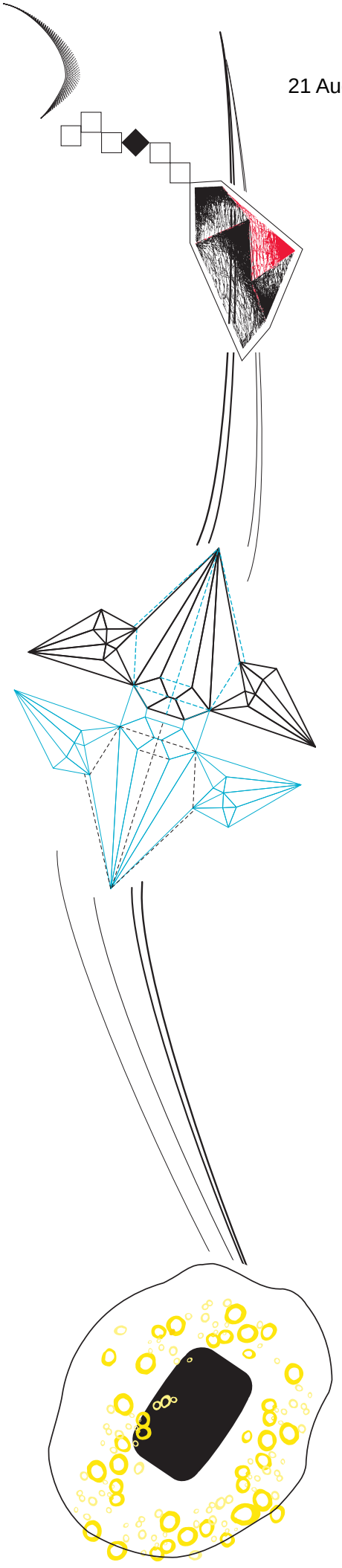


21 August 2015

MASTER THESIS

AN EVALUATION OF THE ADAPTEVA EPIPHANY MANY-CORE ARCHITECTURE

FINAL REPORT

Tom Vocke, BSc. (s0144002)

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)

Chair: Computer Architecture for Embedded Systems (CAES)

Graduation Committee:

Dr.Ir. A.B.J. Kokkeler	(University of Twente)
Ir. E. Molenkamp	(University of Twente)
Ir. J. Scholten	(University of Twente)
Ir. R. Marsman	(Thales Nederland)
Ing. S.T. de Feber	(Thales Nederland)

THALES

UNIVERSITY OF TWENTE.

Contents

Glossary

Term	Meaning
MMAC(S)	**M**illion **M**ultiply-**AC**cumulate operations (per **S**econd)
GFLOP(S)	**G**iga (10^9) **FL**oating point **OP**erations (per **S**econd)
ADC	**A**nalog to **D**igital **C**onverter
MPSoC	**M**ulti-**P**rocessor **S**ystem-**o**n-**C**hip
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
TDP	**T**hermal **D**esign **P**ower
DMA	**D**irect **M**emory **A**ccess
MSB	**M**ost **S**ignificant **B**it
KB / MB / GB	**K**ilo (10^3), **M**ega (10^6) and **G**iga (10^9) **B**yte
KiB / MiB / GiB	**K**ibi (2^{10}), **M**ebi (2^{20}) and **G**ibi (2^{30}) **B**yte
FMADD	**F**used **M**ultiply-**Add (**multiplies two arguments and adds the result to a third)
MS/s	**M**ega-**S**amples / **s**econd. A sample can be of arbitrary bit-length
VLIW	**V**ery **L**arge **I**nstruction **W**ord
RFD	**R**econfigurable **F**abric **D**evice
GPP	**G**eneral **P**urpose **P**rocessor
GP_GPU	**G**eneral **P**urpose **G**raphics **P**rocessing **U**nit
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
eMesh	On-Chip Network of the Epiphany Architecture
eLink	Off-Chip Link of the Epiphany Architecture
eCore	Epiphany Processing Node
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
SDK	**S**oftware **D**evelopment **K**it
FFT	**F**ast **F**ourier **T**ransform
IALU / FPU	**I**nteger **A**rithmetic and **L**ogic **U**nit / **F**loating-**P**oint **U**nit
DSP	**D**igital **S**ignal **P**rocessor

Abstract

In this research we investigate the use of the Adapteva Epiphany Architecture, a modern many-core architecture, as a power efficient alternative to current signal processing solutions in phased-array radar systems. This research is performed using a E16G301 processor, which is a 16-core instantiation of the Epiphany architecture. Several micro-benchmarks are designed and performed to evaluate the processor, network and power performance of the E16G301. Based on the results of these micro-benchmarks a front-end receiver is implemented consisting of a Hilbert filter, a bandpass filter and a beam-former task.

The micro-benchmark results in chapter 5 show that it is difficult to achieve maximum performance for the on-chip and off-chip networks of the Epiphany architecture. For on-chip and off-chip communication only 1445MB/s out of 4.8GB/s and 306MB/s out of 600MB/s were achieved respectively. The low on-chip communication performance is partly caused by an errata item limiting the peak bandwidth to 2.4GB/s instead of 4.8GB/s.

The peak throughput achieved for the Hilbert filter is 53.1% of the theoretical peak performance, for the bandpass filter and beam-former this is 73.1% and 64.2% respectively. The measured power efficiency of are 14.35 GFLOPS/Watt for the Hilbert filter and 19.47 GFLOPS/Watt and 16.22 GFLOPS/Watt for the bandpass filter and beam-former respectively. It is expected that some performance gain is still possible through further optimization of the compiler generated code.

When mapping all the tasks on the E16G301 to form a front-end receiver chain, a sustainable input throughput of 34.4MS/s for four channels was achieved, forming 8 partial output beams for two sets of two input channels. This achieved a power efficiency of 15.8 GFLOPS/Watt, excluding off-chip communication and IO rail standby power.

Finally a summary of observations on the architecture specifics and provided tools is presented. This shows that the deterministic routing scheme has some disadvantages in terms of dependability, as components quickly become critical for correct operation. Also, not much hardware support is provided for health monitoring.

1 Introduction

Radar systems have found many uses over the past decades. For some types of radar systems, such as air-surveillance systems, tracking of multiple possibly fast moving objects is desired. To achieve this, these systems typically use phased-array antenna's which support fast tracking, allow monitoring multiple directions simultaneously, and feature high antenna gain.

Phased-array antennas are made out of many smaller antenna elements. By applying a phase-shift to all individual antenna elements they can be made to sum constructively for signals coming from certain directions, and destructively for others. The process of applying the phase shifts to set the antenna radiation pattern is known as beam-forming, which can be done in both the analog and digital domain.

When beam-forming is done in the digital domain, it is possible to form multiple beams simultaneously by applying different phase-shifts on the same set of data. However, digital beam-forming requires very large amounts of signal processing. Part of this processing is typically done very close to the antenna elements to reduce the bandwidth of the data that needs to be transported through the system. This introduces additional thermal and power constraints.

These large amounts of data and strict thermal constraints in phased-array radar systems require an efficient hardware solution. Currently this is mainly the domain of FPGA devices complemented by DSP processors, typically followed by of-the-shelf GPP/GP-GPU processing platforms. For a number of application domains, such as mobile radar systems, the current solutions are not always appropriate because of high power consumption, large development effort, and/or high component costs.

Recently, platforms with multiple processors and on-chip networks are rapidly gaining popularity. These Multiprocessor Systems-on-Chip (MPSoC) feature highly integrated hardware to offer a cost and energy efficient solution for the processing demands of an application. A special class of MPSoC are the many-core processors, which feature large numbers of processing cores to be able to fully exploit task level parallelism.

This thesis presents an evaluation of the Adapteva Epiphany Architecture [1], a recent many-core architecture, that could be a power efficient alternative to current solutions. The focus of this report is on the front-end radar processing stages, where power and processing requirements are most strict.

1.1 HISTORY AND MOTIVATION

Prior to this project, two large research projects have been conducted in cooperation with Thales, where, among other topics, the possibility of using many-core processors in high throughput streaming applications was investigated.

1. **CRISP [2] (Cutting edge Reconfigurable IC's for Stream Processing, 2008-2011):** The CRISP project researched optimal utilization, efficient programming and dependability of reconfigurable many-cores for streaming applications.
2. **STARS [3] (Sensor Technology Applied in Reconfigurable Systems, 2011-2014):** STARS is a large research project with several themes. One theme focused on how many-core architectures could be used in reconfigurable high-throughput streaming systems.

During these projects a custom many-core platform was developed, called the Recore Xentium RFD. This platform was used to demonstrate the use of MPSoC in a scalable, dependable solution for stream processing. A beam-former application was implemented as a case-study.

The focus thus far has been mostly on scalability, reconfigurability and dependability properties of MPSoC. What remains to be investigated is how the raw performance and power efficiency of MPSoC systems will compare to that of current FPGA solutions. Although the Recore Xentium RFD could be used to research this further, it has a few limitations that introduce the desire to also look at other architectures. The most important are listed here:

- **Precision:** The Recore Xentium RFD excels at 16bit fixed point arithmetic, but lacks floating-point performance. Future applications might require larger bit-widths or floating point support.
- **Performance:** The silicon design was realized in a non-competitive (90 nm) technology [4], limiting the clock speed and power efficiency.
- **Programming Model and Tool Support:** Software is currently hand-written using C and assembly. This might give an unrealistic view of the development effort required for many-core systems in general.
- **Architecture:** Currently, only the Recore Xentium RFD has been evaluated for the use in radar systems. Researching a different architecture could provide more information on the important aspects of many-core architecture for this application.

A survey [5] of possible alternative architectures was made by Thales during the STARS project, of which the results are listed in appendix A (p.73). Here, the Adapteva Epiphany G4 shows the best power-efficiency of the processors listed, is tuned for 32bit floating point performance and features a C/C++ and OpenCL development environment. Unfortunately, the G4 was not available for purchase so instead the E16G301, a 16-core version of the same architecture, is used for the research.

1.2 RESEARCH GOALS

The goal of this research is to answer the following question:

“Can we use the Adapteva Epiphany Architecture as a power efficient alternative in the front-end processing chain of phased-array radar systems”

To answer this question, we will focus on three main topics for this research:

1. **Performance:**
   - What throughput and power efficiency can be expected from the Epiphany Architecture?
   - How does this throughput and power efficiency compare to existing systems?
2. **Programming Model and Tool Support:**
   - How can we best develop software for the Epiphany architecture?
   - What are the strengths and weaknesses of the provided development environment?
3. **Architecture:**
   - What is the impact of the differences between the Epiphany Architecture and the Recore Xentium RFD on earlier results obtained in the CRISP and STARS projects?

1.2.1 Thesis Outline

To answer the questions stated above, first, the general approach taken in this research is presented in more detail and the relevant background information is given. This includes a description of beamforming in general, the domain requirements and an overview of the Epiphany and Recore Xentium RFD architectures.

Next, related work is presented and several missing benchmarks are identified and implemented. The results of these benchmarks are discussed in chapter 5. In chapter 6, the design and optimization of several tasks typical for the front-end receiver chain are discussed.

Based on the micro-benchmark results, and the achieved performance of the individual front-end receiver tasks, a mapping of the front-end receiver on the E16G301 processor is presented. The design choices and performance numbers for this implementation can be found in chapter 7.

We end this report with a short discussion on the experiences with the provided tools and libraries, and the benefits and limitations of this architecture for reconfigurable systems. All the results are summarized in the conclusions, and finally, recommended future work is presented.

2 Research Approach

2.1 PERFORMANCE ANALYSIS

In order to evaluate the Epiphany architecture, first, the theoretical peak performance is determined. This is done based on the specifications provided by Adapteva, and will serve as a reference point throughout this research. After this, the maximum achievable throughput and power efficiency of the Epiphany architecture are evaluated on the E16G301.

The main goal of evaluating this performance is to allow for a comparison to existing solutions. Comparing performance of computing systems is typically done using industry standard benchmarks. These benchmarks can be roughly divided into two classes:

- **Micro Benchmarks:** Specifically tuned software is used to test individual features or components of a system. For instance, a program with specific memory access patterns can be used to test cache performance. Micro benchmarks are typically used for verifying and tuning the performance of separate system components.
- **Application Benchmarks:** A set of full applications or application kernels is used to test the overall system performance. For an application benchmark to be useful, it is very important that it is representative of the desired application domain, and that it is implemented in a similar way to how the real-world application would be implemented [6].

For this research, first, several micro-benchmarks will be built to estimate the achievable percentage of the peak performance presented by Adapteva for real-world applications. These benchmarks will focus on the performance and power efficiency of the on-chip network, off-chip network and individual processor components separately. The details of related research, and the design of these benchmarks are presented in chapter 4.

Actual application performance will depend on many factors such as partitioning, scheduling, memory requirements, data-dependencies and compiler efficiency. It is not uncommon to only achieve small fractions of peak performance for real-world applications, as shown in [7] where for a 2D-Fast Fourier Transform (FFT) application, only 13% of peak performance on the Epiphany architecture is achieved, versus 85% of peak performance for a matrix multiplication problem. This means the result of the micro-benchmarks alone will not say much about the overall performance of an application.

For a good estimate of the achievable throughput a suitable application benchmark is required that can be run on all the platforms that are to be compared. Implementing new software on the Recore Xentium RFD and other platforms will be very time consuming and outside the scope of this research, therefore this research will focus on applications that have already been implemented. One case that is often used to represent the front-end radar processing domain, and has been implemented on both the Recore Xentium RFD and other platforms, is the front-end receiver, consisting of channel filters and a digital beam-former [8].

For this research the digital beam-former and channel filters are implemented on the Epiphany architecture. The beam-former and filter background and requirements are introduced in chapter 3.1. The implementation and mapping of the beam-former and channel filters are discussed in chapter 6 and chapter 7.

2.2 PROGRAMMING MODEL AND TOOL SUPPORT

The following paragraph is taken from [1], and describes the programming model for the Epiphany architecture as presented by Adapteva:

> “The Epiphany architecture is programming-model neutral and compatible with most popular parallel-programming methods, including Single Instruction Multiple Data (SIMD), Single Program Multiple Data (SPMD), Host-Slave programming, Multiple Instruction Multiple Data (MIMD), static and dynamic dataflow, systolic array, shared-memory multi-threading, message-passing, and communicating sequential processes (CSP)”

Essentially this says; anything is possible. It is left to the programmer to determine what best suits the need for the desired application. In this research we will investigate the advantages and disadvantages of several of the mentioned design approaches to determine what can be best used for the digital beam-former in chapter 7.

Adapteva provides an Eclipse C/C++ based development environment. Alternatively, Brown Deer technology provides an OpenCL environment with Epiphany architecture support. In this research, only the C/C++ based environment will be used since the Brown-Deer environment uses the libraries and compiler of the official Adapteva SDK in the background, and will therefore not likely introduce any performance benefits.

2.3 ARCHITECTURE

During the STARS and CRISP project, less tangible properties of a system such as scalability, reconfigurability and dependability were investigated. These properties are important in real-world solutions where products need to be verifiable, are used for multiple products for cost reasons, and need to be relied upon in critical systems.

In [9], the definitions of a few key properties are presented, which are summarized below:

- **Reconfigurability:** The ability of a system to change its behaviour at design or run-time by changing hardware and/or software configurations.
- **Dependability:** This describes the trustworthiness of a system. It consists of both health monitoring capabilities and robustness of the design.
- **Scalability:** The ability of a system to handle growing amounts of work gracefully or its ability to be enlarged to accommodate that growth.

The effect of the differences between the Epiphany architecture and the Recore Xentium RFD on these properties will be discussed in chapter 8.

3 Relevant Background

In this chapter the application domain and Epiphany architecture are introduced to better understand the relevance of some of the topics discussed in this research. First, the basic concepts of phased-array radar systems, beam-forming and their typical requirements are introduced. Then an overview of the Epiphany architecture is presented. A more detailed description of the Epiphany architecture can be found in the Epiphany architecture reference [1].

3.1 RADAR PROCESSING AND BEAM-FORMING

The processing requirements in radar systems depend strongly on the application. In [9], an overview is given of several common radar applications and their processing needs. The processing solutions for these systems are divided into a transmit and a receive chain. This research will focus on the front-end processing stages in the receive chain of a radar, throughput and processing requirements are most strict.

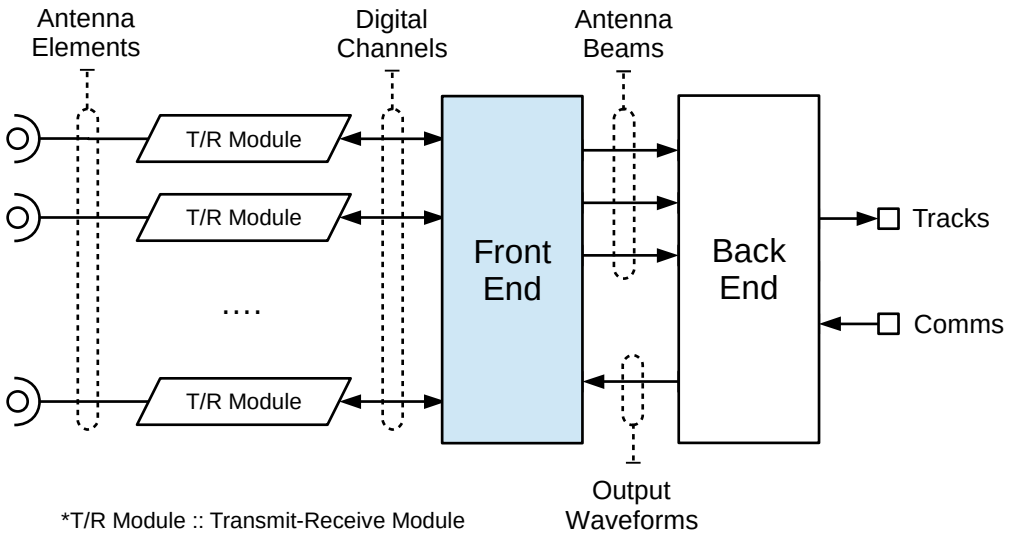


Figure 1: Phased-array antenna receive chain processing overview [9]

Figure 1 shows an overview of a typical phased-array radar receiver processing setup. Here each antenna element is connected to a transmit-receive module that filters and mixes the antenna signal to an intermediate frequency in the analog domain. These modules also convert the resulting signals to the digital domain. The digital streams are sent to the processing front-end, where channel filtering is applied and beam-forming is done. The resulting beams are sent to the processing back-end where algorithms for detection, extraction, classification and tracking are implemented [9].

3.1.1 Channel Filtering: Hilbert and Bandpass Filter

The first step of the digital channel filtering in the front-end is to transform the real-valued sample streams from the transmit-receive modules into complex representations using the Hilbert transform [10]. Complex-valued samples offer more convenient access to important signal characteristics such as instantaneous amplitude, phase and frequency. After the Hilbert filter, a bandpass filter is applied to select the desired frequency band and equalize small differences in the individual antenna elements.

A decimation factor can applied in both filter stages, meaning all input samples are used for the filter operation, but only part of the output samples are computed. Decimation decreases the filter output rate, thereby decreasing the filter processing requirements while preserving more information of the original signal in the output signal than would be achieved if the sample-rate were simply reduced.

On the Recore Xentium RFD, both the Hilbert and the bandpass filter are implemented through direct-form convolution. Efficient FFT based convolution algorithms exist [11], however, additional memory is required for the storage of intermediate frequency domain signals and for small signals lengths, the performance gain is minimal. Since memory is limited on the Epiphany, the extra FFT memory footprint is undesirable. Also decimation will likely result in a relatively small amount of samples to process. For these reasons a direct-form convolution approach is used in this research, similar to the Recore Xentium RFD implementation.

3.1.2 Digital Beam-forming

The beam-forming stage is responsible for introducing the desired phase-shifts for phased-array operation, and for summing all the antenna output channels to form the beam output. The phase-shifts can be applied independently to all channels and the ordering of summation does not matter. This allows concurrent processing of the workload by dividing the channels, and splitting up the channel summation in multiple stages. An example of this is shown in Figure 2.

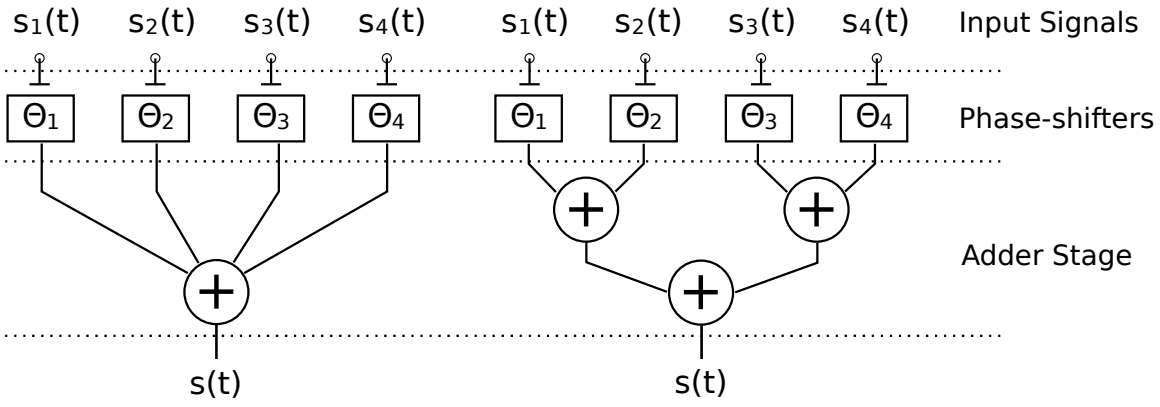


*Figure 2: Beam-former variations. **Left**: Single process adds all channels.*
***Right**: Summation is spread over two stages, three processes*

The beam-former works on a complex valued input stream. Equations (1) and (2) show that multiplying a complex sample with a complex coefficient results in the magnitudes being multiplied, and the phases being added. In practice, this means a phase-shift can be realized by multiplying each complex sample with a suitable complex coefficient.

$$z_1 = r_1 \cdot e^{j \cdot \varphi_1} \quad \vdots \quad z_2 = r_2 \cdot e^{j \cdot \varphi_2} \tag{1}$$

$$z_1 \cdot z_2 = r_1 \cdot e^{j \cdot \varphi_1} \cdot r_2 \cdot e^{j \cdot \varphi_2} = r_1 \cdot r_2 \cdot e^{j \cdot (\varphi_1 + \varphi_2)} \tag{2}$$

3.1.3 Front-End Throughput and Processing Requirements

In [9], the processing requirements are specified in terms of throughput and the number of required operations per task. Although the requirements for the front-end processing differ strongly between applications, all the receive chains perform the same tasks. The requirements for a specific application depend on the input sample rate (F_S), the number of input channels (N_C), the filter decimation factors (D_{HF}, D_{BPF}) the number of filter taps (N_T) and the number of output beams (N_B). For both the throughput and processing requirements a relation between the above parameters and the requirement can be determined.

Throughput Requirements

The throughput requirements per task depend on the output rate of the preceding task, as shown graphically in Figure 3. Here HF and BPF are the Hilbert and bandpass filter respectively, and BF denotes the beam-former stage. The output throughput of each task can differ from the input throughput depending on filter decimation factors and the number of beams that are formed. The throughput requirements between all tasks scale linearly with the input sample rate.

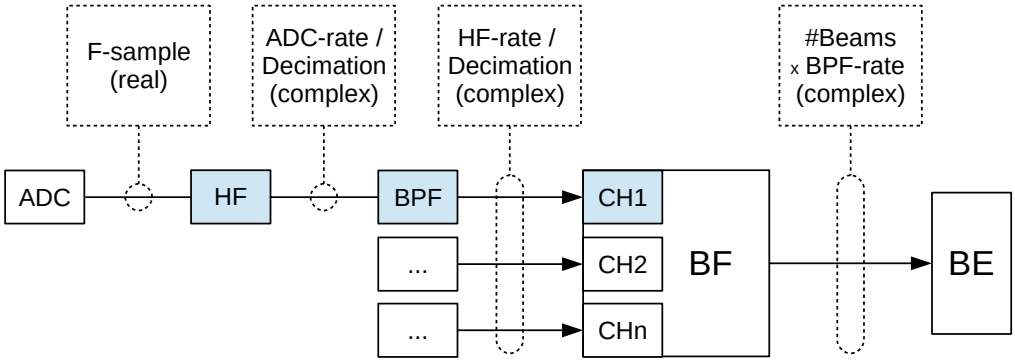


Figure 3: Throughput requirements per tasks in the front-end receiver processing chain. (HF=Hilbert filter, BPF=Bandpass filter, BF=Beam-Former, BE=Back-end)

Processing Requirements

The processing requirements per task depend on the required throughput per task, and the number of operations required to process each sample for a given task. In [9], some estimates are made on the processing requirements based on a direct-form 32 tap FIR filter implementation of the filters. These are defined as the required number of multiply-accumulates (MACs), and are determined by using the following costs for processing operations:

- **Complex Multiply(M_{CMUL}):** 4 (multiply) + 2 (add) = 4 MACs
- **Complex Add (M_{CADD}):** 2 (add) = 2 MACs
- **Complex Multiply Accumulate (M_{CMAC}):** 4 (multiply), 4(add) = 4 MACs
- **Hilbert Filter FIR tap (M_{HF_FIR}):** 2 (multiply) + 2(add) = 2 MACs
- **Bandpass Filter FIR tap (M_{BPF_FIR}):** 1 (CMAC) = 4 MACs

Based on the costs of these operations, the processing costs per task can be defined. This is done in equations (3), (4) and (5) where C_{HF} C_{BPF} and C_{BF} denote the number of MACs required for each task, the M_{HF_FIR}, M_{BPF_FIR} and M_{CMAC} terms represent the number of MACs requires for a specific operation, and N_T is the number of filter taps.

$$C_{HF} = N_C \cdot \frac{F_S}{D_{HF}} \cdot N_T \cdot M_{HF\,FIR} \quad (3)$$

$$C_{BPF} = N_C \cdot \frac{F_S}{D_{HF} \cdot D_{BPF}} \cdot N_T \cdot M_{BPF\,FIR} \quad (4)$$

$$C_{BF} = N_C \cdot \frac{F_S}{D_{HF} \cdot D_{BPF}} \cdot N_B \cdot M_{CMAC} \quad (5)$$

Note that the Hilbert filter and Bandpass filter implementations differ in processing costs since the Hilbert filter operates on a real-valued input stream, and the bandpass filter operates on a complex input stream as shown in Figure 3.

