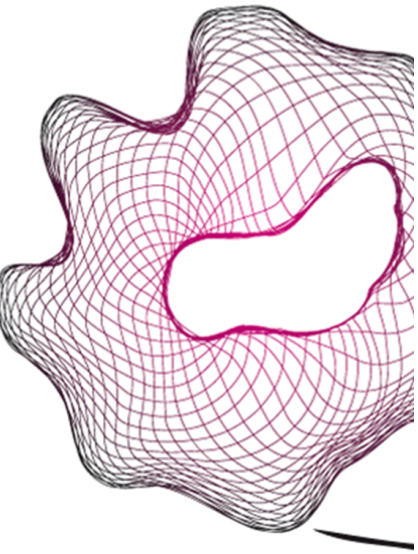
UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**



Evaluating Performance and Energy Efficiency of the Hybrid Memory Cube Technology

ing. A.B. (Arvid) van den Brink

Master Thesis
August 2017

Exam committee:
dr. ir. A.B.J. (André) Kokkeler
ir. E. (Bert) Molenkamp
ir. J. (Hans) Scholten
S.G.A. (Ghayoor) Gillani, M.Sc.

Computer Archtecture for
Embedded Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Embedded systems process fast and complex algorithms these days. Within these embedded systems, memory becomes a major part. Large (amount of bytes), small (in terms of area), fast and energy efficient memories are needed not only in battery operated devices but also in High Performance Computing systems to reduce the power consumption of the total system.

Many systems implement their algorithm in software, usually implying a sequential execution. The more complex the algorithm, the more instructions are executed and therefore the execution time and power consumption increases accordingly. Parallel execution can be used to compensate for the increase in execution time introduced by the sequential software. For parallel execution of regular, structured algorithms, hardware solutions, like an FPGA, can be used. Only the physical boundaries of FPGAs limits the amount of parallelism.

In this thesis a comparison is made between two systems. The first system is using the Hybrid Memory Cube memory architecture. The processing element in this system is an FPGA. The second system is a common of the shelf graphical card, containing GDDR5 memory with a GPU as processing unit.

The Hybrid Memory Cube memory architecture is used to give an answer to the main research question: "*How does the efficiency of the Hybrid Memory Cube compare to GDDR5 memory?*". The energy efficiency and the performance, in terms of speed, are compared to a common of the shelf graphical card. Both systems provide the user with a massively parallel architecture.

Two benchmarks are implemented to measure the performance of both systems. The first is the data transfer benchmark between the host system and the device under test and the second is the data transfer benchmark between the GPU and the GDDR5 memory (AMD Radeon HD7970) or the FPGA and the HMC memory. The benchmark results show an average speed performance gain of approximately $5.5\times$ in favour of the HMC system.

Due to defective HMC hardware, only power measurements are compared when both the graphical card and HMC system were in the *Idle* state. This resulted that the HMC system is approximately 34.75% more energy efficient than the graphical card.

To my wife, Saloewa,
who supported me
the last three years
pursuing my dreams.

