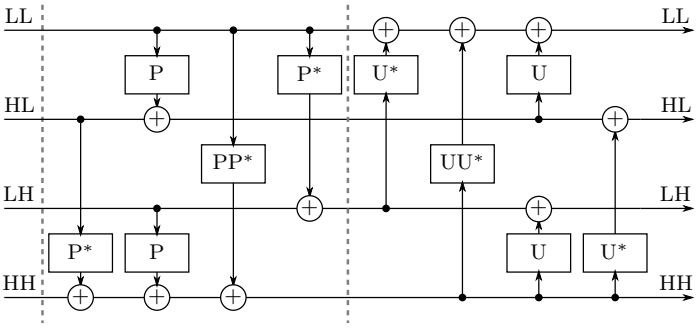


Figure 8.10: Block diagram of the Monolithic scheme. The dashed vertical lines indicate barriers. The left half corresponds to the predict operator, whereas the right half to the update.

$$\mathrm{R}^{E}_{\mathrm{P,U}} = \begin{bmatrix} 1 & \mathrm{U} & \mathrm{U}^* & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \mathrm{P}^* & \mathrm{P} & 1 \end{bmatrix} \tag{8.26}$$

$$\mathrm{S}^{E}_{\mathrm{U}} = \begin{bmatrix} 1 & 0 & 0 & \mathrm{UU}^* \\ 0 & 1 & 0 & \mathrm{U}^* \\ 0 & 0 & 1 & \mathrm{U} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.27}$$

8.3.2 Monolithic Scheme

Motivated by the work of Iwahashi *et al.* [47], we have reorganized the elementary lifting filters in order to remove the middle lifting step. This action consequently reduces the number of memory barriers. As a result, we receive a new two-step non-separable scheme. The first step corresponds to a spatial predict operator. This one is completely responsible for the HH coefficient. In addition, the HL and LH coefficients are partially computed here as well. The second step corresponds to a spatial update. It is responsible for the LL coefficient and completion of the HL and LH ones. Formally, the scheme is defined as

$$\mathbf{y} = \mathrm{S}_{\mathrm{U}} \,\big|\, \mathrm{T}_{\mathrm{P}} \,\big|\, \mathbf{x}, \tag{8.28}$$

where the S_{U} and T_{P} are defined as follows. Moreover, the hypotetical signal-processing diagram is shown in Fig. 8.10. For the CDF 5/3 wavelet, the scheme is graphically illustrated in Fig. 8.11.

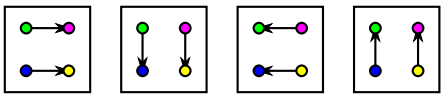
$$\mathrm{T}_{\mathrm{P}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathrm{P} & 1 & 0 & 0 \\ \mathrm{P}^* & 0 & 1 & 0 \\ \mathrm{PP}^* & \mathrm{P}^* & \mathrm{P} & 1 \end{bmatrix} \tag{8.29}$$

(a) T_P (b) S_U

Figure 8.11: 2-D dataflow diagram, CDF 5/3 wavelet, Monolithic scheme. The solid box corresponds to the output coefficients.

step	Sweldens	Monolithic	Iwahashi	Explosive
predict				
middle				
update				

Table 8.1: CDF 5/3 wavelet. Shapes of spatial lifting steps for selected schemes. The step in the middle raised from the combination of the original predict and update steps. Illustrative purpose only.



(a) $\mathrm{T}^H_{\mathrm{P}_0}$ (b) $\mathrm{T}^V_{\mathrm{P}_0}$ (c) $\mathrm{S}^H_{\mathrm{U}_0}$ (d) $\mathrm{S}^V_{\mathrm{U}_0}$

Figure 8.12: 2-D dataflow diagram, CDF 5/3 wavelet, common steps for all improved schemes.

$$\mathrm{S}_{\mathrm{U}} = \begin{bmatrix} 1 & \mathrm{U} & \mathrm{U}^* & \mathrm{U}\mathrm{U}^* \\ 0 & 1 & 0 & \mathrm{U}^* \\ 0 & 0 & 1 & \mathrm{U} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.30}$$

The total number of operations remained the same as for the Iwahashi scheme. However, the number of the explicit barriers has been reduced to only two. This is a crucial contribution of our work. Further in the paper, this scheme will be labeled as Monolithic. One can easily verify the correctness of the proposed scheme by comparing the product $\mathrm{S}_{\mathrm{U}}\mathrm{T}_{\mathrm{P}}$ to the matrix $\mathrm{N}_{\mathrm{P,U}}$ of the Polyphase scheme.

A comparison of the shapes for selected schemes can be found in Table 8.1. Regarding the Polyphase scheme, no spatial predict nor update step can be identified in its calculation.

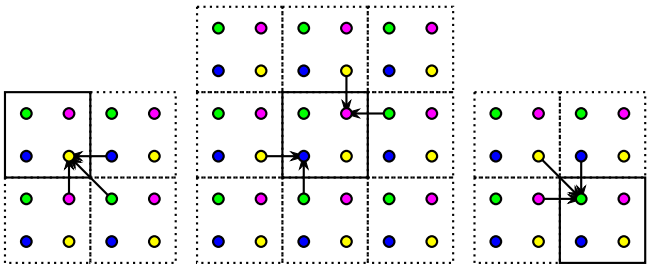
In practical implementations, the formed intermediate coefficients cannot take the same place as the input ones. Otherwise, the race condition occurs. This implies a higher memory consumption compared to the previous schemes. A particular numbers are listed in Table 8.3.

Two simple observations can be made from the scheme presented so far. The Sweldens scheme requires the lowest number of operations. In contrast to this approach, the non-separable scheme proposed above requires the lowest number of memory barriers. Combining these two observations together, new schemes can be formed. This possibility is investigated below.

8.4 Improvements

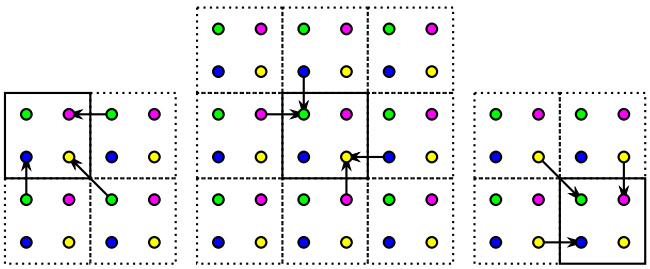
Additionally, we have made another observation. The operation composed as a product of monomials with the exponent of z_n and z_n being equal to zero (i.e., scalars) never touch the coefficients belonging to the surrounding threads. As the convolution is the linear operation, this monomial can be detached from the original operator and subsequently calculated using the Sweldens scheme. This scheme has a minimal number of arithmetic operations. The rest of the original polynomial shall be computed using different scheme, according to suitability for a particular platform.

In more detail, the original filters were split into two halves as $\mathrm{P} = \mathrm{P}_0 + \mathrm{P}_1$, and $\mathrm{U} = \mathrm{U}_0 + \mathrm{U}_1$, where P_0 and U_0 are scalars. This is a fundamental step for the following constructions. Now, the scalars $\mathrm{P}_0, \mathrm{U}_0$ can be utilized in the separable Sweldens scheme. This part will never touch the extraneous threads. For a better understanding, see the dataflow diagram in Fig. 8.12. Conversely, the $\mathrm{P}_1, \mathrm{U}_1$ shall be employed in the Explosive, Iwahashi, Monolithic, or Polyphase scheme in order to minimize the number of required



(a) $\mathrm{T}^{I}_{\mathrm{P}_1}$ (b) $\mathrm{R}^{I}_{\mathrm{P}_1,\mathrm{U}_1}$ (c) $\mathrm{S}^{I}_{\mathrm{U}_1}$

Figure 8.13: 2-D dataflow diagram, CDF 5/3 wavelet, Iwahashi* scheme. The solid box corresponds to the output coefficients.



(a) $\mathrm{T}^{E}_{\mathrm{P}_1}$ (b) $\mathrm{R}^{E}_{\mathrm{P}_1,\mathrm{U}_1}$ (c) $\mathrm{S}^{E}_{\mathrm{U}_1}$

Figure 8.14: 2-D dataflow diagram, CDF 5/3 wavelet, Explosive* scheme. The solid box corresponds to the output coefficients.

memory barriers. Note that these two schemes can be combined into joint lifting steps. However, such optimization is a simple matter of a specific implementation.

Initially, we have employed the idea described in previous paragraphs in conjunction with the Iwahashi scheme. The resulting scheme is defined as

$$\mathbf{y} = \mathrm{S}^{V}_{\mathrm{U}_0}\,\mathrm{S}^{H}_{\mathrm{U}_0}\,\mathrm{S}^{I}_{\mathrm{U}_1} \,\big|\, \mathrm{R}^{I}_{\mathrm{P}_1,\mathrm{U}_1} \,\big|\, \mathrm{T}^{I}_{\mathrm{P}_1} \,\big|\, \mathrm{T}^{V}_{\mathrm{P}_0}\,\mathrm{T}^{H}_{\mathrm{P}_0}\mathbf{x}, \tag{8.31}$$

where the individual matrices are defined above in the paper. The number of barriers remains the same as for the original Iwahashi scheme. The operations represented by the matrices defined for the Sweldens scheme do not need to be preceded by a barrier. The scheme will be further referred to as Iwahashi*. For the CDF 5/3 wavelet, this scheme is graphically illustrated in Fig. 8.13.

Similarly, we have employed the same trick in conjunction with the Explosive scheme. This time, the scheme is defined as

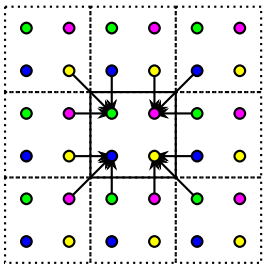
$$\mathbf{y} = \mathrm{S}^{V}_{\mathrm{U}_0}\,\mathrm{S}^{H}_{\mathrm{U}_0}\mathrm{S}^{E}_{\mathrm{U}_1} \,\big|\, \mathrm{R}^{E}_{\mathrm{P}_1,\mathrm{U}_1} \,\big|\, \mathrm{T}^{E}_{\mathrm{P}_1} \,\big|\, \mathrm{T}^{V}_{\mathrm{P}_0}\,\mathrm{T}^{H}_{\mathrm{P}_0}\,\mathbf{x}. \tag{8.32}$$

Also in this case, the number of barriers remains the same as for the original scheme. Analogously to the previous case, this scheme will be referred to as Explosive*. The dataflow diagram for the CDF 5/3 wavelet is shown in Fig. 8.14.

As a next step, consider a new construction based on the Monolithic scheme. The same trick can be utilized here as well. In the matrix notation, the newly composed scheme is

(a) $\mathrm{T}_{\mathrm{P}_1}$ (b) $\mathrm{S}_{\mathrm{U}_1}$

Figure 8.15: 2-D dataflow diagram, CDF 5/3 wavelet, Monolithic* scheme. The solid box corresponds to the output coefficients.



(a) $\mathrm{N}_{\mathrm{P}_1,\mathrm{U}_1}$

Figure 8.16: 2-D dataflow diagram, CDF 5/3 wavelet, Polyphase* scheme. The solid box corresponds to the output.

defined as

$$\mathbf{y} = \mathrm{S}^V_{\mathrm{U}_0}\,\mathrm{S}^H_{\mathrm{U}_0}\,\mathrm{S}_{\mathrm{U}_1} \,\big|\, \mathrm{T}^V_{\mathrm{P}_0}\,\mathrm{T}^H_{\mathrm{P}_0}\,\mathrm{T}_{\mathrm{P}_1}\,\mathbf{x}, \tag{8.33}$$

where the individual matrices are defined above in the text. For the CDF 5/3 wavelet, this scheme is graphically illustrated in Fig. 8.15. We will label this scheme as Monolithic*.

The schemes described above are formed such a way that the first lifting step (comprising $\mathrm{P}_1, \mathrm{U}_1$) after the barrier access coefficients of the surrounding threads. The subsequent or preceding steps (comprising $\mathrm{P}_0, \mathrm{U}_0$) read only the local coefficients, which are not accessed by the other threads. Then, the whole sequence can be repeated. Of course, the calculation of transforms consisting of several pairs of lifting steps comprises several such connected schemes.

Finally, we have decided to remove the last explicit barrier, leaving only the initial one in place. The trick lies in the appropriate combination of the Sweldens and Polyphase schemes. This time, the non-separable parts are merged into a joint step $\mathrm{N}_{\mathrm{P}_1,\mathrm{U}_1}$. This step is inherently preceded by a barrier. In case of an initial pair of lifting steps, the barrier at the beginning of the computation is used for this purpose. In more detail, after the input data have been read by each computation unit, the calculations $\mathrm{T}^V_{\mathrm{P}_0}\,\mathrm{T}^H_{\mathrm{P}_0}$ are immediately performed. At this point, the intermediate results can be appropriately shared. This is followed by the initial barrier. Regarding the transforms consisting of several such schemes, the barrier between the connecting schemes is gratefully exploited. In any case, the scheme is thus composed as

$$\mathbf{y} = \mathrm{S}^V_{\mathrm{U}_0}\,\mathrm{S}^H_{\mathrm{U}_0}\,\mathrm{N}_{\mathrm{P}_1,\mathrm{U}_1} \,\big|\, \mathrm{T}^V_{\mathrm{P}_0}\,\mathrm{T}^H_{\mathrm{P}_0}\,\mathbf{x} \tag{8.34}$$

including the discussed barrier. For the CDF 5/3 wavelet, the steps are illustrated in Fig. 8.16. We will label this scheme as Polyphase*.

For the sake of clarity, the proposed schemes will now be summarized. By reversing the direction of filtering steps in the Iwahashi scheme, the new Explosive scheme was formed. As a next step, the polynomials of the original polyphase matrix were reassembled into a new two-step form. In between the steps, a memory barrier has to be placed. This scheme is denoted as Monolithic. Moreover, the number of arithmetic operations was reduced by splitting the polynomial into two parts. These newly formed polynomials are then employed in appropriate schemes. In this manner, the number of barriers remains unaffected, while the number of operations has been reduced. This simple trick has resulted in the Iwahashi*, Explosive*, Monolithic*, and Polyphase* schemes. Once again, we would like to emphasize that the schemes presented in this paper are general and they are not limited to any specific type of transform.

8.5 Evaluation

This section analyzes in detail various attributes of the schemes described in the previous sections. Namely, synchronization and memory demands for different wavelets are examined. We realize that such properties do not provide sufficient information on a performance in real environments. For this reason, we are interested in comparing the performance of the discussed schemes on real graphics cards in terms of memory bandwidth in the next section.

The evaluation is presented using the following three wavelets. The first wavelet we have employed is the CDF [30] 5/3 wavelet. This one is used for a lossless compression in the JPEG 2000 compression standard. The lifting scheme is defined by

$$\begin{bmatrix} P(z) \\ U(z) \end{bmatrix} = \begin{bmatrix} -1/2(1+z^{-1}) \\ 1/4(1+z \quad) \end{bmatrix}, \tag{8.35}$$

and the scaling factor $\zeta = \sqrt{2}$.

As the second wavelet, we have chosen the CDF 9/7 wavelet. In the JPEG 2000 standard, this wavelet is used as a basis for a lossy compression. The underlying scheme is given by

$$\begin{bmatrix} P^{(0)}(z) \\ U^{(0)}(z) \\ P^{(1)}(z) \\ U^{(1)}(z) \end{bmatrix} = \begin{bmatrix} \alpha(1+z^{-1}) \\ \beta(1+z \quad) \\ \gamma(1+z^{-1}) \\ \delta(1+z \quad) \end{bmatrix}, \tag{8.36}$$

where the $\alpha, \beta, \gamma, \delta$, and the ζ are defined in [31]. Both the CDF wavelets have predict and update convolution operators of degree 1 (two-tap symmetric filters).

The last wavelet included in the comparison is $(4, 4)$ interpolating transform built from the interpolating Deslauriers–Dubuc [72], defined by

$$\begin{bmatrix} P(z) \\ U(z) \end{bmatrix} = \begin{bmatrix} 1/16(z+z^{-2}) - 9/16(1+z^{-1}) \\ 9/32(1+z) - 1/32(z^{-1}+z^{2}) \end{bmatrix}. \tag{8.37}$$

This wavelet is used in Dirac video compression standard. For simplicity, we refer this one to as DD 13/7. The underlying lifting scheme differs from the two previous in employed

predict and update convolution operators. These operators now have a degree of 3 instead of 1. Consequently, this difference has resulted in a significantly higher number of arithmetic operations in the case of non-separable filtering steps.

The first examined parameters include the number of arithmetic operations (the scaling steps were omitted) and the number of memory barriers. The schemes presented in this paper can be directly applied on the CDF 5/3 and DD 13/7 transforms, as these comprises only a single pair of lifting steps. The CDF 9/7 transform is computed by two such connected schemes. The comparison is shown in Table 8.2. Several expectations can be made from the table. On architectures based on serial computation, the schemes should perform accordingly to the number of arithmetic operations. However, on the parallel architectures, the number of employed memory barriers is expected to play an important role. Some of the schemes could benefit from this property.

As can be seen from the referenced table, the Sweldens scheme always leads to the smallest number of operations coupled with the highest number of barriers. The recently proposed Iwahashi scheme reduces the number of barriers by one per one pair of original 1-D lifting steps. Unfortunately, the number of operations is increased at the same time. This increase is particularly noticeable on longer lifting filters, as in the case of DD 13/7 wavelet. The Monolithic scheme further reduces the number of barriers by one per one pair of original steps while keeping the number of operations untouched. In addition to this, the Monolithic* scheme reduces the number of operations. This reduction is most evident on short lifting filters. For instance, in the case of CDF wavelets, the number of operations is reduced to 75 %, whereas in the case of DD 13/7 wavelet, the number of operations is only reduced to 78 %. The number of barriers per one pair of original lifting steps can be even further reduced to a single one by combining all operations into a single step. Such case corresponds to the Polyphase scheme. Unfortunately, the number of operations was increased enormously. For shorter lifting filters, this number can be noticeably reduced using the Polyphase* scheme, in which the number of barriers remains the same. Finally, for lifting factorizations consisting of several pairs of steps, it makes sense to reduce the number of barriers to a single one by using the Convolution scheme. In such case, the number of operations is sadly the highest of all of the schemes.