Equations (3), (4) and (5) will be used to determine the absolute peak performance we can achieve for the front-end receiver on the E16G301 processor. In section 3.1.3 an initial peak performance estimate for the E16G301 is given, which is used as a reference throughout this report. Before this is discussed we first introduce the Epiphany architecture, and the Parallella board that is used for this research.

3.2 EPIPHANY ARCHITECTURE

The Epiphany architecture is a recent many-core architecture design by Adapteva aiming at power efficiency. It features a homogeneous grid of processing nodes, known as mesh-nodes, connected by a Network-on-Chip (NoC). Each node consists of a processor core, local memory, a network interface and a DMA engine. For this research a 16-node variant called the E16G301 is used, which comes on a small computer board called the Adapteva Parallella [12].

The Parallella uses a Xilinx Zync 7000 series device which contains two hardware ARM processors, several peripherals and some FPGA fabric. The FPGA is used to instantiate a link to the E16G301 processor. An overview of this board and the E16G301 is given in Figure 4.

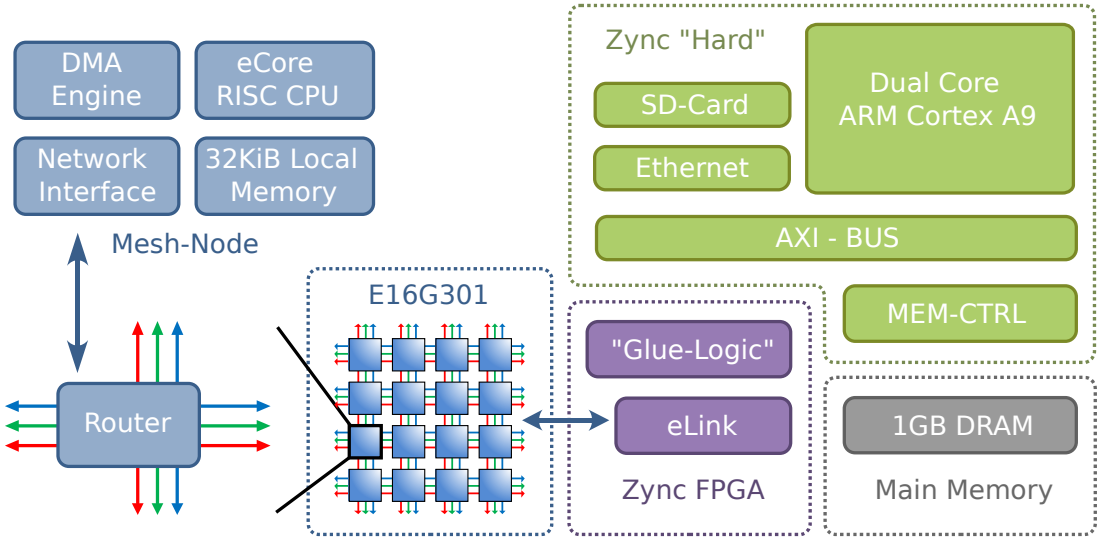


Figure 4: Adapteva Parallella Board and Epiphany architecture overview

3.2.1 Memory Layout

All mesh-nodes in the Epiphany architecture share the same 32 bit address space, and it is possible to read and write directly to each node's local memory. Each node is assigned a 1MiB block of memory in this address space. Currently, for the E16G301, only 32KiB of this 1MiB space is used per node, likely to reduce the chip-area requirements.

The twelve most significant bits in an address are used to identify the node the memory belongs too. To avoid having to recompile software every time it is run on a different node, the 1MiB address space of a node is locally aliased to the [0x00000000 … 0x000FFFFF] address range. This memory layout is shown more clearly in Figure 5.

Part of the 1GiB DRAM on the Parallella board is mapped to the Epiphany address space, and is accessible by both the ARM cores and the E16G301. This allows for communication between the E16G301 and the ARM processor. It is also possible for the ARM processors to write and read directly from the local memory of the mesh-nodes. All transactions with local and shared memory on the E16G301 are handled by the Network-on-Chip.

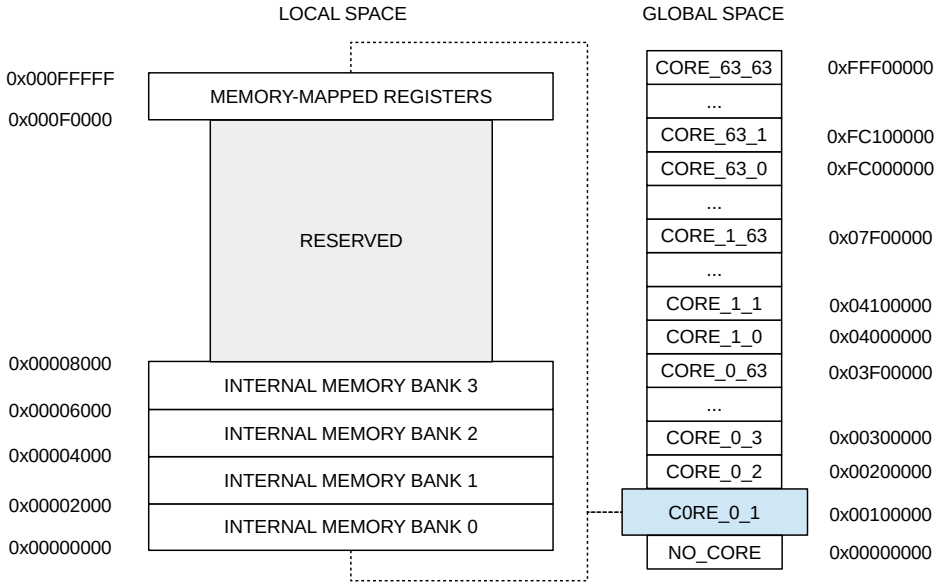


Figure 5: Epiphany architecture local and global memory address space

3.2.2 eMesh Network-On-Chip

All mesh-nodes are connected by a packet-switched Network-on-Chip. Data is sent as 8-64bit packets accompanied by a destination address. The network consists of three independent 2D-Mesh networks, each with a specific function:

1. **cMesh:** The cMesh is used for write transactions to on-chip mesh-nodes. It has a maximum bandwidth of 4.8 GB/s up, and 4.8GB/s down in each of the four routing directions.
2. **rMesh:** The rMesh is used for all read transactions. Read transactions do not contain any data, but instead travel across the rMesh until the destination node is reached. Here, a write transaction is initiated to transport the data back to the requesting node. The rMesh can issue one read transaction every 8 clock cycles, resulting in 1/8th of the maximum cMesh bandwidth.
3. **xMesh:** The xMesh is used for write transactions destined for off-chip resources and for passing through transactions destined for another chip in a multi-chip configuration. It is split in a North-to-South and an East-to-West network. The bandwidth of the xMesh is matched to the off-chip links of the architecture (600MB/s up, and 600MB/s down for the E16G301).

The nodes in the Epiphany architecture only see the global memory space and are not aware of the specifics of this network. Transactions destined for off-node memory addresses are put on the network and routed to the proper destination automatically. The processor pipeline is stalled accordingly if it has to wait on the network due to congestion or long routes.

The on-chip network is terminated at the edges with an off-chip interface known as the eLink. Multiple chips can be connected together to form systems with larger core counts. Both the G4 and the E16G301 feature four eLinks, one on each side of the eMesh (north, south, east and west).

Network Routing Scheme and Coordinate System

The network is organized as a grid of a maximum of 64x64 mesh-nodes, with a 1MiB address spaces assigned to each node. Each node is identified by its top twelve bits, which are used to represent a (row,column) coordinate. The first 6 bits represent the row, and the lower 6 bits represent the column of the destination node, using the numbering conventions shown in Figure 6:

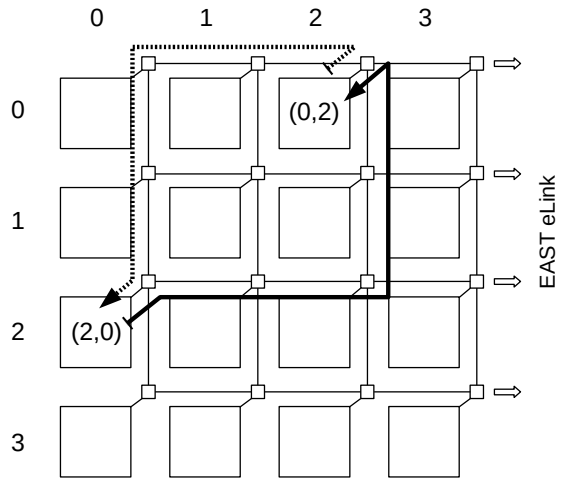


Figure 6: E16G301 Coordinate system conventions and example routes for node (0,2) and (2,0) up and down traffic

The mesh networks use XY routing, meaning that a packet is first routed east or west (x-axis) until it reaches the destination column, and is then routed north or south (y-axis) until it reaches the destination row. The XY routing scheme is deterministic (routes are fixed) and deadlock free [13].

A consequence of this routing scheme is that transactions between nodes take different routes depending on which node is sending the data (shown in Figure 6). Also, transactions can only arrive at an off-chip interface when the top twelve bits of the destination address represents an off-chip row or column. This means that for the Parallella board where only the east eLink is available, only the memory DRAM locations that represent a node east of the E16G301 nodes can be accessed. This leaves the addressable DRAM memory space fragmented.

3.2.3 The Epiphany eCore

The Epiphany processing cores (eCores) are custom dual-issue RISC processors with a 64-word register file. Each eCore features an integer arithmetic logic unit (IALU) for basic integer and memory operations and a floating-point unit (FPU). FPU and IALU instructions can be issued simultaneously each clock-cycle, as long as the instructions don't both use the same registers. Control instructions, such as branching and register data movements cannot be issued in parallel with IALU or FPU instructions. Instructions are executed in-order in a pipelined fashion, but can finish out of order.

Whenever an instruction executes, the target operand registers are locked until completion. If another instruction tries to access this register during this time, the pipeline will stall to ensure the use of valid data by all instructions. A simplified overview of the eCore hardware is given in Figure 7.

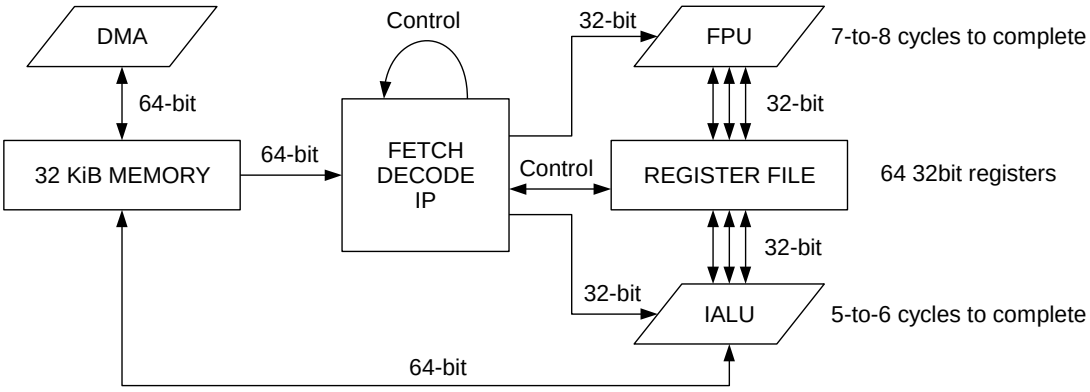


Figure 7: simplified eCore hardware overview, showing dual-issue capability

It is important to note that the IALU only supports data load/store, shift operations, addition, subtraction and bitwise operations. Integer multiplication and division, as well as floating-point division, are emulated in software. These emulated operations are retrieved from external memory when called, and introduce both a large memory and processing overhead. These operations should be avoided.

Each eCore is accompanied by a DMA engine featuring two independent channels, capable of transferring 64 bits every clock cycle. More detailed information on the DMA engines and the eCore instruction set can be found in [1].

3.2.4 Maximizing Epiphany Performance

The specified peak performance for the Parallella board is listed in Table 1. These peak performance numbers are only valid under specific conditions, which are discussed in this section. One important thing to note is that although technically the E16G301 supports clock rates up to 1GHz, it runs at only 600MHz on the Parallella board.

Specification	Value
Zync Frequency	667MHz
E16G301 Frequency	600MHz
E16G301 Peak Floating Point Performance	19.2 GFLOPS
Peak Bandwith Zync – E16G301	600 MB/s up, 600MB/s down
DDR3 Memory	1GiB
Power Consumption E16G301	0.69W@0.86V, 600MHz [14]

Table 1: Adapteva E16G301 / Parallella peak performance specification summary

Maximizing Memory Performance

The local memory of a node in the E16G301 is split into four banks of 8KiB. The local memory banks are 64 bits wide and can be accessed once every clock cycle for 8 to 64bit transfers. This means maximum performance for local memory is obtained when 64 bits are read or written every clock cycle [1]. For 600MHz this would equal 4.8GB/s for each bank simultaneously.

The DMA engine, instruction fetch stage and IALU can access separate memory banks simultaneously, so for optimal performance, different memory banks should be used for the program instructions and input and output data.

Accessing off-chip memory is typically very slow and should be avoided whenever possible.

Maximizing Network Performance

The network is optimized for on-chip writes over the cMesh. This allows an 8-to-64bit transaction every clock-cycle. Maximum performance is achieved for 64bit transactions every clock-cycle, leading up to 4.8GB/s per link at 600MHz.

For the E16G301 specifically, an errata item in the datasheet states that the DMA engine can only issue a transfer every other clock cycle, limiting the maximum performance to 2.4GB/s per link at 600MHz. A 1.5 clock-cycle latency is introduced for every router that is passed in the network, so routes should be kept short if latency is important. How the 1.5 cycle can occur is not elaborated.

Each eLink can send and receive 8 bits per clock-cycle simultaneously, and every eLink transfer requires a 40-bit header per data payload of 32-64 bits. For 64bit transfers with increasing address order only (0x0..0x4..0x8 etc..) the eLink can enter a burst mode, omitting the packet header until the sequence is broken. At 600MHz this results in 600MB/s per eLink in each direction simultaneously. For non-burst and read transfers, only 1/4th of this bandwidth can be achieved ([14]).

Maximizing eCore Performance

The maximum eCore performance is achieved by using fused multiply-add (FMADD) instructions. These instructions will first multiply two arguments and accumulate the result with a third. They can be issued every clock cycle, and are counted as two floating-point operations, resulting in the specified 1.2 GFLOPS/eCore at 600MHz.

To keep the processor busy at this rate, any operations stalling the pipeline, such as branches and memory or network stalls should be avoided. The IALU can be used to load the floating-point arguments from memory and store the results while the FPU is processing the data.

3.2.5 Initial E16G301 Performance Estimates

Now that the requirements of the application domain and the specifics of the Epiphany architecture have been introduced, an initial peak performance estimate of the beam-former and channel filters on the Epiphany architecture can be made. For this we start with the following assumptions:

1. The ADC input stream consists of 16-bit fixed-point samples.
2. The output type of any stage is a complex number. A single precision (2x 32bit) floating point representation will be used since that is most efficient on the Epiphany architecture.
3. Every component can be used at it's peak performance at 600MHz
4. A MAC can be performed by one fused-multiply add/subtract instruction every clock cycle

Based on these assumptions, the cost equations for the tasks (p.16) and the peak performance numbers of the Epiphany architecture, we can determine the peak throughput per task, as is shown in Table 2. These are the absolute maximum rates we can expect for the eLink and eCore at 600MHz, and they scale linearly with the system clock-rate.

Specification	Value
Peak eLink ADC sample throughput (16-bit)	300MS/s per eLink
Peak eLink complex sample throughput (64-bit)	75MS/s, per eLink
Peak Hilbert filter output samples/s per eCore	600 / (N_T * M_{HF}) = 9.38 MS/s
Peak bandpass filter output samples/s per eCore	600 / (N_T * M_{BPF}) = 4.69 MS/s
Peak beam-former input samples/s per eCore	600 / M_{CMAC}= 150 MS/s
Peak beam-former output samples/s per eCore	150 MS/s / #Channels

Table 2: Epiphany architecture peak throughput per front-end receiver task

The numbers presented in Table 2 are peak performance numbers that can only be maintained in very specific situations. The actual peak performance of the tasks and the network is likely to be different. In the next chapters the expected performance of the components for real-world applications, and how this is achieved for our beam-former application is discussed.

3.3 CRISP / RECORE XENTIUM RFD SUMMARY

The closest available platform in terms of architecture to which we can compare the Epiphany architecture is the Recore Xentium RFD. This is a many-core processor developed and used during the CRISP project ([2]) to determine the feasibility of a fault-tolerant and scalable front-end processing system ([15],[4]). An overview of the Recore Xentium RFD is given in Figure 8:

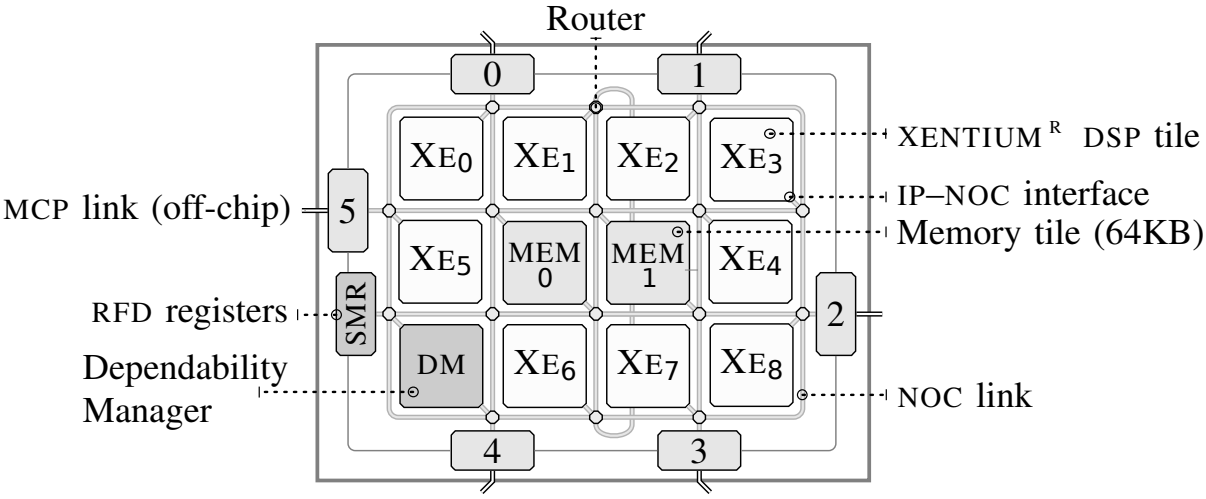


Figure 8: Recore Xentium RFD architecture overview (Taken from [15])

The Recore Xentium RFD features nine Xentium processor nodes [XE0..XE8], a dependability manager (DM) and two memory nodes (MEM). The processor nodes are based around a VLIW architecture, with a data-path that can execute four 16-bit MACs/cycle. Five Recore Xentium RFD processors are combined to form a single CRISP platform with a combined processing power of 180 16-bit MACs per clock cycle ([4]), or 36 GMACS at 200MHz.

Recore Xentium RFD Network-On-Chip

The Recore Xentium RFD on-chip network uses a 2D mesh topology, with one additional link as shown in Figure 8. It is a packet-switched network with wormhole type flow-control, and allows for 4 virtual channels ([16], chapter 1.4) per physical network link.

Instead of XY routing, adaptive source routing is used, where routes are not fixed, but determined at run-time in supporting hardware and software. Routers can be configured to assign different weights to each virtual channel in order to asymmetrically distribute the available bandwidth over the active connections.

Off-chip data communication is done through six independent off-chip connections that function as transparent links to and from the NoC. There is a difference in clock frequency between the on- and off-chip networks (50MHz off-chip vs. 200MHz on-chip) resulting in one quarter of the bandwidth per off-chip link compared to the on-chip links (800Mbit/s vs 3200Mbit/s at 200MHz).

3.3.1 Front-End Receiver Mapping and Performance

During the CRISP project, a front-end receiver with 16 input channels, using 32-tap filters with a Hilbert filter decimation factor of 8 and 8 output beams was implemented on the CRISP platform ([17]). The mapping used for this implementation is shown in Figure 9:

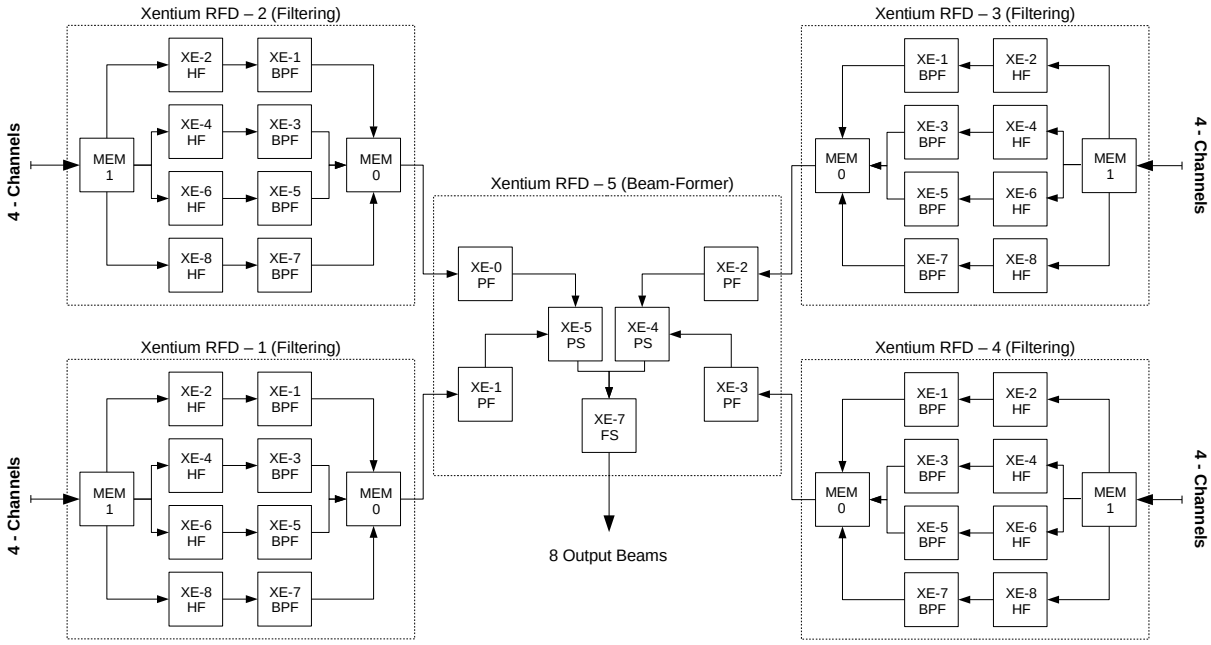


Figure 9: Front-end receiver mapping on CRISP processing platform featuring five Recore Xentium RFDs (N_C=16, D_{HF}=8, D_{BPF}=1, N_T=32, F_S = 22MS/s @ 200MHz)

In Figure 9, the filtering is performed by four out of the five Recore Xentium RFD processors. Each RFD processes four input channels. Beam-forming is done in several stages on the fifth Xentium RFD. First, the four output channels of a single filtering Xentium RFD are multiplied with the suitable coefficients in the partial beam-forming task (PF). The resulting channels are summed in the partial summing (PS) and final summing (FS) tasks.

All processing is done using a 16-bit data representation of each real valued sample, and two 16-bit values for complex valued samples. This allows the efficient use of the parallel data-paths of the Recore Xentium RFD, but also requires proper scaling of the coefficients and input samples to avoid overflow and loss of precision.

The final implementation can sustain an input sample rate of 25MS/s per channel. To achieve this, the filters and beam-former have been optimized and implemented in the assembly language. In order to determine the bottleneck and slack-time in this implementation, several performance measurements where performed, of which the results are repeated in Table 3.

Task to Process	Input Samples	Output Samples	Processing Time	Idle Time (waiting on data)
Filtering				
Hilbert Filter (HF)	1024	128	12 µs	29 µs
Bandpass Filter (BPF)	128	128	14 µs	26 µs
Beam-Forming				
Partial Beam-Forming (PF)	256	512	10 µs	12 µs
Partial Summing (PS)	512	512	2.5 µs	20 µs
Final Summing (FS)	512	512	2.5 µs	20 µs

Table 3: Recore Xentium RFD performance details for front-end receiver implementation

Table 3 shows that a lot of processor time is spent waiting on input data to arrive. Also, it is interesting to note that the bandpass filter takes only marginally longer to process than the Hilbert filter, even though at least twice as many calculations need to be performed for this filter task. This suggests that the implementation of the bandpass filter is more efficient than that of the Hilbert filter task.

Based on the initial peak-performance estimates for the E16G301 in Table 2 (p.22), the processing times of the filters for the same data lengths can be determined. These are given here in Table 4:

Task to Process	Input Samples	Output Samples	Recore Xentium Processing Time	Expected E16G301 Processing Time
Hilbert Filter	1024	128	12 µs	13.7 µs
Bandpass Filter	128	128	14 µs	27.3 µs

Table 4: E16G301 and Recore Xentium RFD initial performance comparison (estimated)

Table 4 shows that in the best-case, a single eCore at 600MHz cannot keep up with a single Xentium processor tile at 200MHz. This is mainly due to the large amount of parallel execution units on the Xentium processors, allowing up to four MACs per clock cycle. A single eCore can only perform a single MAC per clock cycle.

Unfortunately no information is available on the power consumption of the Recore Xentium RFD or CRISP platform. The performance estimates so far are based on peak-performance numbers provided by Adapteva. In the following chapters the achievable performance for a real-world implementation is investigated.

4 Benchmarking & Related Work

Thus-far, we have presented the peak performance numbers for the Epiphany architecture given in Table 1 and Table 2, which are derived from the Epiphany architecture specifications. However, it is unclear what percentage of this peak performance can realistically be achieved. In this chapter the expected peak performance for real-world applications is investigated by looking at related work. Based on the results, several missing benchmarks are identified. The missing benchmarks are implemented and the results are presented in chapter 5.

4.1 RELATED WORK

During this research, five other research papers discussing the performance of the Epiphany architecture have been published. In [18], a very detailed description of the development of a high performance matrix multiplication application on the Epiphany architecture is given. The authors of [19] and [20] compare the Epiphany architecture to a custom many-core processor and an Intel-I7 processor, and in [21] and [7], the use of a standardized Message-Passing-Interface (MPI) on the Epiphany architecture is investigated. The most relevant results in these research projects are repeated here for convenience.

4.1.1 eCore performance

The performance achieved for the application built in [18] are shown in Table 5. Here large matrix multiplication problems are split into smaller pieces over multiple eCores. The results include on-chip communication, but not the time required to get the operands from off-chip memory. A very strong dependence of performance on the problem size is observed. For small matrix sizes a large software and communication overhead limits the performance to only 26% of peak. In the best case 85% of peak performance is achieved.

