Processing Multimedia Workloads with Intel Many Integrated Core Architecture

A comparative study with work scheduling

Sigurhjörtur Snorrason

Thesis submitted for the degree of
Master in Programming and Network

60 credits

Department of Informatics

Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2016

Processing Multimedia Workloads with Intel Many Integrated Core Architecture

A comparative study with work scheduling

Sigurhjörtur Snorrason

Processing Multimedia Workloads with Intel Many Integrated Core Architecture

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

This thesis has a main goal of measuring how the Xeon Phi coprocessor performs with multimedia, especially video encoding. The research focused on utilizing OpenMP and Cilk, both of which are language extensions for C, with an Intel Xeon Phi coprocessor and CUDA, also a language extension for C, with an NVidia GeForce Titan GPU to optimize a simple video encoder. The encoder used was Codec63 which uses a 2D Discrete Cosine Transform JPEG compression and motion estimation. We optimized the encoder through parallelization and used offloading, which is when you use the host CPU to transfer data to the Xeon Phi through instructions in the code as a way to communicate between the host CPU and the Xeon Phi coprocessor. The testing also focused on work scheduling algorithms for parallel execution and we used static, dynamic and guided scheduling for OpenMP along with the work stealing algorithm Cilk uses. Our focus was mostly on parallelizing the motion estimation as the analysis proved it to be the most time consuming part of the encoder. We compared the improvements between the three language extensions and found that OpenMP with static scheduling performed best on the Xeon Phi requiring about 40% of the time the other scheduling algorithms needed. We also found that a comparable CUDA implementation still performed better than the Xeon Phi by about 10%.

Acknowledgements

A thesis is never work of the author alone. There are a number of people that assist and help, both directly and indirectly. This is certainly the case with this thesis and I would like to acknowledge the people who have helped me.

First of all any research like this can never be completed without backing of both family and friends. So I would like to thank my family for supporting me both in days of frustration and happiness. Special thanks to my partner for her patience towards me while I have written this thesis and my mother who has helped me with both support and proofreading.

Also to my friends and fellow students, whom I have shared the study room Assembler with. They have always been willing to give advice in times of need.

And last but not least to my supervisors, Preben Olsen and Håkon Kvale Stensland, for the help they given by answering any questions I have had throughout the research. I would also like to offer a special thanks to professor Pål Halvorsen who helped read through the thesis and gave advice towards the end of the research.

I hope you as a reader enjoy this thesis and it offers you some insight into both the field of multimedia and the hardware used in the research.

Contents

List of Figures

List of tables

List of Code Examples

Chapter 1

Introduction

In recent years, the evolution of computational power has moved to being focused on multi-cores. This comes as a result of hardware reaching a point where we can no longer cool it down at higher CPU frequencies than we currently have [36]. The key to utilize the power of a CPU then is concurrent programing on multiple CPUs. The most commonly used method for concurrent programming is threads. Threads, however, are not just threads as there are multiple ways of setting up and using threads, both in regards to hardware and then various scheduling algorithms. Multimedia is also an important part of daily life and to get the most out of computers we need to consider parallelism for multimedia.

1.1 Background and Motivation

The research in this thesis focused on parallel multimedia code which is likely to be important due to the recent explosion in multimedia [28] of all types. We are moving more towards a world were we can find all we want in a streamable format. YouTube is one of the more popular websites [3] [38] where everything is streamed, and there are a number of similar sites. As multimedia calculations can be very complex and require a lot of computational power using single threaded programs is not optimal. Especially as some of the calculations can often be done in parallel without problems. Research into multimedia use is therefore more important than ever.

One type of hardware utilizing multiple cores is the Xeon Phi. It is based on the Larrabee microarchitecture [21]. Originally, the Larrabee microarchitecture included graphics processing support and there were plans to create a graphics processing unit (GPU) from the research on the Larrabee [37]. As the Xeon Phi was based on this architecture, it bears a bit of resemblance to GPU with the cores it has. However, the actual

Xeon Phi is purely meant as a High Performance Computing (HPC) device. We will call the Xeon Phi non-heterogeneous because of how it requires a host processor for the work we do even if the internals of the Xeon Phi are heterogeneous.

Another type of hardware is the General-purpose computing on graphics processing units (GPGPU). The GPGPU has come a long way from being just a GPU [10]. GPGPU is in short the ability to do calculations on a GPU, and originally, almost solely applied to computations in relation to graphics. However, more recently, they have become more accessible for all types of other calculations that benefit from parallelism.

What will be interesting to see is how the Xeon Phi coprocessor fares against the GeForce Titan GPU as the media encoding done in this thesis will have little load imbalance between threads. This is true as media encoding contains a lot of calculations based on balanced vectors. The GeForce Titan GPU is optimized towards drawing, shading and in general doing smaller calculations. It is therefore expected that the GeForce Titan will achieve better results than the Xeon Phi. All our runs will also be done on the host machine where both the GPU and the Xeon Phi are hosted so that we can compare runs to how they would do on just the host computer.

It is important to do studies on how suited CPU and HPC are for multimedia workloads and comparative studies between GPGPU and HPC. GPU's have been the go to hardware for calculating graphics, including images and other code that could be run in parallel. However, the Xeon Phi is starting to offer new ways of doing so, and it is therefore important to follow how viable the Xeon Phi is. Especially interesting also, is to see how various scheduling algorithms can affect execution times of the parallelization and the speed of the Xeon Phi.

It is also intersting to see how easy it is to manage moving code onto multi-core platforms. The same can be said about the CUDA platform. The coding is not something you can access straight away but will require a somewhat step learning curve to get benefits of running code through it.

As can be understood, there are a lot of potential problems that have to be solved for multimedia. To keep up with demands we need to be able to process multimedia faster and faster. Not only that, but we also need to find a way to do it in a smarter way as CPU power has a limit to how much help it can offer us.

1.2 Problem Statement

With this in mind, the aim in this thesis is to test various ways of utilizing multi-core architecture to optimize a naive and simple video encoder, called

Codec63. Manycore architectures will be examined, specifically the Intel Xeon Phi, in regards to usability. Another examination will be done on a GPU. The thesis will also compare the three language extensions for C we have available, Open Multi-Processing (OpenMP), Cilk and Compute Unified Device Architecture (CUDA) along with a baseline code written in plain C. All three language extensions offer various ways of handling our Codec63 encoder with Cilk and OpenMP running through the Xeon Phi and CUDA through the GeForce Titan GPU.

We will be looking at both the hardware and the ease of programmability. The main hardware used, as stated, is the Xeon Phi and we will compare it to both a GPU and a CPU. The focus will be on how efficient it is to compare hardware and how easily you can code optimized code. We also want to be able to predict problem areas with using the Xeon Phi, which complexities does it give and which areas suffer from badly optimized code? As is known, the Xeon Phi is mostly used as a high performance computing (HPC) accelerator [34] so the aim is also to find out how well it performs in regards to multimedia. Furthermore, the aim is to find more and better ways to handle multimedia. This is important as more and more of computing is going into handling exactly multimedia. Seeing how GPUs have always been the best setup for multimedia it will be interesting to find out how the non-heterogeneous architecture of the Xeon Phi compares.

Part of the research will also be focused on checking how various scheduling methods affect the Xeon Phi. We will test the code with static, dynamic and guided scheduling. We will also look at testing with a mix between the scheduling methods during a single run. We expect that static scheduling will perform better than the other two, but when doing runs with much code and very broad loops that guided and dynamic would outperform and therefore that a mix between static and the other two methods will produce the best results.

We will use the Codec63 [39], a modified variant of the MJPEG encoder. The main difference is that the Codec63 adds intra-frame dependencies compared to MJPEG. This means that Codec63 behaves much like H.264, but in a much less complex way. The Codec63 is meant for teaching and is therefore not how you would assume an encoder would behave but serves a purpose due to the stripped down functionality.

Due to the structure of the Codec63 it is expected that most of the improvements can be found within the motion estimation. This is because of the amount of data each macroblock will go through to reach a result about what the best motion vector is. Because of how the search is done for the motion vectors we can easily share data between threads without many memory access problems for the macroblocks as they cover a relatively large area. However, the deeper into the search pattern one would go with the parallelization the closer each memory access gets. Finding this cut-off point is therefore important. The DCT part however is a much smaller area

of data which will require a very strong division between work areas.

The proposal is that it should be easily achievable to get a speed increase for the code with parallelization. However, reaching the full potential of the hardware will require a lot of effort due to how the encoder is constructed. We also expect to see serious effects from incorrect use of memory access. Due to how the parallelization works in for example OpenMP the expectation is that controlling these memory conflicts will be very hard.

It is expected that the results will fit with what has currently been researched. That good use of the Xeon Phi is a bit harder to do than one would expect. That means that everyone should be capable of some improvements over the original code but not on par with what the Xeon Phi can offer. This is because of how simple OpenMP is and memory control is more done by the compiler than the actual programmer, meaning that while everyone should be able to get results the advanced users can expect to be limited by this [18]. We also expect to see that getting stronger results should be easier on the GPU than the Xeon Phi, although smaller difference on videos with smaller resolution.

We also expect a big ramp up in complexity based on how much we attempt and manage to get out of the Xeon Phi. That is the most optimal coding on the Xeon Phi requires a lot of work so reaching the max theoretical computing speed of the Xeon Phi will be very hard.

The goal of the CUDA part is to compare it to the OpenMP and Cilk implementations so we will be making a implementation mimicking the OpenMP and Cilk setup. CUDA has a very different approach as everything is set up in warps and each thread is more lightweight. That also means that each thread requires less overhead to start. With that in mind the approach would usually be more on maximizing thread usage than ease of coding. As such, the OpenMP and Cilk codes are usually more based on portability and ease of implementation making a portable CUDA code is a challenge on its own, and will be interesting to see how it compares when created in a portable way.

This, however, is not as easy as one would expect as the CUDA language expects that total threads and threads per block should be set up before compiling meaning that dynamic loops based on variable thread counts require some extra calculation when coding. The same setup in both OpenMP and Cilk much simpler to do and if this is as we expect using the Xeon Phi would be easier than the GPU.

1.3 Limitations

All the tests will be done with Codec63 encoder. The experiments will only cover the parts of Codec63 that are relevant, which are the ones relating to the actual encoding. Hard disk drive input and output operations are not relevant to this research. However, other input and output operations, like offloading, are. The reason we exclude hard drive operations is that both the Xeon Phi and the Kepler GK110 do not directly interact with them. They always use some intermediary device. Hard disk drive optimizations are therefore outside the scope of this research.

Codec63 will be run with four different videos, foreman.yuv, tractor.yuv, bagadus.yuv and foreman_4k.yuv. Each of these videos are raw YCbCr format videos with each frame split into the three aspects with no extra data. Foreman.yuv is a 352x288 resolution video, tractor.yuv is a 1920x1080 resolution video, bagadus.yuv is a 4096x1680 resolution video and foreman_4k.yuv is a 3840x2160 resolution video.

We will do 100 frames of encoding per video. The keyframe interval is set by default to 100 frames, so we get the full content between frames per run. Even if the videos do contain varying amount of motion and frequency changes which we might get more of as we run more than 100 frames this is outside the scope of the thesis. This would be more a measurement of how well the Codec63 works in regards to compression and loss of quality and the efficiency of the codec. However, our goal is to see how the Xeon Phi does and compares to the Kepler.

This thesis is will not be looking towards finding the best and fastest use of either the Xeon Phi or the GeForce Titan. The goal is a comparative study and looking at how the Xeon Phi does with multimedia work. As such the code still has other potential optimizations that will not be explored.

We only compare the Xeon Phi to a Kepler GPU. The Kepler GPU is closest to the Xeon Phi we have based on the time of release and it therefore offers the best comparison.

We have decided on using C, Cilk, OpenMP and CUDA for our programming. Even if the Xeon Phi supports Fortran through OpenMP as well, including it should not give us any new results. Fortran would utilize the same paralellization extensions through OpenMP. The encoder we use for testing is also written in C so to test Fortran the encoder would need to be fully rewritten in Fortran. This would only give us a comparision between our encoder written in C and another one written in Fortran which is not what we are looking for in this thesis.

We will not compare SIMD operations on either the Xeon Phi or the Kepler GPU or do assembler instructions specific to the Xeon Phi. This is as Intel's C++ Compiler (ICC) is based on C++ compiler and does not do well with

C based SIMD instructions. As we are limiting the Xeon Phi to non-SIMD instructions we also do so with the GPU. This is something that should be considered for a different thesis and we discuss that in chapter 6.2.

The thesis will also not look towards improving the way the Codec63 encoder works or different encoding algorithms. We will not look at ways to improve either the encoder without parallelization or other related applications. The Codec63 has plenty of areas which can be improved directly, but we chose a very simple encoder to be able to monitor performance. In addition only the basic difference between H.264 and Codec63 will be covered. H.264 has more complex set up and does not suit well for our experiments. Improving the Codec63 encoder directly would therefore hinder the research.

1.4 Research Method

Most of the research will be based on the methodology specified by the ACM Task Force on the core of computer science [16]. The main methodical approach will be quantitative as most of the actual research will be focused on implementing prototypes of Codec63 and from running experiments on the Xeon Phi compared to other platforms.

The project focuses mainly on an actual implementation while being independent from other projects. This means that our role in the whole project covers mostly everything required. The key studying points are therefore the actual performance of the implementation and how easy it is to implement. Also, the data collected is not sensitive so there is a very limited ethical aspect to this research.

1.5 Main Contribution

This thesis will show that getting better results on the Xeon Phi is very much possible. However, offloading operations, where we transfer data to the Xeon Phi, limit the results, especially in regards to multimedia. It will be shown that static work scheduling will provide best results for multimedia when only using a single scheduling algorithm. This is due to how much of multimedia code is structured and does not cause load imbalance. The best result, however, comes from a combination of scheduling algorithms.

We found out that doing to much parallelization impeded the performance of the Xeon Phi. This was specially apparent when we parallelized operations in our sum of absolute differences. All the calculations there happen in a 8x8 pixel block and the Xeon Phi suffered heavily from cache misses.

We also found out that GPUs are still better at parallelizing multimedia code than the Xeon Phi, even if the Xeon Phi has taken big strides towards reaching the level of the GPU. We did this by creating as identical code as possible in C, Cilk, OpenMP and CUDA. This should give the best way of comparing. A few tests were made that aimed at looking into code for the Xeon Phi that are more advanced than what we will do in CUDA to see how the Xeon Phi fares at an even higher level.

The results show that the actual encoding was faster on the Xeon Phi. But because the overhead from offloading the results from our runs show that the native execution was faster. What also has to be taken into account is that the native PC is fairly powerful meaning that the native encoding might be faster just in this case.

The Xeon Phi did not do well with code that allocated and freed memory per frame as the Codec63 encoder was set up to do. A much better approach was reusing memory as much as possible. We therefore altered the encoders basic behavior more than was expected.

Generally according to the study, Cilk performed worse. Mostly this was due to memory control with shared memory but also because of the work stealing algorithm Cilk uses. Changing the behavior to not update the shared memory per memory access but instead only when the offloaded section was completed gave us results that should be expected with work stealing.

1.6 Outline

In this chapter the thesis was introduced. This includes the goal and limitations along with the motivation behind it.

In the next chapter the detail of the software and hardware we use will be presented, along with a bit of history behind it. The work scheduling algorithms in general will be discussed and the ones used with our hardware.

Chapter three is about video processing. This includes the theory behind MJPEG, JPEG and H.264. It includes the algorithms we will mostly be looking at.

The fourth chapter contains an explanation of Codec63, the encoder we will be using for encoding. Chapter four also details the design decisions made in this thesis in regards to how to work on the code. This includes the ideas for what can be optimized for Codec63.

Chapter five details on how the implementation was made, results from the implementations and a discussion about them.

We conclude the thesis in chapter six by summarizing our work and discussing potential future work based of this thesis.

There are two appendices at the back, one containing the tables with the runtime data and the other with code examples of the optimizations done to the code.

Chapter 2

Hardware and Scheduling

In this chapter, it will be described how the hardware for the thesis works. The chapter starts with explaining the Many Integrated Core Architecture (MIC) and the language extensions that apply to it. MIC in this case is the core structure in the Xeon Phi. This is followed by details around GPU and GPGPU programming. A GPU is mostly meant to render and display images. However, the thread structure it has for displaying images makes it also very useful for parallel calculations. Lastly, the chapter discusses the work scheduling algorithms that will be looked at and used in the hardware being used.

We will discuss a bit in chapter 2.3 that we want to find out how the hardware does with multimedia. A part of that is to understand how our hardware and scheduling algorithms work, to get an idea on both how to implement and how to recognize the results we get from the implementation. This will both help us understanding what the hardware does and then what code is good code and what code might not be that good.

2.1 Intel Many Integrated Core Architecture

The Xeon Phi is a multiprocessor PCI card made by Intel. The idea is that one can use it for multithreaded computations, but unlike their CPU counterparts it has a much higher number of cores. This is then comparative to normal GPU's of the same type [37] even if GPU's tend to have higher number of threads.

Xeon Phi is the branding name of the Intel Many Integrated Core Architecture (MIC). It started out with Intel adding 512-bit SIMD units to the x86 architecture, with capability of four-way multithreading. This was based on research done with the Larrabee architecture. This was the

basis for the Knight's Ferry, the Knight's Corner and the Knight's Landing. The Knight's Ferry is the prototype of the MIC and was announced 2010. It had 32 cores with 4 threads per core and reportedly reached over 750 GFLOPS. The first generation was the Knight's corner which has 50 cores per chip. Second generation is the Knight's Landing, which includes 72 cores, while Intel has announced that it will release the third generation with 10nm cores [34].

The basic structure of the Xeon Phi evolves around a ring interconnect connecting caches, tag directories, memory controllers and a PCIe Client Logic [14] as we see in figure 2.1. Each core comes with a private cache connected to this ring interconnect and the cache is kept full by a tag directory. The memory controllers provide a direct interface to the GDDR5 memory on the coprocessor and the PCIe Client Logic does the same towards memory through the PCIe bus. The interconnect ring is a bidirectional ring with each direction having three rings. These rings are a data block ring, address ring and an acknowledgement ring. They then communicate between the various controllers and are for example used on cache misses to check other caches and memory. This allows for a streamlined and high bandwidth performance between multiple cores.

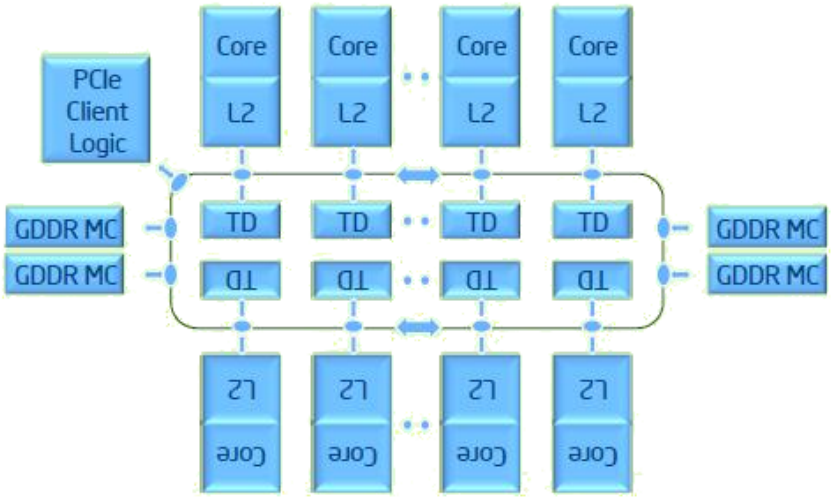


Figure 2.1: Xeon Phi Architecture [14]

The Xeon Phi with a Knight's Corner MIC will be the basis for testing the theories with how to be handle migration of work between processors. As the Xeon Phi is fairly simplistic in use while very powerful, we should be able to get many details around how to best do native work stealing. The MIC has support for OpenMP, OpenCL, Cilk and then specialized versions of Fortran and C++ from Intel.

This gives excellent ways of testing in various languages that have different

setups when it comes to work scheduling in regards to work stealing and work sharing, both of which are discussed in chapter 2.3.

2.1.1 Capabilities of the Knight's Corner

The Xeon Phi used in the experiment is based on the 22nm Lithography with a base frequency of 1.2ghz with 61 cores along with 8MB shared coherent cache and 2GB DDR5 memory. The Xeon Phi has excellent computing power, well exceeding that of your normal CPU. The Intel Xeon Phi is a heterogeneous platform, but can be considered not being one because of how much work it does in collaboration with the CPU.

However, to utilize it, one has to be proficient in coding in a multi-core environment. Even if compilers have gotten far with improving performance, they are far from perfect and require a lot of help from the programmer. Therefore, without proper coding techniques, one can not except to achieve the capabilities that the Xeon Phi can reach [19]. A key thing to consider when parallelizing code, is false positives on memory accesses. A false positive memory access happens when a core tries to access a memory region that is in close proximity to another region that has just been access. The proximity here is a few bytes at most. The cache will then think that memory is also accessible through it, but it is not. This will then cause a second lookup where the memory is moved. This can cause a big degradation in performance in the code and is something that is very important to consider when optimizing code.

2.1.2 Using the Xeon Phi

The Xeon Phi supports multiple ways of utilizing it. In this section the primary focus will be on the OpenMP language and on the Cilk language for testing. OpenMP has an easy way of taking current C code and utilizing the threads on the Xeon Phi. Cilk is then a C dialect based purely on having multi-threaded purpose. There are also other languages that can be used but they are outside the scope of this thesis. One way of running the code can then be through offloading, where certain areas of the code get transferred to the Xeon Phi during execution time. Other options to offloading are for example by using a Message Passing Interface (MPI), which lets processors and coprocessors communicate without offloading, or by running the code directly on the Xeon Phi [24]. We can still use offloading with MPI though. Both ways will be tested to see how comparable they are. A big factor to consider though is that code can not always run natively as data might often be transmitted from other sources, such as the hard disk, various types of PCI cards and other ways. In those cases the data would also first need to be transferred which makes transferring only certain calculations more effective. This is

also very important to consider as GPUs can not be used directly as host environments for calculations as the Xeon Phi can.

A very important thing to consider is that it is possible to utilize multiple Xeon Phis as the same time. The research in this thesis is only aimed towards one but using multiple cards can alter the way you need to think in regards to transfers of data. In that way, one card might become the host offloading to another one, or you might divide work by offloading to both cards from the host computer. This might alter if the way you want to communicate or transfer data between various processors or coprocessors. The biggest thing one has to consider though is the overhead of the transfer versus the limited memory on Xeon Phi when deciding on offloading versus using other methods like MPI.

Offloading

Offloading work and data to the Xeon Phi is most common way of utilizing the Xeon Phi. Offloading uses pragmas to allocate and deallocate data areas and to transfer data to these areas. Offloading can be done by moving data in before the start of the offloaded section, moving data out after the offloaded section, moving data both in and out and then by not transferring any data. Not transferring data usually applies to data that has already been transferred or just to allocate memory to store other data.

Code Example 2.1: Example of an offload

```
1  int a = 0, *b, *c;
2
3  b = malloc(N);
4  c = malloc(N);
5
6  #pragma offload target(mic) inout(a) \
7    in( b : length(N) alloc_if(1) free_if(1) ), \
8    out( c : length(N) alloc_if(1) free_if(0) )
9  {}
10
11 #pragma offload target(mic) \
12   nocopy(c : length(N) alloc_if(0) free_if(1))
13 {}
```

An example of how to offload can be seen in code example 2.1. In this example we can see three variables being transferred. The integer named a gets transferred in and then out. As this is a single integer we do not need to specify the size of it. Then we see a integer pointer called b being transferred in. As this is a pointer we need to specify the size of it. This is done by the length(N) in the call. We then allocate that memory area on Xeon Phi with alloc_if(1) and at the end of the offloaded section we free the

memory with free_if(1). The c variable is also an integer pointer but will only be transferred from the Xeon Phi after the offloaded section ends. It has the same setup as b otherwise except we do not free the memory used after the offloaded section. This is denoted by the free_if(0). We then have a second offload. This offloaded says we will use the c variable again, but this time without any data transfer. It was allocated earlier so we do not need to allocate it again and say so with the alloc_if(0) specifier. We then free the memory when we are done with free_if(1). These are the basic ways we can transfer data and work.

In regards to offloading, it is important to be aware of what is transferred to the Xeon Phi, what is declared on the Xeon Phi and what does not need transferring. Also, it is important to be aware of what gets transferred back. These operations have varying costs and need to be cost efficient to be worth it. According to Jeffers and Reinders [24] the offload model of the Xeon Phi is quite rich, and the syntax and semantics are generally a superset of other offload models. This is the reason that OpenMP can so easily manage and control multiple offload targets during execution time and is an important consideration when deciding if offload is the best way.

Native Xeon Phi Execution

There are other ways of running code through the Xeon Phi. It is also possible to run the code natively on the Xeon Phi. This, instead of moving parts of the code during execution time to the Xeon Phi, moves all the code before execution time to the Xeon Phi memory. Then you got options of running the code on either platforms but communicate important data between processors and coprocessors with message passing. The native code works much like a host running it and we got all the options we would in the host. The message passing lets us transfer important information between various processors and coprocessors that are running [24].

The project will be utilizing offloads most of all. There will be a small test with native executions but considering the size of the data we are working with, we are somewhat limited. By using just one Xeon Phi the research itself is also directed towards offloads as message passing requires multiple Xeon Phis to effectively test.

2.1.3 OpenMP

OpenMP is an API for programming in C and C++. It can also be used for Fortran based programs but this thesis will not cover any details about Fortran. It is used by giving directives to the compiler, library routines and environment variables to control the parallelism of programs. The core

is the constructs for thread creation using so called pragmas. This way it creates work sharing between cores on a given environment. See figure 2.2.

The key goal with OpenMP is to make it easy to convert linear code to parallel programs [20]. This is done by these pragmas that can easily convert loops into parallel executions. However, one must be truly careful with memory access during preprocessing of the pragmas. It is also possible to select if you want to run the whole thing natively or by offloading certain regions of the code. While in short this is fairly easy, it adds a big complexity factor to it.