Contents

List of Figures

List of Tables

Glossary

ADC Analog to Digital Converter

API Application Programming Interface

ASIC Application Specific Integrated Circuit

BFS Breadth-First Search

CAS Column Address Select

CBR $\overline{\text{CAS}}$ Before $\overline{\text{RAS}}$

CK Command Clock

CMOS Complementary Metal Oxide Semiconductor

CMV Common-mode Voltage

CMR Common-mode Rejection

CNN Convolutional Neural Network

CPU Central Processing Unit

CSP Central Signal Processor

DDR Double Data Rate SDRAM

DDR3 Double Data Rate type 3 Synchronous Dynamic Random Access Memory

DIMM Dual In-Line Memory Module

DLL Delay Locked Loop

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

DSP Digital Signal Processing

DUT Device Under Test

DVFS Dynamic Voltage/Frequency Scaling

EOL End Of Life

FLIT Flow Unit

FPGA Field Programmable Gate Array

GCN Graphics Core Next

GDDR Double Data Rate Synchronous Graphics Random Access Memory

GDDR5 Double Data Rate type 5 Synchronous Graphics Random Access Memory

GPU Graphics Processing Unit

GUPS Giga-Updates Per Second

HBM High Bandwidth Memory

HDL Hardware Description Language

HMC Hybrid Memory Cube

HPC High Performance Computing

HPCC HPC challenge

IC Integrated Circuit

I/O Input/Output

LFSR Linear Feedback Shift Register

LM Link Master

LS Link Slave

MCU Micro Controller Unit

MOSFET Metal-Oxide-Semiconductor Field-Effect Transistor

MRI Magnetic Resonance Imaging

ODT On-Die Termination

OpenCL Open Computing Language

P2P Point-To-Point

P22P Point-To-Two-Point

PC Personal Computer

PCB Printed Circuit Board

PCIe Peripheral Component Interconnect Express

PDF Probability Density Function

PLL Phase Locked Loop

POD Pseudo Open Drain

RAM Random Access Memory

RAS Row Address Select

ROR $\overline{\text{RAS}}$ Only Refresh

SDP Science Data Processor

SDRAM Synchronous Dynamic Random Access Memory

SGRAM Synchronous Graphics Random Access Memory

SKA Square Kilometre Array

SLID Source Link Identifier

SR Self Refresh

SRAM Static Random Access Memory

TSV Through Silicon Via

VHDL VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

WCK Write Clock

WE Write Enable

Chapter 1

Introduction

1.1 Context

Many embedded systems are using fast and complex algorithms to perform the designated task these days. The execution of these algorithms commonly requires a Central Processing Unit (CPU), memory and storage. Nowadays, memory becomes a major part of these embedded systems. The demand for larger memory sizes in increasingly smaller and faster devices requires a memory architecture that occupies less area, performs at higher speeds and uses less power consumption. Not only in battery operated devices the power consumption is a key feature, also in High Performance Computing (HPC) systems the total power consumption is important. For example, a smartphone user wants to use the phone for many hours without the need of recharging the smartphone or like the HPC systems used for the international Square Kilometre Array (SKA) telescope [4] by one of its partners ASTRON [5], where the energy consumption is becoming a major concern as well. This HPC system consist of the Central Signal Processor (CSP) [6] and the Science Data Processor (SDP) [7] elements and many more. These elements consume multiple megawatts of power. Reducing the power consumption in these systems is as important as in mobile, battery powered devices.

To reduce the amount of energy used by a system, a new kind of memory architecture, like the Hybrid Memory Cube (HMC) [8], can be used. Despite the reduction in area and energy usage and the increase in speed, the rest of the system still uses the same computer architecture as the general purpose systems and therefore the system is not fully optimised for the execution of the same algorithms. Introducing an FPGA, to create optimised Digital Signal Processing (DSP) blocks, can also reduce the amount of energy and time needed for the complex tasks. The FPGA is a specialised Integrated Circuit (IC) containing predefined configurable hardware resources, like *logic blocks* and *DSPs*. By configuring the FPGA, any arbitrary digital circuit can by made.

The computational power needed depends on the complexity of the executed algorithm. Using a general purpose CPU needs the algorithm to be described in software. This software is made up out of separate instructions which are usually executed in sequence. The more complex the algorithm, the more instructions needed for the execution, hence the execution time increases.

Parallel execution can be used to compensate for the increase in execution time introduced by the sequential software. For parallel execution of regular, structured algorithms hardware solutions, like an FPGA, are ideal. For parallel computations the physical boundaries of FPGAs limit the amount of

parallelism. More complex algorithms can require more resources than available in the FPGA. If a full parallel implementation does not fit in the FPGAs *area*, pipelining[1] can be used to execute parts of the algorithm sequentially over time. The trade-off between resource usage and execution time is made by the developer.