Other examined parameters included the memory footprint, and number of memory loads/stores. These parameters can be determined from Table 8.3 and Table 8.4. All of the numbers are given with respect to the quadruple of coefficients, which usually correspond to a single thread. The number of load (read) operations depends on the length of the lifting operators. For example, the CDF 5/3 and CDF 9/7 factorizations consist of degree-1 convolutional filters. On the contrary, the DD 13/7 consists of degree-3 filters. The number of store (write) operations is independent of the underlying scheme. It may happen that the local memory footprint for the connecting schemes ($K > 1$) differs from the footprint for a single predict/update pair ($K = 1$). These numbers are indicated in the parentheses in Table 8.3. For clarity, the number of memory barriers is not affected by the improvement proposed in Section Improvements.

8.6 Performance

To evaluate the considered schemes, we have decided to use high-performance GPUs programmed using the OpenCL framework. In terms of the OpenCL, the schemes are computed using parallel tasks referred to as the kernels. One item from a collection of parallel executions of a kernel is referred to as the work-item or thread. The threads that execute on a

wavelet	scheme	barriers	operations
CDF 5/3	Sweldens	4	16
	Iwahashi	3	24
	Iwahashi*	3	18
	Explosive	3	24
	Explosive*	3	18
	Monolithic	2	24
	Monolithic*	2	18
	Polyphase	1	63
	Polyphase*	1	23
	Convolution	1	64
CDF 9/7	Sweldens	8	32
	Iwahashi	6	48
	Iwahashi*	6	36
	Explosive	6	48
	Explosive*	6	36
	Monolithic	4	48
	Monolithic*	4	36
	Polyphase	2	126
	Polyphase*	2	46
	Convolution	1	256
DD 13/7	Sweldens	4	32
	Iwahashi	3	64
	Iwahashi*	3	50
	Explosive	3	64
	Explosive*	3	50
	Monolithic	2	64
	Monolithic*	2	50
	Polyphase	1	255
	Polyphase*	1	203
	Convolution	1	256

Table 8.2: Number of operations and memory barriers examined for various wavelets.

scheme	barriers	single	double
Sweldens	4	**2**	**3**
Iwahashi	3	3	4
Iwahashi*	3	3	(6) 4
Explosive	3	**2**	**3**
Explosive*	3	**2**	**3**
Monolithic	2	3	6
Monolithic*	2	3	6
Polyphase	**1**	4	(8) 4
Polyphase*	**1**	4	(8) 4

Table 8.3: Number of memory barriers and local memory cells per quadruple required by the schemes discussed in this paper. Memory cells are given for a single buffering (two barriers) as well as a double buffering (only a single barrier). The numbers in parentheses are valid in the case of connecting schemes. Best features in bold.

scheme	write	read degree-1	read degree-3
Sweldens	$1 + 4K$	$8K$	$24K$
Iwahashi	$2 + 4K$	$10K$	$42K$
Iwahashi*	$6K$	$10K$	$42K$
Explosive	$4K$	$10K$	$42K$
Explosive*	$4K$	$10K$	$42K$
Monolithic	$6K$	$10K$	$42K$
Monolithic*	$6K$	$10K$	$42K$
Polyphase	$4K$	$21K$	$117K$
Polyphase*	$4K$	$12K$	$117K$

Table 8.4: Number of local memory reads and writes for all schemes and wavelets under examination. The K denotes the number of predict/update pairs. The degree-1 polynomials correspond to factorizations of CDF wavelets, whereas degree-3 to DD 13/7.

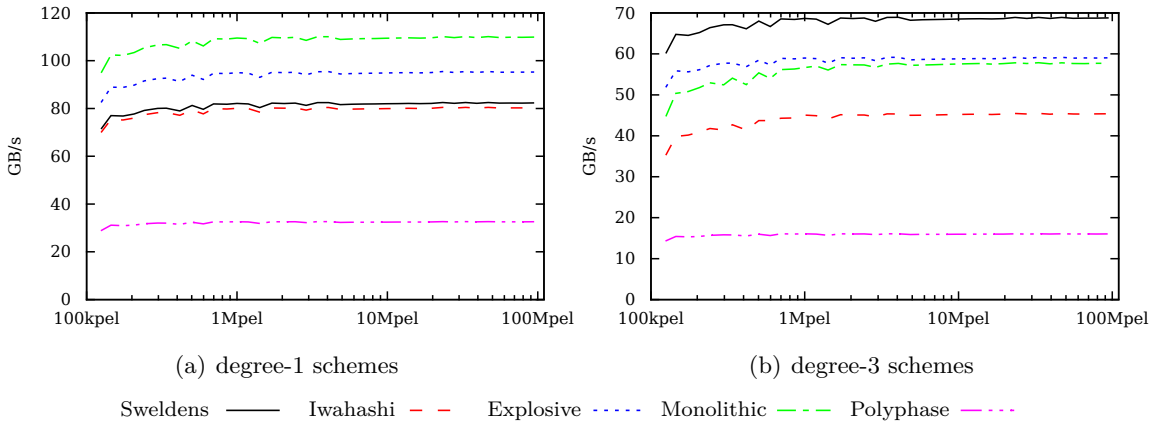


Figure 8.17: The baseline schemes on AMD 6970. Evaluation with the degree-1 and degree-3 lifting schemes. Only the performance of a transform code without the memory throughput was measured.

single compute unit are grouped into so-called work-groups. The threads in the group execute the same kernel and share local memory. Each work-group can synchronize the threads via memory barriers. Work-groups cannot synchronize with each other. Considering the processing of images, we map overlapping (in order to properly compute the coefficients near tile boundaries) image tiles onto the work-groups. Moreover, each thread is responsible for a single quadruple of transform coefficients (LL, HL, LH, and HH). At the beginning of the computation, the input image is placed in the global memory. The tiles are then transferred into the local memory. After the scheme computation, the resulting coefficients are copied back into the global memory. Such strategy fulfills the definition of a single-loop data processing (therefore without unnecessary data transfers).

One needs to recall that the row–column, block-based, and pipelined methods denote an order in which an entire input image is processed. The row–column approach indicates that all image rows are transformed prior to a transformation of all image columns (or vice versa). Between these two parts, intermediate results are stored in the global memory. However, all of the schemes presented in our paper are implemented as the block-based approaches. This means that the entire input image is split into blocks (the tiles), which are then transformed at once using the local memory for storing the intermediate results. Inside the blocks, some sort of separable scheme can be employed, which essentially corresponds to the row–column approach on a different scale. The block-based approaches in various forms were also used in, e.g., [15, 14, 69, 50]. Since the block-based approaches overcome the row–column ones (as shown in [50], or analyzed in [69]), we do not include the classical row–column methods in our performance comparison. Instead, we only compared different schemes employed under the block-based approach. In this context, we would like to make a comment on data transfers between a device and host. Due to the fact that the data are transferred in the same way for all schemes, we measured only a throughput based on a timing of a OpenCL kernels which calculate transforms. Therefore, the transfer times between device and host are not our concern.

The evaluation was performed primarily on two high-end GPUs – AMD Radeon HD 6970 and AMD Radeon HD 5870. Their technical parameters are summarized in Table 8.5. On both of the cards, variable length VLIW instructions are executed using blocks of 64

	AMD 6970	AMD 5870
vendor	AMD	AMD
model	Radeon HD 6970	Radeon HD 5870
VLIW length	4	5
multiprocessors	24	20
VLIW processors	384	320
total processors	1 536	1 600
processor clock	880 MHz	850 MHz
performance	2 703 GFLOPS	2 720 GFLOPS
memory	1 GiB GDDR5	1 GiB GDDR5
memory clock	1 375 MHz	1 200 MHz
bandwidth	176 GB/s	154 GB/s
bus width	256-bit	256-bit
local memory	32 KiB	32 KiB

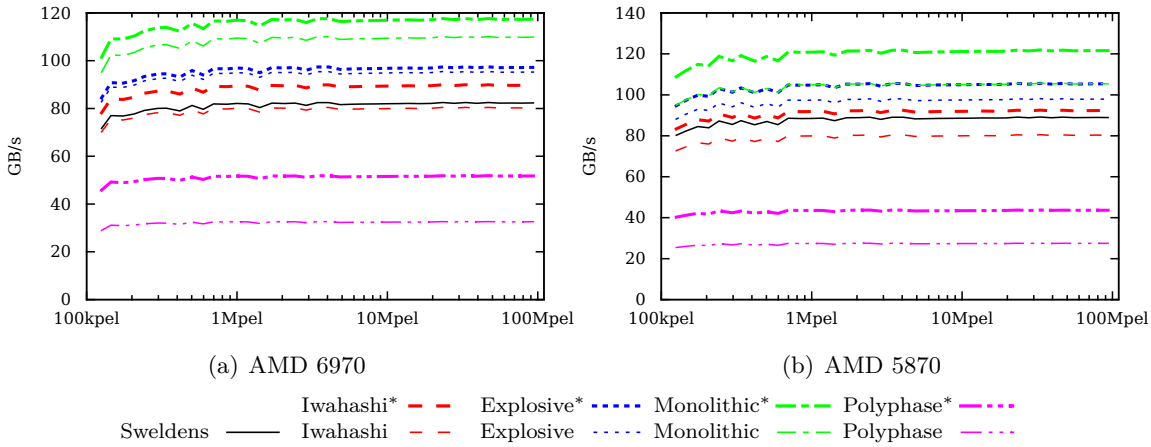
Table 8.5: Description of the GPUs used for the evaluation.

threads. In more detail, VLIW instructions can be categorized into several groups (load/store instructions, barrier instructions, control flow instructions and ALU instructions). To utilize whole processing capability, the VLIW instructions should be of maximal length. In other words, as much as possible blocks of independent instructions should be presented in a kernel.

Several possibilities raised during the implementations of the presented schemes. All of the schemes require several memory cells to interchange the intermediate coefficients. Considering the GPUs, these coefficients can be efficiently stored in the local memory. Unfortunately, it is not possible to rewrite these coefficients using a single memory barrier. As a consequence, two possibilities occur – double buffering using a single memory barrier, and single buffering using two of them. The double buffering increases the memory requirements while maintaining the number of synchronizations. Conversely, the single buffering introduces an addition barrier – separating reading and rewriting of the coefficients. For details, see Table 8.3. In other words, one can choose whether intermediate results are overwritten in their place using two memory barriers or whether these are written to another location by making use of a single barrier. Moreover, another possibility lies in the method of input and output data delivery. For evaluation purposes, it is possible to completely omit the input and output of data. The transform is not limited by memory bandwidth in this case. For real scenarios, the data can be delivered using the global or texture memory. In our experiments, we chose the latter option.

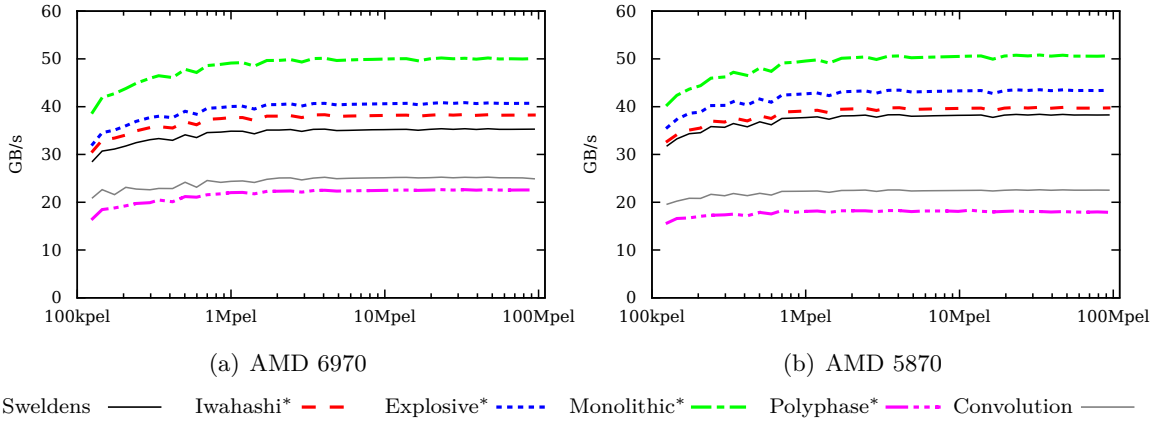
In the following paragraphs, three fundamental experiments on the described GPUs are presented. The first experiment studies the performance of the baseline schemes mentioned in this paper. The second experiment examines the influence of the improvement proposed in Section Improvements. Finally, the third experiment measures the real performance with CDF 9/7 wavelet and texture memory.

In the first experiment, the performance of the baseline schemes (without improvements proposed in Section Improvements) was examined. The measurements were conducted on the AMD 6970 card with two different lifting scheme shapes (degree-1 and degree-3 operators). Only the transform performance was measured, without the influence of



(a) AMD 6970 (b) AMD 5870

Figure 8.18: The schemes on AMD 6970 and AMD 5870. Evaluation with the degree-1 schemes. Only the performance of a transform code without the memory throughput was measured.



(a) AMD 6970 (b) AMD 5870

Figure 8.19: The improved schemes on AMD 6970 and AMD 5870. Evaluation with the CDF 9/7 wavelet and texture memory was performed.

memory throughput. The presented results are the average of ten measurements. The results are shown in Fig. 8.17. One can easily observe a different behavior for short and long lifting operators. For the short operators, the reduction in the number of lifting steps clearly improves the performance. The situation actually corresponds directly to the number of memory barriers. Conversely, in the case of the long operators, the situation is tilted in favor of the number of arithmetic operations. Note that the horizontal axes are in a logarithmic scale. The vertical axes express the transform throughput in GB/s (gigabytes per second).

In the second experiment, the contribution of the improvements proposed in Section Improvements was examined. The measurements were performed on both of the cards under the evaluation. This time we have focused on the degree-1 schemes only. As in the previous case, only the transform performance was measured using the average of ten measurements. The results are shown in Fig. 8.18. As expected, the improvements slightly increase the transform performance. However, the order of the schemes still corresponds

scheme	throughput AMD 6970	throughput AMD 5870
Monolithic*	117.426	121.579
Monolithic	109.865	105.407
Explosive*	97.214	105.344
Explosive	95.263	97.877
Iwahashi*	89.748	92.288
Sweldens	82.336	88.924
Iwahashi	80.284	80.283
Polyphase*	51.776	43.619
Polyphase	32.593	27.462

Table 8.6: The degree-1 schemes on AMD 6970 and AMD 5870. The performance of a transform code without the memory throughput is listed. Values given in GB/s at the end of plots in Fig. 8.18.

	AMD 5870			AMD 6970		
scheme	CDF 5/3	CDF 9/7	DD 13/7	CDF 5/3	CDF 9/7	DD 13/7
Sweldens	32.59	32.69	40.00	40.53	40.25	48.50
Iwahashi	34.88	35.90	50.12	41.33	42.33	61.80
Iwahashi*	32.00	34.29	45.83	42.11	40.75	57.00
Explosive	36.88	39.32	49.57	45.14	46.64	60.47
Explosive*	33.79	33.33	55.56	42.42	42.21	65.42
Monolithic	38.18	39.67	51.59	48.57	47.76	64.44
Monolithic*	38.62	37.36	55.79	48.39	44.58	69.49
Polyphase	43.44	43.49	37.76	52.57	54.30	47.21
Polyphase*	31.50	32.43	33.38	41.88	40.58	41.91
Convolution	—	73.79	—	—	83.95	—

Table 8.7: ALU packing percentage for AMD 6970 and AMD 5870.

to the number of memory barriers. Several schemes perform even worse than the original separable Sweldens scheme – namely, the original Iwahashi and both Polyphase schemes. It is not surprising for the original Polyphase scheme, as this one exhibits quite a high number of operations and load instructions (see Table 8.4 and Table 8.2). In case of Polyphase* scheme, the decisive factor was the number of load instructions coupled with a high local memory footprint (see Table 8.3). A little surprising is the situation regarding the original Iwahashi scheme. In this case, the scheme contains a relatively high number of operations, wherein there is no additional advantage. For convenience, the values at the end of plots in Fig. 8.18 are listed in Table 8.6.

In the last experiment, we were interested in a real performance. This experiment was performed on both of the cards with CDF 9/7 wavelet. The input as well as output raster were supplied by the texture memory. This time, we show only the improved schemes as these always outperform the original ones. The results are shown in Fig. 8.19. The horizontal axes are in a logarithmic scale, and the vertical ones express the total throughput (limited by the memory). The Convolution and Polyphase schemes exhibit a significantly worse performance, according to the number of operations. In contrast to this, the other

	AMD 5870			AMD 6970		
scheme	CDF 5/3	CDF 9/7	DD 13/7	CDF 5/3	CDF 9/7	DD 13/7
Sweldens	100.00	100.00	100.00	95.24	95.24	95.24
Iwahashi	100.00	100.00	100.00	95.24	95.24	95.24
Iwahashi*	100.00	* 83.33	100.00	95.24	95.24	95.24
Explosive	100.00	100.00	100.00	95.24	95.24	95.24
Explosive*	100.00	100.00	100.00	95.24	95.24	95.24
Monolithic	* 83.33	* 83.33	* 83.33	95.24	95.24	95.24
Monolithic*	* 83.33	* 83.33	* 83.33	95.24	95.24	95.24
Polyphase	** 83.33	* 50.00	100.00	95.24	* 57.14	95.24
Polyphase*	100.00	* 50.00	100.00	95.24	* 57.14	95.24
Convolution	—	100.00	—	—	95.24	—

Table 8.8: GPU occupancy measurement for AMD 6970 and AMD 5870. The numbers indicate a percentage. Explanation: * is limited by a local memory (LDS), ** by registers (VGPR).

schemes perform better as compared to the original separable implementation. More specifically, the Monolithic and Explosive schemes have the very best performance. This fact corresponds to the reduction of the number of steps (and thus the memory barriers).

