IT 11 020

Examensarbete 30 hp
Maj 2011

Adding special-purpose processor support to the Erlang VM

Christofer Ferm

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Adding special-purpose processor support to the Erlang VM

Christofer Ferm

This thesis investigates the possibility to extend the Erlang runtime system such that it can take advantage of special purpose compute units, such as GPUs and DSPs. Further more it investigates if certain parts of an Erlang system can be accelerated with help of these devices.

Handledare: Hans Nilsson
Ämnesgranskare: Mikael Pettersson
Examinator: Anders Jansson
IT 11 020
Sponsor: Erlang Solutions AB

Tryckt av: Reprocentralen ITC

Contents

Acknowledgements **iv**

Overview **vi**

1 Introduction **1**
1.1 Introduction 1
1.2 Approach 1

2 Special-Purpose Processing Units **2**
2.1 Introduction 2
2.2 DSP (Digital Signal Processor) 2
2.2.1 An Introduction to DSPs 2
2.2.2 Architecture 3
2.2.3 How to program a DSP 3
2.2.4 DSP Pros and Cons 4
2.3 GPU (Graphics Processing Unit) 4
2.3.1 Introduction 4
2.3.2 Architecture 4
2.3.3 How to program a GPU 5
2.3.4 GPU Pros and Cons 9
2.4 Conclusion 9

3 Erlang **10**
3.1 Introduction 10
3.2 What is Erlang 10
3.2.1 The History of Erlang 10
3.3 The Erlang Compiler in short 11
3.3.1 Core Erlang 11
3.3.2 The BEAM File Format 11
3.4 The Erlang Virtual Machine 14
3.4.1 The BEAM Loader 14
3.4.2 Processes 14
3.4.3 Scheduling 14
3.5 Parts of Erlang that can be parallelised 14
3.6 Conclusion 15

4 **Test Implementation** **16**
4.1 Introduction . . . 16
4.2 Overview of how it works . . . 16
4.3 The translation module bs2cl . . . 17
4.3.1 Erlang Tokens . . . 17
4.3.2 Finding bitstrings comprehensions in a source file . . . 17
4.3.3 The Actual Translation . . . 18
4.3.4 The Right Part . . . 18
4.3.5 The Left Part . . . 18
4.3.6 Creating the Kernel Head . . . 19
4.3.7 Creating the Kernel Body . . . 19
4.3.8 Putting it All Together . . . 20
4.3.9 The runcl module . . . 20
4.4 Summary . . . 21

5 **Benchmarking** **22**
5.1 Approach . . . 22
5.2 How the Test are Performed . . . 22
5.3 The Test Cases . . . 22
5.3.1 Substitution Cipher . . . 22
5.3.2 Data Generation on Device (lists:seq) . . . 23
5.4 The Test Setup . . . 23
5.4.1 Data sets . . . 23
5.4.2 The Hardware . . . 24
5.4.3 Tools Used . . . 24
5.4.4 The Test Module . . . 24
5.5 The Results . . . 26
5.5.1 The Average Results of 10 Runs . . . 26
5.5.2 Figures . . . 28
5.6 Conclusion . . . 29

6 **Proposal: Extending the Erlang VM** **30**
6.1 introduction . . . 30
6.2 Proposal Overview . . . 30
6.3 Extending the Compiler . . . 30
6.4 Extending the Virtual Machine . . . 31
6.5 To Sum Up . . . 31

7 **Conclusion and Discussion** **32**
7.1 Conclusion . . . 32
7.2 What I Have Learned . . . 32
7.3 What I Have Accomplished . . . 33
7.4 Problems Along The Way . . . 33

8 **Future Work** **34**
8.1 Introduction . . . 34
8.2 Extending the Test Implementation . . . 34
8.3 Looking at More Things to Improve . . . 34
8.4 Extending the Virtual Machine . . . 34
8.5 Supporting More Special Purpose Compute Units . . . 35

9 Related Work **36**
9.1 Introduction 36
9.2 OpenCL Binding for Erlang 36
9.3 Beam Emulator in Erlang 36
9.4 Erjang 36
9.5 Erlang-Embedded 36

References **37**

Acknowledgements

Other than my supervisors, I would like to thank Tony Rogvall and Björn Gustavsson for giving me input and pointing me in the right direction. I also like to thank Fredrik Bjurefors for his feedback on my report.

Overview

Chapter 1:Introduction

I will in this chapter introduce the reader to what this thesis is all about, I will also answer questions like how the thesis came to be and what it hopes to achieve. I will also in this chapter briefly explain the approach used to produce this thesis.

Chapter 2: Special-Purpose Processing Units

This chapter will explain what a special purpose processing unit is. I will then focus on two common special purpose processing units and briefly explain the architecture of these devices and I how to program these special purpose processing units. Further more I will also look into which of these devices that would be most suitable for a test implementation.

Chapter 3: Erlang

The reader will be introduced to what Erlang is in this chapter, further more I will explain parts of Erlang that I have learned by looking at the Erlang runtime system. I will also present a part of Erlang that could be run on a special purpose processing unit.

Chapter 4: Test Implementation

A proof of concept test Implementation is presented , it is not an actual virtual machine extension but an Erlang module showing how you could pre-process an Erlang source file. Showing that you can accelerate certain parts of it using a special purpose processing unit.

Chapter 5: Benchmarking

Benchmarks will be provided to show that the test implementation in chapter 4 can outperform the current Erlang implementation.

Chapter 6: Proposal Extending the Erlang VM

In this chapter a proposal is presented. It shows in an abstract way how the test implementation could be converted into an extension of the Erlang compiler and virtual machine.

Chapter 7: Conclusion and Discussion

In this chapter my conclusions will be presented and a short discussions of what I have accomplished in this thesis will also be provided.

Chapter 8: Future Work

What I plan to do in the future and what is left to do before I can build a complete extension of the Erlang runtime system will be presented and discussed in this chapter.

Chapter 9: Related Work

Short descriptions of a number of related projects will be presented in this chapter.

1 Introduction

1.1 Introduction

This thesis came to be as a part of the Erlang embedded project [10] where we are looking to run Erlang on embedded devices, some of these devices such as the beagleboard [4] have special purpose compute units (some times also referred to as special purpose processing units) on them just like most personal computers do. These sepecial purpose compute units are designed to perform specialised tasks, e.g, a GPU do graphics generation and a DSP do signal processing. This means that these compute units are idle most of the time and are basically doing nothing. This was when the idea for this thesis came to be, what if one could utilise these special purpose compute units for something else, maybe to improve the performance of the Erlang runtime environment for example. The goal of this thesis is to investigate the possibilities of adding special purpose processor support to the Erlang runtime system. I will in this thesis try to answer questions like what can we use these special purpose compute units for, how can they improve Erlang and will it work in reality.

1.2 Approach

I started off by thinking about what I needed to know in order to do this thesis and I came up with four different sub areas. First I need some basic understanding of how the architecture of these special purpose compute units look like and what kind of problems and computations they are good at solving. Next I need to know more about how the Erlang runtime environment works and I also need to find parts of Erlang that I can map on these special purpose compute units. Next I need to show that one can gain performance from utilising these special purpose compute units. And finally propose a way to extend the Erlang runtime environment to make this a reality. This has been the approach of this thesis and I even though it as of yet has not yield a complete extension of the Erlang runtime environment. I hope to be able to continue to work on this in the future and also that others can benefit from my work if they feel like they want to take this project further.

2 Special-Purpose Processing Units

2.1 Introduction

There are several different types of compute units in modern computer systems. The obvious one is the CPU (Central Processing Unit), but you also have other compute units like the Graphics Processing Unit (GPU) or the Digital Signal Processor (DSP). These are so called special purpose compute units designed to perform specialised tasks to improve performance. This also means that they are idle a lot more than a CPU. So can one use these compute units for something else than what they were initially intended for. So that their full potential can be utilised and increase performance, e.g.,in an Erlang system. In order to answer these questions, we need to know more about these devices. This section will look into the architecture of these compute units to find out what kind of tasks they are good at performing. This chapter will also look into how to program these compute units in order to take advantage of them. To limit this section a bit, I will focus merely on GPUs and DSPs but the reader should know that there are also other types of special purpose compute units that also might be quite common even though not as common as the ones mentioned above.

2.2 DSP (Digital Signal Processor)

2.2.1 An Introduction to DSPs

A DSP (Digital Signal Processor) [18, 27] is a type of microprocessor that was designed for fast mathematical manipulation of digital signals. The DSP has evolved from the ASP (Analogue Signal Processor) [18, 27] which used analogue hardware to transform physical signals. One of the benefits with using DSPs over ASPs is that it gives the same result for the same signal in different types of environments and weather conditions, which you can't get with an ASP also two DSPs built with the same hardware will produce exactly the same result for the same input, this is not true for an ASP since the result will have a variation about 1% for two ASPs built with exactly the same components.

If you compare a DSP to a GPP (General Purpose Processor) you will discover some main differences, for example DSPs tend to run only one program at a time whilst a GPP may run many. There is no virtual memory or protection in a DSP and DSPs mostly run with hard real-time constraints. The DSP is optimised for signal processing and therefore very well suited to run algorithms like

- IRR (Infinite Impulse Response) Filters
- FIR (Finite Impulse Response) Filters
- FFT (Fast Fourier Transforms)

2.2.2 Architecture

A typical DSP architecture [12] is a so called SIMD (Single Instruction Multiple Data) architecture, this means that you run the same instruction for multiple data simultaneously. The most common SIMD architectures are those found in generic processors, e.g, Intel's MMX and SSE technology, AMD's 3DNOW! or AltiVec for IBM PowerPC. SIMD architectures require more work from programmers in order to be efficient since all problems that are implemented needs to be parallel in order to be efficient (i.e. doing the same thing for a large set of data simultaneously). A typical DSP architecture is also superscalar which means that architecture is using a form of instruction level parallelism this instruction level parallelism is per processor, meaning that each processor can perform several instructions in one clock cycle. In combination with a superscalar architecture most DSPs also use VLIW (Very Long Instruction Word) technology. VLIW is method of packing more than one instruction into a single word, hence creating a Very Long Instruction Word. VLIW is used to reduce frequency requirements in the program buses, which means that less bandwidth will be used since instructions are packed together into one word.

