MASTER'S THESIS | LUND UNIVERSITY 2015

Implementing a streaming application on a processor array

Jerry Lindström, Stefan Nannesson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-13

Implementing a streaming application on a processor array

A case study on the Epiphany architecture

Jerry Lindström
jerrylindstrom91@gmail.com

Stefan Nannesson
stefan.nanneson@gmail.com

June 7, 2015
Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jörn W. Janneck, jorn.janneck@cs.lth.se

Examiner: Krzysztof Kuchcinski, Krzysztof.Kuchcinski@cs.lth.se

Abstract

Today, stream processing and parallel computing are common ways of processing information. In short, stream processing is a programming paradigm, related to single instruction, multiple data (SIMD), that makes it easier for some applications to exploit a limited form of parallel processing. Parallel computing on the other hand is about computing things in parallel, which probably could be guessed from the name.

The goal of this project is to implement a streaming application on a parallel computing machine. This involves porting a real-world application, written in a stream processing language and compiled by tools, developed by the Embedded System Design (ESD) research group at Lund University, onto a platform including an embedded processor array (the Adapteva's Epiphany), an ARM processor and programmable logic. The motive of the project is to be a starting point for benchmarking and improving the tools that the ESD research group in Lund University is developing.

The ESD research group is trying to support a stream-programming model based on the CAL language [21], which has been adopted by MPEG to specify the reference code for their video standards [17]. This is why a video decoder has been chosen as the driver application of this project.

The target platform for this project is a Parallella board by Adapteva, which features a Xilinx Zynq chip as host and an Epiphany 16-core processor array on the side. The target platform could also be anything else, like a supercomputer with many parallel cores. The Parallella board was chosen because processor arrays are what the ESD group thinks might be the future architecture of highly parallel machines. Implementing a parallel programming model on a processor array architecture is a test if the parallel programming model might be useful in building applications for processor arrays.

A key feature of the stream-programming language is that it can be synthesized to hardware [20] and software [18], both for single-core processors, multi-core machines, and embedded processor arrays like the Epiphany [19]. This project covers the creation of some library elements to support complex applications on that platform, such as FIFOs between Epiphany cores and the ARM host on the Zynq, some components that handle access to external RAM, a component that draws pixels onto a screen.

Keywords: Stream processing, Parallel computing, Adapteva, Parallella, Epiphany, Moving Picture Experts Group (MPEG), First In First Out (FIFO), Embedded System Design (ESD), ARM, Multi-core

Acknowledgements

This project was conducted at the **Department of Computer Science, Lund University**.

We thank **Jorn W. Janneck** for being our supervisor for this project and for providing guidance and expertise that greatly assisted the research. We also thank him for his valuable insights, which greatly improve the research, as well as the project report.

We thank **Krzysztof Kuchcinski** for being our examiner for this project. We would also like to thank him for helping us to find this project.

We thank **Flavius Gruian** for his guidance on how to register the project and how to fill out the administrative forms.

We thank **Gustav Cedersjö** for his Cal Actor Language (CAL) to C converter, which made it possible for us to convert a Reconfigurable Video Coding (RVC) decoder written in CAL to C. We would also like to thank him for supplying the bit streams that we used for testing the decoder.

We would also like to show our gratitude to **Lars Nilsson** and **Anders Bruce** for setting up our project computers, supplying us with the necessary hardware and providing us with a Apache Subversion (SVN) account.

Contents

Chapter 1
Introduction

1.1 Research question

The main research question of this project is how to implement streaming applications on a processor array.

Other matters related to the problem of implementing a streaming application on a processor array are also investigated, such as:

- How to communicate between multiple cores in a modular way.
- How to make modularity easy for a stream processing application to run in a parallel environment.
- How to communicate between two different processor types (in this case an ARM and an Epiphany processor).

1.2 Related work

This section covers work that is related to our project. The topics that are extra noteworthy are the following:

- What dataflow, CAL and actors are.
- Compiling dataflow programs to software.
- The Epiphany architecture

1.2.1 Software Code Generation for Dynamic Dataflow Programs

This subsection covers some of the work done by Janneck and Cedersjö [18]. Their research is relevant to this project, because the decoder that we are using is converted with their software code generation program.

The main topic of their research paper is how they address problems of generating efficient software implementations for a large class of dataflow programs. This topic is however not of primary interest to us, as we are just using their end product.

This research paper mentions the term *dataflow program*. The term dataflow program is used for a network of computational kernels called *actors*. Each actor has a distinct functionality and input and output ports, which are used to send packets of data, called *tokens*. The actors are connected to each-other, using *channels*. Channels are buffered, lossless and order preserving connections between the ports of two actors.

In our case, the channels are FIFO structures. The generated code for the driver application use macros to read, write, peek, pop and consume tokens to the channels. The default channels were replaced by changing the macros to use our FIFO structures. An application has to be *modular* in order to work with our framework. Our definition of a modular application, is an application that consists of actors that do not share states with each other.

1.2.2 Realizing Efficient Execution of Dataflow Actors on Manycores

The paper [19] covers research that is similar to that in our project. The Epiphany architecture is mentioned here, and the description is similar to that in 2.5.

Communication through buffered FIFOs on the Epiphany architecture and Epiphany specific features to be considered, when implementing FIFOs on this architecture is also mentioned. The two specific features that were mentioned, are the difference in speed between read and write transactions and the potential use of Direct Memory Access (DMA) to speed up memory transfers, which allows the processor to do processing in parallel with the memory transfer. This is in many ways similar to our own FIFO implementation.

1.2.3 MPEG Reconfigurable Video Coding Framework

This subsection contains a brief overview of the MPEG Reconfigurable Video Coding (RVC) Framework, which is an ISO standard that aims to provide a better way of defining the specification of video coding technologies. All information in this subsection is taken from the following research paper [17].

The Motion Picture Expert Group (MPEG) develops video coding standards. They initially used textual descriptions as specification. However, as complexity increased, programming languages such as C or C++ were used instead, starting with the MPEG-4 standards. A problem with a C implementation in this context is that knowledge, which may be needed for an efficient implementation, is lost. Furthermore, as standards continuously grow in complexity, designing solutions are becoming increasingly difficult.

The RVC framework intents to provide means to define specifications at a higher level, where new decoder configurations can be specified by selecting components from a library of standard algorithms. The framework uses a subset of the dataflow programming language CAL, to specify the standard library and to instantiate the RVC decoder model. By using CAL code, algorithms can be described in a manner that enables solutions, where for example parallel execution can be better exploited.

Using various tools, C-code for software and VHDL-code for hardware synthesis can be generated from a RVC decoder specification. It was shown that hardware code generated from a CAL-model outperformed handwritten VHDL code in terms of performance, resource usage and design efforts for a MPEG Simple Profile decoder. Tools also exist for analyzing and simulating a RVC decoder.

1.2.4 MPEG-4

MPEG-4 is one of many methods for compressing audio and video in order to efficiently transport the data across various data channels.

Relevant parts of MPEG-4 that are worth mentioning in this report are how the *encoder* and the *decoder* are implemented.

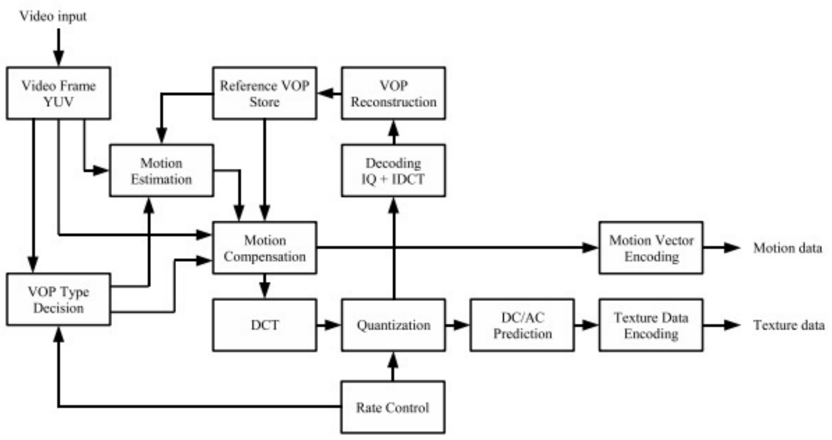


Figure 1.1: A simplified diagram of MPEG-4 simple profile encoding [26]

Figure 1.1 illustrates a simplified diagram of MPEG-4 simple profile encoding. The MPEG-4 encoding process goes through the following steps [26]:

- Determine if the frames are *I-frames* or *P-frames*. I-frames are independently decoded frames without any reference to other images. P-Frames are usually more bit-efficient compared to I-frames, but they are sensitive to transmission errors due to the complex dependency on earlier frames [27].
- The motion compensator calculates the difference between the current frame and the reference frame.
- The discrete cosine transform (DCT) step converts the resulting values to the frequency domain. After this is done, a quantization step reduces the storage precision by cutting the not coefficients.
- The quantized values are scanned and then coded with Huffman encoding and variable length encoding.

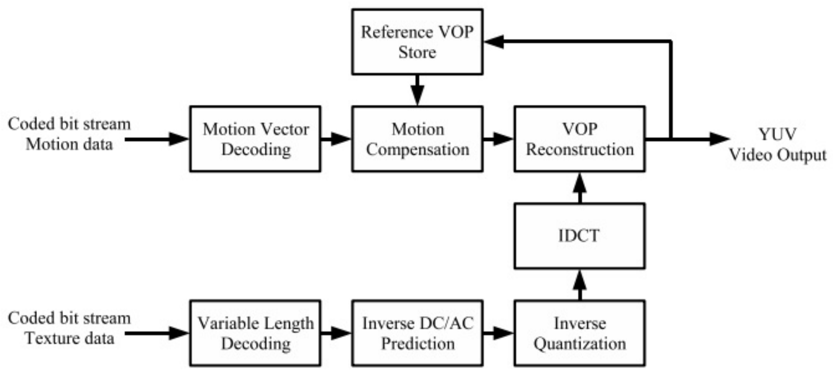


Figure 1.2: An MPEG-4 simple profile decoder [26]

Figure 1.2 illustrates an MPEG-4 simple profile decoder. The decoding process has the following steps [26]:

- Parse the video stream into motion and texture sub-streams for *macroblocks*. A macroblock is a processing unit based on linear block transforms [28].
- The movement vector of the macroblock is represented by the motion sub-stream. The decoder uses the movement vector to copy the corresponding portion of the reference frame to the actual frame.
- The texture sub-streams holds either data, that completely describes the macroblock (if the frames are I-frames) or data that compensates the pixel differences of the moved macroblock (in cases of P and B-frames).

A more general note is that many MPEG-4 decoders have been implemented over the years. The RVC decoder used in our project is one such implementation. Other implementations have been made for different platforms and applications. Good performance is often an important feature for these. For example [25] summarizes techniques for optimizing a MPEG-4 SP decoder for real-time performance on ARM7TDMI processors. These are not relevant for our work, as it is the modular properties of the RVC decoder that is important.

1.3 Division of work

In this project, Jerry is responsible for the communication between cores using FIFO structures, the moving of functionality from the host-processor to the co-processor and optimizations to the generated decoder code in order to fit on the Epiphany cores. Jerry is also responsible for the larger portion of the report. Stefan is responsible for making modularity easy, adapting the video decoder to work on multiple cores and conducting performance experiments which include optimizations.

Jerry's most noteworthy achievements in this project are implementing the FIFO structures, figuring out how to shorten the actor code down by a significant amount and laying the grounds for moving actors out from the host-processor to the co-processor. Stefan's most noteworthy achievements are implementing the core synchronization, figuring out how to draw pixels on the screen and laying the grounds for easily shuffling actors around with a simple make file.

Chapter 2
Background

2.1 Applications

Our framework can probably be used to parallelize other stream processing applications, if the application is modular. The framework that we have developed is currently only working on the selected platform used in this project, but it could be adapted to work with other highly parallel machines by rewriting the architecture specific parts.

Our First In First Out (FIFO) structures for communications between actors are lossless, order preserving and buffered. These FIFO structures are designed with easy modularity in mind.

2.2 Cal Actor Language (CAL)

CAL is a high-level programming language for writing dataflow actors. CAL has been compiled to many different target platforms, such as single core processors, multi-core processors and programmable hardware. Cryptography, compression and video processing are common application areas of CAL [16]. In this project, the driver application (which is the RVC decoder) was initially written in CAL, then translated into C.

The compiler provided by Cedersjö is the compiler that is responsible for the generated CAL to C conversion in this project. The compiler identifies input patterns that are related to the characteristics of the CAL programming language, such as: how many tokens the actor will consume when the action is executed and what the firing rule of a certain actor is.

The code of an actor can be generalized by a controller, followed by a finite number of states. An example of a generated actor that has been translated by the CAL to C compiler is as follows:

CAL code before C translation

```
actor Algo_Byte2bit ()
        uint(size=8) BYTE
        ==>
                bool BITS :

  int(size=4) count := 0;
  int(size=9) buf;

  // Read in an new byte when needed
  reload: action BYTE:[i] ==>
  guard count = 0
  do
    buf := i;
    count := 8;
  end

  // Write out bits in msb first order
  shift: action ==> BITS:[ bit ]
  guard
    count != 0
  var
    bool bit = if bitand( buf, 128) != 0 then true else false end
  do
    count := count - 1;
    buf := lshift( buf, 1 );
  end

  //priority
    //reload > shift;
  //end
end
```

The corresponding C code after translation from CAL

```
// ACTOR MACHINE
// decoder_serialize
static int8_t count_n0v0;
static int16_t buf_n0v0;
static uint8_t i_n0v1;
static _Bool bit_n0v2;
static void init_n0s0(void) {
        count_n0v0 = 0;
```

```
}
static void init_n0s1(void) {
        i_n0v1 = PEEK(_b142, 0);
}
static void init_n0s2(void) {
        bit_n0v2 = ((buf_n0v0 & 128) != 0) ? true : false;
}
static void transition_n0t0(void) {
        buf_n0v0 = i_n0v1;
        count_n0v0 = 8;
        CONSUME(_b142, 1);
}
static void transition_n0t1(void) {
        count_n0v0 = count_n0v0 - 1;
        buf_n0v0 = buf_n0v0 << 1;
        WRITE(_b24, bit_n0v2);
}
static _Bool actor_n0(void) {
        _Bool progress = false;
        static int state = -1;
        switch (state) {
        case -1: break;
        case 0: goto S0;
        case 7: goto S7;
        case 10: goto S10;
        case 11: goto S11;
        case 13: goto S13;
        }
        init_n0s0();
S0:
        if (TOKENS(_b142, 1)) {
                goto S2;
        }
        else {
                goto S1;
        }
S1:
        if (count_n0v0 != 0) {
                goto S4;
        }
        ...
```

For the full code example, please refer to appendix B.