Matrix Size (per-eCore)	Number of eCores used and total GFLOPS achieved					
	2x2		4x4		8x8	
	GFLOPS	%-Peak	GFLOPS	%-Peak	GFLOPS	%-Peak
8x8	1.25	26.1	5.07	26.4	20.30	26.4
16x16	3.12	65.1	12.76	66.5	51.41	66.9
20x20	3.58	74.7	14.36	74.8	57.62	75.0
24x24	3.84	80.1	15.43	80.4	62.17	81.0
32x32	4.06	84.7	16.27	84.7	65.32	85.1

Table 5: Epiphany G4 matrix-multiplication peak performance when using on-chip memory ([18])

When the total matrix size is increased such that the operands no long fit in local memory, additional communication over the eLink is required. Table 6 shows the results for a few of these cases. Here it become very clear that even through the local eCore problem sizes allow high computation efficiency, the bottleneck for these applications is the communication over the eLink interface.

Matrix Size (per eCore, total)	Total GFLOPS	% of Peak	%-Time spent on computation	%-Time spent on communication
32x32 per eCore 512x512 total	8.32	10.8	12.8	87.2
32x32 per eCore 1024x1024 total	8.52	11.1	13.1	86.9
24x24 per eCore 1536x1536 total	6.34	8.2	10.9	89.1

Table 6: Epiphany G4 matrix-multiplication peak performance when using shared memory ([18])

The results in Table 5 and Table 6 were obtained for a highly optimized assembly implementation on the Epiphany G4 processor. The authors note that implementation was time consuming, but achieving maximum performance with the GCC compiler proved difficult. However, in [7], the authors present four applications implemented in the C language for the E16G301 with the use of a special FMADD C function that reaches similar performance levels. The peak performance for each application in [7] is summarized in Table 7.

Application	GFLOPS	% of Peak	%-Time spent on computation	%-Time spent on communication
(32x32 per eCore, 128x128 total) Matrix-Multiplication	12.02	62.6	63	37
N-Body Particle Interaction	8.28	43.1	99	1
Heat-Stencil	6.25	32.6	45	55
2D-FFT Transform	2.5	13.0	60	40

Table 7: E16G301 peak performance and computation/communication ratio achieved in [7] for four different applications

The results Table 5, Table 6 and Table 7 include inter-node communication. The peak performance achieved during computation only for the 32x32 matrix multiplication in Table 5 is roughly 95% of peak according to the authors. For the matrix multiplication application in Table 7, achieving 12.02 GFLOPS when only 63% of the time spent is used for computation means that 99.4% of the 19.2 GFLOPS peak performance for all eCores was achieved during computation. These high percentages indicate that it is possible to come very close to the peak performance of the eCores, however, as Table 7 shows, this percentage of peak performance is very dependent on the application.

4.1.2 eMesh Performance

The E16G301 used by [7] and the G4 used by [18] both suffer from a problem with a FIFO buffer in the network interface of the mesh-nodes. According to the data-sheets this only affects the maximum outgoing bandwidth of a mesh-node. Unfortunately, it is unclear how much the performance is affected by this. Also a problem with the DMA engines on the E16G301 only allows a data transfer every other clock cycle, effectively halving the maximum achievable bandwidth.

Despite these errata, a peak DMA performance of 1963.6MB/s (82% of 2.4GB/s for 32bit transfers) is obtained in [18]. In [21] a peak transfer rate of 1262MB/s is achieved for 64bit direct-writes. Finally, in [7], a peak rate of 1000MB/s is achieved. The results of [21] and [18] are shown in Figure 10. Here a start-up behaviour is observed. This is likely caused by the DMA start-up time and direct-write software overhead. Interestingly, the direct transfer rates achieved in [21] are much higher than the results shown for direct writes in [18].

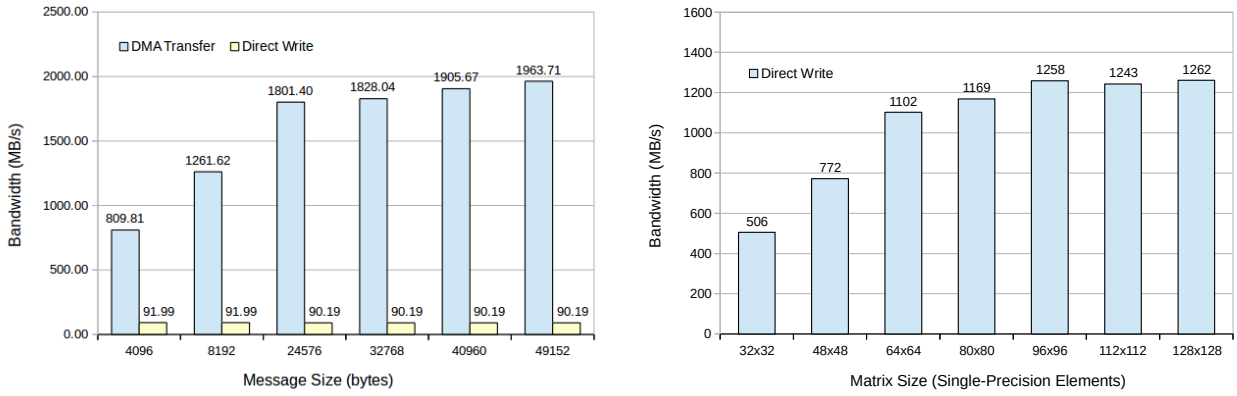


*Figure 10: **Left**: cMesh Bandwidth - DMA vs Direct Write (as presented in [18])*
***Right:** Direct-Write MPI bandwidth using 512-byte transfers (as presented in [21])*

The data sizes presented in both graphs in Figure 10 are somewhat ambiguous. In [18], it is unclear what the message size actually denotes, since 48KiB does not fit in local memory. It is assumed that this is the total data transferred by all 64 mesh-nodes, as their test routes packets through all mesh-nodes. In this case the 49152 byte transfer results would be for 768 byte transfers per node.

In [21], data is transferred in 512-byte blocks, and only total data sent changes by sending multiple packets sequentially. It is noted that the measurement was done for the transfer of all sub-matrices per node. For a 32x32 matrix multiplication, both matrix operands would be divided into sixteen 8x8 sub-matrices, resulting in two 8x8 matrices per node. Each elements takes up 4 bytes, resulting in a total transferred size of 512 bytes per node for this case.

Both representations in Figure 10 are relatively hard to translate to a new application, where the interest would mainly be in the time it takes to transfer a certain amount of data from node to node. A clearer representation is given in [7], however much lower bandwidths are achieved.

Neither [18] or [21] achieve peak-performance, nor use 64bit DMA transfers required to meet this peak performance. The effect of sharing links on this performance, or how the DMA performance is affected when both channels are used simultaneously also has not yet been investigated.

4.1.3 eLink Performance

Table 8 shows the achieved bandwidth per node in [18], when four nodes send 2KiB packets over the same eLink simultaneously during two seconds. The total bandwidth achieved is 150MB/s, one quarter of the maximum achievable bandwidth of 600MB/s. This is expected, as burst transfers for 32bit writes are not possible. The results show that the bandwidth is not equally divided, in fact, the authors claim that for the G4 starvation can occur when even more nodes share an eLink.

Source Node	Sent Packets	eLink Share (%)	Bandwidth (MB/s)
[0,0]	61307	41.8	62.8
[0,1]	48829	33.3	50.0
[1,0]	24414	16.6	25.0
[1,1]	12207	8.3	12.5

Table 8: eLink bandwidth per node when four nodes send 2KiB packets during a two second period (as presented in [18])

The total bandwidth achieved in Table 8 is 150MB/s for 32bit transfers, suggesting that sharing an eLink does not influence the eLink performance. However, when multiple nodes attempt to perform 64bit burst transfers over the same eLink, it is likely that the eLink will alternate communication slots between nodes. This would effectively break the burst-mode capability of the eLink, limiting the peak bandwidth to 150MB/s instead of 600MB/s. This would have a large impact on application design, as only one node should be allowed to communicate over an eLink at any given time.

4.1.4 Epiphany Power Consumption

Currently there is very little information available on the actual power consumption of the Epiphany architecture. The only power consumption numbers available are those from the E16G301 datasheet [14]. Figure 11 shows a more convenient representation of the values in [14]. Also the peak efficiency is shown for these results, assuming peak performance was achieved for the values in [14].

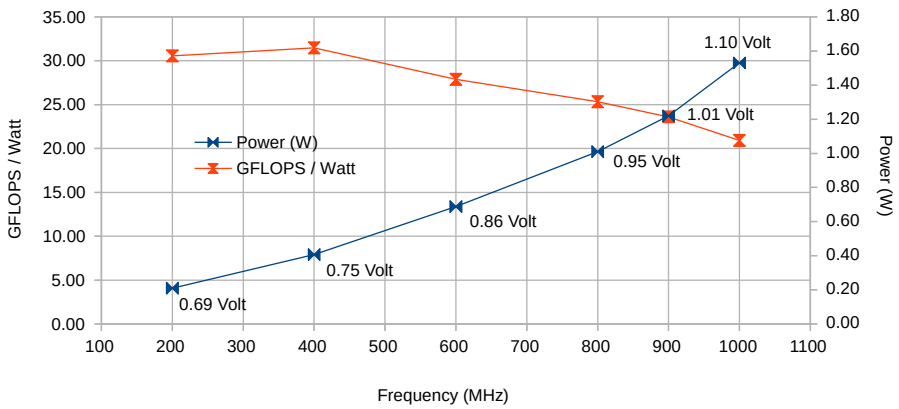


Figure 11: E16G301 Power consumption and GFLOPS / Watt versus frequency at minimum core voltage. Results are derived from [14]

Figure 11 shows that 32 GFLOPS/watt can only be achieved at 400MHz. At 1GHz, this reduces to roughly 21GFLOPS/watt. This result differs from the performance of 32GFLOPS/watt at 1GHz shown in the STARS multi-core survey ([9], Table 12 p.73). The details of the power measurements in [14] (Achieved GFLOPS, network load, part contributed by the separate components) are not given. This means no information is available on the actual achieved performance per watt for a real-world test-case. Also, the effect of core voltage on the power consumption is not known at this point.

4.1.5 Remaining Work

So far we have seen that the achievable performance on the eCore depends heavily on the application. In order to determine a realistic measure of peak performance of our application, the front-end receiver tasks should be implemented and analysed.

For both the eMesh and eLink, the related work has shown some interesting behaviour and performance numbers, however, it is unclear what causes the observed DMA start-up behaviour, and also the performance for 64bit DMA transfers over the eMesh and eLink is still unknown. To investigate this and other topics further, several custom micro benchmarks will be developed. The design of these benchmarks is discussed in the next section, which will cover the following topics:

1. **Message Size vs. Bandwidth:** Here the effect of total transfer size on the achieved bandwidth on both the eMesh and eLink, using DMA and direct-write transfers is measured. The goal is to find the cause of the start-up behaviours observed in Figure 10, and to determine the best method to use for each message size to use for our application.
2. **eLink Bandwidth Sharing:** The peak off-chip bandwidth is only maintainable using burst-mode transfer. We expect sharing the eLink will break the burst mode transfer, however this is not documented or tested in related work.
3. **eMesh Bandwidth Sharing:** The links in the cMesh should support 4.8GB/s, however the results so far show that this bandwidth is not easily achieved. It is interesting to investigate how the eMesh handles sharing of network bandwidth, as this determines for a large part the freedom of placement of tasks on the Epiphany platform.
4. **E16G301 Power Consumption:** In order to determine the real-world efficiency of the E16G301, eCore, eLink and eMesh power consumption will be measured along with the achieved GFLOPS and bandwidths. Doing this at different core voltages will give insight in the dependence of the core voltage on the architecture efficiency.

4.2 CUSTOM MICRO-BENCHMARKS

In this section, several custom micro-benchmarks are introduced to give some insight in the open topics presented previously. In addition, a simple benchmark program provided by Adapteva ([22]) is run to test the Parallella board network bandwidths. The results of the benchmarks presented here are given in chapter 5.

4.2.1 Message Size vs. Bandwidth

The effect of the total transfer size on the effective bandwidth of the DMA engine is tested using one sending mesh-node. This node will send increasing packet sizes up to 8KiB using 64bit DMA transfers. The time it takes to configure the DMA engine is measured, as well as the total time to perform the transfer. This is done for both configurations shown in Figure 12.

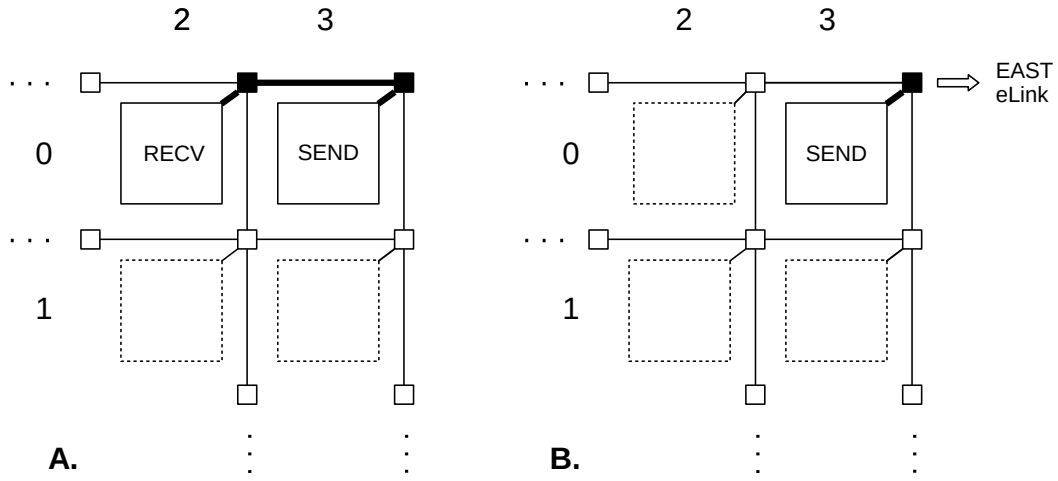


Figure 12: DMA transfer size micro-benchmark test-cases. ***Left:*** *Data is sent over eMesh*
Right: *Data is sent over eLink to shared memory*

4.2.2 eLink Bandwidth Sharing

Measuring the effect of sharing an eLink during maximum performance operation will be addressed by introducing the three different configurations shown in Figure 13. All active nodes will send a continuous stream of 8KiB packets using 64bit DMA transfers. The time required to send each packet is measured.

First, the four nodes closest to the eLink send data simultaneously. Here it is expected that the bandwidth will be shared equally amongst all nodes, but drops significantly. Second, four nodes with varying distance to the eLink, will send data. This should introduce similar starvation as shown in Table 8, but shown for 64bit transfers. Finally all sixteen nodes will send data simultaneously. This allows comparing the bandwidth achieved with one, four and sixteen nodes to see if significant routing overhead is introduced when sharing an eLink.

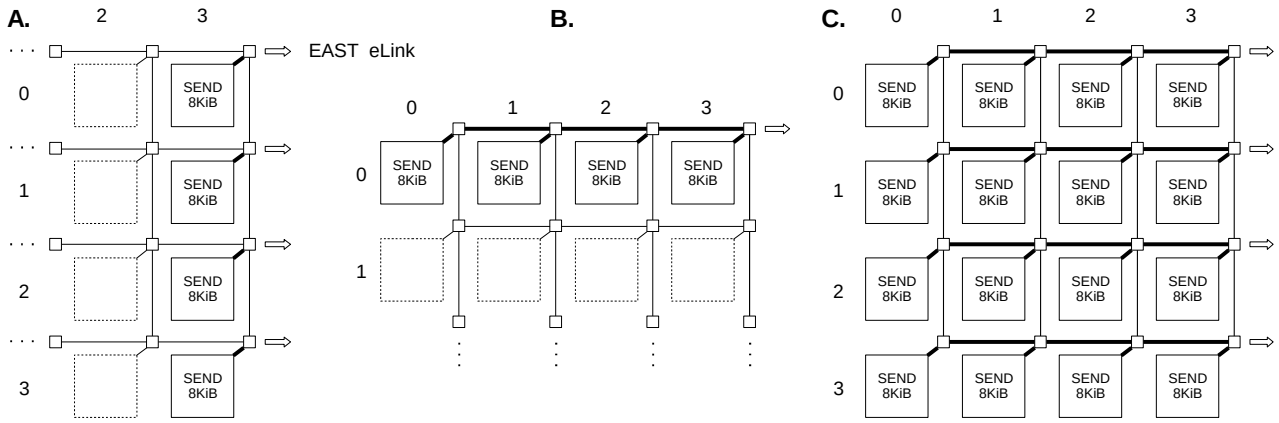


*Figure 13: eLink micro-benchmark test-cases. **A:** Four nodes closest to the eLink send packets. **B:** Four nodes with varying distance to the eLink send packets. **C:** All nodes send packets.*

4.2.3 eMesh Bandwidth Sharing

Here we introduce two cases in Figure 14. First, four nodes in a cross will send a continuous stream of 8KiB packets towards a single node. The achieved bandwidth per node is measured. Here we expect that the total bandwidth is limited by the receiver memory, and that the remaining bandwidth is shared equally among the senders.

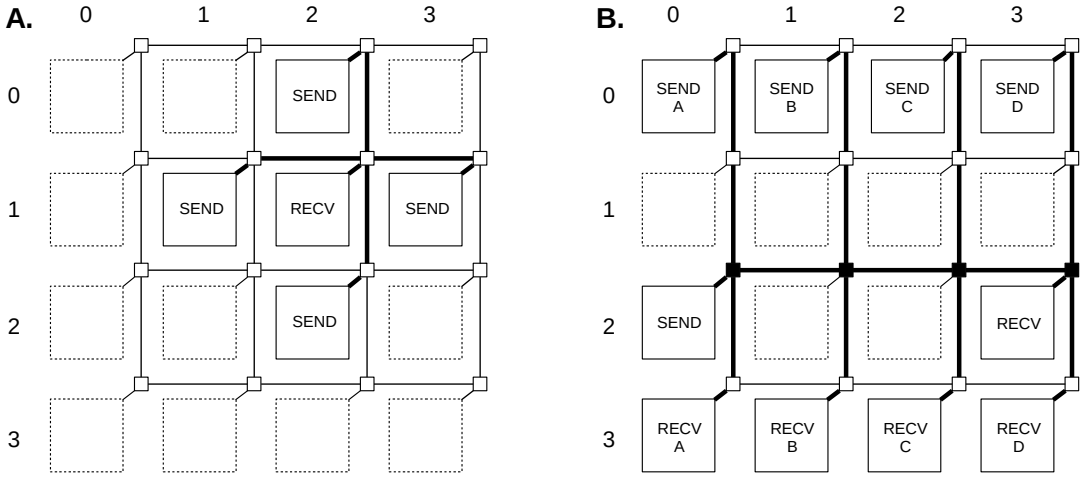


*Figure 14: eMesh sharing configurations. **Left:** four nodes in a cross send to a receiver. **Right:** Four sets of nodes send from North to South, effect on performance from East to West is measured*

In the second case in Figure 14, crossing traffic is introduced. One node continuously sends data to a receiving node from East to West on the cMesh. This traffic is then crossed by one to four other transfers from North to South. The effect on the achieved bandwidth from East to West is measured.

4.2.4 E16G301 Power Consumption

The power supply of the E16G301 is split into a core power and IO power section. Both need to be measured to determine the total power. Unfortunately the IO power rail is shared with many other devices on the Parallella board, and cannot be easily decoupled. For useful measurements a method is required that allows separation of the E16G301 power and that of the other devices. Also, the measurements need to be properly synchronized with the software to be sure that the power measurements don't include possible idle times of the E16G301.

The E16G301 runs at 1 Volt on the Parallella. The datasheet shows that for 600MHz, the E16G301 can go down to as low as 0.86 Volt. This has a very large impact on the power consumption. For a fair minimum power consumption measurement, ideally we would like to vary the E16G301 core voltage to see the effect of this on the power consumption.

The issues with the power measurements are addressed by building a small measurement circuit that reports measured power over an on-board I2C bus. The I2C bus can also be used to configure the core voltage regulator for the E16G301, allowing us to set the E16G301 core voltage from software. The details of this measurement circuit are given in Appendix C: Power measurement setup.

Several test cases will be measured, each at multiple E16G301 core voltages:

- **eCores:** eCore power consumption will be measured by measuring core-power consumption during execution of a heavy load micro-benchmark. No data is transferred during this benchmark keeping the eLinks and eMesh networks idle during this measurement.
- **eMesh:** The eMesh power consumption will be measured increasing the distance of a transfer between nodes, without changing the number of active nodes or the amount of data sent. The increase in power consumption over distance is contributed by the increased network activity.
- **eLink:** the eLinks are powered by the IO power-rail shared with multiple devices. To measure the eLink power consumption, we let a single node repeatedly send and stop sending data over the East eLink. The measured variation in the power consumption on the IO power-rail should reflect the active power consumption of the eLink. Also, the eLinks can be switched off by hardware, by repeatedly switching the North/South/West eLink on and off simultaneously, we can determine the standby power of the eLinks.

Based on these results, estimates can be made of power consumption under different load cases. The power consumption during execution of the front-end receiver chain on the Epiphany architecture is also measured.

5 Micro-Benchmark Results

In this chapter we present the results for the micro-benchmarks on the E16G301. We start by looking at a small benchmark application provided by Adapteva [22]. This benchmark copies 8KiB from multiple sources to several destinations, and measures the time it takes for this transfer to complete. The results for the E16G301 and the Parallella board are shown in Table 9.

From	To	Type	Time (us) / 8Kb	Bandwidth (MB/s)	% of Peak
ARM	eCore	Read	1255.3	6.5	4.4
ARM	eCore	Write	586.9	14.0	2.3
ARM	DRAM	Copy	42.3	193.8	?
eCore	DRAM	Read	89.2	91.8	61.2
eCore	DRAM	Write	32.4	252.5	42.1
eCore	eCore	Read	20.0	409.0	68.2
eCore	eCore	Write	6.1	1342.2	28.0

Table 9: Parallella communication performance measured by Adapteva benchmark application (average of 50 measurements)

On the parallella board, there are two ways to get data onto the E16G301 from shared memory. This is either through a write transaction from the host processors, or through a read transaction from the mesh-nodes. Table 9 shows that the fastest method to get data onto the E16G301 on the Parallella board is through a read transaction initiated by a mesh-node, which achieves 91.8 MB/s. This is much lower than the maximum achievable bandwidth of 600MB/s at 600MHz, and also lower than the limited 150MB/s that should be possible for eLink read transactions.

It is not entirely clear what causes this, but a read transaction has to travel over the eLink twice, first as a read transaction of 9 bytes, and then as a write transfer of 9 to 13 bytes (depending on the transfer size) from the host returning the data. The time it takes to process the read-request on the host-side will influence the achieved bandwidth.

The direct transfers from the ARM processors to the E16G301 are very slow (14.0 MB/s), possibly due to the use of the memcpy() function by the provided host library which can cause a lot of overhead on the eLink transfers, or due to the overhead introduced of reading from memory first, and then writing the result over the same AXI bus to the FPGA eLink hardware. These low host-to-mesh-node transfer speeds are only applicable for the Parallella board and not for the Epiphany architecture in general, demonstrated by the much higher rates when the eCores transfer data over the same eLink.

The inter-node read and write rates of 409MB/s and 1342MB/s are significantly lower than the specified peak performance, but similar to the results presented in [18], [21] and [7]. This is likely caused by a combination of the errata items mentioned in the previous chapter and the DMA start-up time.

5.1.1 Message Size versus Bandwidth

The results for the DMA and direct-write transfer size micro-benchmark measurements are shown in Figure 15. First the DMA transfer function provided by the SDK library was tested over the eMesh and eLink for different transfer sizes, capturing both the total transfer time and start-up time. Here we see a very large portion of the time spent sending data can be attributed to starting up the DMA engine (65.2% to 6.9%).

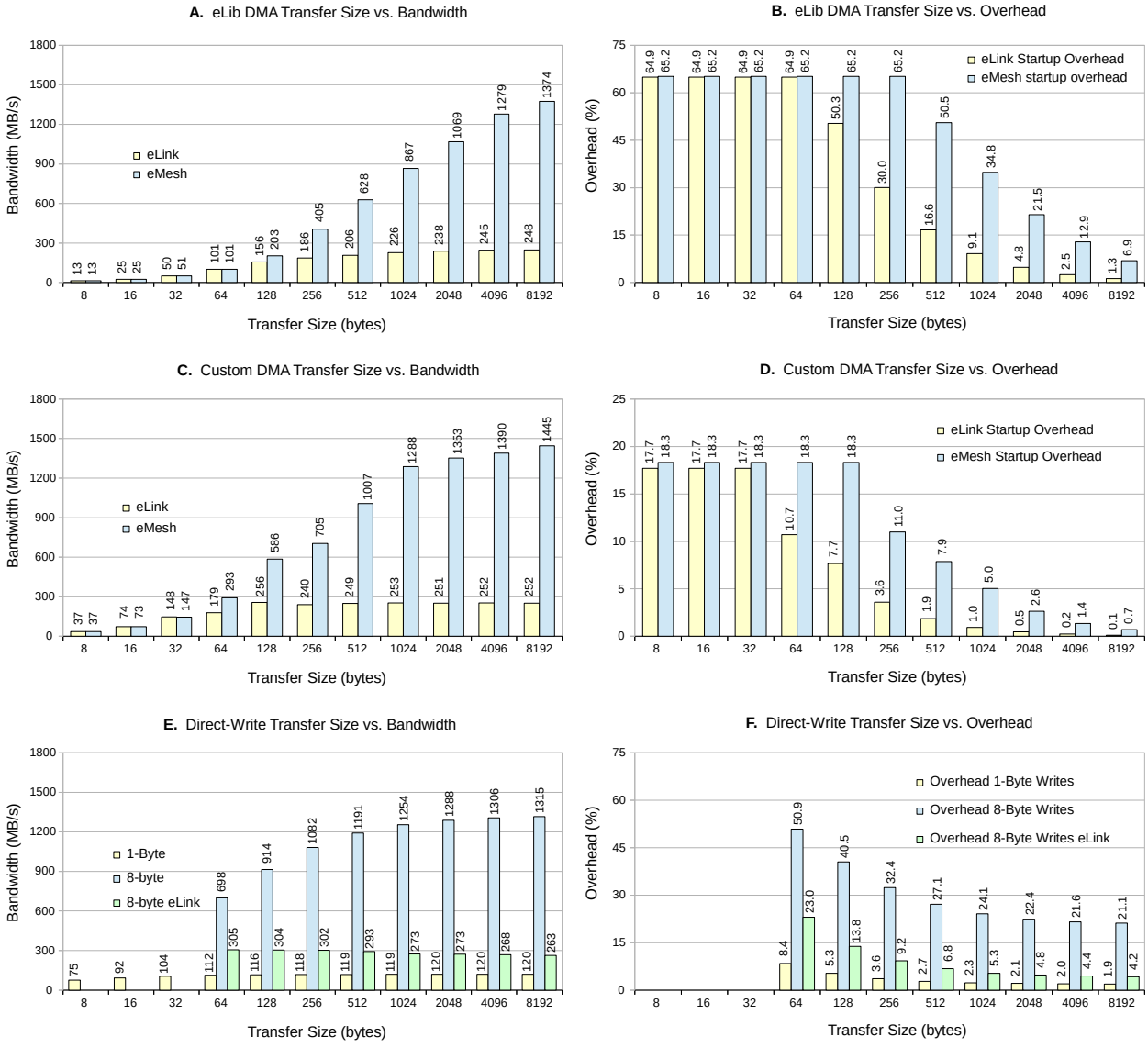


*Figure 15: Data transfer size versus bandwidth and overhead for **A/B:** Library provided DMA transfer. **C/D:** Custom built DMA transfer. **E/F:** Custom direct-write transfer*

This large overhead is mainly due to many nested function calls and switch statements in the provided library functions to keep them generic. One such call is the calculation of the core global address several times, to determine the global addresses of registers to write to. For a custom implementation, where the DMA channel and core address are set only once on start-up, we managed to realize a start-up overhead of only 0.7% for 8KiB transfers at 1445MB/s. These results are shown in Figure 15 C. and Figure 15 D.