If we should look at a typical modern DSP core they often consist of one or more dedicated multiplier or a combined multiplier-accumulate unit (MAC) and since DSP cores are SIMD architectures there are more than one independent execution unit. For example you will in addition to the ones mentioned above have an ALU (Arithmetic Logic Unit) and a shifter. It is important to be able to do processing fast since DSPs need to filter signals in real time. Memory speeds are also important that is why most DSPs use two separate memory buses. A typical DSP have a program memory that is connected to a program cache and then connected to the DSP core and it also has a data memory connected on its own bus to the DSP core. So to sum up, a conventional DSP processors [12] often contain something like dual memory buses with 16bit word size, a 16x16 multiplier, an ALU, a bit manipulation unit and two accumulators. But there are also enhanced versions which could contain more for example in addition to the conventional above an enhanced DSP could have an additional 16x16 multiplier, a dedicated adder, eight accumulators and dual memory buses with 32bit word size instead of 16bit. So as we can see from the architecture, a DSP is capable of performing more than one instruction per cycle and it can perform these instructions for multiple data fragments in parallel. The DSP thus gives you good performance for your money and with low power consumption and this is probably the main reason why DSPs are so widely used.

2.2.3 How to program a DSP

The most common ways of programming DSPs today is through assembly language and in some cases C. But the assembly languages and C APIs may differ between different manufacturers. This makes it difficult to support DSPs from a wide range of manufacturers and therefore the DSP might not be the best candidate for a test implementation. There is also important to do code optimisations on the assembly level to be able to take advantage of the full potential of the hardware. It should also in theory be possible to use OpenCL (see OpenCL under Section 2.3.3) to program DSPs, however I have not been able to find any DSPs that supports OpenCL today, but it could possibly be supported in the future.

2.2.4 DSP Pros and Cons

Pros:

- High performance (real-time).
- Cost effective
- Low power consumption

Cons:

- Slow bus speeds (PCI bus)
- Harder to program than a GPP.
- Lack of good programming APIs that supports different DSPs.

2.3 GPU (Graphics Processing Unit)

2.3.1 Introduction

A GPU is a special purpose microprocessor designed to offload and accelerate graphics rendering from the CPU. The GPU is extremely efficient at manipulating graphics and it is significantly better than a GPP (General Purpose Processor) for some complex algorithms [19]. This is due to that a GPU has a highly parallel architecture, which will be explained more in detail in the section below. GPUs are today very common, you find them in virtually any personal computer or workstation, a lot of different embedded systems, e.g., mobile phones. You also find them in video game consoles and other multimedia devices. This means that taking advantage of GPUs could be very useful thanks to the fact that GPUs exists in so many different devices.

2.3.2 Architecture

A typical GPU architecture [11] is, as mentioned, a highly parallel architecture, due to the fact that they are built up of multiple cores. These cores are not like the cores in a regular multi-core processor, in a regular multi-core processor each core is very powerful. In a GPU each core is more light weight and not as powerful, hence single-thread performance is poor but the GPU makes up for this by having many cores. A typical GPU consist of several multi-processors these multi-processors contain so called streaming-processors these could be considered as processor cores. These streaming processors are as mentioned a little bit different from a regular CPU core. Let's look at the difference between a CPU and a GPU processor core. A CPU core would probably contain something like an instruction fetch/decode unit, an out of order execution control unit, a branch predictor, a memory pre-fetcher, a data cache, an ALU and an execution context (registers). A typical GPU core would only contain something like an instruction fetch/decode unit, an execution context and several ALUs. The GPU core has several ALUs while the CPU core only has one. Having more than one ALU makes it possible to perform the same operation on multiple data fragments simultaneously. This means that the GPU uses a type of SIMD (Single Instruction Multiple Data) architecture. We also see here that a GPU core is missing some of the units from the CPU for example the branch predictor and memory pre-fetcher. These are things that make a single instruction execute faster which is needed in a CPU core since it only do one instruction for one data fragment at a time. Doing the same instruction for multiple data fragments simultaneously can cause problems when branches are used.
Take for example a normal if statement something like `if (X < 0) { X+1 }else{ X-1 }` consider a GPU core with 4 ALUs and the data `[-1,2,3,4]`. What would happen is that when the first

branch is taken we can only do that instruction for 1 data fragment of the list above. Because the others don't full fill the condition for this branch. This will cause 3 out of the 4 ALUs to stall, thus this core is only executing for one data fragment at this cycle. In the next cycle it will do next branch of the if statement and this time 1 out of the 4 ALUs will stall. This means that having branches in code that is supposed to run on the GPU is bad and making processing slow. So one should try to avoid using branches in code that is supposed to run on the GPU.

To wrap this section up I thought that I should compare two different GPUs from two different manufacturers. I thought we should take a look at one GPU from Nvidia and one from ATI. The GPUs I have chosen are the Nvidia GTX285 [23] and the ATI HD4890 [1] which are about equal in performance. We'll start off by introducing the two GPUs, let's start with the Nvidia one. Nvidia markets this GPU as having 240 processor cores what this means is that this GPU has 30 streaming multi-processors each containing 8 streaming processors which adds up to a total of 240 processor cores. This makes the GTX285 a quite powerful GPU. The HD4890 is being marketed as having 800 processing units, what this means in reality is that it has 10 multi-processors with 16 streaming processors which each contains 5 ALUs, thus adding to a total of 800 processing units in ATI marketing terms. If we should use Nvidia terminology to compare these two cards the GTX285 would have 240 processor cores and the HD4890 would have 160 processor cores. From this it would appear that the ATI GPU is slower, but the numbers above do not tell the entire story. In order to work this out we need to look at a GTX285 core and the HD4890 core. The GTX285 core has two ALUs (one multiply-add unit and one multiply unit) while the HD4890 has 5 ALUs (5 multiply-add units). This means that the HD4890 GPU has less cores than the GTX285 but each core is more powerful. To wrap this up we can do some maximum performance calculations. The GTX285 have a clock speed of 1.3Ghz it has 30 multi-processors with 8 streaming processors with two ALUs one multiply-add unit capable of 2 flops (floating point operations) and one multiply unit capable of 1 flop this results in `30*8*(2+1)*1.3= 933` GigaFlops and the HD4890 has clock frequency of 750Mhz and 10 streaming multi-processors with 16 streaming processors each containing 5 ALUs, that is 5 multiply-add units capable of 2flops resulting in `10*16*(5*2)*0.75 = 1200` GigaFlops. This makes the HD4890 faster in theory but I would say these two cards will be pretty equal in performance in the average case.

2.3.3 How to program a GPU

There are several ways of programming a GPU and there exists several technologies. The most commonly known would probably be DirectX (or more accurately Direct3D) and OpenGL. These are used by graphics programmers to generate graphics in order to create things like 3D games or CAD programs. Both DirectX and OpenGL is probably known to most computer gamers in the world, however these technologies are mainly built for generating 3D graphics and does not fit that well for general purpose problems (with the exception of the newer versions of DirectX). However there are other frameworks like CUDA, AMD APP (Accelerated Parallel Processing), DirectCompute (actually a part of DirectX) and OpenCL. These might not be as known as DirectX and OpenGL, they are however not designed with just graphics generation in mind, these technologies also allows you to do general purpose processing on GPUs. I will in this section briefly explain the different technologies and what they do, but I will mainly focus on those designed for general purpose processing. I will also provide some pros and cons of each technology.

Microsoft DirectX (Direct3D and DirectCompute)

Microsoft DirectX [22] is really a collection of APIs created to handle multimedia programming, especially game and video programming but now it also includes general purpose processing. The X in DirectX indicates that it is a collection of APIs like DirectDraw, DirectMusic, DirectPlay, DirectSound and Direct3D and in the newer versions of DirectX you also find DirectCompute. Direct3D is the part of DirectX that do computations on the GPU. This is the API that handles graphics generation on the GPU but there is also DirectCompute which is a part of Direct3D and allows you to do general purpose processing on GPUs under Windows Vista and Windows 7. So to sum up DirectX is a collection of APIs for multimedia and game programming and since DirectX 11 it is also possible to do general purpose processing on GPUs. However DirectX is made by Microsoft and thus only available on Microsoft platforms (i.e. Windows), this also means that DirectX and all that it includes are proprietary technologies owned by Microsoft. The biggest draw back with DirectX is that it is limited to Windows.

Pros:

- Contain APIs for multimedia,game programming and general purpose processing on GPUs.
- Widely used (at least for graphics generation).
- Support for many languages.

Cons:

- Micrsoft platforms (Windows) only
- Proprietary technology

OpenGL (Open Graphics Library)

OpenGL [17] is an open specification defining a cross-language , cross-platform programming API that is used to create 2D and 3D graphics. OpenGL was initially developed by Silicon Graphics in 1992 and are now maintained by the non-profit technology consortium Khronos Group. OpenGL does pretty much the same thing as Microsoft Direct3D , but has several benefits over Direct3D. For example it is an open specification and thus is a non proprietary technology and it is cross-platfrom. This means that OpenGL is available on, e.g, Windows, Mac OS, Linux and on mobile devices using Android or iOS. OpenGL is good for graphics generation and are used in for example computer game programming and in CAD (Computer Aided Design) programming. However even though you can abuse OpenGL to do general purpose processing, OpenGL is not designed for general purpose processing (that is a drawback compared to DirectX). This makes it difficult to do general purpose processing using OpenGL. Thus OpenGL is not the way to go for a test implementation.

Pros:

- Good API for graphics generation
- Widely used
- Cross-platform
- Cross-language
- An open specification

Cons:

- Specialised for graphics generation, thus not good for general prurpose processing.

Nvidia's CUDA Framework

CUDA [24] is Nvidias parallel computing architecture and is the computing engine in Nvidias GPUs, CUDA also gives developers the possibility to get access to the computing engine inside the GPU through variants of standard programming languages, e.g., C for CUDA (which is C with some extensions and certain restrictions) but CUDA is available for other languages as well. This allow developers to do general purpose processing on Nvidia GPUs. CUDA are at present available for different operating systems for example Windows, Linux and Mac OS. but do for understandable reasons only support Nvidias GPUs. CUDA also have support for OpenCL which means that any CUDA enabled GPUs can run OpenCL as well. Since OpenCL can be run on top of CUDA I will not go in to how you program a GPU using CUDA. How to program a GPU will be covered in the OpenCL section.