The structure of this actor is that it has a controller (represented by the switch case in the example above) with a finite amount of states. The controller will choose which one of the available states in the actor to enter. The states may contain reading from an input buffer or writing to an output buffer. We will not go into further details on how the states are built up, as there is no need to know more than this to understand our project.

2.3 Parallella

The Parallella board is a credit card sized computer, which is inspired by the popular Raspberry Pi and Arduino [1]. It has a dual core ARM A9 processor, but what makes this board special is foremost the Epiphany co-processor chip. The board and the Epiphany chip are developed by Adapteva [3] [4] [5]. Figure 2.1 shows a picture of the Parallella board.

Figure 2.1: Picture of the Parallella board [4]

The goal of the Parallella project is "to help speed up the transition from serial to parallel computing" [3], making parallel computing hardware and tools available to an affordable price and facilitating learning and development of software that takes advantage of parallel systems [1].

The technical specifications of the board are as follows:

- Zynq-Z7010 or Z7020 Dual-core ARM A9 CPU
- 16-core Epiphany Coprocessor
- 1GB RAM
- MicroSD Card slot
- USB 2.0
- Up to 48 GPIO signal

- Gigabit Ethernet
- HDMI port
- Linux Operating System
- 54mm x 87mm form factor

2.4 Xilinx Zynq

Xilinx Zynq is a programmable system on chip (SoC) [22]. The Parallella board has either a Zynq-Z7010 or Z7020 depending on the board version.

The Xilinx Zynq mainly consists of the following components:

- An ARM processor, containing two ARM A9 cores
- Processor I/O
- Memory controllers
- Programmable logic

Note that only the most noteworthy components are mentioned in the above list. The architecture of the Xilinx Zynq is illustrated in picture 2.2.

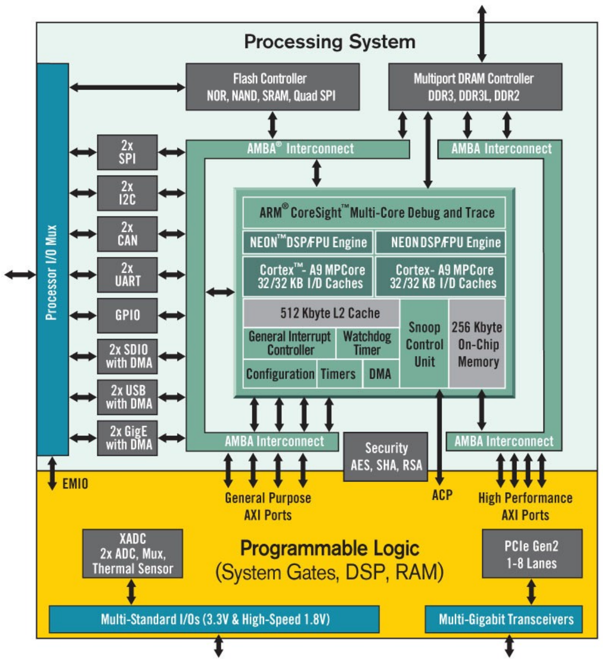

Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
Processor I/O Mux
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
AMBA® Interconnect
AMBA Interconnect
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
Snoop Control Unit
256 Kbyte On-Chip Memory
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Figure 2.2: The architecture of the Xilinx Zynq 7000 [22]

2.5 Epiphany core array

The Parallella board used in this Master thesis project has a 16-core Epiphany array, known as E16G301. This array is what we refer to as the "Epiphany core array".

Epiphany by Adapteva is an architecture for parallel computing. The architecture is scalable and can have up to 4095 cores on a single chip sharing the same 32 bit memory space, where 1MB of memory address space is dedicated for each core. The cores can be programmed using ANSI-C/C++ which make it very accessible for programmers. A simple graphical representation of how this core array looks like internally is shown in 2.3.

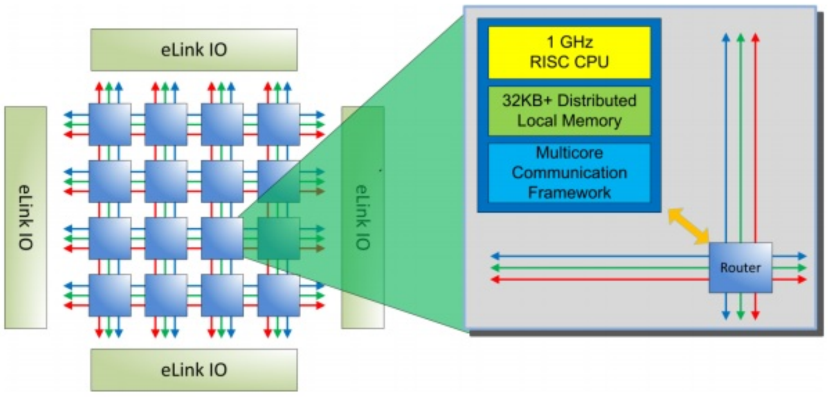

eLink IO
eLink IO
eLink IO
eLink IO
1 GHz RISC CPU
32KB+ Distributed Local Memory
Multicore Communication Framework
Router

Figure 2.3: The internal architecture of the Epiphany core array [6]

As seen in figure 2.3, the Epiphany cores are arranged as a matrix. Each Epiphany core in the matrix is formally referred to as a "mesh node"[7]. Each mesh node (Epiphany core) contains a Reduced Instruction Set Computing (RISC) CPU, Direct Memory Access (DMA) Engine, internal memory and a network interface for communication with other nodes.

There are three different meshes present in the Epiphany core array, each with a unique functionality. The first mesh (referred as the cMesh) writes transactions that are destined for an on-chip mesh node [7]. The second mesh (referred as the rMesh) is used for all read requests to the mesh node [7]. The last mesh (the xMesh) is used for write transactions destined for off-chip resources and for passing transactions destined for another chip through the mesh [7].

The E16G301 has 16 CPUs with support for 32-bit IEEE floating points[6]. Each core has 32kB local memory that can be accessed in a single clock cycle[6][7].

2.6 ARM CPU

The ARM CPU present on the Parallella board is a Zynq Dual Core ARM A9 800MHZ [8]. How the ARM processor is connected to the architecture is illustrated in 2.5.

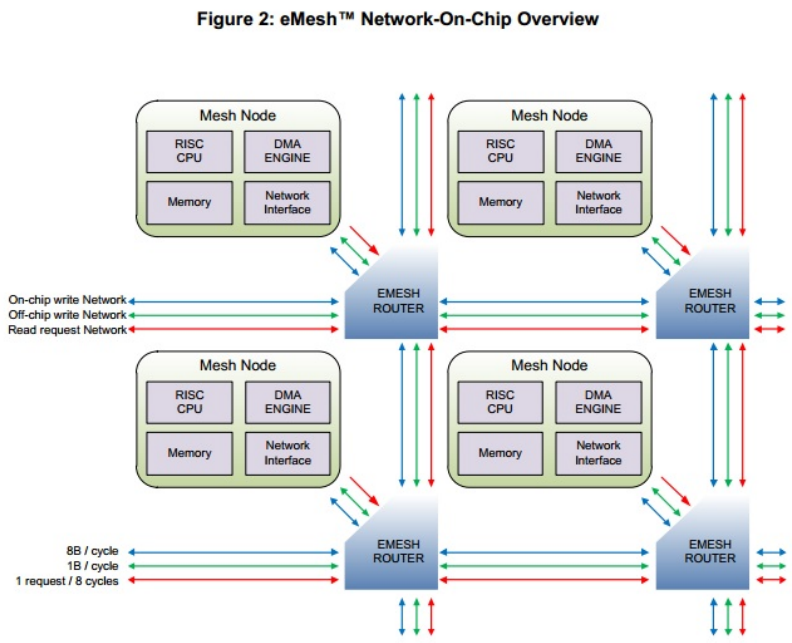


Figure 2.4: A closer look inside a mesh node [7]

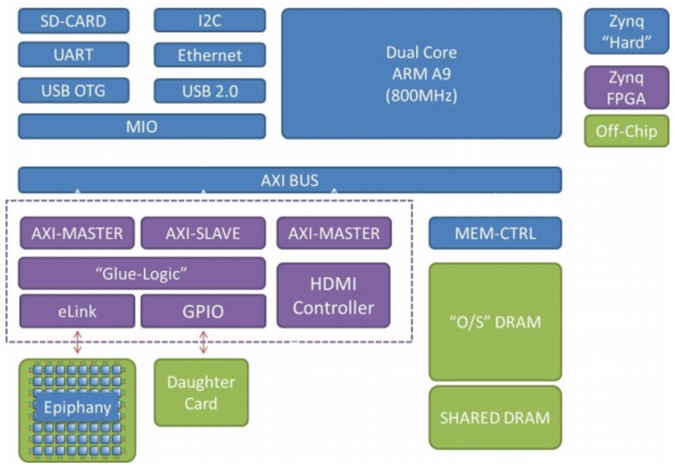


Figure 2.5: Parallella High Level Architecture [8]

A host processor is the processor that runs the operating system. In this case the host processor also controls and loads programs onto the co-processor, which is an Epiphany core array. The host processor of the Parallella board is a dual core ARM CPU. Besides its normal responsibilities as a host processor, we make use of the ARM to contain the remaining parts of the host application (in our case the video decoder), that could not be fully placed on the Epiphany core array. This means that the parts of the video decoder, that does not have room on the Epiphany cores will be placed on the ARM processor.

The host (ARM) can communicate with the Epiphany device, by accessing the Epiphany cores private memory space, or by using shared buffers in the device external memory [10].

The DRAM can be accessed by the host through the system bus and from the Epiphany via eLinks [10].

2.7 Communication between modules

This chapter covers mainly how communication propagates internally in the Epiphany core array, and what to think about when choosing core placements. As described earlier in section 2.5, there are three different internal meshes in the Epiphany core array.

2.7.1 Internal communication

As mentioned in 2.5, the internal communication propagates through the cMesh or rMesh, depending if it is a write or a read transaction [7]. Since the write and the read transactions propagate through different meshes, they should not interfere with each other. This can be seen as separate lanes on a motorway, where read transactions and write transactions have different lanes.

2.7.2 Communications for different core placements

The Epiphany core array is essentially a grid of Epiphany cores. In our case, this grid is 4x4 (16 cores), but there also exist an 8x8 (64 cores) version. Since the cores are arranged in a grid structure, cores can only communicate directly with its neighbour. Cores that are not placed next to each other have to let the communication propagate through other cores, in order for the information to reach its destination.

This means that the placement affects how long time the communication will take from one core to another. This can be illustrated in the following picture:

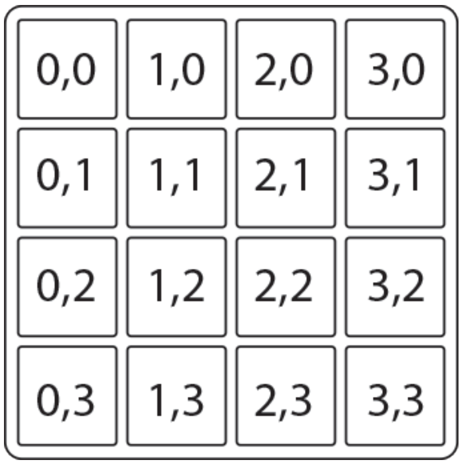


Figure 2.6: The picture illustrates a 16 core processor grid

If for example core 0,0 needs to communicate with core 3,3, the information would need to propagate through five other cores [19]. This means that the choice of grid position can greatly affect communication distance. Besides grid position, one should also consider how many neighbours each core has, in order to choose the optimum core in the grid for a certain program to run on.

For a 4x4 core grid, the amount of neighbours for each core can be illustrated in the following picture:

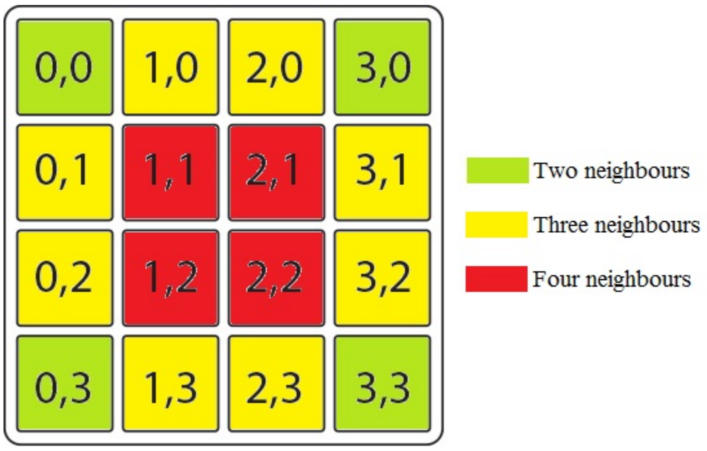


Figure 2.7: The picture illustrates how many neighbours each core has in a 4x4 grid

As seen in figure 2.7 a core either has two, three or four immediate neighbours. We have identified three good practices to minimize the communication on the grid. These are:

- Put actors that communicate frequently on the same core if possible.
- If two frequently communicating actors are on different cores, place them on cores that are located as close to each other as possible in the grid.
- For each program running on a core, determine how many other cores that program frequently communicate with and choose grid position wisely. One example is, if a program communicates only with two other programs running on different cores, it would be a waste to place it on a core with four neighbours.