The use of an FPGA can be seen as the trade-off between area and time as an FPGA can be configured specifically for the application it is used for. In the field of radio astronomy, like in other applications, many algorithms similar to image processing are seen. Image processing algorithms are very suitable for implementation on an FPGA due to the following properties:

- The algorithms are computationally complex
- Computations can potentially be performed in parallel
- Execution time can be guaranteed due to deterministic behaviour of the FPGA

As mentioned before image processing is essential in many applications, including astrophysics, surveillance, image compression and transmission, medical imaging and astronomy, just to name a few. Images in just one dimension are called signals. In two dimensions images are in the planar field and in three dimensions volumetric images are created, like Magnetic Resonance Imaging (MRI). These *images* can be coloured (*vector-valued functions*) or in gray-scale (*single-value functions*). Many types of imperfections, like noise and blur, in the acquired data often degrade the image. Before any feature extraction and further analysis is done, the images have to be first pre-processed.

In this research on energy efficiency of the HMC architecture, image denoising will be used. The technique used is the Median filter. Because this image processing algorithm, like most image processing algorithms, is highly complex and is executed on a fast amount of data, the combination of the HMC technology and FPGAs seems a logical choice.

The configuration of an FPGA can be done with a Hardware Description Language (HDL). Due to the possible parallelism in the hardware and the lower clock frequency of the FPGA in comparison to a CPU, in combination with HMC memory, a much lower energy consumption and execution time can possibly be obtained. To describe the hardware architecture, languages like VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) and Verilog are used. The manual work required is cumbersome.

Another framework is Open Computing Language (OpenCL). OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs and other processors or hardware accelerators. OpenCL specifies a programming language (based on C99) for programming these devices and Application Programming Interfaces (APIs) to control the platform and execute programs on the compute devices. OpenCL provides a standard interface for parallel computing using task-based and data-based parallelism.

[1] A pipeline is a set of processing elements connected in series, where the output of one element is the input of the next. The elements of a pipeline are often executed in parallel or in time-sliced fashion. In that case of time-slicing, some amount of buffer storage is often inserted between the elements.

1.2 Problem statement

To evaluate the efficiency of the HMC, it is necessary to use a benchmark which can also run on other memory architectures, like GDDR5. The benchmark will provide the following metrics:

- Achievable throughput/latency (performance);
- Achievable average power consumption.

This research will attempt to answer the following questions:

- What is the average power consumption and performance of the HMC?
- How does the power consumption of the HMC compare to GDDR5 memory?
- How does the performance of the HMC compare to GDDR5 memory?
- What are the bottlenecks and (how) can this be improved?

Realising a hardware solution for mathematically complex and memory intensive algorithms has potential advantages in terms of energy and speed. To analyse these advantages, a solution on an FPGA driven Hybrid Memory Cube architecture is realised and compared to a similar OpenCL software solution on a GPU. The means to answer the research questions are summarised into the following statements:

- How to realise a feasible image denoising implementation on an FPGA and HMC?
- How to realise a feasible image denoising implementation on a GPU?
- How to realise a feasible benchmark to compare the GPU with the FPGA and HMC?
- Does the FPGA and HMC implementation have the potential to be more energy and performance efficient compared to the GPU solution?

1.3 Approach and outline

Image denoising can be solved in different ways. To realise one solution that fits on both the FPGA/HMC and the GPU this solution must first be determined. This one solution must be implemented in Hardware Description Language (HDL) and in software. CλaSH is suitable for formulating complex mathematical problems and transforming this formulation into VHDL or Verilog HDL, but OpenCL is also supported for both the HMC and GPU architectures.

Part I - Background, contains the information on the different topics used in the research. Chapter 3 describes the different memory architectures of the memory modules used. Chapter 4 introduces the basic concepts regarding memory and CPU/GPU/FPGA benchmarking.

Part II - Realisation and results, shows the realised solutions and the results found.

Part III - Conclusions and future work are given.