The key part of OpenMP is as stated the actual use of pragmas to parallelize code regions. As these are just compiler instructions, they are fairly easy to implement, meaning even programs that have no parallel support can with a few lines become fully parallel. However, this does not guarantee a speed up as best parallel performance happens through contiguous memory while you should not have two threads accessing the same memory area. If two threads access the same memory area you can get so called false sharing hits.

In figure 2.2 you can see the various extensions that OpenMP offers. These include parallel control structures, synchronization directives and runtime functions. It is important to know these to be able to utilize OpenMP.

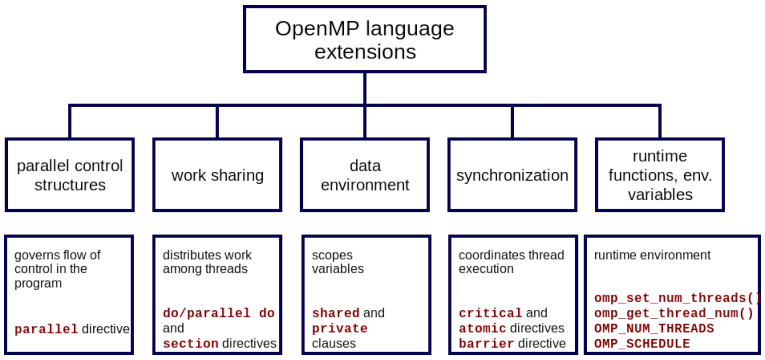


Figure 2.2: OpenMP language extensions [26]

2.1.4 Cilk

The Cilk language started as three separate research projects at MIT [15]. Later on, it was carried on by the Cilk foundation and made into Cilk++ before being acquired by Intel. The key factor with the language is that the programmer is responsible for assigning the areas that can be run in parallel while the scheduler takes care of the actual multi-threading. To assign which things can run in parallel, one sets a so called spawn on a function call. This defines that the function can be run in parallel and the language takes care of the rest.

Code Example 2.2: Example of Cilk code

```
1  void runaround() {
2    int i;
3    cilk_for(i = 0; i < 10; i++) {
4      sleep(1);
5    }
6  }
7
8  int main() {
9    cilk_spawn runaround();
10   printf("Hello World");
11   cilk_sync;
12   return 0;
13 }
```

So as an example, we can create a simple loop as we see in code examples 2.2. The loop should be run in parallel with the cilk_for instruction while we also execute a simple instruction, the printf, in parallel before syncing all the threads on the cilk_sync call. As we can see the code changes are minimal compared to normal code and the coder does not need to do much work.

2.1.5 SIMD Intrinsic

As noted before the Xeon Phi is based on the Larrabee architecture. Part of what remains from the Larrabee architecture is SIMD intrinsics. SIMD intrinsics are CPU instruction specially targeted towards vector calculations. The basis for graphics calculations is vector instructions, often in 2D or more. So, SIMD instructions are mostly used for graphics processing. The biggest difference from what you find on standard CPUs, which is AVX today, compared to our Xeon Phi is that you have a maximum size of 512 bit registers instead of 256 bit.

2.2 Graphics Processing Units and CUDA

A GPU is in short a piece of hardware meant for processing images or videos. As images consist of a lot of pixels that are independent of each other, a parallel drawing of it often tends to be more efficient. You can then assign a pixel or many pixels if needed to a single thread, which then draws it while the other threads draw other ones. As they are independent of each other there is little parallel synchronization required.

GPU's have in general been based on having many but smaller cores than the average CPU. This makes rendering images a lot easier [9]. The first and

most important detail starts with the fact that GPU threads are extremely lightweight and have very little overhead in creation. However, it also means that to gain a proper effect a GPU requires thousands of threads to be effective compared to a standard CPU.

With this in mind, it is worth considering that given a large data calculation we can do lots of smaller calculations faster with the multiple threads a GPU offers than is possible with fewer but more powerful CPU threads in our programs. So, using a GPU or a GPU based setup is something that could easily help with these large data calculations. This is something that NVidia has started working on with their GPU's and especially in regards to how CUDA works [10]. CUDA is in short a basic application programming interface that allows software developers to access core mechanics of a GPU. This is further backed by the fact that GPU's have also been used in games for various non graphical calculations, such as physics effects.

Commonly with a normal CPU you got very large caches with a much more sophisticated control. This includes branch prediction and data forwarding, all to reduce latency. However, a common GPU setup has very small caches, with simpler control. It does not have branch prediction or data forwarding. In general, this low latency setup of a CPU lowers run time on things like sequential code, which is where the GPU suffers a bit. Both in regards to latency and with things like individual thread control. However, as can be understood, with GPU based programming you overcome the latency issues with very heavily pipelined throughput. This means that the actual starting setup is somewhat shared but as soon as the starting part is done and the pipeline starts we have a very easy way of starting a very high number of threads at once.

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110	KEPLER GK210
Compute Capability	2.0	2.1	3.0	3.5	3.7
Threads / Warp	32				
Max Threads / Thread Block	1024				
Max Warps / Multiprocessor	48		64		
Max Threads / Multiprocessor	1536		2048		
Max Thread Blocks / Multiprocessor	8		16		
32-bit Registers / Multiprocessor	32768		65536		131072
Max Registers / Thread Block	32768		65536		65536
Max Registers / Thread	63			255	
Max Shared Memory / Multiprocessor	48K				112K
Max Shared Memory / Thread Block	48K				
Max X Grid Dimension	2^16-1		2^32-1		
Hyper-Q	No			Yes	
Dynamic Parallelism	No			Yes	

Table 2.1: Kepler GK110 and related architectures [31]

2.2.1 Kepler GK110

The GTX Titan was launched by NVidia in 2013, based on the Kepler GK110 core [25]. The GK110 is mainly based on a refresh of the Kepler microarchitecture and as such the Titan card inherits much of the Kepler architecture from before. It also has noticeable improvements and differences from the Fermi cores, see table 2.1. The most noteable difference being the max amount of warps and max registers per thread. A warp is the amount of threads that can and should run in unison. This will be explained in more detail later.

The card draws it name from the supercomputer Titan in Oak Ridge National Laboratory in the United States [13]. The reason for drawing the name from the supercomputer is that the GeForce Titan uses an identical GPU to the ones designed for the Oak Ridge supercomputer, although the supercomputer had 18,688 of them. The design and cooling is based of the GeForce GTX 690. The card has 2,688 CUDA cores and can run 14 warps at the same time. This obviously gives a gigantic potential GFLOPS but reaching that potential is not that easy and might even be impossible. This is due to how little overhead each thread has meaning that things like caching for non aligned memory is severely limited. With very streamlined memory access and aligned programs, however, it is easier to reach this limit. The structure of the Kepler GK100 can be seen in figure 2.3. From the structure we can see that we have 32 cores for every operand collector. This is the most important thing for us to consider from the GK110 for our research which we cover in more detail later.

A GPU works much like a CPU. Still, they have more cores that are each smaller with a smaller overhead but also run at a lower speed. They are, however, focused on processing vertices. To lower the overhead it has smaller controllers and cache and use more cores per controller and cache. This way you can have multiple threads working in parallel in a smaller area within a matrix. The CPU has a bigger cache but for fewer threads meaning that work within said matrix requires more overhead to set up. This lets us do parallel operations that are closer to each other in memory without many issues and is the main difference between the CPU and the GPU. One can obviously draw the conclusion that CPUs does better when it comes to calculations that can not be as easily shared or with varying workload per core.

2.2.2 CUDA

CUDA is a parallel computing platform and an application programming interface invented by NVIDIA [30]. It allows programmers direct access to the GPU cores through the GPU's virtual instruction set and parallel computation elements. This allows execution of GPU kernels. Much like

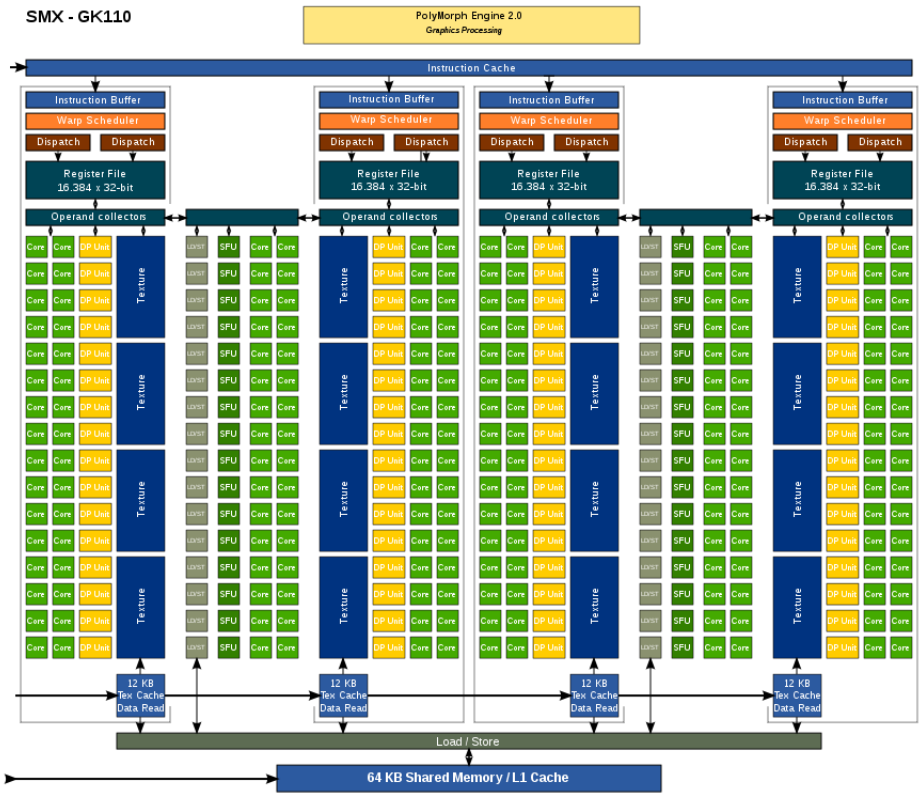


Figure 2.3: Kepler GK110 Architecture [41]

OpenMP and Cilk, CUDA is designed to work with C, C++ and Fortran, although our focus will be on the C part only. Having the language work with other well known languages makes them more easily accessible to more programmers compared to previously when you needed to be sufficient at graphical programming.

In the beginning, GPU's were meant to be graphics accelerators only that only supported specific fixed function pipelines. Over time GPU's became more and more programmable with people utilizing the capabilities for parallel programs on them. However, before CUDA, programming on them was not straightforward. You needed to map your problem solving calculations to triangles and polygons. This obviously meant that programming on a GPU was not for everyone.

In 2003, however, researchers from Stanford lead by Ian Buck [11] made the first compiler and execution time implementation of stream programming aimed at highly parallel GPU's. This allowed most of the GPU to be exposed as a general purpose processor. NVidia then got Ian Buck to join them and together they developed CUDA for an efficient way of coding on GPU's.

CUDA is in short a way to communicate directly with the GPU through various APIs. The basic idea is that the user starts a number of batches

of threads making the GPU into a super-threaded massively data parallel co-processor. This is possible as each thread in the GPU has very little overhead and is therefore very cheap to start up, so you would want to have multiple threads running. We can see the structure of the Kepler GK110 in figure 2.4. As we can see the cores are blocked together in groups of 32. This block is often called a streaming multiprocessor (SM). The cores in the SM are then called a warp. To explain a warp we can say that each SM has 8 streaming processors. Each of these streaming processors can execute 4 instructions each concurrently through threads. As our work is done per multiprocessor in regards to cache and controllers each SM executes 4x8 instructions or 32. This is also apparent in the figure as we can see a single dispatch unit (DP unit) for every four cores. Because of this communication between these cores tends to be very low and they perform badly with separate instructions between each other. In CUDA threads should therefore always have a minimum of 32 threads running per block. This block can then also run any multiple of 32 threads. However, being so lightweight we also require up to thousands more threads per normal CPU thread.

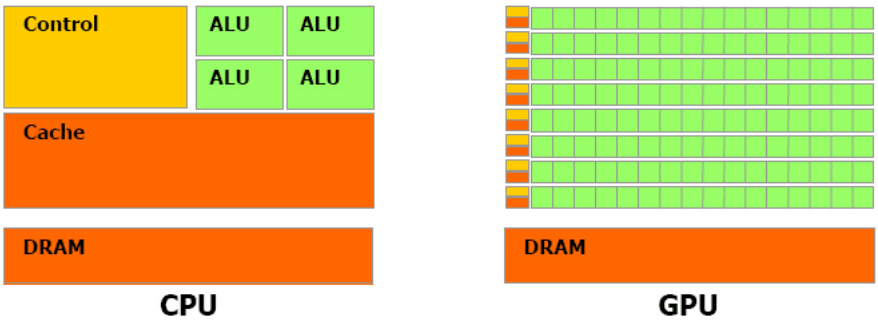


Figure 2.4: CPU vs GPU [29]

2.2.3 Difference in Coding Environments

All three languages extensions, OpenMP, Cilk and CUDA, bring something different to the table and all have big variations on what you need to think about. The biggest difference is allocation and usage of memory, especially in relation between the host memory and the platform memory. Cilk uses mostly shared memory, that is, the compiler will create a memory area on both the host and the platform and then keep both sides up to date [24].

OpenMP will create a memory area on when an offload starts through the offload pragmas before transferring data to it. This gives more control on the actual offload but makes it so that you must allocate stuff inside the CPU first in most cases. There are special cases where you can allocate first in the offloaded area before doing so on the CPU, but that is outside the research within this paper.

CUDA requires that memory is allocated on the platform first in a specific command before you can transfer data to it. The data transfers in CUDA are also a potentially big bottleneck and CUDA therefore supports that the transfers can be run asynchronously.

All three methods have benefits and disadvantages. Having memory on the platform directly accessible from the host like in the case of Cilk gives potentially big benefits on heavily I/O based applications, but it might cause drawbacks with more compute intensive applications. The OpenMP way gives very strong control over memory and memory allocation without the user needing to know much about how it does it. As you can also keep certain memory section available between offloads with the correct use of allocation, transfers are potentially faster after the first setup. CUDA, however, has a much more controllable way of handling memory with each step needing to be very carefully calculated. This way you know exactly where each segment you work on is. Still, at the same time, this means that CUDA demands more from the user implementing code to get good execution times. One can easily increase the execution time with badly controlled memory access and allocation. All this control, however, helps in cases where you can have very streamlined and aligned memory which are vital to utilizing the potential power of the GPU. In our case, our encoder is very easily streamlined so we should expect to see bigger benefits from the GPU than the Xeon Phi.

2.3 Work Scheduling

Work scheduling is the way one distributes work to a processor or multiple processors and can have important effects on execution of a program. As an example we can take what Vrba discusses in his paper [44] where he finds that work stealing on fine-grained parallelism has problems. To solve this a simple alteration gives considerably speedup. As a result using a optimal and good work scheduling can change much for the code running. With parallel programs, it is important to know what kind of scheduling you are using to get the best effect. However, as Lee [27] states, “For concurrent programming to become mainstream, we must discard threads as a programming model.” Currently, we rely too heavily on threads to do all the concurrent work for us. Threads are not native and need to be explicitly set up for the program. As multi-core architectures are here to stay, there is a need for a native way to create programs that can use the power available. Creating a program that is portable and adaptive to different hardware is very complex though. As different hardware might have different amount of cores or communication between cores we need something that can handle it. Work stealing offers us a way to control data loads between cores while being portable. However, currently work stealing has severe limiting factors and [8] [1] that shows, given certain conditions, work stealing can have a negative effect on run speed on multi-

core platforms compared to a single core. Therefore a key part of being able to create a programming language that can intuitively use all the power available to it in an efficient way is by having a more efficient way of various work scheduling algorithms.

There are also other methods available such as work sharing. In work sharing, instead of threads trying to take work from each other, the work gets split up. The threads get assigned parts of the work, complete it and get more work, if there is any work left. The benefits of this are substantial, especially in cases where the workload is evenly balanced. This removes the overhead between threads as they search for more work by just grabbing work from a queue.

2.3.1 History of Work Scheduling

Around 1980, computers starting having an issue with scheduling work properly with regards to parallel execution of programs. This came as a result of computers having increased processors which meant better scheduling was required. The idea stems from the start of the 1980's with Burton and Halstad doing research on parallel execution of functional programs [8]. Halstead then implements it with his programing language Multilisp [22]. The idea is that Multilisp uses an unfair scheduling. By giving each task an active pending status the language can pick up the task that's pending if there are free processors to do so. This is done by creating queues which, unless someone takes it, are just executed on the same processor. But, if one processor gets free, it checks the queue and grabs it. That way we get parallelism between both.

2.3.2 Work Stealing

Above the basic idea of the work stealing is described. Most of the research early on is done by people looking towards getting parallelism in to functional programming languages. Later, people moved to languages such a Cilk for multi-threading. Cilk is a multi-purposed programming language with a clear C style feel. It has been upgraded to include much of the C capabilities. The newer versions Cilk have support for both child stealing and continuation stealing [15].

Work stealing is in short a scheduling mechanism. The goal is to even out the load per processor and make sure that no processor runs out of work. During the runtime of a computer, it will be granted many tasks. These are then divided in a queue per processor. Each one of these processors might then spawn other tasks from their current tasks that they can run in parallel with its current one. However, this means that a processor might not have any tasks currently while another has a number of tasks. These

tasks might be runnable in parallel. So, to make sure the system operates optimally, these tasks should be running at the same time on two different processors, instead of having one idling while the other one does all the tasks. If the task is queued on the working processor and not the idling one, the idling one should go and steal it from the running one. This way we can ensure optimal load and work rate with the tasks it gets. Work stealing can, however, suffer from to much communication or to little time given to process each task [45].

You then have the variations between how to steal work. The two most common are continuation stealing [8] and child stealing [6].

Child Stealing

As a process runs, it might spawn a new thread of execution, a so called child as we see in figure 2.5. This then gets placed in a double sided queue. The work can then be stolen from the bottom while the current process works from the top. The process then creates a child process to execute that task that report back to the parent. However, sometimes the work can not be fetched out of the queue straight away. As the waiting task waits another process might finish somewhere else and become idle. When that happens, it will try to take work from others and by stealing the childs that are waiting in line to be executed.

Continuation Stealing

The other way is so called continuation stealing like in figure 2.6. This is done by the parent thread executing the new threads much like one would execute function calls in C. That way a given function call completes before the next one. As such, the parent executes the child directly and queues the function calls. However, the function call might be stolen by a different process. This is then continuation stealing.

2.3.3 Work Sharing

This differs from the so called work stealing algorithms. In work sharing, the scheduler tries to move certain tasks to processors that look to be underutilized [8]. The key difference here is that in work stealing there is no migration of tasks unless the processors requests them, while in work sharing there is regular migration. As a result, work stealing results in only data needing to be transferred being transferred. However, this does have the side effect that the actual processors have to look for work to migrate to themselves.

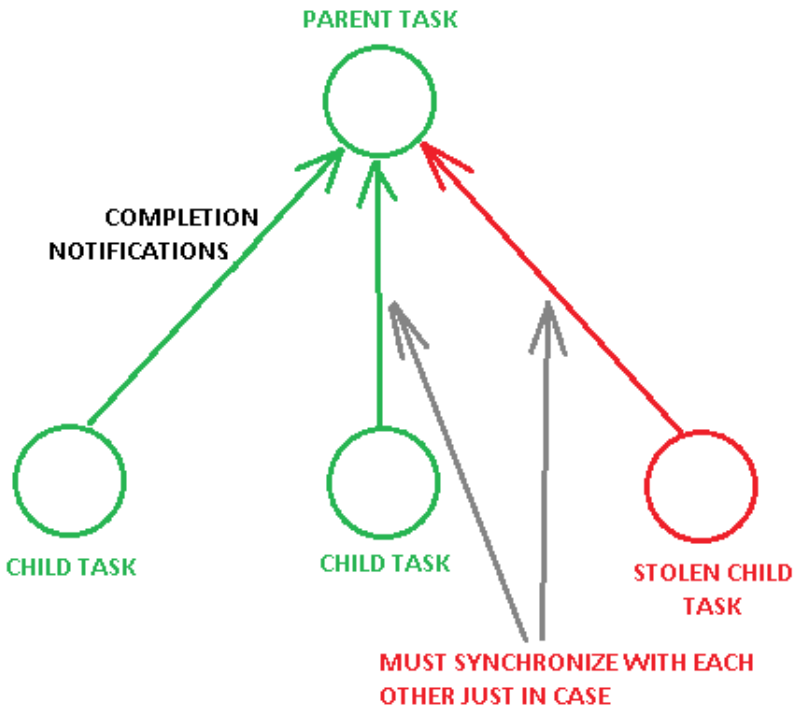


Figure 2.5: Child Stealing [46]

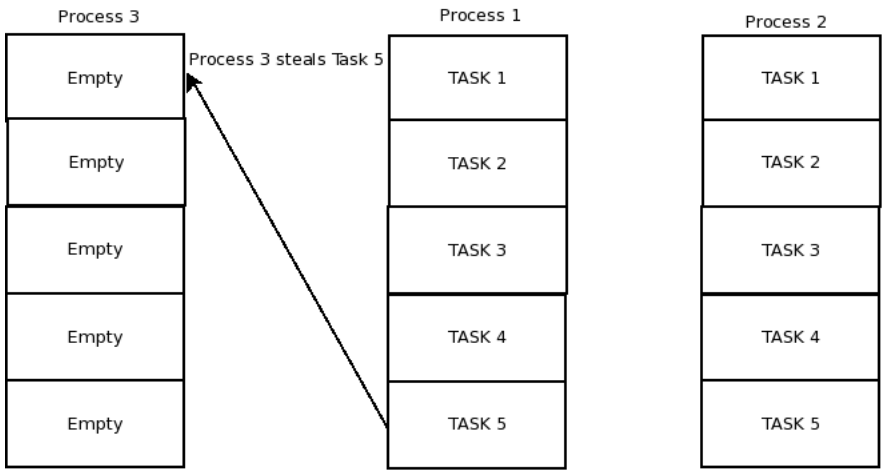


Figure 2.6: Work Stealing

The work can then be divided between all threads where each thread has a queue and will be the only one working on that queue. We could also have a single queue of jobs where each thread grabs the first task in the queue and completes it. Both have benefits and drawbacks like in most cases. Having a single queue per thread removes much of the overhead but if the work is not evenly distributed between threads we might get bottlenecks from waiting for certain threads. A single queue will give us more overhead but

minimize the risk of the workload not being balanced.

2.3.4 Work Requesting

There is a third algorithm, work requesting. Then we have requests sent between workers that the other works grant by sending tasks. This algorithm works in such a way that when a worker requires more work, it contacts another worker and asks for more work. The other then replies, if it has time it either gives work or refuses. The drawback is that if the other worker is busy, it can not answer until it is ready. It would be ready each time it finishes a task and gets to find its next item in queue. However, this removes some of the overhead from work stealing, as that requires a bit of synchronization between both threads when a job has been stolen.

The more processors you have the more checking you need to do as well which means that key time that could be spent on actual calculations is being spent on looking for more work. As we get more and more processors on machines, especially with regards to the Intel Xeon Phi this effect becomes more visible. This will in the end result with mass migration to even out the work load. This means that we might get a higher workload just keeping the workload up, rather than just doing the work needed.

2.3.5 Work Scheduling Algorithms on the Xeon Phi

In general, there are three different work scheduling algorithms on the Xeon Phi with OpenMP and one for Cilk that should be considered. These are static, dynamic and guided for OpenMP [7] and work stealing for Cilk. The biggest difference can be found between static and both dynamic and guided. The difference between the scheduling methods in OpenMP on the surface seems to be only the dynamic allocation of work versus static allocation [12], but this can have big effect on various execution times along with the smaller differences between them. All of them divide the work between threads before the threads start executing, but the difference being that static divides all the work while both guided and dynamic might not, keeping the remaining work in a queue giving it out when there are available workers. This obviously means that the overhead for dynamic scheduling is higher but this can often work better with imbalance in load between iterations. It is also very important to consider the effect each scheduling method has on memory accesses in regards to the size of chunks. A chunk that is too small can sometimes cause memory access that cause problems like cache misses, as we discussed around the Xeon Phi, or problem with too big chunks that can among other things leave some workers without work.

Static Scheduling

Static scheduling divides the work into evenly sized chunks or as equal as possible between the threads running. This means that during execution time, each thread already has a predefined part of the work needing to be done. It is important that this is done beforehand and therefore very suitable for easily dividable tasks. When tasks get more dynamic or irregular, it gets harder to determine the actual load per iteration and as such load imbalance is more likely.

Dynamic Scheduling

Dynamic scheduling divides the work up into chunks with size equal to each iteration. The chunk size can, however, be controlled and increased if needed. By default the chunk size is one, meaning each thread gets one iteration of the work, until finishing before grabbing the next one. All chunks are uniform at all times, so the only work the scheduler gets after the initial setup is handing out each task. This means large overhead compared to other scheduling methods as there is constant communication between workers and the scheduler. It does mean, nonetheless, that during very unbalanced workloads that no worker should expect to get multiple high workload tasks limiting the chance of single threads causing waiting and bottlenecks.

Guided Scheduling

Guided scheduling is a method that starts with very large chunks that then get smaller as works continues. This chunk size resembles static scheduling while still being dynamic. By default, the chunk size is approximately the task being evenly distributed between threads. If we consider each task a loop the chunk is usually close to the size of dividing each iteration in the loop between threads. This can be beneficial as a point between dynamic and static. Seeing how we start with very big chunks at the start, we resemble static a lot while still maintaining a dynamic way of dividing work. Initially, we hand out much bigger chunks than dynamic which cuts down on the overhead while still offering some flexibility to the scheduler.

Cilk Work Stealing

Cilk uses work stealing to optimize load between threads. This means threads will actively look to find extra work upon completing their work. The work stealing is based off the work by Blumofe and Leiserson in their article called Scheduling Multithreaded Computations by Work Stealing

from 1999 [8]. That paper puts forth the idea of that the scheduling algorithm must be able to keep all running threads occupied during execution time, even if you consider unbalanced programs. The best way to do so according to the article is to let the threads themselves grab extra work from other threads when in need. The algorithm is a randomized work stealing algorithm for fully strict multithreaded computations. By fully strict they mean that all join edges from a thread go to the thread's parent. Figure 2.7 shows an example of threads stealing in Cilk.