2.7.3 Epiphany memory model and shared memory

The address space used by the Epiphany architecture consists of 2^{32} bytes. Each core has a local memory space ranging from 0x00000 to 0xFFFFF, where only the first 32KB is usable [7].

When a core wants to access memory from another core, a global address has to be used. The 12 most significant bits (MSB) of the global address denotes the core's coordinate, 6 bits for the row-ID and 6 bits for the column-ID. Thus, the first address of core (1,3) would be:

ROW	COL	OFFSET	
1	3	0x00000	= 0x04300000
000001	000011	00000000000000000000	

Conveniently the SDK includes a library function to perform this conversion: e_get_global_address (e-lib.h). The memory space is illustrated in figure 2.8.

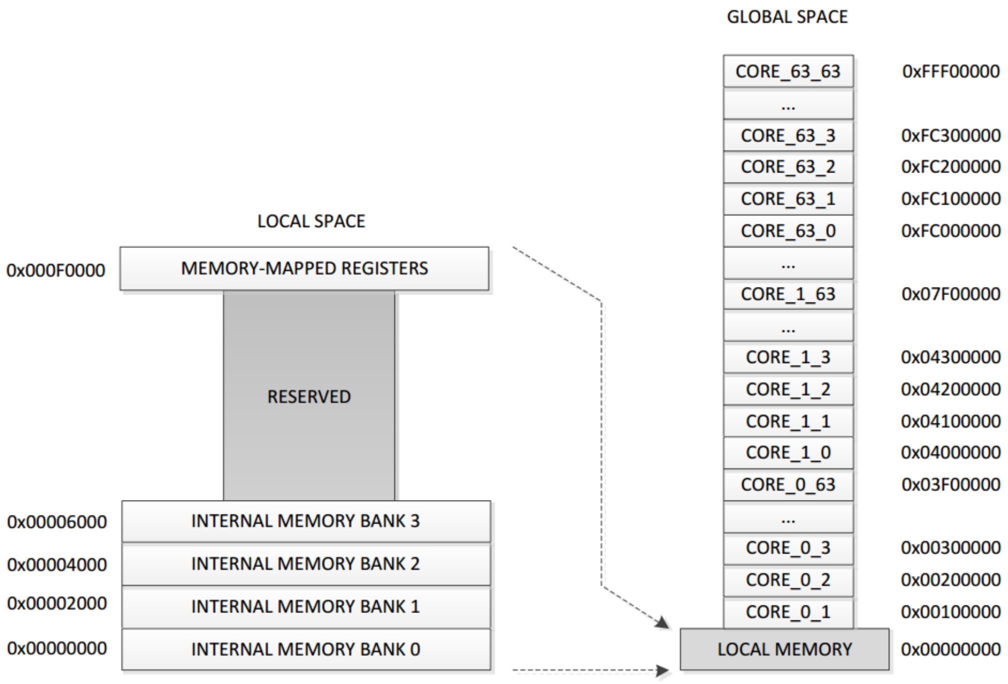


Figure 2.8: The Epiphany architecture's memory address space[7].

From a programmer's perspective, accessing another core's memory is no different from accessing the local memory, as long as global addresses are used. A simple pointer in C/C++ with value 0x043XXXXX, can be used to read from core (1,3). When a program on an Epiphany core wants to read memory located on a different core, a read-request consisting of the address to read from and a return address is sent out to the network. This is done by using the rMesh, mentioned in section 2.5. The desired data will later be sent back to the requester. This is all done automatically by the hardware.

Accessing data from other cores will however be slower as transactions must propagate through the network. Read requests are slower than write transactions; data must return and the rMesh network is also limited to 1 read-request every 8 clock cycle for each of the four directions (north, east, south and west), whereas for write transactions using the cMesh, 8 bytes can be sent every clock cycle in each direction [7].

For more details, refer to the Epiphany Architecture Reference [2].

2.7.4 Shared Memory

The Parallella board we used has 32MB shared memory that is accessible by all the Epiphany cores and the ARM cores. For the host the shared memory starts at the physical memory address 0x3f000000 [9].

Shared memory is accessible from the Epiphany cores, using the addresses 0x8E00 0000 - 0x8FFF FFFF. The linker script specifies the memory layout for the executable and is included in the Epiphany SDK folder. Some of the default linker scripts use the first 16MB of the shared memory for code and data. For example ~bsps/current/legacy.ldf in the Epiphany SDK folder [10].

The Epiphany cores do not behave any different when accessing the shared memory compared to accessing another Epiphany core's memory; read-requests are sent and data are returned as expected, due to the eLink protocol for off-chip communication being implemented, using the programmable logic in Zynq SoC [8][6][7]. Incidentally, 0x8E000000 to 0x8FF00000 would be the memory space reserved for the non-existing cores (35,32) to (35,63). As the Epiphany cores on the Parallella have column-IDs between 8 to 11 [9], messages would have to propagate east.

Virtual memory is a way to simulate that more RAM is present, by extending the address space to secondary memory, such as an SD card or a hard-disk. On the ARM host, a program must map the shared memory to the virtual memory before it can be used. The library function e_alloc (e-hal.h) can be used for this [10]:

```
e_mem_t emem;
e_alloc(&emem, offset, size);
void* addr = emem.base;
```

Alternatively, mmap (sys/mman.h) can be used directly, which might be preferable if you for example want to specify to which address the shared memory should be mapped.

Similarly, a program running on the ARM can access the Epiphany nodes' memory. After calling e_open (e-hal.h) on the host, the mapped memory addresses for the Epiphany cores can be found in:

```
e_epiphany_t edev;
e_open(&edev, 0, 0, ROWS, COLS);
void* addr = edev.core[y][x].mems.base;
```

2.7.5 Direct Memory Access (DMA)

Each Epiphany core has a DMA-engine, that enables data to be moved between nodes independently of the nodes CPUs. Once set up, each engine can transfer 8 bytes per clock cycle [2].

Using DMA instead of ordinary memory transactions ought to improve performance under certain conditions. It should especially be beneficial, when large amounts of data is accessed consecutively and the CPU can perform other tasks at the same time.

To use DMA, a couple of library functions are available. For example, e_dma_copy that probably is the easiest to use, as it behaves like the standard C-function memcpy.

2.8 RVC Decoder Overview

This section is a brief architectural overview of the RVC-decoder. It only contains the details required for this project. For more in-depth information, please refer to [17].

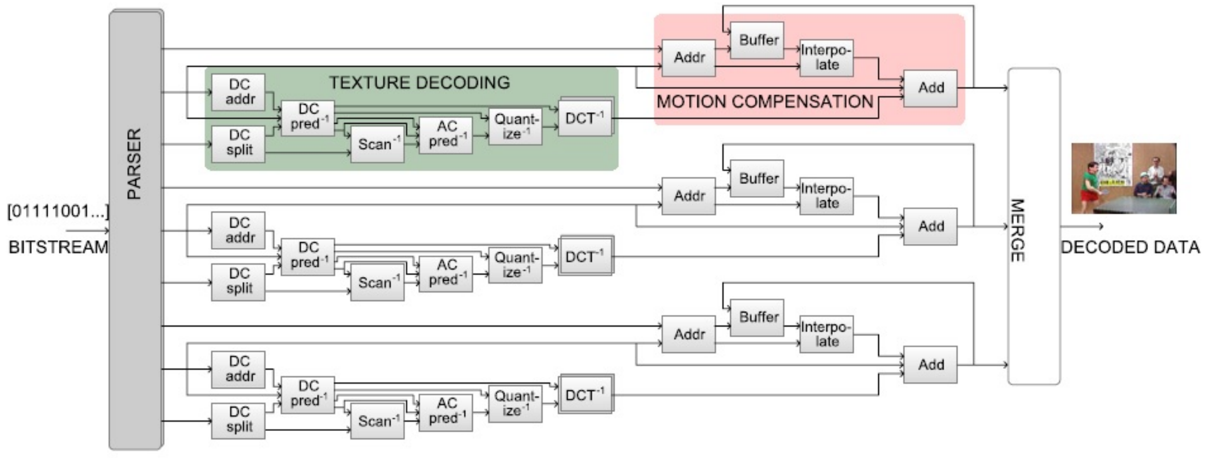


Figure 2.9: An architectural overview of the RVC-decoder. The rectangles represent actors and the arrows represent FIFOs between actors [17].

The decoder consists of a number of actors. These are represented by the rectangles in figure 2.9. As previously mentioned in section 1.2.1, each actor has a distinct functionality and no states are shared between actors. The arrows in figure 2.9 represent FIFOs between actors. Moving out actors or manipulating the connections between actors should not be any problem, since no states are shared between the actors. This means that in order to move an actor, one only has to manipulate the input/output ports of the actors involved. This should be trivial, since our FIFO structures are able to work between different cores. How to exploit the modularity of the decoder is covered in section 4.6.4.

The decoder has a texture decoding and a motion compensation part. Extensive knowledge in what these are, is not required in this project, so they will only be briefly mentioned. The Motion compensation part uses an algorithmic technique to try to predict the next frame, using the previous frame. The texture decoding part on the other hand decodes the compressed image frames according to the MPEG-4 simple profile standard.

2.9 Programming model

As described in section 2.1, it is required that the application has a modular design, where states are not shared between actors. The only way to communicate between actors is to use FIFOs. The generated video decoder fulfilled this property.

2.10 Decoder code structure

The video decoder was handed to us by Cedersjö as one file containing 50 000 lines of C code. It used circular FIFOs for communication between actors.

The relevant parts of the code are covered in the following subsections:

2.10.1 Macro definitions for FIFO operations

In the beginning of the decoder code, the following macros were defined for the FIFO operations:

```
#define WRITE(id, value) buffer##id[((head##id) + tokens##
    id++) % (size##id)] = (value)
#define PEEK(id, pos) (buffer##id[(head##id + pos) % size##
    id])
#define CONSUME(id, n) tokens##id -= n; head##id = ((head##
    id + n) % (size##id))
#define SPACE(id, n) (size##id - tokens##id >= n)
#define TOKENS(id, n) (tokens##id >= n)
```

This design makes it trivial to adapt the decoder to use a custom FIFO implementation, by simply altering the macros.

2.10.2 FIFO buffer definitions

The FIFO buffers were defined as follows:

```
// MV -> MVIN
#define size_b0 512
static int64_t buffer_b0[size_b0];
static size_t head_b0 = 0;
static size_t tokens_b0 = 0;

// RUN -> RUN
#define size_b1 512
static int64_t buffer_b1[size_b1];
static size_t head_b1 = 0;
static size_t tokens_b1 = 0;

...
```

2.10.3 Actor code

The actual code for each actor started after the buffer definitions. Each actor has a method with prefix actor_n as follows:

```
static _Bool actor_n0(void){ ... }
```

This method performs the actor's task and return true if it achieves anything. It will return false if it can not proceed due to for example an empty FIFO, when needing to read from it.

No states are shared between these actors, which means there are no variables that are shared between the actors. The only exceptions are the variables used in the FIFO implementation, which may be used by two actors. This modular property makes it easy to extract an actor from the main code.

2.10.4 Main method and methods for IO

At the end of the file, there is a main method. It simply runs all the actor_n methods repeatedly until all of them have returned false. If this happens, all video frames have been decoded or a deadlock has occurred.

The FIFO buffers 139, 140 and 141 contain the video, width and height outputs from the decoder. The values of these are written to files. FIFO 142 is the last buffer and contains input to the decoder, which consists of a byte stream of compressed video. This byte stream is read from a file. There are 139 additional FIFOs in the video decoder, but the purposes of these are not important for our work.

Chapter 3
Method

This chapter covers the work method for the start-up phase (pre-study) and the development phase of this project.

3.1 Pre-study

The documentation that we had to study, were only essential documents/code in order to understand how the Parallella board [8], Epiphany Architecture[6], Epiphany SDK [10] and RVC-decoder worked. To understand how the RVC-decoder worked, we studied the structure of its C code and identified what each part of the code did.

3.2 Usage of related work

The related work that we had made use of is C code, translated from a RVC-decoder using the tool described in [18].

In order to reach the goal of bringing up data streaming on an Epiphany core array, we have put parts of the RVC-decoder on the Epiphany cores. The parts that did not have space on the Epiphany cores are left on the ARM host. To move parts of the RVC-decoder to the Epiphany cores, we extracted some of the actors in the RVC-decoder and moved these to the Epiphany cores.

3.3 Work methodology: development phase

The work methodology of this project has been to implement functionality in small steps and test these rigorously.

The development steps that we have taken during the project are the following:

- Understand how the Parallella board works.
- Install the SDK and get an example running (in our case "Hello World" and "Mandelbrot").
- Implement FIFO structures for communication between cores.
- Implement the initialization of these FIFO structures.
- Create an adapter to make communication possible between Epiphany cores, ARM cores and between Epiphany and ARM cores.
- Produce a picture on the screen on our own.
- Understand how the RVC-decoder works.
- Test the RVC-decoder in its vanilla state on the ARM host and study the output.
- Draw the output using an Epiphany core.
- Move over actors in the RVC-decoder from the ARM host to the Epiphany cores.