The schemes presented in this paper were also subject of examination at other graphics cards under various scenarios. Note that a link to the results is below. Specifically, we tackled these additional cards – NVIDIA Titan X, AMD Fury X, NVIDIA 580, and AMD 290X. Obviously, the proposed non-separable schemes presented in this paper do not exhibit the best performance in all cases. This is especially true for a lifting factorizations employing a longer convolution operators, as is the case of the DD 13/7 wavelet. On the other hand, the proposed schemes seems to be the proven choice for VLIW architectures combined with a short lifting operators, e.g., the CDF 5/3 and CDF 9/7 wavelets.

This point needs to be explained in detail. In general, the number of memory accesses, instruction dependencies, as well as barriers, decreases the ALU utilization, which then degrades the performance. Unlike other architectures, AMD VLIW architectures pack multiple independent instructions into VLIW bundles. Thus, amount and dependencies of instructions between each two neighboring barriers play a significant role in terms of performance. In other words, the number of barriers in VLIW architectures plays a stronger role than in other architectures. Indeed, on the AMD VLIW architectures, code profiling showed that memory barriers limit an average length of VLIW instructions (ALU packing percentage in Table 8.7), which degrades the performance. The ALU packing percentage refers to the percent of cores in the VLIW processor that are being utilized. On the other (non-VLIW) architectures, the number of local memory accesses (see Table 8.4) and the number of arithmetic instructions (see Table 8.2) play a major role. On such architectures, the ALU packing is not measurable due to an absence of the VLIW bundles. For comparative purposes, the ALU packing percentage can be understood as 100 % for all schemes on these architectures.

It should also be interesting to show another measure provided by an OpenCL profiler. In the first instance, consider AMD 5870 card. Such implementations, in which threads need to store less than 5 coefficients (20 bytes) into a local memory, exhibit an occupancy 100 %, as can be seen in Table 8.8. In detail, 256 threads in work group $\times$ 6 work groups

result in occupancy 1536 of 1536 threads. This is valid for all these implementations with the exception of Polyphase scheme, in which the occupancy is limited by the number of vector registers, due to an optimizing compiler. For AMD 6970, due to the use of 256 threads in work-groups and due to maximal number 1344 of threads in multiprocesors, implementations exhibit only an occupancy 95.24 % (256 threads in work group $\times$ 5 work groups = 1280 of 1344). On the other hand, considering implementations in which threads need to store less than 7 coefficients (28 bytes) into a local memory, the occupancy is not limited by a size of a local memory.

In summary, we can conclude that the reduction in lifting steps can improve performance, at least on some platforms. This is documented by measurements in Figs. 8.17, 8.18, and 8.19. It turned out, however, that such optimization makes sense only for a short lifting operators (exemplary, degree-1 lifting filters).

For the sake of completeness, it should be noted that the improvement proposed in Section Improvements can be also applied on the Convolution. Doing so, the scheme achieves a slightly better performance. However, we understand the Convolution scheme as the reference method. For this reason, we leave it unimproved. Eventually, the proposed improvement makes no sense in conjunction with the Sweldens scheme.

All the source codes used in this article together with all the results are available in a repository on the website of the authors' affiliation.[1]

8.7 Conclusions

In this paper, we have proposed several non-separable lifting schemes for the calculation of the discrete wavelet transform. The proposed schemes produce exactly the same results as the commonly used separable lifting scheme. Using our schemes, the transform can be computed in a smaller number of steps. On parallel architectures, this property has resulted in a smaller number of synchronizations.

Namely, we have proposed two-step 2-D lifting scheme compatible to the commonly used four-step separable one. Unlike the separable scheme, the proposed scheme consists of spatial predict and update operators. Since the number of the lifting steps was halved, our scheme reduces also the number of memory barriers, which form a major bottleneck on parallel architectures. In addition, we have proposed the three-step scheme reducing the memory access overhead. For a moment, let K denote the number of predict-update pairs. In absolute numbers, the original separable scheme requires to write $1 + 4K$ coefficients per predict/update pair, whereas our three-step scheme requires $4K$ coefficients only. Additionally, the proposed two-step scheme requires three memory cells per thread, whereas the proposed three-step scheme requires two cells only (same as the separable scheme). Finally, we have proposed an improvement usable for all non-separable scheme, including the already known ones. This improvement significantly reduces the number of arithmetic operations. More specifically, the original non-separable schemes require 24 arithmetic operations for CDF 5/3 wavelet, whereas the improved variants require 18 operations only for the same case. Even greater savings are achieved in the case of a non-factorized polyphase matrix (same as the convolution for the CDF 5/3 wavelet). In this case, the proposed improvement reduces the number of operations from 63 to 23. All of the proposed schemes are general and can be used in conjunction with any discrete wavelet transform.

[1] http://www.fit.vutbr.cz/research/prod/?id=483

The proposed schemes were subject to performance measurements. In experiments on the two selected high-end GPUs (AMD Radeon HD 6970 and 5870), the proposed schemes outperform all the others for short lifting filters. This includes the well-known CDF 5/3 and CDF 9/7 wavelets, employed, e.g., in JPEG 2000 compression standard.

Future work, we would like to do, consists of extensions to multi-dimensional systems, and extensions to another subband transforms.

8.8 Acknowledgements

This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science – LQ1602.

Chapter 9

Article: Accelerating Discrete Wavelet Transforms on Parallel Architectures

This paper follows the previous one [16] that introduces the separable convolution scheme improved by the optimization approach introduced in the section 8.3.2. This separable convolution scheme is compared to the non-separable lifting and separable lifting$^+$ scheme.

Moreover, that paper is focused on implementation using pixel pipeline, which is suitable for GPU architectures without unified architecture or systems/programs that are not capable of using GPGPU language for some reason. The pixel shader implementation, using multiple passes through the graphics pipeline for calculating 2D DWT, is created for that case. The disadvantage of this approach lies in the absence of user-defined local memory and communication between threads during shader calculation. This leads to mapping of every step of the calculation to the separate pass through the pipeline with loading/storing of intermediate results from/to global memory. In that case, the barrier is much more costly, so the schemes with a low number of barriers become more desirable. As a consequence, no implicit barrier for syncing data after transfer from global to local memory is applied, so predict part of the optimization approach cannot be applied before the first step of the calculation. This fact leads to creating new variants of schemes that are suitable for graphics pipeline processing. Such schemes are compared in Table 11.2.

Paper [16] is omitted because all schemes evaluated in that paper are presented in paper included in this chapter as well.

The paper presented in this chapter extends the previously mentioned one by introducing the improved variant of non-separable polyconvolution sceme and improved variant of separable convolution scheme for both paradigms, the GPGPU one using OpenCL and graphics pipeline one using the Pixel shader. All tested schemes and their properties are described in Table 11.1.

Finally, all improved variant of the mentioned schemes are compared and tested on CDF 5/3 ($K = 1$, $D = 1$), CDF 9/7 ($K = 2$, $D = 1$) and DD 13/7 ($K = 1$, $D = 3$) wavelets on various GPUs. All tested schemes and their properties are described in Table 11.1.

The evaluation shows that the non-separable polyconvolution scheme (partially fused separable lifting scheme) combined with the optimization approach outperforms the previously leading optimized non-separable lifting scheme further by 25% for 1-degreed wavelets, but optimized non-separable lifting scheme still performs better on 3-degreed wavelet on

the tested platforms using GPGPU paradigm. Due to the high impact of barriers, using the optimized non-separable convolution scheme in the graphical pipeline paradigm for 1-degreed wavelets is beneficial in most cases. Optimized separable convolution scheme performs slightly better only for resolution $<$ 2MPix. For 3-degreed wavelets, the optimized separable convolution is about 10% better than the second-best one, the optimized non-separable lifting scheme.

The GPGPU paradigm approach uses our Wavelet Transform Framework and second version of kernel generator described in the section 5.2.

Accelerating Discrete Wavelet Transforms on Parallel Architectures

Barina, D., Kula, M., Matysek, M. and Zemcik, P. Accelerating Discrete Wavelet Transforms on Parallel Architectures. *Journal of WSCG*. Union Agency. 2017, vol. 25, no. 2, p. 77–85. ISSN 1213-6972

Author contribution: 25%

Abstract

The 2-D discrete wavelet transform (DWT) can be found in the heart of many image-processing algorithms. Until recently, several studies have compared the performance of such transform on various shared-memory parallel architectures, especially on graphics processing units (GPUs). All these studies, however, considered only separable calculation schemes. We show that corresponding separable parts can be merged into non-separable units, which halves the number of steps. In addition, we introduce an optional optimization approach leading to a reduction in the number of arithmetic operations. The discussed schemes were adapted on the OpenCL framework and pixel shaders, and then evaluated using GPUs of two biggest vendors. We demonstrate the performance of the proposed non-separable methods by comparison with existing separable schemes. The non-separable schemes outperform their separable counterparts on numerous setups, especially considering the pixel shaders.

9.1 Introduction

The discrete wavelet transform became a very popular image processing tool in last decades. A widespread use of this transform has resulted in a development of fast algorithms on all sorts of computer systems, including shared-memory parallel architectures. At present, the GPU is considered as a typical representative of such parallel architectures. In this regard, several studies have compared the performance of various 2-D DWT computational approaches on GPUs. All of these studies are based on separable schemes, whose operations are oriented either horizontally or vertically. These schemes comprise the convolution and lifting. The lifting requires fewer arithmetic operations as compared with the convolution, at the cost of introducing some data dependencies. The number of operations should be

proportional to a transform performance. However, also the data dependencies may form a bottleneck, especially on shared-memory parallel architectures.

In this paper, we show that the fastest scheme for a given architecture can be obtained by fusing the corresponding parts of the separable schemes into new structures. Several new non-separable schemes are obtained in this way. More precisely, the underlying operations of these schemes can be associated with neither horizontal nor vertical axes. In addition, we present an approach where each scheme can be adapted to a particular platform in order to reduce the number of operations. This possibility was completely omitted in existing studies. Our reasoning is supported by extensive experiments on GPUs using OpenCL and pixel shaders (fragment shaders in OpenGL terminology). The presented schemes are general, and they are not limited to any specific type of DWT. To clarify the situation, they all compute the same values.

The rest of this paper is organized as follows. Section Background formally introduces the problem definition. Section Related Work briefly presents the existing separable approaches. Section Proposed Schemes presents the proposed non-separable schemes. Section Optimization Approach discusses the optimization approach that reduces the number of operations. Section Evaluation evaluates the performance on GPUs in the pixel shaders and OpenCL framework. Eventually, Section Conclusions closes the paper. This section is followed by Section Appendix for readers not familiar with signal-processing notations.

9.2 Background

Since the separable schemes are built on the one-dimensional transform, a widely-used z-transform is used for the description of underlying FIR filters. The transfer function of the filter (g_k) is the polynomial

$$G(z) = \sum_k g_k \, z^{-k},$$

where the k refers to the time axis. Below in the text, the one-dimensional transforms are used in conjunction with two-dimensional signals. For this case, the transfer function of the filter (g_{k_m,k_n}) is defined as the bivariate polynomial

$$G(z_m, z_n) = \sum_{k_m} \sum_{k_n} g_{k_m,k_n} \, z_m^{-k_m} \, z_n^{-k_n},$$

where the subscript m refers to the horizontal axis and n to the vertical one. The $G^*(z_m, z_n) = G(z_n, z_m)$ is a polynomial transposed to a polynomial $G(z_m, z_n)$. A shortened notation G is only written in order to keep the notation readable.

A discrete wavelet transform is a signal-processing tool which is suitable for the decomposition of a signal into low-pass and high-pass components. In detail, the single-scale transform splits the input signal into two components, according to a parity of its samples. Therefore, the DWT is described by 2×2 matrices. As shown by Mallat [55], the transform can be computed by a pair of filters followed by subsampling by a factor of 2. The filters are referred to as $\mathrm{G}_0, \mathrm{G}_1$. The transform can also be represented by the polyphase matrix

$$\begin{bmatrix} \mathrm{G}_1^{(o)} & \mathrm{G}_1^{(e)} \\ \mathrm{G}_0^{(o)} & \mathrm{G}_0^{(e)} \end{bmatrix}, \tag{9.1}$$

where the polynomials $G^{(e)}$ and $G^{(o)}$ refer to the even and odd terms of G. This polyphase matrix defines the convolution scheme. To avoid misunderstandings, it is necessary to say that, in this paper, column vectors are transformed to become another columns. For example, $y = Mx$ and $y = M_2M_1x$ are transforms represented by one and two matrices, respectively. Following the algorithm by Sweldens [72, 31], the convolution scheme in (9.1) can be factored into a sequence

$$\prod_k \begin{bmatrix} 1 & U^{(k)} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ P^{(k)} & 1 \end{bmatrix} \tag{9.2}$$

of K pairs of short filterings, known as the lifting scheme. The filters employed in (9.2) are referred to as the lifting steps. Usually, the first step $P^{(k)}$ in the kth pair is referred to as the predict and the second one $U^{(k)}$ as the update. The lifting scheme reduces the number of operations by up to half. Since this paper is mostly focused on a single pair of steps, the superscript (k) is omitted in the text below. Note that the number of operations is calculated as the number of distinct (in a column) terms of all polynomials in all matrices, excluding units on diagonals.

Considering the shared-memory parallel architectures, the processing of single or several samples is mapped to independent processing units. In order to avoid race conditions during data exchange, the units must use some synchronization method (barrier). In the lifting scheme, the barriers are required before the lifting steps. In the convolution scheme, the barrier is only required before starting the calculation. In this paper, the barriers are indicated by the | symbol. For example, $M_2|M_1$ are two adjacent lifting steps separated by the barrier. For simplicity, the number of barriers is also called the number of steps in the text below.

The 2-D transform is defined as a tensor product of 1-D transforms. Consequently, the transform splits the signal into a quadruple of wavelet coefficients. Therefore, the 2-D DWT is described by 4×4 matrices. See Section Appendix for details. Following the pioneering paper of Mallat [55], the 1-D transforms are applied in both directions sequentially. By its nature, this scheme can be referred to as the separable convolution. The calculations in a single direction are performed in a single step. This means two steps for the two dimensions. The scheme can formally be described as

$$\mathbf{N}^V \,|\, \mathbf{N}^H \,|,$$

where $\mathbf{N}^H$ and $\mathbf{N}^V$ are 1-D transforms in horizontal and in vertical direction. For the well-known Cohen-Daubechies-Feauveau (CDF) wavelet with 9/7 samples, such as used in the JPEG 2000 standard, these matrices are graphically illustrated in Figure 9.1. Here, only the horizontal part is shown. Particularly, the filters in the figure are of sizes 9 and 7 taps. The ●, ●, ●, and ● circles represent the quadruple of wavelet coefficients. Figures shown are for illustration purpose only.

Another scheme used for 2-D transform is the separable lifting. Similarly to the previous case, the predict and update lifting steps can be applied in both directions sequentially. Moreover, horizontal and vertical steps can be arbitrarily interleaved thanks to the linear nature of the filters. Therefore, the scheme is defined as

$$S_U^V \,|\, S_U^H \,|\, T_P^V \,|\, T_P^H \,|,$$

wherein the predict steps T always precede the update steps S. The above mapping corresponds to a single P and U pair of lifting steps. For multiple pairs, the scheme is separately

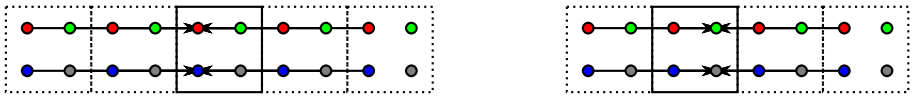


Figure 9.1: Horizontal part of the separable convolution scheme for the CDF 9/7 wavelet. Two appropriately chosen pairs of matrix rows are depicted in separate subfigures. The arrows are pointing to the destination operand and denote a multiply–accumulate operation, with multiplication by a real constant. The arrows in the same row overlap.

applied to each such pair. In order to describe 2-D matrices, the lifting steps must be extended into two dimensions as

$$\begin{bmatrix} \mathrm{G} \\ \mathrm{G}^* \end{bmatrix} = \begin{bmatrix} G\ (z_m, z_n) \\ G^*(z_m, z_n) \end{bmatrix} = \begin{bmatrix} G(z_m) \\ G(z_n) \end{bmatrix}.$$

Then, the individual steps are defined as

$$\mathrm{T}_\mathrm{P}^H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathrm{P} & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \mathrm{P} & 1 \end{bmatrix},$$

$$\mathrm{T}_\mathrm{P}^V = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \mathrm{P}^* & 0 & 1 & 0 \\ 0 & \mathrm{P}^* & 0 & 1 \end{bmatrix},$$

$$\mathrm{S}_\mathrm{U}^H = \begin{bmatrix} 1 & \mathrm{U} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \mathrm{U} \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\mathrm{S}_\mathrm{U}^V = \begin{bmatrix} 1 & 0 & \mathrm{U}^* & 0 \\ 0 & 1 & 0 & \mathrm{U}^* \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

For the CDF wavelets, the matrices are also illustrated in Figure 9.2, again showing the horizontal part only.

(a) T_P^H (b) S_U^H

Figure 9.2: The horizontal part of the separable lifting scheme for the CDF wavelets.

9.3 Related Work

This section briefly reviews papers that motivated our research. So far, several papers have compared the performance of the separable lifting and separable convolution schemes on GPUs. Especially, Tenllado *et al.* [75] compared these schemes on GPUs using pixel shaders. The authors mapped data to 2-D textures, constituted by four floating-point elements. They concluded that the separable convolution is more efficient than the separable lifting scheme in most cases. They further noted that fusing several consecutive kernels might significantly speed up the execution, even if the complexity of the resulting fused pixel program is higher.