Pros:

- API for general purpose processing
- Widely used
- Support for different operating systems
- Cross-language

Cons:

- Only supports Nvidia GPUs
- Proprietary technology

AMD APP (Accelerated Parallel Processing)

AMD APP [2] is actually the new version of ATI Stream which basically is a set of software and hardware technologies that make it possible for (AMD) GPU cores to work together with normal x86 CPU cores to accelerate application in other ways than just graphics generation. So basically one can say that it is a way to do general purpose processing on both GPU and CPU. AMD have also built in OpenCL support in APP which basically means that every AMD (ATI) GPU that supports APP will be able to run OpenCL as well. APP is available for many different operating systems, e.g., Linux, Mac OS and Windows. Since you can use OpenCL in APP I will not go into how the programming of a GPU is done in this section, this will be covered in the OpenCL section below.

Pros:

- An API for general purpose processing on heterogeneous devices.
- OpenCL support
- Available for many different operating systems

Cons:

- Only supports some of AMD (ATI) GPUs
- Proprietary technology

OpenCL (Open Computing Language)

OpenCL [15] is an open standard for parallel programming of heterogeneous systems. What this means is basically that it is designed to utilise multipel compute units within a computer system, e.g, to utilise devices like CPUs, GPUs,DSPs and so on. The OpenCL specification was proposed by Apple but has been developed in conjunction with many other companies like Nvidia, AMD, Intel, Ericsson, Nokia and many many more [16]. The OpenCL specification is maintained by the Khronos Group just like OpenGL is. OpenCL is as mentioned actually a specification which allows for any manufacturer to support OpenCL. OpenCL is today available for Linux, Windows and Mac OS. It is also supported by different manufacturers which means that you can run OpenCL on for example GPUs from both Nvidia and ATI. OpenCL stands for Open Computing Language and consist of a C99 based language in which one writes the programs in or in OpenCL terms called kernels which are then later run a compute device. To make it easier to understand I thought I might cover some of the OpenCL terminology. A device that do the actual computations is called a compute device, the program that is run on that device is called a kernel. A thread that is run on a compute device is called a work item and is part of a group of work items called a work group. A host program is the program you write to be able to enqueue your kernels on to these compute devices. A host program is written in a normal programming language often C or as I have done written it in Erlang using the OpenCL binding for Erlang [26]. The kernels are always written in OpenCL which is as mentioned earlier a C99 based language with some minor modifications. For example you have additional built in vector types like float4 for example (see specification for all vector types [15]).

To get the reader more acquainted with OpenCL I thought I would present and explain a hello world OpenCL kernel.

```
__kernel void hello(__global int *input,
                    __global int *output)
{

   int i = get_global_id(0);

   output[i]=input[i]*input[i];

}
```

What this kernel do is that it calculate the square of the elements in the input array `input` and store the result in the output array `output`. The line `int i = get_global_id(0);` fetches the current id, that is which work-item this is so we operate on the correct data in the arrays. the `__global` declaration means that this is shared data between all work-items. When you execute the kernel slightly simplified this is what happens, first the kernel is compiled to device specific assembly code then the input array is copied from RAM (Main memory) and stored on the device memory, then the execution of the kernel begins, it will calculate the square of each element and write the result to the output array which is stored in device memory. When the kernel has finished the output array is read from device memory back to RAM. The input and output arrays need to be specified in the host program. It is also possible to enqueue multiple kernels. There are a few things that one need to setup in the host program before one can execute a OpenCL kernel, if the reader wants to learn more about OpenCL I suggest looking at the example and tutorial pages at the Khronos Group website [14].

Pros:

- Good API for General Purpose Processing on heterogeneous devices.
- supported by many manufacturers

- open specification
- supports multiple operating systems
- possibility to support many different devices.

Cons:

- Mostly only support CPU and GPU devices today.
- New technology meaning some implementations are still a bit shaky.

2.3.4 GPU Pros and Cons

Let's sum up some pros and cons of the GPU.

Pros:

- Cheap performance
- Power efficient
- Open specification for programming GPUs (OpenCL)

Cons:

- Overhead slow bus speeds (PCI express, AGP and so on)
- Harder to program than a GPP.
- Floating point accuracy most cards only have single precision.
- Not all GPUs support OpenCL

2.4 Conclusion

General purpose processing on special purpose devices is becoming more and more popular in the industry. And for GPUs general purpose processing has already come a long way, this is not really the case for DSPs yet. That is one of the main reasons why the GPU seems to be the best suited device for a test implementation but also due to the fact of the availability of technologies that are supported by different vendors. GPUs are also easy to get their hands on and are quite cheap together with high computational performance. When it comes to choosing a way to do general purpose processing on a GPU, OpenCL seems to be the obvious way to go, due to the fact that it is made with general purpose processing in mind. It is an open standard, it is supported by several vendors and can be run on other devices as well, like a CPU or in the future maybe even a DSP. By using OpenCL it is also very likely that the test implementation can be run on a wide range of different computers with different hardware and software setup. It also make it more likely to be able to handle more devices in the future without the need of doing any large modifications to the implementation.

3 Erlang

3.1 Introduction

In the previous section we have looked at the architectures of special purpose compute units in order to see if we can use them to improve the performance of Erlang. The next step we need to take is to learn more about Erlang to see how we can improve it. This section will introduce the reader to Erlang and briefly describe the Erlang compiler and virtual machine in order to give the reader some basic understanding of Erlang.

3.2 What is Erlang

Erlang [8] is a garbage collected, concurrent , general-purpose programming language and runtime system developed by Ericsson in the 1980's in Ericsson's computer science lab. Erlang is a functional language with built in support for error recovery and concurrency and is well suited for soft real-time systems and non-stop applications. It also supports hot swapping meaning that the code can be change while the system is still running with out the need to restart the system. Erlang is a naturally concurrent language and while threads are considered difficult in many other languages, it is not an issue in Erlang since there exists features for creating and managing processes on the language level with the goal of simplifying concurrent programming.

3.2.1 The History of Erlang

It all started back in the early eighty's in Ericsson computer science lab [9], where experiments were conducted with programming telecom-systems in a vast amount of different languages. The conclusion from these experiments was that the language used must be of a very high level symbolic nature in order to achieve productivity gains. So out of all the tested languages they were left with languages like Lisp Prolog and Parlog. The testing of these languages continued and new conclusion were made that the language also must contain primitives for error recovery, concurrency and also that the execution model may not contain back tracking, and the language must also have granularity of concurrency such that one asynchronous telephony process can and must be represented as one process in the language. The conclusion was made that a new language needed to be created. A language with the desirable features from Prolog, Lisp and Parlog but with things like concurrency and error recovery built in the language. Erlang was then first presented in 1990 at ISS'90. When the amount of users of Erlang increased Ericsson decided to create a new section within Ericsson which would maintain the development of Erlang and Erlang tools. In 1998 Ericsson released the open source version of Erlang and the amount of users has increased every year since then.

3.3 The Erlang Compiler in short

The first ever Erlang compiler was written in Prolog by `Joe Armstrong` [3] but the current Erlang compiler is written in Erlang itself. Today Erlang is compiled into two intermediate languages, first the Erlang source code is translated into something called Core Erlang and then later it is translated into BEAM assembler and then written to a binary file, a so called BEAM file. Between these different stages some code optimisations are done as well.

3.3.1 Core Erlang

Core Erlang is intermediate representation of Erlang and lies as layer between the source code and the intermediate code. One of the reasons this intermediate language came to be, was that the syntax of Erlang had grown and become quite complex and because of this complexity it made it hard to develop programs that operated directly on Erlang source code, e.g, to do code optimisations. The Core Erlang language is designed to make such code optimisations easier by doing them on the Core Erlang language instead of original Erlang source code. The specification of Core Erlang was produced as a cooperation between the Computer Science Department at Uppsala University and Ericsson OTP Team. How the Core Erlang language works is described in the Core Erlang specification [6].
According to `Carlsson` [5] the design goals of Core Erlang were:

- It should be a strict high-order functional language, with a clear and simple semantics.
- It should be as regular as possible, to facilitate the development of code-walking tools.
- Translation from Erlang to equivalent Core Erlang should be as straight forward as possible. And it should be as straight forward to translate Core Erlang in to the Intermediate code used in Erlang Implementations.
- There should be a well-defined textual representation of Core Erlang, with a simple, preferably unambiguous grammar making it easy to construct tools for printing and reading programs. This representation should be possible to use for communication between tools using different internal representations of Core Erlang.
- The textual representation should be easy for humans to read and edit, in order to simplify debugging and testing
- The language should be easy to work with supporting most kinds of familiar optimisations at the functional programming level.

3.3.2 The BEAM File Format

The BEAM file format is based on the `EA IFF 85 Standard for Interchange Format File`. One of the main differences from the standard is the four byte alignment of chunks used in the BEAM file format therefore the ”FOR1” is used in the file header instead of the EA IFF 85 standards ”FORM”. The BEAM file format is described online by `Gustavsson` [13]. But I will now briefly describe it here

Chunks

The BEAM files are divided in chunks these chunks contain different types of information. I will briefly describe 9 different chunks that are also described by `Gustavsson` [13].

Form Header

The header.

`4 bytes 'FOR1'`	This is indicating an IFF form. But FOR1 is used instead of FORM indicating that the BEAM file format is using 4byte alignment of chunks.
`4 bytes n`	Form length (file length - 8)
`4 bytes 'BEAM'`	Form type in our case BEAM
`n-8 bytes ...`	All the chunks, concatenated.

Atom Table Chunk

Note this chunk is mandatory. The first Atom in the table must be the module name.

`4 bytes 'Atom'`	The ID of the chunk (i.e. Atom chunk)
`4 bytes size`	the total chunk length
`4 bytes n`	The number of atoms in the atom table
`xx bytes ...`	And all the atoms, the atoms are represented as strings preceded by the length in a byte.

Export Table Chunk

The chunk is mandatory. In this chunk the list of exported functions is stored, in order for the BEAM loader to know which functions that are exported.

`4 bytes 'ExpT'`	The id of the chunk (i.e. ExpT for Export Table chunk)
`4 bytes size`	The total chunk length
`4 bytes n`	The number of exported functions
`xx bytes ...`	All the function entries, each function using 3 * 4 bytes (Function, Arity, Label)

Import Table Chunk

This chunk is mandatory. This chunk has a list of all imported functions.

