UNIVERSITY OF OSLO
Department of Informatics

Investigating Host-Device communication in a GPU-based H.264 encoder.

Master thesis

Kristoffer Egil Bonarjee

May 16, 2012

Contents

List of Figures

List of Tables

List of Abbreviations

ALU Arithmetic Logic Unit

APU Accelerated Processing Unit

ASMP Asymmeric Multiprocess[ing/or]

B-slice Bi-predicted slice

Bs Boundary Strength

CABAC Context-based Adaptive Binary Arithmetic Coding

CAVLC Context Adaptive Variable Length Coding

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

Cb Chroma blue color channel

Cr Chroma red color channel

DCT Discreet Cosinus Transform

DFI Deblocking Filter Independency

DVB Digital Video Broadcasting

EIB Element Interconnect Bus

FIR Finite Impulse Response

FMO Flexible Macroblock Ordering

GOP Group of Pictures

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphical Processing Unit

I-slice Independent slice

JIT Just In Time

MFP Macroblock Filter Partition

MVD Motion Vector Difference

MVP Motion Vector Prediction

MVp Predicted Motion vector

P-slice Predicted slice

QP Quantization Parameter

RGB Red Green Blue

SAD Sum of Absolute Differences

SAE Sum of Absolute Error

SFU Special Function Unit

SIMT Single Instruction, Multiple Threads

SMP Symmetric Multiprocess[ing/or]

SM Stream Multiprocessor

Y Luma color channel

List of Code Snippets

Preface

Modern graphical processing units (GPU) are powerful parallel processors, capable of running thousands of concurrent threads. While originally limited to graphics processing, newer generations can be used for general computing (GPGPU). Through frameworks such as nVidia Compute Unified Device Architecture (CUDA) and OpenCL, GPU programs can be written using established programming languages (with minor extensions) such as C and C++. The extensiveness of GPU deployment, low cost of entry and high performance makes GPUs an attractive target for workloads formerly reserved for supercomputers or special hardware. While the programming language is similar, the hardware architecture itself is significantly different than a CPU. In addition, the GPU is connected through a comparably slow interconnect, the PCI Express bus. Hence, it is easy to fall into performance pitfalls if these characteristics are not taken into account.

In this thesis, we have investigated the performance pitfalls of a H.264 encoder written for nVidia GPUs. More specifically, we looked into the interaction between the host CPU and the GPU. We did not focus on optimizing GPU code, but rather how the execution and communication was handled by the CPU code. As much manual labour is required to optimize GPU code, it is easy to neglect the CPU part of accelerated applications.

Through our experiments, we have looked into multiple issues in the host application that can effect performance. By moving IO operations into separate host threads, we masked away the latencies associated with reading input from secondary storage.

By analyzing the state shared between the host and the device, we where able to reduce

the time spent synchronizing data by only transferring actual changes.

Using CUDA streams, we further enhanced our work on input prefetching by transferring input frames to device memory in parallel with the encoding. We also experimented with concurrent kernel execution to perform preprocessing of future frames in parallel with encoding. While we only touched upon the possibilities in concurrent kernel execution, the results where promising.

Our results show that a significant improvement can be achieved by focusing optimizing effort on the host part of a GPU application. To reach peak performance, the host code must be designed for low latency in job dispatching and GPU memory management. Otherwise the GPU will idle while waiting for more work. With the rapid advancement of GPU technology, this trend is likely to escalate.

Acknowledgements

I would like to thank my advisors Håkon Kvale Stensland, Pål Halvorsen and Carsten Griwodz for their valuable feedback and discussion. I would also like to thank Mei Wen and the National University of Defense Technology in China for source code access to their H.264 encoder research. This thesis would not have come to fruition without their help.

Thanks to Håvard Espeland for helping me restart the test machine after I trashed it on numerous occasions.

Thanks to my fellow students at the Simula lab and PING for providing a motivating working environment with many a great conversation and countless cups of coffee.

Thanks to my father, Vernon Bonarjee, for his support and help with proofreading.

Thanks to my mother in law, Solveig Synnøve Gjestang Søndberg, for relieving me of household chores in the final sprint of thesis work.

Finally, I would like to thank my wife Camilla and our children Maria and Victor for encouragement, motivation and precious moments of joy and relaxation.

Oslo, May 16, 2012

Kristoffer Egil Bonarjee

Chapter 1

Introduction

1.1 Background and motivation

Since the announcement of ENIAC, the first electronic general-purpose computer in 1946 [42], there has been a great demand for ever increasing processing power. In 1965, Intel co-founder Gordon Moore predicted that the amount of components, eg transistors, in an integrated circuit would double approximately every two years [43]. While Moore anticipated this development would hold for at least ten years, it is still the case today, and has been known as "Moore's law".

For decades, the advancement of processors where driven by an ever increasing clock speed. However, increasing clock speed results in higher power requirements and heat emission. While this trend has resulted in processors with multigigahertz clock frequencies, it has approached the limit of sustainable power density, also known as the power wall.

To further increase the processing power of a single processor beyond the power wall, processor manufacturers have focused on putting multiple processing cores on a single die, referred to as multi-core processors. By placing multiple processor cores on a single unit, enhanced processing power can be achieved with each core running at a lower clock speed. These individual cores are then programmed in a similar fashion as multiple identical processors, known as Symmetric Multiprocessing (SMP).

While multi-core processors increase the total computational power of the processors, they do not improve performance for programs designed for single-core processors. To take advantage of the increased computing power, it is crucial that algorithms are designed to run in parallel. While eight-core processors are a commodity today, the maximum performance of an algorithm is limited by the sum of its serial parts [10]. Thus, programmers cannot exploit modern processors without focusing on parallel workloads.

SMP allows for some scalability by increasing the amount of independent cores on each processor as well as multiple processors per machine. However, each additional processor or core further constrains access to shared resources such as system memory and data bus. The symmetry of the processors also means that each core must be of identical design, which may not yield optimal usage of die space for parallel workloads.

Asymmetric Multiprocessing (ASMP) relaxes the symmetry requirement. Hence, the independent cores of an asymmetric processor can trade the versatility of SMP for greater computing power in a more limited scope. Thus, the limited die space can be used most efficient for the purposed tasks. For instance, each x86 processing core includes support for instruction flow control such as branch prediction. However, if we take careful steps when designing our algorithms, we can make certain our code does not branch. While branch prediction is crucial for the performance of a general purpose processor, we can take the necessary steps to make it obsolete for special purpose processing cores. Thus, we free up die space that can be more efficiently used.

ASMP systems usually consists of a traditional "fat" CPU core that manages a number of simpler cores designed for high computing performance rather than versatility. The main core and the computing cores are connected through a high speed interconnect along with memory and IO interfaces. One example of such a heterogeneous ASMP architecture is the Cell Broadband Engine [11], which consists of a general purpose PowerPC core and eight computing cores called Synergistic Processing Elements, connected through the Element Interconnect Bus (EIB).

Unfortunately, ASMP processors are not as widespread as their symmetrical coun-

terparts, and cannot be considered a commodity. While the first generation of Sony Playstation 3 gaming consoles could be used as a general purpose Cell computing device, the option to boot non-gaming operating systems was later removed from future firmware updates [44].

Modern GPUs on the other hand, can be found in virtually any relatively new computer. They are massively parallel processors designed to render millions of pixel values at a fraction of a second. Frameworks such as nVidia Compute Unified Device Architecture (CUDA) (see chapter 3) and Open Computing Language (OpenCL) allows supported GPUs to be used for general purpose programming. Combined with a general purpose processor such as an Inter Core i7, a modern GPU allows us to perform massively parallel computations on commodity hardware similar to ASMP processors. However, a notable limitation compared to a fullblown ASMP processor is interconnect bandwidth. While for instance the EIB in a Cell processor has a peak bandwith of 204.8GB/s [11], a GPU is usually connected via the PCI Express bus. While the bandwith of the PCI Express bus has doubled the bandwith twice with version 2 [45] and 3 [46], the ratio between available bandwith and the computational power is increasing. Compared to the quadroupled bandwith, the number of CUDA cores have increased from 128 [47] in the first CUDA compatible Geforce 8800 GTX (using PCI Express 1.0) to 1536 [48] in the Geforce GTX 680 using PCI Express 3. This is an increase of 12 times, without taking increased clock speed, improved memory bandwith etc. into consideration.

In the case of our test machine (see table 4.1), the GPU is connected through a PCI Express v2 x16 port with an aggregate bandwith in both directions combined of approximately 16GB/s [45]. Hence, GPU applications must be designed with this limitation in mind to fully utilize the hardware. This is especially the case for high-volume, data driven workloads.

1.2 Problem statement

One task requiring substanial computational power is video encoding. As the computing power of processors has increased, video standards have evolved to take advantage of it. By using advanced techniques with higher processing requirements, the same picture quality can be achieved with lower bitrate. However, due to data dependencies, all parts of the video encoding process might not run efficiently on a GPU. This can lead to the GPU idling for prolonged times unless care is taken to design the application efficiently.

In this thesis we will investigate such performance pitfalls in a CUDA-based H.264 encoder. By cooperating with researchers from the National University of Defense Technology in China, we will work to increase GPU efficiency by analyzing and improving the on-device memory management, host-device communication and job dispatching. We will take an in-depth approach to the memory transfers to identify data that can be kept on the GPU, utilize CUDA Streams to overlap execution and transfer, as well as using conventional multi-threading on the host to make input/output operations in the background while keeping the GPU busy.

1.2.1 Limitations

We limit our focus on the host-device relationship, so we will not go into topics such as device kernel algorithms or design unless specified otherwise.

1.3 Research Method

For this thesis, we will design, implement and evaluate various improvements on a working H.264 encoder. Our approach is based on the *Design* methodoligy as specified by the ACM Task Force on the Core of Computer Science [12].

1.4 Main Contributions

Through the work on this master thesis, we have designed and implemented a number of proposed improvements to the `cuve264b` encoder. These proposals include asynchronos IO through separate readahead and writeback threads, more efficient state synchronization by only sending relevant data, removal of reduntant frame transfers, and usage of CUDA Streams to perform memory transfers and preprosessing work in parallel.

With the exception of our deblocking work, we have reduced the encoding time by 23.3% and GPU idle time by 27.3 % for the *tractor* video sequence. Note that the reduction in idle time is probably even higher, as the CUDA Visual Profiler [49] does not currently support concurrent kernels [50]. We have also identified additional work to further reduce the GPU idle and consequent runtime. While we performed our experiments on a video encoder, our findings may apply to any GPU-offloaded application, especially for high-volume, data driven workloads.

In addition to the work covered by our experiments, we initially ported the encoder to Unix[1]. In connection with our porting efforts, we also added getopt support and a progress bar. Addionally, we made the encoder choose the best GPU available in multi-GPU setups, so it could be debugged with the CUDA debugger.

1.5 Outline

The rest of this thesis is organized as follows; In chapter 2, we give an introduction to video coding and the H.264/MPEG4-AVC Standard. In chapter 3, we introduce GPGPU programming and the nVidia CUDA architecture, used in our evaluations. Chapter 4 introduces the `cuve264b` encoder, the software basis for our experiments, as well as our evaluation method and testbed specifications. In chapter 5, we start our experiments with our investigation of readahead and writeback. Chapter 6 continues with our analysis and proposals to reduce redundancy in host-device memory

[1] While we expect it to work on Mac OSX due to its POSIX support, we have only tested it on Linux.

transfers. In chapter 7, we investigate how we can utilize CUDA Streams to extend our Readahead work to on-device memory. Chapter 8 completes our experiments with our work on a GPU-based deblocking filter. In chapter 9 we discuss our results and lessons learned. Chapter 10 concludes our thesis.

Chapter 2

Video Coding and the H.264/MPEG4-AVC standard

2.1 Introduction

H.264/MPEG4-AVC is the newest video coding standard developed by the Joint Video Team (JVT), comprising of experts from Telecommunication Standardization Sector (ITU-T) Study Group 16 (VCEG) and ISO/IEC JTC 1 SC 29 / WG 11 Moving Picture Experts Group (MPEG). The standard was ratified in 2003 as ITU-T H.264 [51] and MPEG-4 Part 10 [52]. In the rest of the thesis, we will refer to it simply as *H.264*.

Older standards such as ITU-T H.262/MPEG-2 [53] has been at the core of video content such as digital television (both SD and HD) and media such as DVDs. However, an increasing number of HD content require more bandwidth on television broadcast networks as well as storage media. While broadcast networks can handle a limited amount of MPEG-2 coded HD streams, more efficient video coding is necessary to scale. At the same time, emerging trends such as video playback on mobile devices require acceptable picture quality and robustness against transmission errors under bandwidth constraints. H.264 allows for higher video quality per bit rate than older standards and has been broadly adopted. It is used in video media such as Blu-ray, broadcasting such as the Digital Video Broadcasting(DVB) standards, as well as online

streaming services like Youtube.

The H.264 standard improves the video quality by enhancing the coding efficiency. To support a broad range of applications from low powered hand held devices to digital cinemas, it has been divided into different profiles. The profiles support different features of the standard, from the simplest *constrained baseline* to the most advanced *High 4:4:4 P*. This allows implementors to utilize the standard without carrying the cost of unwanted features. For instance, the *constrained baseline* profile lacks many features such as B-slices (see section 2.2.4), CABAC (see section 2.5.3), and interlaced coding. While this profile cannot code video as efficient as the more advanced profiles, the lack of features makes it easier to implement. This makes it well suitable for low-margin applications such as cellular phones.

A modern video coding standard uses a multitude of approaches to achieve optimal compression performance, such as:

- Removing unnecessary data without influencing subjective video quality. By removing image properties that cannot be seen by the human eye, there is less data to compress without perceptible quality loss. This will be explained in the following subsection.

- *Removing data that influences subjective video quality on a cost/benefit basis.* By removing fine-grained details from the images, we can reduce the amount of data stored. However, as this impacts the video quality, it is a matter of data size versus picture quality. This step, known as *quantization*, will be covered in section 2.3.

- *Reduce spatial redundancy across a frame.* As neighboring pixels in a picture often contain similar values, we can save space by storing differences between neighbors instead of the full values. This is known as intra-frame prediction, and will be elaborated in section 2.2.2.

- *Reduce temporal redundancy across series of frames.* Comparable to the relationship between spatial neighboring pixels, there often exist a same kind of relationship between pixels in different frames of a video sequence. By referring to pixels of a

previously encoded frame, we store the differences in pixel value instead. This is known as inter-frame prediction, and will be detailed in section 2.2.3.

- *Use knowledge of the data to efficiently entropy-code the result.*When all the previous steps have been completed, the nearly-encoded video will contain certain patterns in the bitstream. The last part of the encoding process is to take advantage of these patterns to further compress the bitstream. Entropy-coding will be covered in section 2.5.

In the rest of this chapter, we will explain the techniques mentioned above, and how they are used in the H.264 standard.

2.1.1 Color Spaces and human perception

The Red Green Blue Color space

Most video devices today, both input devices such as cameras and display equipment such as HDTV's, uses the Red Green Blue (RGB) color space. RGB, named after its three color channels; red green and blue, and convey the information of each pixel by a triplet giving the amount of each color. For instance, using 8bit per channel, the colors red will be (255,0,0), green (0,255,0), blue (0,0,255), and cyan (0,255, 255).

RGB is well suited for both capturing and displaying video. For instance, the pixel values can be mapped to the lighting sources in displays, such as phosphor dots in CRT monitors or sub-pixels in LCD panels. However, the three RGB channels carry more information than the human vision can absorb.

Limitations of the human vision

The human vision system is actually two distinct systems, from the cells in the retina to the processing layers in the primary visual cortex. The first one is found in all mammals. The second is a complimentary system we share with other primates. The mammal system is responsible for our ability to register motion, depth and position, as well

as our overall field of vision. It can distinguish acute variation of brightness, but it does not detect color. The primate system is responsible for detecting objects, such as facial recognition, and is able to detect color. However, it has a lower sensitivity to luminance and is less acute [13]. As a result, our ability to detect color is at a lower spatial resolution compared to our detection of brightness and contrast.

Knowing this, we can reduce the amount of color information in the video accordingly. We loose information, but because of the limits of the human vision, the subjective video quality experienced will be the same.

Y'CbCr

To take advantage of the human vision in terms of video coding, we need a way to reduce the resolution of the color information while keeping the brightness and contrast intact. As we noted above, this is not possible with RGB, where the pixel values are given solely by their color. However, we might use the derivative Y'CbCr colorspace. It works in an additional fashion similar to RGB, and transforming from the one to the other involves few computations.

Instead of identifying a pixel value by its composition of amounts red green and blue, it is identified by its brightness and color difference. Color difference is the difference between brightness (Luma[1]) and the RGB colors. Only the Chroma blue(Cb) and Chroma red(Cr) is transmitted, as Chroma green (Cg) $= 1 - (Cb + Cr)$. With brightness separated from color, we can treat them separately, and provide them in different resolutions to save space.

Chroma sub-sampling

Using a lower resolution for the chroma components is called chroma sub-sampling. The default form of sub-sampling in the H.264 standard is `4:2:0`. The first number,

[1]Note that the term Luma must not be confused with Luminance, as the nonlinear Luma is only an approximation of the linear Luminance [14].

4, is reminiscent to the legacy NTSC and PAL standards, and represents the Luma sample rate. The second number, 2, indicates that Cb and Cr will be sampled at half the horizontal sample rate of Luma. Originally, the second and third digits denoted the horizontal subsample rate of Cb and Cr respectively, as the notation predates vertical sub-sampling., Today however, a third digit of zero now indicates half ivertical sample rate for both Cb and Cr. (For a more thorough explanation, the reader is referred to [14].)

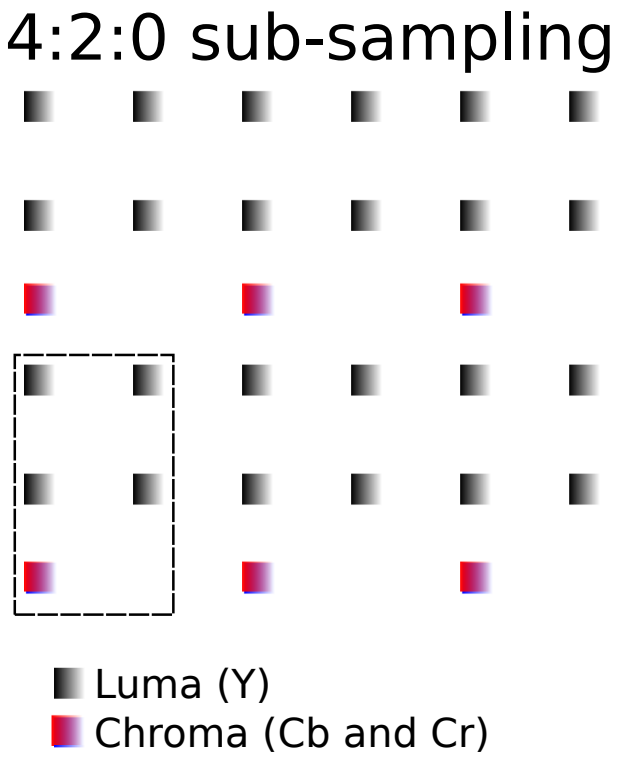

Figure 2.1: 4:2:0 Sub sampling. For each 4 Luma samples, only one pair of Chroma samples are transmitted.

Using 4:2:0, we only use half of the luma sample size to store both chroma components. As shown in figure 2.1, the chroma samples are only stored for every fourth luma sample. H.264 supports richer chroma sampling as well, through the Hi422 and *Hi444P* profiles, which supports 4:2:2 and 4:4:4 respectively, as well as higher bitrate per input pixel [51].

2.2 Frames, Slices and Macroblocks

In H.264, each picture in the video to be encoded can be divided into multiple independent units, called slices. This can make the resulting video more robust, as package loss will only suffer the slices that looses information, while keeping the others intact. It also makes it possible to process different slices in parallel [15]. The number of slices to use is left for the encoder to decide, but it is a trade-off between robustness against transmission errors and picture quality. Slicing reduces the efficiency of prediction, as redundancy over slice boundaries cannot be exploited. This reduces the picture area available for reference, and may reduce the efficiency of both spatial and temporal prediction. When macroblocks must be predicted from sub-optimal prediction blocks, the difference in pixel values, the residual, will increase. This results in an increased bitrate to achieve identical objective picture quality [16], as the larger residuals needs additional storage in the bitstream. On the other hand, independent slices increase the robustness of the coded video, as errors in one slice cannot propagate across slice boundaries.

Slices can be grouped together in groups called slice groups, referred to in earlier versions of the draft standard and in [1] as Flexible Macroblock Ordering (FMO). When using only one slice group per frame, the macroblocks are (de)coded in raster order. By using multiple slice groups, the encoder is free to map each macroblock to a slice group as deemed fit. There are 6 predefined maps, but it is also possible to explicitly define the slice group corresponding to each macroblock. For frames with only one slice, the terms can be interchanged, but we will continue to use the term slice in the rest of the chapter. In inter-prediction, other slices refers to the same slice in another frame, not a different slice in the same frame.

2.2.1 Macroblocks

After a frame has been divided into one or more slices, each slice is further divided into macroblocks. They are non-overlapping groups of pixels similar to the pieces of a

jigsaw puzzle, and they form the basic work units of the encoding process.

Introduced in the H.261 standard [54], macroblocks was always 16x16 pixels in size, with 8x8 for the chroma channels. In H.264, however, they can be a divided in a plethora of sizes; 16x8, 8x16, 8x8, 8x4, 4x8 and 4x4, depending on the prediction mode. This partitioning makes it possible for the encoder to adapt the prediction block size depending on the spatial and temporal properties of the video. For instance, 16x16 blocks might be used to predict large homogeneous areas, while 4x4 sub-blocks will be applicable for heterogeneous areas with rapid motion.

H.264 uses two types of macroblocks depending on their prediction mode. Intracoded macroblocks reference neighboring blocks inside the current slice, thereby exploiting spatial redundancy by only storing the residual pixel differences. Instead of storing the whole block, only the difference between the block and its most similar neighbor must be transmitted. Intercoded macroblocks references blocks in other slices, exploiting temporary redundancy. This allows us to take advantage of similar blocks in both space and time.

Slices made up of only intra-coded macroblocks are referred to as independent slices, as they do not reference any data from other slices. We will elaborate on independent slices in the following subsection.

Slices containing inter-coded macroblocks are known as predicted slices. As inter-coded macroblocks refer to similar macroblocks in prior encoded slices, a predicted slice cannot be used as a starting point for the decoding process. Predicted slices will be discussed in subsection 2.2.3.

An extension of predicted slices, Bi-predicted slices, support predicting each block from two different reference frames, known as bi-prediction. They are only available in the extended or more advanced profiles, and is further explained in subsection 2.2.4.

Lastly, the extended profile also supports two special *switching* slices, briefly explained in section 2.2.5.

2.2.2 Independent slices

Independent (I) slices are the foundation of the encoded video, as they are the initial reference point for the motion vectors in Predicted and Bi-predicted slices. They form random access points for the decoder to start decoding the video, as well as a natural point in the stream for fast forward/rewind.

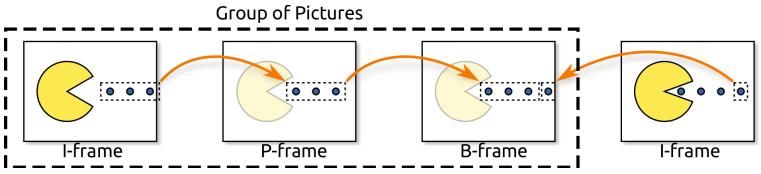

Figure 2.2: I-, P- and B-frames.

An I-slice and the P- and B-slices that references it, is known as a group of pictures (GOP). See figure 2.2.

H.264 uses intraslice predictions to reduce the amount of data needed for each macroblock. If a macroblock is similar to one of its neighbors, we only need to store the residuals needed to predict the pixel values from it.

In prior standards, such as H.262/MPEG2 [53], I-slices where encoded without any prediction. Thus, they did not take advantage the spatial redundancy in the slices.

Intraslices also supports a special mode called `I_PCM`, where the image samples are given directly, without neither prediction nor quantization and transform-coding (described in detail in section 2.3.

Intraslice prediction in H.264 is implemented as intra-coded macroblocks with a set of predefined modes. Depending on the channel to be predicted and the spatial correlations in the slice, the encoder chooses the mode resulting in the minimal residual data. For instance, it might summarize the absolute difference in pixel values for the different modes, and select the one with the least number. This test is often referred to as the Sum of Absolute Error (SAE) or Sum of Absolute Differences (SAD) [2].

For the luma channel, the macroblocks are either predicted for 16x16 macroblocks as a whole, or of each 4x4 sub-block. Chroma blocks on the other hand, are always coded as

8x8. As the prediction mode for a chroma block is only signaled once for both channels, Cb and Cr always share the same prediction mode.

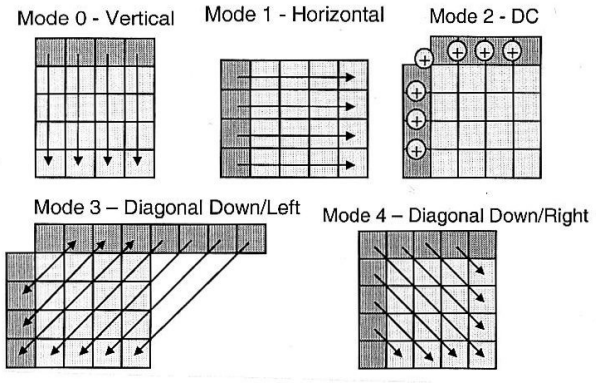

Figure 2.3: A selection of the available 4x4 intra-modes [1].