Part I

Background

Chapter 2

Related Work

Since the Hybrid Memory Cube is a fairly new memory technology, little has been studied on the impact on performance and energy efficiency. As an HMC I/O interface can achieve an external bandwidth up to $480\,GB/s$, using high-speed serial links, this comes at a cost. The static power of the off-chip links is largely dominating the total energy consumption of the HMC. As proposed by Ahn et al. [9] the use of dynamic power management for the off-chip links can result in an average energy consumption reduction of 51%.
Another study by Wang et al. [10] proposes to deactivate the least used HMCs and using erasure codes to compensate for the relatively long wake-up time of over $2\mu s$.

In 2014 Rosenfeld [11] defended his PhD thesis on the performance exploration of the Hybrid Memory Cube (HMC). For his research he used only simulations of the HMC architecture.

Finally, a paper by Zhu, et al. [12], discusses that GPUs are widely used to accelerate data-intensive applications. To improve the performance of data-intensive applications, higher GPU memory bandwidth is desirable. Traditional Double Data Rate Synchronous Graphics Random Access Memory (GDDR) memories achieve higher bandwidth by increasing frequency, which leads to excessive power consumption. Recently, a new memory technology called High Bandwidth Memory (HBM) based on 3D die-stacking technology has been used in the latest generation of GPUs developed by AMD, which can provide both high bandwidth and low power consumption with in-package stacked DRAM memory, offering $> 3\times$ the bandwidth per watt of GDDR5[1]. However, the capacity of integrated in-packaged stacked memory is limited (e.g. only 4GB for the state-of-the-art HBM-enabled GPU, AMD Radeon Fury X [13], [14]). In his paper, Zhu et al. implement two representative data-intensive applications, Convolutional Neural Network (CNN) and Breadth-First Search (BFS) on an HBM-enabled GPU to evaluate the improvement brought by the adoption of the HBM, and investigate techniques to fully unleash the benefits of such HBM-enabled GPU. Based on his evaluation results, Zhu et al. first propose a software pipeline to alleviate the capacity limitation of the HBM for CNN. They then designed two programming techniques to improve the utilisation of memory bandwidth for the BFS application. Experiment results demonstrate that the pipelined CNN training achieves a 1.63x speed-up on an HBM enabled GPU compared with the best high-performance GPU on the

[1] Testing conducted by AMD engineering on the AMD Radeon R9 290X GPU vs. an HBM-based device. Data obtained through isolated direct measurement of GDDR5 and HBM power delivery rails at full memory utilisation. Power efficiency calculated as GB/s of bandwidth delivered per watt of power consumed. AMD Radeon R9 290X ($10.66GB/s$ bandwidth per watt) and HBM-based device ($35 + GB/s$ bandwidth per watt), AMD FX-8350, Gigabyte GA-990FX-UD5, 8GB DDR3-1866, Windows 8.1 x64 Professional, AMD Catalyst 15.20 Beta. HBM-1

market, and the two, combined optimisation techniques for the BFS algorithm makes it at most 24.5x (9.8x and 2.5x for each technique, respectively) faster than conventional implementations.

At the time of writing this report, the Hybrid Memory Cube technology is, as far as known, only used in two applications. The first implementation is the HMC produced by Micron (formerly Pico-Computing) [15], [16], see figure 2.1. This module is used for the experiments in order to get the performance and power measurements, which are discussed in the rest of this thesis.

Figure 2.1: HMC module: AC510 board

The second product known of using HMC, 3D stacking, technology is Intel's Knights Landing products [17], [18].

Chapter 3

Memory architectures

It was predicted by computer pioneers that computer systems and programmers would want unlimited amounts of fast memory. A *memory hierarchy* is an economical solution to that desire, which takes advantage of trade-offs and locality in the cost-performance of current memory technologies. Most programs do not access all code or data uniformly, as stated by the Principle of Locality. Locality occurs in space (*spatial locality*) and in time (*temporal locality*). Moreover, for a given technology and power budget smaller hardware can be made faster, led to hierarchies based on memories of different sizes and speeds.