Kucis *et al.* [48] compared the performance of several recently published schedules for computing the 2-D DWT using the OpenCL framework. All of these schedules use separable schemes, either the convolution or lifting. In more detail, the work compares a convolution-based algorithm proposed in [39] against several lifting-based methods [15, 53] in the horizontal part of the transform. The authors concluded that the lifting-based algorithm of Blazewicz *et al.* [15] is the fastest method. Furthermore, Laan *et al.* [53] compared the performance of their separable lifting-based method against a convolution-based method. They concluded that the lifting is the fastest method. The authors also compared the performance of implementations in CUDA and pixel shaders, based on the work of Tenllado [75]. The CUDA implementation proved to be the faster choice. In this regard, the authors noted that a speedup in CUDA occurs because the CUDA effectively makes use of on-chip memory. This use is not possible in pixel shaders, which exchange the data using off-chip memory. Other important separable approaches can be found in [57, 40, 69, 61].

This paper is based on the previous works in [22, 49]. In those works, we introduced several non-separable schemes for calculation of 2-D DWT. However, we have not considered important structures, such as polyconvolutions. We contribute this consideration with this paper. Moreover, differences and similarities between the separable schemes and their non-separable counterparts are homogeneously discussed here. All these schemes are also thoroughly analyzed and evaluated.

Considering the present papers, we see that a possible fusion of separable parts into new non-separable structures is not considered. Therefore, we investigate on this promising technique in the following sections.

9.4 Proposed Schemes

As stated above, the existing approaches did not study the possibility of a partial fusion of lifting polyphase matrices. This section presents three alternative non-separable schemes for the calculation of the 2-D transform. The contribution of this paper starts with this section. To avoid confusion, please note that the proposed schemes compute the same values as the original ones.

The non-separable convolution scheme is a counterpart to the separable convolution. Unlike the separable scheme, all horizontal and vertical calculations are performed in a single step

$$\mathbf{N}\,\Big|,$$

where $\mathbf{N} = \mathbf{N}^V\mathbf{N}^H$ is a product of 1-D transforms in horizonal and vertical directions. The drawback of this scheme is that it requires the highest number of arithmetic operations. For the CDF 9/7 wavelet, the matrix is graphically illustrated in Figure 9.3. Here, the

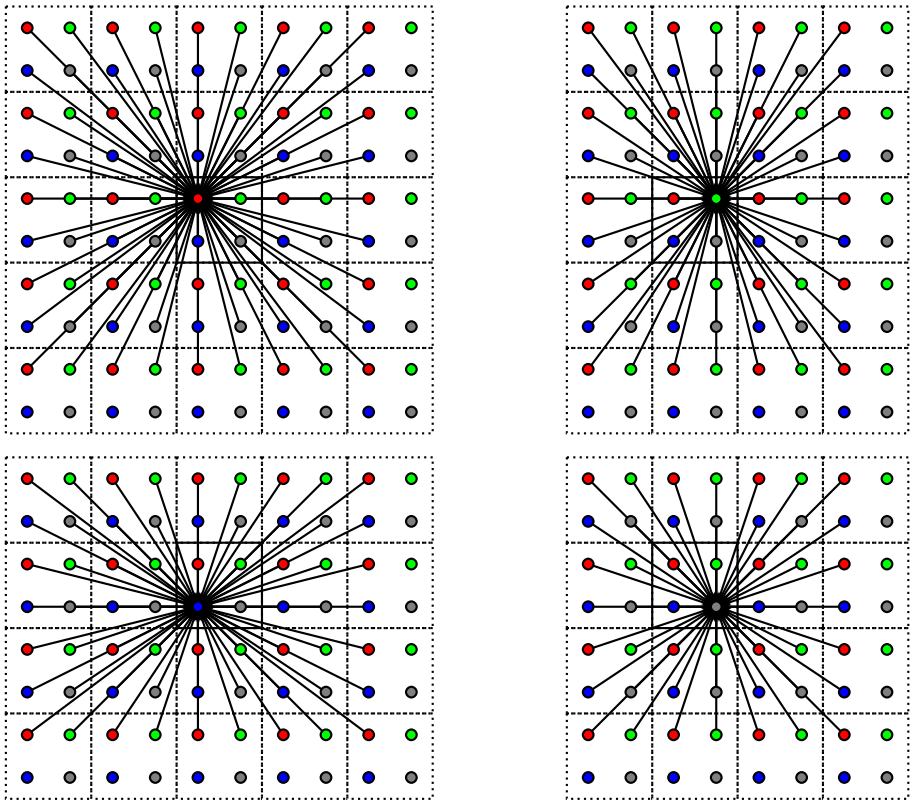


Figure 9.3: Non-separable convolution scheme for the CDF 9/7 wavelet. The individual rows of N are depicted in separate subfigures. The sizes are from top to bottom and left to right: 9×9, 7×9, 9×7, 7×7.

2-D filters are of sizes 9×9, 7×9, 9×7, and 7×7. These sizes make the calculation computationally demanding. Aside from the GPUs, this approach was earlier discussed in Hsia *et al.* [43].

In order to reduce computational complexity, it would be a good idea to construct some smaller non-separable steps. Indeed, the non-separable convolution can be broken into smaller units, referred here to as the non-separable polyconvolutions. For a single pair of lifting steps, the scheme follows from the mapping

$$\mathrm{N}_{\mathrm{P,U}} \,\big|,$$

where

$$\mathrm{N}_{\mathrm{P,U}} = \begin{bmatrix} \mathrm{V^*V} & \mathrm{V^*U} & \mathrm{U^*V} & \mathrm{U^*U} \\ \mathrm{V^*P} & \mathrm{V^*} & \mathrm{U^*P} & \mathrm{U^*} \\ \mathrm{P^*V} & \mathrm{P^*U} & \mathrm{V} & \mathrm{U} \\ \mathrm{P^*P} & \mathrm{P^*} & \mathrm{P} & 1 \end{bmatrix}$$

and $\mathrm{V} = \mathrm{PU} + 1$. For the CDF wavelets, the scheme is graphically illustrated in Figure 9.4. In this case, the employed filters are of sizes 5×5, 3×5, 5×3, and 3×3. Note that only half of the operations are required specifically for the CDF 9/7 wavelet, compared to the non-separable convolution. For the sake of completeness, it should be pointed out that it is

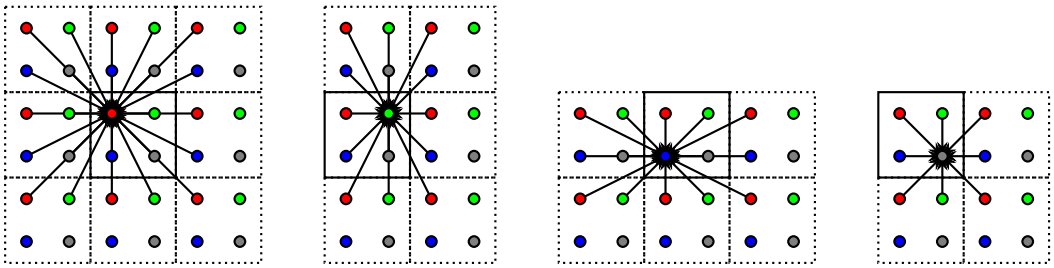


Figure 9.4: Non-separable polyconvolution scheme for the CDF wavelets. The individual rows of N are depicted in separate subfigures.

also possible to formulate the separable polyconvolution scheme. In our experiments, this one was however not proven to be useful concerning the performance.

By combining the corresponding horizontal and vertical steps of the separable lifting scheme, the non-separable lifting scheme is formed. The number of operations has slightly been increased. The scheme consists of a spatial predict and spatial update step, thus two steps in total for each pair of the original lifting steps. Formally, for each pair of P and U, the scheme follows from

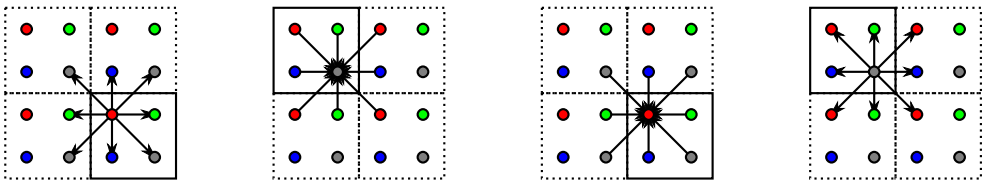
$$\mathrm{S_U} \left| \, \mathrm{T_P} \, \right|,$$

where

$$\mathrm{T_P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathrm{P} & 1 & 0 & 0 \\ \mathrm{P}^* & 0 & 1 & 0 \\ \mathrm{PP}^* & \mathrm{P}^* & \mathrm{P} & 1 \end{bmatrix},$$

$$\mathrm{S_U} = \begin{bmatrix} 1 & \mathrm{U} & \mathrm{U}^* & \mathrm{UU}^* \\ 0 & 1 & 0 & \mathrm{U}^* \\ 0 & 0 & 1 & \mathrm{U} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Note that the spatial filters in PP^* and UU^* may be computationally demanding, depending on their sizes. However, the situation is always better than in the previous two cases. For the CDF wavelets, the scheme is graphically illustrated in Figure 9.5.



(a) $\mathrm{T_P}$ (b) $\mathrm{T_P}$ (c) $\mathrm{S_U}$ (d) $\mathrm{S_U}$

Figure 9.5: Non-separable lifting scheme for the CDF wavelets.

9.5 Optimization Approach

This section presents an optimization approach that reduces the number of operations, while the number of steps remains unaffected. Such an approach was not covered in existing studies.

Regardless of the underlying platform, an important observation can be made. A very special form of the operations guarantees that the processing units never access the results belonging to their neighbors. These operations comprise only constants. Since the convolution is a linear operation, the polynomials can be pulled out of the original matrices, and calculated in a different step. Formally, the original polynomials are split as $\mathrm{P} = \mathrm{P}_0 + \mathrm{P}_1$ and $\mathrm{U} = \mathrm{U}_0 + \mathrm{U}_1$. The P_0 and U_0 are constant. As a next step, the P_0 and U_0 are substituted into the separable lifting scheme. The separable lifting scheme was chosen because it has the lowest number of operations. This part is illustrated in Figure 9.6. In contrast, the P_1 and U_1 are kept in original schemes. These two steps are then computed without any barrier. The observation is further exploited to adapt schemes for a particular platform.

(a) $\mathrm{T}^H_{\mathrm{P}_0}$ (b) $\mathrm{T}^V_{\mathrm{P}_0}$ (c) $\mathrm{S}^H_{\mathrm{U}_0}$ (d) $\mathrm{S}^V_{\mathrm{U}_0}$

Figure 9.6: Separable lifting scheme with the polynomials P_0 and U_0.

Now, the improved schemes for the shaders and OpenCL are briefly described. These schemes exploit the above-described observation with the polynomials P_0 and U_0. On recent GPUs, OpenCL schemes also omit memory barriers due to the SIMD-32 architecture. Note that the non-separable polyconvolution scheme makes sense only when $K > 1$, which is the case of the CDF 9/7 wavelet. Implementations in the pixel shaders map input and output data to 2-D textures. There is no possibility to retain some results in registers, and the results are exchanged through textures in off-chip memory. Considering the OpenCL implementations, a data format is not constrained. The image is divided into overlapping blocks and on-chip memory shared by all threads in a block is utilized to exchange the results. Additionally, some results are passed in registers.

This paper explores the performance for three frequently used wavelets, namely, CDF 5/3, CDF 9/7 [30], and DD 13/7 [72]. Their fundamental properties are listed in Table 9.1: number of steps and arithmetic operations. Note that the number of operations is commonly proportional to a transform performance. Additionally, the number of steps correspond to the number of synchronizations on parallel architectures, which also form a performance bottleneck.

Table 9.1: The total number of steps and arithmetic operations for the optimized schemes.

(a) CDF 5/3

scheme		steps	operations	
			OpenCL	shaders
separable	convolution	2	20	22
separable	lifting	4	16	16
non-separable	convolution	1	23	39
non-separable	lifting	2	18	18

(b) CDF 9/7

scheme		steps	operations	
			OpenCL	shaders
separable	convolution	2	56	58
separable	polyconv.	4	40	56
separable	lifting	8	32	32
non-separable	convolution	1	152	200
non-separable	polyconv.	2	46	62
non-separable	lifting	4	36	36

(c) DD 13/7

scheme		steps	operations	
			OpenCL	shaders
separable	convolution	2	60	60
separable	lifting	4	32	32
non-separable	convolution	1	203	228
non-separable	lifting	2	50	50

9.6 Evaluation

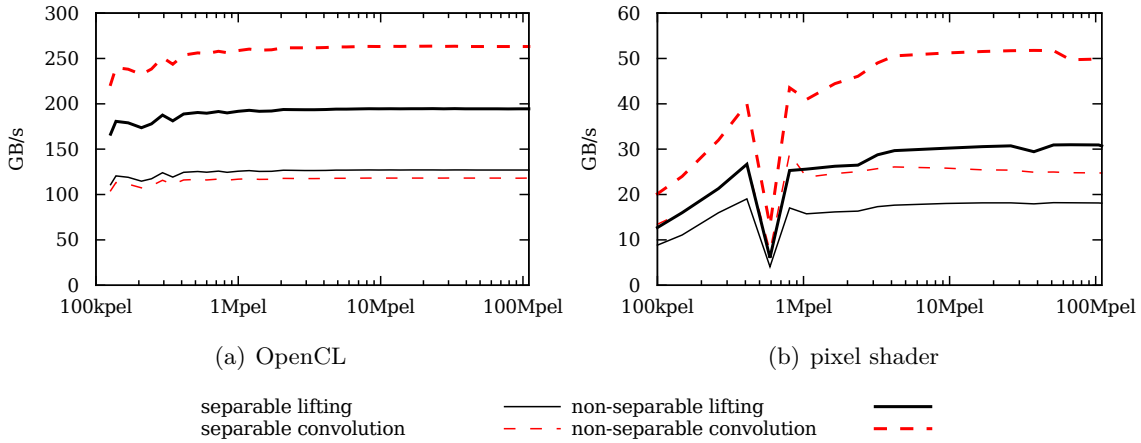
The experiments in this paper were performed on GPUs of the two biggest vendors NVIDIA and AMD using the OpenCL and pixel shaders. In these experiments, only a transform performance was measured, usually in gigabytes per second (GB/s). The host system does not help in the calculation, i.e. with respect to padding or pre/post-processing. Results for only two GPUs are shown for the sake of brevity: AMD Radeon HD 6970 and NVIDIA Titan X. Their technical parameters are summarized in Table 9.2.

Table 9.2: Specifications of the evaluated GPUs.

label	AMD 6970	NVIDIA Titan X
model	Radeon HD 6970	Titan X (Pascal)
multiprocessors	24	28
total processors	1 536	3 584
processor clock	880 MHz	1 417 MHz
performance	2 703 GFLOPS	10 157 GFLOPS
memory clock	1 375 MHz	2 500 MHz
bandwidth	176 GB/s	480 GB/s
on-chip memory	32 KiB	96 KiB

The implementations were created using the DirectX HLSL and OpenCL. The HLSL implementation is used on the NVIDIA Titan X, whereas the OpenCL implementation on the AMD 6970. The results of the performance comparison are shown in Figures 9.7, 9.8, and 9.9. The value on the x-axis is the image resolution in kilo/megapixels (kpel or Mpel). Except for the convolutions for the DD 13/7 wavelet, the non-separable schemes always outperform their separable counterparts. For CDF wavelets, having short lifting filters, the non-separable (poly)convolutions have a better performance than the non-separable lifting scheme. Unfortunately, for the DD 13/7 wavelet, which is characterized by a high number of operations in lifting filters, the results are not conclusive. Considering the implementation in pixel shaders, similar results were also achieved on other GPUs, including NVIDIA unified architectures and AMD GPUs based on Graphics Core Next (GCN) architecture. Whereas for the OpenCL implementation, the non-separable schemes are only proved to be useful for very long instruction word (VLIW) architectures.

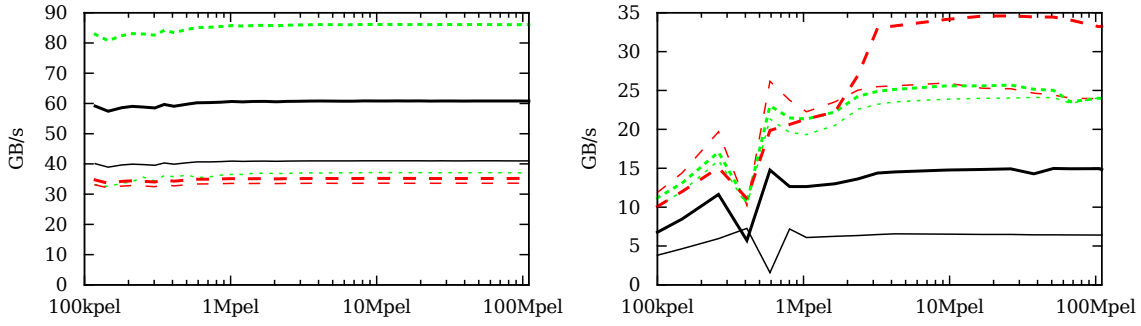
Looking at the experiments with the pixel-shader implementations, some transients can be seen at the beginning of the plots (in lower 2 Mpel region). We concluded that these transients are caused by a suboptimal use of cache system, or alternatively by some overhead made by the graphics API. It should be interesting to show some measures provided by an OpenCL profiler. Our profiling revealed that the implementations exhibit only an occupancy 95.24 %. This occupancy is caused by making use of 256 threads in OpenCL work groups and due to maximal number 1344 of threads in multiprocessors (256 times 5 work groups is 1280 out of 1344).