Chapter 4
Implementation

This chapter covers various implementations and topics that are related to these.

The bulk of the chapter is related to our FIFO implementations. The role of the FIFO structures is to serve as communication channels between the actors in the driver application. FIFO structures are used, because the dataflow programming model use exclusively FIFOs.

4.1 Communication between cores

A ringbuffer is a buffer that "wraps around itself", which uses a single, fixed-size buffer as if it was connected end-to-end. There are read and write indices, which point at the current read and write position. When these reach the maximum index, they will re-start from the zero index. When a ringbuffer is empty, the reader and writer are at the same position. When the buffer is full, the writer is one step behind the reader.

A problem worth highlighting is that not all places in the buffer can be used. There will always be one slot behind the reader that is unusable for the writer, which causes the writer to have to wait longer than necessary when the buffer is full. On the other hand, if there was not an empty slot, the writer and the reader would be on the same index as when the buffer is full, which will fool the reader that the buffer is empty. One remedy to this empty slot problem is to introduce a Boolean variable to keep track if the buffer is full or not. We chose to keep the empty slot, because we did not see it as a problem and we wanted the simplest possible solution.

The indices are updated after a FIFO PUSH or POP is completed. It is important that the indices are updated after the action is completed, not during the action, as doing so would lead to either a reader reading while the writer is in the process of writing in that place, or vice versa.

An illustration of the buffer is in figure 4.1.

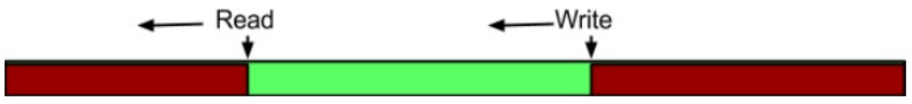


Figure 4.1: Buffer design

One problem with lockless ringbuffers are *race conditions*. Race conditions are undesirable situations that occur when more than one operation is done at once. Examples of such situations are when there are many readers and writers of the same variable, or if updates of the variable comes out of order. In order to avoid race conditions, locks such as *semaphores* are usually introduced.

Our buffer is lockless, but race conditions are non-existent in our implementation for the following two reasons:

- Our FIFO implementation has only one reader and writer per channel, which eliminates the scenario of having many readers and writers.
- The Epiphany architecture guarantees that remote writes are processed at the remote location in the order in which they were sent, which eliminates the out of order update scenario [7].

Figure 4.2 shows an overview of the FIFO structure.

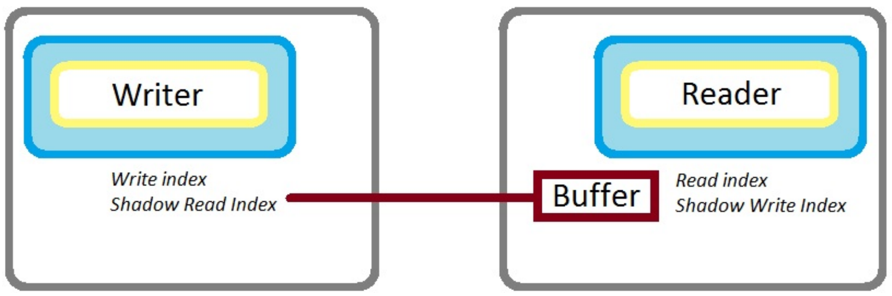


Figure 4.2: This figure shows that the buffer is physically located at the reading side. It also shows the locations of the different pointers.

One property of the Epiphany architecture is that doing a remote write is faster than doing a remote read [19]. In order to exploit this property, we have introduced *shadow pointers*. Shadow pointers are such that they are a copy of the original pointer, which only the owner of the pointer updates.

A side note is that experiments showed that the FIFO implementation could send about 2500-3200 tokens per millisecond between two adjacent Epiphany nodes.

4.2 Initialization

In order for the FIFO to work between two cores, an initialization stage is required, where the reader gets the address of the writer and likewise the writer gets the address of the reader. Both reader and writer need this to update the shadow indices. The writer also needs this when writing to the buffer.

A reader can be used without an attached writer, as it will be empty until a writer is present. However, a writer must know its reader before writing to the FIFO.

This raises the following questions:

- Where can the data structures for the FIFO be placed in memory?
- How can a reader/writer find the address of its corresponding writer/reader?
- When is it safe for an actor to start running?

Since tokens must not be lost, the receiver must be properly initialized before tokens are sent to it. Therefore, an actor cannot execute (and possibly send tokens) unless all its receivers have been initialized. As it is possible to have loops in a dataflow network, a sequence of actor initializations cannot simply be defined.

4.2.1 Memory location of FIFO structures

The architecture of the Parallella imposes some restrictions regarding where we can put data that is intended to be shared between cores.

An Epiphany core has full access to the memory of the other Epiphany cores in the array. The ARM host can also access that memory, as described in section 2.7.4. Thus, it should be possible to place the FIFO structures anywhere in the Epiphany core's local memory (e.g. dynamically on the stack or statically using global variables).

FIFO structures cannot be placed arbitrary on the ARM host, if they have to be accessible by the Epiphany cores. As mentioned in section 2.7.4, the Epiphany cores have access to 32MB shared memory, which is accessible by both the Epiphany and the ARM cores. The host's FIFO structures must therefore be placed in the shared memory to be accessible from the co-processor.

4.2.2 Sharing of FIFO structures' memory addresses

If the FIFOs were global variables they would probably always get the same address, but if cores run different programs, as in our case, they would probably end up on different addresses for different cores. This makes it impractical to only rely on static addresses.

The solution was to make each FIFO structure register its location in a globally accessible memory.

This implementation requires that each FIFO is assigned with a unique index. A table was placed in the shared memory which contains the global Epiphany address of each reader, and the coordinates of the core it is located in (row and column). Though the coordinates are encoded in the global addresses, they were added explicitly to the table, to make the implementation less complex.

The table is implemented using a simple array. Since each core can access the shared memory, an index is all that is needed to retrieve the address of a reader.

The algorithm:

Step 1: The ARM host will initially set all the table's addresses to null.

Step 2: The ARM host starts the programs on the Epiphany cores.

Step 3: When a program starts on a core, all reader structures will be announced, by writing their address and the core's coordinates to the table at their unique index. This will result in a table like the one illustrated in figure 4.3. In the implementation, this index is supplied using a C++ template argument.

Index	Row	Column	Global address
0	1	2	0x04201240
1	0	1	0x00101000
2	1	3	0x04301400

Figure 4.3: Example of a table in the shared memory used for FIFO initialization and synchronization.

Step 4: The program starts listening for readers to be available for its writers. When the reader-address is non-zero in the table at a writer's index, the reader-pointer of the writer will be assigned to the one in the table. At this point, the reader's writer-pointer is set to the global address of the writer. Once all writers have been initialized, the actor can start executing.

It is important that, on each core, step 4 starts after step 3 has completed, otherwise deadlocks may occur.

Extra care is needed when the ARM host is participating, since memory addresses need to be translated to the correct address space. For example, when a writer structure is on the host, its reader-pointer must point to an address in the virtual memory instead of the global Epiphany address, present in the table. To transform a global Epiphany address to a virtual address, accessible by the ARM host the following steps can be performed: Set the 12 most significant bits to 0. A local Epiphany address will remain. Add the core's virtual memory address which can be found as described in section 2.7.4. A separate table with the virtual addresses for each core was placed in the shared memory, to make this translation possible.

4.3 FIFO methods and implementations used in the decoder

4.3.1 General

Instead of using the previously mentioned FIFO methods PUSH and POP, the decoder uses the commands PEEK, WRITE, CONSUME, TOKENS and SPACE. The reasons behind this are the following:

- A traditional PUSH or POP would not be able to provide the functionality to determine when a certain amount of data is available in the buffer, or how much continuous free space there is currently. Replacing PUSH and POP with the five previously mentioned methods, grant the flexibility to wait for a certain amount of free space to be available before starting to write things to the buffer, or to wait until a certain amount of data is available in the buffer before reading it. The latter case is essential when wanting to read in a whole pixel block (for example 16x16) at once.

- The actors in CAL need to be able to inspect the tokens in the input buffers without consuming them, because execution in CAL can be conditional on the values of input tokens.

- An actor may consume several tokens from an input queue during the same step. For example, if an actor consumes 64 tokens in one step, pop() would mean that the read index and the shadow read index had to be updated 64 times, which means 64 add operations, 64 loads and stores, and 64 packets to the producer. Doing this only once is much more efficient locally, and significantly reduces traffic on the internal network.

4.3.2 PEEK(n)

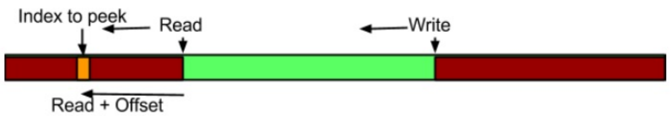


Figure 4.4: An illustration of the logic behind the PEEK method.

The PEEK method is used to retrieve the data from a certain index in a specific buffer. Which buffer the data shall be retrieved from, is specified from an in-parameter that specifies a specific reader object. Since a reader object only has one buffer, the buffer can be retrieved from this in-parameter.

The index of where the data shall be retrieved is specified with an in-parameter that specifies an offset, relative to where the current read index is in the buffer. If this offset specifies an index that is larger than the length of the buffer, the index is computed from the modulus of the buffer size. This gives the desired effect of the buffer "wrapping around itself". This wrapping effect is present in all the FIFO methods, so this will not be mentioned repeatedly from here and onwards.

The PEEK method assumes that the specified offset does not lead to an index in the free space of the buffer. The programmer is assumed to check if the offset can be used, by calling the TOKENS method, before calling the PEEK method.

It is assumed that the code that uses the FIFO methods is generated, and that the code generator performs all required tests to make buffer accesses safe.

4.3.3 WRITE(n)

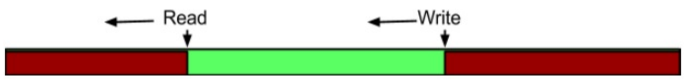


Figure 4.5: An illustration of the logic behind the WRITE method.

The WRITE method is used to write a given value into the next slot of the buffer. Unlike the PUSH method, the WRITE method does not check if the input is valid. It assumes that the programmer has called the method SPACE beforehand to check if there is enough free slots to conduct the write/writes.

The in-parameters to this method are a writer object and the piece of data that shall be written to the buffer. From the writer object, the reader object can be determined, which in turn can be used to determine which buffer to write to. The reason why the reader object has to be accessed is because the buffer is located on the reading side, this means that the writer object does not have a buffer of its own.

4.3.4 CONSUME(n)

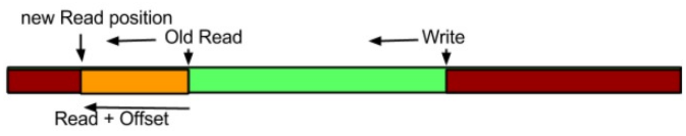


Figure 4.6: An illustration of the logic behind the CONSUME method.

The purpose of the CONSUME method is to move the read index (and the shadow read index). This is usually done after a PEEK method call, because unlike the POP method, PEEK does not increment the read index on its own.

The in-parameters given to the CONSUME method are a specified reader object and a desired offset. The new read index is computed by adding the current read index with the specified offset.

The CONSUME method does not check if the specified offset is valid, as it assumes that the programmer has already checked this with the TOKENS method before calling the CONSUME method. It is also assumed that the data has previously been read with the PEEK method, before calling the CONSUME method to move the read index.

4.3.5 TOKENS(n)

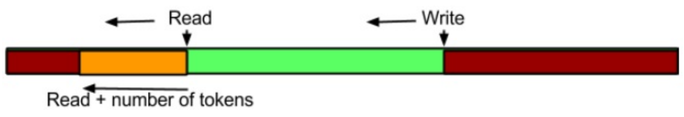


Figure 4.7: An illustration of the logic behind the TOKENS method.

The TOKENS method is used for performing a check if the buffer contains the desired amount of tokens or more. This check is desirable before calling the PEEK method (or a number of consecutive PEEK method calls).

The return value is a Boolean value, which is true if the desired amount of tokens are available, and false otherwise. The in-parameters are a reader object and a number which specifies the desired amount of tokens. The reader object is used to find the current read and shadow write indices.

What this method basically does is that it uses this formula to calculate how many tokens there are:

$$((wPos - rPos) + nbrSlots + 1)mod(nbrSlots + 1) \tag{4.1}$$

If the amount of tokens are equal or greater than the specified value in the in-parameter, then the buffer contains the desired amount of tokens.

4.3.6 SPACE(n)

The SPACE method is used for performing a check if the buffer has the desired amount of free spaces available. This check is desirable before calling the WRITE method (or a number of consecutive WRITE method calls).

The return value is a Boolean value, which is true if the desired amount of free space is available, and false otherwise. Like the TOKENS method, the in-parameters are a reader object and a number which specifies the desired amount of free space. The reader object is used to find the current read and shadow write indices.

The formula used in this method, is basically the inverse of the formula used in the TOKENS method. The exact formula is as follows:

$$(nbrSlots - (wPos - rPos + nbrSlots + 1) mod (nbrSlots + 1)) \tag{4.2}$$

The method will return true if the amount of free tokens are equal or greater than the specified value in the in-parameter.

4.4 Frame buffer

The easiest way to draw on the screen in a Linux operating system is probably to write the pixel values to the frame buffer. From a programmer's perspective the frame buffer is like any other file, which you can open, seek and read [11]. When you write to it, the pixels on the screen will be updated to whatever was written to the frame buffer. Usually "mmap" is used to map the frame buffer to the virtual memory. This way, a program running on the ARM can access the framebuffer. However, as we wanted an Epiphany core to draw on the screen, the physical address was needed instead. The library function "ioctl" can be used to get and set information about the frame buffer, such as resolution, physical address etc.