For Figure 15 E. and F., the DMA calls are replaced with a direct-write implementation which writes 64bit values in a loop, unrolled 8 times. The software overhead for these direct-writes is higher than the start-up overhead measured for the DMA engine, however, similar bandwidths are achieved. This suggests the actual achieved transfer rate is higher than what is achieved by using the DMA engine. Also, in practice, the software overhead needed to send the data can be masked by transferring results directly during calculation as is done in [21], instead of saving them to local memory first.

None of the eMesh transfer implementations achieve the expected peak transfer rates. There are multiple possible causes for this, such as the errata items for the E16G301, memory access during transfer, the 1.5 cycle routing latency per router and alternating load and store instructions in the direct-write case. However, since all approaches, including those of related work, come close to the figures presented in Figure 15, it is assumed that this is a realistic representation for the achievable bandwidth in real-world applications, and this is not investigate further.

For the eLink transfers, a peak rate of 600MB/s is expected, but only roughly 306MB/s is achieved. This is likely related to the Parallella board hardware, and not necessarily a limit of the E16G301. A direct copy of data in the DRAM initiated by the ARM processors achieves only 194MB/s (shown in Table 9 p.35)). This copy only involves the DRAM, showing that memory bandwidth might be a limiting factor. Also the Zync host processor and the E16G301 run at different clock speeds which might influence the achievable bandwidth over the eLink.

5.1.2 eLink Bandwidth Sharing:

The 306MB/s over the eLink is achieved for 64bit sequential transfers. Theoretically this peak off-chip bandwidth is only maintainable using burst-mode transfer, since in all other cases only 150MB/s should be achievable in the best case. The effect of sharing the eLink using 64bit sequential transfers is shown in Figure 16, where nodes simultaneously send an 8KiB packet over the same eLink.

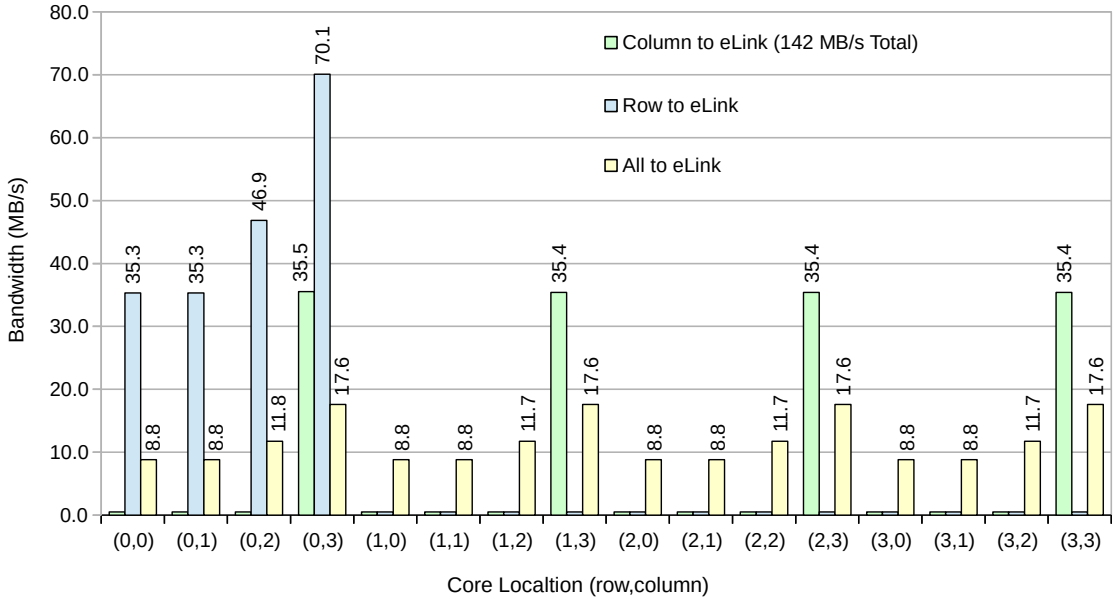


Figure 16: eLink bandwidth sharing result for the row-to-eLink (blue), column-to-eLink (green) and all-to-eLink (yellow) configurations shown in Figure 13 (p.33)

When the four nodes in the column closest to the eLink (nodes [0-3,3]) issue 64bit transfers simultaneously, the bandwidth is very evenly divided, however, as expected, the total bandwidth achieved is only 142MB/s, much less than 306MB/s. When we let four nodes in a row (nodes [0,0-3]) send 8KiB simultaneously, the bandwidth is no longer equally divided. This is a consequence of round-robin arbitration in the routers, as shown in Figure 17:

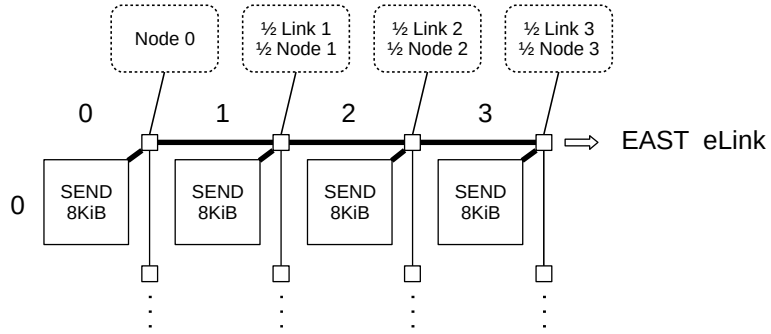


Figure 17: Round-robin bandwidth sharing in routers where bandwidth up-stream is limited by the East eLink

Although it seems the total bandwidth exceeds 150MB/s (70.1+46.9+35.3+35.3 = 187.6MB/s), we cannot simply add these figures, as the nodes finish at a different point in time. Figure 18 shows the expected division of bandwidth over time in this situation, and shows that the bandwidth never exceeds a certain maximum bandwidth, which is only twice the bandwidth achieved by node (0,3).

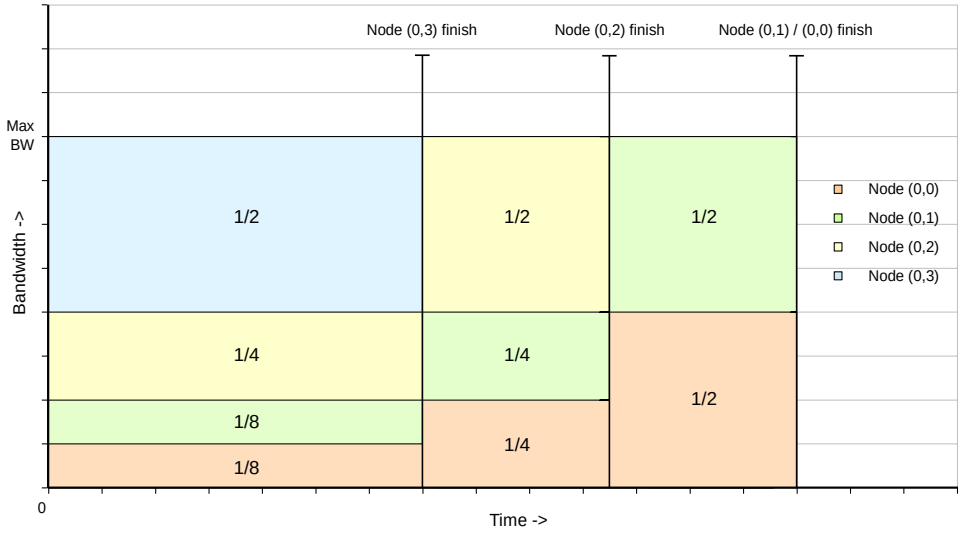


Figure 18: Expected eLink bandwidth share per node over time for four nodes in a row as in Figure 13

This behaviour is captured in equations (6) and (7), where $T_{[r,c]}$ denotes the finish time for node (row, column), N is the number of bytes sent and $BW_{[r,c]}$ is the average bandwidth of a node. Note that these equations only apply if all nodes use the maximum available bandwidth, as in other situations the bandwidth share per router might not be equal for both nodes. Since the eLink bandwidth is much lower than that of the cMesh and nodes, this is the case here.

$$T_{[r,c]} = T_{[r,c+1]} + \frac{\frac{1}{2}N}{\frac{1}{2}BW_{max}} \;,\; T_{[r,4]} = \frac{N}{BW_{max}} \;,\; T_{[r,0]} = T_{[r,1]} \tag{6}$$

$$BW_{[r,c]} = \frac{N}{T_{[r,c]}} = \left(\frac{BW_{[r,c+1]} \cdot BW_{max}}{BW_{[r,c+1]} + BW_{max}} \right) \text{ with } BW_{[r,4]} = BW_{max} \tag{7}$$

When we apply equation (6) and (7) on the situation in Figure 16 (BW_{max}=2*70.1=140.2 MB/s), we get a bandwidth of 70.1MB/s for node (0,3), a bandwidth of 46.73MB/s for node (0,2) and a bandwidth of 35.05MB/s for nodes (0,1) and (0,0) respectively. This is very close to what we observe in Figure 16.

Figure 18 shows that the combined bandwidth never goes over BW_{max} = 140.2MB/s, much lower than the peak achieved bandwidth of 306MB/s for a single node, and also lower than the 150MB/s that should be possible in this case. This suggests that the burst-mode transfers are indeed broken, and that switching overhead is introduced in the eLink arbitration.

The model in equations (6), (7) suggest that complete starvation cannot occur (a node will always receive some portion of the maximum bandwidth). To verify this, in a last test-case, the east eLink is shared by all nodes on the E16G301. The results are shown in Figure 16, where the same behaviour as when sharing the link with four nodes is observed, except the bandwidth is equally divided across the rows. This means that the starvation case shown in [18] for the G4 could not be reproduced, although the available eLink bandwidth will quickly deteriorate for nodes far away from the eLink.

5.1.3 eMesh Bandwidth Sharing

To test the effect of sharing the eMesh bandwidth, we let the nodes around node (1,2) transfer data to node (1,2), so that they share the router of this node. Figure 19 we see that as long as only three nodes share the router (result 1-4), they all reach their peak bandwidth and get a fair share of the router as expected. When all four surrounding nodes transfer data simultaneously (result 5), the bandwidth is no longer equally divided. This is due to the fact that all bandwidths combined exceed the maximum data-rates of the destination node's memory.

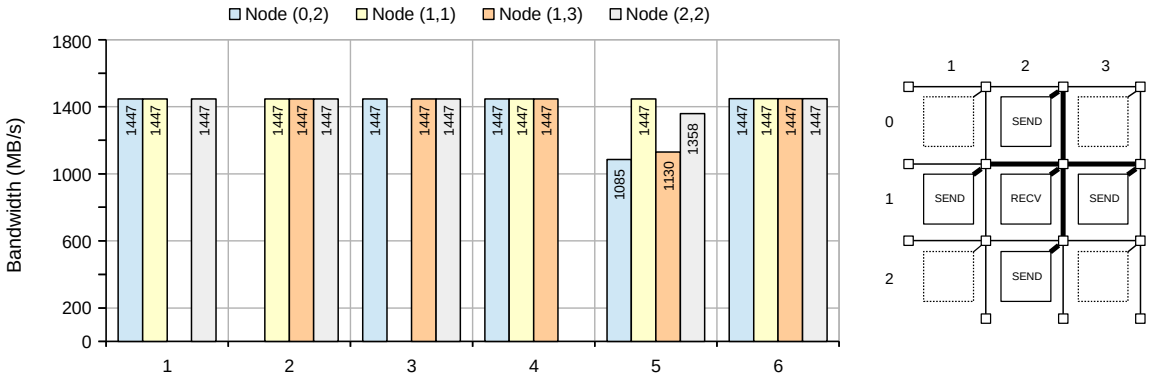


*Figure 19: Bandwidth for cMesh sharing test cases. **1:** Excluding right node. **2:** Excluding top node. **3:** Excluding left node. **4:** Excluding bottom node. **5:** All nodes, single target. **6:** Multiple targets*

If we change the destination of the top and right nodes such that they still share the same router, but no longer write to the memory in node (1,2) (result 6 in Figure 19), we see that all the nodes reach their peak bandwidth again. This suggests that the limitation is not in the router, and that this can maintain fair round-robin arbitration at maximum bandwidth as long as the outgoing links are not saturated.

For our second test case (Figure 14, p.33), we let a single node send data across the network, and introduce multiple crossing streams. No effect of the crossing streams on the achieved bandwidth is expected since these streams do not write to the same node. This is confirmed in Figure 20.

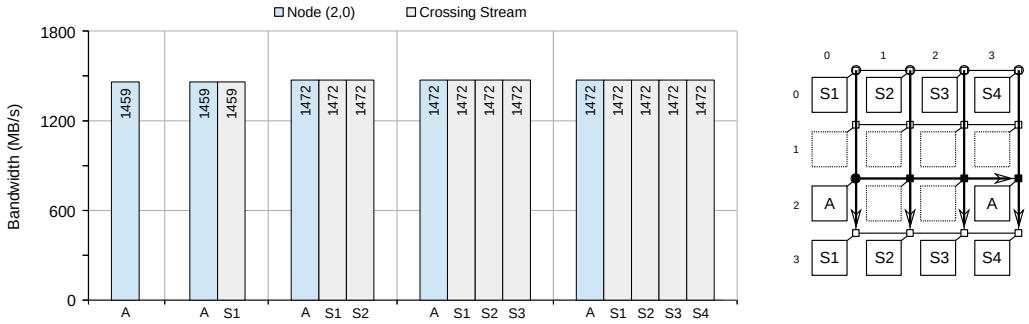


Figure 20: Bandwidth for node (2,0) to node (2,3) versus number of crossing streams

The results in Figure 20 show that in practice, placement of nodes on the E16G301 will not have much influence on the achievable bandwidth between nodes, even in case of crossing traffic, as long as the total bandwidth over a link does not exceed the maximum bandwidth. Longer and shared routes will have increased latency, however [18] shows that this latency is minimal compared to the time it takes to actually transfer all data.

5.1.4 E16G301 Power Consumption

The results of the power measurements will be discussed in three parts, the eCore, eMesh and eLink power measurements. All measurements are performed for a range of twelve different core voltage settings, where the lower limit of 0,825 volt is a limitation of the regulator, and the upper limit of 1.075 is chosen to avoid damage to the processor. The IO voltage is not changed.

eCore power consumption

To measure the eCore power consumption, the eCores are subjected to five different load cases at all core voltages whilst measuring the IO and Core power consumption. The following load-cases were used:

1. **Memory:** 64Bit memory access on every instruction (mixed data loads and stores)
2. **FPU:** Only FMADD instructions
3. **IALU:** Only integer ADD instructions
4. **FPU+IALU:** Alternating integer add and floating-point add instructions
5. **FPU+Memory:** Alternating memory load and FMADD instructions

The load cases are written in the assembly language to assure maximum performance, and care has been taken to avoid pipeline stalls due to registry dependencies or memory loads. Each load case consists of a loop repeating 512 hand-written assembly instructions indefinitely. The implementation of the load-cases can be found in appendix B (p.74).

NOTE: All tests were performed after pre-loading dummy-data into the memory and registers. This was found to have a very large impact (roughly 30%) on the power consumption of the load-cases involving integer or floating-point computations.

In addition to the presented load-cases, the power consumption of the assembly implementation of the matrix multiplication used in [18] was measured. This was obtained from the Adapteva open Github repositories [23]. The core power and IO power for all load-cases are shown in Figure 21, along with the single data point provided in the E16G301 datasheet for 600MHz at 0,86V.

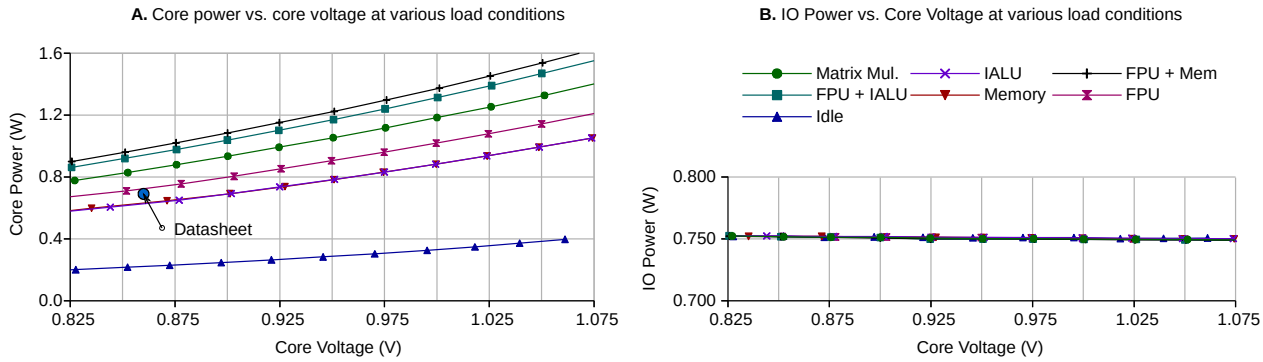


*Figure 21: E16G301 eCore power measurements (f_C=600MHz) **A:** Core voltage vs. chip core power. **B:** Core voltage vs. Parallella board IO power*

The datasheet measurement point in Figure 21A shows a power consumption of 0.69Watt for an unknown load-case at 600MHz with a core voltage of 0.86 Volt. If we assume the load-case reaches 100% of peak performance, this suggests a power efficiency of 27.88 GFLOPS/Watt. This can be compared to the achieved power efficiencies for our load-cases.

Figure 22 shows the achieved performance per load-case for both the FPU and IALU execution path and the achieved processor efficiency (in GFLOPS/Watt) at different core voltages.

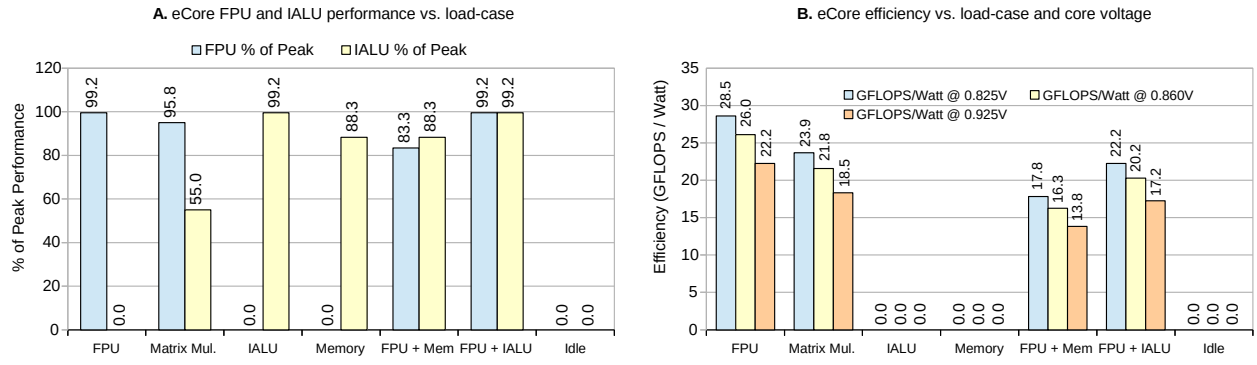


Figure 22: eCore performance and efficiency per load-case at several core voltages

In Figure 22B we see that none of the load-cases achieve a power efficiency of 27.88 GFLOPS/Watt at 0.86 Volt. The FPU load-case comes closest with 26.1 GFLOPS/Watt. This suggests that either the load-case used for the measurements in the datasheet did not reach 100% of peak performance, or that only part of the hardware was active during the measurement. For real-word applications, the use of the IALU to load data from memory and store the results is inevitable.

The matrix multiplication and the FPU+Memory load-case show the effect of using the FPU, IALU and memory in more realistic usage scenario's. Power efficiencies of 21.8 GFLOPS/Watt and 16.3 GFLOPS/Watt are achieved at a core voltage of 0.86V for the FPU and IALU respectively. The difference in power efficiency is due to the fact that for the matrix multiplication load-case, the IALU, and consequently the memory, is used less than for the FPU+Memory load-case. In practice the efficiency of an application can thus be improved by reducing the amount of required memory accesses. Also, the core voltage could be reduced, however, the datasheet notes that highest achievable frequency at 0,86 volt is 600MHz, suggesting that lowering this voltage will also result in a lower maximum operating frequency. In our measurements no faults were discovered running the E16G301 at 600MHz at 0,825 volt, but this could differ per processor as we only tested one sample.

A last important observation is that leaving eCores idle will still result in significant leakage power consumption (0.22Watt at 0.86 volt for the entire processor). The means that if not all eCores in a system are used, the total efficiency will drop very quickly.

eMesh power consumption

The eMesh power consumption can be measured by looking at the difference in power consumption for two identical data-transfers that take a different route. First, we let all nodes in column 2, send data to their neighbours in column 3. Next, the nodes in column 2 stop sending data, and the nodes in column 0 start sending data to column 3. The difference in power consumption is then attributed to the use of 8 additional links and routers. This is shown in Figure 23.

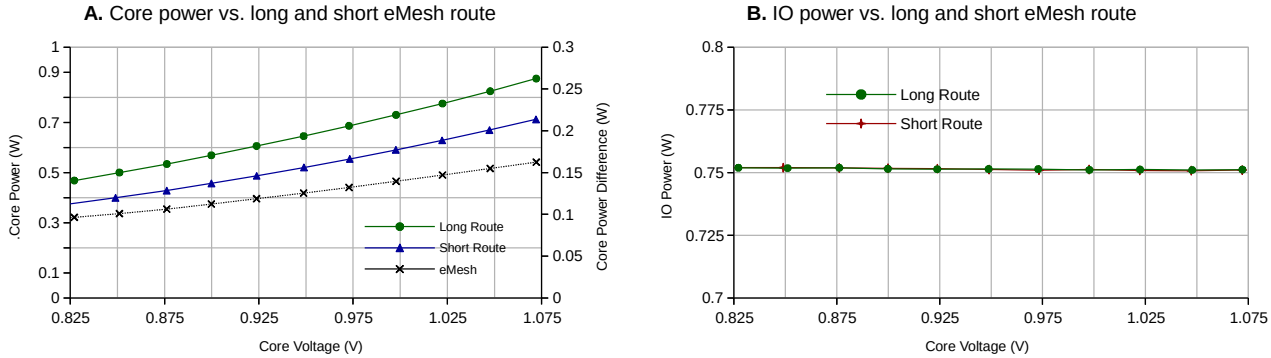


Figure 23: E16G301 eMesh power consumption for long and short data routes (f_{CLK}=600MHz)
***A:** Total chip core power vs. core voltage. **B:** Total Parallella board IO power vs. core voltage*

The total power consumption difference at 0.86V is 104mW, or 12.9mW per Router/Link pair. Since no data is available for comparison, it is hard to verify whether this is an expected result, but it seems reasonable within the results we have seen so far. In any case, it is worth optimizing for reduced communication bandwidth if power efficiency is important.

eLink power consumption

The power of the eLink is sourced by the IO power rail shared with multiple other devices. This makes it somewhat harder to measure. Here, we alternate between two cases, one sending data over the eLink (Active case in Figure 24b), and one doing nothing (Idle case in Figure 24b). The difference in power consumption for these two case is caused by the eLink (eLink plot in Figure 24b).

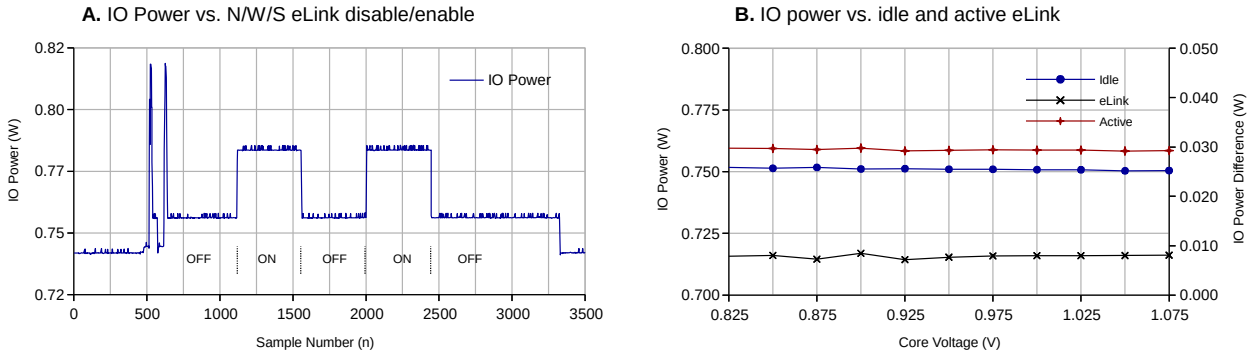


Figure 24: E16G301 eLink power consumption. ***A:*** *Standby power measurement for North, South and West eLink.* ***B:*** *IO power consumption without and without eLink data transfer one-way (f_{CLK}=600MHz)*

Based on Figure 24b, and an achieved bandwidth of 150MB/s, the communication costs over the eLink are roughly 10mW for 150MB/s data transfer one-way. Significant bi-directional data-transfer was not possible since the achievable ARM-to-Epiphany transfer rates are very low.