4x4 sub-blocks can be predicted in nine different ways, some of which are shown in figure 2.3. Eight of these represent a direction; such as mode 0 - vertical and mode 1 - horizontal, where the arrows shows how the block will be predicted. Mode 2 - DC is a special case where all the pixels in the block is predicted from a mean value of upper and left hand samples. As an example, a Mode 0 - vertical prediction uses the last row of pixels from the neighboring block directly above. The residual block is then formed by calculating the difference between the pixel values in the reference row and every row in the current block. In the event of slice boundaries, the modes crossing the boundaries will be disabled, and the input for the DC mode will be similarly reduced.

For smooth areas, 16x16 prediction might yield less signaling overhead than four 4x4 blocks combined. There are four 16x16 modes available, of which the first three are similar to the 4x4 ones. Namely mode 0 - vertical, mode 1 - horizontal and mode 2 - DC. Mode 3 - Plane, uses the upper and left pixels as input in a linear plane function, resulting in a prediction block with a smooth transition between them [2].

The chroma blocks have similar prediction modes as the 16x16 luma blocks. However, the ordering is different: Mode 0 is DC, followed by horizontal, vertical and plane.

The chosen mode for each (sub)block is signaled in the bit stream along with the prediction residuals. However, as there often is a relationship among neighboring blocks, the standard supports an implicit prediction mode called `most_probable_mode`. If

both the block above and to the left of the current block is predicted using the same mode, it will be the most probable mode for the current block. If they differ, the most probable mode defaults to DC. The implicit mode selection makes it possible to signal the most probable mode by only setting the `use_most_probable_mode` flag. Otherwise, `use_most_probable_mode` is nilled, and the `remaining_mode_selector` variable signals the new mode.

2.2.3 Predicted slices

Similar to the spatial redundancy exploited in I-slices, there also often exist a correlation between macroblocks of different slices. Using a reference slice as a starting point, we may further reduce the space requirements by storing the pixel difference between a macroblock and a similar macroblock in another slice. To accomplish this, the H.264 standard use motion vectors. A motion vector allows a macroblock to be predicted from any block in any slice available for reference in memory. Instead of limiting the prediction options to a limited set of modes, a motion vector explicitly points to the coordinates of the referential block. This gives the encoder great flexibility to find the best possible residual block.

Akin to the different prediction block sizes for intra-coded macroblocks, inter-predicted blocks support different sizes. Depending on the amount, speed and extensiveness of motion in the sequence, the best prediction block size for a given macroblock can vary. Picking the best prediction block size to code the motion of a block is known as motion-compensation.

To facilitate efficient motion-compensation for both rapid and slower movement, H.264 supports macroblock partitioning. Each macroblock can be divided into partitions of either the default 16x16, two 16x8, two 8x16 or four 8x8 sub-blocks. The 8x8 partitions can be further divided up into 8x8, 8x4, 4x8 or 4x4 blocks. These sub-blocks are then motion-compensated separately, each requiring a separate motion vector in the resulting bitstream. However, each sub-block might yield a smaller residual. On the other hand, homogeneous regions can be represented by larger blocks with fewer mo-

tion vectors. Depending on the amount and speed of movement in the input video, the encoder can then find the combination of block sizes that minimize the combined signaling of motion vectors and residuals.

Motion vector search

For each (sub)macroblock to be inter-predicted, the encoder must find a similar sized block in a prior coded reference slice. However, as H.264 decouples coding- and display order, the encoder is free to code a slice suitable for motion compensation earlier than it is actually displayed on screen. It also supports pinning slices as long term reference slices that might be used for motion compensation far longer than display purposes would suggest.

Similar to earlier video coding standards from ITU-T and ISO/IEC, the scope of the H.264 standard only covers the bit stream syntax and decoder process; An encoder is valid as long as it outputs a bit stream that can be decoded properly by a conformant decoder; The actual motion search is not defined in the standard. In fact, the H.264 syntax supports unrestricted motion vectors, as the boundary pixels will be repeated for vectors pointing outside the slice [17].

The extensiveness of the motion vector search depends on the use case of the encoder. For instance, a real time encoder might use a small search area to keep deadlines, combined with multiple slices to make the stream robust. Afterward, the video can be encoded offline with a thorough full-slice motion vector search without realtime requirements.

While the procedure of motion vector search is left to the encoder implementation, the general approach is to search a grid surrounding the identical position in the reference slice. Each block with its potential motion vector in the search window is evaluated. The optimal solution is the one which gives the lowest possible residuals. For instance, an encoder might calculate the SAD over all the potential blocks and select the one yielding the lowest sum.

H.264 also supports a special inter-predicted macroblock type `P_Skip`, where no mo-

tion vector or prediction residual is transmitted [15]. Instead, the macroblock is predicted as a 16x16 macroblock with a default motion vector pointing to the same position (i.e. 0,0) in the first available prediction slice. Macroblocks without motion can then be transmitted by only signaling the metadata of macroblock type, while skipping the actual data.

Half- and Quarter-pixels

The motion vectors in H.264 is given with quarter pixel accuracy. To support this within the finite resolution of a sampled video frame, the quarter-pixels (quarter-pels) must be interpolated. For the luma channel, this is done in a two step procedure; we first interpolate half-pixels (half-pels) which then again get interpolated to form quarter-pels.

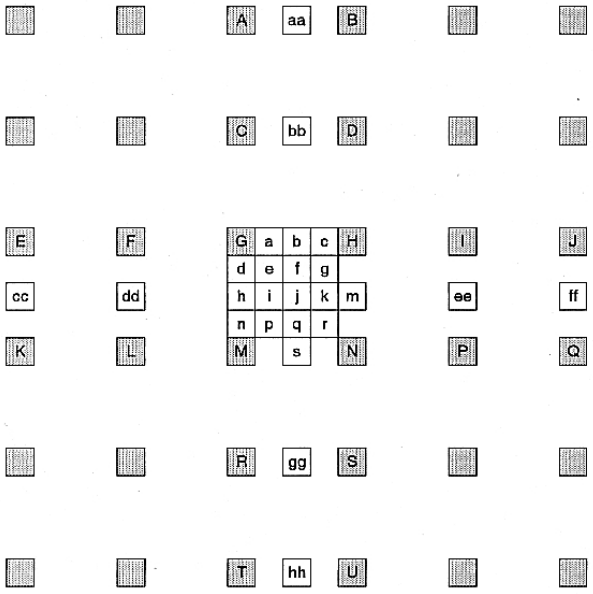

Figure 2.4: Half- and Quarter-pixels [1].

To interpolate a half-pel, a six-tap Finite Impulse Response (FIR) filter [2] is used, in which three pixels on either side is weighted to calculate the half-pel value. Figure 2.4 shows a block, a grid of pixels (in gray) with a selection of interpolated half-pels, and quarter-pels (white). To determine the value of half-pel `b` (shown between pixels `G` and `H`), the FIR filter would be applied as follows [2]:

$$b = \frac{E - 5F + 20G + 20H - 5I + J}{32} \tag{2.1}$$

After the half-pels have been calculated, the quarter-pels are calculated by means of an unweighted linear interpolation. Depending on position, they are either interpolated horizontal, vertically or diagonally. In figure 2.4, `a` would be horizontally interpolated between `G` and `b`, `d` vertically between `G` and `h`, while `e` would be interpolated diagonally between `G` and `j`.

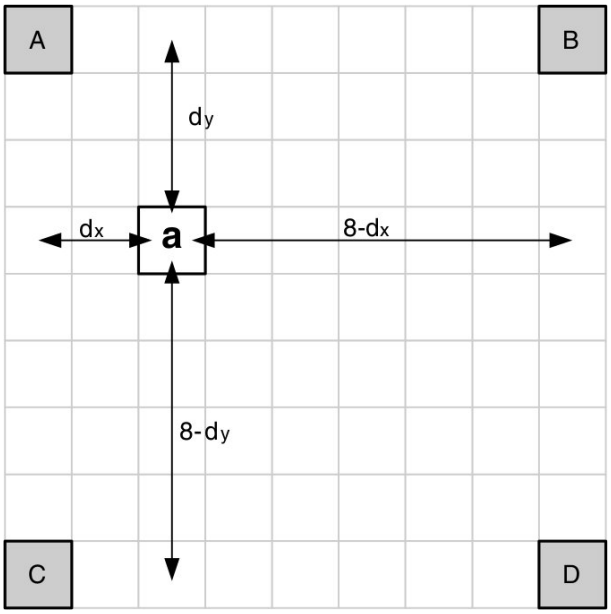

Figure 2.5: Octa-pixel interpolation [2].

Due to the chroma sub sampling, the resolution of the chroma channels are halved in both dimensions. To support the same motion vector accuracy as the luma channel, we must calculate octa-pixels (octa-pels) for Cb and Cr. This is done in one step, where each octa-pel to be calculated in linearly interpolated between 4 chroma pixels. Each chroma pixel is weighted according to its distance from the octa-pel. For instance, the octa-pel `a` in figure 2.5 is calculated by [2]:

$$a = \text{round}\left(\frac{(8 - d_x)(8 - d_y)A + d_x(8 - d_y)B + (8 - d_x)d_yC + d_xd_yD}{64}\right) \tag{2.2}$$

We substitute d_x with 2 and d_y with 3, which gives us

$$a = \text{round}\left(\frac{30A + 10B + 18C + 6D}{64}\right) \tag{2.3}$$

The higher-resolution motion vectors allow us to more precisely represent the motion between the slices. For instance, if two adjacent macroblocks yields the fewest residuals, but both have differences in its direction, the higher resolution enables us to generate a macroblock representing their mean values.

Given the large number of potential motion vectors, quadrupled by the quarter pixel resolution, an exhausting search for motion vectors will require much processing time. Finding more efficient motion search algorithms has resulted in numerous research efforts. For instance, the popular open source X264 supports a range of different algorithms, including Diamond search [18] and Uneven MultiHexagon search [55].

Motion vector prediction

Having performed a motion vector search over the slice, the individual motion vectors often have high correlation, similar to the spatial redundancy exploited in intra-coded macroblocks.

To take advantage of this, H.264 uses Motion Vector Prediction (MVP), to predict the current motion vector. A predicted motion vector (MVp) is made from certain neighboring motion vectors and a residual motion vector difference (MVD). As motion vector prediction is carried out as defined in the standard, only the MVD needs to be encoded in the bitstream.

To calculate the MVp for a macroblock, the median of the block above, to the left and above and to the right is calculated as shown in figure 2.6 for the block E. If any of these are of smaller partition size than the current block, the topmost of the ones to the left, and the leftmost of the ones above is used. If any of the blocks are missing, for instance if the current macroblock is on a slice border, the median is calculated for the remaining blocks.

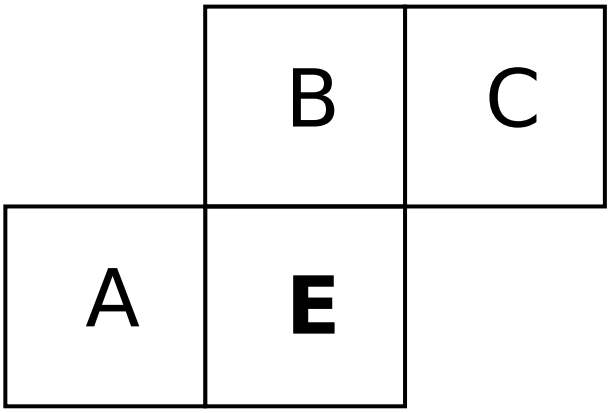

Figure 2.6: The current motion vector E is predicted from its neighbors A, B and C.

A special case is partitions of size 16x8 or 8x16. In the case of 16x8, the upper partition is predicted from the block above, while the lower partition is predicted from the block to the left. Similarly, the leftmost partition of a 8x16 partitioned macroblock is predicted from the block to the left, while the rightmost block is predicted from the block above and to the right.

2.2.4 Bi-predicted slices

Bi-predicted slices (B-slices) are part of the Main profile and extend P-slices with the ability to reference two slices; Macroblocks might use motion vectors pointing to either slice, broadening the potential to exploit temporal redundancy with two directions. In addition, B-slices supports bi-prediction by using two independent motion vectors to predict each block. The weight of each motion vector can be specified by the encoder. Weighted prediction is also supported in P-slices, where it can be used to better code certain special cases such as fade to black. By default, the prediction samples are evenly averaged. H.264 also supports implicit weighted prediction, whereby the weighting factor is calculated based on the temporal distance between each reference slice and the current slice [2].

B-slice macroblocks can also be encoded in *Direct* mode, in which no motion vector is transmitted. Instead, the motion vector is calculated on the fly by the decoder. A direct bi-predicted macroblock without prediction residuals is also referred to as a `B_Skip`

macroblock, similar to the `P_Skip` macroblock we detailed in section 2.2.3.

2.2.5 Switching I and Switching P slices

In addition to the already mentioned slice types, H.264 also supports two special purpose slices, Switching Independent and Switching Predicted [19] as part of the extended profile. Using switching slices, it is possible to change bitstream, such as the same content at a different resolution, without synchronizing on the next or previous I-slice. For instance, a videostream displayed on a mobile phone might switch to a lower resolution if 3G connectivity is lost. The standard have also been annexed with the Scalabe Video Coding Extension to improve its support in such scenarios [20].

2.3 Transform coding and Quantization

The intra- and inter-predictions detailed in previous sections greatly reduces the redundancy in the information, but to achieve high compression ratio, we need to actually remove some data from the residuals.

By using transform coding with its basis in Fourier theory, we may transform the residual blocks to the frequency domain. Instead of representing the residual for each pixel, a transform to the frequency domain gives us the data as a sum of coefficients to a transform-dependent continuous function. Instead of spatial pixel differences, we have transformed the values to a range of coefficients, ranging from low to high-frequency information. The first coefficient is the mean value of the transformed signal, known as the DC coefficient.

By removing high-frequency coefficients from the transformed result, we may represent the data in a more compact form with a controlled loss of information. The transform to and from the frequency domain is by itself lossless [21]. However, as the transforms often operate on real numbers (and even imaginary numbers for the Fourier transform), rounding the result to integer values may lead to inaccurate reconstruction.

Most known transforms in this area are the Fourier transform, Discreet cosine transform (DCT), Walsh-Hadamard transform and Karhunen-Loève transform. The DCT is not the most efficient to pack information. However, it gives the best ratio between computational cost and information packing. This property has established DCT as an international standard for transform coding [22].

In H.264, DCT is not used directly. Instead, it uses a purpose-built 4x4 integer transform that approximates the DCT while providing key properties essential to the efficiency and robustness of the transform [23]. Most notably, it gives integer results. Due to the DCT producing real numbers, floating point operations and subsequent rounding may produce slightly different results between different hardware, and the inverse-transformed residuals will be incorrect. This is known as drifting. As both inter- and intra-coded slices depend on the correctness of the residuals, the inverse transform must provide exact results.

In earlier standards without intra-prediction, this was solved by periodic I-frame refreshes. However, with intra-prediction, drifting may occur and propagate within the I-slice itself. By using an integer transform without risk of drifting, the standard guarantees that the results will be reliable for reference. Another important property of the transform is that is requires less computation than DCT, as it only requires addition and binary shifts [23].

While H.264 uses the described integer transform for most of the residual blocks, it uses the Walsh-Hadamard transform on the four DC components in a 16x16 intra-macroblock, as well as the 2x2 chroma DC coefficients of any block. This second transform of DC coefficients often improves the compression of very smooth regions typically found in 16x16-mode intra-coded macroblocks and the sub-sampled chroma blocks [15].

Quantization

After transforming the residuals into the frequency domain, the next step is quantization. Depending on the desired compression level, a number of high frequency

coefficients will be removed. By removing the frequencies, less information must be stored in the bitstream, which results in very efficient compression. On the other hand, the lost data cannot be restored by the decoder, thereby degrading the output. Hence, efficient quantization does not solely depend on the compression efficiency, but also fine-grained control over the process.

In H.264, quantization is controlled with the Quantization Parameter (QP). It can range from 0 to 51 for luma, and 39 for chroma channels. For every sixth step of QP, the level of quantization doubles. Compared to earlier standards, H.264 allows for better control for near-lossless quantization. For instance, a zeroed QP in H.263+ corresponds to 6 in H.264, giving more control over the information loss [23]. The fine granularity of QP allows for encoding video for a range of different scenarios, depending on the cost and benefit of perceived video quality versus the cost of data transmission. For instance, neither the expected video quality nor associated data rate cost will be the same for the offline playback of a bluray movie as streaming of a live soccer match over a 3G connection.

Depending on the QP, multiplying the coefficients with the quantization table will result in coefficients clipped to zero. When the decoder decodes the slice, it will in turn inverse the multiplication. However, the zeroed coefficients will not be restored, and some details in the picture will be lost. Depending on the grade of quantization, errors will be introduced in the decoded picture. Figure 2.7 shows similar quantization artefacts in a JPEG image. When the inverse block transform is performed with fewer coefficients, details will be lost. Thus, the text in the upper left corner is not readable. As the quantization is performed per block, the edges between the restored blocks will degrade. For instance, the lines in the wall looks jagged, as there is not enough information in the image to restore the transition between the transformed blocks. We will return to how H.264 reduces the extensiveness of these errors when we explain the in-loop deblocking filter in 2.4.

After quantization, the coefficients clipped to zero will introduce redundancy that will be exploited in the coding step explained in section 2.5. Before that, however, we will need to re-arrange the data from raster to a pattern know as "zig-zag" as shown in

figure 2.8.

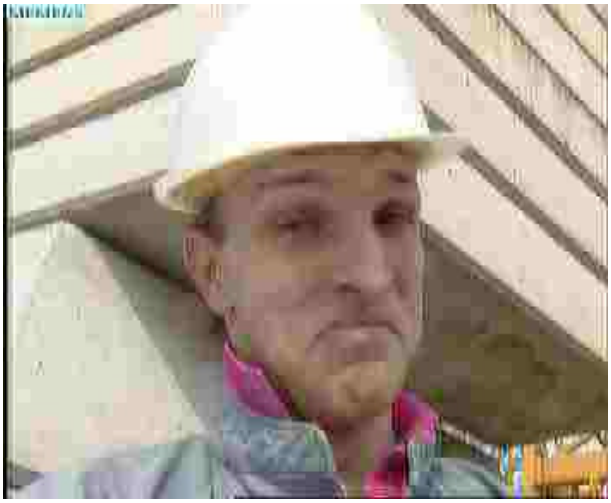

Figure 2.7: A JPEG image showing clear block artefacts.

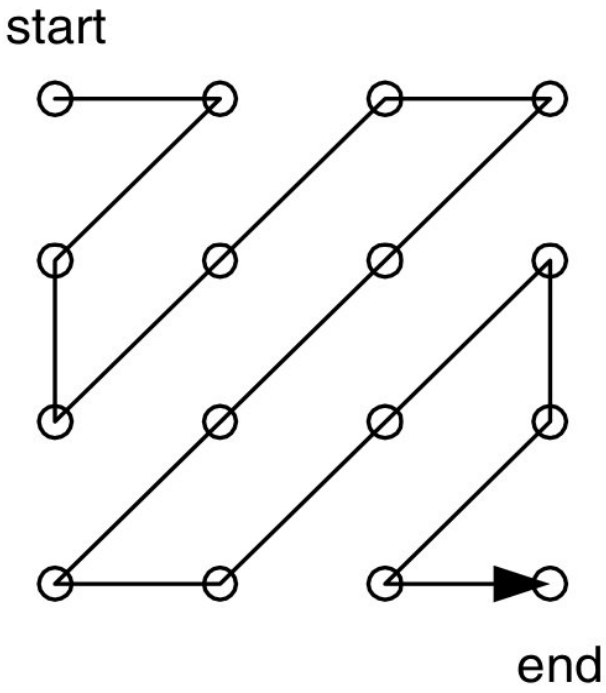

Figure 2.8: ZigZag pattern of a 4x4 luma block [2].

This will order the coefficients so that the zeroed values are grouped together. Having reordered the coefficients, we can compress them more efficiently as continuous runs of zeroes, instead of individual values scattered across the raster order.

2.4 Deblocking

To minimize the blocking artefacts introduced by the quantization step, H.264 mandates an in loop deblocking filter. As such, the deblocking filter is a part of both the encoding and decoding process, as opposed to deblocking as a post-processing step done by the decoder. This ensures a level of quality, as the encoder can guarantee the quality delivered to the end user by a conforming decoder. It also works more efficiently than a post-processing filter, as it greatly reduces the propagation of blocking artefacts through motion vectors [9]. It also reduces the residual size, as the smoothing of artefact results in closer resemblance to the original pixels.

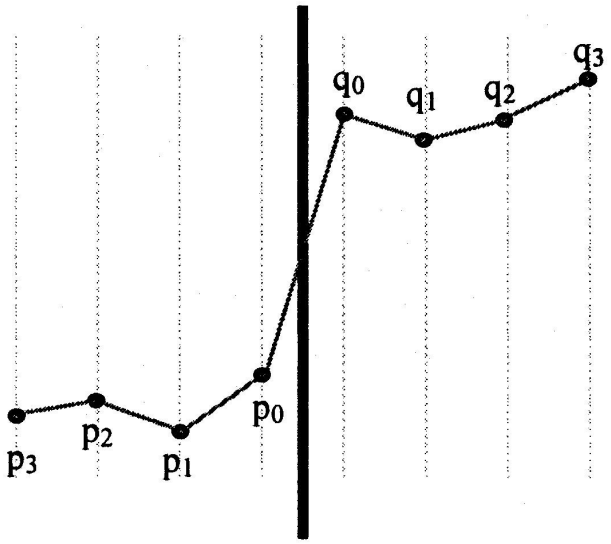

Figure 2.9: Plotted pixel values of an edge showing typical signs of blocking artefacts. As the difference in pixel value between the edges of adjacent blocks p_0 and q_0 is much higher than the differences between $p_0 - p_4$ and $q_0 - q_4$, it is likely a result of quantization [9].

The purpose of the deblocking filter is to evaluate the edges between the 4x4 luma transformation blocks and 2x2 chroma blocks to determine if there is a block artefact between them, as opposed to any other form of edge due to actual picture content. A synthetic edge from blocking artefacts is identified by a pronounced spike in pixel values between the edge pixels that does not continue across the interior samples, as seen in figure 2.9. The sharp spike between pixel values p_0 and q_0 does not propagate

Condition	Bs
One of the blocks is intra-coded and on a macroblock edge	4
One of the blocks is intra-coded	3
One of the blocks has coded residuals	2
Difference of block motion $\geq$ 1 luma sample distance	1
Motion compensation from different reference slices	1
Else	0

Table 2.1: Determining Boundary strength. Data from [9].

in the interior pixels $p_1 - p_3$ and $q_1 - q_3$. Edges that fall into such a pattern is smoothed over by interpolation with interior pixels.

The deblocking filter in H.264 works in a two-step process. First, each edge is given a score, known as Boundary strength(Bs). It is based on the type of macroblock and how it has been predicted. It is then given a value between 0 and 4, determining the expected amount of filtering necessary. For edges with a Bs of 0, deblocking will be disabled. Edges with a Bs between 1-3 might be filtered with the normal filter, while edges between intra-coded macroblocks gets a score of 4 and receives extra strong filtering. The conditions for the assignment of Bs is given in table 2.1. Each condition is tested as ordered in the table, and the corresponding boundary strength is chosen from the first matching condition.

After each edge has been assigned a Bs, the pixel values for those with Bs$\geq$ 1 is analyzed to detect possible blocking artefacts. As the amount of block artefacts depends on the amount of quantization, the tested threshold values, $\alpha(Index_A)$ and $\beta(Index_B)$, is dependent on the current QP. $Index_A$ and $Index_B$ is derived from QP, with optional influence from the encoder. The possible values for α and β is predefined in the standard, and originates from empirical data that should ensure good results for a broad range of different content. For instance, QP values close to zero will result in very low data loss, so the deblocking filter can safely be disabled.

The pixel values are then tested, and the edge is filtered if the following conditions hold:

$$|p_0 - q_0| < \alpha(Index_A) \tag{2.4}$$

$$|p_1 - q_1| < \beta(Index_B) \tag{2.5}$$

$$|p_2 - q_2| < \beta(Index_B) \tag{2.6}$$

where $\beta(Index_B)$ generally is significantly smaller than $\alpha(Index_A)$.

If the above conditions hold, the edge is filtered according to its Bs. For edges with Bs ≤ 3, the edge pixels and up to 1 pixel on either side of the edge might be filtered. For $Bs = 4$, the edge pixels and up to two interior pixels on either side might be filtered. Thus, the filter smooths out artefacts while keeping the original image content sharp. Compared to non-filtered video, the deblocking filter reduces the bit rate by $5 - 10\%$ while keeping the same quality.

2.5 Entropy coding

Before the quantized prediction residuals and associated parameters such as prediction mode or motion vectors are written to the output bitstream, they are further compressed with entropy coding. The prior steps we introduced earlier worked in either the spatial- or frequency domain of the input frame, while entropy coding takes advantage of patterns in the resulting bitstream. Instead of using natural binary coding to write the end result of the prior encoding steps, we use our knowledge of the data to assign shorter codewords for frequent data. For instance, the quantization step in section 2.3 does not reach its compression potential unless we represent the runs of zeroes more densely than just a zero word for each occurrence. H.264 uses context-adaptive entropy coding, by which means that the assignment of codewords adapts with the content. Depending on recently coded macroblocks, the encoder will choose the code tables that it estimates will yield the shortest codes for the current block.