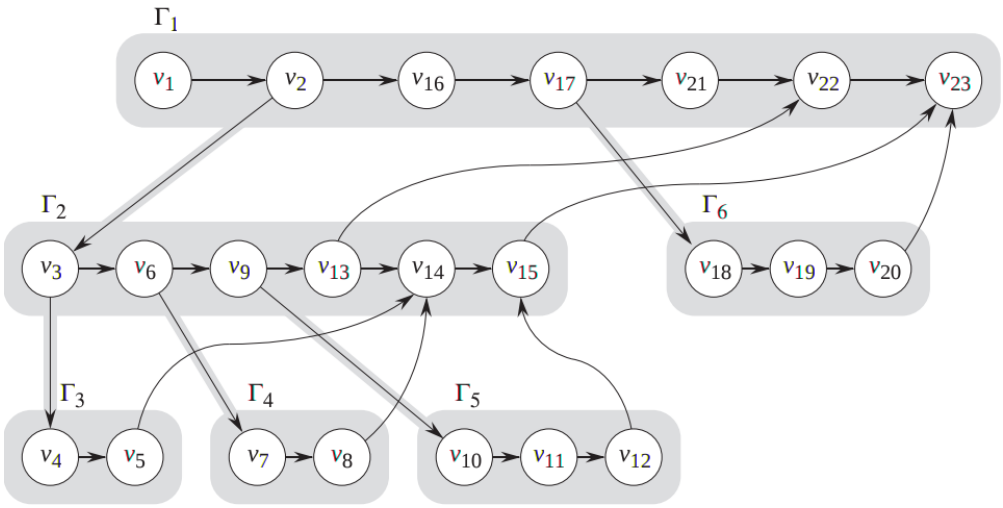


Figure 2.7: Example of work stealing Cilk threads

This might, however, mean that with the fairly evenly balanced code the overhead from the cross communication between threads trying to find work causes a huge increase in execution time. As work stealing has potentially more overhead than dynamic work sharing, it would not be surprising that work stealing does not work as well as work sharing for our project.

2.3.6 CUDA Work Scheduling

The scheduling in CUDA works a bit differently compared to the Xeon Phi. This is because CUDA works in so called warps. A warp is a set of threads that works in unison, see figure 2.4. Each time you start something in CUDA, the scheduler will start up the required streaming multiprocessors (SM), see figure 2.8. The figure shows how the scheduler is setup up within each core, where dispatcher can instruct different warps. Each of these can control 4 warp schedulers each which has two warps of 32 threads each through the instruction dispatch unit, see figure 2.9. As we can see the figure is very like figure 2.4 but has a wider view with the scheduler parts visible. The scheduler then reschedules a SM each time it blocks,

making sure that each SM is active. So, the CUDA scheduler then does the scheduling on a SM level instead of a per thread level.

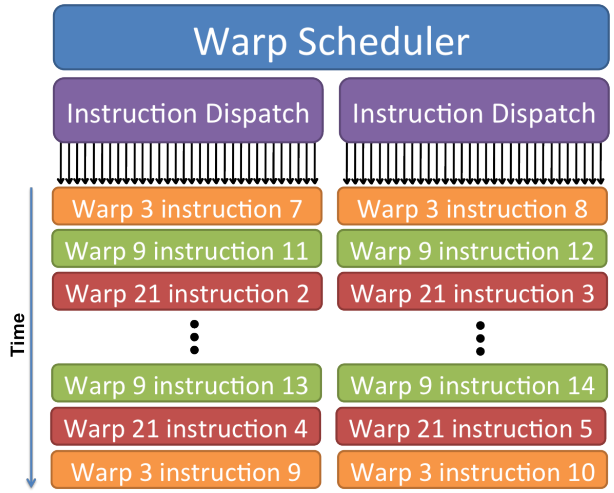


Figure 2.8: A streaming multiprocessor in CUDA

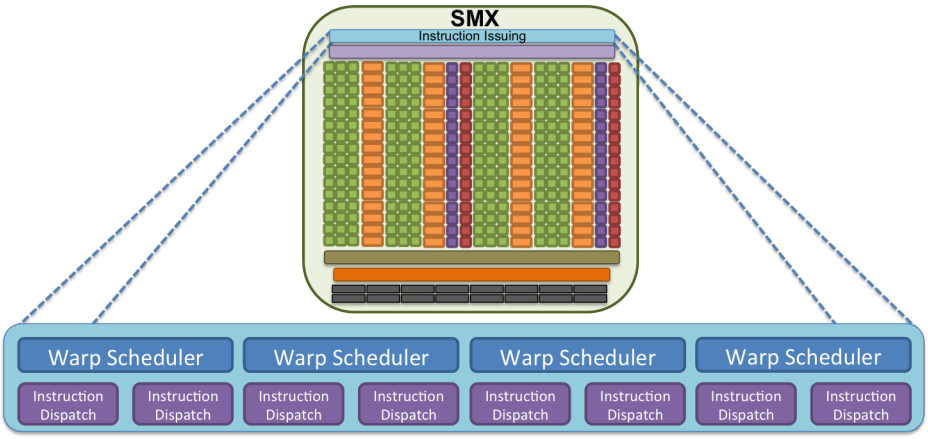


Figure 2.9: A warp scheduler

This means that we got one warp per core, as each core has up to 32 ALU's. This is also why we must consider that each execution has a multiple of 32 as its thread setup, otherwise we are not utilizing each core to its fullest as it will have some threads not doing any work. It also needs to be taken into consideration that the amount of overhead each core has is very low so that things like branching is very hard to schedule with the way the scheduler works. This means that we want all threads in a warp to execute the same code, even if some of the values might differ.

This adds to the idea that CUDA does better with streamlined and aligned calculations while suffering more from imbalance than the Xeon Phi. This, however, should give a better result on fairly structured and streamlined programs.

2.4 Summary

In this chapter the hardware used in the research has been discussed. The hardware includes a detailed discussion on the Intel Manycore architecture, the Xeon Phi, Kepler GK110 and the Geforce Titan. We discussed how it works along with the scheduling algorithms they use. In addition, a bit of the history around both the hardware and the scheduling algorithms were covered. This included parts of what to keep in mind when optimizing code for this hardware.

The most important things we have learnt from this chapter are that while the languages used on the Xeon Phi are meant to be easy to use we still must take care implementing. We can still get parallelization problems with incorrect memory use. We also spent time understanding both work scheduling in general and work scheduling algorithms we will use in our research. This chapter thus serves an important role of being the basis for how to utilize what we have to solve the problem we are trying to with this thesis and makes it easier to plan on how to solve our problem statements.

In the next chapter, we will discuss video processing, with focus on MJPEG and Codec63. The next chapter also covers H.264 a bit as it relates to Codec63.

Chapter 3

Video Processing

In this chapter a discussion will be given on how video encoding works. Understanding video processing gives knowledge about where and what can be optimized. To do this we will detail how Motion JPEG (MJPEG) works, especially how the JPEG part works inside MJPEG. We also mention H.264 as our Codec63 encoder uses motion estimation which MJPEG does not use but H.264 uses.

Like the previous chapter, this helps us understand what we need to do to be able to solve our problem statement with using different hardware for improved video encoding. As mentioned before, video encoding is becoming a bigger part of our lives. More and more of our use on internet depends on multimedia, with everything from standard images to graphical interactive websites to videos. While it is possible to use raw image and video data when transferring data between computers over internet, minimizing the size of the data helps us utilize our network better. Video encoding does exactly that by compressing the data. Compressing and decompressing data costs CPU power so we also must make sure that both our hardware and encoding software utilizes the power available well. This chapter will help to understand one way of encoding which we can take forward into our design and implementation.

3.1 MJPEG

MJPEG is a video compression format where each frame is compressed separately as a Joint Photographic Experts Group (JPEG) image. MJPEG is a intraframe only compression scheme and not a intraframe prediction scheme like our Codec63 encoder. Intraframe would be compression that is only done on a per frame basis and not between frames. This limits the compression rate of the MJPEG compared to other intraframe prediction schemas but is also easier on both the CPU and the memory. It also means

that MJPEG is insensitive to motion complexity and it does not get helped by static videos or hindered by highly random motion.

3.1.1 JPEG

As stated, the MJPEG just compresses each frame as an JPEG image. The JPEG group was founded in 1986 by CITT and ISO Standards organizations to create a standard for video compression [47]. The standard was finished in 1992 with some modifications later on. In short, the standard was to be a fully covering standard where each encoder and decoder had all possible options to select from. This included things like color aspects and JPEG variations. Implementing this, however, was very complex so people just opted for a single format from the JPEG standard. They selected JFIF so when we talk about JPEG today we really mean JPEG File Interchange Format (JFIF). There are other formats within the JPEG standard used today like Exchangeable Image File (EXIF), but JFIF is the most common one [33].

JPEG utilizies how humans detect things through their eyes, both things detected easily and things that are harder to detect. Because of how the human eye works, we can more easily detect difference in brightness than color. That is because we got more brightness sensitive receptors than color receptors [35]. Low frequency changes human eyes detect but we start having issues with higher frequency changes or with chrominance and the JPEG uses this to its advantage. This means that the best option we have is to transform the picture from the usual red, green and blue color model (RGB), which is the color model we detect things in, into the YCbCr model [40]. The Y is the greyscale, or the brightness of the picture, while the Cb and Cr is chrominance, blue and red respectively. So as we know how the eyes work we can safely assume that reducing the size of the chroma parts will not change how we perceive the image. In most cases we reduce the chroma part by halving the horizontal and the vertical aspects of the chroma part of the image. This then gives us one fourth of the size on both the Cb and Cr. We can then start the encoding where we work on each part separately, although all are done in a very similar manner.

The work starts by dividing the image into macroblocks [32]. These macroblocks are 8x8 pixels in size. First, we get a frequency chart of the image. After this, we move the range of the frequency to have a midpoint around zero instead of a pure positive range. The usual range we get is from zero to 255 but subtracting 128 from it gives us a range of -128 to 127. This gives us the ability to apply a Discrete Cosine Transformation (DCT) to our frequencies. The formulas can be seen in figures 3.2 and 3.3. The top left of this frequency map as can be seen in figure 3.1 is the overall hue of the image, while the other 63 values are the alternating components of the image. That is certain patterns in brightness levels. The application of the DCT means we get how often a certain wave setup shows up in the

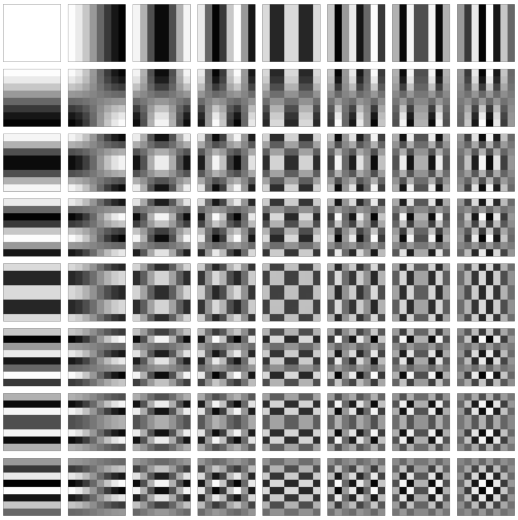


Figure 3.1: 2D DCT Cosine Wave Table [5]

$$X_k = \frac{1}{2}(x_0 + (-1)^k x_{N-1}) + \sum_{n=1}^{N-2} x_n \cos\left[\frac{\pi}{N-1} nk\right] \qquad k = 0, \ldots, N-1.$$

Figure 3.2: 1D DCT formula [49]

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right] \qquad k = 0, \ldots, N-1.$$

Figure 3.3: 2D DCT formula [49]

given block. By applying a coefficient to them we can then quantize these values. The quantization matrix is a table with a spread going from higher values in the top left to zero in the bottom right. This gives us more of the top left type of waves and less of the bottom right ones. This is possible because the human eye is good at detecting brightness at a large area but not the exact difference in brightness. So the higher the compression ratio, the more zeros we will have in this quantized table. We can then take this table and create a sequence by gathering the numbers in a zigzag pattern. We do so by starting at index 0,0 then moving to 0,1 followed by 1,0 before going to 2,0, then 1,1 and then 0,2 and so on, until we reach 8,8. This way we cover the top left corner first before moving towards the bottom right. Our resulting sequence then has a few numbers in the start and ending in a long row of zeros in the end. The reason we got most of our numbers in the start and a lot of zero in the end is because of the quantization and the coefficient meaning we aim for more of those numbers as the numbers towards the bottom right are higher frequency ones. We can then use Huffman encoding to write this to a file, which will compress a lot of data because of all the zeros.

This happens by generating a Huffman tree from these numbers. With the frequency of zero being so high it will occupy the highest slot in the list. Most our checks will therefore go towards the zero, which can easily be written as a single bit. We can decode our decompressed image by reversing the steps back. We extract the data from our Huffman encoded file, before dequantizing to get back to the pre quantized state. This will then show us the image as it was before the compression.

JPEG uses therefore a lossy compression. Other formats, such as Portable Network Graphics (PNG), use lossless compression. That means that when you compress it and then uncompress it, you do not lose any data and it exactly the same as when you compressed it. JPEG, however, loses a bit of the data due to the quantization but will still be very close to the original. The advantage of it is that you get a lot more compression from it. The biggest negative effects we get from the compression is the frequency between pixels. That is, most of the high end frequency we seen in our cosine table gets a bit lost. This means that high contrast areas get more affected. As we also scale the chroma part of the image we lose a bit of color. This can be specially noticed as part of the idea with JPEG is to take advantage of how well our eyes detect color so small changes are not noticeable.

3.2 H.264 and Motion Estimation

The H.264 codec is also known as MPEG-4 making it part of the Motion Picture Experts Group (MPEG) family. The main goal of H.264 was to provide a standard that could provide a better bitrate than previous standards while not increasing the complexity of it [48]. The H.264 works much like MJPEG but has added inter and intra frame prediction.

The frame prediction is the important difference and what Codec63 uses from H.264. Motion estimation works with the idea that a video will contain multiple images where there is little change between frames or in cases where a block of pixels might have moved between frames. As we have already encoded these in the previous frame we can reuse that encoding in our new frame. To do so we use a search for similar pixel blocks between the frames.

H.264 has more complex options available for motion estimation than Codec63. This includes more advanced search patterns for the best match and varying sized macroblocks. The motion estimation Codec63 uses from H.264 works by considering the location of the macroblock we are looking for in the current frame and using that location as the center of the search in the previous frame. We then search all pixels within a predefined search. With each 8x8 pixel block we calculate the differences between the blocks. To find the difference we use an equation called the sum of absolute

differences. We see the annotation for it in image 3.4.

$$\sum_{(i,j)\in W} |I_1(i,j) - I_2(x+i, y+j)|$$

Figure 3.4: Equation for sum of absolute differences [2]

This sums up the absolute values in difference in all pixels in the same block. The closer this value is to zero the more alike the two blocks are. As we only use absolute values the value is never below zero and we remove the chance that two of the blocks get a similar value by having big negative and positive differences. When we have searched all the possible blocks within the search range we find the lowest value, which is also closest to zero, and assume that this is where that block in the previous frame moved to.

This makes it able to compress the image a lot more but increases both complexity and time it takes to encode a video. However, the prediction in H.264 is more complicated than in our Codec63 encoder, which we will explain very briefly. First of all the macroblocks in inter frame prediction it uses are varying in size, from 16x16 to 4x4 pixels. This means we can use fewer bits on areas which stay relatively the same between frames with larger blocks while still keeping the detail in areas with high energy. This is known as tree structured motion compensation [43]. The intra frame prediction also uses varying size on the macroblocks with up to nine predictions modes for each 4x4 luma pixel blocks and four for 16x16 pixel blocks. All of these are optional. Then we have a single mode that is always applied to all 4x4 chroma pixel blocks. These modes depend on the motion vector. H.264 also uses other algorithms like context adaptive variable length coding and context adaptive binary arithmetic coding. Both of which we will not detail here.

This was as stated a very brief description of H.264. As most of our encoding are based on MJPEG and Codec63 a more detailed explanation of H.264 is outside the scope of this thesis.

3.3 Summary

We have now explained video encoding works with regards to the encoder we plan to use. This is vital to understanding how we can improve our Codec63 encoder which is what our next chapter will do. We have gone through exactly how JPEG encodes and compresses images through DCT. We have also mentioned how H.264 works which Codec63 borrows some methods from. This makes us able to estimate where

potential improvements can be made so we can both test our hardware and understand the results we get.

What we learnt was that JPEG aims at using the strength and weaknesses of how our eyes perceive images. To do so it uses the DCT algorithm that allows us to quantize parts of the image. This lets us then compress the image through Huffman encoding. We also learnt that there is a slight loss of data during compression.

The next chapter will look at how our Codec63 encoder works and design what optimizations we are aiming for to be able to run it in parallel.

Chapter 4

Designing Optimizations for Codec63

In this chapter there will be a discussion on how our video encoder works and how it differs from standard practices. The encoder is based on Motion JPEG (MJPEG) but uses inter-frame prediction. This means it resembles H.264 which we described earlier. We also go through how we plan on implementing the research, what we need to look at and why it is important.

To be able to optimize our Codec63 encoder understanding the code itself and how it runs is vital. As we mentioned in our problem statement to be able to test our hardware we must parallelize our code. To be able to optimize and create fair tests for CUDA, OpenMP and Cilk we must first understand the design choices in regards to Codec63. To be able to get good results the optimization has to be smart and we need to know why we are optimizing what we are optimizing. To do so profiling the execution of Codec63 is important to understand what design choices should be made.

4.1 Codec63

The baseline code used will be the single-threaded Codec63 [39]. The codec is available as an open source project on Bitbucket [17] and instructions on using it can be found in appendix C. The code in the Codec63 is based on the MJPEG encoding algorithm with added frame intra-prediction. This means that it adds both motion estimation and motion compensation. It also has an more efficient Discrete Cosine Transformation that runs in 2D instead of 1D. As such it is far from optimal for actual encoding use, but is an excellent tool for studying and teaching. This is because it does all the steps required for encoding very clearly and visibly. See figure 4.1.

It must be considered as well that it only uses motion vectors between

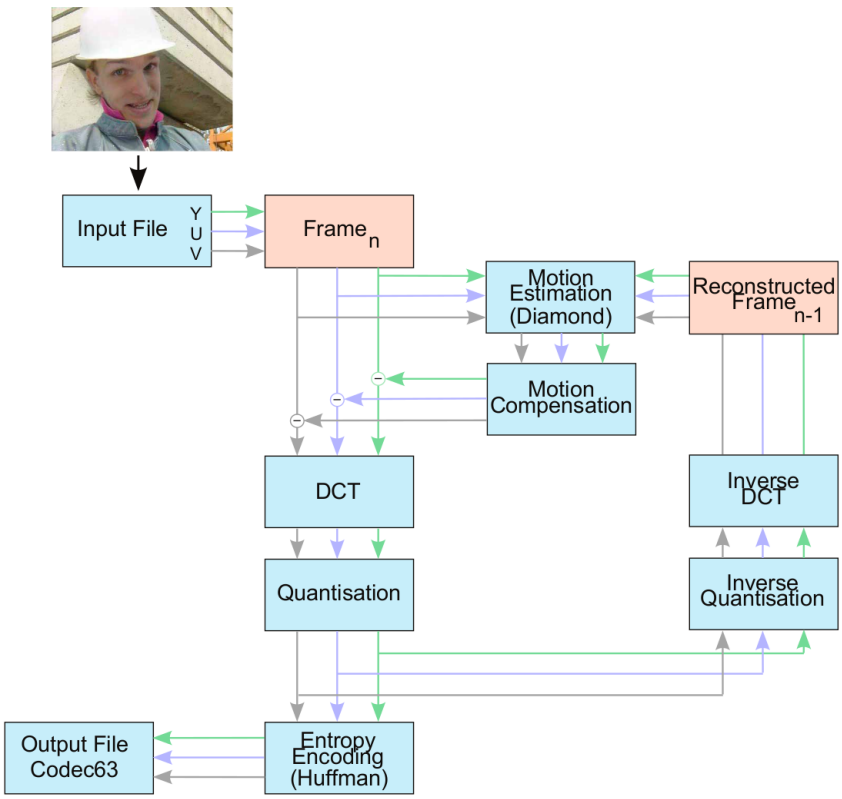


Figure 4.1: Codec 63 [42]

frames in the encoding it does. The motion vectors is the estimated movement of a object in the frame between frames. These are often referred to as P-frames in encoding discussions and in our case our P-frames only consider the frame just before the current one. It does not consider so called B-frames or bidirectionally-predictive-coded frames. B-frames are frames that can consider any other frame in the video.

4.1.1 Inter-frame Prediction

The inter-frame prediction is the main bottle neck of our Codec63 encoder. What we do is split the frame into macroblocks that are 8x8 pixels in size, much like in the JPEG example, and then find the frame that is the most like that frame within a certain search range. By finding a macroblock that is almost the same we can skip saving a lot of data. This will compress the image as a result a lot more. Over time, however, the frames will degrade so we need to reset the frame we compare to, the so called keyframe. The key part of the inter-frame prediction is the motion estimation, where an attempt was made to find a motion vector. This vector points to where the macroblock can be found closest to the current macroblock. The motion vector can then be used to predict our predicted frame. This is done with

motion compensation. This way there is compensation for the movements within the frame and it is not required to write each frame fully as most of the data is stored already. All of this is possible because in most cases the change between frames in a video is very minimal. The frame prediction in Codec63 has a very simple setup. What we do is we take the previous frame and encode it through DCT. The frame is then stored and after that compared to the current frame. We do so by splitting it into macroblocks, which are 8x8 blocks of pixels in the frame. We then compare these macroblocks to areas from the previous frame. This way we try to find a new macroblock that looks exactly like this one. The closer we get to zero the closer the macroblock resembles the macroblock in the previous frame. This we can then use to compress the between frames as well.

4.1.2 The Flow of Codec63

To explain the codec a bit better we must consider the flow inside it, see figure 4.1. We start initialize the standard details like height, width and the input and output file. Then we set up the data areas required, all inside a struct we call c63_common. This struct will store all the data required while running, see code example 4.1.

Code Example 4.1: common_C63 struct

```
1  int width, height;
2  int ypw, yph, upw, uph, vpw, vph;
3  int padw[COLOR_COMPONENTS], padh[COLOR_COMPONENTS];
4  int mb_cols, mb_rows; //Amount of columns and rows
5
6  uint8_t qp; // Quality parameter
7  int me_search_range; //Motion estimation range
8  uint8_t quanttbl[COLOR_COMPONENTS][64]; //Quantization
9
10 struct frame *refframe; //The previous encoded frame
11 struct frame *curframe; //The frame being encoded
12
13 int framenum;
14 int keyframe_interval;
15 int frames_since_keyframe;
16
17 struct entropy_ctx e_ctx; //Output file data
```

As seen the struct contains all the data that would be needed. Specially important is to note the frames, both the reference frame and the current frame. The frame struct can be seen in code example 4.2

Code Example 4.2: A frame struct

```
1  yuv_t *orig; // Original input image
2  yuv_t *recons; // Reconstructed image
3  yuv_t *predicted; // Predicted frame
4                    // from intra-prediction
5
6  dct_t *residuals; // Difference between original
7                    // image and predicted frame
8
9  struct macroblock *mbs[COLOR_COMPONENTS];
10 int keyframe;
```

The frame has three normal image structs, the original frame, the reconstructed frame from dequantizing the quantized image and the predicated frame from the motion compensation. The residuals are the results from the quantization and are stored in a dct_t struct, which is just three int16_t values. We then have the macroblocks. The macroblocks contain the motion vectors for each 8x8 pixel block.

Each frame then gets read into yuv_t struct. This struct only contains three uint8_t pointer, one for each spectrum in the image. After this the encoding of the image can start.

Keyframes are important things to consider. Each keyframe resets our motion estimation. This is important as each time we refer to the previous frame the data gets a bit less accurate. There might be subtle changes happening that get lost in the approximation of motion estimation. Therefore, there is a need to reset the original reference frame at set intervals. The keyframes are used to reset the original reference frame. It should then be obvious that the first frame is a keyframe and does not do any motion estimation or compensation.

So we go straight to the DCT. The DCT then splits the frame into eight pixel tall rows before splitting each row into eight pixel wide blocks. This gives us 8x8 pixel blocks required for our encoding as covered earlier. We then multiply the raw image with the dctlookup table which is our quantization matrix. We transpose the result, before doing the same again with multiplying and again we transpose. We then scale the block before doing the actual quantization. As soon as we complete the quantization we dequantize the frame. The reason being that to be able to estimate the motion in the image we need to do so with our encoded image. This encoded image, therefore, has to be decoded. Decoding the image happens exactly the same was as the encoding except in reverse order.

We then write the frame. When the frame is written we start with putting two bytes at the start to denote the start of a frame. We then define the Quantization tables followed by some important header information like

size and keyframe details. At this point, we write the Huffman tables. We then write the start of the scan followed by the interleaved image. The interleaved image is the residuals from the DCT and the macroblocks from the motion estimation. On keyframes the macroblocks are all zero. After having written the interleaved data we write two bytes marking the end of the frame.

This brings us to the second frame. The only time the second frame is a keyframe is when the keyframe interval is one. In our case, therefore, the second frame is not a keyframe and we do motion estimation and compensation on it. When we have read the image in we start by doing the motion estimation. The motion estimation happens by splitting the image again into 8x8 pixel blocks. It then uses the search range to find the frame that is most like it in the previous frame. The search range is by default 16 so the Y part of the image search 16 pixels in each direction, up, down, left and right while the U and V parts are scaled so the search range there is 8. It finds the closest frame by calculating the lowest sum of absolute differences. After that we compensate the estimation straight away as we use this in the quantization on the same frame. The compensation is just values from the closest previous frame. We then do the same again with the quantization and the dequantization, write it to the output file and carry on to frame 3.