`4 bytes 'ImpT'`	The id of the chunk
`4 bytes size`	The total chunk length
`4 bytes n`	The number of imported functions
`xx bytes ...`	And a list of imported functions each function using 3 * 4 bytes (Module, Function, Arity).

Code Chunk

This chunk is mandatory for obvious reasons, since this chunk contains the actual code.

`4 bytes 'Code'`	The id of the chunk
`4 bytes size`	The total chunk length
`4 bytes sub-size`	The length of the information fields before the actual code. This will allows for the header to be extended and the file would still be loadable by older versions of the loader because it will still know where the code starts.
`4 bytes set`	Instruction set identifier. Will be 0 for OTP R5. Should be increased if the instructions are changed in incompatible ways (i.e. if the semantics or argument types for an instruction are changed, or if the instructions are renumbered or deleted).
`4 bytes opcode_max`	The number of the highest opcode used in the code section. This makes it easier to add new opcodes. The BEAM loader should refuse to load a file if opcode_max is greater than the maximum opcode it knows about.
`4 bytes labels`	The total number of labels (to make it easier for the loader allocate the label table).
`4 bytes functions`	The total number of functions.
`xx bytes ...`	And the actual code.

String Table Chunk

This chunk is mandatory and it keeps all the literal strings of a module, If there aren't any literal strings, the chunk size should be 0.

`4 bytes 'StrT'`	The id of the chunk (i.e. StrT for String table chunk)
`4 bytes sz`	The total length of the chunk
`sz bytes ...`	And all the strings.

Attributes Chunk

This chunk is optional, but it can still cause trouble if this chunk is missing due to the fact that the release handler in OTP might require the -vsn attribute.

`4 bytes 'Attr'`	The id of the chunk
`4 bytes size`	The size of the data in the chunk
`size bytes ...`	A list of all attributes (returned by Mod:module_info (attributes)) in Erlang external term format.

Compilation Information Chunk

This chunk is optional, but might be needed by some OTP tools
They might not work correctly if this chunk is missing.

`4 bytes 'CInf'`	The id of the chunk (i.e. CInf the Compilation Information chunk)
`4 bytes size`	Total size of the data in the chunk
`size bytes ...`	A list of all compilation information (returned by Mod:module_info(compile)) in Erlang external term format.

Local Function Table Chunk

This chunk is optional and ignored by the loader. It can be useful for cross reference tools. It can safely be removed if you don't plan to run any cross-reference tools that operate directly on BEAM files.

`4 bytes 'LocT'`	The ID of the chunk
`4 bytes size`	The total chunk length
`4 bytes n`	The total number of entries
`xx bytes ...`	All the function entries each function using 3 * 4 bytes (Function, Arity, Label). (The Label not really useful, but it is kept to give this table the same format as the export table.)

3.4 The Erlang Virtual Machine

The First ever Erlang virtual machine was a stack based machine written by Joe Armstrong and was called JAM (`Joe's Abstract Machine`) [7]. However nowadays BEAM (`Bogdan/Björns Erlang Abstract Machine`) is used. This virtual machine is a register based machine and it has up to 1024 virtual registers. The Erlang Virtual machine is written in C and consist of several thousand lines of code.

3.4.1 The BEAM Loader

The BEAM loader is a part of the virtual machine and is responsible of loading the code in the BEAM files. The instruction set in the BEAM files differ from the ones actually interpreted by the virtual machine. The BEAM files use a so called limited instruction set. This instruction set will be expanded in the BEAM loader into an extended instructions set. The limited instruction set consist of 152 different instruction as of `Erlang R14B01`. And the extended instruction set consist of 435 different instructions. The reason for two instruction set is that you can keep the limited set smaller hence taking less space and making the BEAM files smaller. The limited set contains generalisations which on runtime will be transformed to a specialised instruction in the extended set. To clear things up, you can for example have several move instructions in the extended set but to save space you use a single move instruction in the limited set. These limited instructions are then transformed using load scripts (transforms) in the file `beam_opcodes.c` by the emulator and the work is done in the file `beam_load.c`. The file `beam_opcodes.c` is actually generated by a Perl script (`beam_makeops`) using a file called `ops.tab` as input containing the specialised instructions. The limited set of instructions can be found in the file `genop.tab`. In `Erlang R14AB01` you will find these files in the following hierarchy.

File	Path
`beam_makeops`	`erts/emulator/utils/`
`ops.tab`	`erts/emulator/beam/`
`beam_opcodes.c`	`erts/emulator/<machine>/opt/smp/`
`beam_load.c`	`erts/emulator/beam/`
`genop.tab`	`lib/compiler/src/`

The file `beam_opcodes.c` is generated when you build Erlang. All this is necessary to know in order to be able to add new instructions to the Erlang VM.

3.4.2 Processes

An Erlang process [20] is something different from an operating system process. The Erlang VM handles all the scheduling and execution of Erlang processes. Erlang processes communicate using message passing and do not share any data.

3.4.3 Scheduling

Erlang uses a simple round-robin scheduling with some minor tweaks for maximum and background priority. The default number of schedulers used, is one scheduler per processor core. That is if you have a processor with two cores in your system, Erlang will as default use two schedulers. The number of schedulers can also be configured so that you can use more or less schedulers but there are rarely anything to gain from having more schedulers than physical cores [20].

3.5 Parts of Erlang that can be parallelised

What parts of Erlang could be a candidate to run on a special purpose compute unit. We already know from the previous chapter that in order to benefit from running anything on a GPU or DSP the problem

needs to be parallelisable. You could start digging in the Erlang VM and see if you could found something in there that could benefit from being parallelisable. But instead one can look at the language and see if we find something there that could benefit from being parallelised. And in Erlang there exist such a thing that in theory could benefit from being run in parallel. In Erlang you have something in the language called bitstring comprehensions and list comprehensions. They look something like this
List comprehension

```
[(X+1) || X <- [1,2,3,4,5,6]].
```

Bitstring comprehension

```
<< <<(X+1)>> || <<X>> <= <<1,2,3,4,5,6>> >>.
```

The difference is that list comprehensions work on Erlang lists while bitstring comprehension work on binary data (i.e. bitstrings). What the two different lines above do is that they add 1 to every element in the list or bitstring (depending on if it is a bitstring comprehension or list comprehension). You can of course do other things to the elements as well but in order to keep the example simple I have chosen to just add 1 to every element. From this we can see that in theory it should be faster to add 1 to several elements simultaneously instead of doing it one by one. This makes bitstring and list comprehensions interesting candidates for a test implementation on a GPU for example. To make the implementation easier one should choose one of these to do the first test implementation with. It seems that bitstring comprehensions could be easier to handle due to the fact that bitstring comprehensions operate on binary data and thus it should be easier to pass the data to OpenCL.

3.6 Conclusion

In this chapter Erlang was briefly explained, some parts of the Erlang virtual machine and compiler were presented and a problem suitable for a test implementation was found. The problem that seemed most suitable for a test implementation was bitstring comprehensions due to the fact that it operates on binary data and that it can be parallelised.

4 Test Implementation

4.1 Introduction

The test implementation I have done is to be considered as a proof of concept. It demonstrates that you can identify certain types of bitstring comprehensions in Erlang source code and translate them into OpenCL kernels, so that you can either run them on a CPU, GPU or any device that follows the OpenCL specification. The point of this is that it should be completely transparent for the Erlang programmer. Who should not need to learn OpenCL in order to run these bitstring comprehensions in OpenCL. The programmer should just go about writing Erlang code as usually and the translations should be done on compile time by using the bs2cl module presented in this chapter.

4.2 Overview of how it works

First of all the Erlang source file is read into a string (strings are handled as lists in Erlang), this list is then tokenised using the function `erl_scan:string()`. This will return the entire source file as a list of Erlang tokens. The list of tokens are then scanned and bitstring comprehensions are identified and returned as a list of bitstring comprehensions. Then each individual bitstring comprehensions is looked at to see if it can be translated or not. If it can't be translated we just move on to the next one. After all bitstring comprehension that can be translated have been translated, a new source file is created in a temporary directory called bs2cl, which is located in the same directory as the original source file. All the newly translated bitsring comprehensions and the remaining code form the original file is now written to the newly created source file. The translation is done by the module bs2cl which is written in Erlang. The translated bitstring comprehensions have now been replaced by a call to the runcl module. The runcl module is then responsible for running the translated bitstring comprehensions in OpenCL. The runcl module is written in Erlang and is using the OpenCL binding for Erlang [26] and is more or less just an OpenCL host program.

Here is an example of a simple file translated using the test implementation.

The original file:

```
%% The Original Module
-module(test).
-export([test/0]).

test()-> << <<(X+5)>> >> || <<X>> <= <<1,2,3,4,5>> >>.
```

After translation:

```
-module(test).
-export([test/0]).
test()-> runcl:run(arr, \"__kernel void bs(__global char *Xin,__global char *Xout){int i=get_global_id(0);
Xout[i]=Xin[i]+5;}\",[<<1,2,3,4,5>>]).
```

4.3 The translation module bs2cl

4.3.1 Erlang Tokens

To extract all the Erlang tokens from a source file I use the `erl_scan:string()` function this function will take a list (string) of Erlang source code and tokenise it, the result will be a list with only valid Erlang tokens in it (meaing that comments are stripped out).