H.264 supports two entropy coding methods, either Context Adaptive Variable Length Coding (CAVLC) or Context-based Adaptive Binary Arithmetic Coding (CABAC). While

CAVLC is available in all profiles, the latter is only available in the Main profile or higher. The choice of encoding is signaled via the `entropy_coding_mode` flag [2].

2.5.1 CAVLC

When the `entropy_coding_mode` is clear, CAVLC is used to code the residual blocks. After a blocks has been reordered in a zigzag pattern, it is coded as shown in figure 2.10:

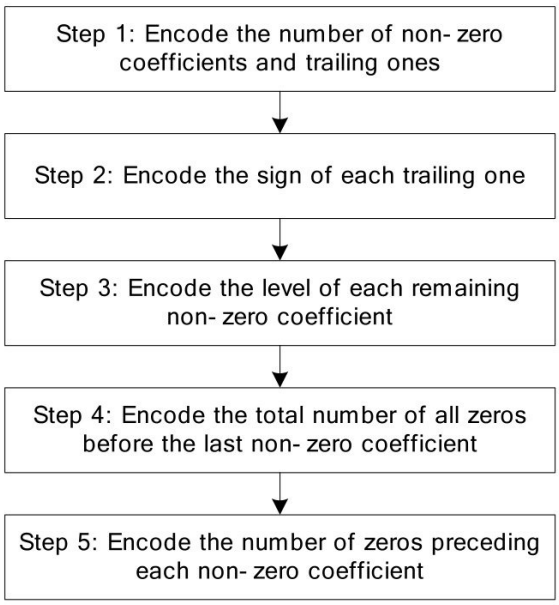

Figure 2.10: Flowchart of the CAVLC coding process [3].

First, a variable `coeff_token` is coded and written to the bitstream. `coeff_token` holds two values; the number of nonzero coefficients and how many of these are ± 1. As `coeff_token` only stores up to 3 trailing ones, any further ones are stored along with the other nonzero coefficients. On the other hand, if there are fewer than 3 trailing ones, we know that the last nonzero coefficient can not be ± 1. CAVLC takes advantage of this by storing the first coefficient decremented by ± 1.

The standard defines four lookup tables for coding `coeff_token`, and the table in current use depend on the number of nonzero coefficients stored in the neighboring blocks above and to the left of the current block. If both the upper and left block has

been coded, the table chosen is the mean value, or the same table is uses if only one is available. If the block is the first to be encoded, it defaults to the first table.

The tables are constructed so that the first table most efficiently codes low values, while the second and third tables code increasingly larger values. The fourth table uses a 6bit fixed length and is used for values not available in the other tables. Thus, the choice of lookup table for `coeff_token` is one of the properties that makes the coding context adaptive.

After coding `coeff_token`, the sign of each ± 1 is coded as a 1bit `trailing_one_sign_flag` following the convention from two's complements negative numbers with the bit set for minus, and cleared for plus. Note that the signs are given in *reverse* order.

Following the special case ones, the rest of the nonzero coefficients are coded, again in *reverse* order. Each one is coded as a two-touple, `level_prefix` and `level_suffix`. The level is divided into two code words to more efficiently code the variety of coefficient values by adapting the size of `level_suffix`. `level_suffix` uses between 0 and 6 bits, and the number of bits used by the currently coded coefficient depends on the magnitude of prior coded coefficients. The number of bits used by `level_suffix` is another property that makes CAVLC context adaptive.

Subsequent to coding all the nonzero coefficients, the total number of zeroes between the start of the block and the highest nonzero coefficient is coded in `total_zeroes`.

Finally, each run of zeroes before a nonzero coefficient is coded in a `run_before` code. As with `level_suffix` and `trailing_one_sign_flag`, this is done in *reverse* order. However, this is not done for the first coefficient, as we already know the total numbers of zeroes through `total_zeroes`.

2.5.2 Exponential Golomb codes

While CAVLC is used for residual block data, another approach is used for other information such as headers, macroblock type, motion vector difference etc, called Expo-

nential Golomb (Exp-Golomb) codes. In contrast to the tailored approach, Exp-Golomb codes is a general coding scheme that follows a general pattern. Each codeword consists of N leading zeroes separated from M bits of information with a 1, except for the first codeword that is coded as simply one. The second codeword is `010`, the third, `111`, the fourth `00100` and so forth. Hence, the shorter the code number, the shorter the codeword will be.

For a given code number n, the corresponding Exp Golomb code can easily be calculated as shown in listing 2.1 (which prints the code as text for readability).

```
import math

def ExpGolomb_code(code_num):

    if code_num == 0: return '1'

    M = int(math.floor(math.log(code_num+1, 2)))
    INFO = bin(code_num +1 - 2**M)[2:] # skip 0b

    pad = M - len(INFO) # number of bits to pad the INFO field

    return M*'0' + '1' + pad * '0' + INFO
```

Listing 2.1: ExpGolomb code example.

Each code number is decoded by counting until reading a 1, and then read the same amount of bits as the `INFO` field. The code number is then found by calculating

$$\texttt{code\_num} = 2^M + \texttt{INFO} - 1. \tag{2.7}$$

As Exp-Golomb coding is a general coding scheme, its efficiency lies in the lookup tables matching code words with actual data. To achieve the best compression, the shorter code words must be used for the most frequent data. As Exp-Golomb coding is used for a range of parameters, four different types of mappings are used:

- `me`, mapped symbols, are coding tables defined in the standard. They are used for, among others, inter macroblock types, and are specifically crafted to map the most frequent values to the shortest code numbers.
- `ue`, unsigned direct mapping, where the code number is directly mapped to the value. This is used for, inter alia, the reference frame index used in inter-prediction.

- `te`, truncated direct mapping, is a version of `ue` where short codewords are truncated. If the value to be coded cannot be greater than 1, it is coded as a single bit `b`, where `b = !code_num` [24].

- `se`, signed mapping, is a mapping interleaving positive and negative numbers. A positive number p is mapped as $2|p| - 1$, while a negative or zero number n is mapped as $2|n|$.

2.5.3 CABAC

When the `entropy_coding_mode` flag is set, CABAC is used. It uses arithmetic coding for both residual data and parameters, and achieves higher compression ratio at the cost of higher computational costs. In test sequences for typical broadcast scenarios, CABAC has shown a mean improvement of 9% - 14% increased compression rate [25].

The process of encoding with CABAC is threefold:

1. First, the syntax element to be encoded must go through *binarization*, by which means that a non-binary value must be converted to a binary sequence, as the arithmetic coder only works with binary values. The resulting binary sequence is referred to as a *bin string*, and each binary digit in the string as a *bin*. If the value to be coded already has a binary representation, this step can naturally be omitted.

   The binarization step represent the first part of the encoding process, as the binarization scheme employed by CABAC assigns shorter codes to the most frequent values. It uses four basic types, unary codes, truncated unary codes, k'th order Exp-Golomb codes and fixed length codes, and combine these types to form binarization schemes depending on the parameter to be coded. In addition, there are five specifically crafted binary trees constructed for the binarization of macroblock and sub-block types. The binarization step also makes the arithmetic coder significantly less complex, as each value is either 0 or 1.

2. After binarization, the next step is *context modeling*, in which a probability distri-

bution is assigned one or more bins for each bin string. The selection of context model depends on the type of syntax element to be encoded, the current slice type, and statistics from recently coded values. The choice of context model for the current bin is influenced by neighboring values above and to the left. CABAC uses nearly 400 context models for all the syntax elements to be coded. CABAC also includes a simplified mode where this step is skipped for syntax values with a near-uniform probability distribution.

The probability states in the context models are initialized for each slice, based on the slice QP. As the amount of quantization has a direct impact on the occurrence of various syntax values. For each encoded bin, the probability estimates are updated to adapt to the data. As the context model gains more knowledge of the data, recent observations are given more impact as the frequency counts gets scaled down after a certain threshold.

3. Following the selection of context model, both the bin to be coded and its context model is passed on to the arithmetic coder, which encodes the bin according to the model. The frequency count of the corresponding bin in the context model is then updated. The model will then continue to adapt based on the encoded data, until a new slice is to be encoded and the models are reset.

2.6 Related work

Many of the features that makes H.264 efficient depend on exploiting the relationship and similarities between neighboring macroblocks. A prime example of this is the motion vector prediction illustrated in figure 2.6. To efficiently compress the motion vector E, it is stored as a residual based on the median of its neighbors A, B and C. In terms of data dependencies, this means that E cannot be predicted until A, B and C has been processed. While this allows the video to be compressed more efficient, it also complicates parallelism, as the interdependent macroblocks cannot be processed in parallel by default. However, research shows that various steps of the encoding process can be parallelized by novel algorithms or relaxing requirements while minimizing the asso-

ciated drawbacks.

Parallelizing of H.264 can be broadly categorized as either strategies to parallelize the overall encoding workflow, or parallelized solutions to specific parts of the encoding process. Of the overall strategies, there are two independent methods; slice-based and I-frame/GOP-based.

The slice-based approach takes advantage of slices (see section 2.2) to divide each frame into self-standing slices that can be processed in parallel. While slices add robustness by being independently decodable, the same independence reduce the efficiency of prediction. Measures by Yen-Kuang Chen et al. has shown that using 9 slices increases the bitrate by 15-20% to maintain the same picture quality [16], with supporting findings by A. Rodríguez et al. [26].

One advantage of the slice-based approach is that it supports real-time applications, as it does not add extra latency or dependence on buffered input. Examples of encoders using the slice-based approach are the `cuve264b` encoder we introduce in chapter 4, including its origin, the Streaming HD H.264 Encoder [6]. Slice-based parallelism was also the original threading model for the free software X264 encoder [56].

The GOP/frame based approach is to use the interval of I-frames to encode different GOPs in parallel [26]. By predefining each GOP, the video can be divided into working units handled by different threads. An important drawback with the GOP-based strategy is that it is unsuitable for real-time applications, as it depends on enough consecutive frames to form a working unit. It also adds latency, as each GOP in itself is encoded sequentially.

As both approaches work independent of each other, they can also be combined to form hierarchal parallelization. By further dividing each GOP into different slices per frame, each GOP can also be encoded in parallel [26].

In addition to the mentioned approaches, there also exists different strategies to parallelize certain parts of the encoding process. However, the algorithmic detail of such designs is out of scope of this thesis.

2.7 Summary

In this chapter, we introduced the H.264 video coding standard, and gave an overview of its main means of achieving effective video compression. We also laid out related work in parallelization of video encoding. In the next chapter, we will introduce GPUs in general, and the nVidia CUDA framework in particular.

Chapter 3

nVidia Graphic Processing Units and the Compute Unified Device Architecture

3.1 Introduction

In the previous chapter, we introduced the H.264 video encoding standard. In this chapter, we give an introduction to the massive parallel architecture of GPUs. We give a brief history on the evolution of GPUs, and a more thorough overview of current GPU architecture and the Compute Unified Device Architecture (CUDA). In later chapters, we will use CUDA-enabled GPUs to improve video encoding performance of a H.264 encoder introduced in the following chapter.

A GPU is a specialized hardware unit for rendering graphics on screen. It can typically be an integrated part of a motherboard chipset such as nVidia Ion [57], or as a discrete expansion card. In addition, modern processors such as AMD Fusion series [58] and Intel Sandy Bridge [59] Accelerated Processing Units (APUs) combine a CPU with a GPU on a single die, enabling more efficient communication between CPU and GPU [27].

While originally limited to rendering graphics, modern GPUs are essentially massively parallel processors. Designed to render 3D scenes onto a frame of 2D pixels, they enable the concurrent computation of large numbers of values. The first generations of GPUs had a fixed pipeline with limited programmability, but modern GPUs enable general purpose programming through C-like languages such as nVidia CUDA and OpenCL by the Khronos Group. Thus, the same thread model applied to pixel processing can be applied for solving problems not limited to graphics. This is known as general-purpose computing on graphics processing units (GPGPU),

GPUs, due to their special purpose design, have a different architecture than CPUs. CPUs spend much die space on control logic, such as branch prediction and out-of-order execution and large cache to maximize performance [4]. GPUs have much less control logic, freeing up more die space for arithmetic logic units (ALUs). This gives a GPU more calculation capacity, at the cost of programming complexity. To reach peak performance, the programmer must explicitly design the application for the target GPU.

As shown in figure 3.1, most die space on a GPU is used for ALUs, while a typical CPU uses the majority of die space for control and cache. Thus, for applicable workloads, GPUs can outperform CPUs with the same number of transistors [28].

The computational strength of a GPU lies in performing the same calculations over a large number of values. While originally limited to shaders performing transform and lightning of 3D graphics, the same processing power can be used for general purpose computations. For instance, rendering a 1080P frame involves about two millions of independent pixels.

3.2 History

The term GPU was defined by nVidia in 1999, when they released their Geforce 256 graphics adapter [29]. While Geforce 256 was not the first 3D graphics adapter, it was the first to support Transform & Lightning accelerated in hardware compatible with

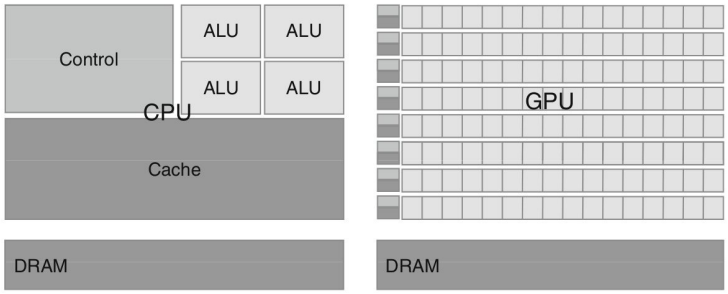

Figure 3.1: CPU transistor usage compared to GPU [4].

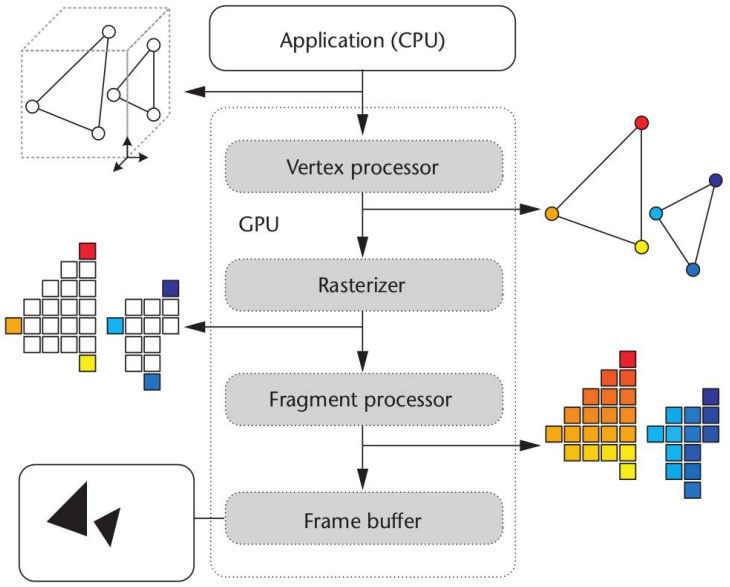

Figure 3.2: Diagram of a pre-DX10 GPU pipeline. [5] The Vertex processor was programmable from DX8, while the Fragment processor was programmable from DX9.

DirectX 7. However, due to the limits of a fixed function pipeline, the GPU was limited to providing visual effects.

The progression of GPUs has been largely driven by the gaming industry, and new generations have been released to implement requirements of the popular Microsoft DirectX API. Each following major version of the framework resulted in a generation of new GPUs.

DirectX 8 compatible GPUs, such as nVidia Geforce 3 and Ati Radeon 8500, added support for programmable vertex shaders. This allowed the programmer to control

how each vertex in the 3D scene was converted to discrete pixels in the output, such as position and texture coordinates.

The next generation of GPUs, compatible with *DirectX 9*, added support for programmable fragment shaders. This gave the programmer direct control over pixel lightning, shadows, translucency and highlighting. This per-pixel control could also be used for general computation, given that the problem could be mapped as a frame to be rendered.

While the pipeline was still primary focused for graphic tasks, the ability to write programs for the GPUs made it possible to utilize the GPU processing power for other tasks. However, to do so, applications needed to be specially adapted for the 3D pipeline in terms of vertices and fragments [28]. As shown in figure 3.2, the programmer could program the *Vertex processor* with DX8, and the *Fragment processor* with DX9.

Data input is stored as textures on the GPU, and computations in calculated by a shader program on streams of textures. The results are then stored in either texture memory or the framebuffer, depending on the number of passes necessary for the computation. While the fixed function pipeline limits the complexity of each pass, complex computations can be solved as multiple passes. For instance, Peercy et al. [30] have written a compiler that translates a higher level shading language into multiple OpenGL rendering passes.

The vertex and fragment processors where originally programmed in assembly language [29]. However, multiple higher-level languages, such as *C for graphics (Cg)* [31], *High Level Shading Language (HLSL)* [60] and *OpenGL Shading Language (GLSL)* [61] emerged, allowing programmers to program in C-like languages.

With *DirectX 10*, the traditional 3D pipeline was replaced with a more general processing architecture [32]. While still being driven by the demands from gaming and other applications of graphic accelerations, the architecture included many features important for GPGPU use. For instance, it added support for integer operations and more stricter IEEE-754 floating point support. This change in architecture made the GPU much more suitable as a general purpose stream processor, without the inherent limits

of the legacy graphics pipeline. CUDA, released by nVidia in 2007, made it possible to program its DirectX 10 compatible GPUs with extensions to the C language [33].

DirectX 11 introduced the Compute Shader, also known as Direct Compute. The Compute Shader expands the DirectX API to also encompass GPGPU programming similar to the CUDA stack. However, the Compute Shader is not dependant on a single hardware vendor, providing a more general abstraction. Although released as part of DirectX 11, Compute Shaders also run on DirectX 10 compatible hardware, within hardware limitations.

Since the initial release of CUDA, both the software stack and the underlying GPU architecture has been through both major and minor revisions, with the current compute capability (see section 3.3.2) at 2.1. Improvements include 32bit atomic integer instructions, 32bit floating point instructions, 64 atomic integer instructions, double precision floating point instructions and improved synchronization and memory fencing instructions.

While writing this thesis, nVidia released its latest generation GPU architecture, Kepler [62]. However, we will focus on the previous revision, Fermi [39].

3.3 Hardware overview

The Fermi GPU architecture consists of a number of processing cores clustered in Stream Multiprocessors (SMs), with a shared level 2 cache, device memory, host interface and a global scheduler which dispatches jobs to the SMs.

3.3.1 Stream Multiprocessors

Each Fermi-class SM, as shown in figure 3.3, contains 32 separate CUDA cores with both integer ALU and floating point units [39]. It also includes registers, instruction cache and local memory shared by all the local cores. Each SM also contains 16 load and store units, allowing one half-warp of threads access to DRAM in parallel.

In addition to the 32 CUDA cores, each SM also contains four Special Function Units (SFUs), responsible for calculating transcendental instructions such as square root and (co)sine. For such functions, CUDA provides two types of operations: Regular functions such as single precision `sinf(x)`, and functions prefixed with double underscore, such as `__sinf(x)`. The latter ones maps directly to SFU instructions on the device, but with limited accuracy. The regular `sinf(x)` is implemented on top of `__sinf(x)`, providing better accuracy[1]. However, the additional instructions lead to a more expensive computation. This is especially the case for trigonometric functions on operands with large magnitude, as the argument reduction code might need to store intermediate results in local memory. Direct usage of the SFU intrinsics can be forced at compile-time through the `-use_fast_math` flag.

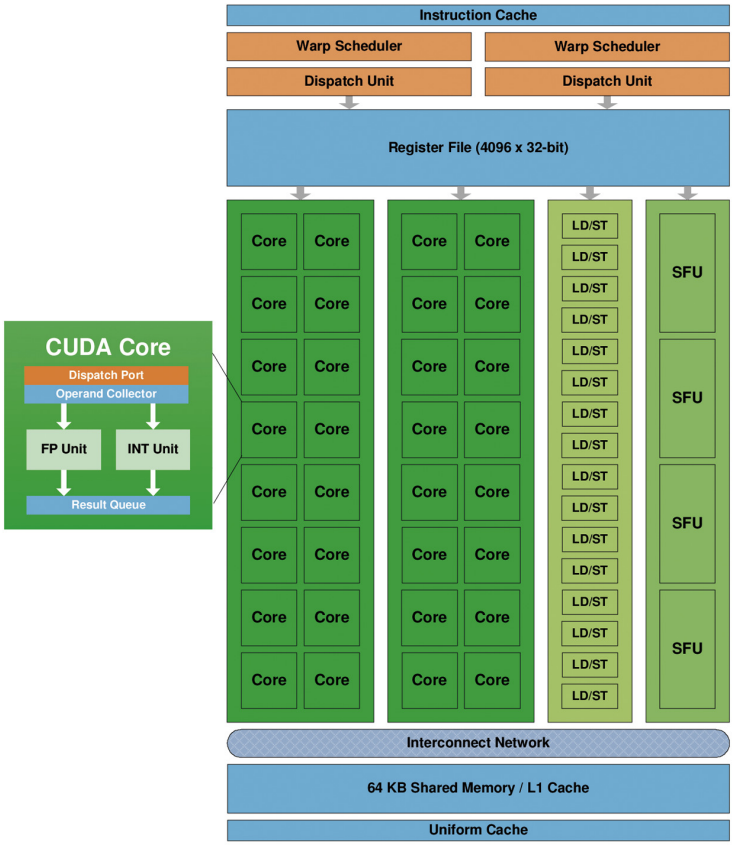

Figure 3.3: Fermi Stream Multiprocessor overview. [39]

Scheduling of the CUDA cores is handled by a dual warp scheduler and instruction

[1] `sinf(x)` has a maximum error of 2 units of least precision. `__sinf(x)` on the other hand, has an error of $2^{-}21.41$ for x in $[-\pi, \pi]$ with no bounds given for other values of x.

dispatching unit. This allow two groups of threads, called warps, to be executed concurrently. Thus, half of each of the two warps may run on 16 CUDA cores, run 16 load/store operations or use the 4 SFUs at any given time. Each of the threads of each such half-warp run the same instruction, and branching is not supported. If branching occurs, each of the branches must be evaluated for all the running threads, reducing performance.

Since the launch of Fermi, nVidia has released an updated SM design supporting 48 CUDA cores and 8 SFUs. It also includes an updated warp scheduler which issues two instructions at a time.

3.3.2 Compute capability

As the CUDA-compatible GPUs continue to evolve, newer features such as support for atomic instructions, double-precision floating point operations and support for multiple simultaneous kernel calls are added. To keep track of the improvements in the hardware, nVidia uses a version numbering system known as device compute capability. Using a major and a minor number, compute capability conveys the supported features of the GPU. See figure 3.4 for features added since 1.0. At compile time, the programmer can specify the target compute capability instruction set through the compile flag `-arch=sm_Mm`, where `M` and `m` are the major and minor version number. For instance, `-arch=sm_13` compiles for the 1.3 compute capability, which is the first to support double precision floating point operations. If the flag is not specified, the code will be assembled in the default instruction set. For instance, double precision floating point operations mentioned above will be truncated to single precision unless the target compute capability is specified. Similarly, the compute capability for compiled binaries can be specified with the `-code=sm_Mm` flag. The important distinction between target assembly instruction set and binary will be explained in section 3.3.5. The compute capability of a GPU can be queried with the `cudaGetDeviceProperties()` function.

<table>
<tr><th></th><th colspan="5">Compute Capability</th></tr>
<tr><th>Feature Support
(Unlisted features are supported for all compute capabilities)</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory</td><td rowspan="2">No</td><td colspan="4" rowspan="2">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory</td><td colspan="2" rowspan="4">No</td><td colspan="3" rowspan="4">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory</td></tr>
<tr><td>Warp vote functions</td></tr>
<tr><td>Double-precision floating-point numbers</td><td colspan="3">No</td><td colspan="2">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory</td><td colspan="4" rowspan="7">No</td><td rowspan="7">Yes</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory</td></tr>
<tr><td>__ballot()</td></tr>
<tr><td>__threadfence_system()</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
__syncthreads_or()</td></tr>
<tr><td>Surface functions</td></tr>
<tr><td>3D grid of thread blocks</td></tr>
</table>

Figure 3.4: Feature support by compute capability. [40] and slightly modified.

3.3.3 Memory hierarchy

As noted in the section above, each SM contains register space and shared memory for its cluster of CUDA cores. In addition to this, the GPU has larger memory resources available, but not integrated in the GPU. This results in a memory hierarchy where memory usage can have a crucial impact on performance. See figure 3.5 for a diagram of the memory hierarchy of a Fermi-class GPU.