4.2 Profiling Codec63

So we must first understand where most of the execution time for the Codec63 lies. The best way to do so would be to profile it. We will use Gnu's profiling tool (gprof) to do so. In table 4.1 and table 4.2 we can see the results from basic profiling on foreman.yuv, the 352x288 video and on foreman_4k.yuv, the 3840x2160 video. On the smaller video we do 100 frames but the larger one we opt for just 10 frames due to the amount of time it takes encoding. It is important to note that the method sad_block_8x8 is a submethod for motion estimation. As we can see in the tables more than 90% of the execution time happens inside the motion estimation while we get minor parts of the execution time happening in the quantization and dequantization methods. With this knowledge we know that most of the results will come with improving motion estimation. As listed in chapter 4.1.1 the motion estimation includes a search range that does a calculation on the sum of absolute differences between two 8x8 blocks.

This then becomes the main target for our optimization and where we will focus most of our tests on. Understanding the bottleneck in motion estimation lies with knowing how the code searches. For every macroblock, that is every 8x8 pixel block, we will look in a 32x32 grid with our macroblock as the center for the best approximation to that block. This

Foreman		
% execution time	**Seconds**	**Method**
90.26	9.25	sad_block_8x8
4.49	0.46	c63_motion_estimate
1.46	0.15	dequant_idct_block_8x8
1.46	0.15	dct_quant_block_8x8
0.1	0.01	dequantize_idct
0.1	0.01	c63_motion_compensate
0	0	dct_quantize

Table 4.1: Profiling of the Foreman video

Foreman_4k		
% execution time	**Seconds**	**Method**
91.21	77.57	sad_block_8x8
4.05	3.44	c63_motion_estimate
1.67	1.42	dequant_idct_block_8x8
1.28	1.09	dct_quant_block_8x8
0.21	0.18	c63_motion_compensate
0.09	0.08	dequantize_idct
0.07	0.06	dct_quantize

Table 4.2: Profiling of the Foreman_4k video

becomes a very large search area as the best approximation is found by the difference in the sum of absolute values between the blocks. So these areas can be very spread out while requiring large amounts of lookups and summing to get results. As such we can easily optimize these areas by running them in parallel as they do not overlap per macroblock being searched with. However, if we have multiple macroblocks running through the search at the same time the size of the search range almost guarantees we get overlap. So it is very important that the effictivization of the motion estimate considers this. It must also be taken into consideration that the overlap inside the comparison between two macroblocks is fairly high as they contain about the same memory areas. So care must be taken there.

4.3 What to Improve

With this in mind we can start planning on what parts of the code we would want to see improved. We will start with the most obvious areas before moving into smaller areas and discussing how we plan to offload.

4.3.1 Motion Estimation

Starting with motion estimation we got a double loop at the start which finds our macroblock. We then do a double loop to search for other macroblocks and end up with a double loop to calculate the sum of absolute differences. We can see the three in code example 4.3.

Code Example 4.3: Loops in Codec63

```
1  //Outermost loops:
2  for (mb_y = 0; mb_y < cm->mb_rows; ++mb_y)
3  {
4    for (mb_x = 0; mb_x < cm->mb_cols; ++mb_x)
5    {
6      me_block_8x8(cm, mb_x, mb_y, cm->curframe->orig->Y,
7          cm->reframe->recons->Y, Y_COMPONENT);
8    }
9  }
10
11 //Middle loops:
12 int best_sad = INT_MAX;
13
14 for (y = top; y < bottom; ++y)
15 {
16   for (x = left; x < right; ++x)
17   {
18     int sad;
19     sad_block_8x8(orig + my*w+mx,
20                   ref + y*w+x,
21                   w,
22                   &sad);
23
24     if (sad < best_sad)
25     {
26       mb->mv_x = x - mx;
27       mb->mv_y = y - my;
28       best_sad = sad;
29     }
30   }
31 }
32
33 //Innermost loops:
34 for (v = 0; v < 8; ++v)
35 {
36   for (u = 0; u < 8; ++u)
37   {
38     *result += abs(block2[v*stride+u] -
39                    block1[v*stride+u]);
40   }
41 }
```

Consequently, the obvious part is to improve the first and outermost loops.

This double loop runs the rows first with each row being 8 pixels high while the inner runs through the columns where each is 8 pixel wide. This gives us macroblocks. The easiest way to improve this, as there is not much else than calling a method inside them, is to make each thread grab an index in the looping, or if there are more indices than threads, hand them out in as an even spread as possible. This will apply to all three languages extensions we use.

The double loop inside the outermost loops can also be parallelized. However, these loops depends on a few shared values. The content on the macroblock and the best sum of absolute differences we have found. If we opt to just straight up parallelize the loop we will probably have memory conflicts that affect the execution time. We do have options of using locks or atomic operations on the shared values but this probably would not improve things much. So the idea is that each thread stores the values it has found, before running some kind of reduction to find the best value. We then use a single thread to update the values in the macroblock.

The innermost loops are probably best left alone for the Xeon Phi while they might be improved for CUDA. On the Xeon Phi we will have problems with how close each lookup call will be, causing unnecessary bank conflicts. In CUDA, however, as each thread is so lightweight and that every 32 threads work together as a whole, they should still do fine. We do need to consider though the difference between CUDA and Xeon Phi so we do not make unfair challenges for the Xeon Phi with the code structure we have. We will, nonetheless, test by optimizing the 3 loops and then being selective on the loops we optimize.

4.3.2 Quantization and Dequantization

Quantization and dequantization both follow closely the path of motion estimation when finding the 8x8 blocks. The real difference lies within each 8x8 block.

Considering the Xeon Phi, however, we know that it does not do to well with smaller memory areas and using multiple threads. Seeing how the code in both the quantization and dequantization is very close to each other memory wise, for example we do a transpose on our 8x8 block during DCT calculations where everything happens within the 8x8 pixel block. All our DCT calculations are like that as well containing only calculations from within the 8x8 pixel block and potentially other tables. The code, however, does have the usual setup where we loop each macroblocks multiple times, or in the case of the base code a total of seven times. This means that we do have options of optimizing these loops. Still, as all the memory overlaps we would be better of just to optimize the code by removing some of the loops and doing parts of it at the same time. This, however, does not have anything to do with the optimizations we are trying to do on the

Xeon Phi and we will consider the integral workings of the quantization and dequantization outside the scope of the research.

4.3.3 Motion Compensation

Motion compensation has a very low execution time so expectations is that any alterations here will not change much. It uses the same structure as motion estimation, dequantization and quantization. That is it goes through each frame height and lengthwise finding 8x8 macroblocks. It then uses the calculated motion vector to find the approximate location of the same macroblock in the previous frame, the so called reference frame. This then becomes the predicated frame which is used by the quantization and dequantization.

We still have a fairly large search range and this is mostly just reading values between frames. By doing this we should be able to run the motion compensation in parallel. However, seeing how little execution time it requires we should not expect a big change from it. The optimization here would probably be better as soon as we find the motion vector, removing the need for looping the same code again. Still, these are again optimizations that are not directly related to the research.

4.3.4 Offloading

The key choices that may be taken into consideration are what and when to offload. We can for example offload everything in the common_c63 struct if we want to and move it all back out again, but seeing how we do not use all the data on both sides it might not be the smartest thing to do. So we will focus on getting key values like height and width stored on both the host and the coprocessor and then moving the raw image over to the coprocessor while moving the residuals and macroblocks back out. This would be the only data we require to be shared so it would be the smart way to offload. We need to consider though that if we test offloading parts, we might not want to transfer the same data but the same idea applies.

There are also certain points where we might consider that not all the calculations should be offloaded. For example, the Xeon Phi suffers a bit from memory access with areas that are to close to each other. So we should consider just offloading the motion estimation and doing the other three parts natively. However, this might mean that the cost of offloading is to high as we still need to transfer the same amount of data. This might mean we actually can not get any real advantage.

To make sure that the parallel code is working, we should start by only creating a parallel code that does not utilize the Xeon Phi in any way. This

will make it easier to test that the actual parallel implementation works as intended without potential side effects caused by data transfers.

4.4 Related Applications

Something that also has to be considered are related applications of our algorithms. This regards both the algorithms themselves and other software using them. We have discussed how the algorithms our encoder uses and also stated that it is fairly naive. We should therefore consider the actual algorithms and how they are used and other software that might benefit from the research.

The motion search in our encoder is in short a simple grid search. As a grid search is in most cases a lookup of all possible values within a grid of some type there are multiple potential applications for it. While there are variations on how a grid search is done they all share the same basic idea. These are used in multiple video encoders and in statistical analysis of DNA among other things. So effective parallelization of a grid search, like our motion estimation, can have widespread effect. Testing how our hardware works with an optimized grid search can hint towards the potential of the hardware in other fields of study as well.

The DCT algorithm gets used in all types of multimedia compression, among other video encoding and audio encoding, and also in applied mathematics for spectral methods. As both can be time extensive in execution optimizing them can also have a widespread effect.

A related field of research is computer vision. Computer vision deals with how computers gain understanding from images and videos at a high level. Our motion estimation relates directly to this as we are trying to identify similar parts of frames between frames. We also have other similar fields like neural network research. This makes our research also interesting for studying other related fields.

4.5 Summary

We have now detailed how our Codec63 encoder works along with making important choices for what we want to improve for it. This has been the first step in trying to answer our problem statement. We have found that the main bottleneck in Codec63 is the motion estimation. We have also identified potential loops that should be optimized. The most important part to optimize is the motion estimation. We have also discussed how to utilize the Xeon Phi.

We have detailed how our encoder implements the algorithms we discussed in the previous chapter. We have also learnt that while there are optimization opportunities in a number of places for Codec63, the expectation is that we will not notice much of the optimizations outside the motion estimation. The reason being that the motion estimation is just that big of a factor for the computation time.

We have also touched on how our research affects other areas. This is important as our encoder might not be very practical, it still contains algorithms and methodology that can be applied to other software and algorithms.

Our next chapter will implement what we have decided and discuss the results from our implementation.

Chapter 5

Parallelizing Video Processing for Xeon Phi and NVidia GPUs

In this chapter the implementation will be discussed along with any variations from the design. Most of the variations are due to problems we had during implementation but some came as a direct follow up to things we found during the implementation. This chapter will give us answers to our problems statements and discuss our results.

We will also go through our host computer setup and the software we are using. We also talk about issues we had with installing and running the software.

5.1 Setting up the Machine

The hardware used to run the experiments on will be a Dell T620. The computer has two Intel Xeon 2GHz E5-2650 CPU with 8 cores each giving us a total of 32 hyperthreaded threads, 8 DIMM DDR3 1600MHz memory chips, giving 64GiB memory and a GeForce GTX Titan GPU. The computer is part of the Media Performance Group (MPG) at Simula Research Laboratory and is meant for multimedia research. The computer is using CentOS 6.5 as the operating system with Gnu C Compiler (GCC) version 4.7.3, Intel C Compiler (ICC) version 16.0.3 and NVidia C Compiler (NVCC) version 7.5.17.

The computer also has a Xeon Phi card from the 3120 series [23]. The card has 57 cores each with a base frequency of 1.1 GHz a 28.5 MB L2 cache and uses 22nm lithography. The maximum memory size on the card is 6 GB with a bandwidth of 240 GB/s.

All the software setup has been done as parts of this thesis and has been maintained throughout the work. The software has been kept up to date as much as possible, as there are a few limits to updates for CentOS with regard to specific software.

As we are using numerous different types of hardware we needed to spend some time setting up software for it. The operating system had to be setup, along with compilers for all four languages, CUDA, OpenMP, Cilk and C. This was done with ICC, GCC and NVCC. We also needed drivers for both the GeForce Titan and the Xeon Phi along with profiling software.

5.1.1 Software for the Xeon Phi

As the Xeon Phi is relatively new there are a few things to consider before using it. The Xeon Phi has support for SUSE and Red Hat Enterprise when it comes to which systems it will run on. We are, however, running on a CentOS so to be able to run the Xeon Phi Intel has provided a script that allows rebuilding the package to fit with the kernel running on the OS. Without this the system will be missing much of its capabilities and probably be incapable of running. As CentOS is a variation of Red Hat, the difference between kernels is usually just a small build difference. Rebuilding and installing is fairly user friendly with the scripts given by Intel.

However, during our installation attempts we missed the rebuilding script. The installation documentation they have for download only contains details on installing the package directly, without any rebuilding. This meant that a number of both OS upgrades and other software upgrades were attempted to be able to install, before finding a separate document about rebuilding the package to fit the kernel. As soon as we found the script the software required to run the Intel Xeon Phi was installed properly. This still caused a few issues regarding login and user account creation as the system had issues moving SSH keys between the card and the OS. This was bypassed by using the root access to reset passwords and user accounts as needed. This was obviously a incomplete fix but we were unable to find the reason why normal way did not work and even after contacting Intel we were still unable to find out why.

The Xeon Phi also requires software from Intel to be useable. To compile runnable code it needs Intel's C++ compiler, also known as ICC. However, as with most Intel programs this is not free and will require licensing. The licenses are available though for students but only in a limited way and only as a one year license. So another issue we ran into was that our license expired during the work on this project. Renewing the license did not come without problems as Intel's suggested way of doing this would be to reinstall all programs and get a new license. Having done so, however, the license for some of the products from the Intel package still did not work. After

many attempts we did a full reinstall, which solved the licensing issue and updated all the software. This way we finally were able to test on as up to date software as possible for the Xeon Phi.

5.2 Programming the Encoder with OpenMP

With functioning software and hardware setup we started with basic ideas on how to parallelize the code we had for the Xeon Phi. We focused first on getting the code working with OpenMP before adapting it to the other language extensions. As discussed before in the design the first option was to focus on motion estimation. Our first steps were to add pragmas to the loops making them run in parallel. The first test we started with was parallelizing all the loops, from the outermost all the way to the innermost loop. This gave us the results in figure 5.1. Our Y axis represents seconds or the time it took to encode each video. The blue line, marked full parallel, shows the time it took when we had when using 32 threads and static scheduling for each of the four videos with all loops parallelized in motion estimation. The red line shows the time required to encode when we were more selective with what we chose to parallelize. We discuss what we chose in the next few paragraphs.

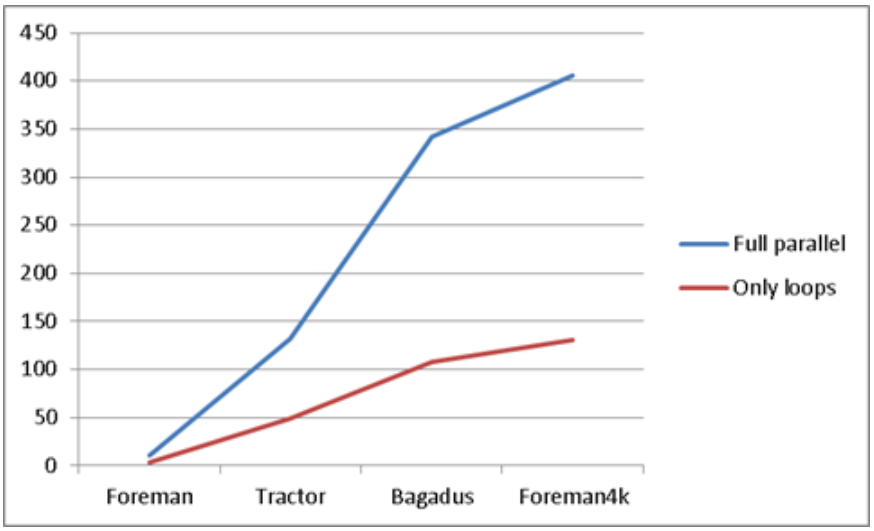


Figure 5.1: Full motion estimation parallelization vs loops only

The first issue we saw was that the sum of absolute differences calculation all relied on a single value being increased. This would either cause a race condition or a bottleneck in the code. So we changed it to calculate all answers into a separate array and then using reduction to find the sum. This caused a huge spike in the amount of time the program required. When profiling the run we could notice that it was due to thread synchronization. See table 5.1.

Function	Module	CPU Time
sad_block_8x8	c63enc	1468.505s
__kmp_fork_call	libiomp5.so	828.010s
__kmp_join_call	libiomp5.so	329.110s
__kmp_get_global_thread_id_reg	libiomp5.so	142.232s
[OpenMP fork]	libiomp5.so	80.999s

Table 5.1: Profiling full parallelization

So we reverted back to using just the outer loops to decrease the granularity, ignoring the loops inside the sum of absolute differences calculation. Considering the shared variable that we use to find the best result we also knew this would cause the same problem as in our sum of absolute difference calculation. So we ended up only using parallel operations between macroblocks and not in the actual search. This gave us improved results that were good enough to start looking at other parts. This is then the red line in figure 5.1. A table showing the results can be found in appendix A.2

We then proceeded to do tests on the motion compensation. As we discussed before the motion compensation is very simple giving us a very minimal result on execution time when using pragmas on all the loops. After this we went to the quantization and dequantization. We did as planned with the design and optimized the outside parts. This gave us a decent speedup which can be seen in figures 5.3, 5.4, 5.5 and 5.6. These figures also contain data from offloaded runs so they are discussed in more details later in this chapter.

So we moved towards testing optimizations inside the quantization and dequantization. There are a number of loops through the 8x8 pixel blocks there. However, as expected we found that these details increased the execution time. With this we had the basic outline for our parallel code in OpenMP which had optimized most of the outer loops.

5.2.1 Preparing to Offload with OpenMP

The main work then consisted of offloading the already parallel code we had. A limitation of offloading through OpenMP is that structs with pointers require some extra work. So our initial idea was to flatten out the common_c63 struct as it contained a lot of pointers to structs that again contained pointers. So we started by removing all the pointers in the internal structures to the common_c63 struct. The struct then required less work to be transferred. However, we still can not offload the struct directly and expect it to also offload the values pointed to by the pointers

inside the struct. So we created separate pointers, pointing to the same area as the pointers in the struct. We were then able to offload each value that the struct had to the Xeon Phi while also offloading the values pointed to by the pointers.

This would then move all the data both in and out from the Xeon Phi. As a result, we started with doing so in the motion estimation part. We moved all the values in and all the values out again. However, as discussed in our design there is no reason to offload all the data and then transfer it back. So we altered the code to initially allocate space for the data that was to be offloaded and then did the transfer afterwards. The initial setup has no actual calculations, just allocations through nocopy directives. The code we used can be seen in code example 5.1.

Code Example 5.1: How we offloaded data

```
 uint8_t *Yrecons = cm->ref_recons_Y;
  uint8_t *Urecons = cm->ref_recons_U;
  uint8_t *Vrecons = cm->ref_recons_V;
  uint8_t *Ypredicted = cm->curr_predicted_Y;
  uint8_t *Upredicted = cm->curr_predicted_U;
  uint8_t *Vpredicted = cm->curr_predicted_V;
  int16_t *Uresiduals = cm->curr_residuals_U;
  int16_t *Vresiduals = cm->curr_residuals_V;
  int16_t *Yresiduals = cm->curr_residuals_Y;
  struct macroblock *Ymbs = cm->curr_mbs_Y;
  struct macroblock *Umbs = cm->curr_mbs_U;
  struct macroblock *Vmbs = cm->curr_mbs_V;
#pragma offload_transfer target(mic:0) in(cm:length(1) ALLOC RETAIN) \
  nocopy(Yrecons:length(cm->ypw * cm->yph) ALLOC RETAIN) \
  nocopy(Urecons:length(cm->upw * cm->uph) ALLOC RETAIN) \
  nocopy(Vrecons:length(cm->vpw * cm->vph) ALLOC RETAIN) \
  nocopy(Ypredicted:length(cm->ypw * cm->yph) ALLOC RETAIN) \
  nocopy(Upredicted:length(cm->upw * cm->uph) ALLOC RETAIN) \
  nocopy(Vpredicted:length(cm->vpw * cm->vph) ALLOC RETAIN) \
  nocopy(Yresiduals:length(cm->ypw * cm->yph) ALLOC RETAIN) \
  nocopy(Uresiduals:length(cm->upw * cm->uph) ALLOC RETAIN) \
  nocopy(Vresiduals:length(cm->vpw * cm->vph) ALLOC RETAIN) \
  nocopy(Ymbs:length(cm->mb_rows * cm->mb_cols) ALLOC RETAIN) \
  nocopy(Umbs:length(cm->mb_rows/2 * cm->mb_cols/2) ALLOC RETAIN) \
  nocopy(Vmbs:length(cm->mb_rows/2 * cm->mb_cols/2) ALLOC RETAIN)
  {
  }

  //Read image and setup loop to run the below

#pragma offload_transfer target(mic:0) \
  nocopy(cm, Ypredicted, Upredicted, Vpredicted, Yrecons, Urecons, Vrecons) \
  in(image_Y:length(cm->vpw * cm->vph) ALLOC FREE) \
  in(image_U:length(cm->upw * cm->uph) ALLOC FREE) \
  in(image_V:length(cm->vpw * cm->vph) ALLOC FREE) \
  out(Yresiduals:length(cm->ypw * cm->yph) REUSE RETAIN) \
  out(Uresiduals:length(cm->upw * cm->uph) REUSE RETAIN) \
  out(Vresiduals:length(cm->vpw * cm->vph) REUSE RETAIN) \
  out(Ymbs:length(cm->mb_rows * cm->mb_cols) REUSE RETAIN) \
  out(Umbs:length(cm->mb_rows/2 * cm->mb_cols/2) REUSE RETAIN) \
  out(Vmbs:length(cm->mb_rows/2 * cm->mb_cols/2) REUSE RETAIN)
    {
      c63_encode_image(cm, image_Y, image_U, image_V);
    }
```

It should be noted that in the code example above we use allocation and not memset for each image pointer we read to. We found that the difference between reusing the data areas pointed to by image pointers with memset versus allocating new areas for each frame had no difference on runtime.

The nocopy lets us set up data areas before we start offloading and as we loop around the second offload clause we would have either had to do the first frame there or by just setting them up beforehand. Third option would have been to allocate all the time but that gives worse results. Especially complications would also happen with the areas being just moved out and areas that are not affected by transfers. This then let us move things as we needed between areas.

5.2.2 OpenMP Optimizations

A key thing to consider is how the motion estimate handles finding the best sum of absolute differences (SAD). For each result we get we compare it to a variable called best_sad and see if our newer result is better. While this is acceptable for a sequential execution, it does cause problems in a parallel one. We can see the code in code example 5.2.

Code Example 5.2: Original way of finding the best SAD

```
1  int best_sad = INT_MAX;
2
3  #pragma omp parallel for schedule(SCHEDULE)
4  for (y = top; y < bottom; ++y)
5  {
6    for (x = left; x < right; ++x)
7    {
8      int sad;
9      sad_block_8x8(orig + my*w+mx, ref + y*w+x, w, &sad);
10
11     if (sad < best_sad)
12     {
13       mb->mv_x = x - mx;
14       mb->mv_y = y - my;
15       best_sad = sad;
16     }
17   }
18 }
```

There had to be a better way of handling this. Instead of having a single variable holding the best value we stored all the SAD values we found in an array that we treated like a 2D array. We consider it a 2D array as the search range applies to a 2D space. We then used a parallel SIMD operation from OpenMP to find the index of the lowest value. We could then use this index to find the motion vector from our current macroblock. This then gave us the correct macroblock. This improved our execution time from before as we can see in figure 5.2. We can see the code for it in code example 5.3.

Figure 5.2 shows the time to encode 100 frames for each of our four test videos, with the y-axis showing time in seconds. All the runs are done on the

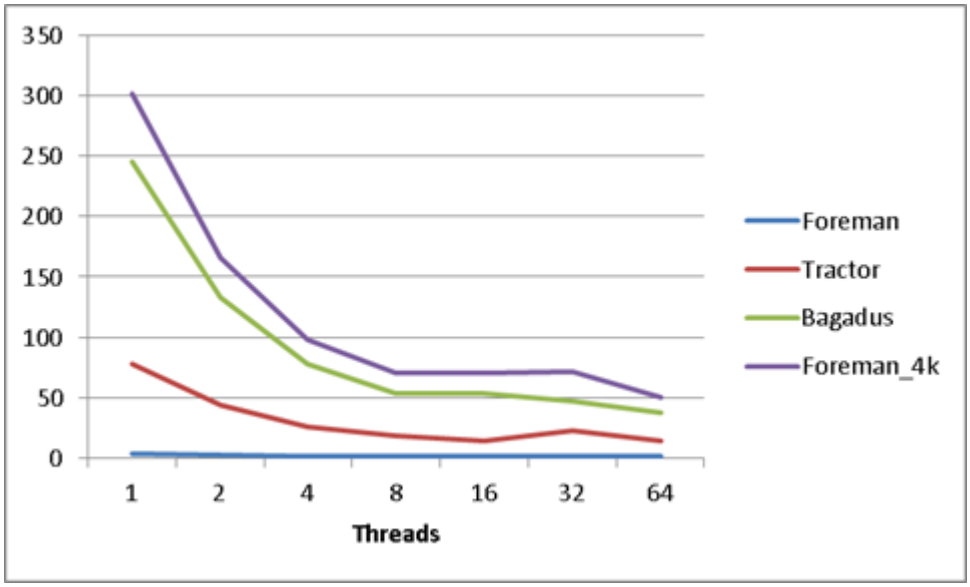


Figure 5.2: Motion estimation and optimized best SAD

host CPU only. The optimizations that have been done are parallelization of motion estimation loops and optimized comparison for finding the best matching block between frames in the motion estimation. We can see that our performance improves for each added thread but the most notable improvements come from going from one to eight threads. We also notice something interesting when using 32 threads. This will be examined better in section 5.5.2.

Code Example 5.3: SIMD operation to find index of lowest value

```
1  int sad[(bottom−top)∗(right−left)];
2  int best_sad = INT_MAX;
3  #pragma omp parallel for schedule(SCHEDULE)
4  for (y = top; y < bottom; ++y)
5  {
6    for (x = left; x < right; ++x)
7    {
8      sad_block_8x8(orig+my∗w+mx, ref+y∗w+x, w, &sad[(y−top)∗(right−left)+x−left]);
9    }
10 }
11 best_sad = __sec_reduce_min_ind(sad[0:(right−left)∗(bottom−top)]);
12 mb−>mv_x = (best_sad%(right−left)) − mx + left;
13 mb−>mv_y = (best_sad/(right−left)) − my + top;
```

This meant we had now also parallelized the only potential race condition calculation which was the biggest bottleneck left from the original adaptation. The work was completed by setting up multiple ways of running the code. As stated earlier we wanted to check the timings of both a encoder that had all the calculations offloaded and where we just offload the motion

estimation. We also wanted to record the time the encoding alone took, excluding the data transfers and running the code on the Xeon Phi directly without offloads. All this was then to be compared to just running the code with offloading.