4.5 Renderer

After a video frame has been decoded, it should be painted on the screen. The last stage of the decoder outputs pixel data in the YUV420 format [17].

The video frame's pixel data was available through a FIFO, as blocks of 16x16 pixels. Likewise, the width and height of the video frame, expressed as number of blocks, are written to two other FIFOs. YUV is a color space consisting of one luma component and two chrominance components. Y denotes the luma or brightness and U and V are the color information. Data in the YUV420 format is grouped by components, illustrated by figure 4.8: First all Y's, then all U's and at last all V's.

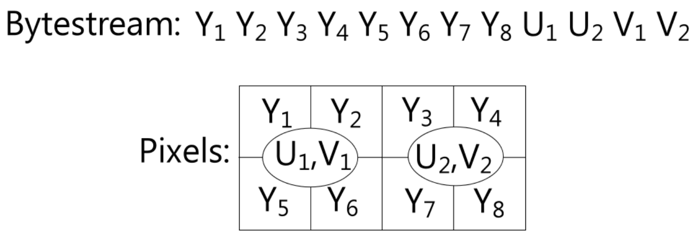

Bytestream: Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_7 Y_8 U_1 U_2 V_1 V_2
Pixels:
Y_1
Y_2
Y_3
Y_4
U_1,V_1
U_2,V_2
Y_5
Y_6
Y_7
Y_8

Figure 4.8: Structure of YUV420

Since RGB is the format in which pixels are drawn into the video RAM, the YUV pixels must be converted. YUV can be converted to RGB using the following formula [13]:

$$R = 1.164 * (Y - 16) + 2.018 * (U - 128) \tag{4.3}$$

$$G = 1.164 * (Y - 16) - 0.813 * (V - 128) - 0.391 * (U - 128) \quad (4.4)$$

$$B = 1.164 * (Y - 16) + 1.596 * (V - 128) \quad (4.5)$$

YUV420 defines a format where one luma value is used for only one pixel and the chroma is shared between four pixels in a 2x2 block, thus each pixel requires 1.5 bytes data.

Assuming that pixels are stored in square blocks, to get the indices of the Y, U and V values in a byte array, the following formulas can be used:

$$i_Y = y * B + x \quad (4.6)$$

$$i_U = y/2 * B/2 + x/2 + B * B \quad (4.7)$$

$$i_V = y/2 * B/2 + x/2 + B * B + B * B/4 \quad (4.8)$$

B denotes the block size which in our case was 16. x and y are the pixel's coordinate within the block.

A naive renderer was implemented. It processes one block at a time, and writes pixels to the frame buffer, as describes in section 4.4, one by one. It uses the formulas mentioned above to extract the YUV values and convert them to RGB. The renderer was run on an Epiphany core, to work in parallel with the rest of the decoder. To make this possible, the host must open the framebuffer, as described in section 4.4 and put the physical DRAM memory address of the framebuffer in the shared memory. RGB values can then be written directly into the framebuffer memory to make it appear on the display.

4.6 Problems and Debugging when integrating the decoder

This section highlights the problems we have had when integrating the decoder on the Parallella board and what we have done to debug these problems. This section contains the following topics:

- Initial testing of the decoder.
- Method of debugging our custom FIFO methods.
- Problems when changing buffer sizes.
- Problem with deadlocks.

As we initially got a faulty decoder, some of these problems will not be reproducible without the same faulty decoder.

4.6.1 Initial testing of the decoder

Our initial test of the decoder was to take a MPEG-4 simple profile video [12] and send it through the decoder, which was running on the ARM host processor. Once the output file was generated, we made an attempt to verify that it was correct. Our process of doing so was to read the file on the ARM processor, then convert the YUV values [13] from the file to RGB values [14], send these values to the Epiphany core array and draw it on the screen.

Once the picture was drawn on the screen, we assumed that the decoder was working. It later turned out that this test was deceptive, as there was indeed a bug in the decoder (as mentioned earlier).

4.6.2 Method of debugging our custom FIFO methods

In order to move the decoder to the Parallella board and some of its functionality to the Epiphany core array, custom FIFOs are needed. These custom FIFOs are the ones written by us and are described in section 4.3.

Our custom FIFOs has the required functionality needed by the decoder, in other words, the PEEK, WRITE, CONSUME, SPACE and TOKENS methods. Initially, we made a tiny test program to test that these methods were working. To test that these methods were working with the decoder, we had to adapt the decoder to use our custom FIFO methods.

To adapt the code to use our custom FIFO methods, we had to do the following:

- Change the macro definitions in the decoder to use our custom FIFO methods.
- Change methods for file IO at the end of the decoder to use the custom macro definitions.
- Extract the original FIFO structures from the decoder and adapt these to work with our custom FIFO class.
- Place these adapted original FIFO structures in our custom C header used for containing shared data. This shared data C header file has data that is accessible by any core on the Parallella board.
- Since our FIFO structures require the FIFO readers and writers to be initialized in order to synchronize, we had to add reader/writer initialization to the extracted FIFOs.

After replacing the original macros with our own, we tested that the decoder was still working and generating the expected output. This assured that the custom FIFOs were working, but as mentioned earlier, this did not assure that the decoder was working properly, as a bug was later found which causes deadlocks in some situations.

4.6.3 Problems when changing buffer sizes

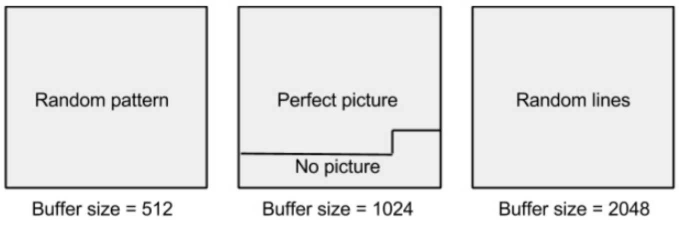


Figure 4.9: A buffer size of 512 gave a consistent random pattern. If the buffer size was 1024, 80% of the first frame was drawn, then came a deadlock. Finally with a buffer size of 2048, random lines were drawn, without any consistent pattern.

Initially, we encountered some strange issues when changing buffer sizes as shown in figure 4.9. Most of these issues were resolved when we got a working decoder, but there still existed issues with certain buffer sizes in situations. One such condition is described under the second point in section 7.3. This made us suspect that there were dependencies on the buffer sizes in the decoder. This theory is, however, not confirmed.

This means that one may want to leave the default buffer sizes alone if possible. If problems similar to that shown in figure 4.9 occurs, one may want to backtrack recent changes in the buffer sizes and revert these to see if it solves the problem.

4.6.4 Problem with deadlocks

At one point, we ran into a deadlock. We noticed that the FIFO methods SPACE and TOKENS were called repeatedly after 80% of the first frame was drawn, which explains why the program deadlocked. By inserting print statements in our FIFO macro calls, we realized that the complete frame was not drawn in the 1024 buffer size scenario. Our guess is that there was not enough space to write to a certain buffer and not enough tokens to read from that buffer.

In our case, it turned out to be a faulty decoder, as the problem was resolved when the decoder was swapped out. If problems similar to this turns up, we think that one should first increase the buffer sizes to see if the fault was an insufficient buffer size, then debug the driver application (the decoder) if the problem still remains.

4.7 Moving over functionality from the ARM processor to the Epiphany core array

Up to this point, all the actors were executing on the ARM host. The actors were moved one by one from the ARM host to the Epiphany co-processor. This section covers general information of how this was done, how the code had to be changed and what limitations had to be considered.

4.7.1 Memory constraints when moving functionality to an Epiphany core

The main problem that we encountered when moving actors from the ARM processor to an Epiphany core was lack of on-chip memory. The on-chip memory on an Epiphany core is 32kB, which was not sufficient to accommodate an actor from the decoder in an unmodified form.

In order to fit an actor on an Epiphany core, some modifications to the actors code had to be done, so as to reduce the size of the actor. The following optimizations have been done to reduce the on-chip memory usage:

- Using for-loops in areas where it was possible to reduce code length.
- Reducing FIFO sizes to reduce on-chip memory usage.

The primary culprit for the excessive code length was the repetitive WRITE and PEEK calls. Usually, there were at least 64 consecutive write calls, each incremented by one step compared to the previous call.

One would hope that the CAL to C translator from [18] would optimize these consecutive macro calls to for-loops. Unfortunately, that was not the case as we had to manually rewrite the consecutive macro calls to for-loops. One before-after example of this is as follows:

Before

```
WRITE(_b138, temp_t28[0]);
WRITE(_b138, temp_t28[1]);
.
.
.
WRITE(_b138, temp_t28[62]);
WRITE(_b138, temp_t28[63]);
```

After

```
for(int i = 0; i<64; i++){
    WRITE(_b138, temp_t28[0]);
}
```

After rewriting the consecutive calls to for-loops, the compiled code size reduced to 22kB from 90kB on one specific actor (specifically decoder_merger420), which means that the code became sufficiently small to fit on the 32kB on-chip memory.

FIFO size optimization

Reducing the FIFO sizes is another way to reduce the overall on-chip memory usage.

The total size of each FIFO is dependent on the following:

- What data type is stored in the FIFO.
- How many slots are in the FIFO.

Some data types require more space per FIFO slot than other data types. One example is that an int64_t FIFO slot requires 8 bytes, whereas a char FIFO slot requires 1 byte [15]. This means that reducing the amount of slots of some FIFOs are more beneficial than others, depending on what data type they contain.

Before reducing the amount of FIFO slots, one has to be aware that some buffers require certain minimum sizes on the FIFO. This means that the FIFO size cannot be reduced below that specific value, or a deadlock would occur.

4.7.2 Moving variables to the shared memory

The actors known as "Decoder-Motion-U-FrameBuff", "Decoder-Motion-V-FrameBuff" and "Decoder-Motion-Y-FrameBuff" had to be modified to work on the Epiphany cores, because they needed to use much more memory than was available in the cores' local memory. They had to use a very large array like:

```
static int16_t frameBuffer_n35v0[549824];
```

The solution to this was to let the array reside on the shared memory by replacing the array with a pointer to an unused location in the shared memory:

```
static int16_t* frameBuffer_n35v0 = (int16_t*)(0
   x8f000000+OFFSET);
```

A consequent of this is that accessing that variable will be slower, as read requests must be sent out and propagate east and then off-chip as described in section 2.7.4. Thus, it might be beneficial to place these actors on a core on the east end of the core array.

4.7.3 Moving FIFOs from the host to an Epiphany core

In order to move a FIFO from the host (that is running the decoder) to an Epiphany core (running an actor), the following has to be done:

- The FIFO definition for the moved FIFO has to be removed from the host and put on the Epiphany core.
- The FIFO initialization for the moved FIFO has to be removed from the host and put on the Epiphany core.
- The actor using the moved FIFO has to be disabled in the decoder. This has to be done in order for the code to be compiled without errors.

Moving of the FIFO definitions

For each moved actor, the FIFO definition of the reader or writer connected to that actor has to be moved to a header file, which is imported to the C-code for the program running on the Epiphany core. This has to be done in order to use these definitions in the C-code that is moved out from the host.

Moving of the FIFO initializations

The FIFO initializations for each moved actor is also moved to the specific Epiphany core, because the FIFO is physically located at the location of the reader. The reason is that the FIFO definitions are only visible to the core that it is located on, which means that only that core can initialize it.

The FIFO initializations are moved to the Epiphany cores the same way as the definitions are. All the FIFO initializations are moved to the same initialization file, where conditional compilation is used to choose which of them to add to a certain core.

Visibility of the moved out FIFOs in the shared data

The shared data is visible for both the ARM processor and the Epiphany core array as described in 2.7.4. There is a FIFO table in the shared data, used for mapping and synchronizing the initialization of each reader and its corresponding writer. Since this FIFO table is in the shared data, and it is an in-argument when calling the initialization method, the host will automatically be able to see and access the moved out FIFOs.

4.7.4 Executing multiple actors on one core

The decoder consists of 40 actors and there are only 16 Epiphany cores available. Thus, each core has to have more than one actor on it, if the whole decoder shall be placed on the Epiphany core array (which is unlikely due to limited on-core memory). In order to run multiple actors on one core, the total size of the compiled code has to be less than or equal to 32kB. This means that if one actor is for example 20kB after compilation, we would be able to room another actor that is 12kB (or less) on the same core.

Our makefile is shown in Appendix C and has a special structure to make modularity easy. This is useful when adding, removing or shuffling around functionalities between cores. The limitation of how many applications you can put on each core is as mentioned the compiled code size. The size of the compiled code must not exceed the size of the internal core memory.

In order to execute many actors on one core, scheduling is needed. We chose to use the *Round Robin* method, as it is the easiest method to implement if performance is not a concern.

Round Robin is the scheduling method that we chose when placing multiple actors on one Epiphany core. The concept of Round Robin is that items are processed sequentially, often beginning again at the start in a circular manner. In other words, this corresponds to a while-loop where each actor takes turns to execute.

Since performance is not a concern in this project, which actors that are placed on the Epiphany cores, how many that are placed on one core, and what scheduling method that is used is not a concern. On the other hand, if performance is a concern, these factors should be taken into consideration.

4.8 Automating the process

To manually extract actor code from the main program and add various FIFO structures for each extracted actor that would run on a core, would be a tiresome and error-prone process. Thus, it was automated. A script was written in C# with the following functionality:

- It loads the RVC decoder C file and:
- Finds all the FIFO buffer definitions (ID, data type, size).
- Finds the code for each actor.
- Turns consecutive WRITE and PEEK calls into loops whenever possible.
- Finds all FIFO reader and writer IDs used by each actor.
- Saves the actor code to separate files for each actor.
- Generates various files for FIFO definition and initialization, as well as code for round-robin execution of actors.