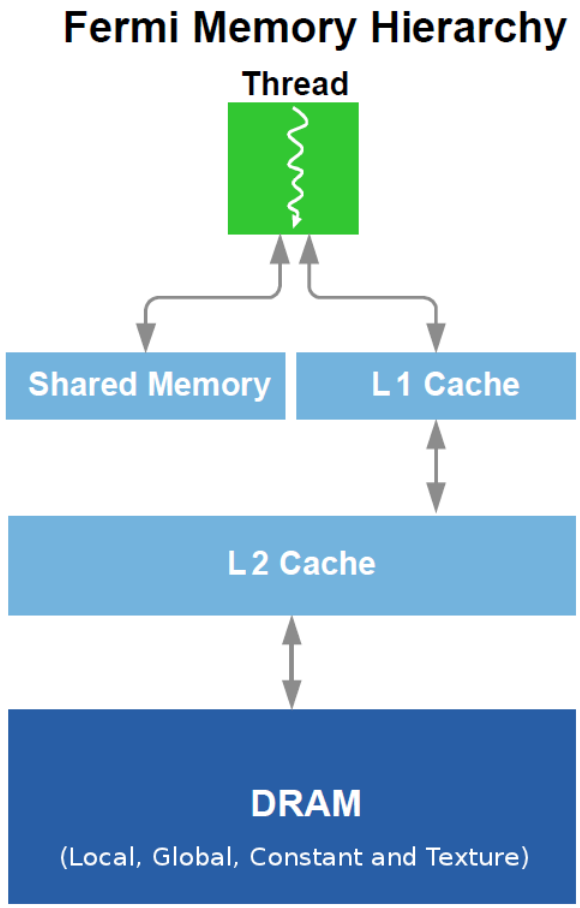

Figure 3.5: Fermi memory hierarchy: Local, global, constant and texture memory all reside in DRAM. [39], slightly modified.

On the first level, not shown in the diagram, we have the registers used by the CUDA cores. They have an access time of one clock cycle, but limited in size. On a Fermi-generation SM, the 32 cores share a total of 32768 four-word registers. If these gets exhausted, due to a large number of variables or large structures or arrays, local memory is used instead. Local memory is private to each thread, but resides in global memory which will be described below. On a Fermi-class GPU, 512KB is reserved as local memory per thread.

The second level of memory is the shared memory, also residing on-chip within each SM. Shared memory also has access time of one clock cycle, but accessible to all the running threads. This give all the cores of each SM a fast pool of memory for sharing data, reducing the need of off-chip memory for data reuse. However, the throughput of

shared memory is dependent on how the accessed data is laid out. The shared memory is uniformly divided into modules, called banks. Prior to Fermi, shared memory was divided into 16 banks, which increased to 32 with Fermi and compute capability 2.0.

Each memory bank can be accessed independently, so n memory accesses across n banks can be performed in parallel. On the other hand, if a single bank gets more than one memory access, they must be serialized. This is known as a bank conflict. For a n-way bank conflict, the conflicting memory accesses are divided into n independent non-overlapping accesses, with a respective reduction in access time.

Each Fermi-class SM contain 64KB of memory used for both shared memory and a L1 cache. Of the 64KB, 48KB can be used for shared memory with additional 16KB used as L1 cache, or vice versa. Shared memory can either be statically allocated in a kernel, or dynamically allocated at kernel execution from the host.

Off chip, the GPU has access to significantly larger amounts of memory, albeit with orders of magnitude slower access times. This memory is referred to as global memory, and is both available to the load/store units on each SM as well as the host. Global memory can be accessed with 32-, 64- or 128-byte transactions, given that the accessed address is aligned to the transaction size. When threads access global memory, the request of each thread will be combined into larger transactions if possible, known as memory coalescing.

The efficiency of coalescence depends on the access patterns and the compute capability of the GPU at hand. For the first generation CUDA devices with compute capability 1.0 and 1.1, the requirements are strict: Each thread needed to access 32, 64 or 128 bits from the same memory segment in sequence. For instance, a half-warp reading a `float` array indexed by a thread's id would be coalesced. Newer compute capabilities have loosened the requirements for memory coalesced as the load/store units have improved. For example, compute capability 1.2 removed the requirements for sequential access and supported coalesced access of 8- and 16-bit words. If the half-warp in the previous example read an array of type `short` or `char` instead, this would be coalesced with compute capability 1.2 or above, but serialized in 1.0 or 1.1.

Even with memory coalescing, global memory access incur significantly more latency than the registers and shared memory local to a SM. Given a sufficient number of threads, the scheduler will try to mask away the latency by scheduling other warps on the CUDA cores while the memory access is carried out. Global memory may be allocated statically in a kernel, or dynamically via CUDA-specific versions of `malloc()` etc, i.e. `cudaMalloc()`.

The GPU also has two additional memory spaces that are read-only from device code, constant and texture memory. They can only be written to from the host. The advantage of these memory spaces is that they are cached on each SM. Depending on application access patterns leading to cache hits, these memory spaces can greatly reduce the access time. A prerequisite for using either constant or texture memory is that the intended data fit in the limited space. Constant memory is limited to a total of 64KB, with 8KB cache per SM. Texture memory is limited to between 6-8KB cache per SM, depending on GPU. When the accessed data is not residing in cache, constant and texture memory have the same access time as regular global memory. The major difference between constant and texture memory is that CUDA exposes a subset of the texturing hardware of the GPU through functions working on texture memory.

As briefly noted above, Fermi introduced a L1 cache in combination with shared memory. While kernels must be explicitly programmed to take advantage of shared memory, the L1 cache caches both local and global memory accesses transparent to the programmer. The caching of local memory mitigates the increased latency incurred due to to exhausted register space.

Fermi also introduces a second level cache, shared between all the SMs on the device. It provides 768KB of unified cache and handles all load, store or texture requests. As the cache is shared, it can be used to efficiently share data between threads across different blocks.

In addition to the different memory regions on the SMs and off chip memory, kernels executing on the device might also access host memory. A prerequisite is that the memory region has been allocated as page-locked, often referred to as pinned memory. Such memory cannot be swapped out to secondary storage and is therefore a scarce

resource. Forgetting to free such memory can easily trash a system. Something I can confirm from personal experience. After allocating a region of pinned memory, it can be mapped into the device memory space with the `cudaHostGetDevicePointer()` function.

3.3.4 Programming model

Programming of the GPU follows the stream programming paradigm; the GPU code is implemented as kernels that gets executed over the data. A kernel is written similarly as a regular sequential function, without any special vector instructions. It is then executed in one instance per thread by the CUDA schedulers. This is referred to as Single instruction, multiple threads (SIMT) model, as all the threads spawned from a single kernel call will issue the same instructions. The only differences between the threads are the special variables `blockIdx` and `threadIdx`. They identify the current thread, and get set at kernel invocation time. In addition, `gridDim` and `blockDim` will contain the maximum dimensions for the thread hierarchy. In the source code, a GPU kernel is denoted by the `__global__` identifier, for instance `__global__ void vec_add(float **a, float **b, float **result)`. CUDA also supports device functions called from a kernel, identified with `__device__`.

Grid, blocks and threads

CUDA uses a two-tiered threading model that maps directly to the architecture of the GPU. Threads are bundled into groups, which are organized in a grid. The global scheduler on the GPU then distributes the groups of threads to the available SMs. All the threads inside each group executes on the same SM, and individual threads are managed by the local scheduler.

The programmer is free to choose how the two tiers are organized, i.e. within the hardware limits and the compute capability of the GPU. The thread groups and grid may be organized as either a one-dimensional row, a two-dimensional grid or a three-dimensional cube. This is done with the `dim3` integer vector types, which can contain

up to three dimensions.

Prior to a kernel call, the programmer must specify the number and distribution of threads per block and blocks per grid. For instance, `dim3 grid( 256 );` will create a variable `grid` with $256 * 1 * 1$ blocks, and `dim3 threads(32, 32, 4)` will create a variable `threads` with $32 * 32 * 32$ threads. Note that for a single dimension, passing a regular `int` will suffice.

A kernel call is then issued on the form as shown in listing 3.1:

```
myKernel<<<grid, threads, shared_memory, stream>>>
    (function, parametres);
```

Listing 3.1: CUDA kernel call

Except from the parameters enclosed in `<<< >>>`, the kernel call looks similar to a regular C-style function call. However, the call executes a kernel in parallel on a massive number of threads. The `grid` and `threads` specify the amount and distribution of the grid and threads per block as described above. `shared_memory` allows the programmer to dynamically allocate shared memory at invocation time. (This is not possible from inside the kernel itself.) `stream` controls which CUDA stream in which the kernel will be scheduled. We will elaborate on CUDA streams in chapter 7 Only `grid` and `threads` are mandatory, so if neither `shared_memory` nor `stream` are used, they need not be specified.

See figure 3.6 for an example with a 2D grid containing 2D thread groups. The outer grid contains 6 thread groups, which each contains 12 threads. When executed, the global scheduler on the GPU will assign the blocks to the available SMs. As the number off threads is less than the number of available cores per SM, each core will not be utilized. Likewise, 6 thread groups will not utilize all the SMs on a modern GPU. This figure originated with the now outdated version 2 of [40], when each SM only contained 8 cores. It shows the nature of the massive parallelism of the GPU; to fully utilize the computation power, the algorithms used must scale to a large number of threads.

As shown in figure 3.6, each thread within each block gets assigned a unique combination of `dim3 blockIdx` and `threadIdx`[2]. `gridDim` will be $(3,3,1)$ and `blockDim`

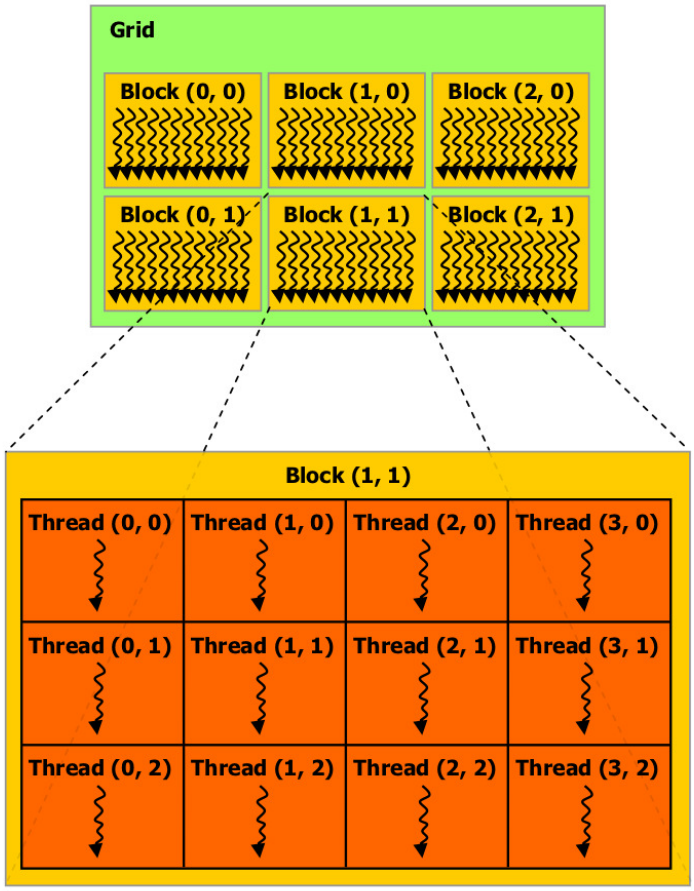

Figure 3.6: Example execution grid [40].

$(3, 2, 1)$ These variables are then used to index the data set, effectively distributing the data among the threads. For instance, if the threads in the figure worked on a 2D array with one element for each thread, the thread-specific datum would be read from global memory as shown in listing 3.2.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;

*mydata = input[i][j];
```

Listing 3.2: Thread array lookup.

[2] Although not shown in the figure, each `blockIdx` and `threadIdx` will have an implicit z-member of 0, with a corresponding z-member in `blockDim` and `threadDim` of 1.

Occupancy and efficient GPU utilization

The division between thread in each group and group in each grid has a direct impact on resource distribution at runtime. As each SM has a finite number of register space and shared memory, it is important to find the optimum division for the workload at hand. If the register space is exhausted, off-chip memory will be used instead, incurring a massive increase in access time. On the other hand, reducing the threads per group also reduces the sharing of shared memory, while enabling more data to be stored per thread. The GPU utilization achieved by a kernel call is referred to as the *occupancy*, and defined as the ratio between the number of resident warps and the maximum number of resident warps supported by the GPU [40].

A poor configuration of thread groups and grid may leave the CUDA cores in each SM blocked on memory accesses while leaving computing resource idle. Even though the numbers of concurrent threads are limited to 16 on each SM, interleaving of warps can keep the CUDA cores busy while other threads wait for memory accesses. To aid the programmers in achieving high occupancy, nVidia has written an occupancy calculator to find the optimum parameters for their kernels.

Flow control

As noted in section 3.3, the CUDA cores does not support branching. However, branching is possible on the outer grid, as each thread in a warp will take the same branch. For workloads with certain edge cases, mapping the edge cases as separate thread groups makes it possible to implement on CUDA without divergent branching. For instance, in our contribution to [34], we implemented a Motion JPEG encoder where the frame edge-cases where handled as separate groups. This enabled us to have special cases at the rightmost and lowest parts of the frame where pixel extrapolation was necessary.

In addition to being an abstraction of the GPU hardware, the grid and thread block hierarchy also gives the programmer some degree of execution control. While the scheduling of blocks and threads on SMs are outside of the programmers control, barriers and fences can be used to synchronize the execution within a thread group as well

as completion of memory accesses within expanding scopes:

- `void __syncthreads()` synchronizes the threads in a block.
- `int __syncthreads_count(int predicate)` synchronizes the threads in a block and evaluates the `predicate` for all threads. The return value is the count of threads where the result was non-zero.
- `int __syncthreads_and(int predicate)` synchronizes and evaluates `predicate`, but returns non-zero only if the result was non-zero for all the threads.
- `int __syncthreads_or(int predicate)` works similar to the `and` function above, but the result of the threads is `OR`-ed together instead of `AND`. It returns non-zero if at least one result was non-zero.
- `int __threadfence_block()` blocks until all accesses of global and shared memory made by the current thread are visible to the other threads in the block.
- `__threadfence()` additionally blocks until global memory accesses are visible to all the threads on the device.
- `__threadfence_system()` expands the fence even further, by blocking until accesses to pinned host memory is visible from the host.

For instance, the `__syncthreads()` barrier is crucial to efficient usage of shared memory. If all the threads in a block work on the same data, such as with the calculation of a DCT coefficient, all the threads can be assigned to fetch one value each from global memory as shown in listing 3.3. The subsequent `__syncthreads()` ensures that all the threads have loaded their assigned value into shared memory before the processing begins.

```
__global__ void syncthreads_exaple(float *input)
{
    __shared__ float values[SIZE];

    /* Read my part of the dataset into shared memory */
    values[threadIdx.x] = input[blockIdx.x*blockDim.x + threadIdx.x];

    __syncthreads();

    /* From this point on, we can assume that
       values[] has been populated */
```

```
}
```

Listing 3.3: `__syncthreads()` example.

3.3.5 Software stack

The CUDA software stack is built around three components; the compiler, runtime and driver.

At compile-time, the CUDA language extensions are used to determine which parts of the code gets compiled for the host with a standard C compiler such as `gcc`, and the kernel code that gets compiled for the GPU. This is handled by the `nvcc` compiler driver. Prior to host code compilation, `nvcc` replaces the CUDA-specific kernel call syntax with corresponding runtime function calls for loading and executing each kernel.

The device code is then either compiled into binary form, known as a `cubin` object, or assembled into the CUDA instruction set, called `PTX`. Any `PTX` instruction code loaded by the driver will be compiled just-in-time (JiT) prior to execution. The cubin-code is then cached, minimizing the time penalty of compilation for further application usage.

The choice between compiled or assembled device code depends on the necessary compatibility, as binary compatibility is only guaranteed for one future minor compute capability. For instance, a binary compiled for a GPU with capability `1.1` is guaranteed to run on `1.2`, but not `1.3` or `1.0`.

`PTX` instructions, on the other hand, can always be compiled for GPUs with equal or higher compute capability. Hence, the assembled code will be future-proof, while incurring the cost of JiT-compilation (and possible exposure of any trade secrets in the assembly code if applicable).

In addition to the compiler, CUDA consists of a runtime library and a driver API. Both are available to the application, as shown in figure 3.7. The runtime library provides convenient functions for most tasks such as memory allocation and copying, lowering the level of entry. These wrapper functions then call the CUDA driver to perform

the operations. The driver API gives the programmer additional control over the application by providing lower-level control over the GPU, such as CUDA contexts, the equivalent of host processes. (We also observe that the runtime does not distinguish between host processes and threads in this regard.) However, more control also means that the programmer must explicitly take care of initialization etc. that is done implicit by the runtime. The programmer can leverage both to achieve the best of both worlds. For instance, if the runtime has already been initialized, the programmer can access the current context by calling the `cuCtxGetCurrent()` function. Vice versa, the runtime will make use of a context initialized from the driver API.

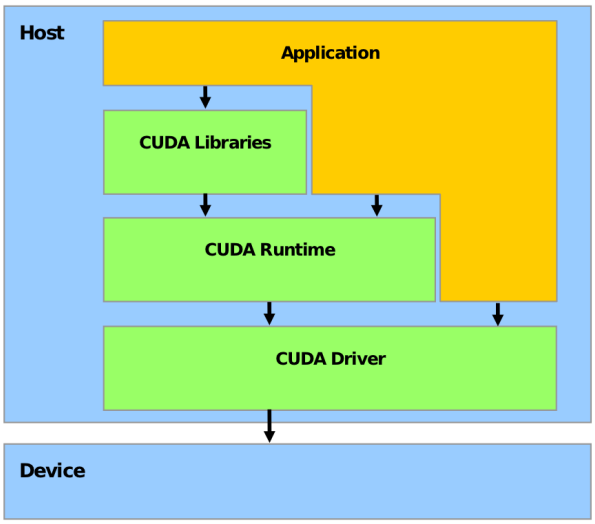

Figure 3.7: An application built on top of the CUDA stack [40].

3.4 Vector addition; a trivial example

To wrap up our chapter on GPUs and CUDA, lets have a look at a trivial example of a complete CUDA program, namely vector addition. A.1 shows an example implementation with explanatory comments. Error handling has been left out for brevity.

Compiling and running the program gives the following output:

```
$ ./cuda_example
0xDEAD0000+0xBEEF=0xDEADBEEF
```

3.5 Summary

In this chapter, we gave an overview of GPUs in general, with a focus on the CUDA framework developed by nVidia. We explained key components of a CUDA compatible GPU and the CUDA programming framework, and introduced the fermi GPU architecture. In the next chapter, we will introduce the software basis of our work, as well as the hardware testbed used to perform our experiments.

Chapter 4

Software basis and testbed

4.1 Introduction

In this chapter, we will introduce the software basis for our experiments, briefly explain how they are carried out, and list the specifications of our test machine. As a foundation for our experiments, we use the `cuve264b` encoder, a research project by Mei Wen et al. from the National University of Defense Technology, China. `cuve264b` is a port of their Streaming HD H.264 Encoder [6] to the CUDA architecture. The stream design of the original encoder makes it a suitable application for the CUDA SIMT model we elaborated is section 3.3.4.

4.1.1 Design

Slices

`cuve264b` uses slices to help parallelize the encoding process. By dividing each frame into multiple slices, the encoder can encode each slice independently, and hence in parallel. However, the number of slices to use is not configurable. Instead, the encoder depends on exactly *34* slices. Specifying any other value will result in corrupted output.

While our snapshot of the encoder only supports 1080P, support for 720P has been added at a later stage. To correctly encode 720P, the number of slices must be set to *45*. The choice of slices is clearly the result of a calculation, but the basis is unfortunately not documented in the source code.

Intra Prediction

To increase parallelization of the encoding process, `cuve264b` deviates from the reference encoder by relaxing the dependencies between neighboring macroblocks. When performing intra-slice prediction, the different prediction modes are evaluated to find the mode that yields the lowest residual block. Due to the quantization step, the macroblocks available for reference might be different from the blocks in the raw input slice. Thus, to achieve the best accuracy, the neighboring macroblocks must be encoded and subsequently decoded before the different prediction modes can be evaluated. However, this limits the number of macroblocks that can be predicted in parallel.

By using the original uncoded macroblocks instead, the prediction tests can be carried out without the prior encoding of neighboring blocks. While this will reduce the accuracy of the search, tests performed on high definition sequences has shown that the quality loss is negligible [35]. After the best mode has been found, the residual blocks are calculated using proper reconstructed macroblocks.

Motion Vector Prediction

As noted in section 2.2.3, H.264 not only uses motion vectors to exploit temporary redundancy, but also predicts motion vectors from its neighbors to take advantage of spatial redundancy between the motion vectors. Unfortunately, MVP constrains the number of motion vectors that can be coded in parallel, as the MVp for a macroblock depends on the blocks above, to the left and above as well as to the right, as shown in figure 2.6. By relaxing the MVp calculation as shown in figure 4.1, each row of macroblocks can be coded in parallel. Tests performed with the *Clair* test sequence

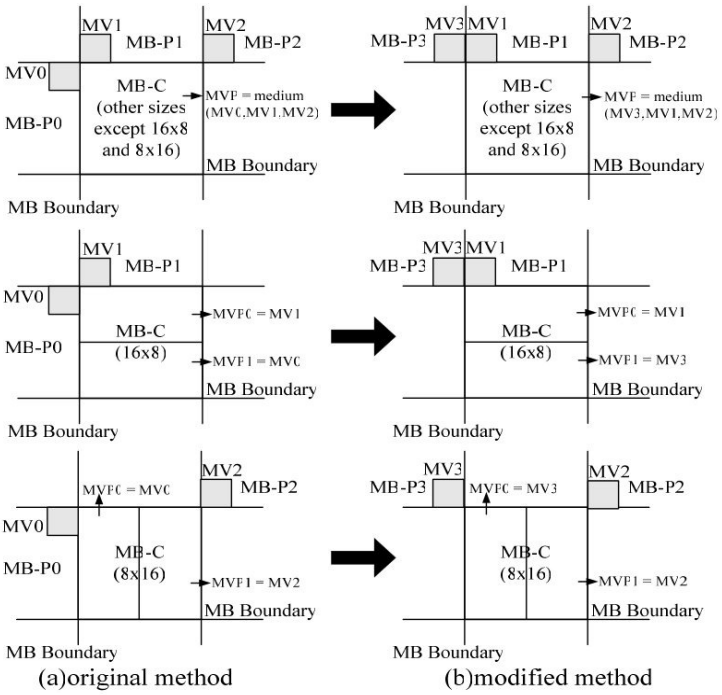

Figure 4.1: Modified Motion vector prediction in `cuve264b` [6].

showed that the Peak Signal to Noise Ratio decreased by at most 0.4DB compared to the X.264 video encoder [6].

CAVLC

To be able to parallelize CAVLC, the entropy coding has been split up into two separate steps, *encoding* and *linking*. The encoding step can be performed independently for each macroblock, and is performed on the GPU. The linking step on the other hand, is performed on the host. In linking, the independently processed macroblock data is processed to correctly place the information correctly in the bitstream. This includes alignment on byte boundaries, calculating each macroblocks position in the bitstream, and writing the information in the right position.

4.1.2 Work in progress

As the `cuve264b` encoder is a work in progress, not all parts of the encoding process is done by the GPU yet. For instance, the deblocking filter runs on the host. This leads to unnecessary copying of reference frames from the host to the device. Due to inter prediction, the encoder must keep at least one prior encoded frame available to reference. To ensure that the frame used for reference is identical to the frame read by the decoder, this frame must be encoded and deblocked. As the deblocking filter is done on the host, the encoded frame must be copied back to the device after it has been deblocked. We will explore this further in chapter 8.

4.2 Evaluation

Except for a small number of special cases, most of our proposed improvements are evaluated by measuring the encoding time and comparing the result. To perform the encoding tests, we use three different test sequences, *blue_sky*, *tractor* and *pedestrian_area* [63]. They all consist of 4:2:0 frames of 1080P video, but varies in length. They also have different content suitable to test various cases for motion estimation and prediction, such as camera panning, zooming and objects in various degrees of motion. However, as we only compare different runs of the same test sequences, we assume that this will have no impact on our results.

To measure integral parts of the encoding process, such as host function calls and device kernel calls, we wrap the code in`gettimeofday()` system calls. `gettimeofday()` was the preferred Linux timing source evaluated by Ståle B. Kristoffersen [36] as part of his work on the P2G framework [37].

Most of our results are presented as the reduction in encoding time, given in percent. However, for the re-ordering and helper kernels measured in chapter 7 and 8, we give them in direct ratio to better clarify the large differences.

All our tests have been carried out on the machine *Kennedy*, with its most significant

CPU	Intel Core i7 860 (2.80GHz)
CPU Cores	4 with Hyperthreading
Memory	8GB
GPU	GTX 480 (GF100)
Harddrive	Seagate Barracuda 7200rpm
Operating System	Ubuntu 10.04.4 LTS 32bit X86 (Linux 2.6.32-38-generic-pae)

Table 4.1: Hardware specifications for the machine Kennedy.

properties listed in table 4.1.

4.3 Summary

In this chapter, we have introduced the `cuve264b` encoder, the software basis for our work. We have also given an overview of the environment used to carry out or benchmarks and explained how they are carried out. In the following chapters, we will introduce and evaluate our proposed improvements to `cuve264b`.

Chapter 5

Readahead and Writeback

5.1 Introduction

As the computational throughput of GPUs increase, it is important for the host to provide new work on time. If the host is not ready to dispatch more work for the GPU when the current work is done, the GPU will idle. Optimizing the GPU code or upgrading the GPU will not give any performance gain, as the host has become the bottle neck.

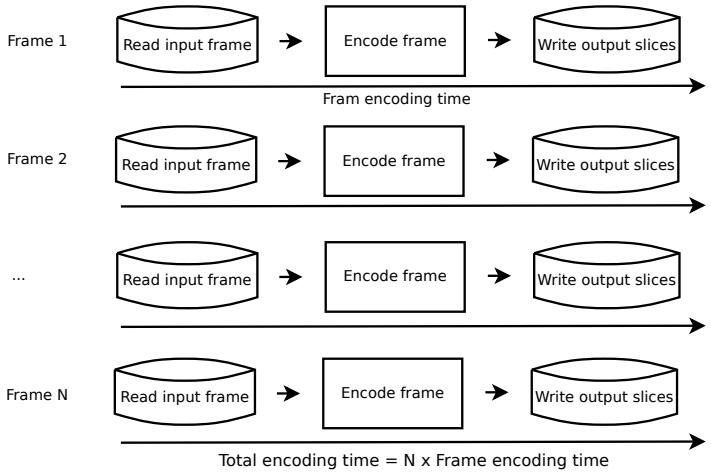

Figure 5.1: Serial encoding as currently done in the encoder.

To keep the GPU busy, the host must ensure that more work is ready, as soon as the GPU has completed its current operations. The next workload should already be resi-

dent in memory, and the host process should not block on IO operations. In the event that the next workload must be read from secondary storage, the GPU will idle. This hurts performance, and can neither be resolved by optimizing the GPU code nor investing in a more powerful GPU. On the contrary, both options will only increase the idle time.

In terms of video encoding, a prime example of such potential pitfalls is reading raw frames from secondary storage, and, albeit to a lesser extent, the subsequent writing of completed slices. Performing such IO operations in sequence with the encoding imposes unnecessary idle time both for the host and the GPU. This is the case for the `cuve264b` encoder in its current state as shown in figure 5.1.

At the start of each frame, the GPU must wait for the host to read the raw frame from secondary storage. When the frame has been read, it is the host's time to idle while the GPU encodes the frame. The GPU is then yet again idling while the host writes the encoded slices to disk before reading in the next frame.

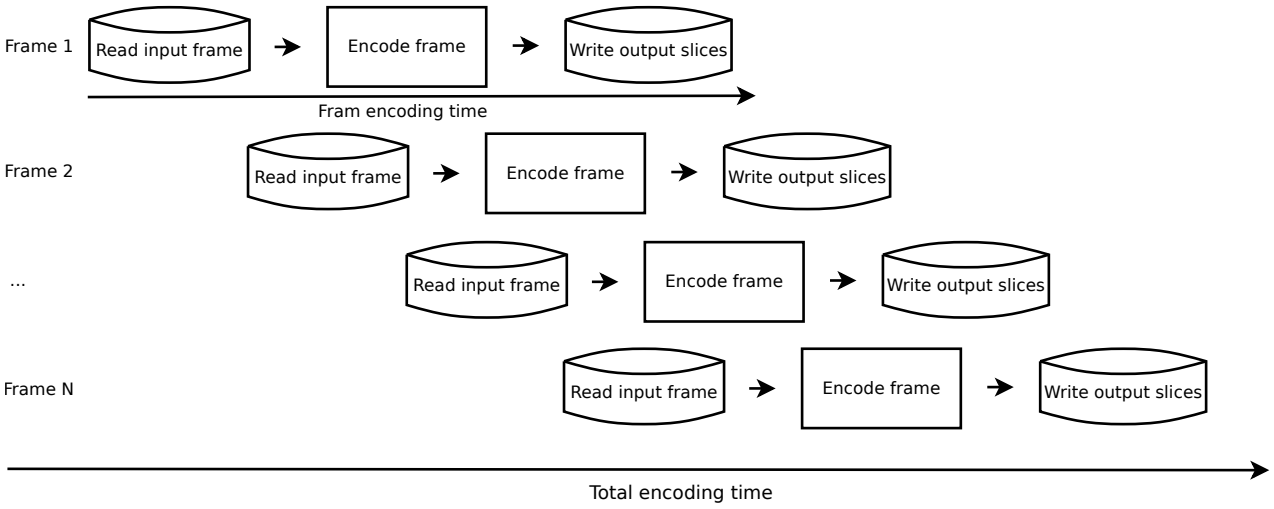

Figure 5.2: Threaded encoding as planned, where the separate threads will pipeline the input, encoding and output steps, reducing the elapsed time.

The idle time can be somewhat reduced by making the host start reading the next frame while the GPU encodes. However, if the GPU completes its operations while the host is blocking on IO access, it will still idle unnecessary. To reduce the idle time as much as possible, we must implement the IO operations asynchronously. This will allow us

to perform IO-operations in separate threads concurrently with the encoding. As show in figure 5.2, this will reduce the encoding time by masking away IO operations while the encoding happens.

5.2 Design

5.2.1 Rationale

Our longest test sequences, *tractor*, is 604 frames[1]of 1920+1080p YUV4:2:0 frames of about 3MB each. These are read from the local hard disk, which is a Seagate Barracuda 7200rpm SATA disk. It has an average latency of 4.16 ms and a best-case 160MB/sec read speed as reported by the manufacturer. Before frame N+1 is read from the disk, the already encoded slices of frame N is written to disk. Hence, every frame read incurs the cost of a disk seek. This gives us the total estimated read time of

$$604 * \left(4.16 * 10^{-4} + \frac{3}{160}\right) = 12 seconds \tag{5.1}$$

However, this does not take file-system metadata or other disk accesses into account. Neither does the manufacturer specify an average read speed, only the peak value.

To get more realistic data, we performed 10 continuous reads of the whole tractor sequence[2]. With care taken to drop any in-memory caching prior to each read, we got a median result of 16.07 seconds. For a single frame, the corresponding median was 0.087 seconds. Our goal is to read new frames into memory while the encoder works on the current frame. This allows us to mask away all reads, except the 0.087 seconds needed for the first frame. As the encoded slices take vastly less space than the raw frames (31MB with QP at 30), writeback will not have the same impact on performance. However, it should have a noticeable reduction in encoding time.

[1]While [63] lists tractor as containing 761 frames, our local copy only contains 604 frames which is within the limits of 32bit IO operations.

[2]As we read the whole sequence, we did not incur the expected seeks prior to each frame.

In addition to masking out IO latency, splitting the application into three threads allows us to divide the workflow into more general pre-processing, encoding and post-processing workloads. One advantage of such a division, is pre-processing of input data. This can be done while reading input frames from disk. For instance, the encoder pads incoming frames to ensure that all blocks are of equal size. While the original encoder did this as a separate step in the process, we do it inline while reading from file.

5.2.2 Implementation

To separate IO operations from the encoding of each frame, the encoder has been divided into three threads: The main encoder thread and a readahead and a writeback helper.

The encoder thread will be nearly identical as the non-threaded version, albeit IO operations will be removed. Instead of reading input frames from file, it will retrieve ready-buffered frames from the readahead helper. When a frame has been successfully encoded, the resulting slices will be sent to the writeback thread so it can continue encoding further frames without waiting on IO. In addition, the main thread must initialize the helper threads at startup and synchronize with the writer thread before exiting.

The readahead thread will read all the input frames from disk, and pass them on to the main thread. As noted in the previous subsection, the thread will also take care of padding the input frames as necessary. The writeback thread will wait for encoded slices from the main thread, and write them to disk as they become available.

Queue

Splitting up the encoder in three different threads, requires that we have an effective way of inter-thread communication. As both raw frames and finished slices are handled first-in-first-out, we base our implementation on a singly linked list with head

and tail. This gives us insert at the head and pop at the tail in $O(1)$ time, while keeping the implementation details simple. To keep the list thread-safe, we use mutexes and condition variables provided by the `pthreads` POSIX standard API.

An important aspect of our multi-threaded implementation is to prioritize the encoding thread at the expense of the helpers. To do this, we implement two sets of functions in the list: Single operations such as `insert()` which grabs the mutex for each invocation, as well as corresponding `_batch()` functions that assume that the mutex is taken. This allows the main encoder thread to lock the list for multiple operations, while the helper threads must release the lock for each access. For example, `cuve264b` uses multiple slices to perform the encoding. These slices are then written to the output container as multiple independent buffers. When the encoding of a frame is done, the main thread can then insert all the slices in the writeback queue without releasing the lock. The writeback thread, on the other hand, must acquire the lock for each slice removed from the list. Hence, the time spent by the main thread waiting for the list is minimized.

Our list can also support resource limits on the number of items in the list. We will use this feature to evaluate the optimum readahead window size for the encoder. For instance, it is not necessary useful to prefetch more raw frames than what is needed to stay ahead of the encoder. In real-time conditions such as live footage, it would not even be possible without delaying the stream. Using condition variables, we can wake up dormant threads when the size of the list gets below the limit.

For a complete list of the features of our list implementation, see listing 5.1.

```
1  #ifndef THREADLIST_H
2  #define THREADLIST_H
3
4  /* thread-safe single linked list used for io caching and write-back */
5
6  /* list element */
7  struct elem
8  {
9          unsigned char *data; /* bitstream data */
10         struct elem *next; /* next element */
11 };
12 typedef struct elem elem;
13
14 /* list metadata, pointers and locking */
15 struct list
16 {
17         elem *head;
18         elem *tail;
19         pthread_mutex_t lock;
```

```
20          pthread_cond_t nonempty, empty, nonfull;
21          int size, limit;
22  };
23  typedef struct list list;
24
25  /* constants */
26  #define LIST_NOLIMIT 0
27
28  /* list functions */
29
30  /* initializes list */
31  void list_init(list *l, int limit);
32
33  /* Lock the list so multiple operations may be run in batch */
34  void list_batch_lock(list *l);
35
36  /* Unlock the list after a batch of operations */
37  void list_batch_unlock(list *l);
38
39  /* Inserts *data at the end of list *l */
40  void insert(list *l, unsigned char *data);
41
42  /* Identical to insert(), but it presumes that the
43   * lock has been taken. Used for batch operations
44   */
45  void insert_batch(list *l, unsigned char *data);
46
47  /* Pops the list and returns the popped data.
48   * blocking if the list is empty*/
49  unsigned char *pop(list *l);
50
51  /* Pops the list if any items are available.
52   * If not, NULL is returned */
53  unsigned char *try_pop(list *l);
54
55  /* Identical to pop(), but it assumes the lock is taken */
56  unsigned char *pop_batch(list *l);
57
58  /* Blocks the calling thread until the list is empty */
59  void block_until_empty(list *l);
60
61  /* Get the length of the list,
62   * assuming the lock has been taken */
63  int list_batch_len(list *l);
64
65  /* Get the length of the list */
66  int list_len(list *l);
67
68  /* Get the enforced limit on the list,
69   * assuming the lock has been taken */
70  int list_batch_limit_get(list *l);
71
72  /* Get the enforced limit on the list */
73  int list_limit_get(list *l);
74
75  #endif /* THREADLIST_H */
```

Listing 5.1: Queue header

5.2.3 Hypothesis

Our hypothesis is that the Readahead and Writeback threads will have a positive effect on the encoder runtime. First because we decouple IO operations from the main execution thread, and second because the Readahead thread may double-buffer incoming

frames for immediate use. The encoder only has to wait for the initial buffering.

As the encoder reads significantly more data than it writes, the writeback implementation will have less impact. However, we do expect it to have a measurable impact.

5.3 Evaluation

We benchmarked our readahead and writeback implementations by measuring the time spent encoding three test sequences, *blue_sky*, *tractor* and *pedestrian_area* [63]. Figure 5.3 shows the mean improvement in encoding time after 10 runs. As can be seen from the figure, the results from our readahead implementation is in line with our hypothesis. The mean improvement in encoding time is 22%. This is consistent with our expectations of masking away the disk accesses as detailed in section 5.2.1. Our earlier measurement for disk accesses showed that reading *tractor* sequentially took a median time of 16.07. The mean encoding time is 91.8 seconds without and 72.6 seconds with readahead enabled. This is an improvement of 19.2 seconds, slightly better than our estimate. This is probably due to the interleaved writes and corresponding increase in seek time.

On the other hand, our writeback implementation did not have a significant impact on the encoding time. In the best case of *blue_sky*, the improvement was a meager 0.55%. In contrast with readahead, the impact of writeback seems to decrease with input size. It would be interesting to see if this trend continued with larger test sequences. Unfortunately, the encoder does only support 32bit file operations, which severely limits the input file size. We spent a short time trying to port the encoder to 64bit file operations, but we ran out of time due to underlying bugs. Hence, we will leave it for future work, as part of a thorough port to `X86_64`.

We can however investigate how the writeback implementation will perform if we increase the output size. With more data being written, the benefits of a separate writeback thread might become more apparent. In our tests, we performed the encodings using a `QP` of 30 and a single `GOP` (see section 2.2.2 and section 2.3 respectively). This

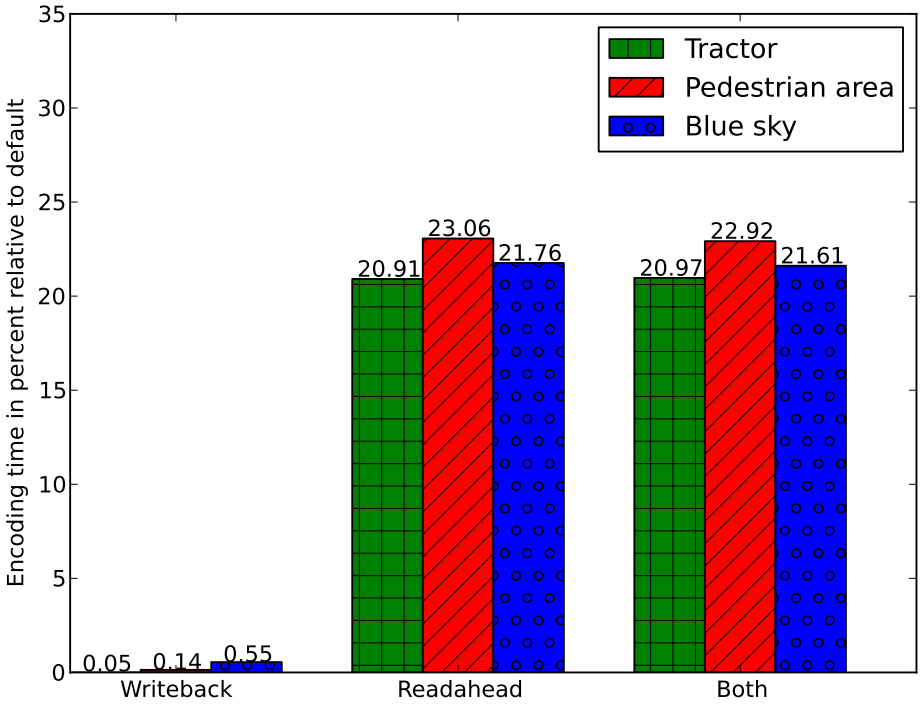

Figure 5.3: Improvement in encoding time due to readahead and writeback.

was the defaults in the encoder. As both the amount of quantization and frequency of I-frames have a major impact on the level of compression, our writeback implementation might perform better on larger writes. To further investigate this, we tested the implementation while varying the QP and GOP levels to increase output size.

Figure 5.4 shows writeback performance with decreasing `GOP` from 60 to 1. Similarly, figure 5.5 shows `QP` from 30 to 0. Decreasing `GOP` and `QP` to increase the output data did not show any pattern in writeback behaviour. This indicates that our implementations perform poorly independent of the amount of data being written.

While we did not expect the writeback implementation to improve performance to the same degree as readahead, we still expected the results to be more prominent than the <1% achieved. To understand the results, we decided to study the path from the `write()`-call until the data resides on disk.

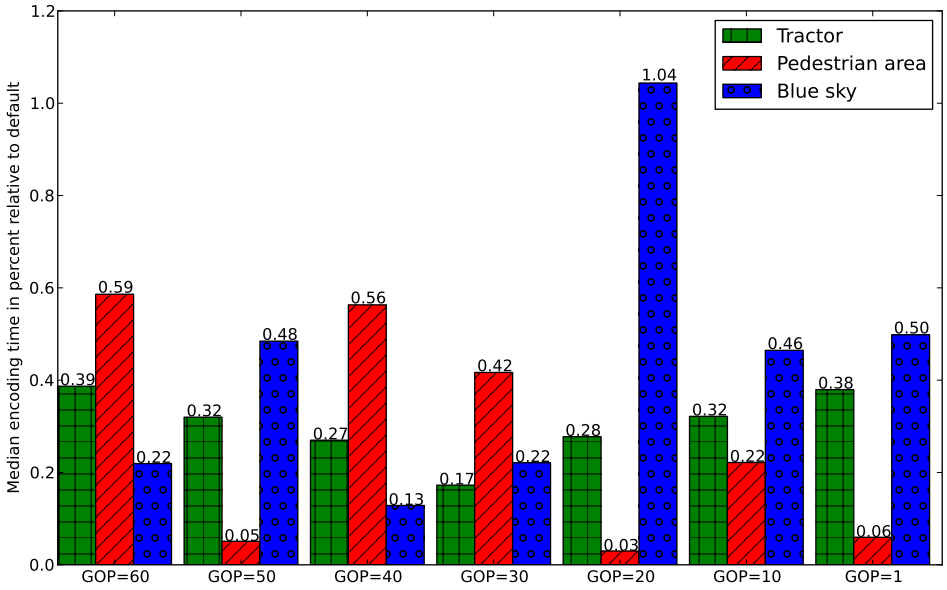

Figure 5.4: Writeback performance over various `GOP` levels.

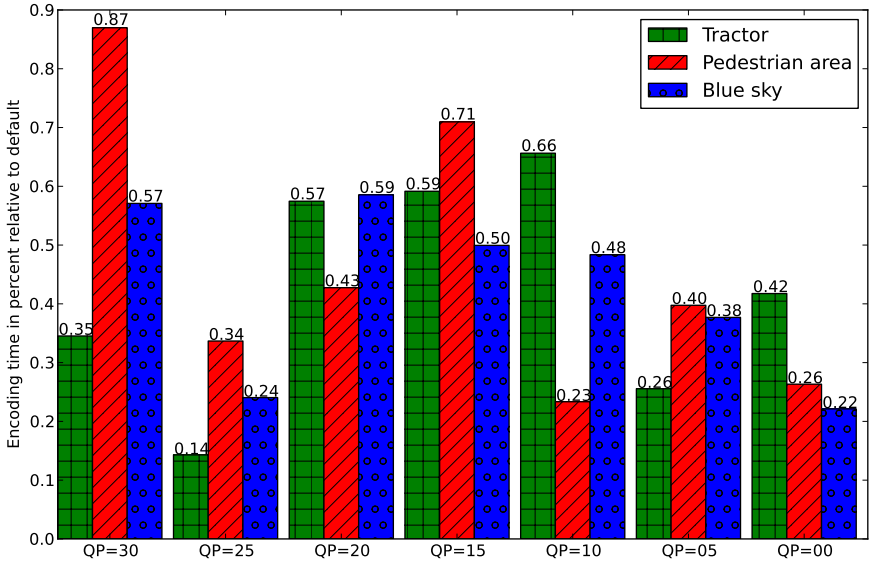

Figure 5.5: Writeback performance over various `QP` levels.

5.4 Write buffering

The rationale behind our writeback implementation was based on the notion that writing to disk in the main encoding thread would incur delays. Each output slice of each

frame would be written separately, which would result in many syscalls to write to disk. Each of these calls would block the encoding thread until finished and the device would idle.

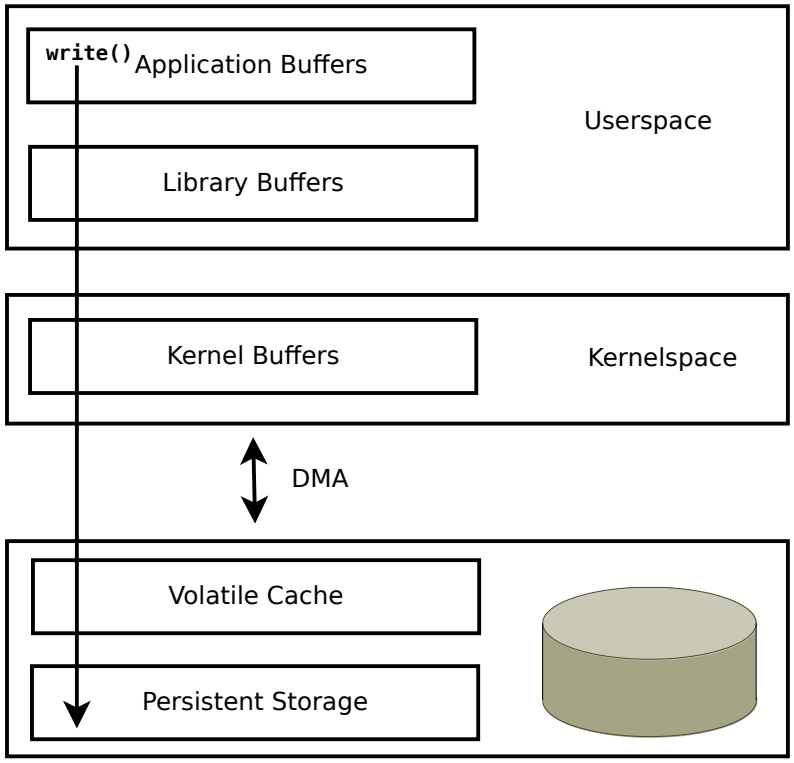

Figure 5.6: IO buffering from application memory to storage device. Based on [41].

This assertion turned out to be wrong. As shown in figure 5.6, both the C library and the kernel contains buffers to optimize disk accesses [41]. This happens implicitly outside the scope of the program. However, the programmer can explicitly flush the C library and kernel buffers with `fflush()` and `fsync()` respectively. This allows the programmer to control the integrity of the data being written to disk.

In our case of video encoding, we can just re-encode any lost video after an outage. On the other hand, we can use `fflush()` and `fsync()` to evaluate how our writeback implementation would have worked if our assumptions held, and which of the C library and kernel buffers masks away most of the write costs. To simulate the absence of either C library or kernel buffering, we performed a new set of writeback tests with an explicit `fflush()` or `fsync()` after each write.

As shown in figure 5.7, our implementation had a significant impact on encoding time.

However, even with writeback, the encoding time is far slower without the buffering. Comparing the bars for `fflush()` and `fsync()`, it is clear that most of the buffering takes place in the C library.

Even though writeback worked better without buffering, it is clear that we based our implementation on wrong assumptions. The buffering in the C library and kernel mitigates the problem well enough without the added complexity of a separate writer thread.

Nevertheless, our implementation is not without merit. While an initial improvement of $< 1\%$ was less than we expected, it can be improved. Separation of the encoder into three threads allows us to distinguish pre- and post-processing from the main encoding process. For instance, the encoder currently performs parts of the entropy coding on the CPU. As this is done as part of the main thread, without dispatching any more work to the GPU, processing time is clearly wasted. However, the upstream project was revising the entropy coding implementation at the time of our discovery. Therefore, moving such post-processing work would be more fit for future work when the code has been overhauled.

5.5 Readahead Window

As noted in our queue implementation in section 5.2.2, we implemented resource limits. This allows us to investigate how our enhancements perform under limited windows sizes. Limiting the writeback queue would only force the encoder to idle, so we focus solely on readahead.

How fast the readahead thread can fill up its buffers depends on multiple factors. For instance how fast the encoder performs, and the throughput of the underlying storage medium. Without any limits, it might buffer more frames than necessary. Even with vast amount of memory, there is no point in buffering more often than needed. Then it is better if the thread can sleep until enough buffers have been freed up.

To evaluate how the readahead thread perform with various limits, we timed the en-

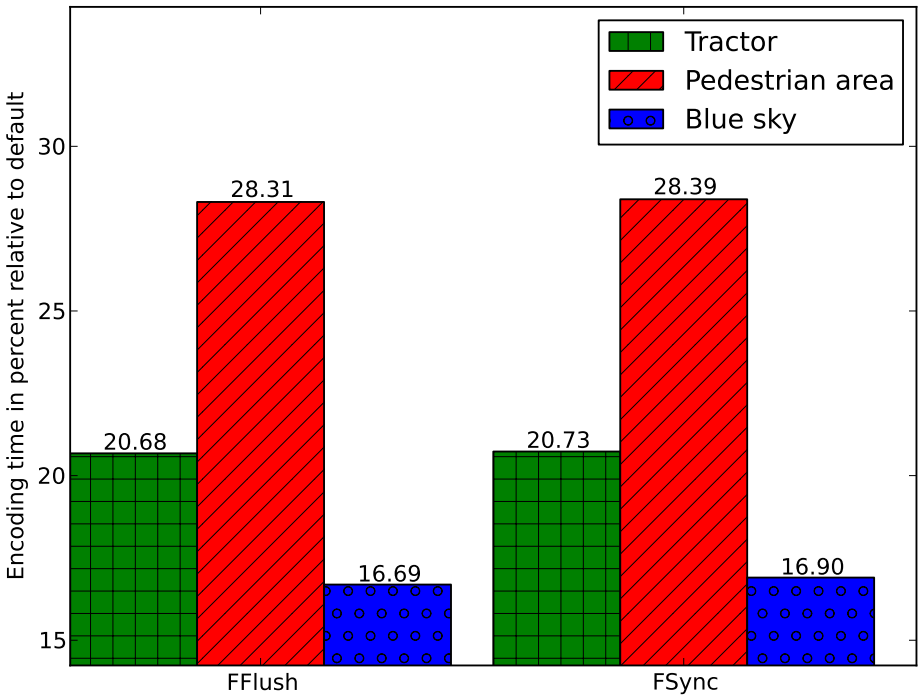

Figure 5.7: Writeback performance with buffer flushing.

coding process with a window size of one, to and three frames with an additional run without limits. We expect the implementation to excel as long as there is at least one cached frame available to the encoder.

As shown in figure 5.8, performance decreases reciprocally to window size. There is nothing to gain by pre-fetching more than one frame at a time, as it is enough for the encoder to consume without delay. Increasing the window beyond a single frame reduces the performance, so there is no gain from allocating more memory.

This allows us to be restrictive in our allocation of pinned memory as we experiment with CUDA Streams in chapter 7. As pinned memory is a scarce resource, allocating more memory than necessary may result in performance loss due to comparably slow allocation times.

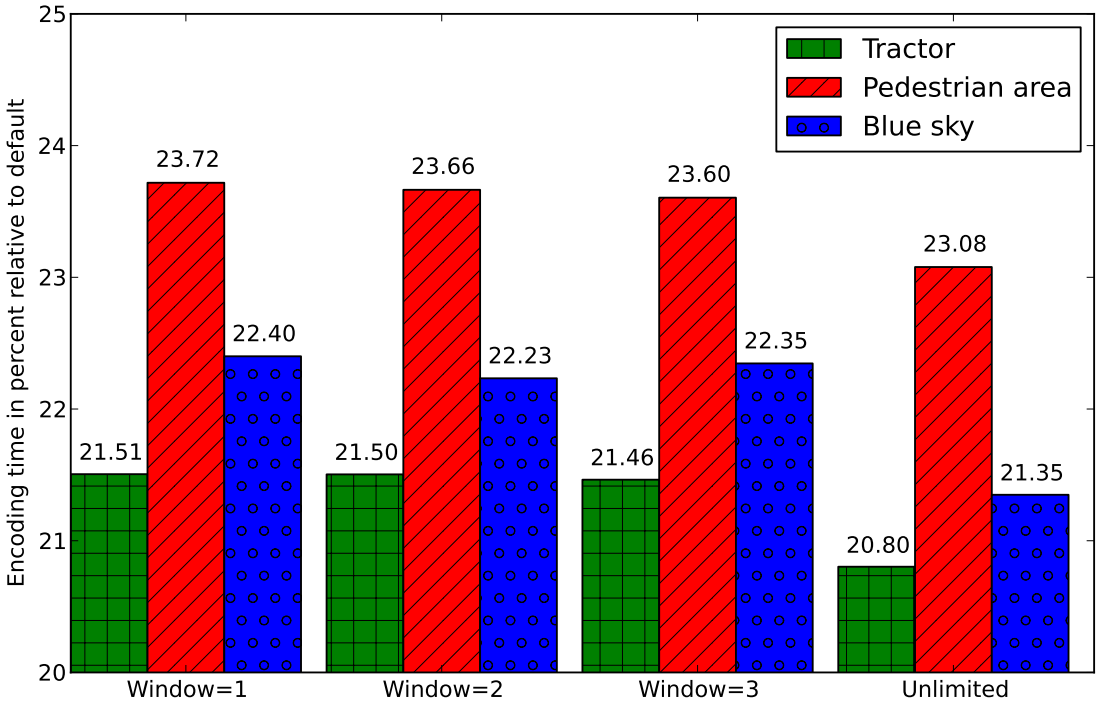

Figure 5.8: Readahead performance under different resource constrains.

5.6 Lessons learned

Our implementation shows the importance of designing efficient host code to take full advantage of GPU offloading. While optimizing device code is crucial to maximize performance, it is imperative that the host code is kept up to speed. By going back to the basics of masking away IO accesses through asynchronous operations, we have reduced the encoding time by 20%. This clearly shows that narrowly optimizing certain code paths might leave out low hanging fruit yielding significant improvements.

At the same time, we have shown how crucial it is to base our research on sound assumptions. The results of our tests shows that our rationale for implementing write-back was wrong. The poor performance of our writeback implementation led us to study why, giving us greater knowledge in how the C library and kernel handles write operations.

5.7 Summary

In this chapter we have looked at the start and end of the encoding process, namely reading and writing from secondary storage. We have implemented readahead of raw frames and writeback of the encoded frames to reduce the encoding time used for IO operations. Our benchmarks shows that particularly readahead have a strong impact on encoding time. This shows that CUDA programs can be optimized with minor changes to the device code.

Chapter 6

Memory

6.1 Introduction

As the computational power of GPUs increase, so does the amount of memory available. Similarly to host memory, memory allocated from the device global memory is reserved for the runtime of an application. Hence, we can use the global memory to keep as much state of the application on the GPU as needed.

However, the `cuve264b` encoder does not currently take advantage of this. Instead, state is only kept for the lifetime of each encoded frame. This leads to unnecessary allocations and memory operations, as the same state must be allocated, transferred and freed for each frame.

This approach has numerous drawbacks. By freeing up the memory after each frame, we cannot reuse any state left from the previous encoding. While the cost of re-allocating the freed memory is negligible, discarding the device state has numerous flaws without any gain.

One major flaw is the superfluous synchronization between the host and the device. As all memory will be freed, we must synchronize all metadata back to the host after completing a frame, before subsequently transferring the metadata back to the device before encoding the next frame. For each block in the frame to be encoded, metadata

such as block position, previous motion vector etc is synchronized.

However, if we keep the state on the GPU, we only need to synchronize the data that actually change. For this, we traced the updates to the state to determine which data must be transferred which way. Listing 6.1 shows the metadata stored for every block.