5.2.3 Testing OpenMP with Offloading

The code was then ready to be tested. Up to 64 threads were done without offloading and 512 with offloading. Results can be seen in figures 5.3, 5.4, 5.5 and 5.6. The tables with the results can be found in appendix A.3. The figures show us the execution time for encoding with varying amount of threads. The y-axis shows time in seconds. We have split the four videos into separate figures as the runtimes for the largest videos removed all detail from the smaller ones. We have also limited the y-axis to 100 seconds for tractor and 300 seconds for Bagadus and Foreman_4k. First of all we can notice the difference between static scheduling and the other two scheduling algorithms. The difference between offloading and running the encoder on the host CPU is also minimal compared to the difference between the scheduling algorithms. In all our runs we also see that going above 16 threads gives almost no improvement.

In the first graph with our Foreman encoding, the execution times with one thread have a very large difference between the statically scheduled runs and the other scheduling algorithms. The improvement per added thread in the static scheduling is lesser than we have in the other two. This mostly comes from the fact that the single thread times for dynamic and guided scheduling is much higher than in static. As tasks are not divided in the preprocessing the extra time spent on getting tasks for a single or even two threads caused a big increase in runtime. We saw, however, as the amount of threads increase, how the effect of the scheduling algorithm and that the overhead when dividing out tasks becomes proportionally less. This is less apparent in the larger videos as the larger and longer a task that needs to be performed per thread means less time spent with overhead on dividing tasks.

In all four videos the difference between the types of runs are about the same. The code that is offloaded is in general a tiny bit slower than the code on the host CPU. We also see a correlation in the improvements between the scheduling algorithms.

We even ran with just motion estimation being offloaded. No real difference could be seen, and if there was some it was closer to being a slight increase in total execution time.

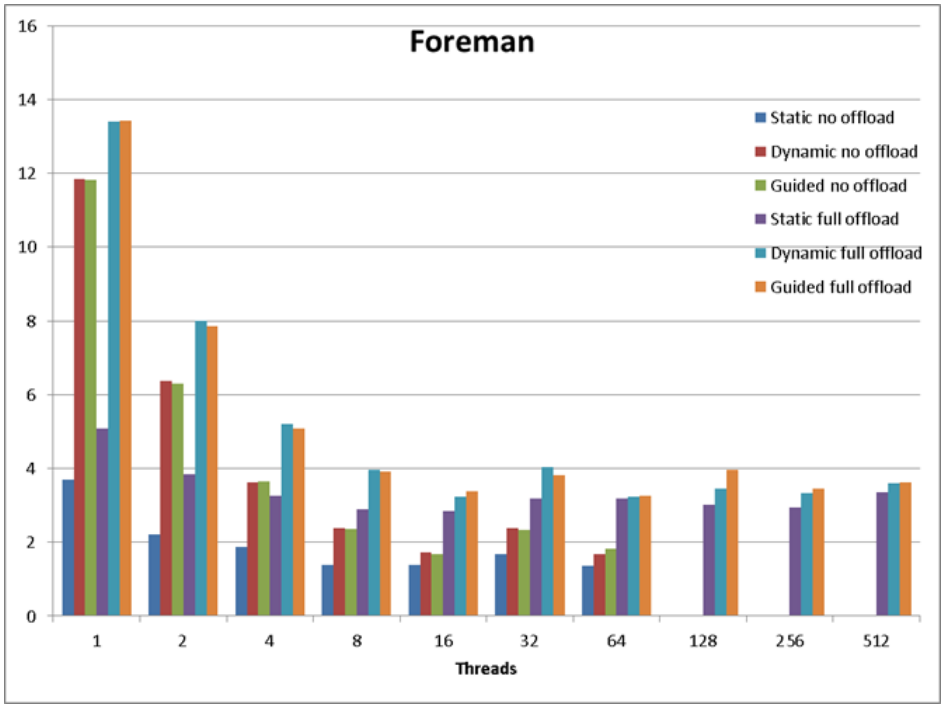


Figure 5.3: OpenMP Foreman execution graph

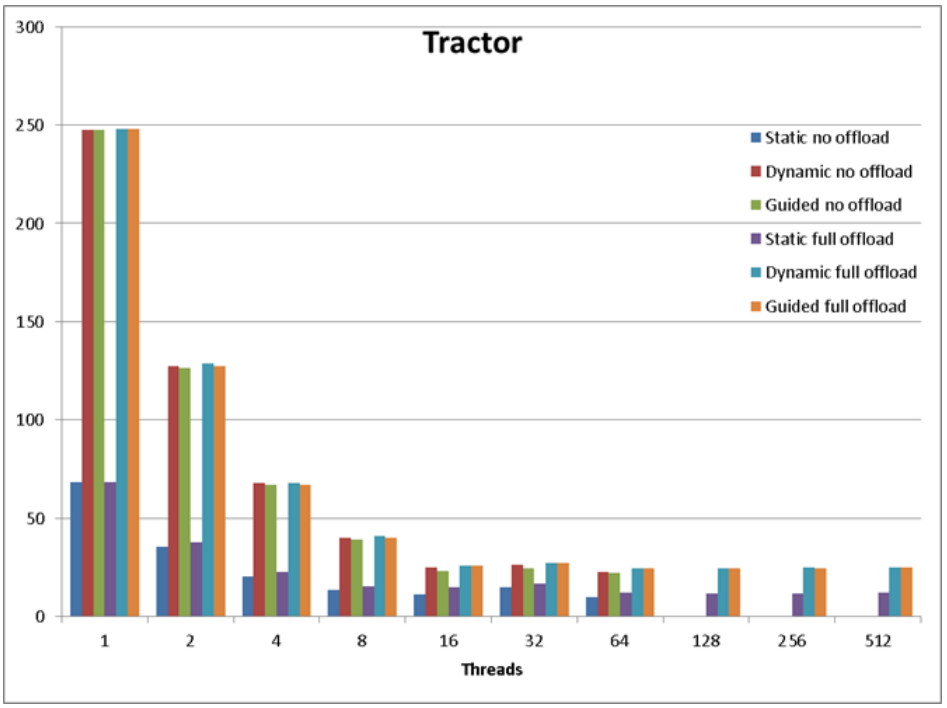


Figure 5.4: OpenMP Tractor execution graph

5.2.4 Measuring Encoding Only

Another part of what we measured was the per frame time on encoding only to see how much of an effect the offloading had. This way it was

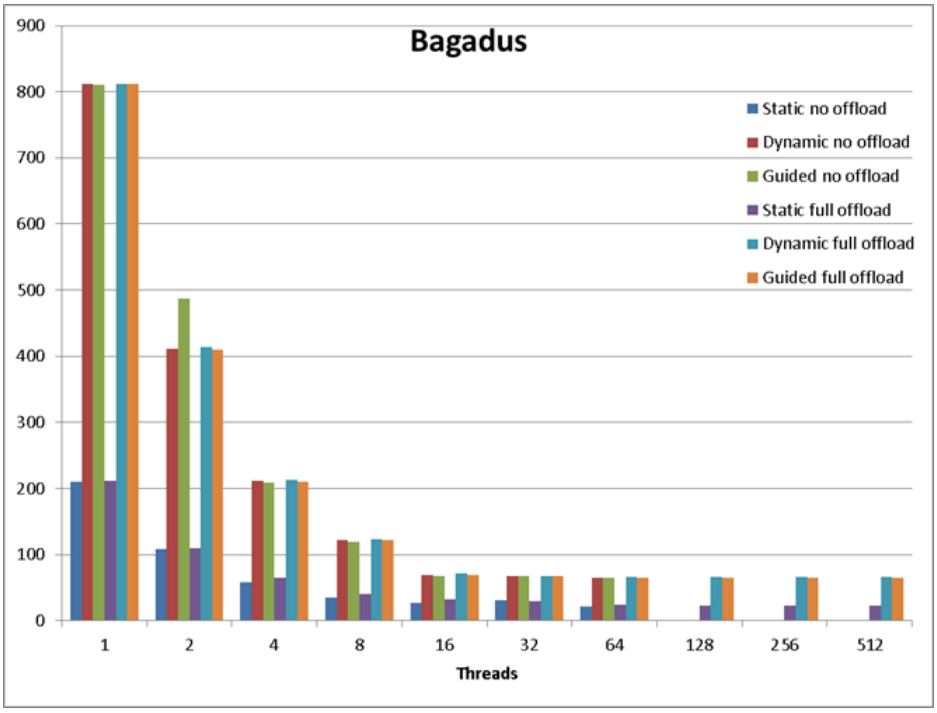


Figure 5.5: OpenMP Bagadus execution graph

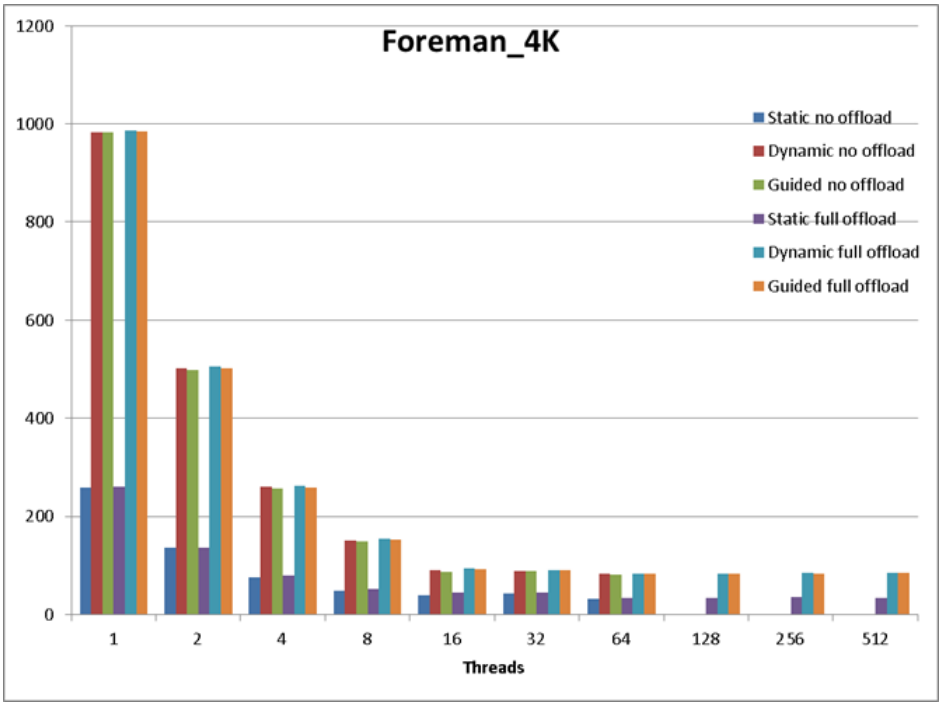
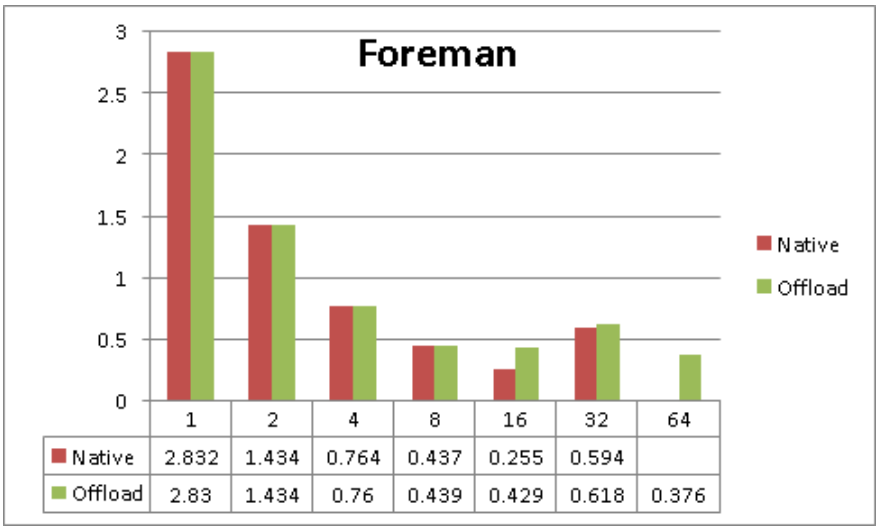


Figure 5.6: OpenMP Foreman_4K execution graph

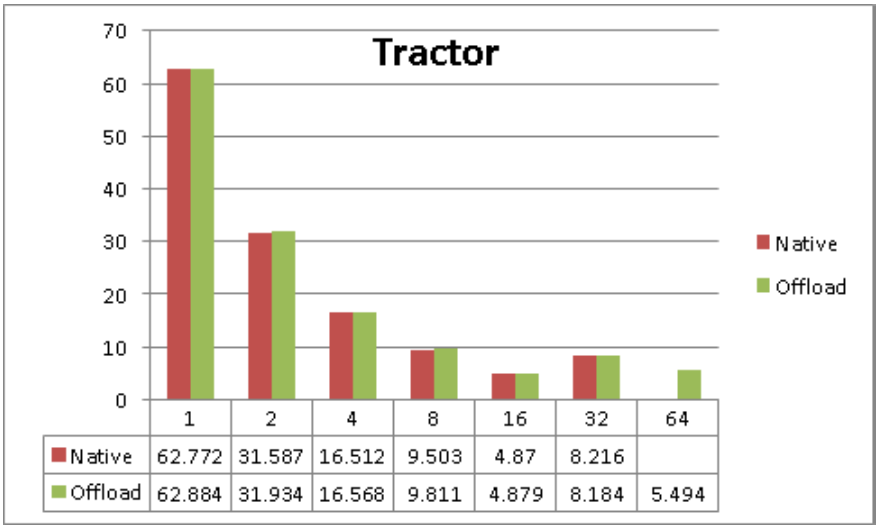
possible to see the execution time of the encoder without interference from the transfers. We only measured in regards to a full offload and we also did fewer threads. We did fewer threads as we had already noticed a big cut-off

in increased speed with more threads than we ran here and this also made it easier to see the comparison between offloading the code to the Xeon Phi and running it natively on the host CPU. The results from static scheduling can be seen in tables 5.7, 5.8, 5.9, 5.10. Each of these runs are 100 frames with a y-axis based on seconds.



	1	2	4	8	16	32	64
Native	2.832	1.434	0.764	0.437	0.255	0.594	
Offload	2.83	1.434	0.76	0.439	0.429	0.618	0.376

Figure 5.7: OpenMP Foreman encoding only



	1	2	4	8	16	32	64
Native	62.772	31.587	16.512	9.503	4.87	8.216	
Offload	62.884	31.934	16.568	9.811	4.879	8.184	5.494

Figure 5.8: OpenMP Tractor encoding only

We notice that the difference between offloading the code to the Xeon Phi and running it on the native host CPU is not very big. But we do see that as the videos get larger and we require more work per frame that the Xeon Phi encoding times get a bit lower.

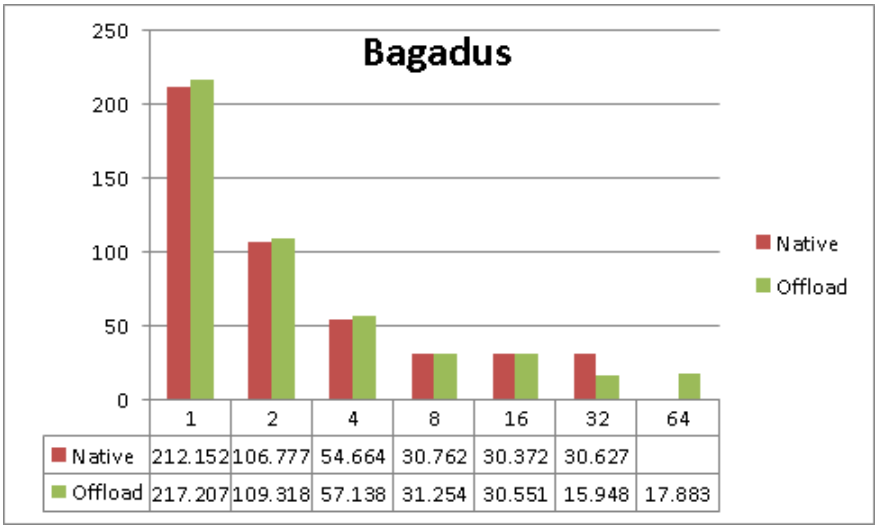


Figure 5.9: OpenMP Bagadus encoding only

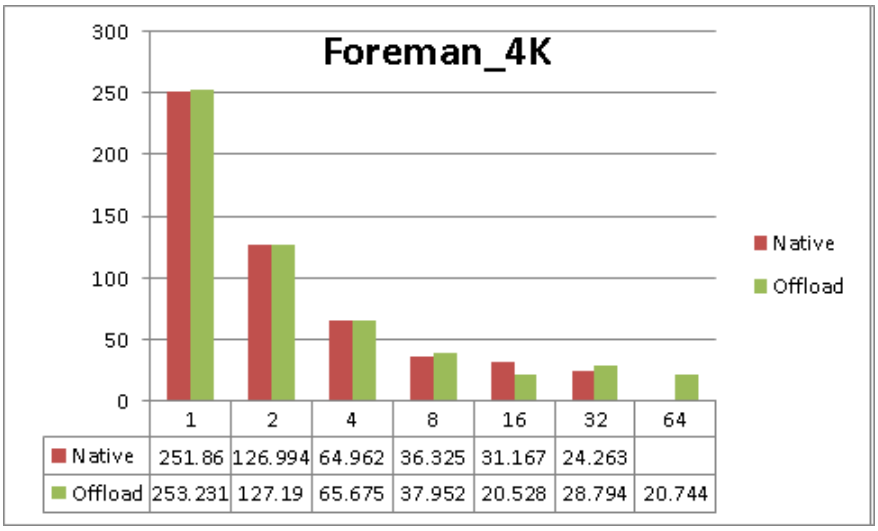


Figure 5.10: OpenMP Foreman_4K encoding only

5.3 Cilk

Cilk offered a different setup. Cilk is more based on shared memory than the offload structure we found in OpenMP. The creation of parallel loops, however, is much the same. An example of this can be seen in code example 5.4.

Code Example 5.4: Comparing Cilk and OpenMP loops

```
#pragma omp parallel for
for(i = 0; i < 10; i++) {
  //code
}

_Cilk_for(i = 0; i < 10; i++) {
  //code
}
```

Both then created the same result, although the actual dividing and sharing of work ends up being very different between the two. The second major difference was that instead of offloading data Cilk used so called shared memory. Thus instead of every normal allocation of memory call we used the Cilk shared memory allocation call and during deallocation the Cilk shared memory freeing. In Cilk the memory can then be worked on by both the native CPU and the coprocessor and will be synced between them. In our case there is no work happening on the native part while the offload runs so we can use an environment variable removing the synchronization during the offload and then performs a pure transfer afterwards instead of merging. An example of allocation and freeing on our reconstructed frame can be seen in code example 5.5.

Code Example 5.5: Allocating and freeing in Cilk

```
  f->recons = _Offload_shared_malloc(sizeof(yuv_t));
  _Offload_shared_free(f->recons);
```

The actual calculations in Cilk are pretty much the same though. The same setup is used in motion estimation with finding the lowest index instead of sharing a shared variable. So in the end the only key difference is memory in regards to code. The other thing that can not be controlled as well is the work scheduling based on the work stealing. For our code this proves to be very inefficient with a increase in execution time compared to work sharing. We can see the results in figures 5.11, 5.12, 5.13 and 5.14. We can also see the results in appendix A.5

The four figures, 5.11, 5.12, 5.13 and 5.14, show encoding time in seconds on the y-axis with the amount of threads used in the x-axis. The blue bars show the time required when we ran the Cilk code directly on the host CPU with the red and green bars showing the encoding time when we offloaded data. The green bars, called no updates, are the encoding times when we instructed that there should be no updates of the shared memory until after the offloaded finished.

As mentioned, one part of what Cilk does is that it has to update shared

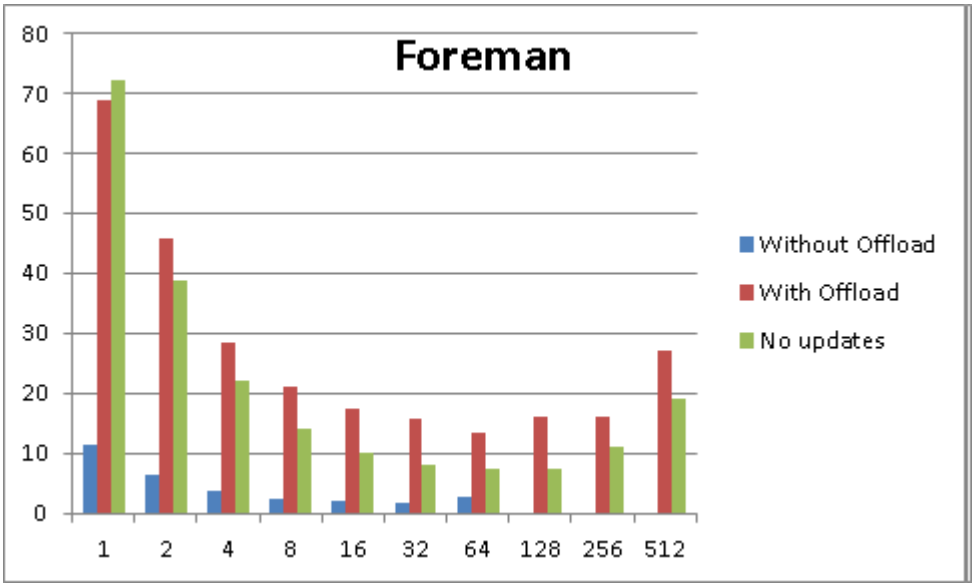


Figure 5.11: Cilk Foreman

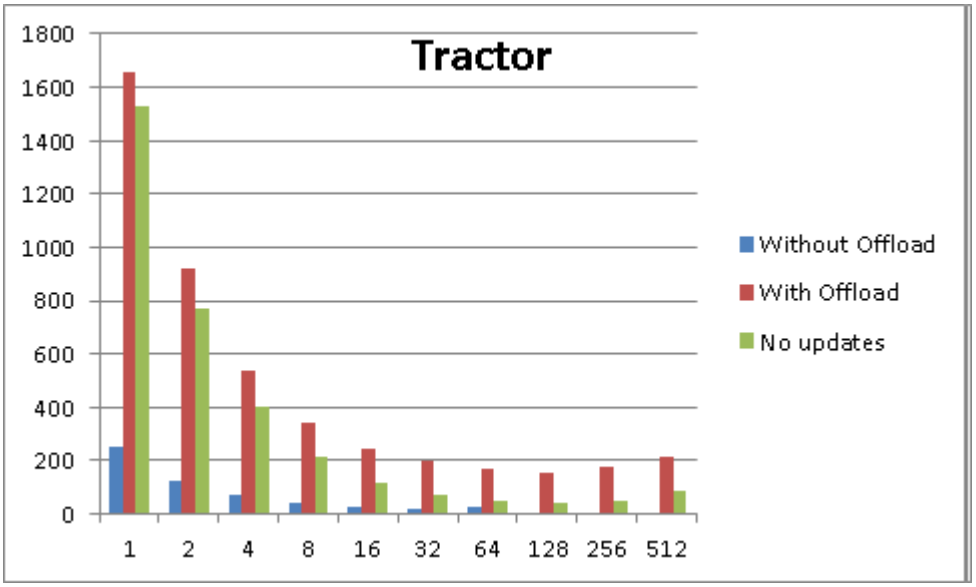


Figure 5.12: Cilk Tractor

memory for read and writes for both sides. This is because we might expect that both the coprocessor and the native processor work on the same memory. This causes a lot of extra writes and read into shared memory. Our model, however, does not have the native processor doing anything while the offload is working on it. As a result we can skip this by setting an environment variable. This obviously only affects the execution time of offloading but it gave a strongly improved result as seen in figures 5.11, 5.12, 5.13 and 5.14 with the data in table A.19.