So For example if you run `erl_scan:string()` on the following Erlang module:

```
%% The Original Module
-module(test).
-export([test/0]).

test()->
      << <<(X+5)>> || <<X>> <= <<1,2,3,4,5>> >>.
```

You would get the following result:

```
[{'-',2},{atom,2,module},{'(',2},{atom,2,test},{')',2},{dot,2},
{'-',3},{atom,3,export},{'(',3},{'[',3},{atom,3,test},{'/',3},{integer,3,0},{']',3},{')',3},{dot,3},
{atom,5,test},{'(',5},{')',5},{'->',5},
{'<<',6},{'<<',6},{'(',6},{var,6,'X'},{'+',6},{integer,6,5},{')',6},{'>>',6},{'||',6},{'<<',6},{var,6,'X'},
{'>>',6},{'<=',6},{'<<',6},{integer,6,1},{',',6},{integer,6,2},{',',6},{integer,6,3},{',',6},{integer,6,4},
{',',6},{integer,6,5},{'>>',6},{'>>',6},{dot,6}]
```

So what you see above is the module test tokenised. I have for clarity reasons divided the list of tokens such that each row above is equal to the rows in the source code for the module test. The first row of the module test is a comment so that row will be skipped entirely. So to try to make everything clear here, I will explain how the second row gets tokenised. First we find the token `'-'` in the module test and that is represented in the result from `erl_scan:string()` as a tuple `{'-',2}` where `'-'` is the token represented as an atom, and the `2` indicating that it was found on line 2 in the source file of the test module. The next token found is the atom `'module'` which is represented as `{atom, 2, module}` in the result from `erl_scan:string()` where `atom` indicates that it is an atom, the `2` indicating the line, `module` is the actual atom `module`. The next token found is `'('` which is represented as `{'(',2}` just like with `'-'` above. The next token is the atom `test` which is represented in the same way as the atom module above that is `{atom, 2, test}` then we find the ending parenthesis `')'` which is represented in almost the same way as the other parenthesis above that is `{')',2}`. Finally we find `'.'` which is the last token on line 2 and is represented as `{dot,2}` where dot is an atom representing `'.'` in the source file of the test module and the `2` is just like above the line number. I hope that the reader now at least have a basic understanding of how the `erl_scan:string()` is working.

4.3.2 Finding bitstrings comprehensions in a source file

Now that we have covered how an Erlang source file gets tokenised it is time to move on to identifying bitstring comprehensions. We start off from the tokenised list from the above example. The list will now be parsed in order to identify all bitstring comprehensions. So to start off how do one identify a bitstring comprehension? What is unique about a bitstring comprehension? Well let's take a look at the one in the test module above.

```
<< <<(X+5)>> || <<X>> <= <<1,2,3,4,5>> >>.
```

What we can see from this is that a bitstring comprehension starts with the token `'<<'` and ends with the token `'>>'`, but that is not all we need to look at, since a normal bitstring or binary also have that syntax. The thing that makes a bitstring comprehension unique is that it starts with `'<<'` and ends with `'>>'` and that it also have the token `'||'` in the middle. With that in mind it should now be possible to identify bitstring comprehension from the list of tokens in the following way. We need two counters one

for counting the occurrences of `'<<'` and one for counting the occurrences of `'||'`. I will call them `Bcount` for counting `'<<'` and `Pcount` for counting `'||'`. The `Bcount` counter will be increased when a `'<<'` is found and decreased when a `'>>'` is found, so we can keep track of open and closing `'<<'` `'>>'` tokens. The `Pcount` counter will only be increased when a `'||'` is found. So a bitstring comprehension is found if we find the token `'>>'` and the `Bcount` and `Pcount` counters are both 1. I also use a counter to count how many tokens that have been processed (i.e. a counter that will be increased for every token). I will now provide an example on the above bitstring comprehension. first we find `'<<'` so we record the position (i.e. the number of processed tokens so far in the source file, this is only done if `Bcount=0`) and increase the `Bcount` counter by 1, we then find another `'<<'` and we yet again increase the `Bcount` counter by one so we now have `Bcount=2`. then we find `'('` which we don't need to do anything with so we just move on. We then find `'X'`, `'+'`, `'5'` and `')'` which we don't need to do anything with so we move on, then we find `'>>'` and on that token we decrease `Bcount` so we now have `Bcount=1`, `'||'` is found so `Pcount` is increased. Then `'<<'` is found and `Bcount` is increased, `'X'` is found , a `'>>'` token is found and `Bcount` is decreased (`Bcoount=1`). Then the token `'<='` is found which we don't do anything with. We now find `'<<'` and `Bcount` is increased (`Bcount=2`), we then find the tokens `'1'`, `','`, `'2'`, `','`, `'3'`, `','`, `'4'`,`','` and `'5'` which we don't need to do anything with. We then find `'>>'` which decreases `Bcount` (`Bcount=1`) and then the token `'>>'` is found and we now have `Bcount=1, Pcount=1` which means that we have found a bitstring comprehension. So we now record the position of the last token of the bitstring comprehension and create the following tuple {`bscomp, Start, Stop, Line`} where `bscomp` is an atom indicating that it was a bitstring comprehension that was found. `Start` is a variable containing the start position of this bitstring comprehension, `Stop` is containing the end position of this bitstring comprehension and `Line` is a variable containing the line number that this bitstring comprehension was found on.

4.3.3 The Actual Translation

Lets look into how the actual translation is done. I will now focus on the translation of one bitstring comprehension. First of all we split the bitstring comprehension into two parts. We split on the Erlang token "`||`" and thus we get a left and a right part. We start by translating the right part first. The right part translation is not that advanced yet but will probably be more advanced in the future.

4.3.4 The Right Part

The right part is basically to determine what the input will be to the OpenCL kernel. There are two cases either we have a list/binary as argument or we can identify a generative function (at present only `lists:seq` is supported). If it is just a list or binary we extract that part and make into to a bitstring, that will then later be passed as an argument to the OpenCL kernel. If we have a generative function (that is if we can identify a generative function like the `lists:seq`) we will translate that into a list of inputs for example [X, Y] where X is the start of the interval and Y is the end of the interval. The X will be the argument to the kernel while Y will be used to determine the number of times to run the OpenCL kernel.

4.3.5 The Left Part

The Left part of the translation will be given the output from the right part as input (i.e. the argument list). It will also be given what kind of translation to do (that is if we have a supported generative function on the right hand side or not). At present the generative function case could be considered as a special case and the non generative case as a general one. In the future the generative translation needs to be more generalised.

The process of translating the left part starts by parsing the left part and see that we can support it. It

looks at tokens to see if it can handle them or not, if a token that was not expected or can not be handled is found this entire bitstring comprehension will be skipped. After the token scan is done one more scan is done to see if there are any external variables (i.e.. variables that was bound outside of the bitstring comprehension) if one is found then the current bitstring comprehension is skipped. This is due to that there is currently no support to deal with external variables. This is also something that will need to be supported in the future.

4.3.6 Creating the Kernel Head

The next step now is to create the kernel head this is something that will depend on the encoding of the resulting bitstring but at present 8bit encoding is assumed since only integers are supported at the moment and the standard encoding of an integer in Erlang is 8bit. However there is support for adding an encoding checker into the current implementation to be able to support different encodings. The kernel head will be created based on the result of the right part translation. When the kernel head has been created it will look something like this:

```
"__kernel void bs(__global char *Xin,  __global char *Xout){int i = get_global_id(0);"
```

`char` is used as the type because of the assumed 8bit encoding. A `char` in OpenCL is 1byte (8bits) and since chars are essentially 8bit integers I choose to use char to represent the Erlang type int in the OpenCL kernels when 8bit encoding is used.

4.3.7 Creating the Kernel Body

The next step now is to create the kernel body. At present it will only allow one variable in the left part that is the one that is bound in the bitstring comprehension for example X in this bitstring comprehension

```
<< <<(X+1)>> || X <= <<1,2,3,4,5>> >>.
```

So when it come across a variable it assume it is X and would there for translate X to Xin[i] for the kernel head shown above. When we come across a normal token we just convert it to a list (i.e. a string). If we come across an integer we do the same thing we convert it to a list. When it is done the kernel body would look like this:

```
"Xout[i] = Xin[i]+1"
```

we then add the following tokens ";" and "}" so the end result is:

```
"Xout[i] = Xin[i]+1;}"
```

The Kernel head and Kernel body is now put together and the result of translating the left part to OpenCL is:

```
"__kernel void bs(__global char *Xin,  __global char *Xout)
{ int i = get_global_id(0); Xout[i] = Xin[i]+1;}".
```

4.3.8 Putting it All Together

We now have the right and left translations done and what we have left to do is to generate the call to the runcl module

Creating the Call to the runcl Module

We pass the kernel source which is the result of the left part translation and the kernel arguments which is the result of the right part translation and we also pass which type of translation we have done. The call is generated in three stages first stage we construct the function call and concatenate it with the type (either arr or gen) resulting in something like the following string "`runcl:run(arr`". The next stage is to concatenate the first stage with the kernel source resulting in something like:

```
"runcl:run(arr, \" __kernel void bs(__global char *Xin,  __global char *Xout)
{ int i = get_global_id(0); Xout[i] = Xin[i]+1;}\""
```

and in the final stage we extract the actual arguments from the argument tuples and concatenate it with the result from the second stage resulting in a complete call to the runcl module looking something like this:

```
"runcl:run(arr, \" __kernel void bs(__global char *Xin,  __global char *Xout){ int i = get_global_id(0);
Xout[i] = Xin[i]+1;}\", [<<1,2,3,4,5>>])."
```

4.3.9 The runcl module

The runcl module is using the OpenCL binding for Erlang. The module is quite simple as of now. It works like this, a call is made to the runcl module instead of the bitrstring comprehension. The call contains an OpenCL kernel, the arguments for that kernel and what type of problem it is (i.e. if it is a generative function or not). And what the runcl module then does is basically to enqueue the OpenCL kernel and apply the arguments and then execute the kernel and return the results. The setup is slightly different between a generative problem and a regular problem and that is why the type is included in the call to the runcl module. There is currently no advanced code in the runcl module, it is more or less a basic OpenCL host program and I will therefore not go into any further details about the code. I will instead focus on explaining the call and arguments to the runcl module below.

Calling the runcl module

A Call to the runcl module looks like this,what the supplied arguments are will be explained below

```
"runcl:run(Type, KernelSource, ListOfArguments)."
```

Type

The run function have different cases depending on the `Type`. The Type is an atom specifying what kind of translation that has been done. The type can be any of the following:

```
Type = arr | gen
```

KernelSource

The KernelSource variable contains the OpenCL kernel source code. For example:

```
KernelSource = "__kernel void bs(char *Xin, char* Xout){int i=get_global_id(0); Xin[i] = Xout[i]+1;}"
```

TheListOfArguments

This is simply a normal list containing arguments to the OpenCL kernel. For example:

```
[<<1,2,3,4,5>>] or [1,1000].
```

4.4 Summary

I hope the reader now has basic understanding of how the presented test implementation works. The goal of the test implementation was to show that it is possible to identify certain types of bitstring comprehensions in Erlang source code and translate them to OpenCL. Thus enabling those type of bitstring comprehensions to run on special purpose compute units like a GPU. This chapter has also briefly explained how this implementation was done.