```
1  /* Motion Vector */
2  struct S_MV
3  {
4      int x;
5      int y;
6  };
7
8  /* Block Metadata */
9  struct S_BLK_MB_INFO
10 {
11         int SliceNum;
12     int Type;
13     int SubType;
14         int Pred_mode;
15     int IntraChromaMode;
16     int QP;
17     int CBP;
18     int TotalCoeffLuma;
19     int TotalCoeffChroma;
20     int RefFrameIdx;
21     int Loc;
22         int MinSAD;
23         struct S_MV MV;
24 };
```

Listing 6.1: Block Metadata.

6.2 Design

6.2.1 Rationale

Without any state kept on the device, the whole `S_BLK_MB_INFO` array needed to be transferred back and forth for every encoded frame. By analyzing the changes made, we found multiple sources for redundancy. One such source is constant information determined by the host at the start of the encoding process. For instance, the number of slices per frame and the resolution of each frame is the same throughout the encoding. Thus, the grouping of blocks into slices will stay constant, and the `int SliceNum` field will never be updated. Similarly, the location of a block, `int Loc`, is kept constant. This information only needs to be sent to the device once, instead of once per frame as done currently. Similarly, some state kept by the GPU are never read by the

host, such as `Pred_mode` and `TotalCoeffChroma`. Hence, we can differentiate the state into three groups: Constants shared once from the host, state private to the GPU, and GPU state shared with the host. Figure 6.1 shows the state shared between the host and the device, and in which direction. The constants need only be transferred once, while the shared GPU state is synchronized for every frame. `cuve264b` currently synchronizes all the state in both directions, while the arrows shows which updates are strictly necessary.

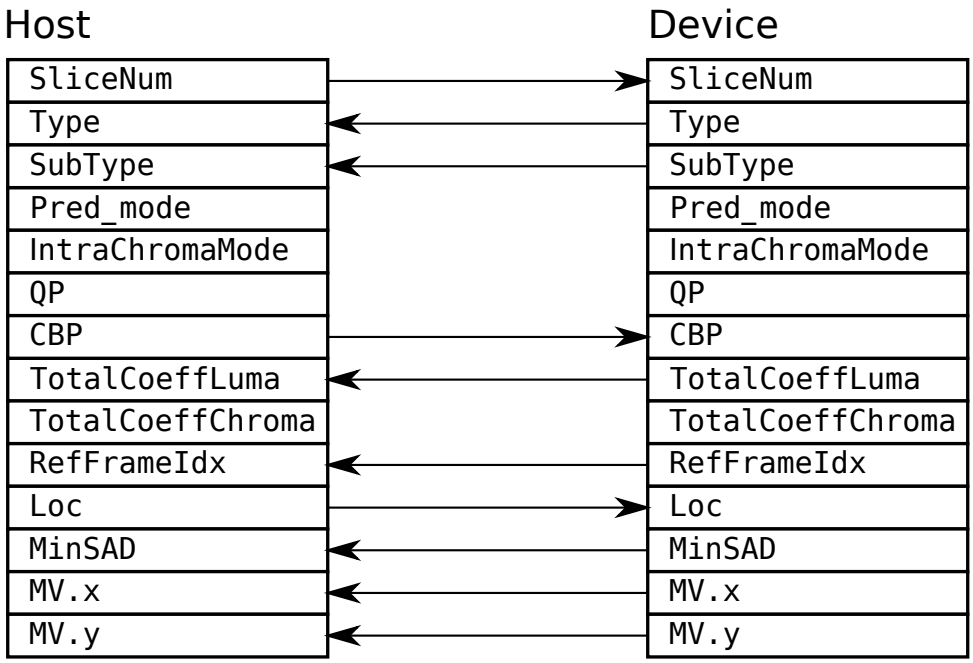

Figure 6.1: Encoder state shared between host and device.

Another source of redundancy is the size of the variables. While storing variables at 32bit boundaries yield better computation performance, it also increases the bandwidth needed to transfer the data. Our analysis shows that most of the fields can be stored using only one byte, reducing the bandwidth by 4. The only exceptions are `Loc` and `MinSAD`, which both require 2 bytes. However, due to the 32bit GPU alignment requirements [40], it is infeasible to reduce the size of the structure itself.

To reduce the data transfer size, we need an alternative representation of the data. As we know which data are shared between the host and device, we devised a simple marshalling scheme. Depending on the direction of the transfer, we simply write the shared state in a bytestream, using as few bytes as necessary per variable. This gives

us a transfer size of 8 bytes per block for state shared by the device, and 4 bytes per block for the initial state from the host.

Even though the transfer size is reduced, our scheme incurs additional costs, as the marshalling process itself delays the transfer. To evaluate the efficiency of our marshalling scheme, we wrote two different implementations: A proof of concept, and an implementation optimized for the GPU.

6.3 Implementation

6.3.1 Necessary groundwork

Before we could implement our marshalling, we needed to refactor the encoder to keep state throughout the encoding. Instead of allocating and releasing memory as part of the main `encode_cuda()` encoding function, we created separate functions for encoder initialization and destruction, `init_cuda_encoder()` and `free_cuda_encoder()`. This allowed us to keep memory allocation and freeing in separate functions, reducing memory allocations and cleaning up the code. Keeping the memory allocations over the lifetime of the application also allowed us to transfer constant data only once, and allocating them in constant memory. However, as such data amounts to very little data, the reduction in transfer time was negligible. Nonetheless, the refactoring was a necessary foundation for further improvements.

6.3.2 Marshalling implementation

To implement our simple marshalling scheme, we first wrote an implementation for the host side. Afterwards, we implemented the GPU part as a minimal rewrite. Without any regards for the GPU hardware, we simply rewrote the host function to use `blockIdx` and `threadIdx` instead of an outer loop. As shown in figure 6.2, each thread reads its data from a different part of the bytestream. Each adjacent thread will

read from global memory with a stride the size of a marshalled structure. As a result, none of the memory accesses will be adjacent, and none of the requests will be coalesced.

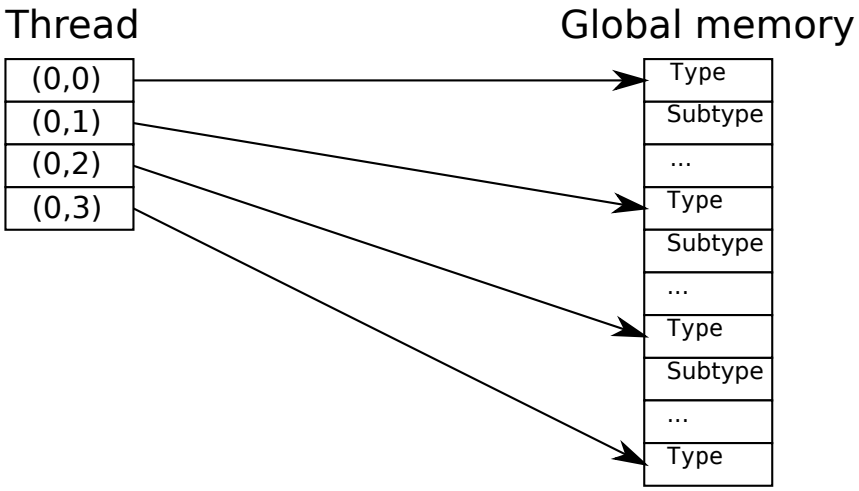

Figure 6.2: Device memory accesses for the direct port.

To achieve coalesced memory accesses, each thread must read from adjacent global memory. Therefore, we redesigned the order of the bytestream so that it is ordered by structure field before the structure items. Thus, the bytestream will first contain the `type` field for every structure, the `subtype` field of every structure, and so forth.

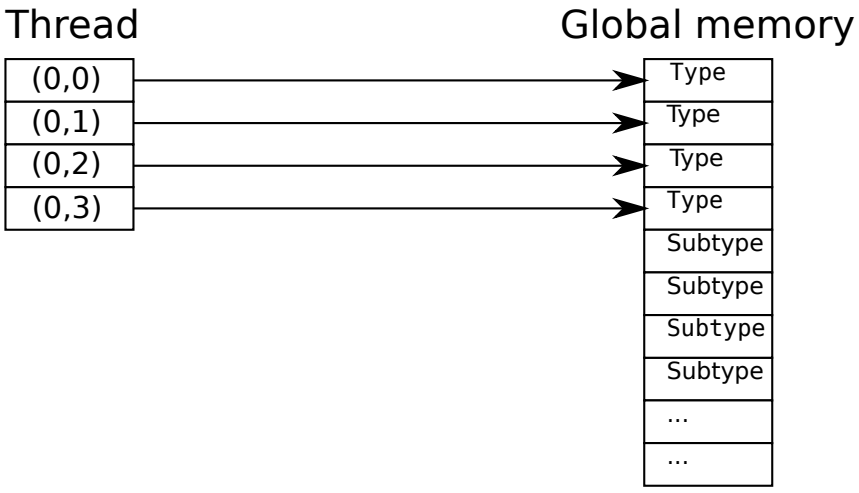

Figure 6.3: Device memory accesses for the optimized port.

As shown in figure 6.3, the new bytestream order meets the coalescence criteria explained in section 3.3.3. All the threads in each half-warp reads from the same seg-

ment, in sequence. While the first generation of CUDA-enabled GPUs did not support coalescing 8-byte operations, this was gained with compute capability 1.2 [40].

Given this, our new implementation should increase device performance. By optimizing memory accesses, the cost of the marshalling process should be reduced. However, the efficiency of our marshalling approach also depend on the host marshalling performance. While our second approach should increase device efficiency, it might have side effects for the host code. While the initial host code consisted of sequential reads and writes, this is not the case for the new variant.

6.4 Evaluation

To evaluate the performance of our two marshalling implementations, we measured the time spent encoding *blue_sky*, *tractor* and *pedestrian_area* [63].

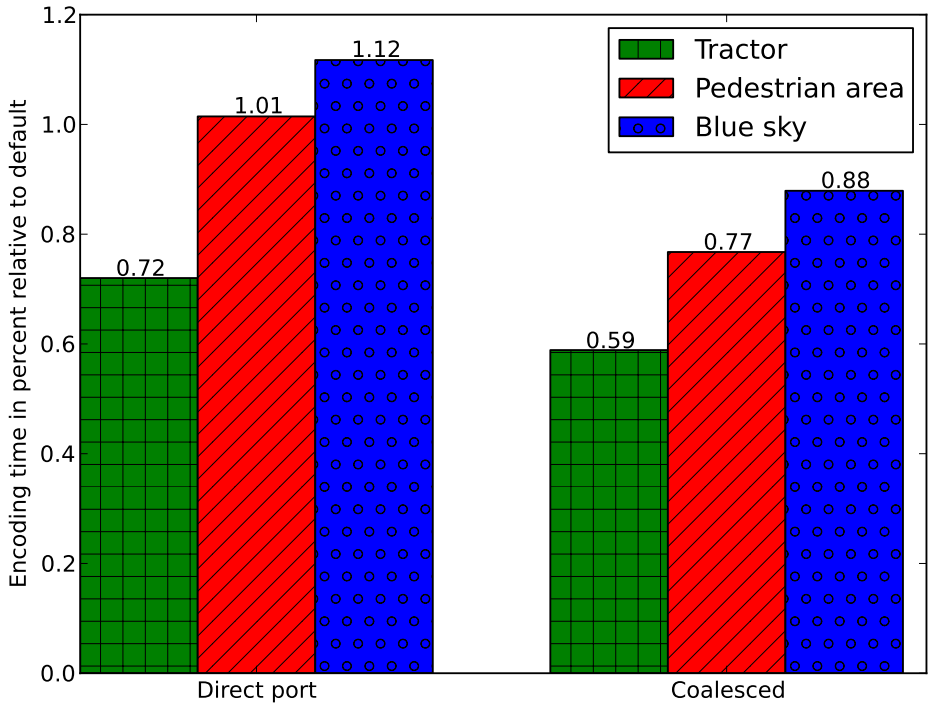

Figure 6.4: Performance of the two marshalling implementations.

Implementation	GPU time	L2 Write requests	Dram writes
Direct port	203.712	151044	228249
Coalesced	199.584	78768	33284
Difference	2.03%	47.85%	85.42%

Table 6.1: Selected profiling data from our marshalling implementations.

Figure 6.4 shows the improvement in percent on total encoding time for both our implementations. They both improve the encoding performance, as less time is spent on data transfer. As can be expected, the improvement increases linearly with the numbers of frames to be encoded. Somewhat unexpectedly, the unoptimized implementation outperform the device-optimized one. To be sure, we profiled the optimized version with the CUDA Visual Profiler [49] to be sure that the memory accesses improved. Table 6.1 shows selected profiling data from the bytestream writing kernel, `dev_pack_mbinfo()`, after encoding the *tractor* sequence. As the numbers shows, the rearrangement of the bytestream significantly reduces the number of write operations performed.

Unfortunately, the drawbacks for the host implementation are greater than the device improvements, decreasing total performance.
By timing the corresponding `unpack_mbinfo()` on the host, we measured an average of *0.831* seconds spent in the device-optimized function during the course of encoding *tractor*. The corresponding time for the unoptimized implementation was *0.739* , resulting in a 10.97% increase in runtime.

To take a closer look at the host code performance, we wrote a minimal test tool to explore the host code in isolation. Simulating the *tractor* sequence, it runs the `unpack_mbinfo()` host function 604 times with 8160 blocks. By profiling the test tool with `cachegrind`, part of the Valgrind instrumentation framework [64], we simulated the memory and cache behaviour.

While the number of cache misses was consistent, the device-optimized implementation resulted in an increase in both number of instructions and data accesses. As shown in table 6.2, the device-optimized implementation uses roughly 20% more memory ac-

Implementation	Instructions	Data read	Data write
Direct port	336,531,570	182,775,445	59,290,116
Coalesced	425,247,090	236,990,485	74,076,036
Difference	20.86%	22.88%	19.96%

Table 6.2: Instruction and data accesses of host marshalling implementations.

cesses than the direct port.

6.5 Lessons learned

The performance of our two marshalling implementations, show the importance of the relationship between the host and the GPU. While our coalesced implementation performs better on the GPU, the same access patterns results in multiple scattered accesses on the CPU, reducing performance. As solutions in the interface between host and device may have multiple solutions playing to the strengths of different architectures, it is important to evaluate them to make sure that the best solution has been found.

6.6 Summary

In this chapter, we took a closer look at the memory allocations on the GPU, and the encoder state shared between the host and the device. By extending the lifetime of memory allocations to last throughout the encoding process, we have enabled the encoder to keep state on the device. Prior to this, all the encoder state was synchronized between the host and the device for every encoded frame.

With state kept on the device, we analyzed the changes in metadata to determine which updates was necessary, and in which direction. Afterwards, we used the results to design a simple marshalling scheme by writing the updates with the shortest variable sizes needed.

To verify the efficiency of our design, we wrote to different implementations of the de-

sign, one with the starting point in the host code and written to be efficient on the host, and a second implementation optimized for the GPU to achieve coalesced memory accesses. While the optimized implementation greatly reduced the number of memory accessed on the GPU, the additional costs in memory accesses for the host outnumbered the improvement. Hence, the first implementation was the best solution to our problem.

Chapter 7

CUDA Streams

7.1 Introduction

In chapter 5, we reduced IO lag by prefetching raw frames from secondary storage. However, the transfer of a raw frame is not completed until it reaches device memory. Thus, our readahead thread is not a complete solution, as the GPU still has to wait for the frames to be copied from host.

As mentioned earlier, our readahead (and writeback) design uses multiple host threads to asynchronously perform IO operations. However, we cannot use host threads alone to perform concurrent operations on both host and device. To enable this, CUDA provides an asynchronous API, referred to as CUDA Streams.

To copy the frames directly to device memory, we must use CUDA Streams, allowing us to queue up multiple operations on the GPU in independent queues, referred to as Streams [40]. In essence, a CUDA Stream is a series of commands performed asynchronous to the host thread. By queueing work to be performed by the device in a CUDA stream, the host may perform other tasks in parallel with the GPU. While the first generation of CUDA-enabled GPUs did not support any form for overlap or concurrency, compute capability 1.1 added support for overlapping memory transfers with kernel execution. This made it possible to introduce prefetching by performing an asynchronous memory transfer in parallel with kernel execution. Finally, compute

capability 2.X added support for concurrent kernel executions. This enables us to not only prefetch raw frames, but also perform any necessary preprocessing on the device itself.

To enable asynchronous memory transfer, the involved memory regions on the host must be allocated as pinned memory. This allows the GPU to access the memory directly through DMA transfer, without involving the CPU, as the pinned pages is guaranteed to be resident in main memory.

7.2 Design

7.2.1 Implementation

Readahead thread adaption

We designed our frame buffering mechanism as an extension of our prior work implementing readahead. As the readahead thread already reads the raw frames from disk including memory allocation, we adapted it to use pinned memory instead of regular paged memory. The change itself only altered a couple of lines (`cudaHostAlloc()` instead of `malloc()` and `cudaFreeHost()` instead of `free()`, but affected the lifetime of the memory alloactions.

As raw frames are read by the readahead thread, the memory used for storing them gets assigned a different CUDA context than the main encoder thread. This means that any resources such as allocated memory is isolated between different host processes and threads. Fortunately, the API function for allocating pinned memory, `cudaHostAlloc()`, accepts a flag `cudaHostAllocPortable`. This allows us to share the memory allocated by the readahead thread with the main encoder.

While this allows us to share the memory, normal restrictions regarding the lifetime of allocated memory still applies. Hence, the lifetime of the allocated memory is limited to that of the readahead thread. In our original implementation of the readahead thread, we simply ran the thread in a for-loop until all the frames were read from

memory. This will not suffice, as the allocated memory will be automatically released on thread termination. To keep the thread alive throughout encoding the last frame, we added a condition variable `encoder_done` to signal that memory might be freed.

Memory reuse

Pinned memory is a limited resource, and such allocation requests often takes longer time to satisfy. Especially for the large continuous areas necessary to hold the raw frames, allocation can take substantially longer time than the unpinned equivalent.

To measure this, we wrote a trivial test utility in which we allocated and freed up the same amount of memory as required to read all frames from the *tractor* sequence. Timing the utility for paged and pinned allocations show that paged allocations took a negligible total of 0.0055 seconds. On the other hand, it took 2.746 seconds to satisfy the allocations of pinned memory, an increase of nearly 500 times.

To minimize the cost of allocations, we must recycle already allocated memory instead of free up used memory. To implemented this, we use an additional linked list to send used buffers back to the readahead thread. Before allocating more memory, we test to see if any recycled buffers are available.

Implementation

We implemented overlapped memory transfers by replacing the memory transfer code with a transfer scheduling function, `schedule_framestreaming()`. `schedule_framestreaming()` checks if any frames are available from the readahead thread, and as long as a new frame is available, it schedules a new transfer.

Similar to our previous readahead work, we also use the overlapping memory transfer to perform some preprocessing, further reducing the waiting time. Prior to encoding a P-frame, the raw frame must be available in both the default raster order and grouped by macroblock, known as block flattened order. The re-ordering was performed on the host, before both versions of the frame was transferred. This meant that the delay

caused by the file transfer was doubled, in addition to the wait incurred by the re-ordering function itself.

As part of the transition to overlapping memory transfers, we ported the host code to the GPU, and perform the reordering as part of the memory transfer. This should further reduce the delays prior to encoding each frame. Note that all the measurements shown in figure 7.1 perform the reordering on the GPU, including the measurements with overlapping disabled. Figure 7.2 shows the effect on device re-ordering in isolation.

7.2.2 Hypothesis

As we saw in chapter 5, buffering raw frames from disk reduced the encoding time by 20%. However, the performance gap between disk and main memory is much wider than between the 4x PCI Express 2.0 slot used by the GPU and global memory. While some increase in encoding time is to be expected, it is significantly lower than what was achieved by the readahead implementation. The GPU only need to wait for the first frame to be transferred, so the improvement should scale linearly with the number of frames in the encoded sequence. Another aspect to look into is the effect of pinned memory in and of itself, as the CUDA runtime accelerates accesses to such allocations [65].

7.3 Evaluation

To measure the improvements on the overall encoding time, we repeated the measurements of *blue_sky*, *tractor* and *pedestrian_area*.

As shown in figure 7.1, overlapping memory transfers decrease the encoding time even further than our disk readahead implementation. This exceeded our expectations. However, the results without reordering is closer to our expectations.

To further measure the improvement on memory transfer overlap in isolation, we mea-

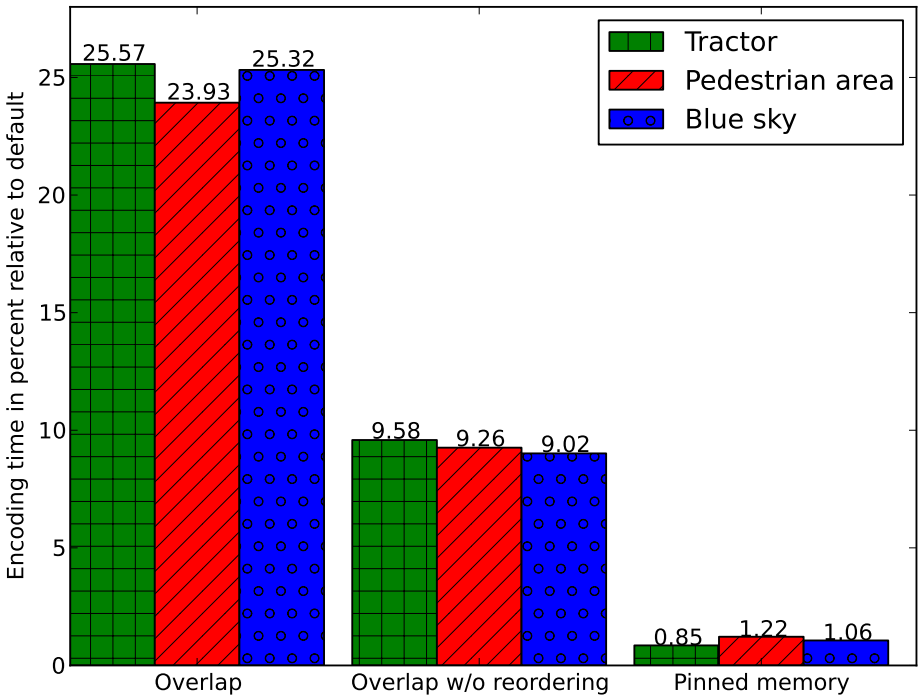

Figure 7.1: Improvement in encoding time due to streams and pinned memory.

sured the time spent in the `schedule_framestreaming()` function used to schedule an asynchronous memory transfer, versus the synchronous memory transfers and subsequent reordering in the existing code. As shown in table 7.1, the overlapped transfer is about ten times faster than the synchronous one, depending on the number of frames. As our test sequences are of limited length, the differences would be far larger for longer videos such as television shows and feature films.