(a) OpenCL (b) pixel shader

separable lifting
separable convolution
non-separable lifting
non-separable convolution

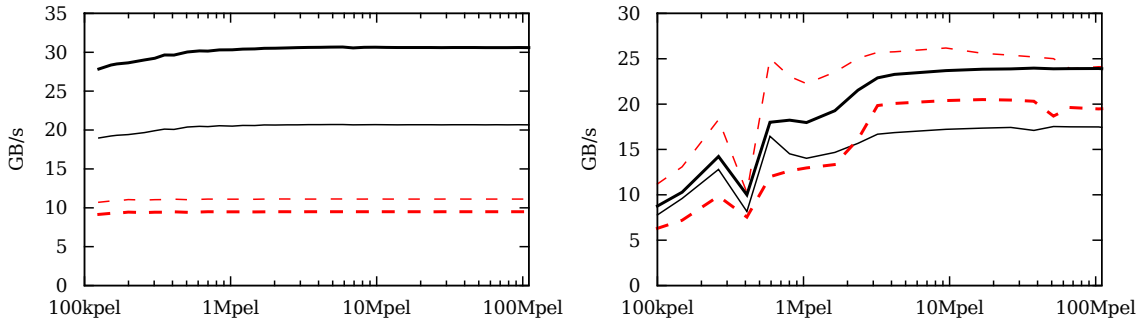
Figure 9.7: Performance for the CDF 5/3 wavelet.



(a) OpenCL (b) pixel shader

separable lifting
separable polyconvolution
separable convolution
non-separable lifting
non-separable polyconvolution
non-separable convolution

Figure 9.8: Performance for the CDF 9/7 wavelet.



(a) OpenCL (b) pixel shader

separable lifting
separable convolution
non-separable lifting
non-separable convolution

Figure 9.9: Performance for the DD 13/7 wavelet.

9.7 Conclusions

This paper presented and discussed several non-separable schemes for the computation of the 2-D discrete wavelet transform on parallel architectures, exemplarily on modern GPUs. As an option, an optimization approach leading to a reduction in the number of operations was presented. Using this approach, the schemes were adapted on the OpenCL framework and pixel shaders. The implementations were then evaluated using GPUs of the two biggest vendors. Considering OpenCL, the schemes exploit features of recent GPUs, such as warping. For CDF wavelets, the non-separable schemes exhibit a better performance than their separable counterparts on both the OpenCL and pixel shaders.

In the evaluation, we reached the following conclusions. Fusing several consecutive steps of the schemes might significantly speed up the execution, irrespective of their higher complexity. The non-separable schemes outperform their separable counterparts on numerous setups, especially considering the pixel shaders. All of the schemes are general and they can be used on any discrete wavelet transform. In future work, we plan to focus on general-purpose processors and multi-scale transforms.

Acknowledgements This work has been supported by the Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science (no. LQ1602), and the Technology Agency of the Czech Republic (TA CR) Competence Centres project V3C – Visual Computing Competence Center (no. TE01020415).

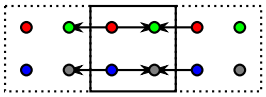
Appendix

For readers who are not familiar with signal-processing notations, a relationship between polyphase matrices and data-flow diagrams is explained here. The 2-D discrete wavelet transform divides the image into four polyphase components. Therefore, the 4×4 matrices of Laurent polynomials are used to describe the 2-D discrete wavelet transform. These matrices are commonly referred to as the polyphase matrices. The Laurent polynomials correspond to 2-D FIR filters, that define the transform. In most cases, the transform is described using a sequence of such matrices. One particular matrix thus defines a step of calculation in this case.

For example, the matrix

$$\mathrm{T}_{\mathrm{P}}^{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \mathrm{P} & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \mathrm{P} & 1 \end{bmatrix}$$

maps four polyphase components to another four components, while using two 2-D FIR filters represented by the polynomials P. Moreover, when we substitute a particular polynomial, say $P(z) = -1/2(1 + z^{-1})$, into the matrix, the mapping gets a specific shape. Such a substitution illustrated by the data-flow diagram in Figure 9.10. The solid arrows correspond to multiplication by $-1/2$ along with subsequent summation.



(a) $\mathrm{T}_{\mathrm{P}}^{H}$

Figure 9.10: Visual representation of the polyphase matrix. The four polyphase components are represented by color circles.

Chapter 10

Article: The Parallel Algorithm for the 2-D Discrete Wavelet Transform

In contrast with previously mentioned papers focused on acceleration on GPU architectures, this one is focused on acceleration on multicore CPUs. In this paper, the separable lifting$^+$ and non-separable lifting schemes presented in the section 8.3.2 are adopted to CPU multicore platforms using the OpenMP paradigm.

Finally, the schemes are evaluated on CDF 5/3 ($K = 1$, $D = 1$) wavelet using multicore Xeon CPU and Xeon Phi accelerator on various tile sizes, image sizes, and thread counts. The evaluation shows that the novel spatial 2D scheme for the calculation of 2D DWT presented in [49] improves throughput compared to the separable lifting$^+$ scheme variant by 10%. Note that implementation is not using any SIMD extensions in any of those schemes.

This version is based on the hand-written implementation of these schemes and is not using our generator from the section 5.2.

The Parallel Algorithm for the 2-D Discrete Wavelet Transform

Barina, D., Najman, P., Kleparnik, P., Kula, M. and Zemcik, P. The Parallel Algorithm for the 2D Discrete Wavelet Transform. In: *Ninth International Conference on Graphic and Image Processing (ICGIP 2017)*. SPIE - the international society for optics and photonics, 2017, vol. 10615, no. 4, p. 1–6. DOI: 10.1117/12.2302881. ISBN 978-1-5106-1741-4

Author contribution: 20%

Abstract

The discrete wavelet transform can be found at the heart of many image-processing algorithms. Until now, the transform on general-purpose processors (CPUs) was mostly computed using a separable lifting scheme. As the lifting scheme consists of a small number of operations, it is preferred for processing using single-core CPUs. However, considering a parallel processing using multi-core processors, this scheme is inappropriate due to a large number of steps. On such architectures, the number of steps corresponds to the number of points that represent the exchange of data. Consequently, these points often form a performance bottleneck. Our approach appropriately rearranges calculations inside the transform, and thereby reduces the number of steps. In other words, we propose a new scheme that is friendly to parallel environments. When evaluating on multi-core CPUs, we consistently overcome the original lifting scheme. The evaluation was performed on 61-core Intel Xeon Phi and 8-core Intel Xeon processors.

10.1 Introduction

The two-dimensional discrete wavelet transform (DWT) is a very versatile image processing instrument. It is employed in several image-compression standards (e.g., JPEG 2000). As a consequence, many works deal with its fast implementation on all sorts of computer systems, including parallel architectures. As it might be expected, many developers have adapted this transform on massively-parallel architectures, especially on GPUs. However, all of these adaptations are based on the most popular separable schemes – the convolution and lifting schemes. The separable convolution scheme can be computed in just two calculation steps, however, using a large number of arithmetic operations. Whereas, the separable lifting scheme exhibits the smallest number of operations, and, on the contrary, the largest number of steps. It is natural to expect that the number of operations should be proportional to

a transform performance. This is especially true on single-core CPUs. However, it is essential that also the number of steps forms a bottleneck. This is mainly meant in relation to multi-core processors.

In this paper, we show that the optimal scheme for multi-core CPUs lies aside the separable convolution and lifting schemes. To the best of our knowledge, this problem has not been addressed in the literature yet. The newly introduced scheme does not retain the separable property, as its operations cannot be associated with a horizontal or vertical direction. In order to evaluate the proposed scheme, we performed several experiments on high-end server CPUs. The evaluation is performed with CDF 5/3 wavelet, employed, e.g., in JPEG 2000 standard. However, the presented schemes are general and they are not limited to any particular wavelet.

The rest of this paper is organized as follows. Section 2 discusses a related work and introduces a mathematical notation used in the rest of the paper. Section 3 presents the proposed non-separable scheme and its adaptation to a particular platform. Section 4 evaluates the discussed schemes on multi-core CPUs. Eventually, Section 5 summarizes and closes the paper.

10.2 Background and Related Work

This section introduces some notations and definitions to be used in the paper, and then it reviews conventional methods for computation of the 2-D transform.

The widely-used z-transform is used for the description of wavelet filters. Such filters are represented by polynomials in z like $G(z)$. Since this paper is focused on 2-D transform, it is necessary to extend this notation into two dimensions. So, two-dimensional filters look like $G(z_m, z_n)$, where the subscript m refers to the horizontal axis and n to the vertical axis. The G^* indicates a polynomial transposed to the original G.

The DWT splits the input signal into two components, according to a parity of its samples. The components are often referred to as L and H. The transform can be computed by a pair of quadrature mirror filters, referred to as G, followed by subsampling by a factor of 2. Formally, this can be represented by the polyphase matrix

$$\begin{bmatrix} G_1^{(o)} & G_1^{(e)} \\ G_0^{(o)} & G_0^{(e)} \end{bmatrix}, \tag{10.1}$$

where operators (e) and (o) denote the even and odd terms of G. This equation defines one-dimensional convolution scheme. Further, Sweldens showed [31] how the convolution scheme can be decomposed into a sequence of simple steps. These filters are referred to as the lifting steps and the scheme as the lifting scheme. The following paragraph discusses the lifting scheme in detail.

The initial polyphase matrix (10.1) is factored into several pairs of lifting steps. In each pair, the first step is called the predict step and the second one as the update step. Formally, this can be represented by the product of polyphase matrices

$$\prod_k \begin{bmatrix} 1 & U^{(k)} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ P^{(k)} & 1 \end{bmatrix}, \tag{10.2}$$

where $2K$ is the number of the lifting steps, and $P^{(k)}$ and $U^{(k)}$ represent the kth predict and update filter. For simplicity, the superscript (k) is omitted in the following text.

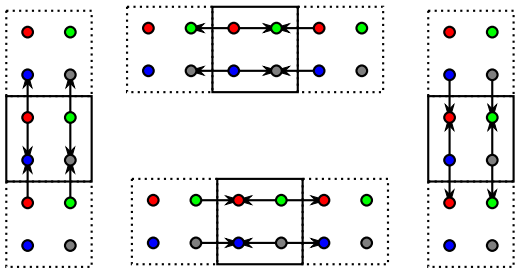


Figure 10.1: Shapes of steps in separable lifting scheme for the CDF 5/3 wavelet. The colored boxes correspond to LL, HL, LH, and HH quadruple. The arrows denote a multiply–accumulate operation.

On multi-core CPUs, the processing of single or several adjacent signal samples is mapped to independent cores. Due to the data exchange, the cores must use some synchronization method to avoid race conditions. In the lifting scheme, these synchronizations can be required before the lifting steps. In this paper, the synchronizations are indicated by the | symbol placed before a polyphase matrix. For example, $M_2|M_1$ refers to a sequence of two lifting steps separated by some synchronization method.

Usually, the 2-D transform [55] is defined as the tensor product of 1-D transforms. Unlike the 1-D case, the 2-D transform splits the input signal into a quadruple of wavelet coefficients (LL, HL, LH, and HH). To describe 2-D matrices, the predict and update operators must be extended into two dimensions. Considering the separable lifting scheme, the predict and update lifting steps can be applied in both directions sequentially. It should be noted that the horizontal and vertical steps can be arbitrarily interleaved. The 2-D lifting then follow from a sequence

$$\begin{bmatrix} 1 & 0 & U^* & 0 \\ 0 & 1 & 0 & U^* \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \Bigg| \begin{bmatrix} 1 & U & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & U \\ 0 & 0 & 0 & 1 \end{bmatrix} \Bigg| \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ P^* & 0 & 1 & 0 \\ 0 & P^* & 0 & 1 \end{bmatrix} \Bigg| \begin{bmatrix} 1 & 0 & 0 & 0 \\ P & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & P & 1 \end{bmatrix} \Bigg| . \tag{10.3}$$

Note the synchronization | before the matrices. As the sequence can be hard to imagine, the individual matrices are illustrated in Figure 10.1 for the CDF 5/3 wavelet [30]. For multiple lifting pairs, the scheme is separately applied to each such pair. Recall that the separable lifting scheme has the smallest possible number of arithmetic operations and the highest number of steps.

Another scheme used for 2-D transform is the separable convolution. In this case, all calculations in a single direction are performed in a single step. The drawback of this is the highest number of operations. The scheme can formally be described as

$$\mathbf{N}^V \big| \, \mathbf{N}^H \big| , \tag{10.4}$$

where $\mathbf{N}^H$ is a product of all steps in the horizontal direction and $\mathbf{N}^V$ is in the vertical one. The convolution is followed by the subsampling.

So far, several studies have compared the performance of the separable lifting and convolution schemes on parallel architectures. In an exemplary manner, the authors of [75] compared these schemes on GPUs. Although the results of their comparison are ambiguous,

they concluded that the separable convolution is more efficient than the separable lifting counterpart in most cases. They also claimed that fusing several consecutive steps might significantly speed up the execution, even if the complexity of the resulting fused step is higher. In this regard, the authors failed to consider the possibility of a partial fusion, where the number of steps is reduced but it remains greater than a single step. Other notable works can be found in [53, 40, 69].

This work is based on our previous work in [22, 17]. In these papers, we introduced several non-separable schemes for calculation of 2-D DWT suitable for graphics cards (GPUs). We also presented a trick leading to a reduction of arithmetic operations. The trick is also exploited in this paper. In this paper, we extend previously presented schemes to multi-core CPU platform. This is the point investigated in the following section.

10.3 Proposed Scheme

This section presents non-separable schemes suitable for multi-core CPUs. A contribution of the paper starts with this section.

The above-described approaches did not exploit the possibility of a fusion of polyphase matrices. Having this in mind, all horizontal and vertical calculations of the corresponding pair of matrices can be performed in a single step. The drawback of this approach is a higher number of operations and memory accesses. Since CPUs are very sensitive to the total number of arithmetics operations, the fusion will be appropriate to apply to the lifting scheme. In this way, non-separable lifting scheme is formed. The scheme has the same number of steps as its separable counterpart. On the other hand, the number of operations has been increased. The scheme consists of a spatial predict and spatial update steps. For curiosity, The predict step is completely responsible for the HH coefficient, whereas the update step for the LL one. Formally, the scheme is defined by

$$\begin{bmatrix} 1 & U & U^* & UU^* \\ 0 & 1 & 0 & U^* \\ 0 & 0 & 1 & U \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ P & 1 & 0 & 0 \\ P^* & 0 & 1 & 0 \\ PP^* & P^* & P & 1 \end{bmatrix}. \tag{10.5}$$

The PP^* and UU^* are the spatial filters (tensor products of 1-D filters). For CDF 5/3 wavelet, it is illustrated in Figure 10.2.

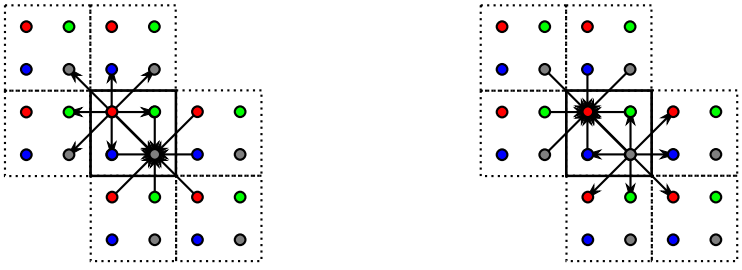


Figure 10.2: Non-separable lifting scheme for the CDF 5/3 wavelet. The predict step on the left, update on the right.

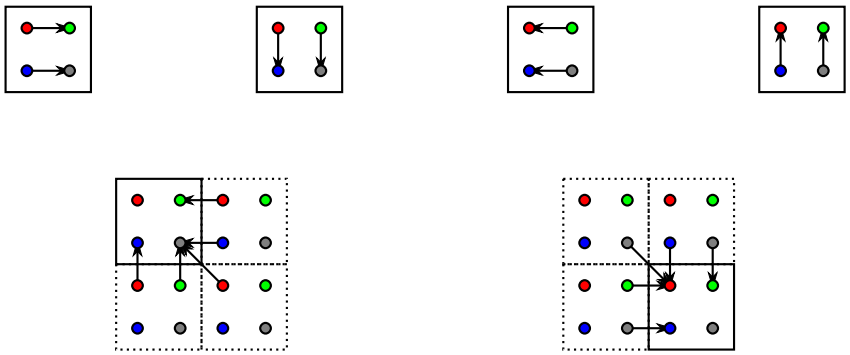


Figure 10.3: The platform-adapted non-separable scheme for the CDF 5/3 wavelet. The predict on the left, the update on the right.

As mentioned above, an optimization approach can adapt the schemes to a particular platform. The number of operations or memory accesses can be reduced, while the number of computational steps remains unaffected. Regardless the underlying platform, an important observation can be made. A special form of the operations guarantees that the CPU cores never access the results belonging to extraneous cores. These operations comprise constants (monomials with the zero exponents). As the convolution is the linear operation, these polynomials can be detached from the original operations, and calculated using the separable scheme (due to the lowest number of operations). Such schemes are referred to as adapted schemes. Formally, the original polynomials are split as $P = P_0 + P_1$ and $U = U_0 + U_1$, where P_0 and U_0 are the desired constants. The P_1 and U_1 are kept in the original non-separable scheme. For a better understanding, the adapted non-separable scheme is illustrated in Figure 10.3.

10.4 Evaluation

Since the above-listed properties do not provide sufficient information on performance in real environments, the performance on real multi-core CPUs is compared in this section.

In order to evaluate the considered schemes, high-performance server CPUs were used, along with the code written using the C language and OpenMP interface. The evaluation was performed primarily on Intel Xeon and Intel Xeon Phi server processors. Their technical parameters are summarized in Table 10.1. In the following paragraphs, several experiments on these CPUs are presented.

In the first experiment shown in Figure 10.4, the optimal number of threads was examined. The measurements were conducted with separable and non-separable schemes and CDF 5/3 wavelet. The transform performance was measured with tiles of 1024×1024 size, comprising single-precision floating-point values. The presented results are a median of 100 measurements. The time is given in nanoseconds per pixel (ns/pel). It is clear from the figure that the curves roughly approximate the $1/x$ function, where x is the number of threads. Therefore, the measurements made show that the optimal number of threads roughly corresponds to the maximum number of threads available. Note the phenomenon

Table 10.1: Description of the CPUs used for the evaluation.