5 Benchmarking

5.1 Approach

To see if there is anything to gain from running bitstring comprehensions on the GPU, some benchmarking was needed to be done. Since the implementation is done in OpenCL I will compare the performance of a normal Erlang bitstring comprehension to an equivalent OpenCL implementation on both CPU and GPU. The two problems I choose for benchmarking is a simple substitution cipher which will read a text file and add 1 to each char to shift the character 1 steps to right. The second problem is generating data with the function lists:seq to see if we gain anything from generating the data on the device itself. Since there is some overhead in sending data to and from the GPU, the GPU could benefit from generating the data itself for large datasets. The test problems will be described in the section test cases.

5.2 How the Test are Performed

The tests are divided into two problems both these problems are run for 6 different data sizes. The test is run 10 times for each data size and the average running time of all these 10 runs are used as the result. This is done to get consistent results.

5.3 The Test Cases

For doing benchmarking two test problems were used and they are presented below.

5.3.1 Substitution Cipher

The substitution cipher is quite simple it shifts an entire text 1 character code to the right by adding 1 to the character code. The Erlang implementation of the problem looks like this:

```
<< <<(X+1)>> || <<X>> <= Data >>.
```

Where `Data` is a bitstring containing the content of a text file.
The equivalent OpenCL kernel implementation looks like this:

```
__kernel void cipher(__global char* input,
                     __global char* output)
{
   int i = get_global_id(0);
   output[i] = input[i]+1;


}
```

5.3.2 Data Generation on Device (lists:seq)

The second test is to generate the data on the device it self. Using the `lists:seq` function. The Erlang code looks like this.

```
<< <<X>> || X <- lists:seq(1,To) >>
```

Where `lists:seq` is our generative function and `To` is the end of the interval so if `To` is equal to 10 we will generate a list starting with 1 and ending with 10 (i.e. [1,2,3,4,5,6,7,8,9,10]). The equivalent OpenCL kernel for the Erlang code above would look like this:

```
__kernel void  gen(__global char start,
                   __global char *output)
{
   int i = get_global_id(0);
   output[i] = (start+i);


}
```

5.4 The Test Setup

5.4.1 Data sets

The input data for the tests consists of six different data sizes for each problem.

Substitution Cipher Input

The substitution cipher test is run with six different text files of different sizes as input. See the table below.

Input	size (in bytes)	size (in MB)
small.txt	316282	0.3
Large.txt	546304	0.52
XL.txt	1092608	1.04
XL2.txt	2185728	2.08
XL3.txt	21860352	20.85
XL4.txt	43999744	41.96

Data Generation on the Device (lists:seq)

The input for the data generation tests are intervals, that is we have a start value and a stop value, for example 1, 10 which would generate 10 elements (i.e. [1,2,3,4,5,6,7,8,9,10]). The inputs used in the test are found in the table below.

From	To	Number of generated elements
1	1000	1000
1	10000	10000
1	100000	100000
1	1000000	1000000
1	10000000	10000000
1	110000000	110000000

5.4.2 The Hardware

The tests have been conducted on a Apple Mac Mini (model number A1283) with the following specifications.

CPU: `Intel Core 2 Duo (two cores at 2Ghz each)`
RAM: `4GB (2x2GB 1067Mhz DDR3)`
GPU: `Nvidia Geforce 9400M (VRAM: 256Mb)`

5.4.3 Tools Used

The tests are run using `Erlang R14B01` with the NIF version of the `OpenCL bindings for Erlang` [26]. And using the test module described below. Also `Apples` implementation of OpenCL is used. The resulting graphs are produced using `Matlab`.

5.4.4 The Test Module

In order to perform these benchmarks a test module was written. The module contains functions for running tests and meassuring execution times for the two test problems described above.

Measuring running time

In order to measure the running time of each test a couple of functions was written to do this. First function is the `get_time` function that retrieve a timestamp (i.e. current time in milliseconds). It looks like this.

```
get_time() ->
      {Mega,Sec,Micro} = now(),
      ((Mega*1000000+Sec)*1000000+Micro) div 1000.
```

What this function does is to retrieve a timestamp using the Erlang BIF (Built In Function) `now()` and convert the result to milliseconds. Then a function for calculating the difference between timestamp was needed `calc_time()` and that function looks like this.

```
calc_time(Start,Stop)->
      Stop - Start.
```

This is a very simple function it takes two timestamps as input and calculates the difference between them. `get_time` and `calc_time` are the two functions needed to calculate the running time of a test.

How a Test is Executed

We have two test problems hence we have two test functions, the test function for the substitution cipher looks like this

```
compare_result(File)->
   Data = data(File),
   io:format("Byte size: ~p\n",[byte_size(Data)]),
   Res1 = test_bc(Data),
   Res2 = test(gpu, Data),
   Res3 = test(cpu, Data),
   Res4 = test(all, Data),
   Res1 = Res2 = Res3 = Res4.
```

It starts by calling the `data()` function which is just simple function that reads in a file specified by the variable `File`. Then on the next line the size of the file is printed out. Then `test_bc` is called which will run the substitution cipher on the data as an Erlang native bitstring comprehension (see test cases). The result will be stored in `Res1`. On the next line we do the same but in OpenCL for the GPU and store that result in `Res2` and we then run the same test again in OpenCL but this time for the CPU and the result is stored in `Res3` and then yet again for the case where the CPU and the GPU are working together, these results are stored in `Res4`. On the final row we match all the results to see that they are equal, so we can make sure that all devices and implementations produced the same result.

The `test_bc()` function looks like this

```
test_bc(Data)->
  Start = get_time(),
  RData = << <<X>> || <<X>> <= Data >>,
  Stop  = get_time(),
  Result = calc_time(Start, Stop),
  io:format("Exec Time Erlang: ~p ", [Result]),
  RData.
```

The function start by retrieving a timestamp and store it in `Start`, then it performs the actual computation that is the substitution cipher and save the result in `RData`. Then it retrieve yet another timestamp and store it in `Stop`. Then the difference between the two timestamps are calculated and stored in result and then the execution time is printed out and then `RData` is returned.

The `test()` function looks like this

```
test(DevType, Data)->
  Start = get_time(),
  RData = run(Data, DevType),
  Stop  = get_time(),
  Result = calc_time(Start, Stop),
  io:format("Exec Time ~p: ~p ", [DevType, Result]),
  RData.
```

This function does the same as the `test_bc` with one difference that it calls the function `run()` instead. The function `run` is responsible for running in OpenCL instead it uses the kernel presented in test cases for the substitution cipher. I will not go into how the run function works since it basically do the same thing as the runcl module does, that is a simple OpenCL host program.

The test function for generating data looks like this:

```
compare_resultGen(To)->
   Res1 = test_bcGen(To),
   Res2 = testGen(gpu, To),
   Res3 = testGen(cpu, To),
   Res4 = testGen(all, To),
   Res1 = Res2 = Res3 = Res4.
```

This function dose basically the same as the one for the substitution cipher the difference is the input is the end of the interval for the generative function. And it also calls test functions for generating data on device, instead of the substitution cipher test files. The functions `test_bcGen` and `testGen` does pretty much the same thing as `test_bc` and `test()` presented above, so I will not explain them further.

5.5 The Results

The tests have been run for the two test cases, they have been run for six different inputs each. Each test has been run 10 times for each input and the results presented here are the average running time over these 10 runs.

5.5.1 The Average Results of 10 Runs

The Average Results for 10 Runs for the Substitution Cipher

Erlang Bitstring comprehension

Input	Input size	Average running time
small.txt	316282 bytes	20.7ms
Large.txt	546304 bytes	34.5ms
XL.txt	1092608 bytes	68.5ms
XL2.txt	2185728 bytes	135.5ms
XL3.txt	21860352 bytes	1362.2ms
XL4.txt	43999744 bytes	2742.3ms

OpenCL GPU

Input	Input size	Average running time
small.txt	316282 bytes	60.5ms
Large.txt	546304 bytes	39.9ms
XL.txt	1092608 bytes	41.0ms
XL2.txt	2185728 bytes	48.8ms
XL3.txt	21860352 bytes	243.5ms
XL4.txt	43999744 bytes	471.5ms

OpenCL CPU

Input	Input size	Average running time
small.txt	316282 bytes	3.5ms
Large.txt	546304 bytes	5.4ms
XL.txt	1092608 bytes	10.6ms
XL2.txt	2185728 bytes	17.8ms
XL3.txt	21860352 bytes	197.1ms
XL4.txt	43999744 bytes	371.8ms

OpenCL GPU+CPU

Input	Input size	Average running time
small.txt	316282 bytes	55.8ms
Large.txt	546304 bytes	33.3ms
XL.txt	1092608 bytes	37.1ms
XL2.txt	2185728 bytes	47.9ms
XL3.txt	21860352 bytes	215.2ms
XL4.txt	43999744 bytes	412.0ms

The Average Results for 10 Runs for Data Generation on Device

Erlang Bitstring Comprehension

Number of generated elements	Average running time
1000	0.3ms
10000	1.1ms
100000	25.2ms
1000000	138.1ms
10000000	2600.2ms
110000000	27437.2ms

OpenCL GPU

Number of generated elements	Average running time
1000	26.4ms
10000	34.1ms
100000	31.5ms
1000000	32.1ms
10000000	84.8ms
110000000	665.9ms

OpenCL CPU

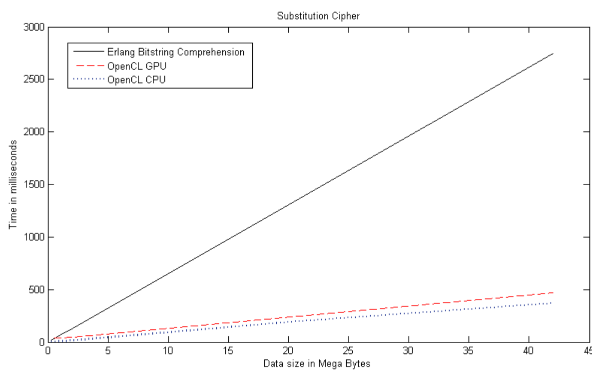
Number of generated elements	Average running time
1000	5.7ms
10000	1.2ms
100000	1.8ms
1000000	8.1ms
10000000	67.0ms
110000000	710.5ms