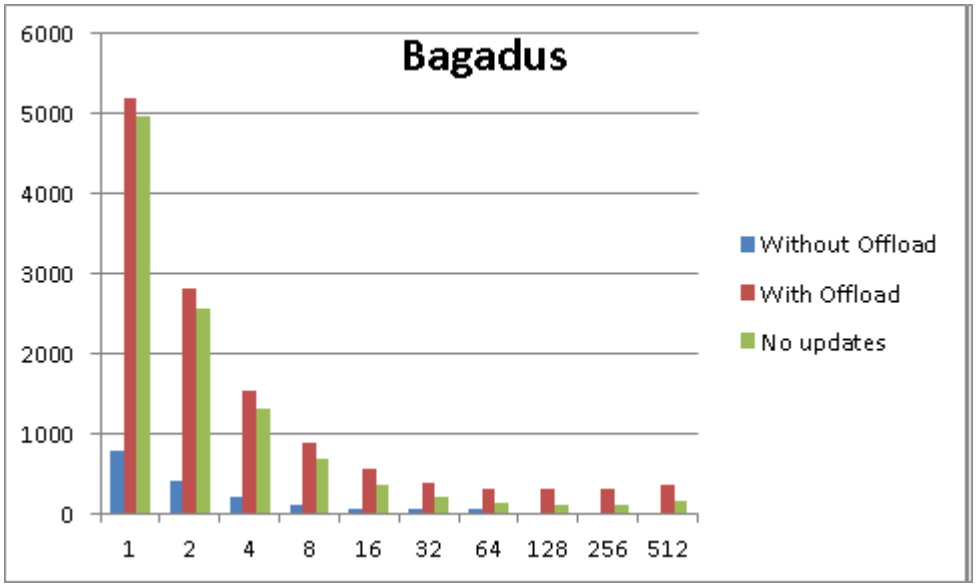


Figure 5.13: Cilk Bagadus

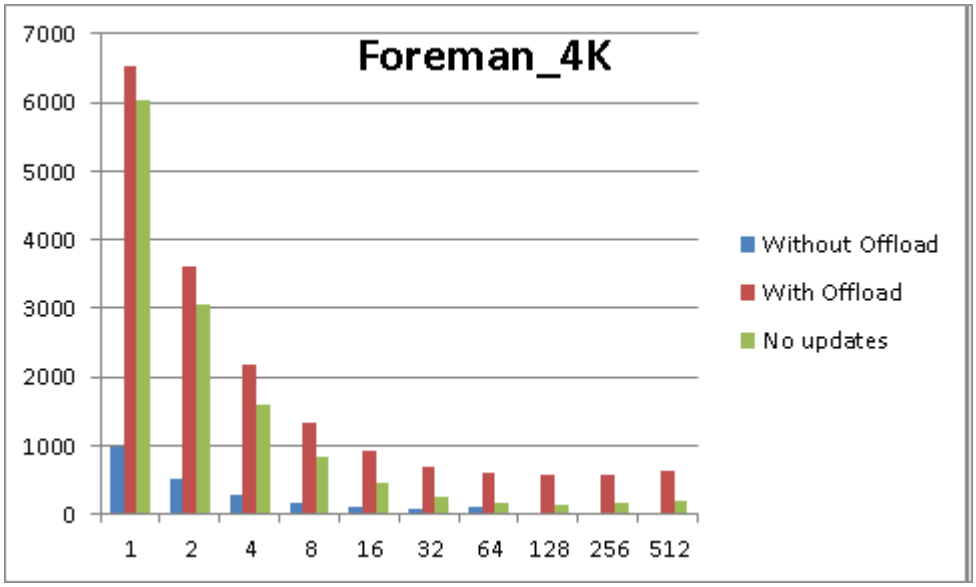


Figure 5.14: Cilk Foreman_4K

5.4 Fixing the Baseline Code

Most of the initial experiments were done with a very low number of frames, as some of the execution times where very high with some of the worse optimizations. So after finishing a decently fast code in both languages extensions we tested with the test setup we discussed in chapter 4. What we found was that this caused big memory issues later in the program. In most cases we ran out of memory on the Xeon Phi. The basic code allocated new memory for every frame, including the residual frames, predicated frames,

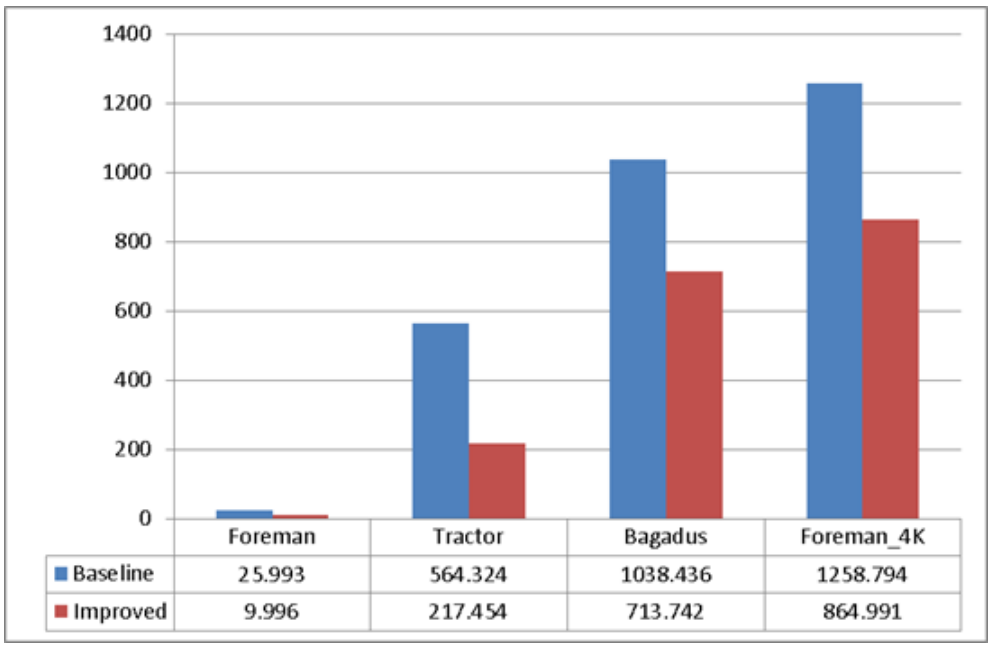
original frame and macroblocks. This meant we were allocating and freeing a very large amount of memory per run. The Xeon Phi, however, did not manage to sync up fast enough so in the end we ran out of memory crashing the program. We also realized that allocating and deallocating memory like the code does instead of resetting all the values inside it was not the best way to do it. So we started working on redoing the basecode by removing the allocations. Unneeded data areas were also removed. The residuals are only used as part of the previous frame and are also the only thing used in the previous frame. So we removed the setup with two frames inside the struct and used one only. We also altered the code so that instead of using two methods, one destroying the old frame and one creating a new frame we just memset the current frame, except for the reference data which we memset after doing the motion estimation. We can see the new function in code example 5.6.

Code Example 5.6: The next frame function

```
1  void set_frame(struct c63_common *cm, yuv_t *image) {
2    cm->curframe->orig =image;
3
4    memset(cm->curframe->predicted->Y, 0,
5           cm->ypw * cm->yph * sizeof(uint8_t));
6    memset(cm->curframe->predicted->U, 0,
7           cm->upw * cm->uph * sizeof(uint8_t));
8    memset(cm->curframe->predicted->V, 0,
9           cm->vpw * cm->vph * sizeof(uint8_t));
10
11   memset(cm->curframe->residuals->Ydct, 0,
12          cm->ypw * cm->yph * sizeof(int16_t));
13   memset(cm->curframe->residuals->Udct, 0,
14          cm->upw * cm->uph * sizeof(int16_t));
15   memset(cm->curframe->residuals->Vdct, 0,
16          cm->vpw * cm->vph * sizeof(int16_t));
17
18   memset(cm->curframe->mbs[Y_COMPONENT], 0,
19          cm->mb_rows * cm->mb_cols * sizeof(struct macroblock));
20   memset(cm->curframe->mbs[U_COMPONENT], 0,
21          cm->mb_rows/2 * cm->mb_cols/2 * sizeof(struct macroblock));
22   memset(cm->curframe->mbs[V_COMPONENT], 0,
23          cm->mb_rows/2 * cm->mb_cols/2 * sizeof(struct macroblock));
24 }
```

We then tested the old baseline code towards our new baseline code to see what effect it had on the code. Surprisingly the effect was much larger than we expected and halving the time required. See figure 5.15. Again the y-axis is the time in seconds for encoding the four videos we used. As can be seen, on the smaller videos our improvement is substantial while still being good in the larger videos. The reason we get a better improvement with smaller videos is that they spend proportionally more time with setup

between frames than the larger ones. This is because the larger videos require more time per frame encoding.



	Foreman	Tractor	Bagadus	Foreman_4K
Baseline	25.993	564.324	1038.436	1258.794
Improved	9.996	217.454	713.742	864.991

Figure 5.15: Baseline execution times without language extensions

This was then brought into our OpenMP and Cilk implementations. Both worked afterwards without issues with any amount of frames and both had marked improvements in execution time. We do not have any direct time measurements between them though as we only had limited amount of frames we could run and especially on the larger videos we barely had any frames.

5.5 Results from Encoding on the Xeon Phi

So with these results, it is necessary to understand their meaning. There are a number of interesting details to consider, like any unexpected data and how they compared.

5.5.1 General Results from OpenMP Encoding

For OpenMP we can see that we get a good speedup until about 16 threads, at which point the speedup is limited. We showed this earlier with figures 5.3, 5.4, 5.5 and 5.6. This happens both in regards to code we offload to the coprocessor and native code and with all three scheduling methods. We do note, however, that the cut off point for guided and dynamic scheduling

is more extreme than for static scheduling in both offloading and native running.

The important difference to notice is between static and the other two scheduling methods, dynamic and guided. What can be seen, as mentioned in chapter 1, is that static performs much better than dynamic and guided. This is because of the extra overhead during the sum of absolute differences (SAD) calculations. By looking at figures 5.16, 5.17, 5.18 and 5.19 which has our execution time where we use static scheduling on the SAD parallel calculations while using either dynamic or guided on all other parallel loops. This gave us a performance increase, albeit a small one, compared to just having static scheduling. This is very interesting as we know that static scheduling works better when work is evenly divisible and in thread size chunks but dynamic works better with more uneven balance and potentially varying chunk sizes with guided scheduling. So we can easily stipulate that with the varying size of width and height per video, we can have unbalanced workload. In the example of Tractor, the frame size is 1920x1080 which when divided by eight gives a natural number but with 16 it does not so we get unbalanced. Static in this case would divide all the work meaning that some threads get more work while dynamic would divide it on a per need basis giving the extra work to the threads that are done first. However, our SAD loop is always evenly distributed with the number of threads we tested with. This means that static can create a very even and balanced workload between threads and the overhead of dynamic has a bigger effect. The reason for the big difference in execution time though is that almost all our time is spent calculating SAD so an inefficient setup there will have a bigger effect.

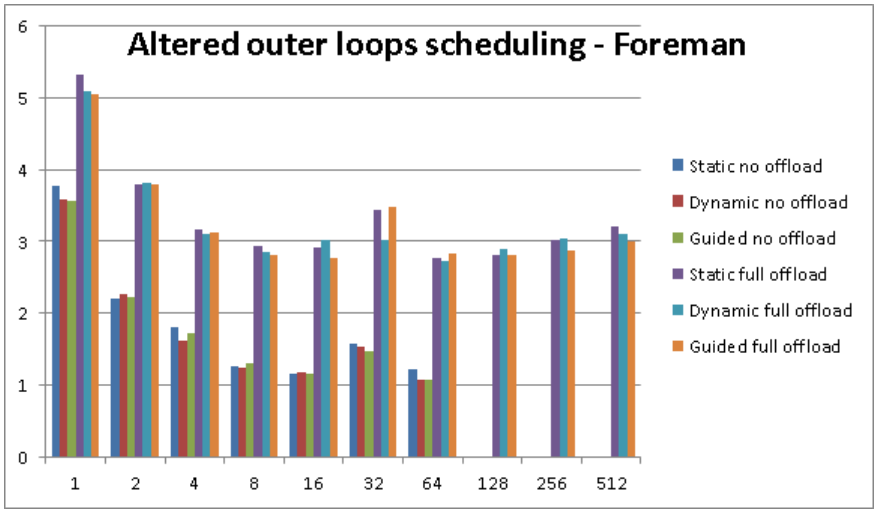


Figure 5.16: Combined scheduling for Foreman

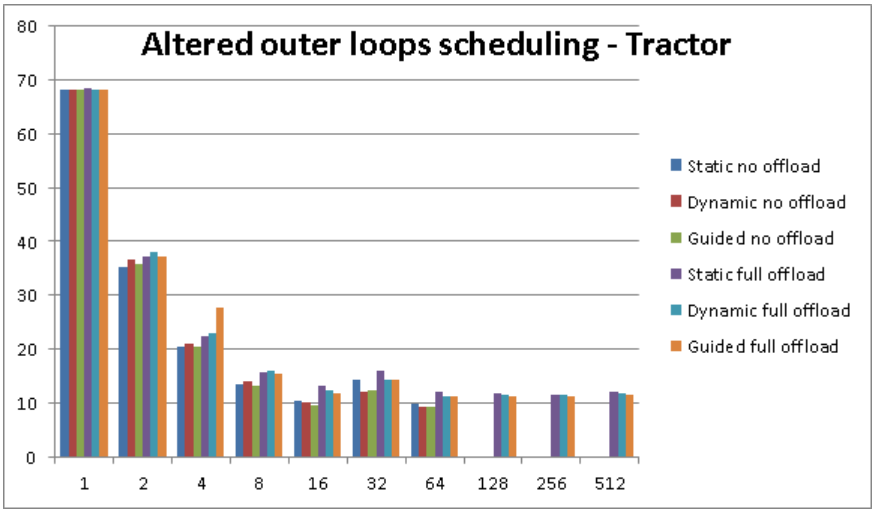


Figure 5.17: Combined scheduling for Tractor

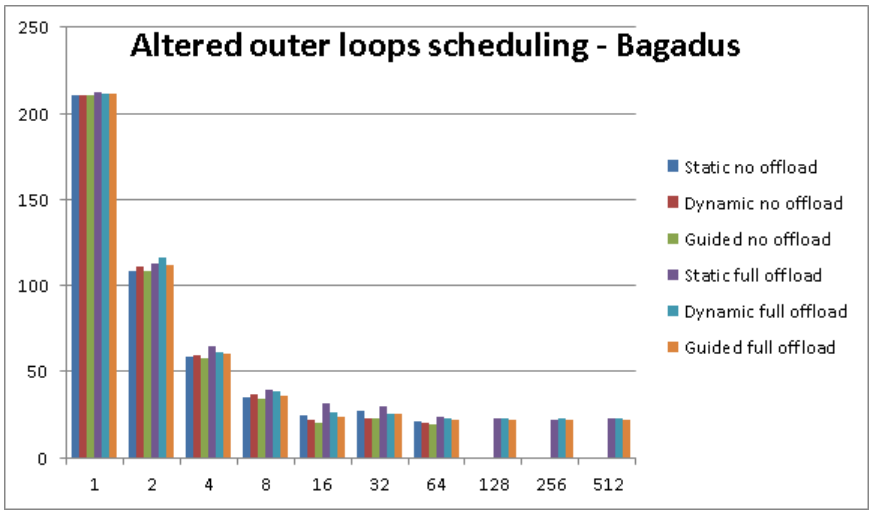


Figure 5.18: Combined scheduling for Bagadus

5.5.2 Increased Run Time with 32 Threads

Interestingly, we saw that when running the code with 32 threads we got a spike in the execution time. This happened in both offloading to the coprocessor, in native running on the host CPU and with all scheduling methods. This required further analysis as there was no direct reason for it. We could consider the size of the frames with the first logical reason being unbalanced workload. However, as we have the same problem with all videos and some videos having sizes that are dividable by 32 this did not seem like the reason. When doing thorough testing on why this happened we noticed that there was a very high variation in the execution time with 32 threads. We did consider that the CPU did some caching with multiple runs of the program and we did repeated runs to test for it. What we

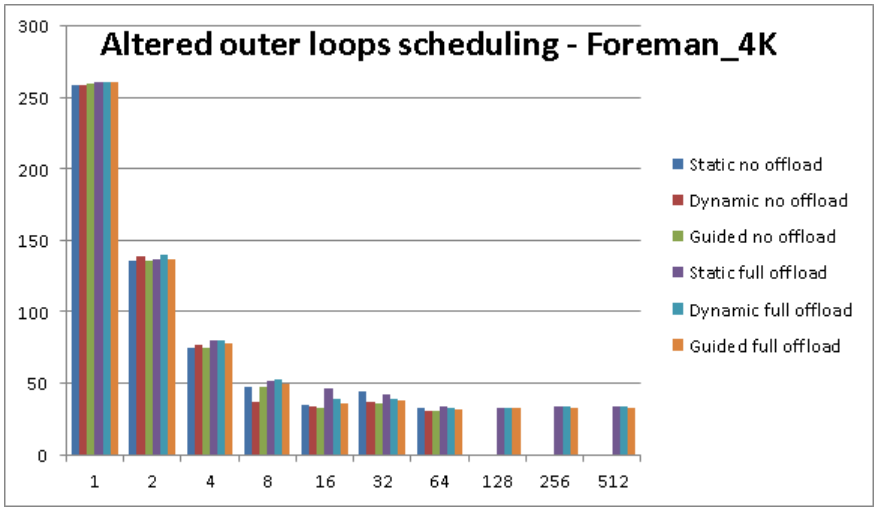


Figure 5.19: Combined scheduling for Foreman_4K

noticed was that other thread counts seemed to stay fairly steady or within 5% difference between runs on Foreman_4K while 32 threads had a much more fluxuating execution time. Variance between runs there was about 30%.

So we profiled the run with 32 threads to see if we could spot anything. However, given how random the difference seemed to be it was hard to detect the problem. What we did notice was that CPU utilization seemed to drop from time to time as seen in figure 5.21 compared to when the runs did execute in a timely manner, which can be compared with figure 5.20. The two figures show us the load per worker thread with the brown parts being actual work and the green idle time. During the slower run the idling is much more common. Seeing how this happens on regular occasions the more threads we have the more likely it is. Still, going above the 32 threads the machine has the utilization seemed more even. This was the only conclusion we were able to reach as this requires more research. This will be discussed further in chapter 6.2. It is also extremely important to note that we also see these spikes in other runs as well, but not as frequently as we do with 32 threads. This is something we consider during our analysis of our results.

5.5.3 Encoding Only

Another comparison done was to see the time the actual encoding required, removing the timing for reading and writing the image and offloading the data. We can see that encoding during offloading is a bit faster. We would expect it to be and but we would have expected more than that. We can see that when comparing the same number of threads the time is about the same, but as we add more threads the offloaded code gets faster. The

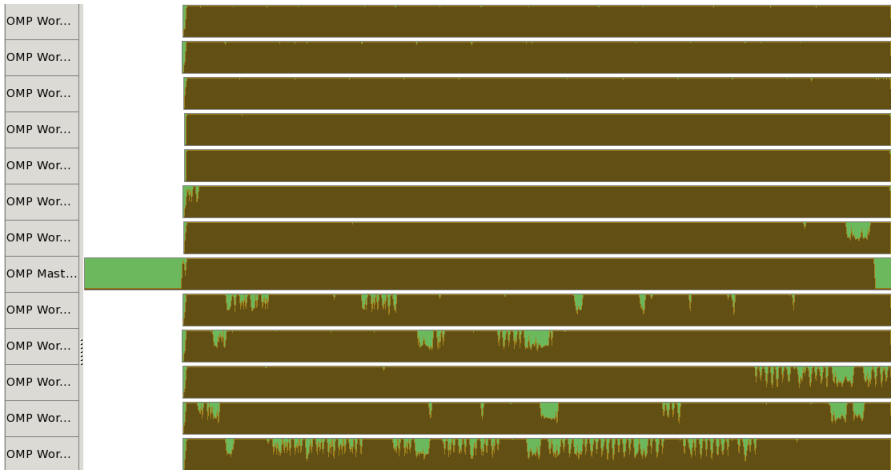


Figure 5.20: The faster 32 thread run

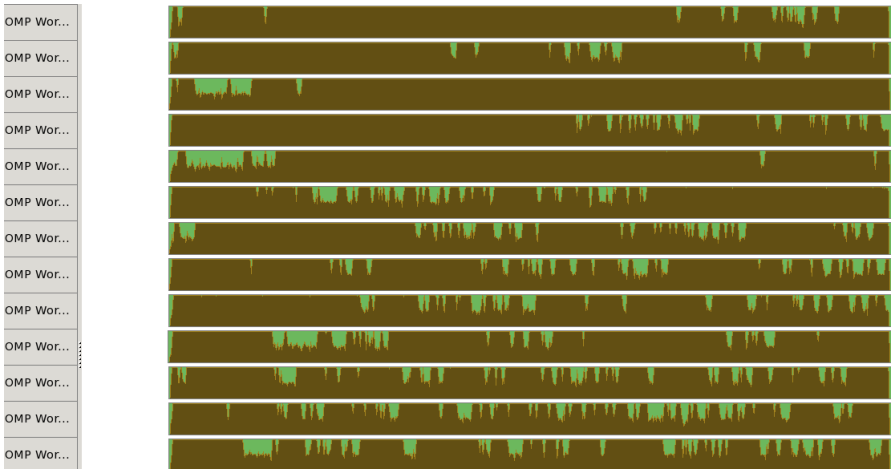


Figure 5.21: The slower 32 thread run

difference is still not as big as we would expect. We assume though that the difference is not bigger because of how powerful our native processors are. In figures 5.22 and 5.23, we can see the execution times for our foreman video and our foreman_4K video. In general, we do not see much of a difference between threads when using the same amount of threads but we do see a substantial difference between when comparing 32 threads to 128 threads or more on the foreman_4k video. The difference on the smaller video, however, is very small and if any skewed towards not offloading. We can find the tables with values in appendix A.7.

5.5.4 Running Natively on the Xeon Phi

We also ran the encoding of foreman and tractor on the Xeon Phi directly, that is we used the Xeon Phi as the native host. When we ran with 128

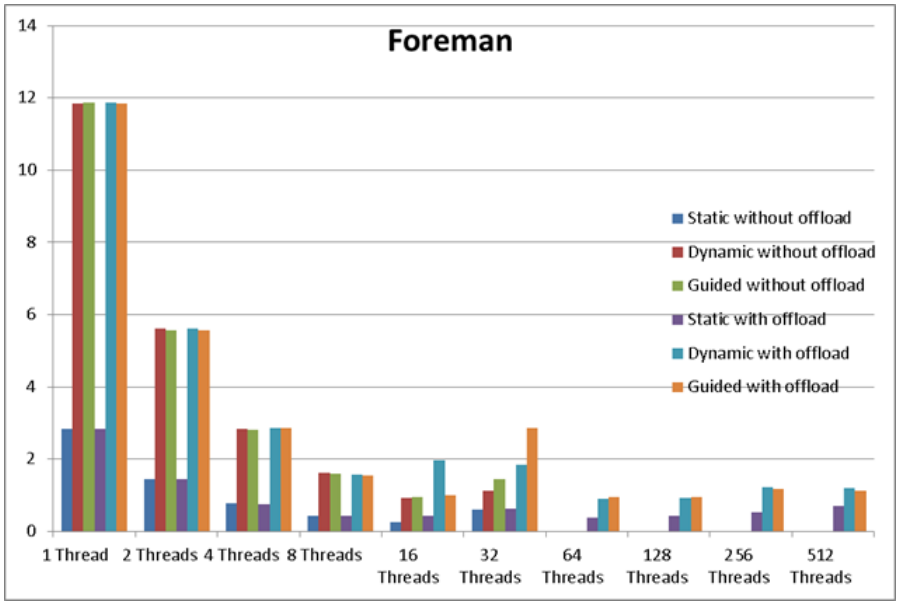


Figure 5.22: Measurement of encoding only on Foreman

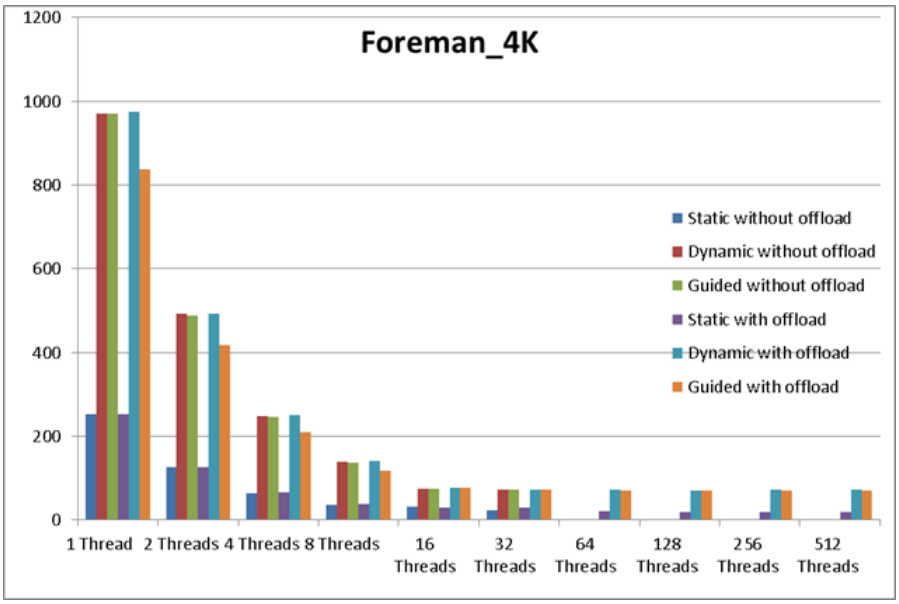


Figure 5.23: Measurement of encoding only on Foreman_4K

threads for tractor and measured the encoding only it took 36.1857 seconds and foreman took 2.167 seconds. This does fit with what Intel says about what code can be run natively [4]. Comparable execution times, with offloading, where 5.321 seconds for tractor and 0.437 seconds for foreman. Native applications most of all require that as much as possible of the code is parallel and a modest memory footprint. As much of the I/O operations in our code are sequential and we have a fairly complex memory footprint our execution time is higher than with offloading. This was fully expected.

5.5.5 Cilk

As suspected Cilk suffers a bit from the scheduling it uses. After fixing how it handled shared memory, removing the constant updating as was discussed in chapter 5.3, our biggest effect on the execution time is the scheduling. We could see a lot of downtime between worker threads as we are doing a lot of updates in memory between the native processor and the coprocessor. But as discussed in chapter 2.1.4 Cilk uses a random work stealing algorithm. When profiling the runs, as seen in figure 5.24, that the biggest difference compared to OpenMP is exactly the work stealing. This is as expected and discussed in chapter 1. As the work is mostly evenly distributable, especially the SAD function where most of the execution happens, the overhead for the work stealing results that can not compete with OpenMP.

Function / Call Stack	L2_DATA...	CPU_CLK_UN...
[Loop@0x4083d3 in _BE_._me_block_8x8_73_cilk_for_lambda_1]	24,480,000	5,108,400,000,000
[vmlinux]	103,560,...	200,094,000,000
[Loop@0x4082d0 in me_block_8x8]	1,920,000	55,872,000,000
[Loop@0x404101 in dct_quant_block_8x8]	60,000	54,642,000,000
[Loop@0x1949b in search_until_work_found_or_done]	5,820,000	52,152,000,000
[Loop@0xd0d0 in cilk_for_recursive<unsigned int, void (*)(void*, ui	1,140,000	22,062,000,000
cilk_for_recursive<unsigned int, void (*)(void*, unsigned int, unsigr	120,000	19,236,000,000
cilk_for_recursive<unsigned int, void (*)(void*, unsigned int, unsigr	360,000	18,834,000,000
[Loop@0x4099b0 in _BE_._mc_block_8x8_144_cilk_for_lambda_1]	3,300,000	14,988,000,000

Figure 5.24: Profiling Cilk

What also is interesting is that as the thread count goes up we see a increase in runtime as can be seen in figures 5.11, 5.12, 5.13 and 5.14. This is another proof that the effect of the overhead is to big. This is specially noticeable on the smaller videos as there is not enough work to go between all the threads for them to be effective. With the larger videos the effect is not as extreme as the smaller ones but still present. This is though mostly as the low thread number require such a long execution time and the increase at high thread number does not share as steep a curve.