3.1 Principle of Locality

When executing a program on a computer, this program tends to use instructions and read or write data with addresses near or equal to those used recently by that program. The Principle of Locality, also known as the *Locality of Reference*, is the phenomenon of the same value or related storage locations being frequently accessed. There are two types of locality:

- Temporal locality: refers to the reuse of specific data and/or resources within relatively small time durations.

Figure 3.1: Temporal locality: Refer to block again

- Spatial locality: refers to the use of data elements within relatively close storage locations. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, e.g. traversing the elements in a one-dimensional array.

Figure 3.2: Spatial locality: Refer nearby block

For example, when exhibiting spatial locality of reference, a program accesses consecutive memory locations and during temporal locality of reference a program repeatedly accesses the same

memory location during a short time period. Both forms of locality occur in the following code snippet:

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

In the above code snippet, the variable `i` is referenced several times in the for loop where `i` is compared against `n`, to see if the loop is complete, and also incremented by one at the end of the loop. This shows *temporal* locality of reference in action since the CPU accesses `i` at different points in time over a short period of time.

This code snippet also exhibits *spatial* locality of reference. The loop itself adds the elements of array `a` to variable `sum`. Assuming C++ stores elements of array `a` into consecutive memory locations, then on each iteration the CPU accesses adjacent memory locations.

3.2 RAM memory cell

SRAM is a type of semiconductor memory that uses *flip-flops* to store a single bit. SRAM exhibits data remanence (keeping its state after writing or reading from the cell) [19], but it is still volatile. Data is eventually lost when the memory is powered off.

The term *static* differentiates SRAM from DRAM which must be periodically refreshed. SRAM is faster but more expensive than DRAM, hence it is commonly used for CPU cache while DRAM is typically used for main memory. The advantages of SRAM over DRAM are lower power consumption, simplicity (no refresh circuitry is needed) and reliability. There are also some disadvantages: a higher price and lower capacity (amount of bits). The latter disadvantage is due to the design of an SRAM cell.

A typical SRAM cell is made up of six Metal-Oxide-Semiconductor Field-Effect Transistors (MOSFETs). Each bit in an SRAM (see figure 3.3) is stored on four transistors (M1, M2, M3 and M4). This cell has two stable states which are used to denote 0 and 1. Two additional transistors (M5 and M6) are used to control the access to that cell during read and write operations.

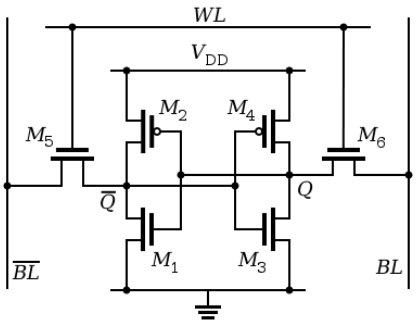


Figure 3.3: Six transistor SRAM cell

A four transistor (4T) SRAM (see figure 3.4) is quite common in standalone devices, which is implemented in special processes with an extra layer of poly-silicon, allowing for very high-resistance pull-up resistors. The main disadvantage of using 4T SRAM is the increased static power due to the constant current flow through one of the pull-down transistors.

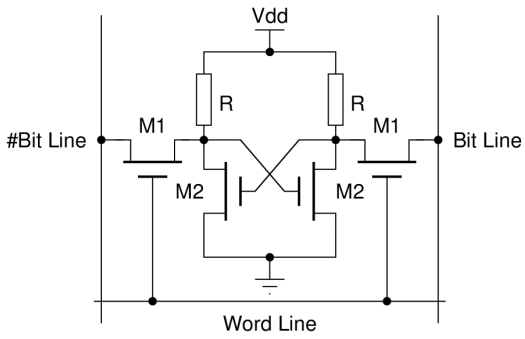


Figure 3.4: Four transistor SRAM cell

Generally, the fewer transistors needed per cell, the smaller each cell. Since the cost of processing a silicon wafer is relatively fixed, therefore using smaller cells and so packing more bits on a single wafer reduces the cost per bit. In figure 3.5a the layout of a 6T cell with dimensions $5 \times P_{Metal}$ by $2 \times P_{Metal}$ is shown. A 4T cell, as can be seen in figure 3.5b, has only a dimension of $5 \times P_{Metal}$ by $1.5 \times P_{Metal}$. P_{Metal} denote the Metal Pitch used by the manufacturing process. The pitch is the centre to centre distance between the metals, having minimal width and minimal spacing.