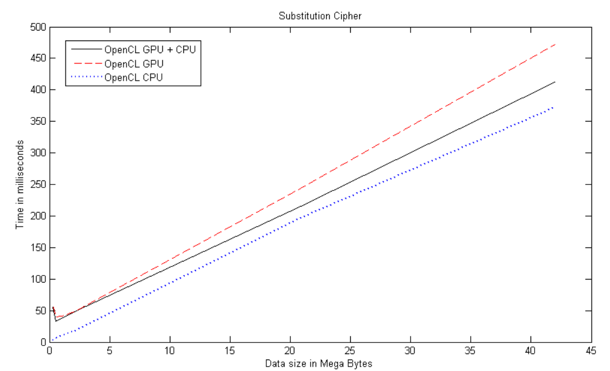
OpenCL GPU+CPU

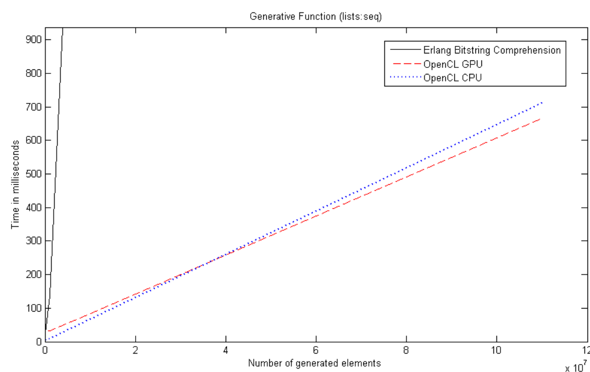
Number of generated elements	Average running time
1000	29.9ms
10000	32.7ms
100000	28.3ms
1000000	53.1ms
10000000	82.4ms
110000000	613.5ms

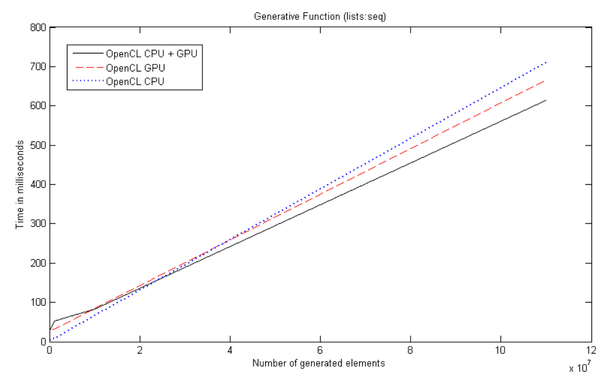
5.5.2 Figures

And here you can see the data above plotted. The first two graphs show the substitution cipher test case. In the first graph OpenCL CPU and GPU is compared to a normal Erlang bitstring comprehension and in the second graph OpenCL CPU and GPU is compared to OpenCL with CPU and GPU working together. The last two graphs shows the generative function test case (`lists:seq`). The third graph shows the generative function test case with OpenCL CPU and OpenCL GPU compared to Erlang bitstring comprehension the fourth graphs shows the same problem but OpenCL CPU and OpenCL GPU is now compared to OpenCL with CPU and GPU working together.









5.6 Conclusion

To conclude this chapter I thought I should discuss the results. From the result we can see the following. Erlang bitstring comprehension is faster then any of the OpenCL devices for small data sets. This is most likely due to the overhead introduced of running OpenCL (for example compiling the OpenCL kernels since this is done at runtime). But we also see that as soon as the data size increases the normal Erlang bitstring comprehension doesn't scale as well as OpenCL does. Whether it is OpenCL on the CPU, OpenCL on the GPU or OpenCL where CPU and GPU work together, they all outperforms Erlang bitstring comprehensions for sufficiently large data sets. It is also interesting to see that in the substitution cipher test case OpenCL on CPU scales better than OpenCL on the GPU, but in the generative function test case OpenCL on the GPU scales better than the OpenCL on the CPU. The reason for the CPU being faster than the GPU for the substitution cipher is most likely due to memory latency. And this is why the GPU outperforms the CPU in the generative function test because then the data is generated on the device and we only need to send the data one way. The reader should know however that the GPU used in these tests is far from the most powerful GPU on the market and that the results could be different with a more powerful GPU. However the overhead compared to the CPU will exist either way and this is why the CPU will outperform the GPU for small data sets. Unless we will do really high intensity calculation on each element then perhaps the GPU could out perform the CPU on quite small datasets but on small enough data sets the CPU would probably always be faster.

It is also interesting to notice that one does not gain as much as one would think from having the CPU and GPU work together. The reason for this is probably that the GPU scales better for larger datasets and since the CPU and GPU is sharing the problem the GPU will work on a smaller amount of data hence not doing optimal work. It will however catch up compared to GPU only because of that the CPU will work on the other part of the problem which will be smaller as well. And we know that the CPU scales better for smaller data sets. That is the reason for that the CPU+GPU scales better then GPU only but worse than CPU for small datasets. But if you look at the generative function test case you can see that for a big number of elements to generate, OpenCL CPU+GPU scales best of all followed by GPU only and then CPU. This is because the data set is large enough so that the GPU has plenty to do thus being effective and the parts the GPU can't work on is handle by the CPU at the same time thus outperforming GPU only.

6 Proposal: Extending the Erlang VM

6.1 introduction

In the previous sections the following conclusions have been made. It has been pointed out that a GPU could be a good candidate for test implementation due to the high availability and due to that there exists an open standard to program GPUs (thanks to OpenCL). We also know that OpenCL can be used on other devices as well like the CPU and could in theory be used to program a DSP as well (if the manufacturers would choose to support it). We also know that in order to take advantage of the GPU we need a parallelisable problem. In the Erlang chapter it was pointed out that bitstring comprehensions could be a suitable problem for test implementation. In this section I will present a simplified proposal to extend the Erlang compiler and virtual machine to run certain types of bitstring comprehensions in OpenCL on both the GPU and CPU. In chapter 4 I provided a proof of concept test implementation that shows that Erlang could benefit from running certain bitstring comprehensions in OpenCL both on a GPU and on a Multi-Core processor. I also provided some benchmarking in chapter 5, comparing regular bitstring comprehensions to OpenCL implementations on both the CPU and GPU to back up my assumptions.

6.2 Proposal Overview

I propose a way of running bitstring comprehensions in OpenCL by extending the Erlang virtual machine and compiler. The goal is that the compiler should identify certain types of bitstring comprehensions and translate these to OpenCL. Much like in the same way as I have done in the test implementation, but with the exception that the original bitstring comprehension should also be kept. So that we can determine where the bitstring comprehension should be run (i.e. if it is going to be run as Erlang native bitstring comprehension, OpenCL CPU or OpenCL GPU). Where the bitstring comprehension should be run should be decided on runtime in the Erlang virtual machine, where we can know what size the input is and what the size the resulting output data will be. Another reason for keeping the original bitstring comprehension as well as the OpenCL kernel is that OpenCL is not supported on all systems and in those cases the normal Erlang bitstring comprehension should be used instead.

6.3 Extending the Compiler

The first step is to extend the compiler to recognise certain types of bitstring comprehensions so that it can translate them into OpenCL kernels. The question here is in which level it would be most suitable to do this. I have in the test implementation done translations directly on Erlang source code. But it might be better to do it on the Core Erlang level, since Core Erlang was designed to be easier to do code optimisations on. Either way one must (as mentioned earlier) keep both the original bitstring comprehension as well as the OpenCL kernel. That is because of the fact that we don't know on this level if data size is small or not (i.e. if we can gain from running it in OpenCL). We also don't know if the system support

OpenCL or not. These things will be known at runtime and therefore we will leave it up to the virtual machine to decide. When the bitring comprehensions have been translated to OpenCL kernels we also need to introduce new beam instructions that can handle them. Exactly how these instructions would look like is something that will have to go under future work, when the whole idea has been evaluated in more depth. One might also want to introduce a new chunk in the BEAM file where the translated bitstring comprehensions could be kept, in this chunk the kernels should be ordered so that virtual machine can know which OpenCL kernel belongs to which bitstring comprehension, so when the virtual machine comes across a bitstring comprehension it can first check if there exist an OpenCL kernel for this bitstring comprehension and then decide if there is anything to gain from running the OpenCL kernel instead of the Erlang native bitstring comprehension.

6.4 Extending the Virtual Machine

The final step is to extend the virtual machine we need for the virtual machine to understand and handle the new instructions. In the virtual machine it must be known if it is possible to run OpenCL, also the virtual machine must determine if there is anything to gain from running the bitstring comprehensions in OpenCL or not (i.e. if the data size is large enough). In the virtual machine it is already possible to know what size the resulting bitstring comprehension will be, so that part could possibly be quite straight forward. The code for running the OpenCL kernels like the runcl module presented in chapter 4 needs to be modified and reimplemented in C, since the Erlang virtual machine is implemented in C. How the virtual machine should know if there is OpenCL support on the current system or not, I think should be decided when building Erlang for the system in mind. I think this is the easiest way to determine if OpenCL support exists or not, this could be checked by the build script. The drawback is that if OpenCL is then later supported on the same system then Erlang would have to be rebuilt for that system.

How the Virtual Machine Would Handle Bitstring Comprehensions

Whenever the Erlang virtual machine comes across a bitstring comprehension it should first check to see if there exists a OpenCL kernel associated with this bitstring comprehension. If there is, it should determine if the input is large enough to benefit from running in OpenCL instead of Erlang native bitstring comprehension. It also needs to determine if it should be run on the CPU or the GPU. Note that this is very hard to determine since it depends on what kind of GPU and CPU that is available in the system. I don’t really know how to solve this at present time. This could possibly be solved by providing some benchmarking module that can run once and evaluate the system and build a record over certain problems. And from that it could be determined if it the bitstring comprehension is going to be run on the GPU or CPU. So it could be better to focus on only one device for example the GPU to begin with. Then one only need to compare it to Erlang native bitstring comprehension. This was a bit of a side note so let’s move on. When the virtual machine has determined where the bitstring should be executed it will either run the bitstring as an Erlang bitstring comprehension, that is in the way it does in the current virtual machine or it will set everything up to run the OpenCL kernel instead on the device best suited.