5.6 Implementing in CUDA

The CUDA code offered a different challenge. The aim was to create a code that resembled the OpenMP code, being highly portable. CUDA on the other hand, expects a lot of manual setup in regards to how many threads and blocks you use. So the first challenge was to find out a way to do loops dynamically in CUDA instead of the general static way with thread and block ID's. As we are also working in a 2D environment all loops are double. So to create dynamic loops we opted for the code seen in 5.7.

Code Example 5.7: Our loops in CUDA

```
for(i = (blockIdx.y * blockDim.y + threadIdx.y) * blockDim.x * gridDim.x +
        (blockIdx.x * blockDim.x + threadIdx.x);
    i < height*width;
    i += blockDim.y * gridDim.y * blockDim.x * gridDim.x)
  {
    x = i/width; //x is here the outer value
    y = i%width; //y is here the inner value
    //code
  }
```

This gave us very portable loops that could run with any setup of threads in each kernel as we wanted. However, the loops were a bit more complicated. As we had varying amounts of depth in the loops we had to consider. In the case of motion estimation we had first a double loop finding our 8x8 block of pixels followed by a 2D search for similar blocks. To adapt to this we used the block dimensions to create these dynamic loops in the outer parts where we found the pixel blocks and the thread dimensions in the inner part. An example of both can be found in code example 5.8.

Code Example 5.8: Motion estimation loop example in CUDA

```
for(i = blockIdx.y * gridDim.x + blockIdx.x;
    i < columns*rows;
    i += gridDim.y * gridDim.x) {
      //Find best sad
}

for (i = threadIdx.y * blockDim.x + threadIdx.x;
     i < search_range;
     i += blockDim.x * blockDim.y) {
       //Start motion estimation on block
}
```

We applied these types of loops to the same loops as in our OpenMP code. However, considering how lightweight threads are in CUDA it would not be a fair comparison so we added the inner loop inside the motion estimation, as it would guarantee our threads running code that fit with the warp size. In the same regard we could have taken loops even deeper, all the way into the DCT calculations, but that would not have created a fair comparison with OpenMP.

We also tried to mimic how the Xeon Phi found the best SAD value. To do so we created our own reduction method to run in parallel finding the best index. We created an array initialized with indices and then stored all results in an array from the sad calculations. See code example 5.9.

Code Example 5.9: Find smallest value reduction

```
for(; half > 0; half>>=1) {
  for(i = threadIdx.y * blockDim.x + threadIdx.x;
      i < half;
      i += blockDim.x * blockDim.y) {
    if(values[i] >= values[i+half]) {
      values[i] = values[i+half];
      ind[i] = ind[i+half];
    }
  }
}
```

The only real issue with this code is the branching based on the if but that is a general issue with parallel programs. This returned the index based on where we were in the image. The same method as in OpenMP was applied to calculate the actual point in regards to where we were in the image to get the right motion vector.

The only real difference then became the offload part, or in the case of CUDA, the allocation and copying. So to start with we initialized the c63_common struct on the host processor before allocating it on the GPU. After this we started the encoding, much like in the baseline code. However, we did need to split the encoding into three parts due to synchronization between blocks between the motion estimation, motion compensation and quantization and dequantization. We did not need to split the quantization and dequantization as each block would work on the same area so we only had to synchronize the threads. Our moving between frames also required extra work as we had to both copy and memset on the GPU for each run.

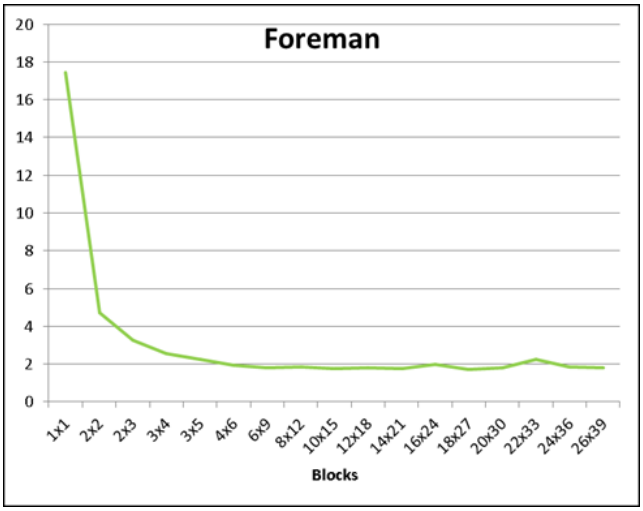


Figure 5.25: CUDA Foreman execution time graph

In the end the execution time results showed strong improvements in regards to increased amount of blocks we ran. We can also see a strong cut-

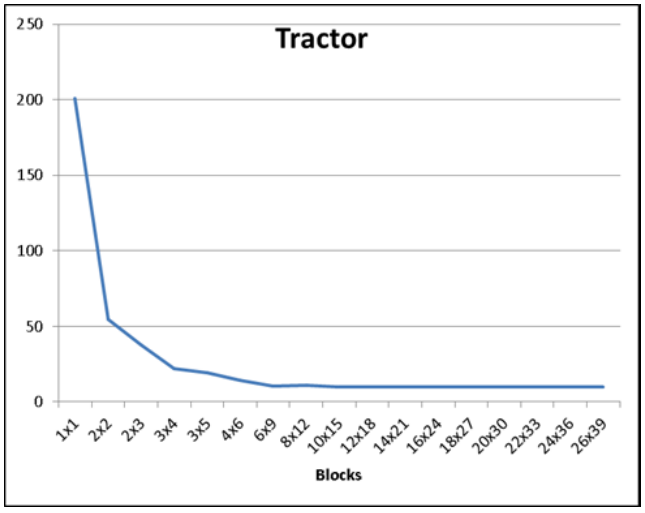


Figure 5.26: CUDA Tractor execution time graph

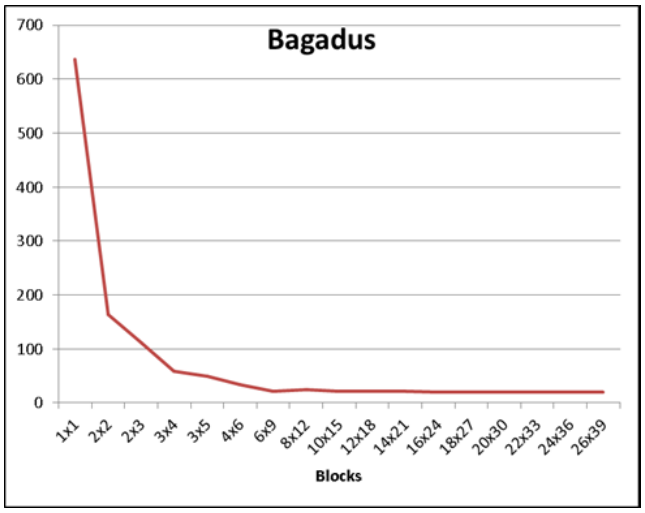


Figure 5.27: CUDA Bagadus execution time graph

off in improvements from about 6x9 blocks with only minimal increases. On around 18x27 we notice no more notable improvements in execution times. It should also be noted that each block had 16x16 threads. As we know running code with less than 32 threads is a big waste of potential as each warp has a multiple of 32 threads running. We can also see that execution times with just 1x1 blocks with 16x16 threads are extremely slow and much slower than both Cilk and OpenMP when they run with a single thread. We therefore did not test with less than a minimum of 256 threads. See figures 5.25, 5.26, 5.27 and 5.28. These figures show us the encoding time needed in seconds on the y-axis and the amount of blocks we used for each run. A table with the runtimes in can be seen in appendix A.6.

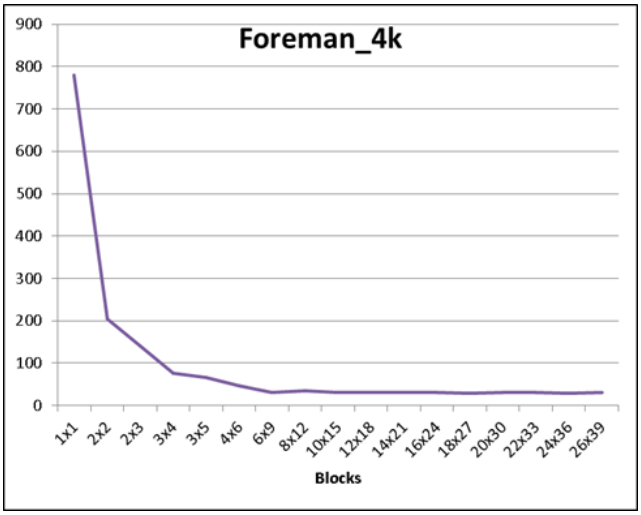


Figure 5.28: CUDA Foreman_4Kexecution time graph

5.7 Results from the CUDA Implementation

CUDA offers us a bit of a different setup and requires different thinking. Most of our data is done with a minimum of 256 threads compared to the lowest in both OpenMP and Cilk being one thread. And at the same time our increases in the amount of threads is much bigger. So when looking at the results this is something that is important to consider. As can be seen in figures 5.25, 5.26, 5.27 and 5.28, we do not gain much when moving above 54 blocks or 13824 threads.

Most important to note is that we still got better execution times with CUDA than we did with anything on the Xeon Phi. This was even if we tried to keep as close in implementation to the Xeon Phi as we could. However, we need to consider that we still optimized the actual SAD calculation. As expected, though, the Xeon Phi would not have as good results as the Titan. We can both deduct this from the fact that the less overhead we have the better our results and the fact that everything we do works in a 2D vectorized space. These are areas where CUDA based programs should excel.

5.8 Problems During Implementation

There were two notable problems that occurred while implementing our solutions.

The first real problem we had was with shared memory allocation for Cilk. As we were trying to allocate fairly big parts of memory as we were reading each frame we noticed something we could not find documented. So if we tried to use fread to read more than 524.287 bytes of data into our memory

area we got weird segfaults. However, reading 524.287 bytes multiple times in a row into the area worked fine. We can see our looped reading method in appendix B.3. We also noticed that if we memset the whole area before reading into it it worked fine. We therefore drew the assumption that memory allocated as shared memory in Cilk got lazily allocated. As fread uses a bit of buffering and other optimizations we can assume that the memory just was not ready when we tried to write into it. It is worth noting that 524.287 is exactly 19 bits so we can assume that is the limit of the eager part of the allocation.

The second problem during our implementation was with the parallelization of our SAD calculations. As the calculation is basically a summation of numbers our idea was to use a parallel reduction to calculate the value, much like we ended up doing in CUDA. This did in no way go as we planned and we saw probably the biggest negative effect out of any optimization. The new execution times were about 10 times slower than our other attempts. We quickly deducted from profiling that this was a result of the massive amount of thread synchronization. We still used it in the CUDA code as the threads there are much more lightweight meaning it was almost required to get meaningful results.

5.9 Comparing All Three

When we compare the execution time based on threads between all three language extensions we most of all notice that much of the results fit with what we discussed in chapter 1. Static scheduling proves to be better than both work stealing in Cilk and the other two scheduling methods in OpenMP. The difference, however, between dynamic, guided and Cilk when running on the native processor is minimal at best. We also notice that the actual encoding on the Xeon Phi is a bit faster than the on the host CPU. The time spent transferring data, on the other hand, makes the actual encoding time close to the same.

Comparing Cilk to the rest it is important to notice that the single thread executions are slower than the comparable ones in OpenMP. As can be seen the scheduling has no effect on single threads between OpenMP we can conclude that the difference relates to the memory management of Cilk. Removing the constant updates on shared memory during offloads removed most of the time required to the difference in time required to encode the videos on the Xeon Phi compared to the host CPU. Cilk still performed slower than our OpenMP and CUDA solutions. We attribute this to the scheduling used.

Like we stated CUDA performed better than both OpenMP and Cilk. This is directly related to the fact work stealing and dynamic work sharing perform worse than static work sharing. As the code which requires most of the CPU

time is very structured with little imbalance CUDA should perform better in all cases with the lightweight threads that it has. This was also the case and we saw the best results with CUDA. Most interesting was that we mainly created optimizations for Xeon Phi and tried to mimic these in CUDA but still managed to get better encoding times.

5.10 Complexity

An important aspect of our research was how easily the language extensions are in regards to optimizing the code. So far this has not discussed much at all. First of all, as both Cilk and OpenMP can be run without the need for specific hardware, adapting the code is possible from the start. Compared to CUDA which requires a GPU to be executable OpenMP and Cilk have an advantage.

However, the important thing to look at is the utilization of a Xeon Phi compared to a GPU. This complicates things a bit more. First problem came due to our baseline code not being very well set up. There were to many allocations and data transfers became very complicated and not at all easy to transfer between the host and the coprocessor. This applied to both Cilk and OpenMP. While Cilk had an easier setup with shared memory we could see the effect, even on a optimized code, the shared memory had if not used correctly. To much synchronization between host and coprocessor meant that the Cilk worker threads were idling a lot. OpenMP had another challenge with it requiring a more controlled memory setup, not unlike CUDA. You could allocate new areas every time you do an offload section but this creates a plenty of overhead and is generally negative. So setting up a dummy offload with no executable code beforehand might not be the most intuitive way of doing things. It is also necessary to setup areas only used in offloaded code which also provided a challenge.

Comparing OpenMP and Cilk the actual functioning code through Cilk was easier. This was as a result of shared memory saving us from any real thought when it came to doing the Cilk part. However, due to the constant synchronization of memory and the work stealing algorithm any code written in Cilk had low speedups meaning it was not very viable. OpenMP however had good speedup with static scheduling.

CUDA in our case was more complicated. It requires calculation on registry and kernel usage as our goal was create portability we had to do extra calculation to figure out how to do the loops. Synchronization between blocks is also limited so we had to finish an execution kernel execution before we moved to the next step in our encoding calculations.

5.11 What Does It Mean?

So what can be learnt from our experiment is that while CUDA still gives us faster results OpenMP is not far behind when it comes to execution times. We did discover that not offloading OpenMP code in our cases was better than offloading, however it has to be noted that the host CPU was more powerful and better suited than most CPU's. OpenMP in itself is easier in implementation with less knowledge required from the programmer than CUDA. Offloading though did require some knowledge, especially in regards to knowing what we to offload and controlling when and where things get offloaded. This was a bit more intuitive in CUDA compared to the other extensions although, still, very similar. Cilk was probably the hardest to set up in regards to getting decent execution times but at the same time easiest to get executable code with. The reason being how control one has over communication between the host and the coprocessor.

It is important to note that the encoder we optimized is far from effective so the results are a bit exaggerated. However, it is possible to transfer the results towards better encoders with both the workload being similar and we can see the effect of the work schedules on the way our data is structured. As was mentioned it would be interesting to see what effect the schedules have on varying size macroblocks.

So in the end we learnt that the smaller the overhead the better the performance. This comes from CUDA with the lowest overhead per thread, towards static scheduling in OpenMP then to dynamic scheduling types until we reach the work stealing in Cilk. However, CUDA requires a bit more expertise to get the desired results meaning that OpenMP is still a very viable option.

With all this in mind even if the GPU is potentially more powerful, using the Xeon Phi can give strong benefits. Especially in regards to older code that might be hard to optimize, making the real question how much a few extra milliseconds saved matter. If they are not that important the time saved with the potentially easier portability of the Xeon Phi can have a bigger value.

5.12 Summary

In this chapter the implementation used was discussed. All the steps taken in all the language extensions and variations of the design were listed in detail. We also went through our results and any problems we had. In the end we discussed what all of this meant in a broader context.

Our initial work went into creating a basic parallelized version of the Codec63 encoder. We then started working towards offloading data to

the Xeon Phi to run before creating a similar implementation for Cilk. We discovered a problem with how the basic behavior of the code when offloading and worked on improving the code. We then finished by implementing our CUDA code.

We tested all the work scheduling algorithms we had available, as we set out to in our problem statement, and learnt that static scheduling performed best on the Xeon Phi when only using a single algorithm. However, a mix between guided and static proved to be the best setup even if the difference in results was minimal. We also discovered that CUDA still performed better than anything on the Xeon Phi.

Finally, we discussed a bit around how complex the coding was and did a comparison on all three.

Chapter 6

Conclusion

To draw a conclusion from our results it can be stated that most of our theories that had been outlined were confirmed. As expected evenly chunked work division with little overhead gave better results. It is also obvious that the offloading is not as expensive as we thought even though it has an effect. Finally, GPGPU proved to be better than HPC as was expected.

6.1 Summary and Main Contributions

The thesis started by going through the basic background and ideas on why we wanted to do this research. It defined the problem we were looking at and any limits of the research we did. We discussed the hardware and scheduling algorithms before detailing video processing. We described Codec63 and how we planned on implementing our optimizations. We then covered how we implemented our solutions and described our results.

As mentioned in our problem statement we compared the hardware we had and the three language extensions we used. What we learnt was that GPU still performed better although Xeon Phi offered an easier way of implementing a parallelized version of our encoder. Utilizing the Xeon Phi and CUDA still requires a lot of attention to fine detail and very broad knowledge.

This all worked towards our goal of finding better ways to handle multimedia. This is especially important as multimedia is becoming ever so important. We tested various scheduling algorithms with the goal of seeing how they did with the encoding algorithms we used. We found out that static scheduling performed best although mixing the algorithms also provided good results.

We looked at how Codec63 worked and where we could look towards parallelizing it. We discovered that the Codec63 has some issues with memory allocation during offloads so we improved memory usage in it. We noticed that that most of the time was spent on motion estimation. This meant that most of our work was targeted towards improving motion estimation.

A big factor in optimizing the code was that memory serves a very important part. If there were to many threads trying to access memory within a certain boundary we started having synchronization problems and noticed that our encoding time increased substantially. Our optimizations also have some room to work with, like SIMD instructions.

CUDA still proved to have better encoding times but required more complex work. Especially important was thread synchronization and selecting the amount of threads to utilize. This was specially the case with the larger videos as with the smaller the difference between Xeon Phi and CUDA was too small to measure.

6.2 Future Work

There are a number of things that require further research. As multimedia keeps evolving we need to improve our methods for encoding it. We only looked at a 2D DCT algorithm and a very standard grid search for motion estimation. Potential future work would include other motion search algorithms such as a diamond search and other types of DCT algorithms. Our motion estimation only depends on the previous frame but motion estimation in modern encoder usually depends on previous frames and future frames. Researching improvements on these algorithms should also be of interest.

We barely touched the fall in execution time on 32 threads when running the code natively on the CPU. We did see that there seemed to be more idling on the CPUs but not if that was caused by overhead or something else.

We also did not do any research on an optimized encoder. Something that would be especially interesting to research is an encoder with varying macroblock size and what effect this would have on the scheduling algorithms. We can theorize that the workload would become more unbalanced this way which can potentially make a scheduling algorithm with more overhead more effective. We also did not try doing work on both the native and coprocessor at the same time. Using the encoder multiple Xeon Phis as a distributed system should also be of interest. Researching the potential of using the Xeon Phi to encode incomming data over a network should also be considered as potential future research.

We have not tried any SIMD vectorization optimizations. These require us to do work on how our code is written along with large parts of the code needing to be changed. This would therefore be an excellent next step in this research.

The research we did also has potential implications for workload in related applications. Computer vision is a growing field that requires more research. A computer could use much of the same algorithms we used, especially motion estimation, to process images. Our research could, therefore, be used in relation to computer vision. Another growing field is the research into neural networks which also relies on some of our algorithms.

Appendix A

Execution Times

A.1 Baseline

	Baseline code	**Improved baseline**
Foreman	25.993	9.996
Tractor	564.324	217.454
Bagadus	1038.436	713.742
Foreman_4K	1258.794	864.991

Table A.1: Baseline execution times

A.2 OpenMP First Attempts

These are the first attempts at parallelization with OpenMP. They contain data were we optimize all of the loops in motion estimation, just the outer loops of motion estimane and a combination of optimized outer loops and optimized sad.

Foreman	10.661
Tractor	132.187
Bagadus	342.264
Foreman4k	405.748

Table A.2: Execution Time with motion estimation parallelized

Foreman	3.907
Tractor	48.804
Bagadus	108.463
Foreman4k	130.910

Table A.3: Improved Parallel motion estimation

	Foreman	**Tractor**	**Bagadus**	**Foreman_4k**
1 Thread	3.619	77.750	245.507	301.492
2 Threads	2.510	44.102	132.715	165.689
4 Threads	1.783	25.712	78.133	98.483
8 Threads	1.462	18.954	53.571	70.678
16 Threads	1.452	14.677	53.242	70.689
32 Threads	1.830	23.150	47.602	72.113
64 Threads	1.381	14.021	37.560	50.914

Table A.4: Motion estimation and optimized best SAD

A.3 Fully Parallel OpenMP Tables

These are execution times for our fully parallelized OpenMP executions.

Foreman	**Static**	**Dynamic**	**Guided**
1 Thread	3.707	11.856	11.809
2 Threads	2.212	6.379	6.309
4 Threads	1.868	3.618	3.652
8 Threads	1.388	2.393	2.362
16 Threads	1.39	1.716	1.669
32 Threads	1.672	2.376	2.328
64 Threads	1.373	1.686	1.818

Table A.5: Foreman execution times without offload

Tractor	**Static**	**Dynamic**	**Guided**
1 Thread	68.081	247.516	247.521
2 Threads	35.389	127.496	126.506
4 Threads	20.2	67.735	66.823
8 Threads	13.374	40.13	39.093
16 Threads	11.374	24.824	23.208
32 Threads	14.865	26.145	24.58
64 Threads	9.953	22.455	22.316

Table A.6: Tractor execution times without offload

Bagadus	Static	Dynamic	Guided
1 Thread	210.51	811.454	810.457
2 Threads	108.359	411.622	4087.795
4 Threads	57.796	212.052	208.38
8 Threads	35.012	121.161	118.962
16 Threads	27.063	69.202	67.948
32 Threads	30.501	67.774	66.912
64 Threads	21.027	64.933	64.074

Table A.7: Bagadus execution times without offload

Foreman_4K	Static	Dynamic	Guided
1 Thread	259.007	983.146	983.06
2 Threads	135.582	501.539	498.277
4 Threads	75.069	259.947	256.074
8 Threads	47.985	151.267	148.943
16 Threads	39.77	89.886	87.171
32 Threads	42.096	88.972	88.694
64 Threads	32.204	82.652	81.652

Table A.8: Foreman_4k execution times without offload

Foreman	Static	Dynamic	Guided
1 Thread	5.082	13.393	13.428
2 Threads	3.831	8.011	7.867
4 Threads	3.267	5.198	5.074
8 Threads	2.883	3.969	3.925
16 Threads	2.855	3.241	3.386
32 Threads	3.175	4.033	3.819
64 Threads	3.578	3.232	3.261
128 Threads	8.821	3.457	3.959
256 Threads	2.941	3.343	3.442
512 Threads	3.367	3.601	3.633

Table A.9: Foreman with offload

Tractor	Static	Dynamic	Guided
1 Thread	68.379	247.926	247.914
2 Threads	37.661	128.417	127.437
4 Threads	22.43	67.808	67.083
8 Threads	15.223	40.715	39.982
16 Threads	14.92	25.95	25.806
32 Threads	16.421	27.311	27.094
64 Threads	11.901	24.574	24.402
128 Threads	11.631	24.572	24.362
256 Threads	11.595	24.74	24.504
512 Threads	11.883	24.779	24.712

Table A.10: Tractor with offload

Bagadus	Static	Dynamic	Guided
1 Thread	211.414	811.344	811.53
2 Threads	110.081	414.298	409.79
4 Threads	64.861	213.284	210.528
8 Threads	39.831	122.936	121.295
16 Threads	32.508	71.326	69.19
32 Threads	29.509	68.07	67.667
64 Threads	23.721	65.743	64.718
128 Threads	22.59	65.717	64.71
256 Threads	22.318	65.647	65.063
512 Threads	22.531	65.708	65.035

Table A.11: Bagadus with offload

Foreman_4K	Static	Dynamic	Guided
1 Thread	260.318	986.819	984.343
2 Threads	136.703	505.489	501.195
4 Threads	79.11	261.888	259.431
8 Threads	51.135	154.02	153.103
16 Threads	44.197	94.015	92.302
32 Threads	45.309	90.708	89.745
64 Threads	34.042	83.604	82.605
128 Threads	33.406	83.947	83.054
256 Threads	35.27	84.411	83.71
512 Threads	33.683	84.951	84.155

Table A.12: Foreman_4k with offload

A.4 Encoding Times with ME Only

Foreman	Native	Offload
1 Thread	2.832	2.83
2 Threads	1.434	1.434
4 Threads	0.764	0.76
8 Threads	0.437	0.439
16 Threads	0.255	0.429
32 Threads	0.594	0.618
64 Threads		0.376

Table A.13: Foreman encoding per frame - 100 frames

Tractor	Native	Offload
1 Thread	62.772	62.884
2 Threads	31.587	31.934
4 Threads	16.512	16.568
8 Threads	9.503	9.811
16 Threads	4.87	4.879
32 Threads	8.216	8.184
64 Threads		5.494

Table A.14: Tractor encoding per frame - 100 frames

Bagadus	Native	Offload
1 Thread	212.152	217.207
2 Threads	106.777	109.318
4 Threads	54.664	57.138
8 Threads	30.762	31.254
16 Threads	30.372	30.551
32 Threads	30.627	15.948
64 Threads		17.883

Table A.15: Bagadus encoding per frame - 100 frames

Foreman_4k	Native	Offload
1 Thread	251.86	253.231
2 Threads	126.994	127.19
4 Threads	64.962	65.675
8 Threads	36.325	37.952
16 Threads	31.167	20.528
32 Threads	24.263	28.794
64 Threads		20.744

Table A.16: Foreman_4k encoding per frame - 100 frames

A.5 Cilk

These are the execution times for Cilk.