	Intel Xeon	Intel Xeon Phi (MIC)
model	Xeon E5-2620 v4	Intel Xeon Phi 7120P
cores	8	61
concurrent threads	16	244
clock (turbo)	2.1 GHz (3.0 GHz)	1.238 GHz (1.333 GHz)
on-chip memory	20 MB (L3 cache)	30.5 MB (L2 cache)
off-chip memory	DDR4 (2.133 GHz)	GDDR5 (2.75 GHz)

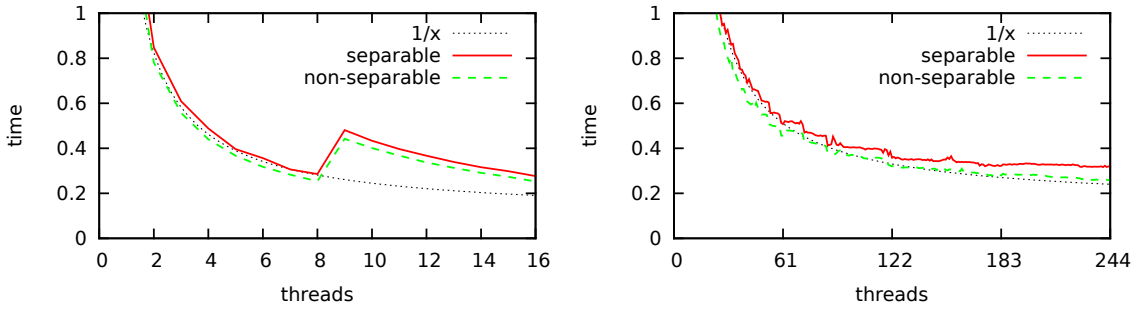


Figure 10.4: Examination of the optimal number of threads. The Xeon on the left, Xeon Phi on the right. The performance scales almost linearly.

that occurs when the number of threads exceeds the number of CPU cores (i.e. 8 for the Xeon, 61 for the Xeon Phi).

In the second experiment in Figure 10.5, the optimal transform tile size was examined. The number of threads that was found optimal in the previous experiment was used. For the Xeon CPU, the optimal power-of-two tile size was chosen as 1024×1024. For the Xeon Phi, the size 2048×2048 was chosen. Note that the tile size does not necessarily have to be a power of two, but this is a suitable choice, for example, due to JPEG 2000.

In the last experiment in Figure 10.6, we were interested in a real performance. The x-axis shows the size of the image edge. The input and output images were supplied by the main memory. Note that the image sizes exceed CPU cache size after a while. The experiment confirms that the non-separable scheme consistently overcomes the original

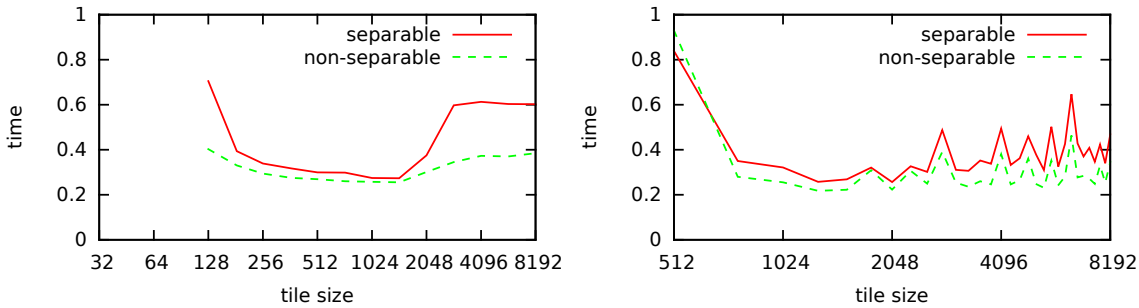


Figure 10.5: Examination of the optimal transform tile size. The Xeon on the left, Xeon Phi on the right.

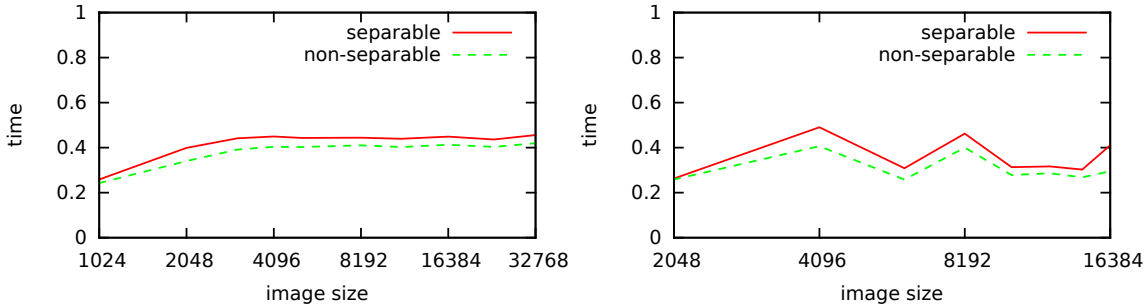


Figure 10.6: Performance on large images. The Xeon on the left, Xeon Phi on the right.

separable lifting scheme. For example, for 8192×8192 image, the speedup factor is about 10% on the Xeon and 25% on the Xeon Phi processor.

In summary, we can conclude that the reduction in transform steps can improve performance, at least on some platforms. All the source codes used in this article together with all the results are available in a repository [59] on the website of the authors' affiliation.

10.5 Summary

This paper introduces and discusses the non-separable lifting scheme for computation of the two-dimensional discrete wavelet transform on multi-core CPUs. We found that the non-separable scheme outperforms its separable counterpart in most cases. We can confirm that fusing consecutive steps of the original lifting scheme might speed up the execution, irrespective of its higher complexity in terms of arithmetics operations. The presented scheme is general and it can be used in conjunction with any wavelet transform.

For future work, we plan to extend our approach to other wavelets and possibly other non-separable schemes. The implementation can also further be improved using appropriate SIMD extensions. Finally, we look for other multi-core platforms such as multi-core ARM processors.

Acknowledgements This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science (LQ1602), and the Technology Agency of the Czech Republic (TA CR) Competence Centres project V3C – Visual Computing Competence Center (no. TE01020415).

Chapter 11

Summary, Applications and Future Work

The proof of the hypothesis, introduced in the chapter 4, is summarized in this chapter in the section 11.1. Additionally, the list of novel schemes, used for proving the hypothesis, and comparisons of these schemes against the existing ones are described in the section 11.2. Possible applications of novel approaches are presented in the section 11.3. And finally, the potential possible future work is summarized in the section 11.4.

11.1 Summary

The papers presented in the chapters 6, 7, 8, 9 and 10 contain experimental proof of the hyphotesis formed in the chapter 4 or, more preciselly said, its individual parts.

Summary of these experimental proof parts is as follows:

1. The paper presented in the chapter 6 proposed a novel approach for calculating the vertical pass of the pipeline method. It examined the impact of balance between barriers, arithmetic operations and local memory usage for such pass. The evaluation of the approach, presented in the section 6.3.2, proves the hypothesis for vertical pass of the pipeline method by overcoming throughput of state-of-the-art method one by 30%.

2. The paper presented in the chapter 7 introduced seamless block-based 2D DWT approach. The evaluation of the approach, presented in the section 7.3, proves the hypothesis for seamless 2D DWT by outperforming the best performing separable approach by 60-100% on the tested platforms.

3. The paper presented in the chapter 7 also proposed the novel non-separable scheme denoted as Kula2016. The evaluation of the scheme, presented in the section 7.3, proves the hypothesis for block-based strategy by outperforming throughput of separable lifting scheme on reasonable resolutions by 5% on the tested platforms.

4. The optimization approach, introduced in the paper within the chapter 8, is focused on reducing the number of arithmetics operations of various schemes by separating intra-thread calculations at the end of calculation steps and before implicit synchronization only when local memory is used. The evaluation of the approach, presented in the section 8.5, proves the hypothesis on the tested platforms by increasing the 2D DWT

throughput by 80% on non-separable polyconvolution and 5-20% on the rest of the schemes except for separable lifting$^+$ one where its usage is not beneficial.

5. The paper presented in the chapter 7 also introduces the novel non-separable lifting scheme, denoted in the section 8.3.2 as Monolithic, focused on lowering memory barriers and arithmetics operations. The evaluation of the scheme, presented in the section 8.5, proves the hypothesis by outperforming the state-of-the-art schemes used by other authors (implosion described in the section 8.2.6 as Iwahashi and separable lifting$^+$ described in the section 8.2.5 as Sweldens) by 30% on the tested platforms.

6. The paper presented in the chapter 9 examines the fusion of 1D schemes in 2D DWT. The evaluation of the schemes created by such fusion, presented in the section 9.6, proves the hypothesis for 1-degree schemes (CDF 5/3 and CDF 9/7) on the selected platforms by showing that partially fused non-separable polyconvolution scheme combined with optimization approach outperforms optimized non-separable lifting scheme by more than 25% using GPGPU paradigm and by 50% using graphics pipeline paradigm.

7. The paper presented in the chapter 10 adapts non-separable lifting scheme, presented in the section 8.3.2 as Monolithic, for CPU/Xeon-Phi platforms. The evaluation of the scheme, presented in the section 10.4, proves the hypothesis by improving the 2D DWT throughput by 10% compared to the separable lifting$^+$ scheme on the tested CPU and Xeon-Phi platforms.

The presented experimental results of the introduced approaches prove the hypothesis for selected platforms, wavelets, and paradigms. It can also be pressumed that the hyphothesis, formed in the chapter 4, is more most probably valid in much wider circumstances.

However, other techniques leading to improving of the throughput of the algorithms exist (e.g. use of SSE/AVX/AVX512 SIMD extensions on CPU platforms, use of warp-shuffle instructions on supported GPU platforms, etc.); nevertheless, these techniques can be used parallelly with the proposed ones and they should be orthogonal to them.

Moreover, evaluation on the last generations of Nvidia and AMD GPU architectures shows a memory bandwidth as the main bottleneck of 2D DWT calculation when using GPGPU paradigm. In those architectures, the performance is the same regardless of the underlying scheme (except for the slow ones like non-separable convolution without optimization approach). The possible beneficial usage of the approaches on such platforms lies in the fusion of 2D DWT with its application on denoising or additional step of JPEG 2000 encoding/decoding.

11.2 Schemes List

This section contains comprehensive summary descriptions and visualizations of the existing schemes, described in the sections 3.1.1, 3.1.2 and 3.1.3, and the novel ones introduced in the sections 8.3, 8.4, 9.4 and 11.4.

Dataflow diagrams of Non-separable schemes and separable lifting$^+$ scheme without the optimization approach, presented in the section 8.4, are visualized in Figure 11.1. The diagrams of Non-separable schemes with the optimization technique are visualized on the right side of Figure 11.2 along with optimization steps on the left side. Finally, the separable and non-separable variants of schemes formed by fusion of lifting steps, with and without optimization techniques, are presented in Figure 11.3.

The computation parameters of the schemes suitable for paradigms and implementations that use implicit synchronization before the first step of calculation are shown in Table 11.1 (e.g. synchronization after copying the inputs to the local memory on GPGPU paradigm, memory re-organization before the first step of scheme on CPU platforms, etc.). In contrast to Table 11.1, Table 11.2 contains parameters of schemes suitable for paradigms and implementations that do not use such synchronization (e.g. graphics pipeline paradigm, direct calculation of 2D DWT without re-organizing memory before the first step of the scheme on CPU platforms, etc.).

Comparison of computation parameters from the tables clearly shows that reducing one computation parameter results in increasing of another one. Generally, 3 types of schemes exist:

- Schemes focused on lowering the number of barriers are suitable for platforms and paradigms with costly barriers (like graphics pipeline paradigm that forces to write data to global memory after each step). In such parameter, non-separable convolution and polyconvolution schemes are dominating followed by separable convolution, polyconvolution and non-separable lifting schemes.
- Schemes focused on lowering the number of operations are preferred especially on platforms with low cost barriers. Schemes oriented to lowering the number of operations are separable lifting scheme variants followed by non-separable lifting, explosion, and implosion schemes. Note that non-separable lifting scheme seems to have well balanced barriers and number of operations, however, it is not suitable for platforms that suffer from high memory usage.
- Schemes focused on lowering the memory usage are suitable for platforms with high latency or low throughput of local memory. The schemes with relativelly low number of operations and memory usage are non-separable explosion scheme and separable interleave lifting$^+$ scheme.

11.3 Possible Applications

The evaluation on the last generations of Nvidia and AMD platform GPUs shows that global memory bandwidth is the main bottleneck of calculation, especially in combination with simple wavelets like CDF 5/7. In that case, the performance is the same regardless of the underlying scheme (except for the slow ones like non-separable convolution/polyconvolution without optimization approach). The possible beneficial usage of schemes lies in the fusion of 2D DWT with applications such as denoising or additional step of JPEG 2000 encoding. Another beneficial usage can be in transforming wavelets with more predict-updated steps or more degreed wavelets.

One of the ways how to continue the work can be the use of 2D DWT implementation as a part of the JPEG 2000 compression/decompression algorithm. Our previous research showed that well-implemented 2D DWT, which uses the best performing scheme for undergoing platform, often has its bottleneck in global memory throughput. Test of the presented algorithms with global memory surpassed by reading of constant memory and writing output based on unfulfillable condition shows that modern platforms can process wavelets up to 10 times faster[1] compared to tests that used global memory for storing and loading data.

[1] speedup is dependent on used wavelet

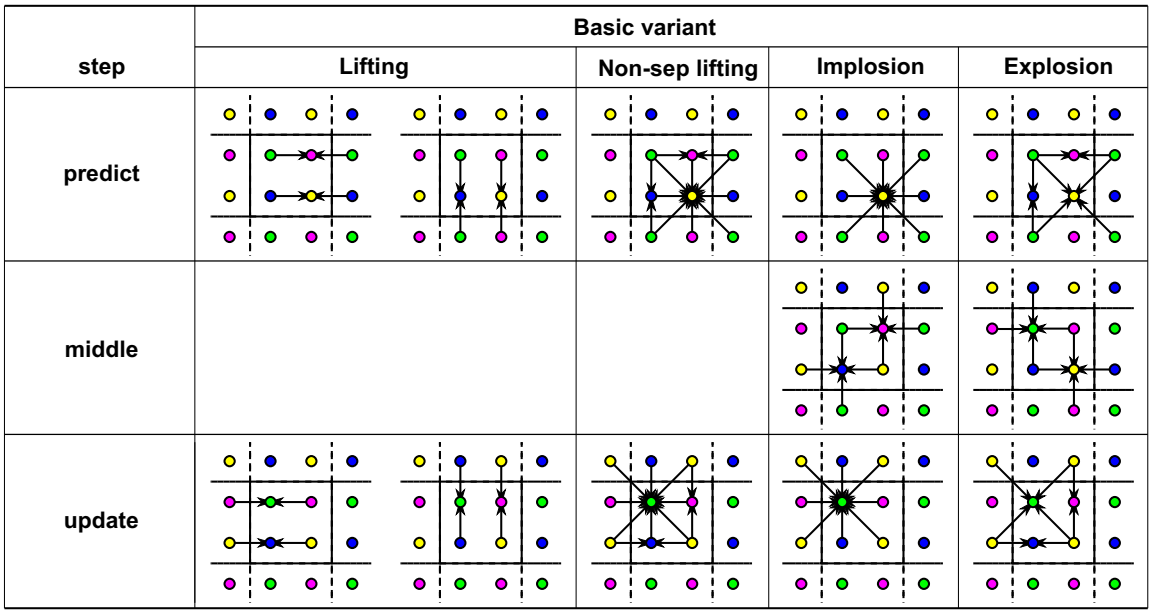


Figure 11.1: 2D dataflow diagram representing calculation of 2D DWT schemes without optimization approach from view of single thread. The thread is mapped onto quadruple of pixel elements (dots in solid box). The arrows indicate fused multiply-add operations issued by the thread. Each vertical block represents calculation of single scheme.

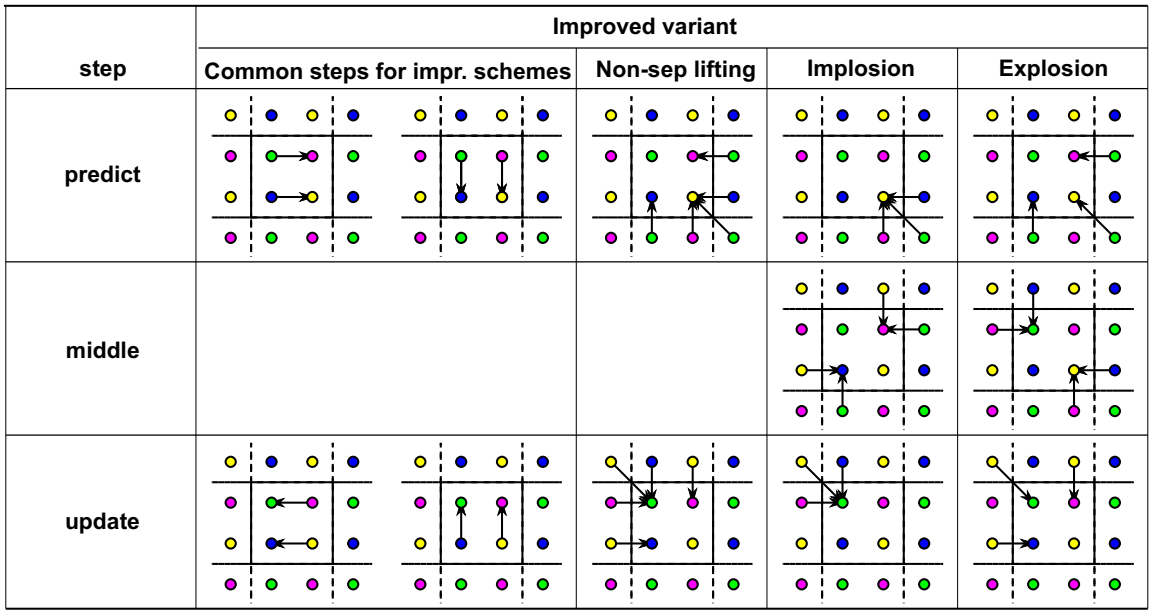


Figure 11.2: 2D dataflow diagram representing calculation of 2D DWT schemes with optimization approach from view of single thread. The thread is mapped onto quadruple of pixel elements (dots in solid box). The arrows indicate fused multiply-add operations issued by the thread. Each vertical block represents calculation of single scheme. The calculation of the schemes is preceded by the predict common steps and followed by the update common steps.