6.5 To Sum Up

What I am actually proposing here is to take the concept of the test implementation provided in chapter 4 and move it into the Erlang runtime system. So that all these things can be handled by the Erlang compiler and virtual machine and make it completely transparent for the Erlang programmer. But in order to do this I need to learn even more about the Erlang runtime system, I need to learn Core Erlang in order to evaluate it to see if it is better to identify bitstring comprehensions on that level.

7 Conclusion and Discussion

7.1 Conclusion

We have shown that it is possible to extend the Erlang runtime system to use special purpose processing units. However it might not be straight forward, in this thesis a GPU was used as a proof of concept and a test implementation was produced and benchmarks were made. However these results may differ depending on the hardware, that is if a new special purpose device is supported it does not necessarily mean that it will be good on solving the same problems or produce similar results. By adding support for another special purpose compute unit you might be able to find other parts of Erlang that fit this new device.
The main problem to take this to the next step is how to determine which problem should run on which device. We have seen that both the GPU and CPU scales better than Erlang bitstring comprehensions for the two test problems and probably do for most bitstring comprehensions for large enough data sets. We have also seen that for one problem the CPU scales better and for another the GPU scales better and also that a CPU and GPU working together scales better than the CPU for one problem and not for another. We have also seen that even though the GPU scales better than the CPU, it is still not until the data size is large enough that you benefit from running on the GPU over the CPU this was true for a dual core processor but maybe it would be a different result with a single core processor. This means there are still things left to investigate further. So is it useful to utilise the GPU. It all depends on how you look at it, if you just want to use the GPU to offload some problems to gain performance over native Erlang than this thesis has showed that it is possible and you will gain performance for large enough data sets, but bare in mind it might not always be the most effective solution, but still better than the way Erlang do it today (for large enough data sets).

7.2 What I Have Learned

Doing this thesis has led me to a greater understanding of special purpose compute units. It has also given me a greater insight behind the scenes in Erlang that is the compiler and virtual machine. I have also learned that Erlang can benefit from utilising special purpose compute units but maybe for more specialised tasks than I initially hoped for. I also learned that it takes considerably more time to complete a full extension of the Erlang VM than I initially expected, but it is something that has given me a great experience and I hope to be able to continue pursuing this in the future.

7.3 What I Have Accomplished

I have not revolutionised the world nor have I made an actual extension of the Erlang virtual machine. But I have come a long way down the road. I have identified characteristics of common special purpose compute units. I have looked into how Erlang can benefit from this. I have identified a part of Erlang (i.e. bitstring comprehensions) that can benefit from running on a special purpose device compared to how they are being evaluated in Erlang today. I have also showed that it can be done by doing a test implementation. I have also shown how much one can gain from doing that by providing benchmarks. I have also proposed on an abstract level how one could take this to the next level by integrating into the Erlang compiler and virtual machine.

7.4 Problems Along The Way

One of the main problems has been the lack of documentation regarding the Erlang virtual machine, there is no documentation available at all, basically the source code is the documentation. I have now a basic but still limited understanding of how the Erlang virtual machine works but it is something I can build on from here on. I got some pointers in the right direction from `Tony Rogvall` and `Björn Gustavsson` and for that I am thankful.

8 Future Work

8.1 Introduction

This thesis has only scratch the surface by proving it is possible to use special purpose compute units to enhance parts of Erlang. There is still a lot to be done. In this chapter I will go through some points regarding future work.

8.2 Extending the Test Implementation

The test implementation provided in chapter 4 is so far quite basic. It is limited to a subset of bitstring comprehensions. The support for more bitstring comprehensions is needed, functionality to detect the encoding of bitstrings is needed. Support for external variables (that is variables that are bound outside the bitstring comprehension) is needed and increased support for different types like float is needed. These are the ones closest to the future but there is more things to be done.

8.3 Looking at More Things to Improve

Bitstring comprehensions was a good first step to look at. But there might be more things that can be improved using special purpose compute units. For example one might be able to the same thing with list comprehension as what has been done with bitstring comprehensions. Further more one might look at sorting, maybe sorting can be accelerated on special purpose compute units like the GPU and DSP. Cryptography could be yet another area to look at.

8.4 Extending the Virtual Machine

When the test implementation has been fully evaluated and improved, it is time to move on to actually extend the virtual machine, in order to do this in a good way I need to learn even more about the compiler and virtual machine I also need to learn Core Erlang in order to evaluate if it is better to do the translations on this level or not.

8.5 Supporting More Special Purpose Compute Units

Another next step is to look at how to utilise other special purpose compute units, e.g, the DSP. The test implementation should be extended to support a DSP maybe with the help with OpenCL or maybe not as of today I have not seen any DSPs that claim to support OpenCL but as mentioned earlier it is possible that they will support OpenCL in the future and this is something I will have to look into in the future. Also other special purpose compute units should be investigated to see how they could be utilised, a next step could be to look at crypto processors.

9 Related Work

9.1 Introduction

The are not to my knowledge anyone who has tried to extend the Erlang runtime system to support special purpose compute units. There are however some related projects that have relevance to this project. I will in this chapter mention a few of them.

9.2 OpenCL Binding for Erlang

`Tony Rogvall` has made an OpenCL binding for Erlang [26]. This allows you to write your host program in Erlang, it allows you to run OpenCL kernels directly from Erlang without the need to write a C/C++ host program. The OpenCL binding for Erlang is available for Mac OS, Linux and Windows. The OpenCL binding is currently maintained by `Tony Rogvall` and `Dan Gudmundsson`. I had great use of the OpenCL binding for my test implementation.

9.3 Beam Emulator in Erlang

`Tony Rogvall` is working on a BEAM emulator [25] built in Erlang. The purpose of this is project is to document how the BEAM emulator works and to get a better understanding of the semantics.

9.4 Erjang

`Kresten Krab Thorup` has made a virtual machine for Erlang which runs on top of Java [28]. It loads Erlang's binary file format (`.BEAM`), converts it into Java's `.class` file format, and loads it into the JVM (Java Virtual Machine). The purpose of this project was that `Kresten` wanted to learn Erlang and he thought this seemed like a good way to do it.

9.5 Erlang-Embedded

Erlang-embedded [10] is a platform for developing embedded Erlang applications. The project started as a collection of thesis projects at Uppsala University, under the supervision of Erlang Solutions. This thesis is a part of this project.

References

[1] AMD. ATI Radeon HD4850. Description and specification available at: *http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-4000/hd-4850/Pages/ati-radeon-hd-4850-specifications.aspx*.

[2] AMD. Programming guide AMD Accelerated Parallel Processing, OpenCL, 2011.

[3] Joe Armstrong et al. Use of Prolog for developing a new programming language. The Practical Application of Prolog, 1992.

[4] BeagleBoard.org. The BeagleBoard, 2010. More information about the beagleboard can be found at *http://www.beagleboard.org*.

[5] Richard Carlsson. An introduction to Core Erlang. Erlang Workshop, 2001.

[6] Richard Carlsson et al. Core Erlang 1.0.3 language specification, 2004.

[7] Ericsson. Erlang.org implementations FAQ. Visit: *http://www.erlang.org/faq/implementations.html*.

[8] Ericsson. Erlang/OTP an Overview.
Information available at: *http://www.erlang.se/productinfo/erlotp_overview.shtml*.

[9] Ericsson. History of Erlang. Available at: *http://www.erlang.org/course/history.html*.

[10] Erlang-Solutions. Erlang-Embedded. Erlang Factory, 2010. More information about the Erlang-Embedded project can be found at *http://www.embedded-erlang.com*.

[11] Kayvon Fatahalian. From shader code to a Teraflop: How Shader Cores Work, 2009. The slides are available at: *http://www.khronos.org/registry/cl/*.

[12] William Fornaciari. DSP Architecture. Lecture Slides, 2003. The slides are available for download at: *http://home.dei.polimi.it/fornacia/didattica/ES0405/L15/04ESdsp.pdf*.

[13] Björn Gustavsson. File Format for BEAM R5 and Later. Website, 2000. Available online at: *http://www.erlang.se/~bjorn/beam_file_format.html*.

[14] Khronos-Group. OpenCL Tutorials and examples. Visit: *http://www.khronos.org/developers/resources/opencl/#ttutorials*.

[15] Khronos-Group. OpenCL Specification, 2009. The OpenCL specification is available at *http://www.khronos.org/registry/cl/*.

[16] Khronos-Group. OpenCL Introduction and Overview, 2010. Download available: *http://www.khronos.org/developers/library/overview/opencl_overview.pdf*.

[17] Khronos-Group. The Industry's Foundation for High Performance Graphics. Website, 2011. Find more information about OpenGL at: *http://www.opengl.org*.

[18] Kevin Kuang. DSP vs ASP. Slides, 2004. Download available at: https://ccrma.stanford.edu/∼kuangzn/uc/oral/DSP%20vs%20ASP.ppt.

[19] Victor W. Lee et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. ISCA 10, 2010.

[20] Kenneth Lundin. Inside the Erlang VM. Erlang User Conference, 2008. Download available at: *http://ftp.sunet.se/pub/lang/erlang/euc/08/euc_smp.pdf*.

[21] Machresearch.org. OpenCL Tutorials, 2009. Available at: *http://www.macresearch.org/opencl*.

[22] Microsoft. DirectX Developer Center, 2011. Find more information about Microsofts DirectX technology at: *http://msdn.microsoft.com/en-us/directx/*.

[23] Nvidia. Nvidia GeForce GTX285. Description and specification available at: *http://www.nvidia.com/object/product_geforce_gtx_285_us.html*.

[24] Nvidia. Nvidia CUDA Programming guide version 2.1, 2008.

[25] Tony Rogvall. A BEAM Emulator written in Erlang. GitHub, 2009. The BEAM Emulator written in Erlang is available at *http://github.com/tonyrog/beam*.

[26] Tony Rogvall. OpenCL Binding for Erlang, 2009. The OpenCL Binding for Erlang is available at *http://github.com/tonyrog/cl*.

[27] Muhammad Shaaban. DSP vs ASP. EECC 722, 2003. Download available at: http://meseec.ce.rit.edu/eecc722-fall2003/722-10-8-2003.pdf.

[28] Kresten Krab Thorup. Erjang, 2010. More information and download available at: *http://github.com/trifork/erjang/wiki/*.