We wanted to be able to easily specify which actors were to run on each core, without changing any code. The C preprocessor was used to enable this flexibility. If-derivative was used to include code only if an actor's name was defined, as demonstrated by the following code:

```
#if (defined(ACTOR_decoder_serialize) ^ defined(
    EXCLUDE_ACTORS))
      // code specific to actor serialize ...
#endif
```

For example, this pattern is used in the generated file actors.h. It contains a function to run actors in a round-robin fashion:

```
void actor_loop(){
#if (defined(ACTOR_decoder_serialize) ^ defined(
    EXCLUDE_ACTORS))
        actor_n0();
#endif
#if (defined(ACTOR_decoder_Merger420) ^ defined(
    EXCLUDE_ACTORS))
        actor_n1();
```

```
#endif
#if (defined(ACTOR_decoder_parser_parseheaders) ^ defined(
    EXCLUDE_ACTORS))
        actor_n2();
#endif
```

The -D option in the compiler GCC is used to predefine a macro to 1. We used this to specify actors to be included. The following example would only include the serialize actor:
`gcc -DACTOR_decoder_serialize ...`
It also made it possible to instead specify the actors that should not be included by defining EXCLUDE_ACTORS. This was useful for the host because it initially had all actors. The following example would include all actors, except serialize:
`gcc -DEXCLUDE_ACTORS -DACTOR_decoder_serialize ...`

During the performance optimizations described in section 5.3, there was a need to use a different FIFO implementation if both the reader and the writer were on the host.The generated file fifo_if.h defines macros that are used to enable different macro definitions, depending on whether a specific FIFO reader/writer is defined:

```
#if (defined(ACTOR_decoder_serialize) ^ defined(
    EXCLUDE_ACTORS))
      #define IF_READER_142(x,y) x
      #define IF_READER_150(x,y) x
      #define IF_WRITER_24(x,y) x
#else
      #define IF_READER_142(x,y) y
      #define IF_READER_150(x,y) y
      #define IF_WRITER_24(x,y) y
#endif
// ...
```

For example, by using the following macros:

```
#define CONCAT(a,b) a ## b
#define IF_R(id,x,y) CONCAT(IF_READER_,id)(x,y)
#define IF_W(id,x,y) CONCAT(IF_WRITER_,id)(x,y)
#define R_AND_W(id , ifyes , ifno )  IF_R(id, IF_W(id ,
    ifyes , ifno) , ifno)
```

`R_AND_W(1, "A", "B")` would expand to the "A" if both the reader and writer 1 were defined, otherwise it would expand to "B". This solution was chosen because it did not required us to change much of the existing code. A problem with this solution is that the code can hard to understand.

The script did not handle FIFOs used for the video decoder's input and output. This required us to manually edit some of these generated files.

Chapter 5
Optimizations

The original task for this project was just to get the decoder to work on the Parallella board. However, some additional efforts were carried out to optimize the performance.

5.1 Floating point operations

To determine if the renderer could be a bottleneck, its functionally was temporally reduced to only consume all incoming tokens and do nothing else.

We used the number of consumed tokens per millisecond by the renderer to evaluate performance of the decoder. This number is proportional to the number of frames per second (FPS). Higher numbers are better.

We noticed that the number of consumed tokens increased. By reviewing the code for YUV to RBG conversion, we discovered that 64 bit floating point numbers were used. We replaced all those 64 bit floating point variables with its 32 bit counterparts as the Epiphany has only native support for 32 bit floating point operations. With no other optimization, the average number of consumed tokens by the renderer per millisecond increased from 103 to 117.

5.2 Memory writes from host to Epiphany

There are a few library functions for copying data to the shared memory or an Epiphany core. On the host, the functions e_write and e_read from e-hal.h are available. We suspected that these functions might be slower than writing directly to the mapped memory, as described in section 2.7.4, for small amounts of data. This was relevant for our FIFOs since those only write one token at a time. With no other optimization, the average number of consumed tokens by the renderer per millisecond increased from 103 to 162. However, together with the floating point optimization, the number increased to 300.

5.3 Accessing shared memory from host

Many actors of the decoder and its FIFOs were on the host. We discovered that having these FIFO structures as global variables in host program, thus only accessible by the host, were faster than having them in the shared memory. With this and the previously mentioned optimization, the average number of consumed tokens by the renderer reached 726 tokens per millisecond. The cause was not investigated. Then we tested the original FIFO implementation used in the unaltered C-file, for those instances where both the reader and writer are located on the host. This improved the measured performance to 792 tokens per millisecond.

5.4 Accessing shared memory from Epiphany

The frame-buffer actors have much larger memory requirements than other actors and the Epiphany core's 32kB local memory is not sufficient for a frame-buffer actor. This means that large FIFOs in the frame-buffer have to be stored in the shared memory. This could possibly become a bottleneck as accessing the shared memory is slower than accessing local memory.

When an Epiphany core accesses the shared memory, it sends a read request to the network and has to wait until the data returns. Doing this repeatedly will build up to more and more latency. Using the DMA engine, one can fetch larger blocks of data using a single request which will eliminate some of the latency. An illustration of this is shown in figure 5.1.

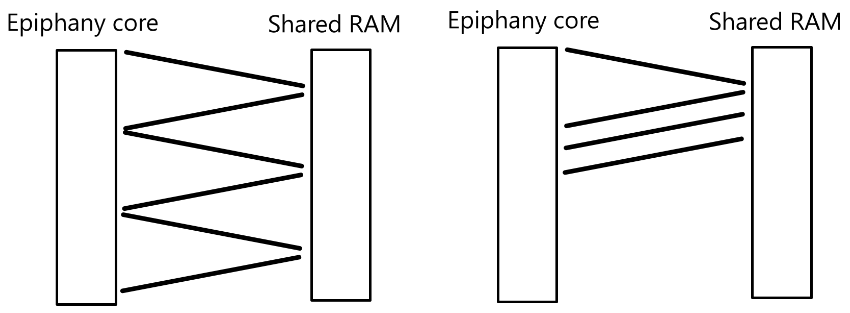


Figure 5.1: Illustration of latency introduced by memory reads without (left) and with DMA (right).

The code for a frame-buffer actor accesses the frame-buffer memory, one 16x16 macroblock at a time. Thus, it should be possible to fetch one macroblock using DMA and cache it in the local memory to improve performance. This approach can be further developed to predict and fetch the next macroblock. These actors also need to write to the shared memory, but it should be much faster than reading since it does not need to wait for anything to return, just “fire and forget”. However, we did not get this to function correctly. Deadlocks seemed to occur randomly, as if the DMA interfered with the FIFO communication.

5.5 Modulo on the Epiphany

There is no instruction for division or modulo for Epiphany. If the base is a power of two, then $x \bmod y = (x \& (y - 1))$ can be used instead, which is faster. As we used modulo in the FIFO implementation it might be beneficial for buffer sizes to be powers of two. Experiments showed that if the compiler is able to figure out that the base is a power of two, then it will automatically optimize this. However, changing the buffer sizes to be powers of two did not affect the performance of the decoder.

Chapter 6
Evaluation

6.1 Lab setup

In order to replicate the lab setup used in this Master thesis, the following are needed:

- A Parallella board with a 16 Epiphany core array, first revision.
- A 16GB (or higher) Micro SD card.
- A cooling fan, to counter the insufficient cooling on the Parallella board.
- A powered USB hub.
- An Ethernet cable.
- A micro HDMI cable.
- A screen.
- A RVC Video decoder, translated to C.
- MPEG-4 simple profile video streams.
- SDK version: esdk.5.13.09.10_linux_armv71.
- Keyboard and mouse.
- (Recommended) A desktop computer, with all standard accessories.

Other versions of the Parallella board and SDK could possibly be used, but there are no guarantees that these would behave exactly the same way as the versions used in this Master thesis.

Using a separate desktop computer for development is recommended, as the Parallella board is very slow to use as a coding platform. It is highly recommended to do the coding on the computer and send the files over to the Parallella board via SCP.

6.2 Performance evaluation

To verify that the decoder functioned properly, a few different video files, with different encoder settings were used. Information about the source video files are summarized in table 6.1 and information about the encodings are listed in table 6.2

Even though the performance in terms of frames per second was not important for this project, some tests were carried out to evaluate performance. To evaluate performance, the video output rate was measured (in bytes per milliseconds). The number of frames per second can be calculated using formula 6.1. 1.5 refers to the average number of bytes per pixel as described in section 4.5

$$fps = rate * 1000/(1.5 * pixelsPerFrame) \tag{6.1}$$

6.3 Results

6.3.1 Performance

All of the files described in section 6.2 were successfully decoded by the application. The measured performance after applying all optimizations is listed in table 6.3.

The optimizations described in section 5 improved the performance significantly. Table 6.4 and figure 6.1 shows the performance for bus_m.m4v using different combinations of optimizations. We also had other test files, but the difference in measured results compared to bus_m.m4v were negligible, which is the reason to why results based on other files were not presented. Pointers refer to the optimizations in section 5.2. Float refers to section 5.1 and Local FIFO refers to section 5.3. Final refers to having all optimizations enabled and using the original FIFO implementation for host-host communications.

The measured performance after moving all the actors to the ARM host, having the output either discarded or sent to the renderer core, is shown in figure 6.2.

Table 6.1: Source video files used for performance evaluation

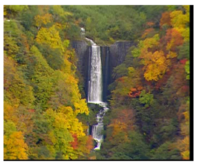
Video file	Thumbnail	Comment
akiyo		Little motion. static background.
bus		A lot of movements.
mobile		
stefan		Sport - a lot of movements.
waterfall		
bunny		

Table 6.2: Encoded video files used for performance evaluation

Video file	Resolution	Bitrate	Frames
akiyo_s.m4v	176x144	512k	90
akiyo_m.m4v	320x180	1024k	90
bus_s.m4v	176x144	512k	90
bus_m.m4v	320x180	1024k	90
mobile_s.m4v	176x144	512k	90
mobile_m.m4v	320x180	1024k	90
stefan_s.m4v	176x144	512k	90
stefan_m.m4v	320x180	1024k	90
waterfall_s.m4v	176x144	512k	90
waterfall_m.m4v	320x180	1024k	90
bunny_720.m4v	1280x720	4096k	90
bunny_1080.m4v	1920x1080	8192k	90

Table 6.3: Measured performance for various files

Video	Data rate (B/ms)	FPS	Macroblock/s
akiyo_m.m4v	776.70	5.1077	2022.6
akiyo_s.m4v	736.48	19.37	1917.9
bus_m.m4v	792.92	5.2147	2065.0
bus_s.m4v	773.77	20.354	2015.0
mobile_m.m4v	794.84	5.2270	2069.9
mobile_s.m4v	726.89	19.121	1893.0
stefan_m.m4v	835.11	5.4918	2174.8
stefan_s.m4v	785.99	20.675	2046.8
waterfall_m.m4v	762.63	5.0152	1986.0
waterfall_s.m4v	723.10	19.021	1883.1
bunny_720.m4v	822.07	0.5948	2140.7
bunny_1080.m4v	846.36	0.2721	2204.1

Table 6.4: Measured performance after various optimizations

Optimizations	Data rate (B/ms)	FPS	Macroblock/s
No optimizations	103.50	0.68	269.54
Float	117.06	0.77	304.85
Pointers	161.52	1.06	420.62
Float & Pointers	300.22	1.97	781.83
Local FIFO (host-to-host)	726.32	4.78	1891.45
All optimizations	792.92	5.21	2064.89

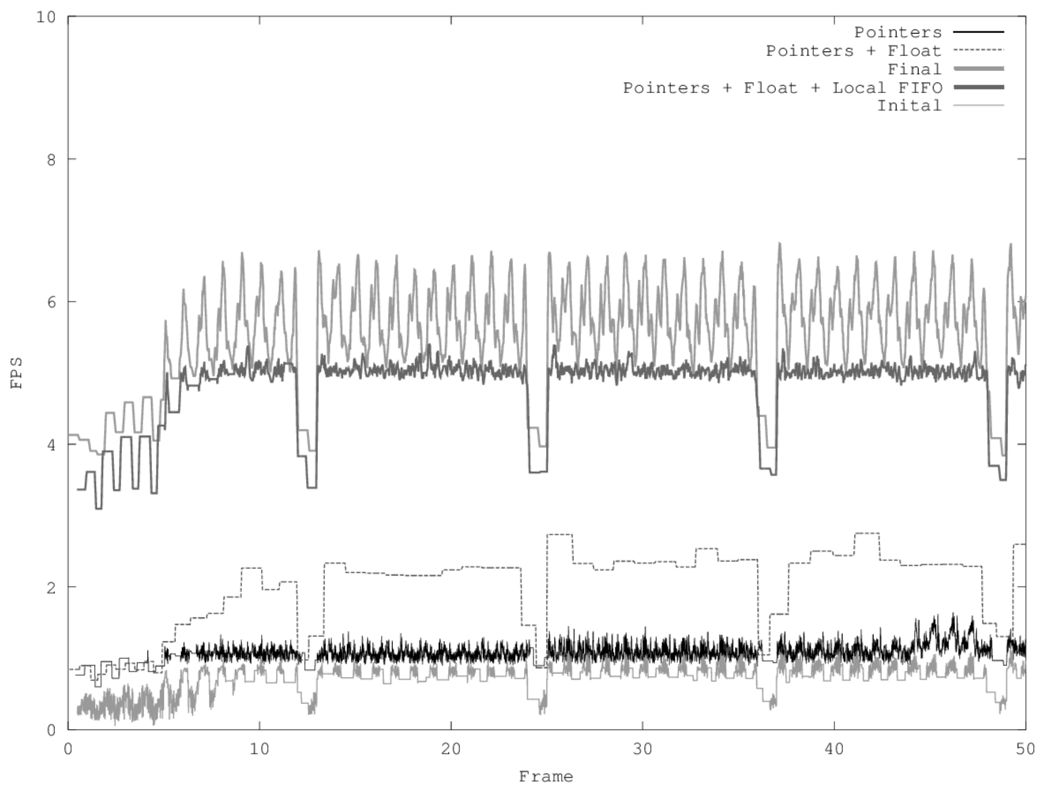


Figure 6.1: Performance with different optimizations.