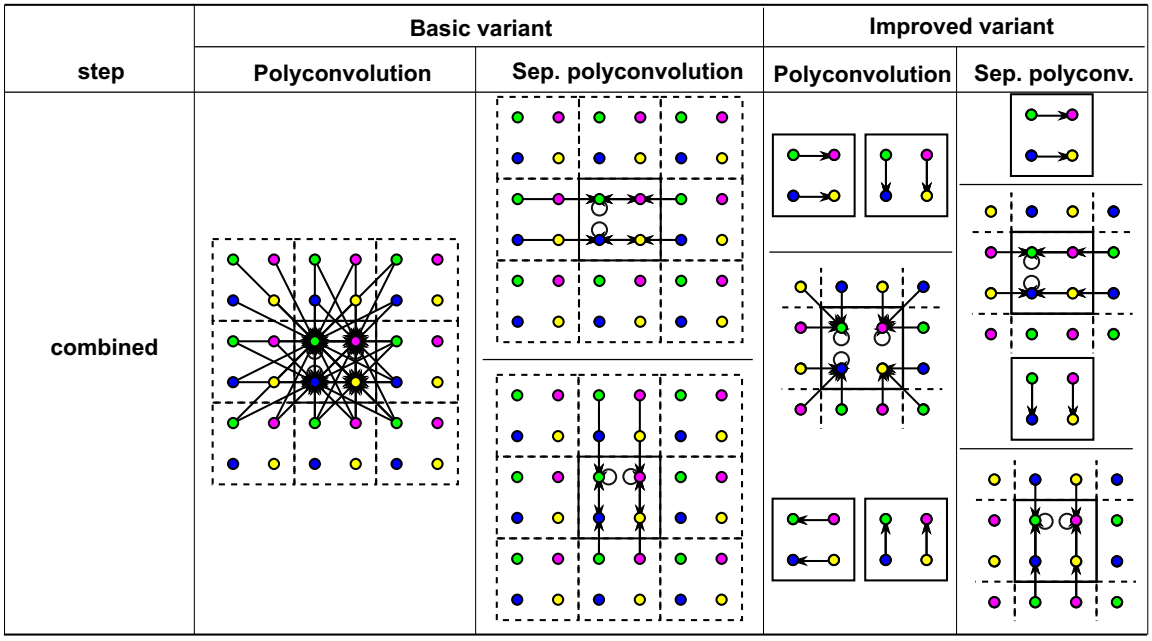


Figure 11.3: 2D dataflow diagram representing calculation of 2D DWT polyconvolution scheme variants from view of single thread. The thread is mapped onto quadruple of pixel elements (dots in solid box). The arrows indicate operations issued by the thread. The straight arrows indicate fused multiply-add operations, and the self-directed circle arrows indicate multiplication operations. Each vertical block represents calculation of single scheme. Vertical lines represent synchronizations. Note that non-improved variants precede implicit barrier as well.

scheme	read write	read degree-1	read degree-3	barriers	memory single	memory double	operations degree-1	operations degree-3
convolution$^{K=2}$	8	28		2	4	8	64	
convolution$^{*K=2}$	8	24		2	4	8	56	
polyconvolution	8K	12K	36K	2K	4	8	28K	60K
polyconvolution*	8K	**8K**	36K	2K	4	8	20K	60K
polyconvolutionX	8K	10K	30K	3K	4	8	22K	46K
polyconvolutionX*	8K	**8K**	30K	3K	4	8	18K	46K
polyconvolutionY	8K	10K	30K	3K	4	8	22K	46K
polyconvolutionY*	8K	**8K**	30K	3K	4	8	18K	46K
lifting	1+6K	**8K**	**24K**	4K	**2**	4	**16K**	**32K**
lifting^{+}	1+4K	**8K**	**24K**	4K	**2**	**3**	**16K**	**32K**
implosion	2+4K	10K	42K	3K	3	4	24K	64K
implosion*	6K	10K	42K	3K	3	(6) 4	18K	50K
explosion	**4K**	10K	42K	3K	**2**	**3**	24K	64K
explosion*	**4K**	10K	42K	3K	**2**	**3**	18K	50K
lifting	6K	10K	42K	2K	3	6	24K	64K
lifting*	6K	10K	42K	2K	3	6	18K	50K
polyconvolution	**4K**	21K	117K	**1K**	4	(8) 4	63K	255K
polyconvolution*	**4K**	12K	117K	**1K**	4	(8) 4	23K	203K
convolution$^{K=2}$	**4**	77		**1**	4	4	256	
convolution$^{*K=2}$	**4**	60		**1**	4	4	152	

Table 11.1: Comparison of schemes properties for graphics pipeline processing. Table consists of local memory reads/writes, barriers, local memory elements per quadruple in single and double buffering and operations for non-separable schemes. The K denotes the number of predict/update pairs. Schemes marked by symbol * use the optimization approach. The degree-1 wavelets correspond to CDF wavelets, whereas degree-3 to DD 13/7. Schemes above horizontal line are separable and contain only separable steps, whereas schemes below the line are non-separable and contain spatial steps as well. Separable and non-separable convolution schemes with 1 predict-update pair ($K = 1$) and their improved counterparts are in fact polyconvolution with $K = 1$.

scheme	read			steps	memory	operations	
	write	degree-1	degree-3			degree-1	degree-3
convolution$^{K=2}$	8	36		2	8	64	
convolution$^{\#K=2}$	8	34		2	8	58	
polyconvolution	$8K$	$20K$	$44K$	$2K$	8	$28K$	$60K$
polyconvolution$^{\#}$	$8K$	**2+16K**	$44K$	$2K$	8	2+$20K$	$60K$
polyconvolutionX	$8K$	$22K$	$42K$	$3K$	8	$22K$	$46K$
polyconvolutionX$^{\#}$	$8K$	2+$20K$	$42K$	$3K$	8	2+$18K$	$46K$
polyconvolutionY	$8K$	$22K$	$42K$	$3K$	8	$22K$	$46K$
polyconvolutionY$^{\#}$	$8K$	$20K$	$42K$	$3K$	8	$18K$	$46K$
lifting	$8K$	$24K$	**40K**	$4K$	**4**	**16K**	**32K**
lifting^{+}	$8K$	$24K$	**40K**	$4K$	**4**	**16K**	**32K**
implosion	**4K**	$22K$	$54K$	$3K$	**4**	$24K$	$64K$
implosion$^{\#}$	-1+$7K$	4+$17K$	1+$53K$	$3K$	(7)6	3+$18K$	7+$50K$
explosion	$8K$	$22K$	$54K$	$3K$	**4**	$24K$	$64K$
explosion$^{\#}$	-1+$9K$	3+$19K$	1+$53K$	$3K$	(5)4	1+$18K$	1+$50K$
lifting	$6K$	**18K**	$50K$	$2K$	6	$24K$	$64K$
lifting$^{\#}$	$6K$	**1+17K**	1+$49K$	$2K$	(7)6	$18K$	$50K$
polyconvolution	**4K**	$25K$	$121K$	**1K**	8	$63K$	$255K$
polyconvolution$^{\#}$	**4K**	$25K$	$121K$	**1K**	8	16+$23K$	25+$203K$
convolution$^{K=2}$	**4**	81		**1**	**4**	256	
convolution$^{\#K=2}$	**4**	81		**1**	**4**	200	

Table 11.2: Comparison of schemes properties for graphics pipeline processing. Table consists of global memory reads/writes, number of scheme steps and operations for non-separable schemes. The K denotes the number of predict/update pairs. Schemes marked by $^{\#}$ symbol use graphics pipeline version of the optimization approach. The degree-1 wavelets correspond to CDF wavelets, whereas degree-3 to DD 13/7. Schemes above horizontal line are separable and contain only separable steps, whereas schemes below the line are non-separable and contain spatial steps as well. Separable and non-separable convolution schemes with 1 predict-update pair ($K = 1$) and their improved counterparts are in fact polyconvolution with $K = 1$.

When merging the 2D DWT with applications like denoising or subsequent parts of JPEGs 2000 coding (quantization and code block arithmetic coding), it would reverse the ratio between global memory reads and the calculation itself.

For JPEG 2000, the limiting factors for these merged kernels are the amount of registers and local memory that can be allocated to the work-group. Due to limitations based on resources rather than thread counts, the higher number of threads per work-group would result in a better occupancy. Modern AMD and Nvidia GPUs allow creating the work-groups with up to 1024 threads in OpenCL/Cuda frameworks. Modern GPU architectures seem to have their limitations in 128x128 pixel elements block size. If program state, intermediate and index registers are not considered, storing pixel elements uses 16k registers from 64k currently provided by modern platforms (Nvidia). Another issue can arise with the amount of local memory mappable to work-group where a platform is: capable of using only half of the available local memory per work-group[2], configurable only by platform-specific framework[3]. Thus, modern GPU architectures can use only 32kB - 64kB of local memory, allowing storing only 2 - 4 shared elements per quadruple. That greatly reduces the number of schemes usable for this application (memory requirements are presented in Table 11.1 and Table tbl:schemes-pipeline). Additionally, the exact sizing[4] is required (in Wavelet transform framework presented in the section 5.5 is the exact sizing available only on the schemes where it is possible) for storing number of the shared elements per quadruple corresponding to maximum amount of local memory.

11.4 Future Work

Although many schemes, wavelets, and optimizations are proposed and evaluated in the previous chapters, extensions that are not explored yet still exist. Such extensions can be integrated in Wavelet transform framework presented in the chapter 5. The extensions are presented in this chapter.

Additional Schemes

As described in the chapter 9, the fusion of lifting steps combined with the optimization approach (introduced in the section 8.4) leads to several new schemes. In some cases, it should be beneficial to fuse steps only in one axis, which leads to forming of new schemes denoted as polyconvolutionX for horizontal polyconvolution and vertical lifting steps, polyconvolutionY for horizontal lifting and vertical polyconvolution steps and polyconvolutionX* and polyconvolutionY* as their improved variants in Table 11.1.

N-Dimensional Wavelet Transform

Novel schemes and optimization approach introduced in the sections 6.3.2, 8.3, 8.4 and 9.4 can be further extended to N-Dimensional form. Every thread can be mapped onto an N-Dimensional cube with 2^N elements in a single N-Dimensional DWT kernel solution. Separable schemes, including those proposed in the chapter 9, can be used as-is with the same steps rotated to every dimension. Additional schemes, presented in the previous section, can be extended for various combinations of lifting, polyconvolution and convolution

[2] 32kB/64kB on AMD GCN 64lk/128kB on ADM RDNA

[3] configuration of ratio between local memory size and L1 cache can be set by Nvidia Cuda Runtime API

[4] Exact sizing requires zero sized preface and postface local memory parts.

separable schemes form for each axis. N-Dimensional non-separable lifting scheme can be formed as a 2-step scheme that consists of 1 predict step formed by fusion of N predict lifting steps each rotated to N axis and 1 update step formed similarly using N update lifting steps. The optimization approach, introduced in the section 8.4, contains separable operators in the x following y-axis; thus, it can be extended to N-Dimension in the same matter as well.

Shuffles, Multiple Quadruples per Thread

Many schemes and improvements are proposed in the sections 8.3, 8.4 and 9.4 and evaluated in the sections 8.5 and 9.6. Still, none of them is transformed using warp-optimization techniques (described in the section 5.6) and tested on work-group sizes and shapes combined with various sizes and shapes of quadruples per thread. For such purpose, the Wavelet transform framework described in the chapter 5 is already capable of evaluation of such extensions. The evaluation is currently work-in-progress.

CPU Acceleration

The evaluation, presented in the section 10.4, shows that the non-separable lifting scheme (introduced in the section 8.3.2) is usable for multicore CPUs and GPUs. But not all schemes nor SIMD instructions sets like AVX512/AVX2/SSE are examined. For such purpose the CPU SIMD extension for Wavelet generator used in Wavelet transform framework is currently in implementation phase. After finalization, the framework will be capable of evaluation of any scheme presented in the section 11.2 combined with mentioned SIMD extensions. Additionally, the better memory layout is part of the new approach.

Chapter 12

Conclusion

Despite a 2D DWT being an object of many studies during the last two decades, some aspects have not been studied yet. One of these aspects is the calculation of 2D DWT with variously balanced barriers, arithmetic operations and memory usage focused on various architectures. This thesis shows several new methods of computation of such transform with variously balanced operations. The hypothesis of the thesis, which is focused on acceleration of 2D DWT calculation, is formed in the chapter 4, experimentally proved in the chapters 6, 7, 8, 9 and 10, and these proofs are summarized in the beginning of the chapter 11.

The Wavelet transform framework was created for evaluation purposes. The framework is responsible for generating and evaluating wavelet implementation for the combination of desired scheme, destination platform and wavelet type. Generation itself is not limited to a list of supported wavelet types; nevertheless, it is useful for generating an implementation of wavelets of any number of predicted-update steps and any degree. The recent version is also capable of generating CPU implementations of schemes with various SIMD extensions.

During the 2D DWT researches, several novel methods are proposed: new method for vertical pass of pipelined processing; the optimization approach that is capable of separation of some operations outside the calculations that leads to lowering the number of arithmetic instructions on almost all available schemes; non-separable lifting scheme focused on lowering the number of barriers with reasonable amount of operations in various variants including those preserving the number of operations even when the implicit barrier is not used (graphics pipeline processing, CPU processing); non-separable explosion scheme focused on lowering the local memory usage; non-separable polyconvolution scheme focused on reduction of barriers while preserving reasonable amount of operations. The methods were evaluated on various GPU and CPU platforms and paradigms. The details regarding the schemes and possible applications is presented in the chapter 11.

Although many novel methods for 2D DWT calculation are proposed and evaluated in the thesis, some aspects remain unexplored and can be a subject of the future work. Examples of such aspects include: extension of proposed schemes to N-Dimension form for N-Dimensional DWT calculation; a combination of the proposed methods with other optimization techniques like usage of GPU shuffles or SSE/AVX2/AVX512 extensions on CPUs; formation and evaluation schemes formed by fusion of separable lifting scheme steps unequally on a various axis; fuse the 2D DWT with its application like JPEG 2000 or denoising algorithm. More details regarding the future work are described in the chapter 11.

Bibliography

[1] *Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. Santa Clara, California, USA: NVIDIA Corporation, 2012. Available at: http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf.

[2] *AMD Accelerated Parallel Processing OpenCL User Guide*. Sunnyvale, California, USA: Advanced Micro Devices, December 2014. Available at: http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide.pdf.

[3] *Whitepaper: NVIDIA GeForce GTX 980 - Featuring Maxwell The Most Advanced GPU Ever Made*. Santa Clara, California, USA: NVIDIA Corporation, 2014.

[4] *The Compute Architecture of Intel Processor Graphics Gen9*. Santa Clara, California, USA: Intel Corporation, 2015. Available at: https://www.intel.com/content/dam/develop/external/us/en/documents/the-compute-architecture-of-intel-processor-graphics-gen9-v1d0.pdf.

[5] *Intel Processor Graphics Gen11 Architecture*. Santa Clara, California, USA: Intel Corporation, 2019. Available at: https://www.intel.com/content/dam/develop/external/us/en/documents/the-architecture-of-intel-processor-graphics-gen11-r1new.pdf.

[6] *RDNA Architecture*. Sunnyvale, California, USA: Advanced Micro Devices, 2019. Available at: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf.

[7] *Cuda C Programming Guide - Design Guide*. Santa Clara, California, USA: NVIDIA Corporation, September 2020. Available at: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf.

[8] *AMD INSTINCT^{TM} MI100 ACCELERATOR*. Sunnyvale, California, USA: Advanced Micro Devices, 2021. Available at: https://www.amd.com/system/files/documents/instinct-mi100-brochure.pdf.

[9] *Cuda Occupancy calculator*. Santa Clara, California, USA: NVIDIA Corporation, 2021. Available at: https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls.

[10] *NVIDIA A100 Tensor Core GPU - Unprecedented Acceleration at Every Scale*. Santa Clara, California, USA: NVIDIA Corporation, 2021. Available at: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf.

[11] *NVIDIA Ampere GA102 GPU architecture*. Santa Clara, California, USA: NVIDIA Corporation, 2021. Available at: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[12] *Parallel Thread Execution ISA*. Santa Clara, California, USA: NVIDIA Corporation, October 2021. Available at: https://docs.nvidia.com/cuda/pdf/ptx_isa_7.5.pdf.

[13] *The OpenCL Specification*. Beaverton, Oregon, USA: Khronos OpenCL Working Group, October 2021. Available at: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

[14] Arguello, F., Heras, D. B., Boo, M. and Lamas Rodriguez, J. The split-and-merge method in general purpose computation on GPUs. *Parallel Computing*. Elsevier. 2012, vol. 38, 6–7, p. 277–288. DOI: 10.1016/j.parco.2012.03.003. ISSN 0167-8191.

[15] Błażewicz, M., Ciżnicki, M., Kopta, P., Kurowski, K. and Lichocki, P. *Two-Dimensional Discrete Wavelet Transform on Large Images for Hybrid Computing Architectures: GPU and CELL*. Springer, 2012. 481–490 p. LNCS. ISBN 978-3-642-29736-6.

[16] Barina, D., Kula, M., Matysek, M. and Zemcik, P. Accelerating Discrete Wavelet Transforms on GPUs. In: *International Conference on Image Processing (ICIP)*. IEEE Signal Processing Society, 2017, p. 2707–2710. DOI: 10.1109/ICIP.2017.8296774. ISBN 978-1-5090-2175-8.