In another measurement we disable and enable the NORTH, SOUTH and WEST eLink hardware and measure the effect on the IO power consumption. The results are shown in Figure 24a. Here a 27mW difference in power consumption is observed for the combined North, South and West eLink (3-eLinks) standby power. This suggests a total combined standby power of 36.3mW for all four eLinks.

Note that there is no way for us to measure if there is any remaining standby power being consumed since we cannot separate the E16G301 IO power from the other devices on this rail. However, judging by the idle power observed for the eCores (Idle case in Figure 21a), it is likely that some leakage current is present for the eLink hardware.

E16G301 combined power consumption and efficiency

In practice, not all hardware will be used simultaneously. This means the measured power figures cannot simply be added together to get to a total system power consumption. However, if the load percentages of the IALU and FPU, and the network behaviour of an application are known, the measured power consumption figures can be used to get a good estimate of the total power consumption of an application.

5.1.5 Micro-Benchmark result summary

So far we have seen that the data-transfer rates achieved in practice are significantly lower than the specified peak performance, partly caused by errata items of the hardware that was used, and for some part caused by a large start-up overhead introduced by the provided libraries. The latter can be improved by using a custom data-transfer implementation. Also, DMA descriptors could be reused to reduce the required start-up time even further.

For external communication over the eLinks, performance drops significantly when the eLinks are shared. This is likely due to breaking of the burst-mode transfers. This means it is important that either one node gets exclusive access to an eLink, or that some form of scheduling / synchronization is applied to share the eLink efficiently if maximum bandwidth is desired.

Finally, a detailed breakdown of the eCore, eMesh and eLink power consumption has been presented, which shows that the power consumption figures presented in the datasheet likely do not include all hardware components. A peak efficiency of 21.8GFLOPS/Watt was achieved at 600MHz for the matrix multiplication application presented in [18], with a core voltage of 0.86 Volt. Slightly higher efficiencies are achieved at lower core voltages, however it is uncertain if these voltages can be sustained error free on all E16G301 processors. The achieved power efficiencies do not include any on or off-chip network traffic.

6 Front-end Task Implementation

Here we present the results for the Hilbert filter, bandpass filter and beam-former tasks individually. First a breakdown of how the Hilbert filter was optimized is given, and it is shown to what extend this can be done automatically. This is done for a fixed input sample set (video) size of 512 samples, at multiple decimation factors. The same is done for the bandpass filter and beam-former. This chapter concludes with power measurement results for the best performing implementations of the tasks.

6.1.1 Hilbert Filter Optimization

For the Hilbert filter we start with the convolution implementation shown in Snippet 1. Here we iterate through the real valued input samples using the appropriate decimation factor in an outer loop, and perform the convolution per output sample in an inner loop. It is assumed that the coefficients are stored in reverse order for convenient loop iteration. The input samples are padded with zeros in memory on each end so that we don't have to worry about exceeding the input array limits. Multiplying the filter coefficients with this zero padding introduces a few additional computations, but was found to be faster than treating the edges in a separate case for small filter sizes.

```
uint32_t n,n_out=0;
float re,im;
const float *cr = (const float *)&coeff_r[0];
const float *ci = (const float *)&coeff_i[0];

//perform convolution
for (n=0; n < INPUT_LENGTH; n+=DECIMATION) {
	re=0;
	im=0;
	p_in = (float *)&samples[n-(FILTER_LENGTH/2)+1];

	for (s=0; s < FILTER_LENGTH; s++) {
		re += p_in[s] * cr[s];
		im += p_in[s] * ci[s];
	}

	vid_out_r[n_out]=re;
	vid_out_i[n_out]=im;
	n_out++;
}
```

Snippet 1: Basic direct-form time domain convolution implementation

Figure 25 shows multiple performance results of this filter implementation, given as a percentage of peak performance. Here “*Throughput*” denotes the achieved output sample rate of a 32-tap Hilbert filter. “*FPU*” and “*IALU*” denote the percentage of peak floating point and integer performance respectively. “*Dual-Issue*” denotes the dual-issued instruction rate, and “*Reg. Stalls*” and “*Mem. Stalls*” denote the rate of pipeline stalls due to register dependencies and instruction fetches respectively. All other activity of the processor, such as branching, updating status flags etc. is shown as “Control”.

Note that the FPU results can be a mix of fused-multiply-add (FMADD) and other floating point instructions. Ideally only FMADD instructions are used to calculate the output samples, in which case the FPU and throughput percentages should be identical. The best-case scenario is when the throughput is at 100%, which can only occur when FPU is at 100%. This, in turn is only possible when there are very little (no) control instructions and all necessary IALU instructions are issued as dual-issue instructions as shown in Figure 7 (p.20).

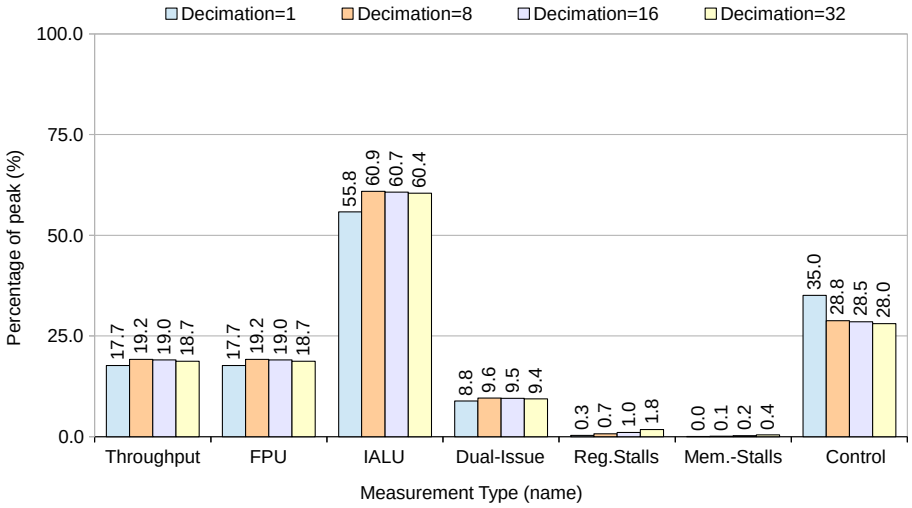


Figure 25: 32-tap Hilbert Filter (basic) performance metrics per eCore versus filter decimation for 512-element input video size

A best-case throughput of 19.2% is achieved in Figure 25. The IALU instruction rate is roughly 3 times higher than that of the FPU, and only a small fraction is issued as dual-issue instructions. This, and the high rate of control instructions suggests that the loop overhead is a significant bottleneck. When the inner loop is unrolled manually by a factor 8, the throughput of the filter increases to 30% of peak as shown in Figure 26.

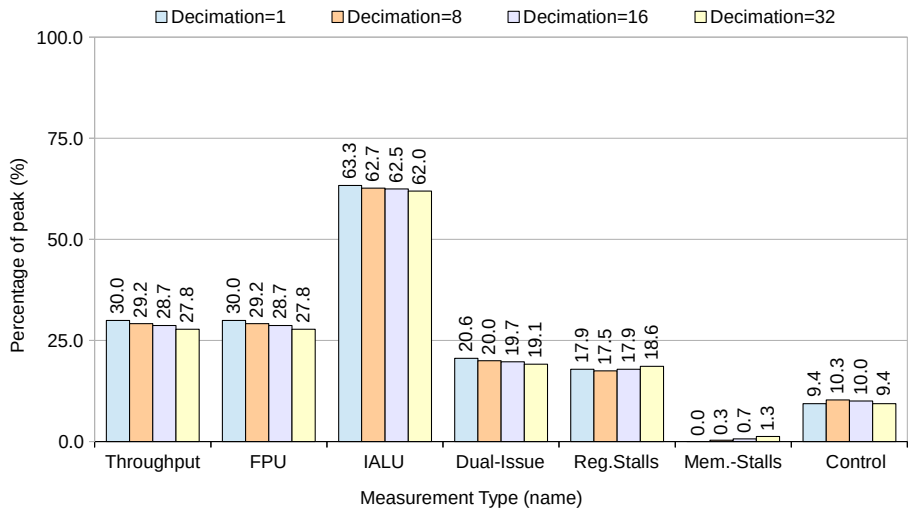


Figure 26: 32-tap Hilbert Filter (manually-unrolled) performance metrics per eCore versus filter decimation for 512-element input video size

As expected, both the FPU and IALU instruction rates go up, the dual-issue rate increases and the processor spends less time issuing control instructions. However, also a steep increase in register dependency pipeline stalls is observed. These are the result of inefficient register allocation by the compiler, where the accumulation variables *re* and *im* are reused for every fused multiply-add instruction. No compiler optimization options were found that improved this.

It is possible to let the compiler handle the loop unrolling automatically by passing “-funroll-loops” as an argument. This option is not enabled in any of the default compiler optimizations settings. The results of the original Hilbert filter with this option enabled are shown in Figure 27.

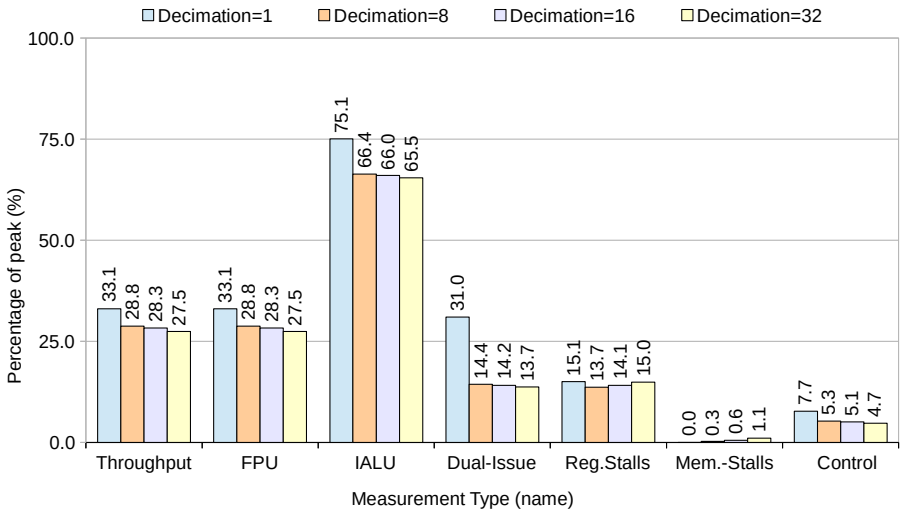


Figure 27: 32-tap Hilbert Filter (compiler-unrolled) performance metrics per eCore versus filter decimation for 512-element input video size

The results in Figure 27 are bit better than for the manually unrolled implementation in Figure 26 when there is no decimation in the filter (decimation=1). The reason for this is mainly an increase in dual-issue instructions, allowing for an overall increase in IALU and FPU instructions. Why the compiler does not manage to achieve this for the other cases is unclear.

A best-case throughput of only 33.1% of peak performance is achieved, dropping to around 28% when decimation is applied in the filter. This is mainly due to the high IALU instruction rate, a significant number of pipeline stalls due to register dependencies (13.7%-15.1%) and a relatively large portion of time spent issuing control instructions (4.7%-7.7%). Also, it should be noted that enabling the “-funroll-loops” compiler option for the entire application results in significant code-size increase, and in many cases this option resulted in failed compilation due to the resulting code not fitting in memory. For this research this option was only enabled for specific functions by using GCC compiler pragma's.

The IALU instruction overhead is caused mostly by load operations required to load the appropriate coefficients and input samples from memory. This can be improved by unrolling the loop over video items (multiplying multiple video samples with the same coefficient in the inner loop) instead of over coefficients (multiplying multiple coefficients with multiple video samples). This allows the reuse of the coefficients, reducing the number of required memory accesses. Also, this resolves most of the register dependencies, as for multiple output samples more than one accumulator variable is required. The results of this approach are shown in Figure 28.

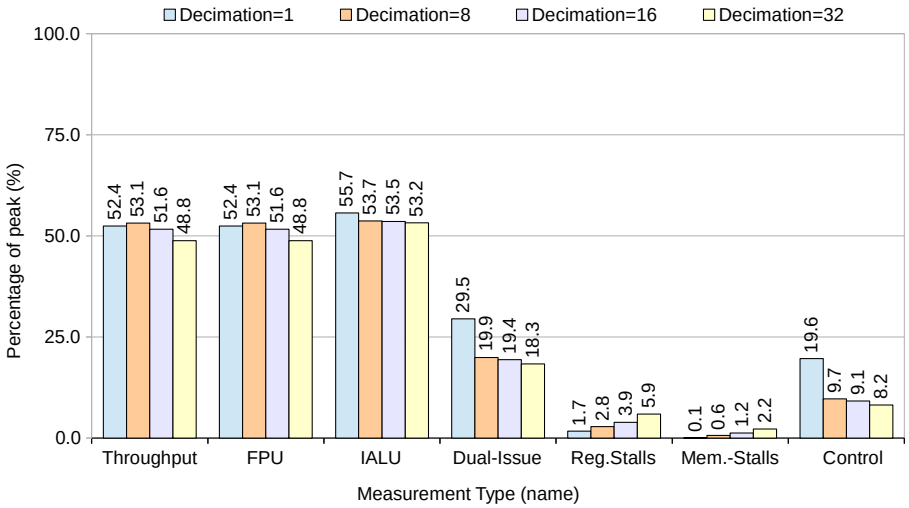


Figure 28: 32-tap Hilbert Filter (video-unrolled) performance metrics per eCore versus filter decimation for 512-element input video size

As expected the IALU overhead and register dependency stalls are reduced significantly. We now manage to achieve a throughput of 53.1% in the best-case scenario for a decimation factor of 8, with other decimation factors achieving between 48.8% and 52.4% of peak. Further optimizations could be performed by further unrolling the inner loop to reduce the required number control-instructions. Also it should be possible to increase the ratio of dual-issue instructions, however, we did not manage to achieve this using the provided C compiler. No attempt was made to improve this using assembly code. For small filter sizes it might be possible to reduce the number of required IALU operations further by keeping coefficients in the registers permanently.

Floating-point vs. fixed-point input samples

So far we have been working with floating point input samples. However, in practice, the ADC's will output fixed-point samples. This means a conversion is required. In Figure 29 we show the effect of using fixed-point input samples on the performance achieved in Figure 28 by changing only the input video data-type.

We see a dramatic performance decrease from 53.1% to 12.9% of the peak throughput, mainly caused by a large increase in register stalls. Also the FPU instruction rate is higher than the achieved throughput, indicating that not only fused-multiply-add instructions are being used. This is caused by a conversion performed by the compiler just prior to multiplying the input samples with the coefficients. This conversions takes more than one clock cycle to complete, stalling the pipeline. This could be improved by converting the input samples prior to filtering.

Ultimately, in the best-case, at least one clock cycle per input sample is required to perform the conversion, reducing the overall performance of the Hilbert filter tasks in any real-world system. The bandpass filter and beam-former do not suffer from this problem, as these will work on the single-precision floating-point output of the Hilbert filter stage.

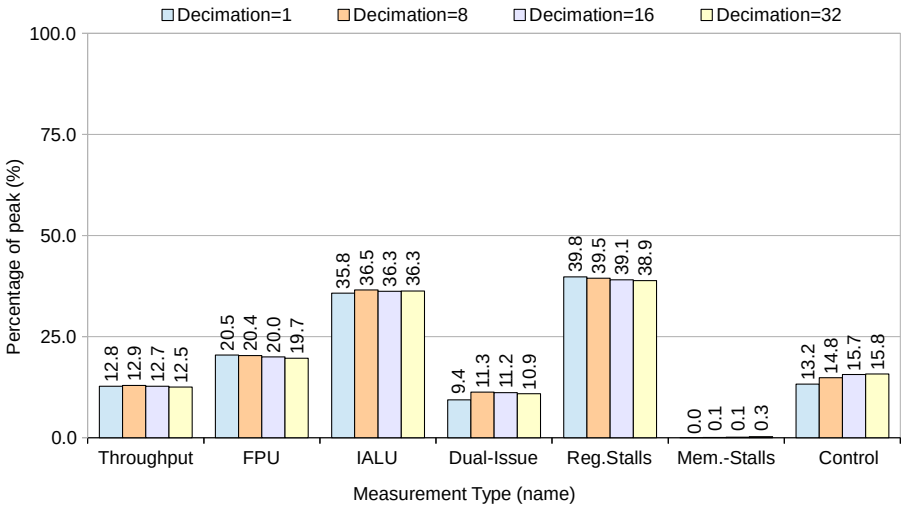


Figure 29: 32-tap Hilbert Filter (uint32_t input) performance metrics per eCore versus filter decimation for 512-element input video size

6.1.2 Bandpass Filter Optimization

For the bandpass filter we take the same approach as the Hilbert filter. However, where the Hilbert filter operates on a real-valued input stream, the bandpass filter operates on a complex-valued input stream. This results in a full-complex multiplication, requiring more operations, as shown in Snippet 2.

```
r0 += p_in_re[s] * c_re[s] - p_in_im[s] * c_im[s];
i0 += p_in_re[s] * c_im[s] + p_in_im[s] * c_re[s];
```

Snippet 2: Complex multiplication, direct-form

The ordering of the calculations matters due to rounding-errors. The compiler has to take this into account, which limits the options to optimize the statements above. The best case performance measured for the implementation above reaches 47.9% of peak, shown in Figure 30.

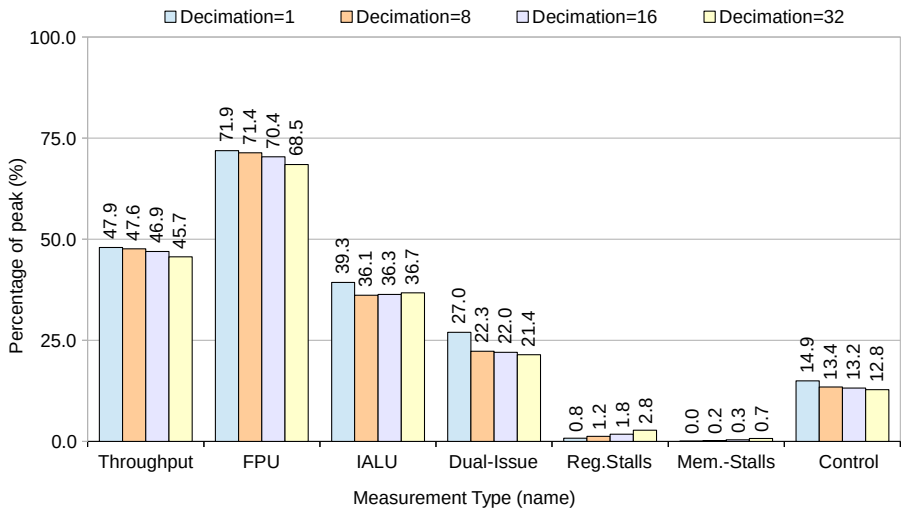


Figure 30: 32-tap Bandpass Filter (initial approach) performance metrics per eCore versus filter decimation for 512-element input video size

Figure 30 shows that the compiler does not only use fused-multiply add instructions, whereas when we split this multiplication over multiple lines as shown in Snippet 3, we get the results in Figure 31.

```
r0 += p_in_re[s] * c_re[s];
r0 -= p_in_im[s] * c_im[s];
i0 += p_in_re[s] * c_im[s];
i0 += p_in_im[s] * c_re[s];
```

Snippet 3: Complex multiplication, split direct-form

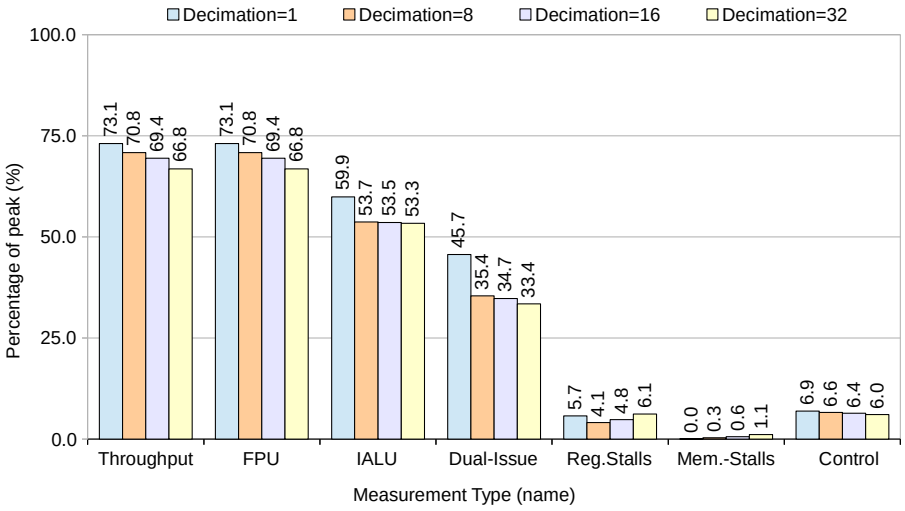


Figure 31: 32-tap Bandpass Filter (split-calculation) performance metrics per eCore versus filter decimation for 512-element input video size

There is a significant performance difference between the results in Figure 30 and Figure 31. It is possible to allow the compiler to re-order floating point operations automatically by setting the “-ffast-math” and "-fassociative-math" options during compilation, however, this can have unpredictable side-effects, and can violate existing floating-point standards. This option was tested (not shown here), and the throughput was still roughly 6% worse compared to the results shown in Figure 31.

The best-case performance of the bandpass filter in Figure 31 is 73.1% when no decimation is applied, gradually going down to 66.8% when the decimation factor increases. This is much better than was achieved for the Hilbert filter. The main reason for this that the full-complex multiplication in the bandpass filter requires more FPU operations per input and output sample. This reduces the ratio of load and store operations per FPU operation, which leads to a better ratio between FPU and IALU instructions, and consequently a better throughput. There is still a significant portion of time spent on instructions not directly related to calculating output samples. We did not manage to improve on this using the provided C compiler.

6.1.3 Beam-former Optimization

Figure 32 shows the results that were achieved for the beam-former, multiplying all samples in a single channel with a coefficient, and adding the result to an output beam. This process can be repeated for every channel, ultimately forming a single beam out of multiple input channels. There is no decimation in the beam-former. Instead the performance is plotted against several input video sizes.

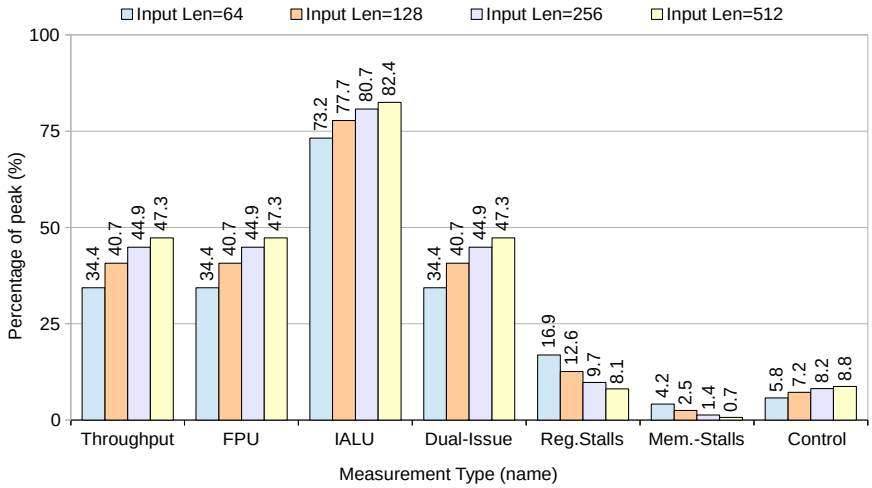


Figure 32: Beam-former performance for a single channel, multiple input lengths

A large IALU instruction overhead and some register stalls are observed. These are mainly due to the high amount of memory accesses required for every sample, since the current output value has to be fetched, the input sample has to be fetched, and the output value needs to be stored again for a single multiplication. This overhead can be reduced by processing multiple channels simultaneously, so that the current output beam-value can be re-used. This is done in Figure 33, where two input channels are processed simultaneously.

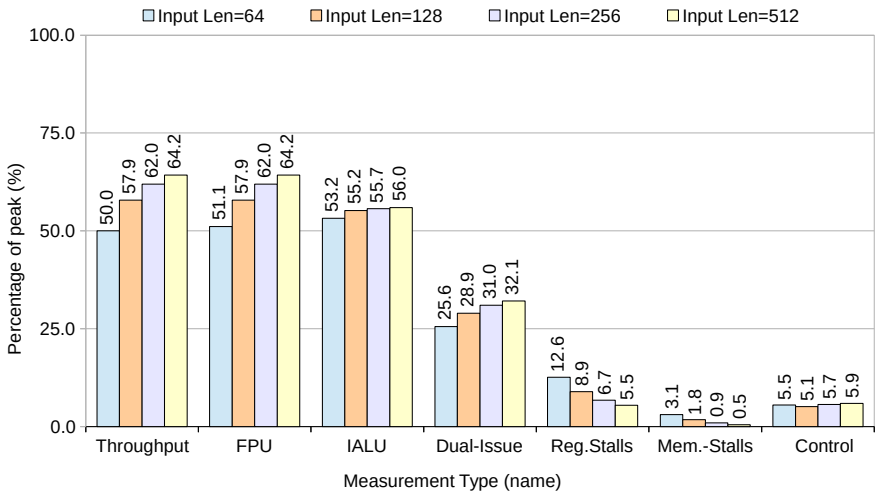


Figure 33: Beam-former performance while processing two channels simultaneously versus multiple input lengths

As expected, a decrease in the IALU overhead and an increase in overall performance is observed. It is important to note that in practice processing multiple input channels simultaneously might be difficult, since these would have to fit in local memory. Using small input sizes will introduce some overhead as shown in Figure 33.

6.1.4 Task Power Consumption

The power consumption of the Hilbert filter and bandpass filter without decimation (decimation=1), and the power consumption of the beam-former for input video lengths of 512 samples has been measured during operation at several core voltages. The results are shown in Figure 34.