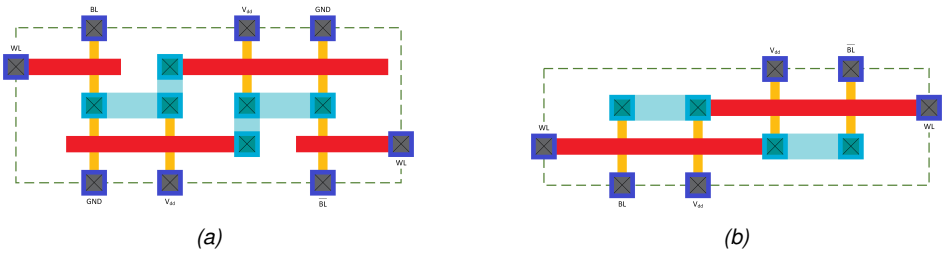


Figure 3.5: SRAM cell layout: (a) A six transistor cell; (b) A four transistor cell

To access the memory cell (see figure 3.3) the *word line* (WL) gives access to transistors M5 and M6 which, in turn, control whether the cell (cross-coupled inverters) should be connected to the *bit lines* (BL). These bit lines are used for both write and read operations. Although it is not strictly necessary to have two bit lines, the two signals (BL and $\overline{BL}$) are typically provided in order to improve noise margins.

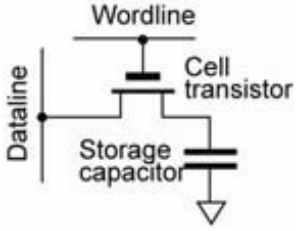


Figure 3.6: One transistor, one capacitor DRAM cell

Opposed the the *static* SRAM, there is the *dynamic* DRAM cell (see figure 3.6). In this type of memory cell the data is stored in a separate capacitor within the IC. A charged capacitor denotes a 1 and discharged denotes a 0. However, a non-conducting transistor will always leak a small amount, discharging the capacitor, and the information in the memory cell eventually fades unless the capacitors charge is refreshed periodically, hence *Dynamic* in the name. The DRAM cell layout is even smaller as can be seen in figure 3.7

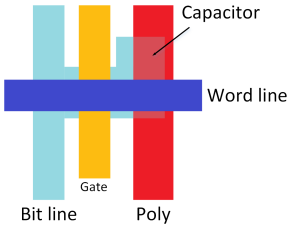


Figure 3.7: One transistor, one capacitor cell

The structural simplicity is DRAMs advantage. Compared to the four or even six transistors required in SRAM, just one transistor and one capacitor is required per bit in DRAM. This allows for very high densities, billions of these 1T cells can fit on a single chip. On the other hand, due to its *dynamic* nature, DRAM consumes relatively large amounts of power.

3.3 Principles of operation

Both types of RAM architectures have three main operations:

SRAM	DRAM
Standby	No Operation
Reading	
Writing	

Table 3.1: RAM main operations

Due to the volatile nature of the DRAM architecture there is a fourth main operation, called *Refresh*

An SRAM cell has three different states: standby (the circuit is idle), reading (the data has been requested) or writing (updating the contents). SRAM operating in read mode and write modes should have *readability* and *write stability*, respectively. Assuming a six transistor implementation, the three different operations work as follows:

3.3.1 SRAM - Standby

If the word line is not asserted, the transistors M5 and M6 disconnect the cell ($M1$, $M2$, $M3$ and $M4$) from the bit lines. The two cross-coupled inverters in the cell will continue to reinforce each other as long as they are connected to the supply V_{dd}.