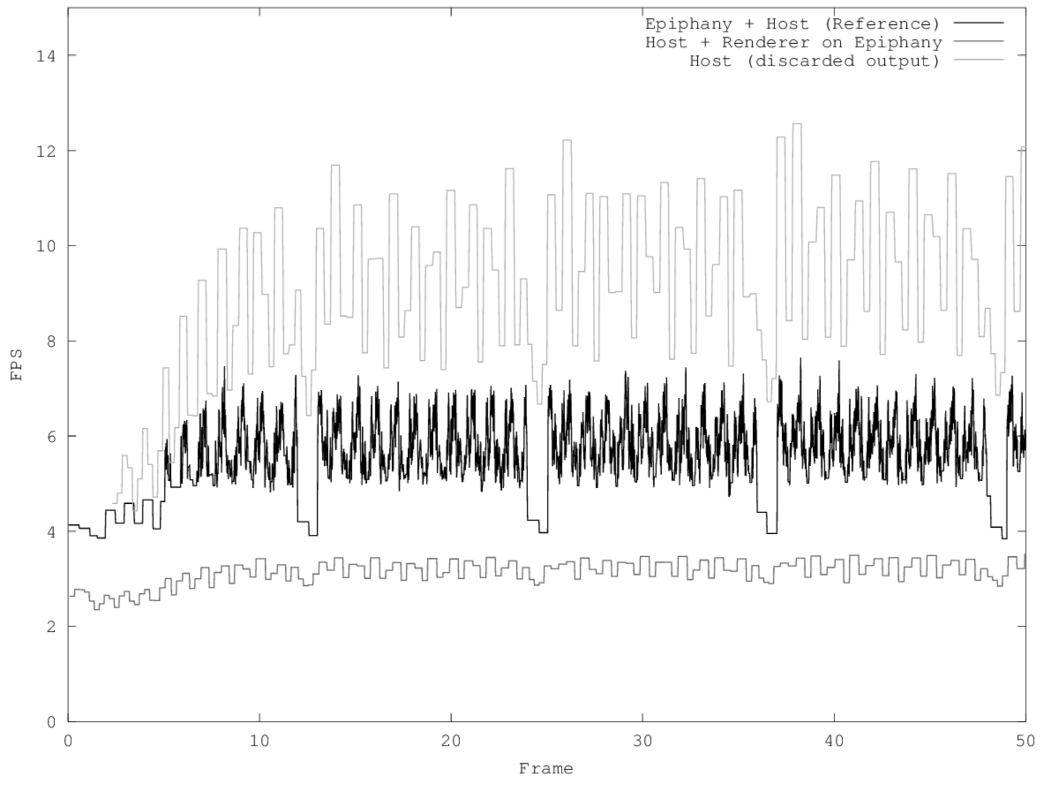


Figure 6.2: Performance after moving all functionality to the ARM host.

Chapter 7
Discussion

Our goal was to get a streaming application to run on a processor array, which we have succeeded with. We have taken a complete real-life application that was generated from an unadulterated MPEG reference code onto a small embedded platform.

Our other goals of gaining a solid understanding of the platform and its capabilities 2.5 2.6 2.7.3 and the software tools that support it [10], developing software infrastructure for communicating between cores 4.1 2.7.4, developing a systematic and safe process for initializing this highly parallel machine 4.2, and creating an interface with off-chip memory 2.7.4 has all been reached and discussed in various parts of this report.

Some parts of this chapter is about performance, which is not one of the goals of this project. All facts and experiments related to performance are extra work that we have chosen to do, as we had extra time after completing the baseline work.

7.1 Performance analysis

The decoding speed is not high enough for real-time video. Figure 6.2 shows that it is faster not to use the Epiphany coprocessor at all.

The performance of the initial implementation was evidently impeded by the host-side's FIFOs and the slow renderer. By improving these, the overall performance increased significantly.

The experiments revealed that sending data from the host to an Epiphany core using our FIFO structures could be limiting the performance. It would be interesting to have the entire decoder on the Epiphany cores, however the limited numbers of cores and memory capacity probably make this impossible. Further more, our FIFOs seemed to be limited to around 2500-3200 tokens per millisecond between two Epiphany cores, which may very well be a problem as a full HD video at 25fps would have to output 51840000 pixels per second.

If increasing the performance of this driver application on this host platform is a goal, we recommend investigating the source of the limited FIFO output. This means finding out if the limit is due to the FIFO implementation, or if it is a hardware constraint. Increasing the limit of the FIFO output might open up possibilities for smooth video frame rates at higher resolutions.

7.2 Implementation choices

Initially, actors were extracted by hand, as described in section 4.6.4. Though it worked, it was a tiresome, time consuming and error prone process. A script was later made to automate this process as described in section 4.8. It also gave us an opportunity to improve the process to make it easier to move actors to different cores. This solution generated many files for just the FIFOs. A design problem was that it became tedious to add additional FIFOs, since many files had to be changed and without care you would risk replacing those changes if you ran the script again. When both the FIFO reader and writer were on the host, we wanted a different FIFO implementation to be used. Still we wanted the ability to easily move actors around. Our solution involved some complex and nested macros. Though it worked perfectly, the nested macros made it harder to find the cause of compilation errors as the error messages became even more cryptic than normal.

It is debatable if this benefit is worth the drawback. In our case, the ease of moving around actors was well worth the more complicated error messages, as moving actors between the host-processor and the co-processor was frequently done at the end of this project. The generated files are usually not altered and it will therefore probably not be a problem. The easy movability of actors is also beneficial for people who are interested of shuffling actors between the cores in order to test what core placements yield the highest performance.

7.3 Unresolved issues

During the course of the work, we encountered several issues that still need to be fixed.

The most noteworthy issues that we have found are the following:

- Using DMA for communication between Epiphany and host introduced seemingly random deadlocks. The cause of this is unknown.
- The actors decoder_texture_Y_IAP and decoder_texture_Y_idct2d will cause the decoder to deadlock if placed on the same Epiphany core. If both of these are on the same ARM core, the deadlock does not occur. The same problem occurs on the U and V counterparts of these two actors. Our guess is that this is not caused by

dependencies between these two actors, as they work flawlessly when both are on the host. The real cause of this problem is not investigated and is still unknown.

- The actor decoder_motion_Y_FrameBuff does not work on an Epiphany core if it has 512 slots per FIFO. It, however, works on an ARM core with these buffer sizes which is strange. If the amount of slots per FIFO is changed to 256, it will work on an Epiphany core. The reason behind this is unknown, but we guess that there is some type of size dependency between this actor and another one, that causes the problem. It is confirmed that the fault is not caused by a memory overflow on the Epiphany core and this bug is not present on the U and V counterparts.

- There is a bug in either the decoder or in the converted bit streams, which causes the last 16*16 square in the last frame not to be drawn. We have traced the problem to that the buffers are not fully emptied in the decoder, but it is still unknown if it is the bit stream or the decoder that causes this particular problem. This problem is only present when converting certain bit streams, which make us suspect that the cause of the problem is the particular bit stream.

7.4 Outcomes

- We have successfully managed to implement data streaming in form of a video decoder on the Epiphany core array. This was one of the main goals of the project.

- We have laid the grounds for future work with data streaming on an Epiphany core array. This could possibly lead to a future project for future students, where they could focus on optimizing the performance.

- We have provided a solid starting point of a benchmark, that could help the ESD group with their research.

- We have managed to move out 22 actors on 13 Epiphany cores, including the frame buffers. This also means that there are three unused cores, which could be used for performance optimization experiments or other things. This is positive, because performance experiments like adding more actors on the Epiphany cores, shuffling actors among cores or adding more cores that draw to the screen can be carried out easier.

- The frame rate is not that impressive. This was expected, as performance is not one of goals in this project. However, the groundwork is done, so whoever wants to take over and optimize the performance could do so.

- No performance was gained by moving out actors to the Epiphany core array. Instead, it turned out to be on the contrary. This is not a negative outcome, as performance is not one of the goals in this project, also the driver application is not optimized to run in parallel as it is made to execute sequentially.

7.5 Research questions

This section discusses the research questions mentioned in chapter 1.1.

How to communicate between multiple cores in a modular way

This basically refer to our FIFO structures. The FIFO implementation and discussion is covered in the sections 4.1 and 4.3.

How to make modularity easy for a stream processed application to run in a parallel environment

The discussion of this topic is covered in section 7.2.

How to communicate between two different processor types

A discussion of how this works is covered in section 2.7.3.

Chapter 8
Conclusions

One main question is what our research can be used for. Our answer is that it could probably be used for anything that is parallelizable. Our product brings together stream processing and parallel execution by running a video decoder on a multi-core machine. Swapping the video decoder for any other modular application would not change the research question of this project, which is why it did not matter how fast the resulting decoder ran.

Our role in this project was to implement a streaming application on a parallel computing machine. Our work is a part of an effort to build a tool chain that compiles stream programs to processor arrays. Besides the automated script for extracting the actors from the main driver application, everything has been done by hand in C, which means that we had to take care of all the communication, synchronizations and memory accesses. If future work results in a tool chain that does these things automatically, then programmers will not have to deal with fine details of the Epiphany platform that we had to deal with. That would mean that the compiler would take care of these complicated details.

Performance was not a concern in this project, but since we had extra time, performance experiments were conducted along with some optimizations. The various performance optimizations that were implemented in chapter 5 had a major impact on the overall performance. The initial performance of the finished product was very low, no performance measurements were taken before the optimizations, but a conservative estimation would be that the optimized performance is a factor ten faster than the original performance.

What the hardware used in this project is good at, is still unknown because it is relatively new on the market. Our initial prediction was that this hardware was not suitable for the task that we had used it for, which was running a stream processed application in parallel. After implementing the performance optimizations, we think that it might have potential after further optimizations as 20 fps is currently reachable at the lowest resolution. If optimizations will lead to an acceptable performance at 720p and upwards is currently unknown. The performance may reach watchable frame rates if more advanced optimizations are carried out, or if the 64 Epiphany core version of the Parallella board proves itself to be more powerful.

Using the Epiphany co-processor with the ARM host, was slower than just using the ARM host on its own. As mentioned in section 7.4, we think that the problem lies in the fact that the entire decoder did not have space on the Epiphany. We still think that the Epiphany chip has great potential, if the whole stream-processing application could be placed on the Epiphany co-processor.

This project has laid the groundwork for future research in the field of combining stream processing and parallel execution. The framework that has been developed in this project is fully functional for what it is made for, which is combining stream processing and parallel computing. The performance, on the other hand, depends mostly on the driver application and the host platform, as changing the video decoder to another application or the Parallella board to another parallel platform will have effects on the performance. How optimized the host application is for parallel execution greatly affects the end performance.

A good topic for a future project would be to continue from where we have stopped. This means optimizing the performance and automating things that we have done by hand, like separation of actors, placing them onto cores, generating the corresponding header files to set up FIFOs and glue code for the round robin scheduling, as all the groundwork is already done.

The work to investigate how to improve the performance and implementing the performance improvements would probably be sufficient for a future project. There are several ways of doing this, some examples are the following:

- One can, for example, move things around on the Epiphany cores randomly and see how placement affects the performance.
- One can use engineering skills and determine what actors are beneficial to have close to each other on the Epiphany cores.
- One can optimize the frame buffer actors and see if any performance is gained.

Bibliography

[1] About the Parallella Project,
https://www.parallella.org/about/,
Visited: 30-03-2015

[2] Epiphany Architecture Reference,
http://adapteva.com/docs/epiphany_arch_ref.pdf
Visited: 25-04-2015

[3] Parallella Project FAQ,
https://www.parallella.org/faq/,
Visited: 30-03-2015

[4] The Parallella Board,
https://www.parallella.org/board/,
Visited: 30-03-2015

[5] The Adapteva Story,
http://www.adapteva.com/about-us/the-adapteva-story/,
Visited: 30-03-2015

[6] Adapteva Datasheet,
http://www.adapteva.com/docs/e16g301_datasheet.pdf,
Visited: 16-02-2015

[7] Adapteva Architecture Reference,
http://www.adapteva.com/docs/epiphany_arch_ref.pdf,
Visited: 15-02-2015

[8] Parallella Manual,
http://www.parallella.org/docs/parallella_manual.pdf,
Visited: 26-04-2015

[9] Parallella 16 Memory Map, Epiphany SDK 5.13.09.10
~docs/parallella_16_memory_map.pdf

[10] Epiphany SDK References,
`http://adapteva.com/docs/epiphany_sdk_ref.pdf`,
Visited: 16-02-2015

[11] The Frame Buffer Device,
`https://www.kernel.org/doc/Documentation/fb/framebuffer.txt`,
Visited: 27-04-2015

[12] ISO/IEC. Information technology – MPEG systems technologies – Part 4: Codec configuration representation. ISO/IEC 23001-4, 2nd Edition, 2011.

[13] Maller, Joe. RGB and YUV Color, FXScript Reference.

[14] Specification of sRGB, IEC 61966-2-1:1999.

[15] Brian Kernighan och Dennis Ritchie, The C Programming Language, second edition.