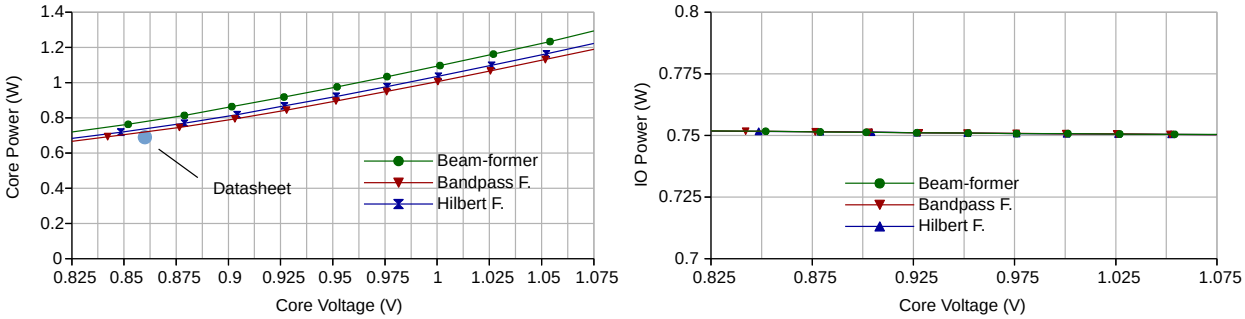


Figure 34: Core (left) and IO (right) power consumption for each individual task

The core power for all the tasks is slightly higher than was reported in the datasheet, and lower than our previous test-cases. We would expect the bandpass filter to consume the most power, as this is the best-performing tasks of the three, and uses both the FPU and IALU at a higher rate than both other tasks. Interestingly, it consumes the least power of the three tasks. When we replace the input data with 0-values, the power consumption drops significantly and the Bandpass filter now consumes more power than the Hilbert filter, as shown in Figure 35. This suggests that part of the lower power consumption of the bandpass filter can be explained by a difference in the input data, however the beam-former still consumes the most power. This is not investigated further.

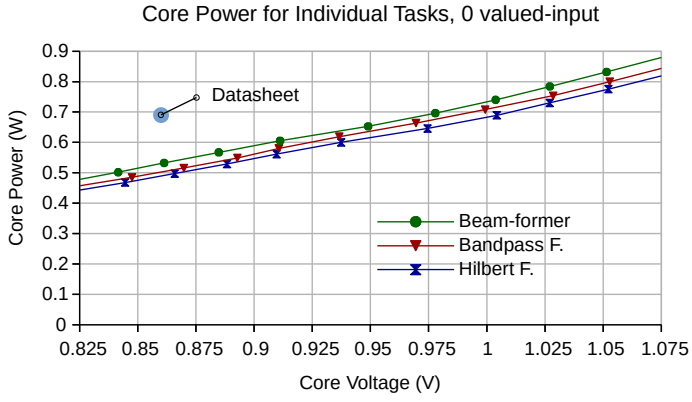


Figure 35: Core power consumption for each individual task (0 values as input samples)

With the results in Figure 34, we achieve a power efficiency of 14.35 GFLOPS/Watt for the Hilbert filter and 19.47 and 16.22 GFLOPS/Watt for the bandpass filter and beam-former respectively.

6.1.5 Task Implementation Summary and Further Improvements

So far we have seen that the performance of the individual tasks when using the C compiler is dependent on many factors. Several compiler options can be used to improve overall performance at the cost of memory size and loss of control. The peak throughput achieved for the Hilbert filter is 53.1% of the theoretical peak performance, for the bandpass filter and beam-former this is 73.1% and 64.2% respectively.

Achieving better performance using the provided C compiler proved to be difficult, however it is expected that some performance can still be gained by further inspection and optimization of the assembly output, as still not all IALU and FPU instructions are issued simultaneously, and some register dependency stalls still occur for all three tasks.

Also, using another convolution approach entirely, such as a direct-form FIR filter approach or a frequency domain approach might improve performance. A 16-tap direct-form FIR filter was reported to achieve 82% of peak performance in [24]. The main reason we do not achieve this level of performance here, is that complex coefficients are used. For a 16-tap real valued FIR filter, all coefficients can be kept in the register file for fast access, however for a complex valued filter with 32 coefficients as in our case, this would require all registers in the register file. Splitting up the filter kernel will introduce more control overhead, and will require a summation of intermediate results.

Performance of the beam-former can be improved by processing more than two channels simultaneously, however the memory footprint of the channels will quickly become a bottleneck in this approach.

The measured power efficiency of the individual tasks are 14.35 GFLOPS/Watt for the Hilbert filter and 19.47 and 16.22 GFLOPS/Watt for the bandpass filter and beam-former respectively. No network traffic was present in any of these measurements. The input data that was used for all tasks was pre-generated and stored in local on-chip memory

7 Front-End Receiver Design

In this chapter we present several different design considerations for the beam-former application and the effect of the Epiphany architecture on their implementation. First we introduce the topic of communication between processes on the Epiphany architecture. Next we discuss how processing loads can be matched by splitting up a problem into smaller concurrent tasks. This is then used to introduce a possible mapping of the beam-former on the Epiphany architecture and measure it's performance and power-efficiency.

7.1 PROCESS COMMUNICATION

The Epiphany architecture is optimized for on-chip writes. Although the flat memory space makes a shared memory approach possible, accessing data that is not in local memory results in slow read transactions over the network. Instead, for optimal performance, non-local data should be copied into local memory through write transactions before it is accessed. Since write transactions can only be issued from the node holding the data, a form of communication is needed for efficient data transfers between nodes. Typically, message passing is used for this.

In message passing systems, data is moved from the address space of one process to that of another process through explicit communication requiring coordinated actions on both sides. It has been applied on the Epiphany architecture in [21] and [7], showing that message passing can indeed achieve high performance on the Epiphany architecture.

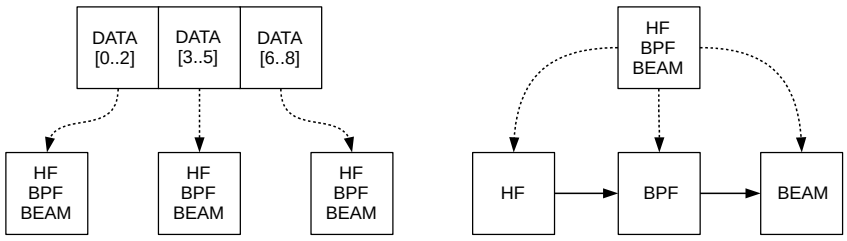
The implementation in [7] uses a send and receive buffer approach, where duplicate copies of data exist in memory for every transaction (one in the send buffer, and one in the receive buffer). To keep the memory footprint small, data is transferred in small blocks, introducing a performance penalty due to DMA start-up overhead. Another approach is used in [21], where results are written to a receiving buffer directly using direct-write transactions. Here there is no memory or processing overhead, and high data-rates are achieved. This seems like a better approach as it can also be used efficiently between tasks running on the same node if needed.

When tasks are blocked upon lack of data, or in case of input buffer overflow, the processing load between tasks must be well balanced. This is required if data-loss is not allowed. If the tasks are not well balanced, idle time will be introduced for the processes that have to wait for data being consumed or produced. In some cases this idle time can be compensated for by scheduling multiple tasks on the same node to fill the idle times. In order to do this efficiently on the epiphany architecture, task implementations and states need to be kept in local memory and a scheduler needs to be implemented.

Due to the limited memory on the Epiphany this approach can be impractical, and the high performance tasks implemented in [7], [18] and [21] run only one task per mesh-node that uses most of the available memory. Another option to balance the load of individual tasks is by decomposition. Tasks are split into multiple smaller parts that all match in terms of processing requirements. This is discussed in the next section.

7.2 PROBLEM DECOMPOSITION

Problem decomposition is the process of identifying parallel sub-tasks within a problem. It can can be divided into domain decomposition and functional decomposition. With domain decomposition, the input data that needs to be processed is divided over multiple tasks that then process it concurrently. Functional decomposition focusses on separating the problem into functionally different tasks, each processing a different step in the algorithm. Figure 36 shows an example of both approaches:



A. Domain Decomposition B. Functional Decomposition

Figure 36: Domain versus Functional Decomposition

7.2.1 Filter Stage Decomposition

Domain decomposition of the convolution operation in the filter stages can be done using the overlap-save or overlap-add methods [11]. The input data is split into smaller sections and each part is solved separately. A small overlap is required between the sections for correct operation. In the overlap-save method, some redundant samples are computed that are discarded, whereas with the overlap-add method partial results need to be added for the final result. This is shown in Figure 37.

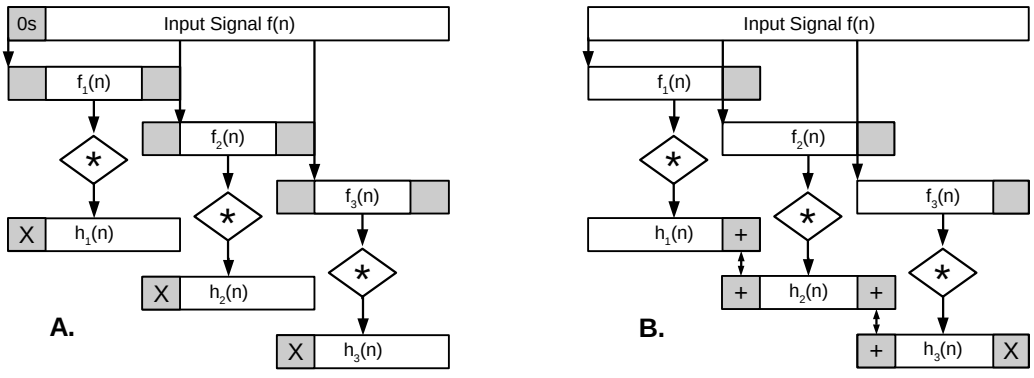


Figure 37: Overlap-Save (A.) and Overlap-Add (B.) convolution decomposition. An input signal is split into smaller parts and sub-results are concatenated (A) or summed (B) to produce the final result

Functional decomposition of the filters can be done by splitting the convolution kernel over multiple processes. This increases parallelism of the filters and reduces the processing load per task. Here each task still processes the entire input stream. This can introduce some communication and memory overhead since all tasks will need a local copy of the data for optimal performance.

Hilbert and bandpass filter throughput versus decomposition

In Figure 38 the throughput of both the Hilbert and Bandpass filters are plotted for different video input sizes, and three different filter coefficient lengths. The results in Figure 38 A. and B. for video lengths of 512 elements correspond to the results obtained for the filters in the previous chapter.

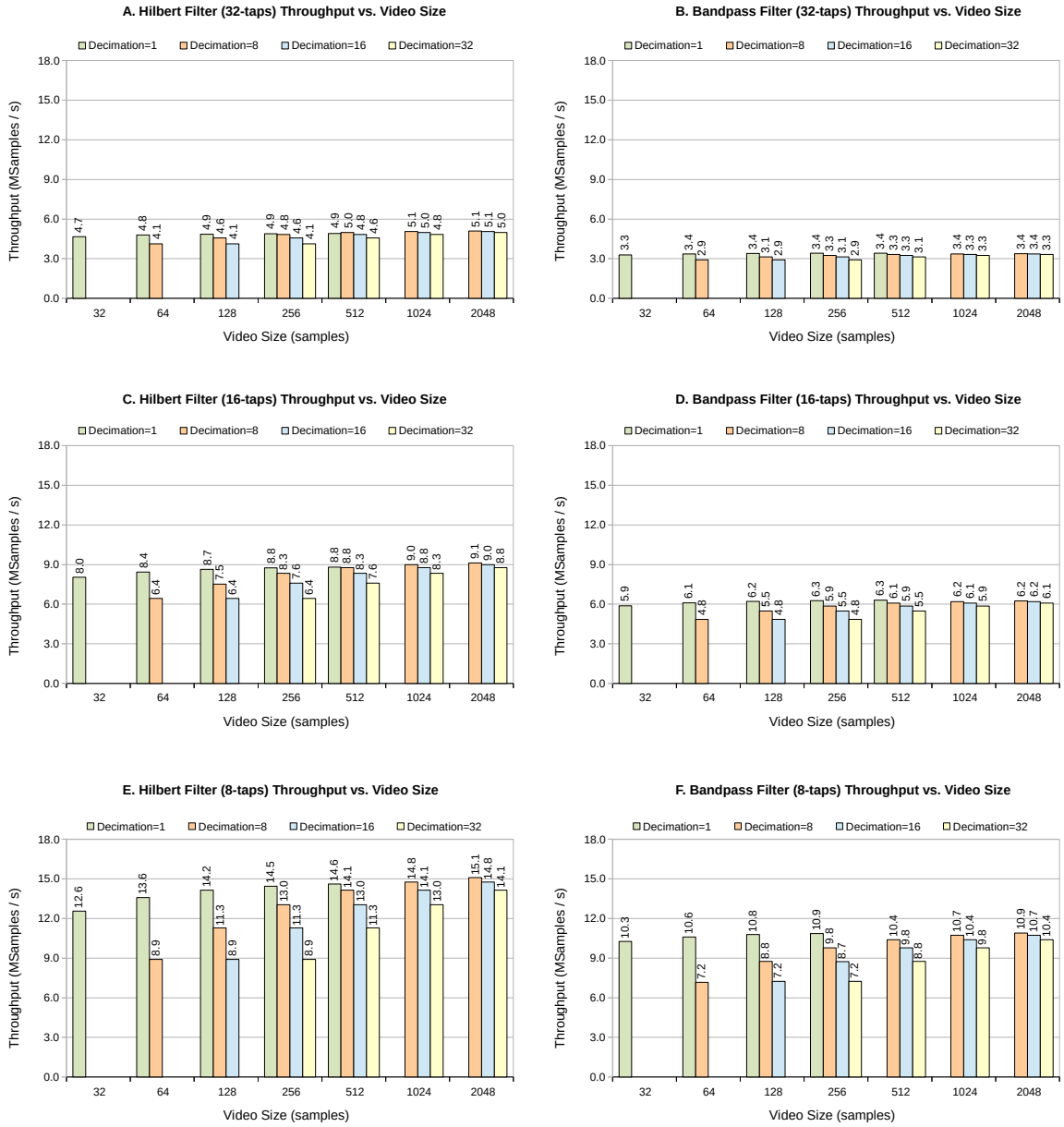


Figure 38: Hilbert and bandpass filter throughput versus filter size, input size and decimation factors

For small input video sizes, applying decimation will result in the loop-unrolling no longer being possible (for instance for 128 input samples with a decimation factor of 32 will result in 4 output samples, whereas we apply a loop unrolling factor of 8). These cases have not been measured.

When no decimation is applied, the output video size is the same as the input video size. Each complex output video sample consists of two 32bit single precision floating point numbers, results in 64bits per sample, or 8KiB and 16KiB of data for 1024 and 2048 output samples respectively. This size

is impractical in our situation, as the output video will either serve as the input for a next filter stage, in which case it will be zero padded and will no longer fit in a single 8KiB memory bank, or will go to the beam-former, where at least two channels should be processed for performance, making these video sizes infeasible. For this reason these cases are not measured.

7.2.2 Beam-Former Decomposition

Domain decomposition of the beam-former stage can be done by processing the phase shift for all channels separately, and by splitting the summation over multiple stages as shown in Figure 2 (p.14). The phase-shift operation can also be further split into multiple sections (one core processing the first half of the input video, and another core processing the second half). Functional decomposition can be done by separating the phase-shift and summation.

Figure 39 Shows the input throughput of the beam-former in MS/s (rather than a percentage of peak performance as in Figure 33) versus the input video size per channel. The output throughput can be derived from this by dividing the input throughput by the number of input channels per beam.

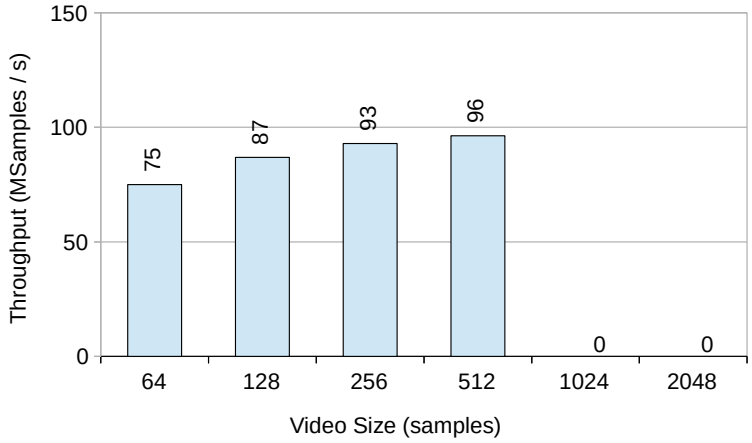


Figure 39: Beam-former input throughput in MS/s versus the input channel video size in number of samples (64bit/sample)

7.2.3 Load Balancing

Although problem decomposition can be used to match the task loads over multiple processors, it can introduce new dependencies as with the overlap-add method in Figure 37. Also, if functional decomposition is used, sequential tasks may still be unbalanced. This is the case for the mapping used on the Recore Xentium RFD, where for instance the band-pass filter is ~15% slower than the Hilbert filter. In the following section an example mapping of the front-end receiver is presented that takes the processing times of the individual tasks into account to minimize this idle time.

7.3 EXAMPLE FRONT-END RECEIVER MAPPING

In this section we present a sample mapping of a front-end receiver chain on the E16G301. The same load-case as that was used for the Recore Xentium RFD is investigated, however as we only have one E16G301 at our disposal, only part of the load-case will be mapped onto a single processor. The mapping focusses on achieving a good load balance between processors and minimizing the influence of data communication on the throughput and/or latency.

7.3.1 Load case and video sizes

The load-case used on the Recore Xentium RFD is repeated in Table 10 for convenience. Based on Figure 38, we can say that for a decimation of 8 in the Hilbert filter, the best performance is achieved for input video sizes of 512 samples or higher for a 32-tap filter. For the bandpass filter, with a decimation factor of 1, all input video sizes of up to 512 samples are suitable. The beam-former performs best at 128 samples per input video or more. For this research a bandpass filter and beam-former input video size of 128 video samples is chosen. Consequently, with a decimation factor of 8 in the Hilbert filter task, an input video size of 1024 samples is used for the Hilbert filter.

Specification	Value
Hilbert filter Decimation (D_{HF})	8
Bandpass filter decimation (D_{BPF})	1
Number of input channels (N_C)	16
Number of output beams (N_B)	8
Hilbert and bandpass filter taps (N_T)	32

Table 10: Epiphany architecture peak throughput figures for the front-end application domain

7.3.2 Task Mapping on the E16G301

For 32-taps, an input sample size of 1024 samples and a decimation factor of 8, the Hilbert filter can sustain 5.1 MS/s as shown in Figure 38. A 32-tap bandpass filter can only sustain 3.4 MS/s for an input video size of 256 samples without decimation. In order to balance the loads of the filters, a form of domain decomposition is applied. The output of two Hilbert filter tasks is split over thee bandpass filter tasks, all running on dedicated nodes. This allows all tasks to run at their maximum throughput, reaching a maximum throughput of 10.2 MS/s, or 5.1 MS/s per channel, using five mesh nodes for the filter tasks.

A single beam-former task can sustain a total input throughput of 87 MS/s if two 128 sample channels are processed simultaneously, as is shown in Figure 39. For eight output beams, this allows 5.4 MS/s per channel of input throughput if two channels are processed. This is slightly higher than the 5.1MS/s achieved for the filters, introducing an expected 6.2% of idle time for the beam-former tasks. This configuration is mapped twice onto a single E16G301, and is shown in Figure 40.

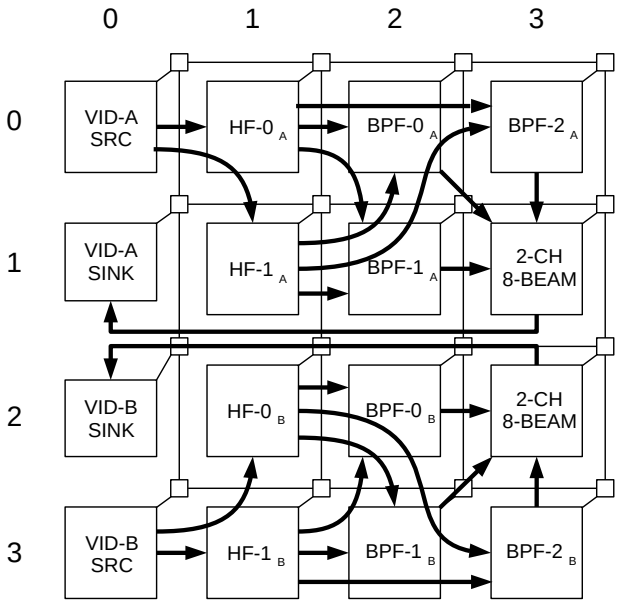


Figure 40: Task mapping of 4 channels, 8 partial beams on the E16G301

On the Parallella board, not all eLinks are available, and the bandwidth for transferring data from the host processors to the E16G301 was shown to be limited. To allow a more realistic test-scenario on the E16G301, the remaining nodes are used to generate video data on-chip. It is assumed that the eLinks can sustain 600MB/s instead of the 306MB/s measured in chapter 5. This is done based on the assumption that the measured bandwidth of 306MB/s is currently limited by the host hardware, and not necessarily by the eLink.

The required rates for this mapping can be calculated as shown in equation (8) and (9). (8) shows the input bandwidth for a single Hilbert filter tasks, based on the time it takes to process 1024 input samples of 32-bits each. Four Hilbert filter tasks are mapped, so the combined input bandwidth required is 652.8 MB/s. For the beam-former output, the input rate of the channels is 5.1 MS/s. With this rate, 8 output beams are formed, with samples consisting of 8 bytes each. Two of these tasks are mapped on the platform, resulting in an output bandwidth of 652.8 MB/s. Both rates are higher than a single eLink can support, so this has to be split over two ingoing and outgoing eLinks. This is simulated by two separate source and sink nodes in Figure 40.

$$BW_{\text{HF\_IN}} = \left(32\,bits \cdot 1024\right) \div \frac{128\,\text{Samples}}{5.1\ MSamples/s} = 163.2\ MB/s \tag{8}$$

$$BW_{\text{BEAM\_OUT}} = 8 \cdot 5.1\ MSamples/s \cdot 8\,bytes = 326.4\ MB/s \tag{9}$$

A schedule is needed to allow the sharing of the three bandpass filter tasks by the two Hilbert filter tasks. A schedule for the Hilbert filters and bandpass filters that allows this sharing is shown in Figure 41. A round-robin schedule is used in the beam-formers that handles all input bandpass filters.

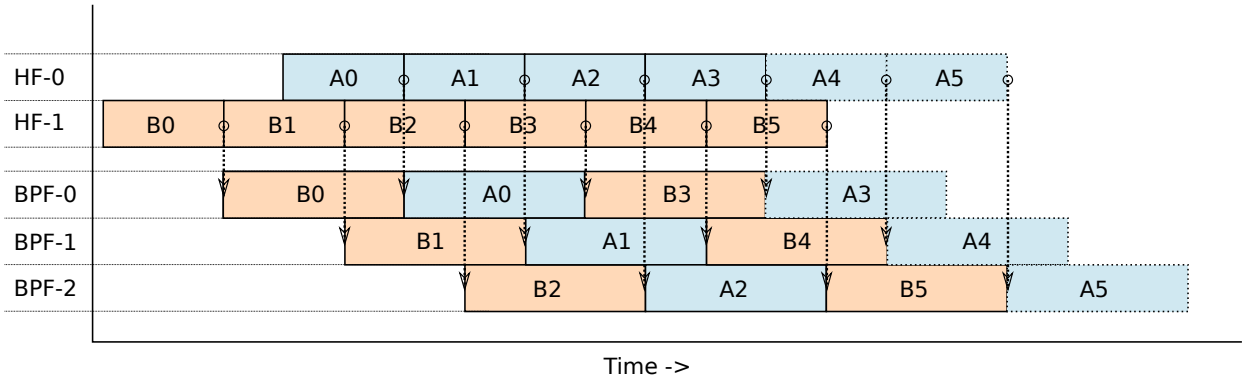


Figure 41: Hilbert filter and bandpass filter schedule used for mapping in Figure 40

Note that many other possible mappings and decompositions exist. Also, in many cases the scheduling can become quite complex. An unfruitful attempt was made to use a tool called SDF3 to automate this mapping, decomposition and scheduling process. A summary of this attempt is presented in appendix E: SDF3: Automated MPSOC Task Mapping.

7.3.3 Data Communication

In the schedule in Figure 41, tasks immediately begin processing the next set of data after the previous computation has finished. To avoid overwriting data currently being used for computations on the receiving node, a double buffer scheme is used where one input buffer is being filled by the preceding task, and the other is being used for calculations on the current node. Data communication is done through direct-writes and is blocking, meaning that if a buffer is being used, or is full, the producer has to wait until it becomes available. This buffer and communication scheme is shown more clearly in Figure 42 (note that only one bandpass filter task is shown, where in practice the Hilbert filter nodes alternate between the three bandpass filter nodes according to the schedule in Figure 41).

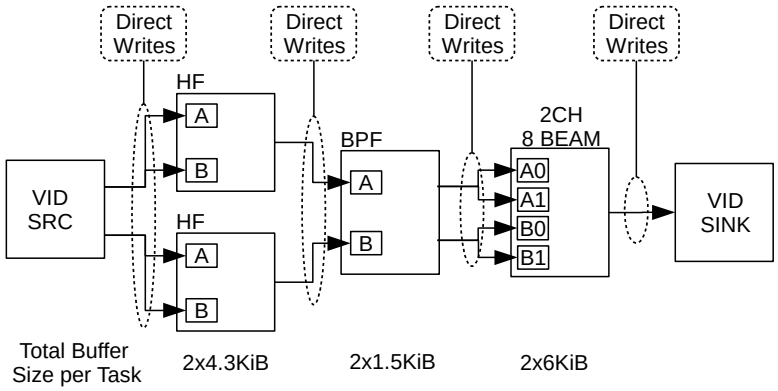


Figure 42: Front-end receiver mapping communication and buffer scheme

The Hilbert filter nodes have two input buffers for 1024 real-valued samples each, padded with 32 zeros on each end. This results in 4.3KiB per input buffer. The bandpass filter nodes have two input buffers for 128 complex-valued input samples, padded with 32 zeros on each end. This results in 1.5KiB per input buffer. Finally the beam-former has two input buffer pairs per bandpass filter, of 128 complex-valued elements each, resulting in 6KiB per pair for all bandpass filter nodes.

7.3.4 E16G301 Front-End Receiver Performance

Figure 43 Shows the achieved output throughput, floating point performance and slack times per tasks in the front-end receiver mapping shown in Figure 40, as well as several other performance metrics.