Our results show that overlapping memory transfers had a positive effect on the over-

Implementation	tractor	pedestrian area	blue sky
Overlap	0.032722	0.0191878	0.0121936
Synchronous	0.3754249	0.2041722	0.1183091
Difference	11.47x	10.64x	9.70x

Table 7.1: Time used to copy and reorder input frames with and without overlapped transfer.

all encoding time, and reduced the time spent transferring raw frames to the device by about ten times for our test sequences, with a positive trend with increasing number of frames. We also determined that the improvements was to a lesser degree if we removed reordering of P-frames from the equation. To study the effect on reordering in detail, we compared the encoding time of host reordering with device reordering. As this step only accounts for a minuscule fraction of total encoding time, we expect the change to have little impact on overall encoding time.

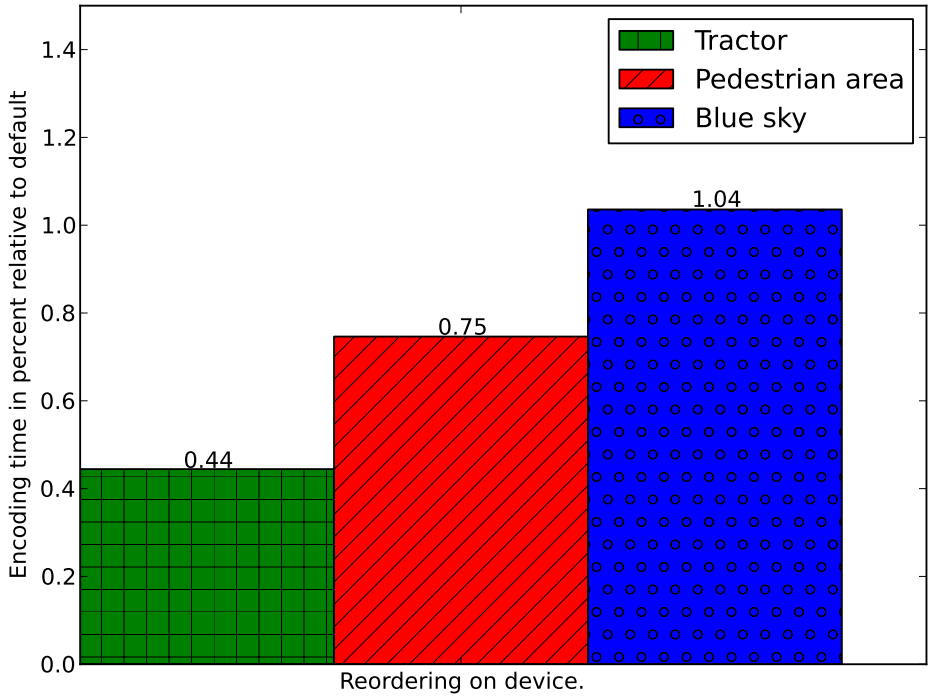

Figure 7.2: Improvement in encoding time due to reordering on device.

As shown in figure 7.2, the effect on device re-ordering is of little impact on the total encoding process in and of itself. Combined with overlapped memory transfers however, it increases the amount of work that can be performed concurrently. The re-ordering should also perform faster on the device, due to the concurrent memory accesses. To investigate this further, we measured the time spent performing the re-ordering including memory transfer, and the re-ordering in isolation. In addition to our regular test sequences, we measured the two first frames from *tractor*. This gave

Implementation	tractor	pedestrian area	blue sky	single p
Host re-order	1.1564997	0.6235881	0.3641031	0.0019702
Device re-order	0.7108312	0.3785024	0.218602	0.0010049
Difference	1.63x	1.65x	1.67x	1.96x

Table 7.2: Time used to re-order input frames on host and device, *including transfer*.

Implementation	tractor	pedestrian area	blue sky	single p
Host re-order	0.4293988	0.2283571	0.1357333	0.0009586
Device re-order	0.3727366	0.1986199	0.1147213	0.0005289
Difference	1.15x	1.15x	1.18x	1.81x

Table 7.3: Time used to re-order input frames on host and device, *excluding transfer*.

us the measurements for a single P-frame in the last columns. Table 7.2 and 7.3 shows the measured times with and without transfer time, respectively. As expected, removing the redundant memory transfer has naturally reduced the time spent copying the frames. The device re-order kernel was designed solely to enable this.

However, as table 7.3 shows, the re-ordering itself is handled faster by the GPU. While the effect is indeed minuscule on the total encoding process, it shows that the GPU is suitable for a broader workload than just heavy computation. Due to the multiple load and store units on each SM, the GPU can also handle more mundane tasks such as preprocessing. The difference in run-time might be minuscule or negligible in itself, but offloading more tasks onto the GPU reduces the synchronization points between the host and device. Using the asynchronous CUDA Streams API, multiple jobs can be scheduled on the GPU, freeing up the host CPU for other work. Using newer GPUs with compute capability 2.x further allows us to run multiple kernels in parallel, enabling efficient pipelining of tasks on the GPU without undue synchronization with the host.

7.4 Lessons learned

Our experiments shows that using CUDA Streams to overlap memory transfer reduces GPU idle time and consequently encoding time. While overlapped memory transfer is a benefit in and of itself, compute capability 2.x significantly expands this by enabling concurrent kernel execution. Without the limitation of only queueing asynchronous memory transfers, it is possible to schedule any kind of preprocessing (or post processing) work. As our results regarding device re-ordering suggests, there can be advantages in porting more trivial tasks to the GPU as well. While every task might not perform better on the GPU, it can be more time efficient to pre-process more work units on the GPU in parallel with the main computationally intensive task. This is particularly true for applications such as `cuve264b`, where the GPU is underutilized.

7.5 Summary

In this chapter, we expanded on our work in chapter 5, continuing the readahead caching onto device memory. Using the asynchronous CUDA Streams API, we overlapped the memory transfer and reordering of raw frames with encoding. This further reduced the overall encoding time by decreasing the waiting time between frames. While we expected the increase in performance to be lower than the disk readahead implementation, the asynchronous memory transfers turned out to improve even more than disk readahead.

Chapter 8

Deblocking

8.1 Introduction

As we mentioned in our introduction to the `cuve264b` encoder in chapter 4, the deblocking filter is currently implemented on the host. In addition to making the GPU idle, this also leads to extra memory transfers between the host and the device, as the filtered frames must be copied back to the device for further reference. By running the deblocking filter on the GPU, we could take advantage of its computing power as well as saving the mentioned data transfer.

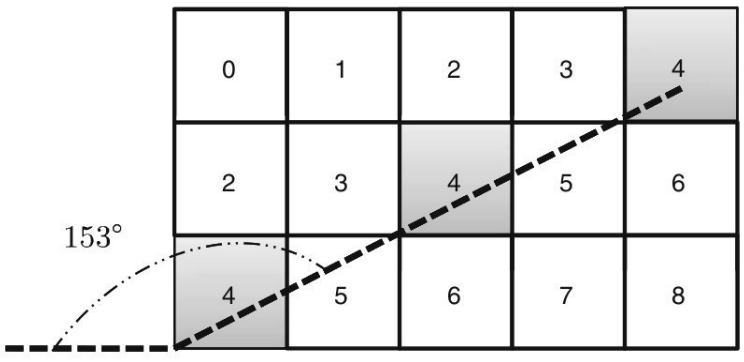

Figure 8.1: Wavefront deblock filter [7].

Due to the inter-dependencies between macroblocks, parallelizing the deblocking filter is a complex problem, especially on a massively parallel architecture such as a GPU.

There exists solutions with limited parallelism, such as the wavefront parallelization method. A wavefront deblocking example is shown in figure 8.1. Each square represent a macroblock, and the number inside shows in which pass the macroblock will be deblocked. The number of macroblocks processed in parallel is initially only one, and grows to a maximum when the wavefront reaches its widest length. After each pass, the state of the wavefront must be synchronized. [7]. While the wavefront approach might work with limited parallelism, it does not scale to the massive number of concurrent threads of a GPU.

Building on research by Sung-Wen Wang et al. [7], Bart Pieters et al. have shown that it is possible to implement the H.264 deblocking filter on massive architectures with only six synchronization points [8]. Due to the significant data dependencies between neighboring macroblocks in the deblocking filter, it is not straightforward to implement on massively parallel architectures. However, by analyzing the error propagation of parallel deblocking, Sung-Wen Wang et al. observed that the errors only propagate for a limited number of samples, called the limited error propagation effect [7]. However, Bart Pieters et al. found that the proposed parallelization method induced incorrect results, and hence not compliant with the standard. Their proposed modification, which they named deblocking filter independency (DFI), is shown in figure 8.2. By dividing each macroblock into different macroblock filter partitions(MFPs), the macroblocks can be filtered independently. Depending on the filter strength applied for a given edge, all samples in a MFP will not be filtered correctly after one pass. Hence, to perform the deblocking filter according to the standard, it is necessary to filter the MFPs in a strict order.

8.2 Design

Our approach to device deblocking was to port the host functions to device kernels as simple as possible. We could then independently verify the different kernels, and later integrate them with the encoder proper using the MB partitioning scheme proposed by Bart Pieters et al. [8]. After reaching function parity, we would optimize the kernels

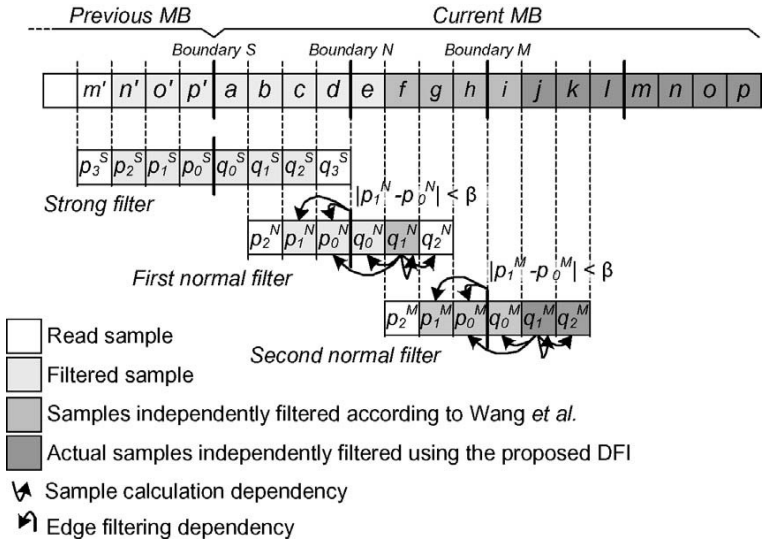

Figure 8.2: Limited error propagation with DFI [8].

with regards to minimizing branching, coalescing memory accesses and so forth. Before we could port the deblocking filter itself, we needed to port the underlying helper functions.

8.2.1 Implementation

We implemented the device kernels similarly to the unoptimized marshalling implementation in chapter 6, by replacing relevant loops with `blockIdx` and `threadIdx`.

As we laid out in section 2.4, the deblocking filter relies on a Bs score to determine filter strength. The current host code determine the Bs of an edge directly prior to the filtering operation. To adapt to the massive parallelism of the GPU, we rewrote the Bs determination code to evaluate all edges in a single kernel call. A similar refactoring would presumably also improve the host function, by better utilizing the CPU cache. As the Bs kernel consists of principally a series of branch evaluations, we expect it to perform poorly on the GPU.

After the deblocking itself has been performed, the frame is padded with edge pixels before it is used as a reference frame. As shown in figure 8.3, `PadAmount` numbers of pixels are filled with edge pixels surrounding the frame. Unfortunately, the CUDA

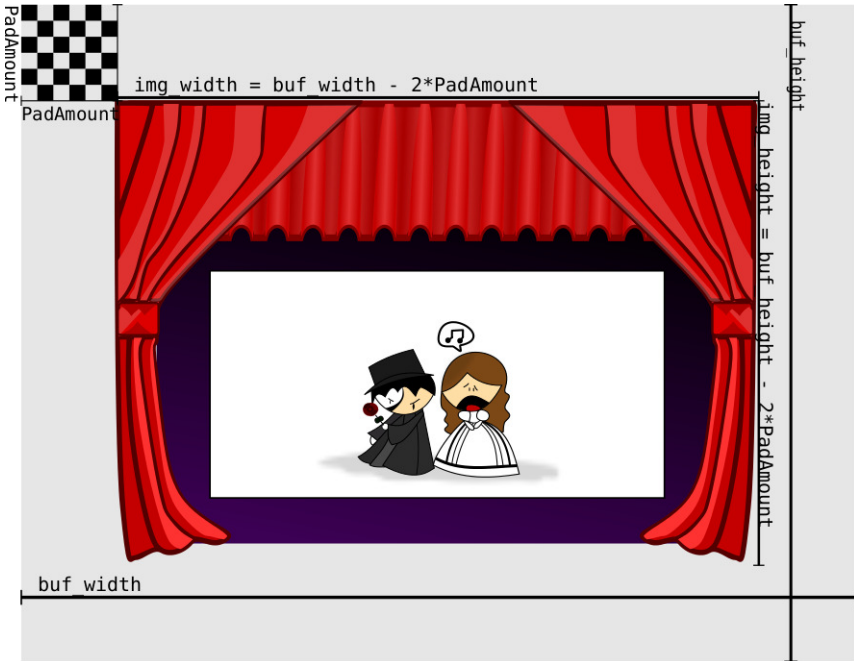

Figure 8.3: Padding of a deblocked frame prior to referencing.

thread model does not support other shapes than rows, planes or grids, so we cannot use a straightforward block hierarchy to pad the frame. To overcome this limitation, we use a single row for the outer grid, with a length corresponding to the circumference of the frame. The correct length is calculated as

$$grid = \frac{\text{buf\_width}}{\text{PadAmount}} * 2 + \left(\frac{\text{buf\_height}}{\text{PadAmount}} - 2 \right) * 2 \tag{8.1}$$

`buf_height` is subtracted by two so we do not count the corners twice. As one might deduce from the calculation, we use a 2D plane of `PadAmount*PadAmount` as block size. We then check which range the current `blockIdx.x` lies in, to determine the correct edge position. If it is less than `buf_width`, the current block is above the frame. Between `buf_width` and `buf_width+buf_height` is the left hand side of the frame, and so forth.

After the deblocked frame has been padded, it is copied to the reference frame buffer and re-ordered. Instead of the regular raster order, the frame is column flattened. Figure 8.4 shows an example where a 4x4 block is flattened. As the figure shows, each row of raster data is rotated 90 degrees to form a column.

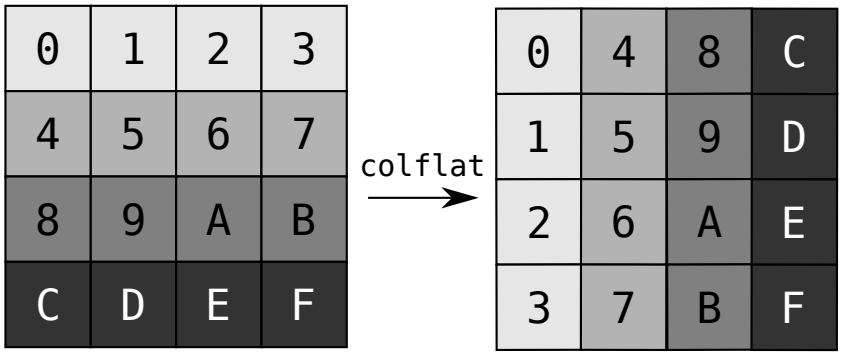

Figure 8.4: Column flatting of raster order.

The column flattening kernel used two outer loops to iterate between each row and column, with the inner body re-ordering four values for each iteration. Our corresponding kernel uses a 2D outer grid with 3D blocks, in which the Z-dimension corresponds to the four statements of the inner loop. Thus, each thread perform a single copy operation.

For the deblock filters proper, we did not manage to complete the implementation. The functions have been ported to kernel equivalents and compiles, but we have not completed the integration with the encoder or implemented the macroblock partitioning scheme.

8.3 Evaluation

As we did not manage to finish the implementation, we cannot evaluate the performance of the overall deblock filter. Nonetheless, we can still evaluate the helper functions we successfully ported. They do not give any indication of the performance of the deblock filter itself, but the number reported by the original research are promising [8]

Figure 8.5 shows the relative time difference for our kernels compared to the original host functions. As the figure shows, not all of the kernels ran faster than the respective host function. However, the ones that ran faster, contributed more to total difference than the one that was slower. The last series of bars shows the total runtime for all the helpers, which takes the relative contribution of the different kernel/functions into

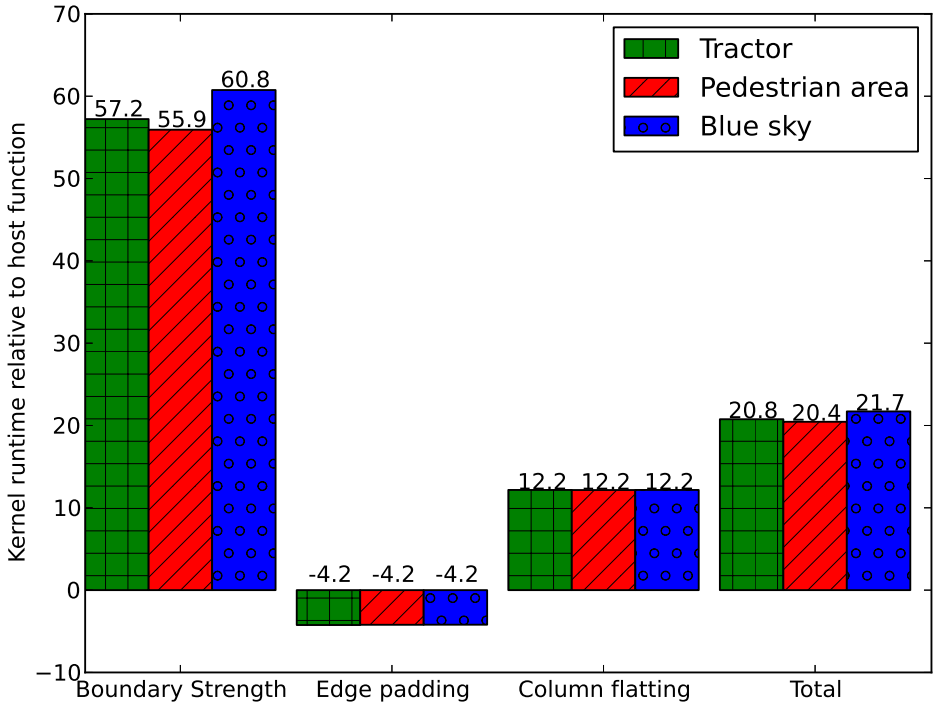

Figure 8.5: Performance of deblock helper kernels vs host functions

account.

In total, the device kernels perform the same operations about 20 times faster than the host functions. Especially the determination of Bs is significantly faster than we expected. Due to the large number of if-statements and resulting branches, we expected it to perform poorly. However, the massive number of concurrent threads allowed the GPU to brute-force all the different branches faster than the host CPU with its branch prediction.

We do note that the host function has much potential for optimization, so the comparison is not necessary fair. In addition to the weaknesses regarding caching described earlier, the host function only uses a single thread. While they certainly do not increment at the same speed as GPU cores, the number of cores of a CPU is increasing. Another notable difference is that the host code issued a separate function call for each MB. Cache issues notwithstanding, the high number of function calls could incur an

Relative runtime	Host function	Device kernel
Bs determination	0.734	0.266
Edge padding	0.003	0.358
Column flattening	0.358	0.448

Table 8.1: Relative runtime of deblock helper functions and kernels.

overhead. To quantify this, we inserted the `gettimeofday()` probes in the function body instead of the function call. However, the difference in comparison to the GPU was negligible.

Table 8.1 gives a clearer picture of the runtime of the various helper functions and kernels. As shown, the host spends nearly ¾ of the measured time determining Bs values. On the other hand, the edge padding is very efficient. However, as the device kernel is 50 times faster on Bs determination, the fact that the edge padding runs 4 times slower is of little concern.

8.4 Lessons learned

In this experiment, we first and foremost learned to plan our work more realistically. Regrettably, we did not find the time to complete the implementation. While much of the work is done, the remaining effort must be left for future work. This would also have allowed us to further improve on our state marshalling scheme covered in chapter 6, as the state shared between the device and host would be even less. In particular, the round-trip copying of the reference frame would be obsoleted.

That aside, the work we did perform supports our earlier results, namely that the GPU is able to efficiently perform other work than just heavy computation. While our port of the edge padding function shows otherwise, the GPU was able to outperform the host CPU for the port as a whole. The performance of the Bs determination kernel is particularly exciting, as the GPU still manages to perform on such heavily branching code, without any support for branch prediction.

As the interface between the host and the device becomes a tighter bottleneck, the looser we can couple the necessary synchronization, the better performance we can achieve. While certain tasks will surely perform poorer on the GPU, the reduction in data transfer between host and device will free up more bandwidth for data throughput.

8.5 Summary

In this chapter, we planned to port the deblocking filter onto the GPU. While we did not succeed in porting the filter completely, we managed to port and verify the helper functions used former and latter. Our evaluation of the helper functions support our results for the previous chapter, namely that the GPU can efficiently accelerate more tasks than just the most computationally intensive ones.

Chapter 9

Discussion

9.1 Introduction

In this chapter, we will discuss the results of our experiments, how the development in GPU technology effect GPGPU as a development platform, and how our findings can benefit other applications outside the scope of video encoding.

9.2 State of GPU hardware

In our introduction to GPUs in section 3.1, we highlighted the differences between a CPU and a GPU, and the trade-off between die space for control logic and calculation capacity. One consequence of this trade-off is that the SMs on the GPUs does not support branching: Whenever branching happens, every possible branch must be evaluated. Thus, branching should be avoided.

One strategy we employed on our earlier CUDA work [34] and in the chroma column flattening kernel covered in section 8.2.1, is to rewrite a potential branching in variable assignment with a calculation based on the predicates. For instance, we use the expression shown in listing 9.1 to choose the correct chroma channel in the chroma column flattening kernel. Depending on the value of `k`, `src` will either point to `in1` for Cb or

`in2` for Cr. While this prevents branching, it is not as self-documenting as the branching equivalent in listing 9.2. The latter is obvious and easy to maintain, but will result in two branches. A trade-off between readability and micro-performance.

However, Bs determination, which in essence is a series of branches, still outperforms the branch-predicting host CPU when executed on our Fermi GPU. Even without support for branching, the GPU can perform an exhaustive branch execution faster than the CPU. Such results makes us question if modern GPUs have become so powerful that writing *good enough* code should be the goal instead of spending much manual labour optimizing the GPU code. Unless the advancement of GPU processing power subsides, this trend is likely to continue. While we will not discard such optimizations as worthless, their effectiveness depend on peak utilization of the GPU. Otherwise, the only achievement will be an increase in GPU idle time.