	Foreman	Tractor	Bagadus	Foreman_4k
1 Thread	11.399	249.049	802.719	973.861
2 Threads	6.387	128.227	414.196	501.552
4 Threads	3.886	70.336	222.945	272.561
8 Threads	2.503	41.352	120.693	151.912
16 Threads	2.09	24.721	72.745	97.202
32 Threads	1.838	23.17	67.935	86.821
64 Threads	2.846	25.301	68.964	97.226

Table A.17: Cilk without offload

	Foreman	Tractor	Bagadus	Foreman_4k
1 Thread	68.976	1658.854	5195.592	6538.103
2 Threads	45.956	919.136	2828.585	3619.548
4 Threads	28.463	540.767	1536.767	2180.474
8 Threads	21.327	343.777	892.757	1319.767
16 Threads	17.665	246.058	574.136	930.477
32 Threads	16.037	200.997	402.648	702.666
64 Threads	13.466	165.885	324.708	600.117
128 Threads	16.307	151.119	311.872	579.945
256 Threads	16.359	175.085	315.788	563.936
512 Threads	27.127	217.299	372.156	621.518

Table A.18: Cilk with offload

Foreman	Foreman	Tractor	Bagadus	Foreman_4k
1 Thread	72.37	1528.584	4966.935	6025.113
2 Threads	39.034	770	2559.096	3062.186
4 Threads	22.165	402.165	1304.897	1596.666
8 Threads	14.056	213.468	687.016	833.214
16 Threads	10.245	118.818	368.818	450.926
32 Threads	8.157	70.766	210.807	259.73
64 Threads	7.546	48.614	137.88	167.838
128 Threads	7.554	41.577	115.579	140.961
256 Threads	11.186	51.597	125.937	150.027
512 Threads	19.305	87.948	154.968	184.458

Table A.19: Cilk without shared memory updates

A.6 CUDA Execution Times

CUDA	Foreman	Tractor	Bagadus	Foreman_4k
1x1 blocks	17.435	201.063	637.251	780.698
2x2 blocks	4.729	54.484	164.415	203.926
2x3 blocks	3.282	37.776	111.964	140.276
3x4 blocks	2.549	21.806	59.111	76.211
3x5 blocks	2.268	19.231	49.285	65.916
4x6 blocks	1.927	14.159	33.807	46.163
6x9 blocks	1.82	10.231	21.352	31.752
8x12 blocks	1.842	11.228	24.015	34.61
10x15 blocks	1.783	10.02	21.127	31.32
12x18 blocks	1.824	10.134	20.936	31.572
14x21 blocks	1.782	10.072	20.942	30.918
16x24 blocks	1.963	10.083	20.369	30.477
18x27 blocks	1.737	10.021	19.765	29.764
20x30 blocks	1.82	10.049	20.111	30.345
22x33 blocks	2.231	10.111	19.785	29.845
24x36 blocks	1.863	10.107	19.823	29.735
26x39 blocks	1.789	10.134	19.84	29.816

Table A.20: CUDA execution times. Each block has 16x16 threads

A.7 Encoding Only Measurements

Foreman	Static	Dynamic	Guided
1 Thread	2.832	11.832	11.881
2 Threads	1.434	5.611	5.57
4 Threads	0.764	2.837	2.819
8 Threads	0.437	1.617	1.591
16 Threads	0.255	0.935	0.956
32 Threads	0.594	1.114	1.442

Table A.21: Encoding only without offload

Foreman	Static	Dynamic	Guided
1 Thread	2.83	11.865	11.836
2 Threads	1.434	5.61	5.576
4 Threads	0.76	2.869	2.86
8 Threads	0.439	1.581	1.553
16 Threads	0.429	1.971	0.993
32 Threads	0.618	1.83	2.85
64 Threads	0.376	0.902	0.944
128 Threads	0.437	0.935	0.956
256 Threads	0.538	1.222	1.173
512 Threads	0.692	1.198	1.116

Table A.22: Encoding only with offload

Foreman_4k	Static	Dynamic	Guided
1 Thread	251.86	970.16	970.482
2 Threads	126.994	491.423	487.648
4 Threads	64.962	249.102	246.167
8 Threads	36.325	139.224	137.143
16 Threads	31.167	74.998	75.518
32 Threads	24.263	72.553	71.763

Table A.23: Encoding only without offload on Foreman_4k

Foreman_4k	Static	Dynamic	Guided
1 Thread	253.231	974.303	836.706
2 Threads	127.19	492.293	418.512
4 Threads	65.675	250.191	209.959
8 Threads	37.952	140.111	117.131
16 Threads	30.528	75.991	76.785
32 Threads	28.794	72.551	71.949
64 Threads	20.744	71.631	70.798
128 Threads	19.641	71.476	70.624
256 Threads	19.5	71.563	70.662
512 Threads	19.805	71.625	71.1

Table A.24: Encoding only with offload on Foreman_4k

Appendix B

Code Details

This appendix contains a list of encoding optimization done in the thesis.
This will not include the full code, just the important optimizations.

B.1 OpenMP

Code Example B.1: The full c63_common for OpenMP after optimization

```
1  struct c63_common
2  {
3    int width, height;
4    int ypw, yph, upw, uph, vpw, vph;
5
6    int padw[COLOR_COMPONENTS], padh[COLOR_COMPONENTS];
7
8    int mb_cols, mb_rows;
9
10   uint8_t qp; // Quality parameter
11
12   int me_search_range;
13
14   uint8_t quanttbl[COLOR_COMPONENTS][64];
15
16   int framenum;
17
18   int keyframe_interval;
19   int frames_since_keyframe;
20
21   struct entropy_ctx e_ctx;
22
23   //Refframe (we only need recons)
24   uint8_t *ref_recons_Y;
25   uint8_t *ref_recons_U;
26   uint8_t *ref_recons_V;
27
28   //Curframe (We do not use recons on current frame)
29   uint8_t *curr_orig_Y;
30   uint8_t *curr_orig_U;
31   uint8_t *curr_orig_V;
32   uint8_t *curr_predicted_Y;
33   uint8_t *curr_predicted_U;
34   uint8_t *curr_predicted_V;
35
36   // Difference between original image and predicted frame
37
38   int16_t *curr_residuals_U;
39   int16_t *curr_residuals_V;
40   int16_t *curr_residuals_Y;
41
42   struct macroblock *curr_mbs_Y;
43   struct macroblock *curr_mbs_U;
44   struct macroblock *curr_mbs_V;
45
46   int curr_keyframe;
47 };
```

Code Example B.2: Optimizing SAD calculations

```
__declspec(target(mic:0)) static void me_block_8x8(struct c63_common *cm, int mb_x,
    int mb_y, uint8_t *orig, uint8_t *ref, int color_component)
{
  struct macroblock *mb = 0;
  if(color_component == 0) {
    mb = &cm->curr_mbs_Y[mb_y*cm->padw[color_component]/8+mb_x];
  } else if(color_component == 1) {
    mb = &cm->curr_mbs_U[mb_y*cm->padw[color_component]/8+mb_x];
  } else {
    mb = &cm->curr_mbs_V[mb_y*cm->padw[color_component]/8+mb_x];
  }

  int range = cm->me_search_range;

  /* Quarter resolution for chroma channels. */
  if (color_component > 0) { range /= 2; }

  int left = mb_x * 8 - range;
  int top = mb_y * 8 - range;
  int right = mb_x * 8 + range;
  int bottom = mb_y * 8 + range;

  int w = cm->padw[color_component];
  int h = cm->padh[color_component];

  if (left < 0) { left = 0; }
  if (top < 0) { top = 0; }
  if (right > (w - 8)) { right = w - 8; }
  if (bottom > (h - 8)) { bottom = h - 8; }

  int x, y;

  int mx = mb_x * 8;
  int my = mb_y * 8;

  int sad[(bottom-top)*(right-left)];
  int best_sad = INT_MAX;
#pragma omp parallel for schedule(SCHEDULE)
  for (y = top; y < bottom; ++y)
  {
    for (x = left; x < right; ++x)
    {
      sad_block_8x8(orig + my*w+mx, ref + y*w+x, w, &sad[(y-top)*(right-left)+x-left]);
    }
  }
  best_sad = __sec_reduce_min_ind(sad[0:(right-left)*(bottom-top)]);
  mb->mv_x = (best_sad%(right-left)) - mx + left;
  mb->mv_y = (best_sad/(right-left)) - my + top;
  mb->use_mv = 1;
}
```

B.2 Cilk

Code Example B.3: The looped reading of a frame

```
1  size_t read_file(char *buf, int amount_to_read, FILE* file) {
2    int read_so_far = 0;
3    int max_read = 524287;
4    size_t len = 0;
5    while (amount_to_read) {
6      if(amount_to_read > max_read) {
7        len += fread(buf+read_so_far, 1, max_read, file);
8        read_so_far += len - read_so_far;
9        amount_to_read -= max_read;
10     } else {
11       len += fread(buf+read_so_far, 1, amount_to_read, file);
12       read_so_far += len - read_so_far;
13       amount_to_read -= amount_to_read;
14     }
15   }
16   return len;
17 }
```

B.3 CUDA

Code Example B.4: The method executing our CUDA kernels

```
dim3 blocks(BLOCKY,BLOCKX);
dim3 threads(16,16);
void c63_encode_image(struct c63_common *cm, yuv_t *image) {
  /* Advance to next frame */
  set_frame(cudaCm, image);
  //Encode
  cudaDeviceSynchronize();
  //We use 3 encode methods to sync between parts
  cuda_encode1<<<blocks,threads>>>(cudaCm, image);
  cudaDeviceSynchronize();
  cuda_encode2<<<blocks,threads>>>(cudaCm, image);
  cudaDeviceSynchronize();
  cudaMemset(cudaCm->curframe->recons->Y, 0, cm->ypw * cm->yph * sizeof(uint8_t));
  cudaMemset(cudaCm->curframe->recons->U, 0, cm->upw * cm->uph * sizeof(uint8_t));
  cudaMemset(cudaCm->curframe->recons->V, 0, cm->vpw * cm->vph * sizeof(uint8_t));
  cudaDeviceSynchronize();
  cuda_encode3<<<blocks,threads>>>(cudaCm, image);
  cudaDeviceSynchronize();

  cudaMemcpy(cm->curframe->residuals->Ydct, cudaCm->curframe->residuals->Ydct,
    cm->ypw * cm->yph* sizeof(int16_t), cudaMemcpyDeviceToHost);
  cudaMemcpy(cm->curframe->residuals->Udct, cudaCm->curframe->residuals->Udct,
    cm->upw * cm->uph* sizeof(int16_t), cudaMemcpyDeviceToHost);
  cudaMemcpy(cm->curframe->residuals->Vdct, cudaCm->curframe->residuals->Vdct,
    cm->vpw * cm->vph* sizeof(int16_t), cudaMemcpyDeviceToHost);

  cudaMemcpy(cm->curframe->mbs[Y_COMPONENT], cudaCm->curframe->mbs[Y_COMPONENT],
    cm->mb_rows * cm->mb_cols * sizeof(struct macroblock), cudaMemcpyDeviceToHost);
  cudaMemcpy(cm->curframe->mbs[U_COMPONENT], cudaCm->curframe->mbs[U_COMPONENT],
    cm->mb_rows/2 * cm->mb_cols/2 * sizeof(struct macroblock), cudaMemcpyDeviceToHost);
  cudaMemcpy(cm->curframe->mbs[V_COMPONENT], cudaCm->curframe->mbs[V_COMPONENT],
    cm->mb_rows/2 * cm->mb_cols/2 * sizeof(struct macroblock), cudaMemcpyDeviceToHost);
  cudaDeviceSynchronize();
}
```

Code Example B.5: The first encoding method CUDA

```
__global__ void cuda_encode1(struct c63_common *cm, yuv_t *image)
{
  if(cm->framenum == 0) cm->curframe->keyframe = 1;
  /* Check if keyframe */
  if (!cm->curframe->keyframe)
  {
    /* Motion Estimation */
    c63_motion_estimate(cm);
    __syncthreads();
    /* Motion Compensation */
  }
}
```

Code Example B.6: The second encoding method for CUDA

```
__global__ void cuda_encode2(struct c63_common *cm, yuv_t *image){
  if (!cm->curframe->keyframe)
    c63_motion_compensate(cm);
}
```

Code Example B.7: The third encoding method CUDA

```
__global__ void cuda_encode3(struct c63_common *cm, yuv_t *image){
  /* DCT and Quantization */
  dct_quantize(cm->curframe->orig->Y, cm->curframe->predicted->Y, cm->padw[Y_COMPONENT],
      cm->padh[Y_COMPONENT], cm->curframe->residuals->Ydct,
      cm->quanttbl[Y_COMPONENT]);

  dct_quantize(cm->curframe->orig->U, cm->curframe->predicted->U, cm->padw[U_COMPONENT],
      cm->padh[U_COMPONENT], cm->curframe->residuals->Udct,
      cm->quanttbl[U_COMPONENT]);

  dct_quantize(cm->curframe->orig->V, cm->curframe->predicted->V, cm->padw[V_COMPONENT],
      cm->padh[V_COMPONENT], cm->curframe->residuals->Vdct,
      cm->quanttbl[V_COMPONENT]);
  /*}

    __global__ void cuda_encode4(struct c63_common *cm, yuv_t *image){*/
  __syncthreads();
  /* Reconstruct frame for inter-prediction */
  dequantize_idct(cm->curframe->residuals->Ydct, cm->curframe->predicted->Y,
      cm->ypw, cm->yph, cm->curframe->recons->Y, cm->quanttbl[Y_COMPONENT]);
  dequantize_idct(cm->curframe->residuals->Udct, cm->curframe->predicted->U,
      cm->upw, cm->uph, cm->curframe->recons->U, cm->quanttbl[U_COMPONENT]);
  dequantize_idct(cm->curframe->residuals->Vdct, cm->curframe->predicted->V,
      cm->vpw, cm->vph, cm->curframe->recons->V, cm->quanttbl[V_COMPONENT]);
  /* Function dump_image(), found in common.c, can be used here to check if the
     prediction is correct */
  if(threadIdx.x == 0 && threadIdx.y == 0 && blockIdx.x == 0 && blockIdx.y == 0) {
    ++cm->framenum;
    ++cm->frames_since_keyframe;
  }
  //Having this at start means we must sync after it, we skip that now
  if(threadIdx.x == 0 && threadIdx.y == 0 && blockIdx.x == 0 && blockIdx.y == 0) {
    if (cm->framenum == 0 || cm->frames_since_keyframe == cm->keyframe_interval)
      {
        cm->curframe->keyframe = 1;
        cm->frames_since_keyframe = 0;
      }
    else { cm->curframe->keyframe = 0; }
  }
}
```

Code Example B.8: The setup for SAD calculations in CUDA

```
__device__ static void me_block_8x8(struct c63_common *cm, int mb_x,
    int mb_y, uint8_t *orig, uint8_t *ref, int color_component)
{
 struct macroblock *mb =
   &cm->curframe->mbs[color_component][mb_y*cm->padw[color_component]/8+mb_x];

  int range = cm->me_search_range;

  /* Quarter resolution for chroma channels. */
  if (color_component > 0) { range /= 2; }

  int left = mb_x * 8 - range;
  int top = mb_y * 8 - range;
  int right = mb_x * 8 + range;
  int bottom = mb_y * 8 + range;

  int w = cm->padw[color_component];
  int h = cm->padh[color_component];

  if (left < 0) { left = 0; }
  if (top < 0) { top = 0; }
  if (right > (w - 8)) { right = w-8; }
  if (bottom > (h - 8)) { bottom = h-8; }

  int x, y, i;

  int mx = mb_x * 8;
  int my = mb_y * 8;

  //This is the boundary of our search range. If we have a limited
  //search range it will apply to just the the area as if the lower and
  //right most parts are not there
  int __shared__ sad[32*32];

  for (i = threadIdx.y * blockDim.x + threadIdx.x;
        i < (bottom - top) * (right - left);
        i += blockDim.x * blockDim.y)
    {
      y = i/(right - left) + top;
      x = i%(right - left) + left;
      sad_block_8x8(orig + my*w+mx, ref + y*w+x, w, &sad[(y-top)*(right-left)+x-left]);
    }
  __syncthreads();
  reduce_best_sad(mb, mx, my, top, bottom, right, left, sad);

  mb->use_mv = 1;
```

Code Example B.9: The parallel reduction to find the lowest value

```
1   __device__ void reduce_best_sad(struct macroblock *mb, int mx,
2                                   int my, int top, int bottom, int right,
3                                   int left, int* values) {
4   int half = ((bottom-top)*(right-left))/2; //This is the middle of the array
5   int i;
6   int __shared__ ind[32*32];
7
8
9   for(i = threadIdx.y * blockDim.x + threadIdx.x;
10      i < 32*32;
11      i += blockDim.x * blockDim.y) {
12    ind[i] = i;
13  }
14  __syncthreads();
15  for(; half > 0; half>>=1) {
16    for(i = threadIdx.y * blockDim.x + threadIdx.x;
17        i < half;
18        i += blockDim.x * blockDim.y) {
19      if(values[i] >= values[i+half]) {
20        values[i] = values[i+half];
21        ind[i] = ind[i+half];
22      }
23    }
24  }
25  __syncthreads();
26  if(threadIdx.x == 0 && threadIdx.y == 0) {
27    mb->mv_x = (ind[0]%(right-left)) - mx + left;
28    mb->mv_y = (ind[0]/(right-left)) - my + top;
29  }
30 }
```

Appendix C

Codec63

Codec63 can be found at https://*bitbucket.org/mpg_code/inf5063-codec63* and is an open source encoder. The easiest way to download and use the codec is through git. Using git the code can be downloaded with either clone or fetch. `git clone git@bitbucket.org:mpg_code/inf5063-codec63.git` or `git fetch git@bitbucket.org:mpg_code/inf5063-codec63.git`.

The encoder uses a makefile for automated building. `make c63enc` builds the encoder while `make c63dec` builds the decoder.

The encoder requires parameters about height, width, input file and output file to run. An example of how it is run on the foreman.yuv video is `./c63enc -h 288 -w 352 -o foreman.c63 foreman.yuv` which will encode the video. The video can be decoded through `./c63dec foreman.c63 foreman.yuv`.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. "The Data Locality of Work Stealing". In: *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*. Bar Harbor, Maine, USA: ACM, 2000, pp. 1–12.

[2] Siddhant Ahuja. *Sum of Absolute Differences (SAD)*. Accessed: 2016-07-14. 2010. URL: https://siddhantahuja.wordpress.com/tag/sum-of-squared-differences/.

[3] Alexa. *The top 500 sites on the web*. Accessed: 2016-07-14. 2016. URL: http://www.alexa.com/topsites.

[4] AmandaS. *Building a Native Application for Intel® Xeon Phi™ Coprocessors*. Accessed 2016-07-09. URL: https://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors.

[5] Anonymous. *DCT-8x8*. Accessed: 2016-04-25. n.d. URL: https://upload.wikimedia.org/wikipedia/commons/2/24/DCT-8x8.png.

[6] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2006.

[7] Eduard Ayguadé et al. "Is the Schedule Clause Really Necessary in OpenMP?" In: *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools, WOMPAT 2003 Toronto, Canada, June 26–27, 2003 Proceedings*. Ed. by Michael J. Voss. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 147–159.

[8] Robert D. Blumofe and Charles E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748.

[9] Ian Buck. *GPU Computing with NVIDIA CUDA*. Accessed: 2016-06-09. 2007. URL: http://www-hep.uta.edu/~stradlin/GPGPU/International%20Conference%20on%20Computer%20Graphics%20and%20...%202007%20Buck.pdf.

[10] Ian Buck. *The Evolution of GPUs for General Purpose Computing*. Accessed 2016-07-14. 2010. URL: http://www.nvidia.com/content/GTC-2010/pdfs/2275_GTC2010.pdf.

[11] Ian Buck et al. “Brook for GPUs: Stream Computing on Graphics Hardware”. In: *ACM Trans. Graph.* 23.3 (Aug. 2004), pp. 777–786.

[12] J Mark Bull. “Measuring synchronisation and scheduling overheads in OpenMP”. In: *Proceedings of First European Workshop on OpenMP*. Vol. 8. 1999.

[13] Andrew Burnes. *Introducing the GeForce GTX TITAN*. Accessed 2016-06-09. URL: http://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-titan.

[14] George Chrysos. *Intel® Xeon Phi™ X100 Family Coprocessor - the Architecture*. Accessed: 2016-07-14. 2012. URL: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[15] Cilk. *A brief History of Cilk*. Accessed: 2015-06-29. n.d. URL: https://www.cilkplus.org/cilk-history.

[16] D. E. Comer et al. “Computing As a Discipline”. In: *Commun. ACM* 32.1 (Jan. 1989). Ed. by Peter J. Denning, pp. 9–23.

[17] Håvard Espeland, Preben Olsen, and Håkon Kvale Stensland. *inf5063-codec63*. Accessed: 2016-08-11. 2013. URL: https://bitbucket.org/mpg_code/inf5063-codec63.

[18] Jianbin Fang et al. “Test-driving Intel Xeon Phi”. In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. Dublin, Ireland: ACM, 2014, pp. 137–148.

[19] Jianbin Fang et al. “Test-driving intel xeon phi”. In: *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*. ACM. 2014, pp. 137–148.

[20] Richard Friedman. *What problem does OpenMP solve?* Accessed: 2016-08-04. n.d. URL: http://openmp.org/openmp-faq.html#Problems.

[21] Rupert Goodwins. *Intel unveils many-core Knights platform for HPC*. Accessed 2016-07-14. URL: http://www.zdnet.com/article/intel-unveils-many-core-knights-platform-for-hpc/.

[22] Robert H. Halstead Jr. “MULTILISP: A Language for Concurrent Symbolic Computation”. In: *ACM Trans. Program. Lang. Syst.* 7.4 (Oct. 1985), pp. 501–538.

[23] Intel. *Intel® Xeon Phi™ Coprocessor 3120A*. Accessed 2016-06-13. URL: http://ark.intel.com/products/75797/Intel-Xeon-Phi-Coprocessor-3120A-6GB-1_100-GHz-57-core.

[24] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[25] Hwancheol Jeong et al. “Performance of Kepler GTX Titan GPUs and Xeon Phi System”. In: *arXiv preprint arXiv:1311.0590* (2013).

[26] Khazadum. *OpenMP language extensions*. Accessed: 2015-06-29. 2008. URL: https://commons.wikimedia.org/wiki/File:OpenMP_language_extensions.svg.

[27] E.A. Lee. “The problem with threads”. In: *Computer* 39.5 (May 2006), pp. 33–42.

[28] Michael Masnick, Michael Ho Joyce Hung, and Leigh Beadon. *The sky is rising*. Accessed 2016-07-14. 2014. URL: https://www.ccianet.org/wp-content/uploads/2014/10/Sky-Is-Rising-2014.pdf.

[29] NVIDIA. *CPUvsGPU*. Accessed: 2016-06-07. n.d. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[30] NVIDIA. *CUDA Parallel Computing Platform*. Accessed 2016-06-09. URL: http://www.nvidia.com/object/cuda_home_new.html.

[31] NVIDIA. *Kepler GK110 Whitepaper*. Accessed: 2016-06-07. 2012. URL: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[32] Mike Pound. *JPEG DCT, Discrete Cosine Transform*. Accessed: 2015-06-29. Computerphile. 2015. URL: https://www.youtube.com/watch?v=Q2aEzeMDHMA.

[33] Mike Pound. *JPEG ’files’ & Colour*. Accessed: 2015-06-29. Computerphile. 2015. URL: https://www.youtube.com/watch?v=n_uNPbdenRs.

[34] Rezaur Rahman. *Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.

[35] Austin Roorda and David R Williams. “The arrangement of the three cone classes in the living human eye”. In: *Nature* 397.6719 (1999), pp. 520–522.

[36] P. E. Ross. *Why CPU frequency stalled*. Accessed: 2014-11-23. 2008. URL: http://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled.

[37] Larry Seiler et al. “Larrabee: a many-core x86 architecture for visual computing”. In: *ACM Transactions on Graphics (TOG)*. Vol. 27. ACM. 2008, p. 18.

[38] SimilarWeb. *Website Ranking*. Accessed: 2016-07-14. 2016. URL: https://www.similarweb.com/global.

[39] Håkon Kvale Stensland. *Home Exam 1: Video Encoding on Intel x86 using Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX)*. Accessed: 2016-03-29. 2015. URL: http://www.uio.no/studier/emner/matnat/ifi/INF5063/h15/slides/inf5063-exam-01.pdf.

[40] Håkon Kvale Stensland. *(M-)JPEG*. Accessed: 2016-03-29. 2015. URL: http://www.uio.no/studier/emner/matnat/ifi/INF5063/h15/slides/inf5063-mjpeg-code-figures.pdf.

[41] Håkon Kvale Stensland. “Processing Multimedia Workloads on Heterogeneous Multicore Architectures”. PhD thesis. UiO, 2015.

[42] Kristoffer Robin Stokke, Håkon Kvale Stensland, and Pål Halvorsen. *The Workload, Codec 63*. Accessed 2015-03-25. n.d. URL: https://www.simula.no/file/gtc2015embeddedsystems01p5193webpdf/download.

[43] Vcodex. *H.264 Advanced Video Coding*. Accessed: 2016-07-14. n.d. URL: https://www.vcodex.com/h264-resources/.

[44] Željko Vrba, Paal Halvorsen, and Carsten Griwodz. "A simple improvement of the work-stealing scheduling algorithm". In: *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*. IEEE. 2010, pp. 925–930.

[45] Željko Vrba et al. "Limits of work-stealing scheduling". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2009, pp. 280–299.

[46] Dmitry Vyukov. *Child Stealing*. Accessed: 2015-06-29. 2008. URL: http://www.1024cores.net/home/parallel-computing/concurrent-skip-list/work-stealing-vs-work-requesting.

[47] G. K. Wallace. "The JPEG still picture compression standard". In: *IEEE Transactions on Consumer Electronics* 38.1 (Feb. 1992), pp. xviii–xxxiv.

[48] T. Wiegand et al. "Overview of the H.264/AVC video coding standard". In: *IEEE Transactions on Circuits and Systems for Video Technology* 13.7 (July 2003), pp. 560–576.

[49] Wikimedia Foundation Inc. *Discrete cosine transform*. Accessed: 2016-07-14. n.d. URL: https://en.wikipedia.org/wiki/Discrete_cosine_transform.