[17] Barina, D., Kula, M., Matysek, M. and Zemcik, P. Accelerating Discrete Wavelet Transforms on Parallel Architectures. *Journal of WSCG*. Union Agency. 2017, vol. 25, no. 2, p. 77–85. ISSN 1213-6972.

[18] Barina, D., Kula, M. and Zemcik, P. Simple Signal Extension Method for Discrete Wavelet Transform. In: *Proceedings of 2016 IEEE International Conference on Signal and Image Processing*. IEEE, 2016, p. 534–538. DOI: 10.1109/SIPROCESS.2016.7888319. ISBN 978-1-5090-2375-2.

[19] Barina, D., Najman, P., Kleparnik, P., Kula, M. and Zemcik, P. The Parallel Algorithm for the 2D Discrete Wavelet Transform. In: *Ninth International Conference on Graphic and Image Processing (ICGIP 2017)*. SPIE - the international society for optics and photonics, 2017, vol. 10615, no. 4, p. 1–6. DOI: 10.1117/12.2302881. ISBN 978-1-5106-1741-4.

[20] Barina, D. and Zemcik, P. Diagonal Vectorisation of 2-D Wavelet Lifting. In: *IEEE International Conference on Image Processing (ICIP)*. Paris, France: IEEE, October 2014, p. 2978–2982.

[21] Barina, D. and Zemcik, P. Vectorization and parallelization of 2-D wavelet lifting. *Journal of Real-Time Image Processing*. Springer. 2015. DOI: 10.1007/s11554-015-0486-6. ISSN 1861-8200.

[22] Bařina, D., Kula, M. and Zemčík, P. Parallel wavelet schemes for images: How to make the wavelet transform friendly to parallel architectures. *Journal of Real-Time Image Processing*. Springer Science and Business Media LLC. 2019, vol. 16, no. 5, p. 1365–1381. DOI: 10.1007/s11554-016-0646-3. ISSN 1861-8200.

[23] Bařina, D., Musil, M., Musil, P. and Zemčík, P. Single-Loop Approach to 2-D Wavelet Lifting with JPEG 2000 Compatibility. In: *IEEE 27th International Symposium on Computer Architecture and High Performance Computing Workshops*. IEEE Computer Society, 2015, p. 31–36. DOI: 10.1109/SBAC-PADW.2015.10. ISBN 978-1-4673-8621-0.

[24] Bařina, D. and Zemčík, P. Minimum Memory Vectorisation of Wavelet Lifting. In: *Advanced Concepts for Intelligent Vision Systems (ACIVS)*. Springer London, 2013, vol. 8192, p. 91–101. Lecture Notes in Computer Science (LNCS) 8192. DOI: 10.1007/978-3-319-02895-8_9. ISBN 978-3-319-02894-1.

[25] Bernabé, G., Guerrero, G. D. and Fernández, J. CUDA and OpenCL implementations of 3D Fast Wavelet Transform. In: *2012 IEEE 3rd Latin American Symposium on Circuits and Systems (LASCAS)*. IEEE, 2012, p. 1–4. DOI: 10.1109/LASCAS.2012.6180318.

[26] Chatterjee, S. and Brooks, C. D. Cache-efficient Wavelet Lifting in JPEG 2000. In: *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME)*. IEEE, 2002, vol. 1, p. 797–800. DOI: 10.1109/ICME.2002.1035902.

[27] Chaver, D., Tenllado, C., Pinuel, L., Prieto, M. and Tirado, F. Vectorization of the 2D wavelet lifting transform using SIMD extensions. In: *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, 2003, p. 8 pp.–. DOI: 10.1109/IPDPS.2003.1213416.

[28] Chaver, D., Tenllado, C., Pinuel, L., Prieto, M. and Tirado, F. Vectorization of the 2D Wavelet Lifting Transform using SIMD extensions. In: *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2003, p. 8. DOI: 10.1109/IPDPS.2003.1213416. ISSN 1530-2075.

[29] Chaver, D., Tenllado, C., Pinuel, L., Prieto, M. and Tirado, F. *Wavelet Transform for Large Scale Image Processing on Modern Microprocessors*. Springer, 2003. 549–562 p. LNCS. ISBN 978-3-540-00852-1.

[30] Cohen, A., Daubechies, I. and Feauveau, J.-C. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*. Wiley. 1992, vol. 45, no. 5, p. 485–560. DOI: 10.1002/cpa.3160450502. ISSN 1097-0312.

[31] Daubechies, I. and Sweldens, W. Factoring Wavelet Transforms into Lifting Steps. *Journal of Fourier Analysis and Applications*. 1998, vol. 4, no. 3, p. 247–269. DOI: 10.1007/BF02476026. ISSN 1069-5869.

[32] Descampe, A. and Devaux, F. A flexible, line-based JPEG 2000 decoder for digital cinema. In:. June 2004, p. 1165 – 1170 Vol.3. DOI: 10.1109/MELCON.2004.1348272. ISBN 0-7803-8271-4.

[33] Dillen, G., Georis, B., Legat, J. and Cantineau, O. Combined line-based architecture for the 5-3 and 9-7 wavelet transform of JPEG2000. *IEEE Transactions on Circuits and Systems for Video Technology*. IEEE. 2003, vol. 13, no. 9, p. 944–950. DOI: 10.1109/TCSVT.2003.816518.

[34] Enfedaque, P., Aulí Llinàs, F. and Moure, J. C. Implementation of the DWT in a GPU through a Register-based Strategy. *IEEE Transactions on Parallel and Distributed Systems*. IEEE. 2015, vol. 26, no. 12, p. 3394–3406. DOI: 10.1109/TPDS.2014.2384047.

[35] Fang, J., Varbanescu, A. L. and Sips, H. A Comprehensive Performance Comparison of CUDA and OpenCL. In: *Proceedings of the 2011 International Conference on Parallel Processing*. IEEE Computer Society, 2011, p. 216–225. ICPP. DOI: 10.1109/ICPP.2011.45. ISBN 978-0-7695-4510-3.

[36] Franco, J., Bernabe, G., Fernandez, J. and Acacio, M. A Parallel Implementation of the 2D Wavelet Transform Using CUDA. In: *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE Computer Society, February 2009, p. 111–118. DOI: 10.1109/PDP.2009.40. ISSN 1066-6192.

[37] Franco, J., Bernabe, G., Fernandez, J. and Ujaldon, M. The 2D wavelet transform on emerging architectures: GPUs and multicores. *Journal of Real-Time Image Processing*. Springer. 2011, vol. 7, no. 3, p. 145–152. DOI: 10.1007/s11554-011-0224-7. ISSN 1861-8219.

[38] Franco, J., Bernabé, G., Fernández, J. and Ujaldon, M. Parallel 3D fast wavelet transform on manycore GPUs and multicore CPUs. *Procedia CS*. may 2010, vol. 1, p. 1101–1110. DOI: 10.1016/j.procs.2010.04.122.

[39] Galiano, V., Lopez, O., Malumbres, M. P. and Migallon, H. Improving the discrete wavelet transform computation from multicore to GPU-based algorithms. In: *Int. Conf. on Computational and Mathematical Methods in Science and Engineering*. CMMSE, June 2011, p. 544–555. ISBN 978-84-614-6167-7.

[40] Galiano, V., Lopez, O., Malumbres, M. and Migallon, H. Parallel strategies for 2D Discrete Wavelet Transform in shared memory systems and GPUs. *The Journal of Supercomputing*. Springer. 2013, vol. 64, no. 1, p. 4–16. DOI: 10.1007/s11227-012-0750-5. ISSN 0920-8542.

[41] Hopf, M. and Ertl, T. Hardware Based Wavelet Transformations. In: *In Erlangen Workshop 99 on Vision, Modeling and Visualization*. 1999.

[42] Hopf, M. and Ertl, T. Hardware Accelerated Wavelet Transformations. In: Leeuw, W. C. de and Liere, R. van, ed. *Data Visualization 2000*. Vienna: Springer Vienna, 2000, p. 93–103. ISBN 978-3-7091-6783-0.

[43] Hsia, C. H., Guo, J. M., Chiang, J. S. and Lin, C. H. A novel fast algorithm based on SMDWT for visual processing applications. In: *IEEE International Symposium on Circuits and Systems*. IEEE Computer Society, May 2009, p. 762–765. DOI: 10.1109/ISCAS.2009.5117860. ISSN 0271-4302.

[44] Ikuzawa, T., Ino, F. and Hagihara, K. Reducing Memory Usage by the Lifting-Based Discrete Wavelet Transform with a Unified Buffer on a GPU. *J. Parallel Distrib. Comput.* USA: Academic Press, Inc. july 2016, vol. 93, C, p. 44–55. DOI: 10.1016/j.jpdc.2016.03.010. ISSN 0743-7315. Available at: https://doi.org/10.1016/j.jpdc.2016.03.010.

[45] Iwahashi, M. and Kiya, H. A new lifting structure of non separable 2D DWT with compatibility to JPEG 2000. In: *Acoustics Speech and Signal Processing*. IEEE, 2010, p. 1306–1309. DOI: 10.1109/ICASSP.2010.5495427. ISSN 1520-6149.

[46] Iwahashi, M. Four-band decomposition module with minimum rounding operations. *Electronics Letters*. 2007, vol. 43, no. 6, p. 27–28. DOI: 10.1049/el:20073479. ISSN 0013-5194.

[47] Iwahashi, M. and Kiya, H. Non Separable Two Dimensional Discrete Wavelet Transform for Image Signals. In: *Discrete Wavelet Transforms – A Compendium of New Approaches and Recent Applications*. InTech, 2013. DOI: 10.5772/51199. ISBN 978-953-51-0940-2.

[48] Kucis, M., Barina, D., Kula, M. and Zemcik, P. 2-D Discrete Wavelet Transform Using GPU. In: *International Symposium on Computer Architecture and High Performance Computing Workshop*. IEEE, October 2014, p. 1–6. DOI: 10.1109/SBAC-PADW.2014.13. ISBN 978-1-4799-7014-8.

[49] Kula, M., Barina, D. and Zemcik, P. New Non-Separable Lifting Scheme for Images. In: *IEEE International Conference on Signal and Image Processing*. IEEE, 2016, p. 292–295. DOI: 10.1109/SIPROCESS.2016.7888270. ISBN 978-1-5090-2375-2.

[50] Kula, M., Bařina, D. and Zemčík, P. Block-based Approach to 2-D Wavelet Transform on GPUs. In: *Information Technology: New Generations*. Springer International Publishing, 2016, vol. 448, p. 643–653. Advances in Intelligent Systems and Computing. DOI: 10.1007/978-3-319-32467-8_56. ISBN 978-3-319-32467-8.

[51] Kutil, R. A Single-Loop Approach to SIMD Parallelization of 2-D Wavelet Lifting. In: *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. Montbeliard-Sochaux, France, 2006, p. 413–420. DOI: 10.1109/PDP.2006.14. ISBN 0-7695-2513-X.

[52] Laan, W. van der, Roerdink, J. B. T. M. and Jalba, A. Accelerating wavelet-based video coding on graphics hardware using CUDA. In: *Proceedings of 6th International Symposium on Image and Signal Processing and Analysis (ISPA)*. IEEE, September 2009, p. 608–613. DOI: 10.1109/ISPA.2009.5297658. ISSN 1845-5921.

[53] Laan, W. J. van der, Jalba, A. C. and Roerdink, J. B. T. M. Accelerating Wavelet Lifting on Graphics Hardware Using CUDA. IEEE. january 2011, vol. 22, no. 1, p. 132–146. DOI: 10.1109/TPDS.2010.143. ISSN 1045-9219.

[54] Liu, P., Zhang, H., Lian, W. and Zuo, W. Multi-Level Wavelet Convolutional Neural Networks. *IEEE Access*. IEEE. 2019, vol. 7, p. 74973–74985. DOI: 10.1109/ACCESS.2019.2921451.

[55] Mallat, S. A theory for multiresolution signal decomposition: the wavelet representation. IEEE. 1989, vol. 11, no. 7, p. 674–693. DOI: 10.1109/34.192463. ISSN 0162-8828.

[56] Mallat, S. *A Wavelet Tour of Signal Processing: The Sparse Way. With contributions from Gabriel Peyre*. 3rd ed. Academic Press, 2009. ISBN 9780123743701.

[57] Matela, J. GPU-Based DWT Acceleration for JPEG2000. In: *Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*. 2009, p. 136–143. ISBN 978-80-87342-04-6.

[58] Meerwald, P., Norcen, R. and Uhl, A. Cache Issues with JPEG2000 Wavelet Lifting. In: *Visual Communications and Image Processing (VCIP)*. 2002, vol. 4671, p. 626–634. SPIE.

[59] Najman, P., Kleparnik, P. and Barina, D. *Non-Separable Schemes for Discrete Wavelet Transform for Multi-Core CPUs*. VUT, 2017. Supplementary materials. Available at: http://www.fit.vutbr.cz/research/prod/?id=534.

[60] Quan, T. M. and Jeong, W.-K. A fast mixed-band lifting wavelet transform on the GPU. In: *2014 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2014, p. 1238–1242. DOI: 10.1109/ICIP.2014.7025247.

[61] Quan, T. M. and Jeong, W.-K. A fast discrete wavelet transform using hybrid parallelism on GPUs. IEEE. november 2016, vol. 27, no. 11, p. 3088–3100. DOI: 10.1109/TPDS.2016.2536028. ISSN 1045-9219.

[62] Quesada Barriuso, P., Argüello, F., Heras, D. B. and Benediktsson, J. A. Wavelet-Based Classification of Hyperspectral Images Using Extended Morphological Profiles on Graphics Processing Units. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*. IEEE. 2015, vol. 8, no. 6, p. 2962–2970. DOI: 10.1109/JSTARS.2015.2394778.

[63] Quesada Barriuso, P., Heras, D. B., Argüello, F. and Mouriño, J. C. Wavelet-based multicomponent denoising on GPU to improve the classification of hyperspectral images. In: Huang, B., López, S. and Wu, Z., ed. *High-Performance Computing in Geoscience and Remote Sensing VII*. SPIE, 2017, vol. 10430, p. 75 – 90. DOI: 10.1117/12.2277960.

[64] Rauber, T. and Runger, G. *Parallel Programming: for Multicore and Cluster Systems*. Springer, 2013. ISBN 978-3-642-37800-3.

[65] Seo, Y.-H. and Kim, D.-W. VLSI Architecture of Line-Based Lifting Wavelet Transform for Motion JPEG2000. *IEEE Journal of Solid-State Circuits*. IEEE. 2007, vol. 42, no. 2, p. 431–440. DOI: 10.1109/JSSC.2006.889368.

[66] Shahbahrami, A., Juurlink, B. and Vassiliadis, S. Improving the memory behavior of vertical filtering in the discrete wavelet transform. In: *Proceedings of the 3rd conference on Computing frontiers (CF)*. ACM, 2006, p. 253–260. DOI: 10.1145/1128022.1128056. ISBN 1-59593-302-6.

[67] Sharma, B. and Vydyanathan, N. Parallel discrete wavelet transform using the Open Computing Language: a performance and portability study. In: *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, p. 1–8. DOI: 10.1109/IPDPSW.2010.5470830.

[68] Silva, D. D. N. D., Vithanage, H. W. M. K., Fernando, K. S. D. and Piyatilake, I. T. S. *Multi-Path Learnable Wavelet Neural Network for Image Classification*. 2019.

[69] Song, C., Li, Y., Guo, J. and Lei, J. Block-based two-dimensional wavelet transform running on graphics processing unit. *IET Computers Digital Techniques*. IET. september 2014, vol. 8, no. 5, p. 229–236. DOI: 10.1049/iet-cdt.2013.0141. ISSN 1751-8601.

[70] Song, C., Li, Y. and Huang, B. A GPU-Accelerated Wavelet Decompression System With SPIHT and Reed-Solomon Decoding for Satellite Images. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*. IEEE. 2011, vol. 4, no. 3, p. 683–690. DOI: 10.1109/JSTARS.2011.2159962.

[71] Strang, G. and Nguyen, T. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, 1997. ISBN 978-0-9614088-7-9.

[72] Sweldens, W. The Lifting Scheme: A Custom-Design Construction of Biorthogonal Wavelets. *Applied and Computational Harmonic Analysis*. 1996, vol. 3, no. 2, p. 186–200. DOI: 10.1006/acha.1996.0015. ISSN 1063-5203.

[73] Taubman, D. and Marcellin, M. Springer, Boston, MA, 2002. ISBN 978-1-4615-0799-4.

[74] Tenllado, C., Lario, R., Prieto, M. and Tirado, F. The 2D Discrete Wavelet Transform on Programmable Graphics Hardware. In: *Visualization, Imaging and Image Processing Conference*. September 2004, p. 808–813.

[75] Tenllado, C., Setoain, J., Prieto, M., Pinuel, L. and Tirado, F. Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting. IEEE. 2008, vol. 19, no. 3, p. 299–310. DOI: 10.1109/TPDS.2007.70716. ISSN 1045-9219.

[76] Wang, J., Wong, T.-T., Heng, P. and Leung, C. Discrete wavelet transform on gpu. In: *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors*. August 2004.

[77] Wang, L., Liu, H., Sun, L., Zhou, M. and Zhuang, Q. A Parallel Strategies for 2-D Lifting Wavelet Transform Using GPU. In: *2019 12th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*. IEEE, 2019, p. 1–5. DOI: 10.1109/CISP-BMEI48845.2019.8965965.

[78] Wong, T.-T., Leung, C.-S., Heng, P.-A. and Wang, J. Discrete Wavelet Transform on Consumer-Level Graphics Hardware. *IEEE Transactions on Multimedia*. IEEE. 2007, vol. 9, no. 3, p. 668–673. DOI: 10.1109/TMM.2006.887994.

[79] Zhang, C., Wang, C. and Ahmad, M. O. A Pipeline VLSI Architecture for Fast Computation of the 2-D Discrete Wavelet Transform. *IEEE Transactions on Circuits and Systems I: Regular Papers*. IEEE. 2012, vol. 59, no. 8, p. 1775–1785. DOI: 10.1109/TCSI.2011.2180432.