[16] Johan Eker and Jorn W. Janneck. CAL Language Report. Technical Memo UCB/ERL M03/48, Electronics Research Lab, University of California at Berkeley, December 2003

[17] Shuvra S. Bhattacharyya, Johan Eker, Jorn Janneck, Christophe Lucarz, Marco Mattavelli, and Mickael Raulet. Overview of the MPEG Reconfigurable Video Coding Framework. Journal of Signal Processing Systems, 63:251–263, 2011.

[18] Gustav Cedersjo and Jorn W. Janneck. Software code generation for dynamic dataflow programs. In 17th International Workshop on Software and Compilers for Embedded Systems, 2014.

[19] Gebrewahid, E.; Mingkun Yang; Cedersjo, G.; Abdin, Z.U.; Gaspes, V.; Janneck, J.W.; Svensson, B., "Realizing Efficient Execution of Dataflow Actors on Manycores," Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on , vol., no., pp.321,328, 26-28 Aug. 2014

[20] Endri Bezati, Marco Mattavelli, and Jorn W. Janneck. High-level synthesis of dataflow programs for signal processing systems. In Proceedings of the 8th International Symposium on Image and Signal Processing and Analysis, 2013.

[21] Johan Eker and Jorn W. Janneck. CAL Language Report. Technical Memo UCB/ERL M03/48, Electronics Research Lab, University of California at Berkeley, December 2003.

[22] Zynq-7000 All Programmable SoC, `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`,
Visited: 04-04-2015

[23] The Parallella Board, tech specs, `https://www.parallella.org/board/`
Visited: 04-04-2015

[24] Quick Start Guide, https://www.parallella.org/quick-start/
Visited: 05-04-2015

[25] K. Ramkishor; V. Gunashree, "Real Time implementation of MPEG Video decoder on ARM7TDMI" Proceedings of 2001 International Symposium on Intelligent Multimedia, Video and Speech Processing. pp 522-526. 2001.

[26] Laszlo Felfoldi, MPEG-4 video encoder and decoder implementation on RMI Alchemy. Au1200 processor for video phone applications. , http://www.it2.bme.hu/sites/default/files/secvid_doc.pdf
Visited: 29-05-2015

[27] Video compression, http://www.axis.com/se/sv/learning/web-articles/technical-guide-to-network-video/video-compression,
Visited: 29-05-2015

[28] G.J. Sullivan; J.-R. Ohm; W.-J. Han; T. Wiegand (2012-05-25). Overview of the High Efficiency Video Coding (HEVC) Standard. IEEE Transactions on Circuits and Systems for Video Technology.
Visited: 29-05-2015

Appendices

Appendix A
Manual

This appendix is a manual that covers what one has to do in order to continue from where we have stopped. This manual does not cover the basic hardware and software setup, as one is expected to be able to interpret the "Quick Start Guide" on the Parallella web page [24].

A.0.1 File structures of our repository

In order to get started, one has to know how to get around among our files. The following list will help the reader understand our file structures:

Root folder

The root folder of the repository contains the following:

- A test video sequence in MPEG-4 simple profile. This test sequence is called "video.bit".
- Our FIFO implementation ("fifo.cpp" and "fifo.h").
- "fifo_sizes.h" contains all the size definitions for the FIFO structures.
- "fifo_init.h" contains all the FIFO initializations.
- "actor_common.h", common macros used in all actors.
- "actor_macros.h", common macros used in all actors.
- "actors.h", a file which determines what to include in each program, depending on what actors are chosen. This file also contains a method called "actor_loop", which runs the selected actors in a round robin loop.

- "epiC2.cpp", a program that draws the buffer contents of buffer 141 from the RVC-decoder on the screen. This program needs to run on one Epiphany core.
- "epiphany_actors.cpp", a program that should run on every Epiphany core containing an Actor. How to determine what actors are ran on each Epiphany core is covered later in this manual.
- "host.cpp" runs every part of the RVC-decoder that is not moved to Epiphany cores on the ARM host. This program has to run on the ARM host.
- "Program.cs" is a C# script to extract the actors, common macros between actors, FIFO sizes and FIFO definitions from the RVC-decoder.
- "RVC-decoder.c" is the RVC-decoder in its vanilla state.
- "shared_data.h" contains the data shared between cores.
- "go++.sh", is a shell scrip which transfers all the repository contents to the Parallella board via SCP, compiles the code and runs it.
- "Makefile", a make file which defines what actors shall run on what Epiphany cores.
- "YUVtoRGBConverter.h" is a YUV to RGB converter.
- A folder called "actors", contains headers which represent actors. One header per actor.

Defining what actors shall run on which Epiphany core in the makefile

In order to understand how this works, one has to take a look at these particular rows in the makefile

```
host =
core1 = decoder_Merger420
core2 = decoder_motion_Y_add     decoder_motion_U_add
core3 = decoder_motion_V_add     decoder_texture_U_idct2d
core4 = decoder_texture_Y_IAP
core5 = decoder_texture_V_IAP    decoder_texture_U_IQ
core6 = decoder_texture_Y_IQ     decoder_texture_Y_idct2d
core7 = decoder_texture_V_IQ     decoder_texture_U_IAP
core8 = decoder_texture_V_idct2d
core9 = decoder_texture_Y_IS     decoder_texture_U_IS
   decoder_motion_Y_interpolation
core10 = decoder_texture_V_IS
   decoder_motion_U_interpolation
   decoder_motion_V_interpolation
core11 = decoder_motion_V_FrameBuff
core12 = decoder_motion_U_FrameBuff
core13 = decoder_motion_Y_FrameBuff
```

```
core14 =
core15 =
core16 =
```

In order to define what actor should run on which Epiphany core, one has to alter these rows. One example of how the structure should be is the second row: "core2 = decoder_motion_Y_add decoder_motion_U_add", which defines that "decoder_motion_Y_add" and "decoder_motion_U_add" should run on core 2.

A.0.2 Transferring, compiling and running the code on the Parallella board

The easiest way to do this is to use our shell script "go++.sh", which does all these things serialized. What you will need to alter in this file is the IP address to the Parallella board. After replacing the our Parallella IP address with your Parallella IP address, this script will handle everything else automatically.

Appendix B

CAL to C code example

```
// ACTOR MACHINE
// decoder_serialize
static int8_t count_n0v0;
static int16_t buf_n0v0;
static uint8_t i_n0v1;
static _Bool bit_n0v2;
static void init_n0s0(void) {
        count_n0v0 = 0;
}
static void init_n0s1(void) {
        i_n0v1 = PEEK(_b142, 0);
}
static void init_n0s2(void) {
        bit_n0v2 = ((buf_n0v0 & 128) != 0) ? true : false;
}
static void transition_n0t0(void) {
        buf_n0v0 = i_n0v1;
        count_n0v0 = 8;
        CONSUME(_b142, 1);
}
static void transition_n0t1(void) {
        count_n0v0 = count_n0v0 - 1;
        buf_n0v0 = buf_n0v0 << 1;
        WRITE(_b24, bit_n0v2);
}
static _Bool actor_n0(void) {
        _Bool progress = false;
        static int state = -1;
        switch (state) {
        case -1: break;
        case 0: goto S0;
        case 7: goto S7;
        case 10: goto S10;
        case 11: goto S11;
        case 13: goto S13;
        }
        init_n0s0();
S0:
        if (TOKENS(_b142, 1)) {
                goto S2;
        }
        else {
                goto S1;
        }
S1:
        if (count_n0v0 != 0) {
                goto S4;
        }
        else {
                goto S3;
        }
S2:
        if (count_n0v0 == 0) {
                goto S6;
        }
        else {
                goto S5;
        }
S3:
        state = 7;
        return progress;
S4:
        if (SPACE(_b24, 1)) {
                goto S9;
        }
        else {
                goto S8;
        }
S5:
        if (count_n0v0 != 0) {
                goto S11;
        }
        else {
                goto S10;
        }
S6:
        init_n0s1();
        transition_n0t0();
        progress = true;
        goto S0;
S7:
        if (TOKENS(_b142, 1)) {
                goto S12;
        }
        else {
                goto S3;
        }
S8:
        state = 13;
        return progress;
S9:
        init_n0s2();
        transition_n0t1();
        progress = true;
        goto S0;
S10:
        state = 10;
        return progress;
S11:
        if (SPACE(_b24, 1)) {
                goto S15;
        }
        else {
                goto S14;
        }
S12:
        if (count_n0v0 == 0) {
                goto S16;
        }
        else {
                goto S10;
        }
S13:
        if (TOKENS(_b142, 1)) {
                goto S17;
        }
        else {
                goto S4;
        }
```

```
S14:
        state = 11;
        return progress;
S15:
        init_n0s2();
        transition_n0t1();
        progress = true;
        goto S0;
S16:
        init_n0s1();
        transition_n0t0();
        progress = true;
        goto S0;
S17:
        if (count_n0v0 == 0) {
                goto S18;
        }
        else {
                goto S11;
        }
S18:
        init_n0s1();
        transition_n0t0();
        progress = true;
        goto S0;
}
```

Appendix C Makefile

```
ESDK=$(EPIPHANY_HOME)

ELIBS=$(ESDK)/tools/host/lib
EINCS=$(ESDK)/tools/host/include

ELDF=$(ESDK)/bsps/current/internal.ldf

LIST=core1 core2 core3 core4 core5 core6 core7 core8 core9
   core10 core11 core12 core13 core14 core15
EACTORS=$(addprefix -DACTOR_,$($(@:.elf=)))
HOSTACTORS=$(addprefix -DACTOR_,$(host))

host =
core1 = decoder_Merger420
core2 = decoder_motion_Y_add     decoder_motion_U_add
core3 = decoder_motion_V_add     decoder_texture_U_idct2d
core4 = decoder_texture_Y_IAP
core5 = decoder_texture_V_IAP    decoder_texture_U_IQ
core6 = decoder_texture_Y_IQ     decoder_texture_Y_idct2d
core7 = decoder_texture_V_IQ     decoder_texture_U_IAP
core8 = decoder_texture_V_idct2d
core9 = decoder_texture_Y_IS     decoder_texture_U_IS
   decoder_motion_Y_interpolation
core10 = decoder_texture_V_IS
   decoder_motion_U_interpolation
   decoder_motion_V_interpolation
core11 = decoder_motion_V_FrameBuff
```

```
core12 = decoder_motion_U_FrameBuff
core13 = decoder_motion_Y_FrameBuff

all:    $(LIST:=.srec) epiC2.srec main

main: host.cpp shared_data.h
        ${eval host= ${addsuffix ),${addprefix $$(,$(LIST)
           }}}
        g++ fifo.cpp host.cpp -DHOST -DEXCLUDE_ACTORS $(
           HOSTACTORS) -std=c++11 -o main -I $(EINCS) -L $(
           ELIBS) -le-hal -lrt


epiC2.elf: epiC2.cpp shared_data.h fifo.h
        e-g++ -O3 -funroll-loops -ffast-math -T $(ELDF)
           epiC2.cpp -o epiC2.elf -le-lib



$(LIST:=.elf): epiphany_actors.cpp shared_data.h
        e-g++ -O3 $(EACTORS) -ffast-math -T $(ELDF)
           epiphany_actors.cpp -o $@ -le-lib

%.srec: %.elf
        e-objcopy --srec-forceS3 --output-target srec $(@:.
           srec=.elf) $@


clean:
        rm main *.elf *.srec
```

EXAMENSARBETE Implementing a streaming application on a processor array – A case study on the Epiphany architecture
STUDENTER Jerry Lindström, Stefan Nannesson
HANDLEDARE Jörn Janneck (LTH)
EXAMINATOR Krzysztof Kuchcinski (LTH)

Breaking an application to pieces and running them in parallel

POPULÄRVETENSKAPLIG SAMMANFATTNING **Jerry Lindström, Stefan Nannesson**

A modern way of processing information is to do it in parallel. This master's thesis conducts a case study of how to parallelize an application on a highly parallel platform.

We have worked on making a computer program, meant to run on one processor or run on several processors on a very special computer. This computer is called "Parallella" and has sixteen small processors, but future models could possibly have thousands! This means that it can do a lot of things at once, which defines the term parallel computing. It can be hard to divide a large program to make it work on many processors. We have done just that, but there is a catch!

The program must have been built in a certain way. It must consist of many small programs that mostly do not affect each other. The only way one of these programs can "talk" to one another is by sending packets to the receiver's mailbox.

The program we used was not specifically planned to work on a computer like the Parallella, but it was built as described earlier. This meant that we could split up the program and make it run on several processors. This required us to build special mailboxes, that could work for the communication between programs on different processors.

Mailboxes are of little use if we do not know the addresses. We came up with a system that works like a bulletin board, where each processor could let the other know where it could be found.

To summarize, one can say that we have broken a program to small pieces, placed these pieces on different processor cores, built special mailboxes between the cores and set up a bulletin board to keep track of the mailboxes.

A layman may ask what the point is of doing all this. Our answer is, that this Master Thesis is mainly to aid future research on how to advance in the field of parallel computing.

Another question that may arise is, why is it beneficial to advance in the field of parallel computing? The reason why this is an important area is because regular computers, smartphones, tablets and other common devices are getting more processor cores for every year that passes. In order to harness the power of the extra cores, parallel computing must exist. If our research can aid the development in parallel computing, more processor cores may be able to cooperate effectively.

Are there any consequences that may arise with the increased parallelism in devices and applications? Our answer to this question is that it depends on if you are a user or a developer. From a user's perspective, there will probably be no drawbacks. From a developer's perspective, the drawback is that it is harder to program many cores to cooperate effectively in parallel. More cores means more things to consider, which make it harder to develop applications. Our Master Thesis research tries to counter this problem, by developing structures that simplifies the process of parallelizing certain applications.