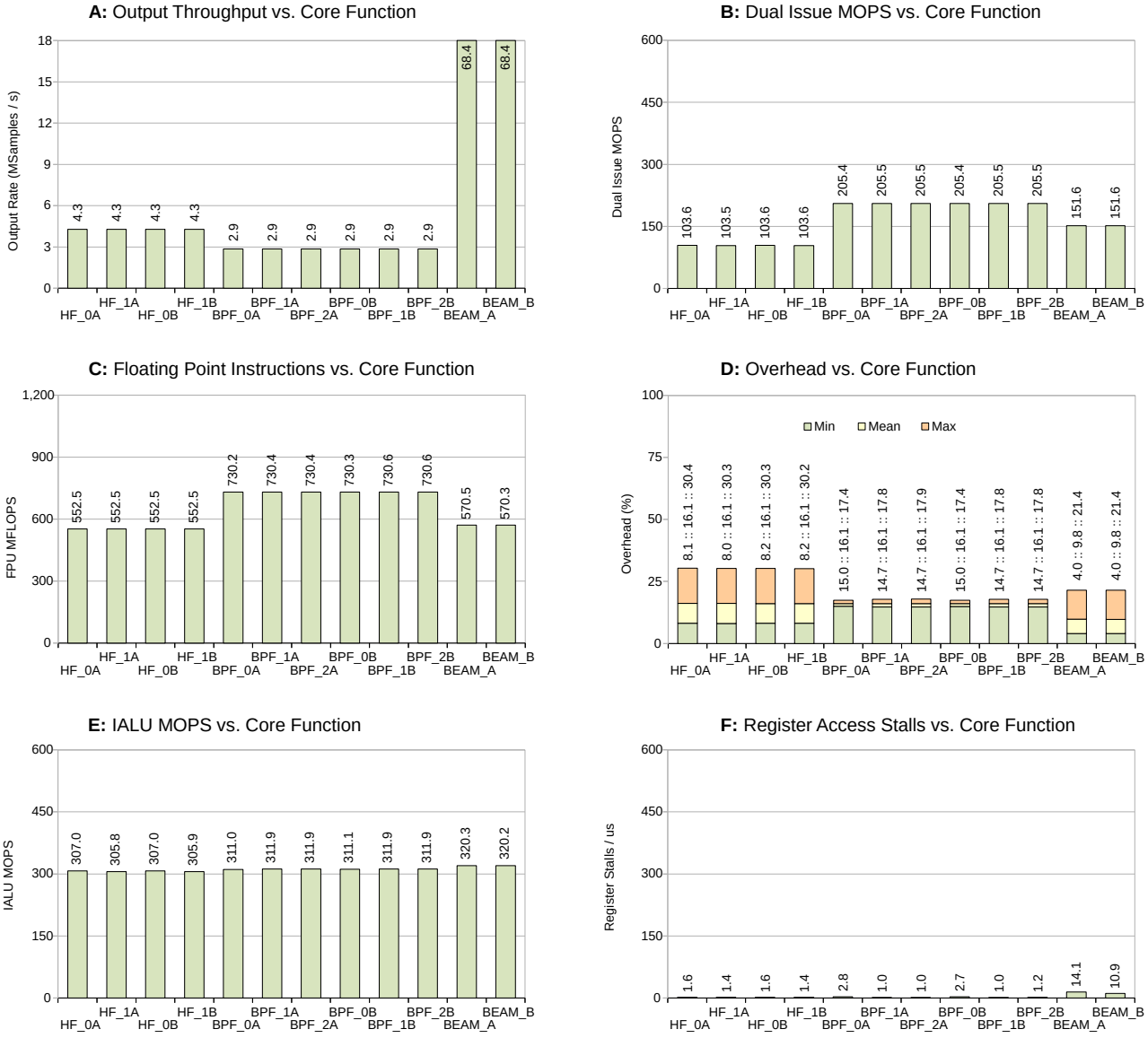


*Figure 43: Frond-end receiver mapping performance results versus core function. **A:** Output throughput. **B:** Dual Issue MOPS. **C:** FPU MFLOPS. **D:** Wait Times. **E:** IALU MOPS. **F:** Register Access Stalls*

When we look at the achieved output throughput per task in Figure 43 A. we see that all tasks suffer from a significant performance drop compared to the initial performance measurements (HF: 5.1MS/s to 4.3MS/s, BPF 3.4MS/s to 2.9MS/s and BEAM: 87MS/s to 68.4MS/s). This drop is caused by the synchronization and control overhead required to achieve the schedule in Figure 41, as well as a mismatch in throughputs per tasks. The combined overhead and slack-time is shown in Figure 43 D. The difference between the minimum overhead and the maximum overhead is caused by waiting on input data or output buffers to become available. A fraction of the minimum overhead can also be caused by this, however the exact fraction can not be determined based on this measurement data.

Further investigation is required to determine the exact cause of the performance drop of the tasks. It is likely that another mapping or schedule exists that performs better than what is presented here, however this is not easy to determine. For this research, the results in Figure 43 will be used for comparison with the Recore Xentium RFD.

Comparing the E16G301 and Recore Xentium RFD

Unfortunately we can only compare the Recore Xentium RFD based on performance per task and the front-end receiver as a whole. Also, the cores feature different amounts of execution units, the data-type used is different, the used clock frequencies are different, both chips are implemented in a different technology, and the task implementation was not done in the same programming language. This means the comparison only says something about the better performing implementation and chip, and does not say much about which architecture is more suitable for Radar applications.

Table 11 Shows the processing times for the individual tasks for both the E16G301 and the Recore Xentium RFD. Here we see that the input/output bandwidth and task loads are more evenly balanced for the E16G301 implementation, allowing for a higher sustainable throughput, even though the actual processing times are higher. For the Epiphany, a sustained input throughput of $4.3*D_{HF} = 34.4$ MS/s is achieved. The Xentium RFD can sustain an input samples rate of 25MS/s, which is mainly limited by the input bandwidth of the device.

Task to Process	Input Samples	Output Samples	Processing Time / Core	Idle Time / Core
Recore Xentium RFD				
Hilbert Filter (HF)	1024	128	12 µs	29 µs
Bandpass Filter (BPF)	128	128	14 µs	26 µs
Partial Beam-Forming (PF)	256	512	10 µs	12 µs
E16G301				
Hilbert Filter (HF)	1024	128	29.8 µs	4.8 µs
Bandpass Filter (BPF)	128	128	44.1 µs	7.1 µs
Partial Beam-Forming (PF)	256	1024	29.9 µs	2.9 µs

Table 11: Recore Xentium RFD and E16G301 front-end receiver performance comparison

Note: The comparison between the beam-former tasks is not entirely fair, as with our implementation, 16 2-channel intermediate beams are formed, and on the Recore Xentium RFD 8 4-channel intermediate beams are formed.

7.3.5 E16G301 Front-End Receiver Power Consumption

Figure 44 shows the measured power consumption during operation of the front-end-receiver. Some of the consumed power is used by the video sources and sinks, which are not present in a real-world implementation. These video sources and sinks only execute IALU operations, based on which we can assume that these nodes will have a similar power consumption as the “Memory” micro-benchmark shown in Figure 21 (p.41). This is used to estimate the portion of the power consumed by the front-end receiver tasks and network traffic, resulting in the estimated power consumption shown in Figure 44.

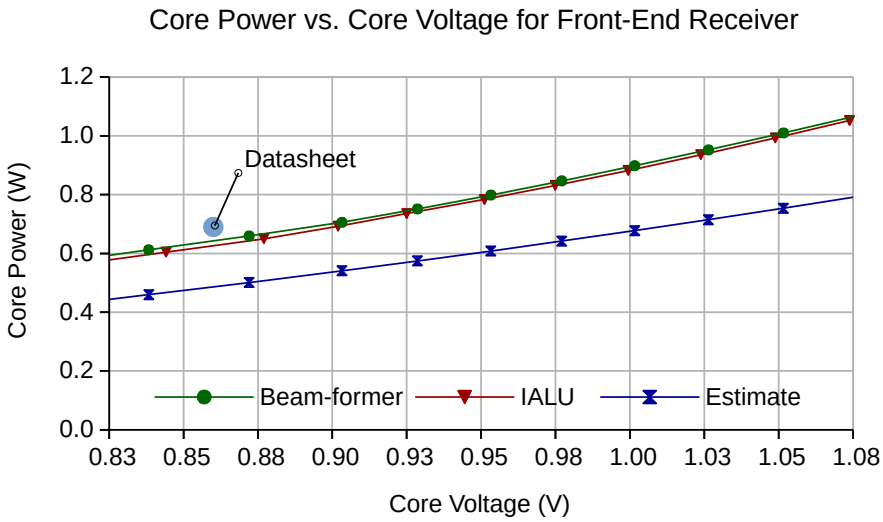


Figure 44: E16G301 Front-end Receiver Power Consumption

At the datasheet voltage of 0.86V, the estimated power consumption is 0.49 Watt. In total, 7.73 GFLOPS are achieved by all tasks (sum of Figure 43 B.), resulting in an estimated power consumption of the front-end receiver of 15.78 GFLOPS/Watt. This is roughly 57% of the suggest peak performance noted in the datasheet [14], and 49% of the absolute peak performance listed in the STARS many core survey shown in Table 12 (p.73).

8 Tools and Architecture

Due to time limitations, a thorough evaluation of the programming model, provided tools and Epiphany architecture design choices was not possible, and part of this research is left as future work. However, many observations were made simply by working with the tools and provided libraries, which are summarized in this chapter.

8.1 PROGRAMMING MODEL AND TOOL SUPPORT

As mentioned in chapter 2.2, Adapteva presents the Epiphany architecture are programming model neutral, essentially meaning that anything is possible. During this research it was found that developing software for a single eCore is straight forward, and can achieve high performance and efficiency. However, the large design space and many architecture details make mapping an application on this platform a challenging task. In this research only the provided C development environment was evaluated. Many other programming languages have been ported to this architecture, and even a supported OpenCL based environment exists [25].

Although OpenCL could provide a level of abstraction to ease development for the Epiphany, care must be taken with the host-Epiphany communication that is typical for OpenCL programs. This communication is bound to go over the eLink, and can drastically influence the performance of other tasks currently running on the system when sharing this eLink.

The topic of managing the large design space of many-core systems is still an active research topic, and several research project have already proposed several methods to quickly evaluate this design space ([26],[27], [28], [29]). An attempt was made to use a tool called SDF3 to automate the mapping process, without success. A summary of this attempt is given in appendix E.

8.1.1 Compiler

The provided e-gcc compiler is based on the open-source GCC compiler (v4.8.2). In [18] it was noted that it is difficult to get full performance out of the compiler, as it was reluctant to use all available registers. In this research, the compiler has been found to use all available registers for compiled code, however, the performance of the generated code is very sensitive to the way the algorithm is implemented. Getting full performance out of this platform using the C language requires a good understanding of the C language and it's keywords, knowledge of compiler optimizations and their effect on the produced code and a lot of detailed knowledge on the underlying architecture.

GCC itself is open source, has a large community, is well used and well documented, and allows for a large number of compiler optimizations that can be used to optimize for performance. This makes it a good choice for an open-source project like the Epiphany architecture. Also, the SDK reference manual ([30]) includes a nice overview of the available optimization flags and their influence.

Recently, an online version of the compiler ([31]) and editor have been made available, which translate C-code into assembly while you type. This allows for very quick feedback during optimization of performance critical functions. Also, the influence of different compiler optimizations can be easily determined. This just came out after the bulk of this research had been performed, and as such has not been tested thoroughly, however, at a first glance it seems to produce the same output as the compiler that was used for this research.

8.1.2 The Debugger and Instruction-Set Simulator

A port of the GDB debugger is provided, which can be used to debug programs either through a provided instruction-set simulator, or on the hardware directly. This debugger was used shortly, during initial setup of the measurement environment, and seems to work, however, debugging massively parallel applications is very challenging by nature. Many of the bugs found where either due to difficult to trace race conditions or involved host-to-Epiphany communication which is not easily supported by the debugger.

During this research, most of the debugging was done by sending status messages and performance measurement results to the host, as this least influences the program flow of an application. A more thorough evaluation of the debugger and the debugging environment is still left as future work.

8.1.3 Performance Evaluation and Profiling

To our knowledge, at the moment of writing, there is no tool support for convenient profiling on the Epiphany architecture. Also, only two, non auto-resetting, hardware timers are available to monitor program flow. This proved to be an important limitation during our performance measurements, as measurement times had to be short (see appendix D:Software Implementation Details) and only one performance metric could be monitored at a time.

For the measurements in chapter 6 and 7, the application under test had to be run multiple times to obtain the required measurement data. This limitation has been noted by the designers of the Epiphany architecture on the community forums ([32]), and might change in the near future.

8.1.4 SDK Software Libraries

No functional issues were found in the supported libraries, and the provided functions are convenient to use. However, many provided functions use deeply nested function calls making them unsuitable for fast register access or interrupt routines. Also the DMA copy functions introduce significant overhead as was shown in chapter 5.

Fortunately, all libraries are open source, allowing for easy custom adaptations of available library functions. Also, currently Adapteva is working on a community provided parallel software library called PAL ([33]), which will provide basic high performance building blocks such a matrix multiplication, filtering etc. As this effort is relatively new, it remains to be evaluated.

8.2 ARCHITECTURE DESIGN CHOICES

The Epiphany architecture was designed as a power efficient many-core co-processor, and as such many components have been designed towards low complexity to improve area and power efficiency. These design choices can have a large impact on and less tangible qualities such as reconfigurability, dependability and scalability. This influence is shortly discussed here.

8.2.1 Reconfigurability

Reconfigurability is the ability of a system to change its behaviour at design or run-time by changing hardware and/or software configurations. The software environment of the Epiphany architecture support run-time programming of the mesh-nodes without influencing the other mesh-nodes. This allows for a large degree of software reconfigurability. Also, the work-group programming model allows convenient movement of tasks spanning over multiple mesh-nodes within or over across chips.

8.2.2 Dependability

This describes the trustworthiness of a system. It consists of both health monitoring capabilities and robustness of the design. One important limitation of the Epiphany architecture in terms of dependability is the XY routing scheme used for the on-chip network. This means that if a single router fails, large amounts of mesh-nodes will no longer be able to communicate with each other, sine the routes between these nodes cannot be diverted.

An option exist to force certain routers to route incoming traffic towards a fixed direction, allowing some degree of repair, however, this in turn will also influence routes that in normal conditions would not suffer from the broken router. Also multiple routers will need to be reconfigured to fully avoid the broken router, requiring complex software management. With adaptive routing as is used on the Recore Xentium RFD architecture, routes are determined at run-time, and faulty network components can be more easily avoided.

To our knowledge, no specific health monitoring hardware is available on the Epiphany architecture, although this has not been thoroughly investigated. Custom solutions can be implemented through software mechanisms such as heartbeats or bit-error checks, however, this would introduce a performance of memory footprint penalty.

8.2.3 Scalability

Scalability is the ability of a system to handle growing amounts of work gracefully or its ability to be enlarged to accommodate that growth. The Epiphany architecture has been designed to easily scale to large amount of processors, and to even allow multiple processors to be linked together to increase the processing power of a platform.

Although this sounds like the Epiphany architecture should easily be able to handle growing amounts of work there are some limitations. For one, the eLink bandwidth does not scale with the amount of processors, but only with clock-frequency. This clock-frequency does not grow with a growing amount of processors, and as such the amount of available IO bandwidth per processing core will decrease if the on-chip core count is increased. This off-chip bandwidth is already somewhat limited. It is unclear if larger core-count processors will feature more off-chip links, but for now both the E16G301 and G4 feature only four eLinks. It has been recently announced that a new eLink design is in the making ([34]), supporting much higher bandwidths, possibly solving this problem.

When multiple chips are used to scale a processing platform, data movement between these chips will be done over the eLink interfaces. Care must be taken that the eLinks are not shared improperly, or saturated, when sending data across this platform to a certain mesh-node.

9 Conclusions and Future Work

Based on the results in this report we can draw a few conclusions on the suitability of the Epiphany architecture for use in front-end radar systems. This will be done based on the original research questions. Also, some interesting future work is presented.

9.1.1 Epiphany Architecture Performance

The micro-benchmark results in chapter 5 show that it is difficult to achieve maximum performance for the on-chip and off-chip networks of the Epiphany architecture. For on-chip communication only roughly 30% of the maximum achievable performance of 4.8GB/s was reached for both DMA transfer and direct writes, due to errata items, software overhead and further unknown bottlenecks. For the eLinks, 51% of the peak performance was achieved (306MB/s out of 600MB/s) using direct-writes, however this performance quickly drops when the link is shared by multiple nodes.

For the eCore processing performance, several micro benchmarks achieved 95% to 99% of peak processing performance, indicating that this is indeed possible for certain applications. The best power efficiency for the micro-benchmark test-cases was 21.8 GFLOPS/Watt, which is slightly lower than what is suggested in the datasheet.

For the individual tasks, the peak throughput achieved for the Hilbert filter is 53.1% of the theoretical peak performance, for the bandpass filter and beam-former this is 73.1% and 64.2% respectively. The measured power efficiency of are 14.35 GFLOPS/Watt for the Hilbert filter and 19.47 GFLOPS/Watt and 16.22 GFLOPS/Watt for the bandpass filter and beam-former respectively. It is expected that some performance gain is still possible through further optimization of the compiler generated code.

Finally, when mapping all the tasks on the E16G301 to form a front-end receiver chain, a sustainable input throughput of 34.4MS/s for four channels was achieved, forming 8 partial output beams for two sets of two input channels. This was done achieved at a power efficiency of 15.8 GFLOPS/Watt. This excludes actual off-chip communication and IO rail standby power.

Based on these results, we can conclude that a large portion (>50%) of peak performance can be achieved for the frond-end receiver tasks within allowable effort using the provided C compiler. However this does require the developer to have a good understanding of the architecture, C language and compiler optimizations. Running an arbitrary application on this architecture will likely not result in good performance.

9.1.2 Programming Model and Tool Support

The provided tools and libraries are still under active development, but already provide a good base for development. However, some of the provided library functions introduce significant software overhead. For optimal performance, it is likely that most of the performance-critical target software will have to be manually implemented. A new library is in the making that should provide basic high-performance building blocks for signal processing applications. This needs to be evaluated.

In general, the hardware, tools and libraries provided by Adapteva are very much community driven. It is somewhat unclear how they will develop, and what hardware and software will be available in the coming years. This might make it less suitable for military systems, where generally long product support contracts are required. However, the open-source nature of the project does allow for easy replacement by custom implementations of the same architecture.

9.1.3 Architecture

In terms of reconfigurability, dependability and scalability there are no significant advantages over the Recore Xentium RFD architecture, other than a deadlock-free on-chip network. One issue with scaling this architecture is the ratio between the on-chip processing power, and the off-chip bandwidth. A new eLink has been announced that might provided better performance, but this is yet to be released.

The deterministic XY routing quickly makes specific on-chip routers and links essential for correct operation of an application. The Recore Xentium RFD network is more dependable in this aspect, as here routes can are determined at run-time that can actively avoid faulty components.

Finally, the Epiphany architecture is optimized for floating-point performance. This can be an advantage for future systems compared to the Recore Xentium RFD where only 16-bit fixed point data-types can be used for full performance. For optimal performance, this does require the input samples to be single precision floating-point as well.

9.1.4 Future Work

In order to be able to use the Epiphany architecture (or other many-core architectures) efficiently in future systems, a suitable programming method and model is required. In this research the tasks where manually mapped onto the architecture, which is a difficult process. Further research of existing methods and tools, such as the SDF3 presented in appendix E could prove to be very valuable.

For the Epiphany architecture specifically, further research is required on the actual eLink performance, as this requires more suitable hardware than the Parallella board platform. Also, In order to be able to better compare this architecture to current systems, power measurements should be performed on current solutions.

References

[1] Adapteva, “Epiphany Architecture Reference REV 14.03.11.” Adapteva, 2011.
[2] “CRISP Project Website,” *Cutting Edge Reconfigurable ICs for Stream Processing*, 2011-2008. [Online]. Available: http://www.crisp-project.eu/.
[3] STARS Consortium, “STARS Project Website,” *Sensor Technology Applied in Reconfigurable Systems*, 2014-2011. [Online]. Available: http://www.starsproject.nl/.
[4] “CRISP Project Brochure FP7 (ICT-215881).” Recore, Mar-2011.
[5] J.W. Melissant, “Many-Core processors: State of the art many-core processor study,” Thales Nederland B.V., Hengelo, Market Study 0.3, 2013.
[6] BDTI, “The Art of Processor Benchmarking : A BDTI White Paper.” Berkeley Design Technology, 2006.
[7] J. A. Ross, D. A. Richie, S. J. Park, and D. R. Shires, “Parallel Programming Model for the Epiphany Many-Core Coprocessor Using Threaded MPI,” *CoRR*, vol. abs/1506.05442, Jun. 2015.
[8] Marcel D. van de Burgwal, Kenneth C. Rovers, Koen C.H.Blom, Andre B.J. Kokkeler, and Gerard J.M. Smit, “Adaptive Beamforming using the Reconfigurable Montium TP,” in *DSD 2010*, 2010, pp. 301–308.
[9] J.W. Melissant, R. Marsman, H. Schurer, H. van Zonneveld, and H. Kerkhoff, “STARS documentation : R3.1a V1.3: Specification of Novel Concepts.” Thales Hengelo, NL, 2012.
[10] David Ernesto Troncoso Romero and Gordana Jovanovic Dolecek, “Digital FIR Hilbert Transformers : Fundamentals and Efficient Design Methods,” in *MATLAB – A Fundamental Tool for Scientific Computing and Engineering Applications*, vol. 1, Puebla, Mexico: INTECH, 2012, pp. 445–482.
[11] Karas Pavel, Svoboda David, and Gustavo Ruiz, “Algorithms for Efficient Computation of Convolution,” in *Design and Architecture for Digital Signal Processing*, INTECH, 2013, pp. 179–208.
[12] “Parallella 1.X Reference Manual.” Adapteva, 14-Sep-2009.
[13] J. Glass and Lionel M., “The Turn Model for Adaptive Routing,” *J. Assoc. Comput. Mach.*, vol. 41, no. 5, pp. 874–902, Sep. 1994.
[14] “E16G301 Datasheet Rev.140311.” Adapteva, 14-Mar-2011.
[15] Timon D. ter Braak, “Run-time Spatial Resource Management in Heterogeneous MPSoCs,” University of Twente, Enschede, 2009.
[16] Maurizio Palesi and Masoud Daneshtalab, “Basic Concepts on On-Chip Networks,” in *Routing Algorithms in Networks-on-Chip*, Springer, 2014, pp. 1–16.
[17] Sébastien Baillou and Klaas Hofstra, “CRISP Beam-forming implementation.” Recore Systems BV, 07-Apr-2011.
[18] Anish Varghese, Bob Edwards, Gaurav Mitra, and Alistair P Rendell, “Programming the Adapteva Epiphany 64-core Network-on-chip Coprocessor,” in *IPDPSW*, Phoenix, AZ, 2014, pp. 984–992.
[19] Zain-ul-Abdin, Anders Ahlander, and Bertil Svensson, “Energy-Efficient Synthetic-Aperture Radar Processing on a Manycore Architecture,” in *ICPP*, Lyon, 2013, pp. 330–338.
[20] Zain-ul-Abdin, Anders Ahlander, and Bertil Svensson, “Real-time Radar Signal Processing on Massively Parallel Processor Arrays,” in *Asilomar 2013*, Pacific Grove, CA, 2013, pp. 1810–1814.
[21] D. Richie, J. Ross, S. Park, and D. Shires, “Threaded {MPI} programming model for the Epiphany {RISC} array processor,” *J. Comput. Sci.*, vol. 9, no. 0, pp. 94 – 100, 2015.
[22] Andreas Olofsson, “Epiphany Bandwidth Test,” May-2014. [Online]. Available: https://github.com/adapteva/epiphany-examples/tree/2015.1/apps/e-bandwidth-test.

[23] “Parallella Example Repository,” *Parallella-Examples*, Jun-2015. [Online]. Available: https://github.com/parallella/parallella-examples/tree/master/matmul_optimized.
[24] Yaniv Sapir, “Approaching Peak Theoretical Performance with standard C,” *Approaching Peak Theoretical Performance with Standard C*, 09-Mar-2012. [Online]. Available: http://www.adapteva.com/white-papers/approaching-peak-theoretical-performance-with-standard-c/.
[25] Browndeer, “COPRTHR Repository,” Jul-2015. [Online]. Available: https://github.com/browndeer/coprthr.
[26] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere, “A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MP-SoCs,” presented at the CODES+ISSS’07, Salzburg, Austria, 2007, pp. 9–14.
[27] Rabie Ben Atitallah, Smail Niar, Samy Meftali, and Jean-Luc Dekeyser, “An MPSoC Performance Estimation Framework Using Transaction Level Modeling,” presented at the RTCSA 2007, University of Lille, France, 2007, pp. 525–533.
[28] T. Basten, E. van Benthum, M. Geilen, M. Hendriks, F. Houben, G. Igna, F. Reckers, S. de Smet, L. Somers, E. Teeselink, N. Trčka, F. Vaandrager, J. Verriet, M. Voorhoeve, and Y. Yang, “Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset,” in *Leveraging Applications of Formal Methods, Verification, and Validation*, vol. 6415, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2010, pp. 90–105.
[29] Sander Stuijk, “Predictable Mapping of Streaming Applications on Multiprocessors,” PHD Thesis, University of Eindhoven, Eindhoven, 2007.
[30] “Epiphany SDK Reference.” Adapteva, 2013.
[31] Andreas Olofsson, “GCC Explorer,” 07-Aug-2015. [Online]. Available: http://gcc.parallella.org/. [Accessed: 10-Aug-2015].
[32] Adapteva, “Parallella Community Forum,” *Parallella Memory Benchmark (11-12-2012)*, 2012. [Online]. Available: http://forums.parallella.org/viewtopic.php?f=23&t=1972&p=12095#p12095.
[33] Community, “Epiphany PAL Software Repository,” 13-Aug-2015. [Online]. Available: https://github.com/parallella/pal.
[34] Andreas Olofsson, “An open source 8Gbps low latency chip to chip interface,” 07-Aug-2015. [Online]. Available: http://www.parallella.org/2015/08/07/an-open-source-8gbps-low-latency-chip-to-chip-interface/.

Appendix

A. THALES MANY-CORE PROCESSOR SURVEY

The following tables show the result of a survey performed during the STARS project ([3]). It lists several many-core processors and their advertised peak performance. The processor used for this research is the E16G301 in Table 14, as an alternative to the Epiphany G4, which unfortunately was not available for purchase.