```
1   /* As branching on z will cause divergent branches, we use pointer
2    * arithmetic to choose the correct image source without branching. */
3   char *src = (char *) ( (int)in1 * (k<2) + (int)in2 * (k>=2));
```

Listing 9.1: Branch-free variable assignment

```
1   char *src;
2
3       if(k <2)
4               src = in1;
5       else
6               src = in2;
```

Listing 9.2: Branching variable assignment

Another hardware disparity is the memory hierarchy we explained in section 3.3.3. While correct memory handling on the device originally involved strict requirements, they have been relaxed with newer hardware. As tested by Alexander Ottesen in his master thesis [38], the added flexibility in global memory coalescence and shared memory accesses enabled by compute capability 1.3 makes it easier for developers to achieve low latency memory accesses. This trend is amplified in the Fermi architecture, with the addition of level 1 and level 2 cache. While some algorithms might map poorly to coalesced memory accesses, the additional cache layers should improve the performance for such cases as well.

As shown in table 6.1, correct memory accesses can have a great impact on the number

of memory requests issued. In our case, the GPU time was reduced with 2% after little effort on our part. However, as our kernel did not do any computational work, we expect the impact to be greater on computational kernels, where memory latency reduces the occupancy. Thus, our results probably indicate a lower bound of optimization that is possible using targeted memory access optimization.

9.3 GPU Multitasking

As we noted in section 3.3.4, CUDA uses a programming model called SIMT. However, with the release of the Fermi architecture, it became possible to run at most 16 concurrent kernels [39]. This enables a *Multiple* Instruction, Multiple Threads model, thereby changing how we can interact with the GPU. In this regard, our work with overlapped re-ordering in chapter 7 is a meagre exploit of the possibilities that concurrent streams provide. However, they indicate the benefit of not only running multiple tasks in parallel, but also that the GPU can be efficient at performing such tasks. By moving more work onto the GPU, less synchronization and data transfer between the host and the device will be necessary. As the ratio between interconnect bandwidth and GPU processing power increases, it might be a net benefit to perform traditionally unsuited tasks on the GPU to reduce host-device communication.

With the original limit of single kernel execution, it was impossible to take advantage of task-level parallelism. As the computational power of GPUs increase, it will be harder for any given task to saturate the GPU alone. Task parallelism through concurrent kernels allows for a wider approach to GPU application design. This broadens the range of workloads that can be efficiently processed by a GPU. For instance, workloads which does not scale to the number of threads accessible to a single kernel, might scale using a hierarchical approach with both data parallelism and task parallelism.

As we described in section 2.6, the overall H.264 encoding process can be parallelized by means of multiple slices and/or independently processed GOPs. While the former is already implemented in `cuve264b`, the latter is unfeasible without task parallelism. As we have already established that the current design under-utilizes the GPU, a GOP-

based task parallelism might further increase the efficiency of the encoder. By managing a separate CUDA stream for each concurrently encoded GOP, such an additional layer of parallelism can easily be added.

Task parallelism also allow us to better structure our applications as separate tasks. By running tasks without further data dependencies in separate streams, we make sure to remove artificial bottle necks from the main progress. As an example, `cuve264b` uses a two-step CAVLC procedure as explained in [6]. Parts of the entropy coding that scales well run on the GPU, while the rest of the work is subsequently performed by the host CPU. As we mentioned in section 2.5, entropy coding is the last step performed before the encoded slice is written to the bitstream[1]. Most notably, the entropy-coded data is not used for further prediction. Thus, by running CAVLC in a separate stream, CAVLC coding of the current slice can commence in parallel with the encoding of the next frame.

9.4 Design emphasis for offloaded applications

To achieve GPU peak performance, it is important to design the application with overall performance in mind. As long as the GPU is under-utilized, spending manual labour further optimizing the device code will only increase the GPU idle time. The results of our readahead experiments in chapter 5 show the importance of a broad focus on performance. Even if a GPU has immense processing power, any task offloaded will still be bound by Amdahls law [10]; it will have to wait for sequential parts such as reading the dataset from a mechanical disk. By using classic optimization techniques on the host, such as asynchronous IO, we mitigate this cost. By minimizing the latencies incurred by the host, we decrease the GPU wait time.

This is not limited to external issues such as the seek time of rotational media. IO operations can be mitigated by investing in faster equipment such as high-end solid state drives. Any task which requires a nontrivial amount of processing or blocking time,

[1] Technically H.264 does not use a specific bitstream format, but rather a logical packet format referred to as a NAL unit [1]. However, NAL units are outside the scope of this thesis.

should be masked away. Every algorithm cannot be effectively parallelized, and many can only be parallelized in part. In such cases, it is essential to make sure that a sequential step does not become a tighter bottle neck than strictly necessary. Our writeback implementation is a poor example due to the output buffering performed in the C library. A more applicable example would be the second stage of the CAVLC entropy coding we brought forward in section 9.3. Using the `gettimeofday()` approach we employed in some of our experiments, we measured the second CAVLC step to occupy nearly 10% of the execution time. However, the scope of the CAVLC process is limited to the current frame, without any dependencies to future frames. Thus, the encoder could start processing the next frame in parallel. By running the entropy coding inside the main thread, the scope of bottle neck is artificially enlarged to cover the whole video sequence instead of a single frame. If the CAVLC step is performed in a separate thread, the bottle neck will only effect the slice in question.

Some algorithms might not scale to the number of threads necessary to be efficiently performed on a GPU. However, they might be suitable for limited parallelism, such as the comparably small number of cores in contemporary CPUs. To achieve peak application performance, it is important to also identify which parts can be efficiently parallelized on the host CPU. For instance, our test machine (see section 4.2) has four CPU cores. Except for the trivial CPU-time used by the readahead and writeback threads, only a single core is used in `cuve264b`. Even if we accounted the single core as 100% saturated, that still means that the CPU will idle 75% of the time. By taking full advantage of the host CPU as well, the algorithms in question will perform better, reducing both overall processing time and the waiting time of the GPU.

One example of such an algorithm is the H.264 deblocking filter. As we noted in chapter 8, large scale parallel deblocking has been researched by Bart Pieters et al [8]. However, our snapshot of `cuve264b` predates its publication. Nevertheless, deblocking can be parallelized for a limited number of processing cores through the wavefront method as explained by Sung-Wen Wang et al. [7]. As the deblock filter is an integral part of the encoding process, it will delay the encoding of subsequent frames. Hence, optimizing the host deblocking filter will reduce the time delay until the next P- or

B-slice can be processed. A subsequent I-slice can be encoded without delay as long as the deblocking filter is executed outside the main thread.

It is also important to be terse in the memory transfers between the host and the device. As we showed in chapter 6, global device memory can be utilized to keep GPU state through the lifetime of the application. By keeping such permanent state, it is possible to only synchronize updates between the host and the device that is strictly necessary. As state must be synchronized without any concurrent writes to guarantee correctness, it is essential to keep the synchronization time as short as possible.

9.5 Our findings in a broader scope

While we focused on H.264 video encoding for our thesis, our findings are applicable for all kinds of data driven workloads, and many are relevant for any kind of accelerated workload; being performed by a GPU, a number of Cell SPEs or a custom FPGA design. Common for all of them is the adage of keeping the device busy, by minimizing the impact of Amdahls law. To do so, it is important to design the application with responsiveness at the core.

Latencies experienced at the extremities of the application, such as reading input from rotational media, can easily be masked away by performing them in separate worker threads. By employing separate threads, the main thread need not perform blocking operations, and the worker threads can continue prefetching additional datasets in parallel with the computational task.

Using overlapping memory transfers provided by CUDA streams, a similar benefit can be achieved in transferring data between host and device memory. For GPUs supporting concurrent kernel execution, it is also possible to overlap any pre- or post-processing work associated with the transfer. As our results show, such additional processing steps might actually perform better on the GPU even without overlapping.

Concurrent kernel execution can also be used to implement task parallelism on GPUs. As earlier compute capabilities where limited to sequential kernel execution, task par-

allelism was not possible. While we have not performed any extensible experiments on task parallelism, the results of our minuscule attempts look promising.

While task parallelism broadens the scope of tasks applicable for GPU offloading, many algorithms do not scale sufficiently to be offloaded. In such cases, it is important to identify the data dependencies associated with the algorithms. If the algorithm is an integral part of the application, it might be possible to parallelize it using the host CPU. Otherwise, it might create a bottle neck tighter than necessary. To ensure peak performance, it is essential to focus on overall application performance, not solely the parts offloaded on the GPU.

If no further part of the application depend on the output of such an algorithm, it should run in a separate worker thread, similar to the input from rotational media mentioned to begin with. This allows the bottle neck associated with the limited scalability to be masked away. However, such algorithm may still cause a long tail in execution if not properly optimized.

Chapter 10

Conclusion

10.1 Summary

In this thesis, we have investigated Host-Device communication in GPU-accelerated applications. This was done using a hands-on approach, by performing experiments on an existing GPU-based H.264 encoder. However, our findings are relevant beyond the scope of video encoding.

During our investigations, we have designed and evaluated different approaches to reducing host-device communication, both in terms of data transfers on the PCI Express bus, and responsiveness in the host code.

We looked into how the host performed IO operations, and how this affected GPU waiting time. By implementing asynchronous IO in two separate threads, we managed to reduce the penalty of input operations to a single frame. The remaining frames would be read from disk in parallel with the encoding progress. The writeback implementation did not achieve similar performance. This led us to further investigate how write operations are handled lower in the operative system stack. Our meagre results could be explained by output buffers in both the C library and kernel. By repeating our tests with flushing of library buffers and kernel buffers, we determined that the buffers in the C library already mitigated our hypothetical output costs.

By analyzing state transfers between the host and the device, we identified which parts of state needed to be synchronized, and in which direction. Based on this, we designed a simple marshalling protocol to send partial updates instead of the full state. We implemented and evaluated two different variations of the design; one written with basis on the host, and another optimized for GPU memory coalescence. While the coalesced implementation performed better on the GPU, the penalty observer on the host led to a net loss compared to the host-based implementation.

While our readahead implementation efficiently masked away disk reads, it only placed the frames in host memory. Using CUDA Streams, we extended the transfer to include device global memory, overlapped with the encoding kernel runs. We also took advantage of concurrent kernel execution to perform reordering of input frames after transferring them to global memory. Overlapped transfer in combination with re-ordering yielded even better results than the original readahead implementation, exceeding our expectations. Not only did the overlapping mask away the associated delays, but the re-ordering kernel itself performed about 15% better than the original host function.

We started implementing the H.264 deblocking filter on the GPU. Regrettably, we did not finish the implementation. Our deblocking code compiles, but we did not find the time to integrate our work with the encoder. Nonetheless, we could verify some of the helper kernels we successfully ported. Of the three kernels combined, the net gain was about 20 times faster than the respective host functions. The kernel that exceeded our expectations most was the Bs determination kernel. With a body consisting of mainly branch statements, we expected it to perform poorly. However, it ran about 50 times faster than the host function equivalent. While one of the kernels performed poorer than the host function, the net result where significantly faster on the GPU. This supports our findings on device re-ordering, and shows some of the possibilities in the concurrent kernel support.

10.2 Further work

During our thesis work, we have looked at different strategies to improve the Host-Device communication in `cuve264b`. While we managed to carry out many experiments, we would have looked into many more if we had more time.

A natural starting point would be to finish our work on a GPU-based deblocking filter. As the code is already written, it only needs integration with the encoder proper and an undisclosed number of debugging hours. Completing the implementation would improve the performance of the deblocking filter itself [8]. In addition, the deblocked frame would not need to be copied back to the device, saving a (padded) frame transfer for each frame. The state sent from the device to the host would also be reduced, as the information to correctly choose Bs on the host would be obsoleted.

The performance potential in concurrent kernel execution could probably be the starting point of another master thesis on its own. As we noted in the discussion, task parallelism provided by multiple streams make it possible to implement a hierarchal multiple slices and multiple concurrent GOP parallelization strategy, similar to that described by A. Rodríguez et al. [26].

Concurrent kernels could also be used to experiment with running the different image channels in parallel. For instance, one stream could encode the Y component, while another could handle Cb and Cr.

Similarly, concurrent streams makes it possible to run the first part of the CAVLC entropy coding in a separate stream, independent of the previous encoding steps.

The second part of the CAVLC step, performed by the host, could be separated into a new thread. In addition to better utilizing the multiple cores of a contemporary CPU, this would also mask away most of the time spent performing CAVLC in the main thread today.

10.3 Conclusion

We have shown that when writing GPU-accelerated applications, we were able to achieve significant performance increase by focusing our efforts on the host implementation, and the Host-Device communication. As the processing power of GPUs increase, the effective rate of utilization will be limited by Amdahls law [10]. To reach peak performance, it is important to minimize the sequential parts before and after the GPU computation. Such sequential parts include IO operations, memory transfers between the host and the device, and any nontrivial computations performed on the host CPU.

Appendix A

Code Examples

A.1 CUDA Vector Addition Example

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <assert.h>
4
5  #define ARRAY_SIZE 1024
6  #define ARRAY_SPACE ARRAY_SIZE*sizeof(int)
7
8  /* Trivial kernel doing vector addition */
9  __global__ void vec_add(unsigned int *a, unsigned int *b, unsigned int *result)
10 {
11         /* calculate global index */
12         int i = blockIdx.x * blockDim.x + threadIdx.x;
13
14         /* Read both operands from global memory and write the result back. */
15         result[i] = a[i] + b[i];
16 }
17
18 int main(void)
19 {
20         /* device pointers */
21         unsigned int *dev_input1, *dev_input2, *dev_output;
22
23         /* statically allocated host memory */
24         unsigned int input1[ARRAY_SIZE], input2[ARRAY_SIZE], output[ARRAY_SIZE];
25
26         int i;
27
28         /* Step 1: Allocate memory on device */
29         cudaMalloc((void**)&dev_input1, ARRAY_SPACE);
30         cudaMalloc((void**)&dev_input2, ARRAY_SPACE);
31         cudaMalloc((void**)&dev_output, ARRAY_SPACE);
32
33         /* Set some default values */
34         for(i=0; i<ARRAY_SIZE; ++i)
35         {
36                 input1[i] = 3735879680;
37                 input2[i] = 48879;
38         }
39
40         /* Step 2: Transfer input to device global memory.
41                    Note the last parameter, which gives
42                    the direction of the copy operation  */
```

```
43          cudaMemcpy(dev_input1, input1, ARRAY_SPACE, cudaMemcpyHostToDevice);
44          cudaMemcpy(dev_input2, input2, ARRAY_SPACE, cudaMemcpyHostToDevice);
45
46          /* Step 3: Launch kernel, note the shorter form of kernel call
47                     without shared memory allocation or cuda stream */
48          vec_add<<<ARRAY_SIZE/64, 64>>>(dev_input1, dev_input2, dev_output);
49
50          /* As the kernel call is asynchronous, host code continues until a
51           * synchronous cuda call, such as a memory transfer or an explicit
52             cudaThreadSynchronize() waits for the gpu to complete */
53
54          /* Step 4: Retrieve results. The host process will block until completion.*/
55          cudaMemcpy(output, dev_output, ARRAY_SPACE, cudaMemcpyDeviceToHost);
56
57          /* Step 5: ????  */
58          printf("0x%X+0x%X=0x%X\n", input1[0], input2[0], output[0]);
59
60          /* Step 6: Profit */
61          return EXIT_SUCCESS;
62  }
```

Listing A.1: Vector addition; a trivial CUDA example

References

[1] T. Wiegand, G. J. Sullivan, G. Bjøntegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, 2003.

[2] I. E. G. Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia*. Wiley, 2003.

[3] J. Heo and Y.-S. Ho, "VLC table prediction algorithm for CAVLC in H.264 using the characteristics of mode information," vol. 5353 of *Lecture Notes in Computer Science*, pp. 119–128, Springer, 2008.

[4] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. Morgan Kaufmann, 2010.

[5] N. Goodnight, R. Wang, and G. Humphreys, "Computation on programmable graphics hardware," *IEEE Computer Graphics and Applications*, vol. 25, no. 5, pp. 12–15, 2005.

[6] N. Wu, M. Wen, W. Wu, J. Ren, H. Su, C. Xun, and C. Zhang, "Streaming HD H.264 encoder on programmable processors," in *Proceedings of the 17th International Conference on Multimedia 2009, Vancouver, British Columbia, Canada, October 19-24, 2009*, pp. 371–380, ACM, 2009.

[7] S.-W. Wang, S.-S. Yang, H.-M. Chen, C.-L. Yang, and J.-L. Wu, "A multi-core architecture based parallel framework for H.264/AVC deblocking filters," *The Journal of Signal Processing Systems*, vol. 57, no. 2, pp. 195–211, 2009.

[8] B. Pieters, C.-F. Hollemeersch, J. D. Cock, P. Lambert, W. D. Neve, and R. V. de Walle, "Parallel deblocking filtering in MPEG-4 AVC/H.264 on massively parallel architectures," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 1, pp. 96–100, 2011.

[9] P. List, A. Joch, J. Lainema, G. Bjøntegaard, and M. Karczewicz, "Adaptive deblocking filter," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 614–619, 2003.

[10] G. Amdahl, "Validity of the single-processor approach to achieving large-scale computing requirements," *Computer Design*, vol. 6, no. 12, pp. 39–40, 1967.

[11] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation - A performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[12] Denning, Comer, Gries, Mulder, Tucker, Turner, and Young, "Computing as A discipline," *CACM: Communications of the ACM*, vol. 32, 1989.

[13] M. Livingstone, *Vision and Art: The Biology of Seeing*. Harry N. Abrams, 2002.

[14] C. Poynton, *Digital Video and HD: Algorithms and Interfaces (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, 2002.

[15] G. J. Sullivan and T. Wiegand, "Video compression: From concepts to the H.264/AVC standard," *Proceedings of IEEE*, vol. 93, pp. 18–31, Jan. 2005.

[16] Y.-K. Chen, X. Tian, S. Ge, and M. Girkar, "Towards efficient multi-level threading of H.264 encoder on intel hyper-threading architectures," in *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, IEEE Computer Society, 2004.

[17] T. W. Ralf Schäfer and H. Schwarz, "The emerging h.264/avc standard," *EBU Technical Review*, vol. 293, 2003.

[18] Y. Ismail, J. McNeely, M. Shaaban, and M. A. Bayoumi, "Enhanced efficient diamond search algorithm for fast block motion estimation," in *International Symposium on Circuits and Systems (ISCAS 2009)*, pp. 3198–3201, IEEE, 2009.

[19] M. Karczewicz and R. Kurceren, "The SP- and SI-frames design for H.264/AVC," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 637–644, 2003.

[20] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the scalable video coding extension of the H.264/AVC standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, pp. 1103–1120, Sept. 2007.

[21] N. Efford, *Digital Image Processing: A Practical Introduction Using Java (With CD-ROM)*. Addison Wesley, 2000.

[22] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Prentice Hall, 2007.

[23] H. S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-complexity transform and quantization in H.264/AVC," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 598–603, 2003.

[24] J.-B. Lee and H. Kalva, *The VC-1 and H.264 Video Compression Standards for Broadband Video Services (Multimedia Systems and Applications)*. Springer, 2008.

[25] D. Marpe, T. Wiegand, and H. Schwarz, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 620–636, 2003.

[26] A. Rodríguez, A. González, and M. P. Malumbres, "Hierarchical parallelization of an H.264/AVC video encoder," in *Fifth International Conference on Parallel Computing in Electrical Engineering (PARELEC 2006)*, pp. 363–368, IEEE Computer Society, 2006.

[27] M. Doerksen, S. Solomon, and P. Thulasiraman, "Designing APU oriented scientific computing applications in openCL," in *13th IEEE International Conference on High Performance Computing & Communication, HPCC 2011, Banff, Alberta, Canada, September 2-4, 2011*, pp. 587–592, IEEE, 2011.

[28] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kr"uger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hard-

ware," in *STAR Proceedings of Eurographics 2004*, (Grenoble, France), pp. 21–51, Eurographics Association, Sept. 2004.

[29] E. Wu and Y. Liu, "Emerging technology about gpgpu," *IEEE Asia Pacific Conference On Circuits And Systems*, pp. 618–622, 2008.

[30] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar, "Interactive multi-pass programmable shading," in *Siggraph 2000, Computer Graphics Proceedings,*, Annual Conference Series, pp. 425–432, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[31] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, 2003.

[32] D. Blythe, "The direct3D 10 system," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 724–734, 2006.

[33] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.

[34] H. K. Stensland, H. Espeland, C. Griwodz, and P. Halvorsen, "Tips, tricks and troubles: optimizing for cell and GPU," in *Network and Operating System Support for Digital Audio and Video, 20th International Workshop, NOSSDAV 2010, Amsterdam, The Netherlands, June 2-4, 2010, Proceedings*, pp. 75–80, ACM, 2010.

[35] T. A. da Fonseca, Y. Liu, and R. L. de Queiroz, "Open-loop prediction in h.264/avc for high definition sequences," XXV Simposio Brasileiro de Telecomunicacoes, Sep 2007.

[36] S. B. Kristoffersen, "Utilization of instrumentation data to improve distributed multimedia processing," Master's thesis, University of Oslo, Norway, May 2011.

[37] P. B. Beskow, H. K. Stensland, H. Espeland, E. A. Kristiansen, P. N. Olsen, S. Kristoffersen, C. Griwodz, and P. Halvorsen, "Processing of multimedia data using the P2G framework," in *Proceedings of the 19th International Conference on Multimedea 2011, Scottsdale, AZ, USA, November 28 - December 1, 2011*, pp. 819–820, ACM, 2011.

[38] A. Ottesen, "Efficient parallelisation techniques for applications running on GPUs using the CUDA framework," Master's thesis, University of Oslo, Norway, May 2009.

References from the Internet

[39] nVidia Corporation, "Fermi compute architecture whitepaper." http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[40] nVidia Corporation, "Nvidia cuda c programming guide, version 4.0." http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, June 2011.

[41] J. Moyer, "Ensuring data reaches disk." https://lwn.net/Articles/457667/, september 2011.

[42] M. H. Weik, "The eniac story." http://ftp.arl.mil/~mike/comphist/eniac-story.html, 1961.

[43] G. E. Moore, "Cramming more components onto integrated circuits." ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf, april 1965.

[44] S. C. E. America, "Ps3 firmware (v3.21) update." http://blog.us.playstation.com/2010/03/28/ps3-firmware-v3-21-update/, march 2010.

[45] P. S. I. Group, "Pci-sig delivers pci express 2.0 specification." http://www.pcisig.com/news_room/PCIe2_0_Spec_Release_FINAL2.pdf, january 2007.

[46] P. S. I. GRoup, "Pci-sig delivers pci express 3.9 specification." http://www.pcisig.com/news_room/PCIe_3_0_Release_11_18_10.pdf, november 2010.

[47] nVidia Corporation, "Geforce 8800 series specifications." http://www.nvidia.co.uk/page/geforce_8800.html.

[48] nVidia Corporation, "Geforce gtx 680 specifications." http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-680/specifications.

[49] nVidia Corporation, "Cuda visual profiler." http://developer.nvidia.com/nvidia-visual-profiler.

[50] S. Rennich, "Cuda c/c++ streams and concurrency [webinar slides]." http://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf, January 2012.

[51] ITU-T, "H.264 : Advanced video coding for generic audiovisual services." http://www.itu.int/rec/T-REC-H.264-201003-I/en, May 2003.

[52] ISO/IEC, "ISO/IEC 14496-10:2003, 2003. information technology - coding of audio-visual objects - part 10: Advanced video coding." http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43058, May 2003.

[53] ITU-T and I. J. 1, "Generic coding of moving pictures and associated audio information: Video. ITU-T recommendation H.262 and ISO/IEC 13818-2 (MPEG-2)." http://www.itu.int/rec/T-REC-H.262, July 1995.

[54] ITU-T, "H.261 : Video codec for audiovisual services at p x 384kbit/s." http://www.itu.int/rec/T-REC-H.261-198811-S/en, Nov. 1988.

[55] N. L. Xiaoquan Yi, Jun Zhang, , and W. Shang, "Improved and simplified fast motion estimation for jm (jvt p021)." http://wftp3.itu.int/av-arch/jvt-site/2005_07_Poznan/JVT-P021.doc.

[56] L. Aimar, L. Merritt, J. Garrett-Glaser, S. Walters, A. Mitrofanov, H. Gramner, and D. Kang, "X264." http://www.videolan.org/developers/x264.html.

[57] nVidia Corporation, "nvidia ion specifications." http://www.nvidia.com/object/picoatom_specifications.html.

[58] N. Brookwood, "Amd white paper: Amd fusion™ family of apus." http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf, March 2010.

[59] I. Corporation, "Products (formerly sandy bridge)." http://ark.intel.com/products/codename/29900.

[60] Microsoft, "Programming guide for hlsl." http://msdn.microsoft.com/en-us/library/bb509635(v=VS.85).aspx.

[61] T. K. Group, "The opengl shading language v4.20.8." http://www.opengl.org/registry/doc/GLSLangSpec.4.20.8.clean.pdf, september 2011.

[62] nVidia Corporation, "Geforce gtx 680 whitepaper." http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf, 2012.

[63] T. M. Technik, "1080p test sequences." http://media.xiph.org/video/derf/ftp.ldv.e-technik.tu-muenchen.de/pub/test_sequences/1080p/ReadMe_1080p.txt, 2001.

[64] J. Seward, C. Armour-Brown, J. Fitzhardinge, T. Hughes, N. Nethercote, P. Mackerras, D. Mueller, B. V. Assche, J. Weidendorfer, and R. Walsh, "Valgrind instrumentation framework." http://valgrind.org.

[65] nVidia Corporation, "NVIDIA CUDA library: cudahostalloc." http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/group__CUDART__MEMORY_g15a3871f15f8c38f5b7190946845758c.html#g15a3871f15f8c38f5b7190946845758c.