	Bittware Annemone	Tilera TILE-Gx	Adapteva Epiphany G4	Adapteva E16G301
Cores / Threads	16	16-100	64	16
Max. Frequency	1 GHz	1-1.5 GHz	700MHz	1GHz
Fixed / Floating Point	32b Floating	64b Fixed 32b Floating	32b Floating	32b Floating
Performance	32 GFLOPS	50 GFLOPS	88 GFLOPS	32 GFLOPS
Max. Power	2W	55W	1.25W	1W
GFLOPS / Watt	16	0.9	70	32
Off-Chip Bandwidth	8 GB/s	20 GB/s	5.6 GB/s	8 GB/s
Bandwidth / GFLOP	0.25 GB/s	0.4 GB/s	0.06 GB/s	0.25 GB/s
Programming Language	C	C / C++	C, OpenCL	C, OpenCL

Table 12: Thales many-core survey results taken from [5] : Part 1

	XMOS XS1-G4	CSX700 ClearSpeed	KALRAY MMPA 256	TI 66AK2H12
Cores / Threads	32 Threads	192	256+16	4+8
Max. Frequency	400 MHz	250MHz	400MHz	1.4/1.2 GHz
Fixed / Floating Point	32b Fixed	32b Floating 64b Floating	32b Floating	32b Floating
Performance	1600 MIPS	96 GFLOPS	230 GFLOPS	153 GLOPS
Max. Power	1.8W	<15W	5W	14W
GFLOPS / Watt	1.3 GIPS/W	6.4	46	11
Off-Chip Bandwidth	0.8 GB/s	4 GB/s	18 GB/s	5 GB/s
Bandwidth / GFLOP	0.5 GB/s	0.04 GB/s	0.08 GB/s	0.03 GB/s
Programming Language	C / C++	C	C / C++	C / C++

Table 13: Thales many-core survey results taken from [5] : Part 2

B. POWER MEASUREMENT LOAD CASES

All the test-cases below are unrolled 128 times within a single loop iteration, looping indefinitely. Care was taken to avoid register dependency and memory load pipeline stalls, by not reusing registers within 8 clock cycles (longest instruction execution), and by alternating memory banks for instruction fetch, memory loads and memory stores. All the power measurements were performed with pre-loaded dummy data in the registers.

```
ldrd rAA,[rBB,#2];  //double-word memory load (bank 1)
strd rCC,[rBB,#4];  //double-word memory store (bank 2)
ldrd rDD,[rBB,#6];  //double-word memory load (bank 1)
strd rEE,[rBB,#8];  //double-word memory store (bank 2)
```

Snippet 4: Load case 1: Memory Access only (uses IALU at ~85% of peak)

```
fmadd rAA,rBB,rCC; //fused-multiply-add
fmadd rDD,rEE,rFF; //fused-multiply-add
fmadd rGG,rHH,rII; //fused-multiply-add
fmadd rJJ,rKK,rLL; //fused-multiply-add
```

Snippet 5: Load case 2: FPU (reached ~100% of peak)

```
add rAA,rBB,rCC; //integer-add
add rDD,rEE,rFF; //integer-add
add rGG,rHH,rII; //integer-add
add rJJ,rKK,rLL; //integer-add
```

Snippet 6: Load case 3: IALU (reached ~100% of peak)

```
fmadd rAA,rBB,rCC; //fused-multiply-add
add rDD,rEE,rFF;   //fused-multiply-add
fmadd rGG,rHH,rII; //fused-multiply-add
add rJJ,rKK,rLL;   //fused-multiply-add
```

Snippet 7: Load case 4: FPU + IALU (reached ~100% of peak)

```
ldrd rAA,[rBB,#2];  //double-word memory load (bank 1)
fmadd rCC,rDD,rEE;  //fused-multiply-add
strd rFF,[rGG,#4];  //double-word memory store (bank 2)
fmadd rHH,rII,rJJ;  //fused-multiply-add
```

Snippet 8: Load case 5: FPU + Memory (reached ~85% of peak)

```
idle;  //put the eCore to sleep until woken by interrupt
```

Snippet 9: Load case 6: Idle

C. POWER MEASUREMENT SETUP

The electronics on the Parallella board are powered by the two regulators shown in Figure 45. The E16G301 is powered by two separate supplies, the VDD_DSP supply on U30, and the 1P8V supply on U29. The VDD_DSP output powers only the E16G301 core, and comes from a regulator that is controllable through I2C which has a range of 0,825 volts to 1.2+ Volts. This I2C interface is accessible from the ARM-host processors.. The 1PV8 output voltage is fixed trough a resistor divider, and is shared between multiple devices, such as the HMDI and USB controllers and the E16G301.

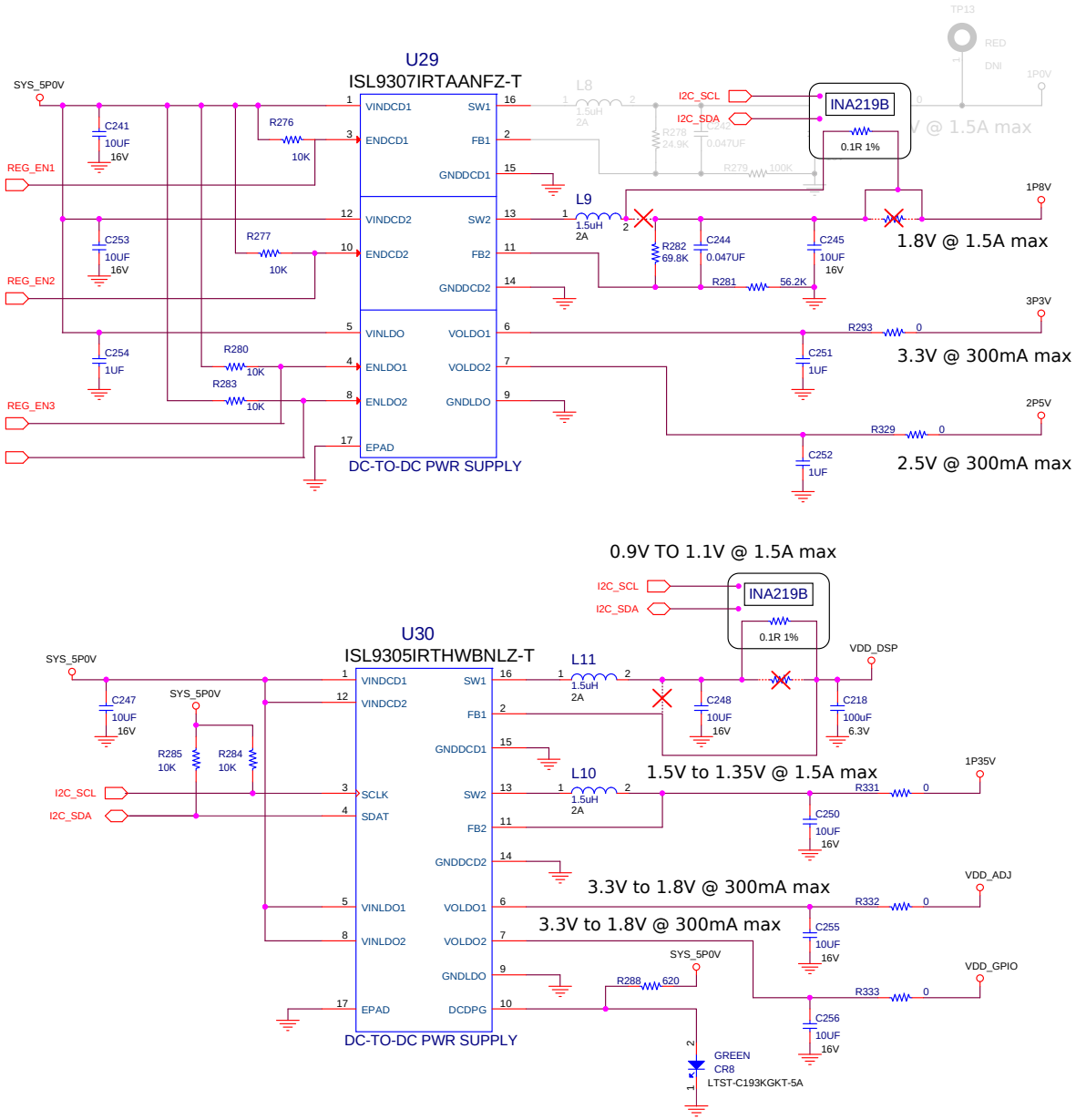


Figure 45: Parallella power regulator schematic

INA219B

Figure 45 shows the modifications that were made to allow automated and synchronized power measurements. Two INA219B boards have been added in series with the regulator output, that measure both the regulator output voltages and currents.

The INA219 is a high-side current shunt and power monitor with an I2C interface. It monitors both the voltage drop over a shunt, and the output voltage on the “low-side” of the shunt. For the measurements done in this report, the current and voltage ADC samples were read directly from the INA219B registers, and conversion to the proper units was done in software on the ARM processors.

For both INA219B boards, a 0R100 1% shunt was used, accompanied with a simple input filter to account for the low sample rates (200Hz per INA219B). The relatively high shunt allows for good measurement accuracy, but comes with a significant voltage drop. To account for this, the parallella board was modified such that the regulator feedback inputs were put after the shunt. These inputs draw a negligible input current, and allow the regulator to regulate the voltage after the shunt, rather than prior to the shunt.

Table 14 was taken from the INA219B datasheet by Texas Instruments, it shows the measurement accuracy and conversions times for the INA219B (highlighted in grey). The LSB step sizes correspond to 100uA and 4mV for the current and voltage measurement respectively. All measurements were performed at room temperature.

PARAMETER	TEST CONDITIONS	INA219A			INA219B			UNIT
		MIN	TYP	MAX	MIN	TYP	MAX	
DC ACCURACY								
ADC Basic Resolution			12			12		Bits
1 LSB Step Size								
Shunt Voltage			10			10		µV
Bus Voltage			4			4		mV
Current Measurement Error			±0.2	±0.5		±0.2	±0.3	%
over Temperature				**±1**			**±0.5**	**%**
Bus Voltage Measurement Error			±0.2	±0.5		±0.2	±0.5	%
over Temperature				**±1**			**±1**	**%**
Differential Nonlinearity			±0.1			±0.1		LSB
ADC TIMING								
ADC Conversion Time	12-Bit		532	586		532	586	µs

Table 14: INA219B Measurement accuracy summary (highlighted in grey)

The inaccuracy of the shunt contributes to the power measurement error as follows:

$$e_{PWR} = \sqrt{\left(e_{SHUNT} + e_{CURRENT}\right)^2 + \left(e_{VOLTAGE}\right)^2} = \sqrt{\left(0.01 + 0.003\right)^2 + \left(0.005\right)^2} = 1.39\,\% \quad (10)$$

The reason this setup was chosen over possibly more accurate off-the-shelf solutions, is that a lot of measurements were to be performed. The easy control of the INA219B over the I2C bus allows for convenient automation and synchronization of the measurements with the test applications, significantly speeding up the measurement process.

D. SOFTWARE IMPLEMENTATION DETAILS

This section contains information on the profiling method used, the tool-chain settings and the programming method of the E16G301.

Programming the E16G301: Work-groups

Adapteva provides several utilities to aid in the programming of the Epiphany architecture. These utilities are built around a relocatable work-group model shown in Figure 46. Here, multiple E16G301 chips are arranged in the global address space. All nodes within a single E16G301 chip are assigned a global address relative to the origin of the E16G301. In Figure 46 the origin is (32,8) for the left E16G301. This is also used on the Parallella board.

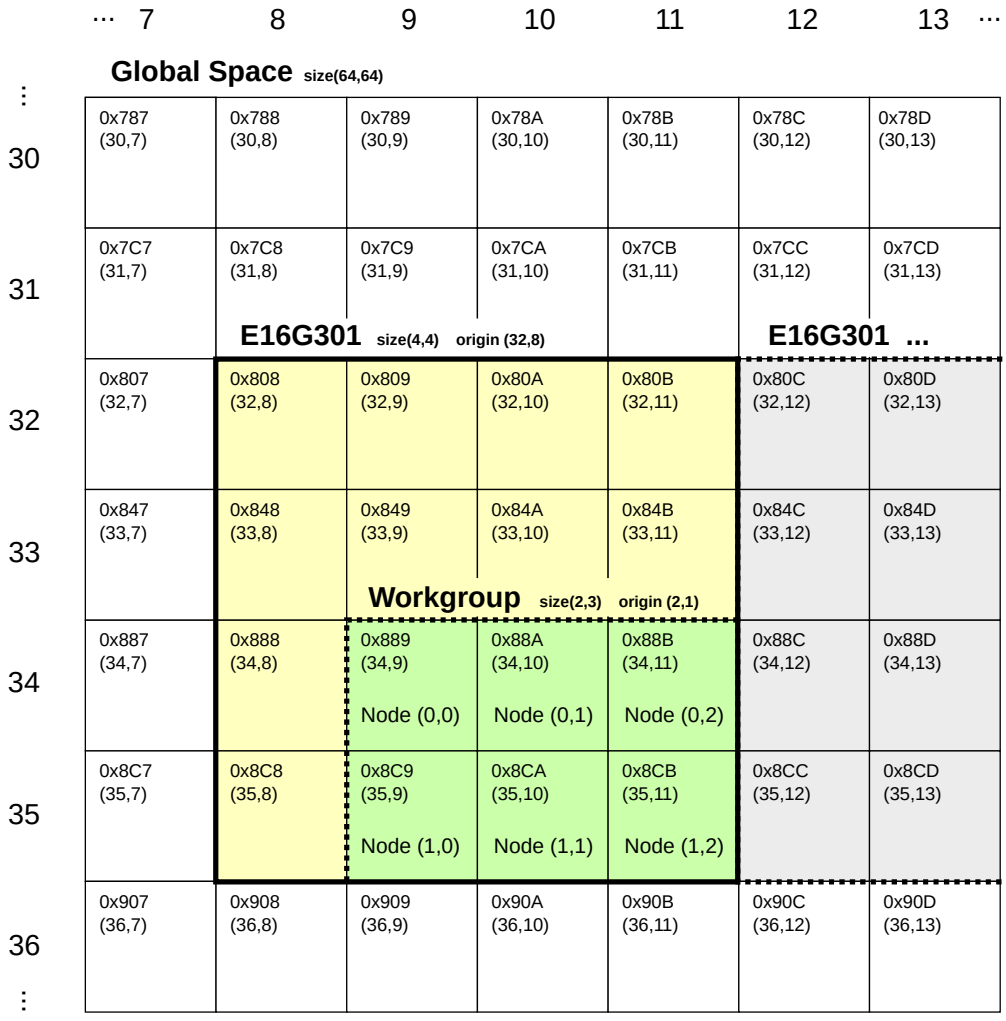


Figure 46: Epiphany platform, work-group and eCore coordinate system

A tool called the e-loader is provided, which allows uploading binaries to the mesh-nodes and starting execution. This is done on a work-group basis. A work-group is a collection of adjacent nodes that shared an aliased coordinate system. The nodes in a work-group can run the same of different binaries, and can be programmed independently. A work-group is always rectangular, and it's size and origin are defined at run-time. To program a single core, a work-group size of 1x1 can be used.

The nodes within a work-group are referenced in terms of their row and column in a work-group, relative to the work-group origin. This allows free placement of work-groups across the Epiphany platform. The global address of the nodes will change if the work-group is moved, so this should only be used to address specific nodes, not nodes relative to the current node.

The provided SDK utilities, such as the e-loader, work with a so-called hardware description file to define the platform on which they operate. This file contains a description of the chips in the platform, in our case only the E16G301, their origin and the presence of external memory. An example is given in Table 15. Here the EMEM_BASE_ADDRESS is the base address of the section in DRAM that is reserved for shared memory on the parallella board, as seen from the ARM host processors.

```
// Platform description for the
// Parallella/1GB/E16G3
PLATFORM_VERSION   PARALLELLA1601
ESYS_REGS_BASE         0x70000000

NUM_CHIPS                     1
CHIP                      E16G301
CHIP_ROW                     32
CHIP_COL                      8

NUM_EXT_MEMS                  1
EMEM                    ext-DRAM
EMEM_BASE_ADDRESS      0x3e000000
EMEM_EPI_BASE          0x8e000000
EMEM_SIZE              0x02000000
EMEM_TYPE                    RDWR
```

Table 15: Parallella Board hardware description file (HDF) for the board used in this research

The EMEM_EPI_BASE address is an alias to the EMEM_BASE_ADDRESS, which is used within the mesh-node applications. On the host-side on the parallella board, transactions to the EMEM_EPI_BASE address space are translated to the EMEM_BASE_ADDRESS. This translation allows the user to virtually place the shared memory anywhere in the address space, such that transactions with this shared memory are routed in the right direction on the on-chip network.

In this case, 0x8e0 represents the (35,16) coordinate. Since column 16 is not on the E16G301 chip, but EAST of this chip in the global address space, all traffic with this destination is routed over the EAST eLink, arriving at the FPGA and ARM host processors on the Parallella board. Note that this only works as long as the size of the shared memory is not larger than 1MiB, since after this, some of the top 12 bits are needed to represent all the addresses in the shared memory, which will cause the destination coordinate to change. Slightly larger regions are possible in our particular case, since changing the five least significant bits of 0x8e0 will result in a coordinate range of (35,16-63). All these coordinates are east of the E16G301 mesh-nodes, allowing for a 52MiB contiguous shared memory space. In practice, the maximum contiguous shared memory size depends on the placement of the E16G301 in the global address space shown in Figure 46.

Tool chain

All research was done used the SDK provided by Adapteva. An alternative development environment is available, the COPTHR IDE by brown deer. However in the background this uses the compiler and libraries provided by Adapteva ([25]). This means for this research there is little gain using this environment other than for programming convenience at the cost of performance and control. Also all examples provided by Adapteva are written for the SDK provided by Adapteva.

Table 16 shows the compilers and settings used for all measurements performed in research.

Item	Settings
ARM Compiler	arm-linux-gnuabihf-gcc (version 4.8.2)
ARM Compiler Flags	-Wall -std=gnu99
ARM Linker Flags	-Wall -std=gnu99 -lmatio -lm -le-hal -le-loader
Epiphany Compiler	e-gcc (v.ersion 4.8.2 20130729)
Epiphnay Compiler Flags	-Wall -std=gnu99 -O3
Epiphany Optional Optimizations	-frename-registers -fno-cprop-registers -funroll-loops -ffast-math --param max-unroll-times=8
Epiphany Linker Flags	-Wall -std=gnu99 -lm -le-lib
SDK version	2015.1
SD-Card Image	ubuntu-14.04-headless-z7020-20150130.img

Table 16: Compiler and tool-chain versions and settings

Profiling on the E16G301

The E16G301 features two hardware timers (TIMER0 and TIMER1) per eCore. These timers are down-counting timers, and can be set to monitor several events shown in Table 17: No automatic reset of the timers is supported. Instead an interrupt is fired when the timers reach zero, at which the user can perform the appropriate actions.

Timer capture inputs	Measurable Mesh-Events **Access**=incoming data toward router **Wait**=incoming data toward router has to wait
• Clock cycles • Idle cycles • IALU valid instructions • FPU valid instructions • Dual issue clock cycles • Load (E1) stalls • Register dependency (RA) stalls • Local memory fetch stalls • Local memory load stalls • External fetch stalls • External load stalls • Mesh traffic monitor 0 • Mesh traffic monitor 1	• Any wait • Core wait • South wait • North wait • West wait • East wait • South-East wait • North-West wait • South access • North access • West access • East access • Core access

Table 17: E16G301 available eCore timer capture inputs for both TIMER0 and TIMER1

Using these two timers per eCore, three different approaches are possible:

Time-based sampling:

TIMER0 is set to trigger an interrupt every fixed amount of clock-cycles. When the interrupt occurs, TIMER0 is reset and a sample of TIMER1 is taken. TIMER1 monitors the desired input event, and the sample time is calculated based on the number of interrupts that have fired thus-far. This method produces evenly-spread samples in time, resulting in a predictable influence on the program flow. Unfortunately, it requires TIMER0 to be reset by software, causing the running time to drift between processors over time.

Even-count based sampling:

TIMER0 is set to capture clock cycles, without reset, allowing a maximum run-time of 7.15 seconds without a clock pre-scaler at 600MHz. TIMER1 is set to a fixed count, and captures the desired event. Whenever TIMER1 reaches zero, say after for instance 1000 occurrences of the event to capture, the current time in TIMER0 is captured and stored. The sample rate of this method is proportional to the

rate the monitored event occurs, allowing for higher accuracy at critical moments in time. However, a significant amount of interrupts can occur at random intervals depending on the event that is being monitored. This can significantly influence the program flow and performance.

Manual sampling:

The last option is to manually annotate the sample moments in code. This allows for precise control of when the sample moments will occur (for instance just before and after a function-under-test). TIMER0 is set to count clock-cycles, and TIMER1 is set to count the desired event. Both values are stored for every sample. This method was used during this research, as it is least invasive, and most accurate for profiling of floating-point performance of specific functions, however this method is not very suitable for profiling for instance network behaviour over time, as sample intervals are typically quite large. It is possible to combine this method with both time-based sampling and event-count based sampling.

Figure 47 shows an overview of all three methods:

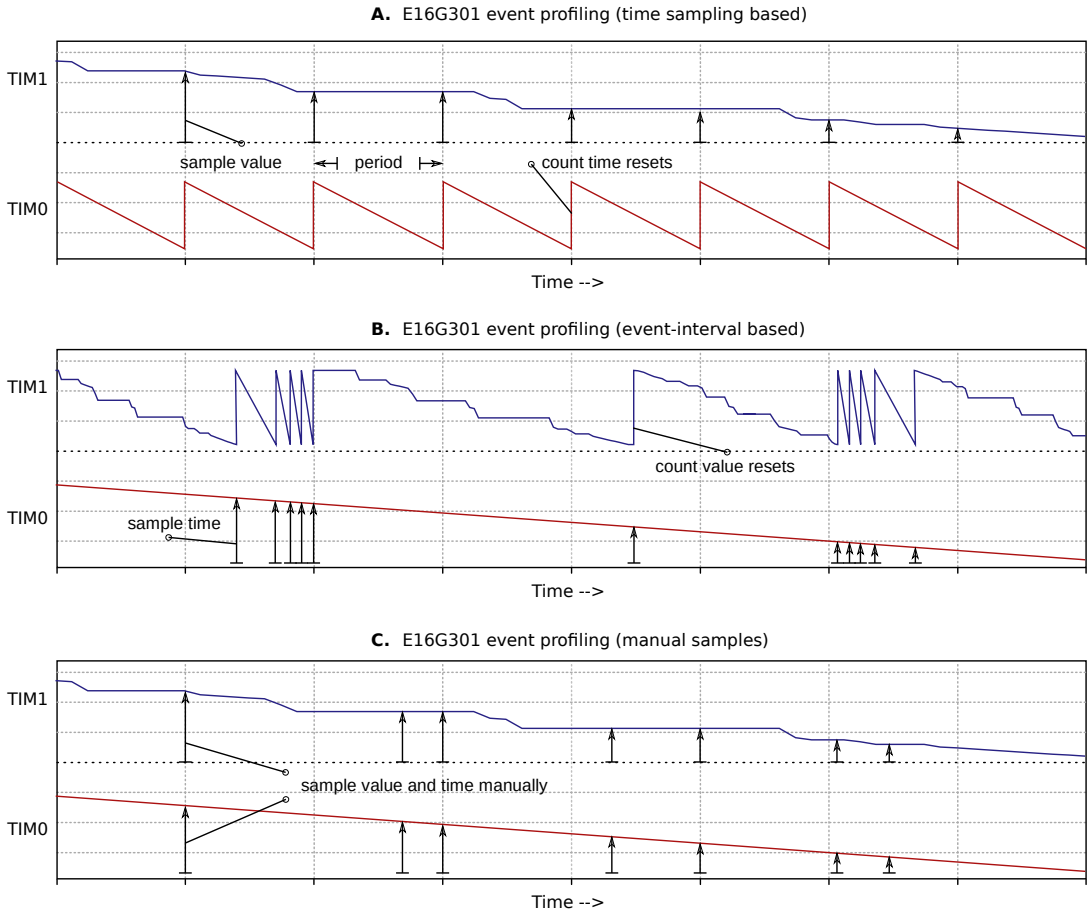


Figure 47: Three profiling methods on the E16G301 ***A:****Time-based sampling.* ***B:****Event-count based sampling.* ***C:****Manual sampling*

In all cases, only one event can be captured during a single measurement. In this research, multiple measurements were performed sequentially to obtain information about all the desired events. Most performance measurements take in the order of microseconds or milliseconds to complete, so the limited timer runtime of 7.15 seconds does not really hinder the measurements.

E. SDF3: AUTOMATED MPSOC TASK MAPPING

SDF3 ([29]) is a dataflow driven research tool, developed at the University of Eindhoven. It can do automated throughput and latency analysis on dataflow application models, as well as automated mapping of a given dataflow graph on a generic (user configurable) MPSoC model. An attempt was made to use this tool to provide a valid, automated mapping and scheduling of the front-end receiver based on the measurements done on the individual tasks in Figure 38 (p.57).

To this end a data-flow model (XML) generation script was developed that could generate multiple data-flow representations of the front-end receiver with various decompositions and amount of channels. These data-flow models where presented to SDF3 in combination with a platform model. SDF3 would then iterate through the models, finding a mapping of that model on the platform model meeting a certain throughput constraint. This throughput constraint was relaxed until a valid mapping was found.

A few issues where encountered in this process that led to the decision to do the mapping by hand rather than to use SDF3:

- **Platform Model:** SDF3 comes with a standard architecture model that is very versatile and allows for easy configuration to match many existing platforms. However, it does assume adaptive routing is used, making it difficult to model the deterministic routing scheme used by the Epiphany. The default model can be adapted but this was not done in this research.
- **Throughput Performance:** Due to the inaccuracy if the platform model, the achieved performance for the automated mapping was much worse than what we achieved manually. For instance, for a single channel, a throughput of 0,02MS/s was achieved, versus the 4.3MS/s throughput in our mapping.
- **Transparency:** SDF3 used many complex heuristics, that make the outcome very in-transparent for the user. Simple changes in the data-flow models can lead to big differences in the outcome, that are hard to predict.
- **Running Time:** For bigger problems, it can sometimes take hours for SDF3 to complete, reducing the advantage of using an automated tool.

More testing and research is needed in order to adapt SDF3 or other tools for the use of automated task mapping and design space exploration. One example use-case could be finding a suitable schedule of tasks based on the decompositions in Figure 38 (p.57). The mapping can then be done manually. Also auto generating code based on this schedule could be used to save a significant amount of development time and debugging effort.