3.3.2 SRAM - Reading

In theory, reading only requires asserting the word line and *reading* the SRAM cell state by a single access transistor and bit line, e.g. $M6/BL$ and $M5/\overline{BL}$. Nevertheless, bit lines are relatively long and have large parasitic capacitance. To speed up reading, a more complex process is used in practice:

1. Pre-charge both bit lines BL and $\overline{BL}$, that is, driving both lines to a threshold voltage midrange between a logic 1 and 0 by an external circuitry (not shown in figure 3.3).
2. Assert the word line WL to enable both transistors M5 and M6. This causes the BL voltage to either slightly rise (nMOS[1] transistor M3 is *OFF* and pMOS[2] transistor M4 is *ON*) or drop (M3 if *ON* and M4 *OFF*). Note that if BL rises, $\overline{BL}$ drops and vice versa.
3. A sense amplifier will sense the voltage difference between BL and $\overline{BL}$ to determine which line has the higher voltage and thus which logic value (0 or 1) is stored. A more sensitive sense amplifier speeds up the read operation.

3.3.3 SRAM - Writing

To write to SRAM the following two steps are needed:

1. Apply the value to be written to the bit lines. If writing a 1, $BL = 1$ and $\overline{BL} = 0$.
2. Assert the word line WL to latch in the value to be stored.

This operation works, because the bit line input-drivers are designed to be much stronger than the relatively weak transistors in the cell itself so they can easily override the previous state of the cross-coupled inverters. In practice, the nMOS transistors M5 and M6 have to be stronger than either bottom nMOS (M1/M3) or top pMOS (M2/M4) transistors. This is easily obtained as pMOS transistors are much weaker than nMOS when same sized. Consequently when one transistor pair (e.g. M3/M4) is only slightly overridden by the write process, the opposite transistors pair (M1/M2) gate voltage is also changed. This means that the M1 and M2 transistors can be easier overridden, and so on. Thus, cross-coupled inverters magnify the writing process.

3.3.4 DRAM - Refresh

Due to the charge leaking away out of the capacitor over time, this charge on the individual cells must be refreshed periodically. The frequency with which this refresh must occur depends on the silicon technology used to manufacture the memory chip and the design of the memory cell itself.

Each cell must be accessed and restored during a refresh interval. In most cases, refresh cycles involve restoring the charge along an entire row. Over the period of the entire interval, every cell in a row is accessed and restored. At the end of the interval, this process begins again.

Memory designers have a lot of freedom in designing and implementing memory refresh. One choice is to fit the refresh cycles between normal read and write cycles, another is to run refresh cycles on a fixed schedule, forcing the system to queue read/write operations when they conflict with the refresh requirements.

[1] n-type MOSFET. The channel in the MOSFET contains electrons
[2] p-type MOSFET. The channel in the MOSFET contains holes

Three common refresh options are briefly described below:

- **$\overline{\text{RAS}}$ Only Refresh (ROR)**[3]
  Normally, DRAMs are refreshed one row at a time. The refresh cycles are distributed across the refresh interval so that all rows are refreshed within the required time period. Refreshing one row of DRAM cells using ROR, occurs in the following steps:
  - The address of the row to be refreshed is applied at the address pins
  - $\overline{\text{RAS}}$ is switched from *High* to *Low*. $\overline{\text{CAS}}$[4] must remain *High*
  - At the end of the, by specification, required amount of time, $\overline{\text{RAS}}$ is switch to *High*
- **$\overline{\text{CAS}}$ Before $\overline{\text{RAS}}$ (CBR)**
  Like $\overline{\text{RAS}}$ Only Refresh, CBR refreshes one row at a time. Refreshing a row using $\overline{\text{CAS}}$ Before $\overline{\text{RAS}}$, occur in the following steps:
  - $\overline{\text{CAS}}$ is switched from *High* to *Low*
  - $\overline{\text{WE}}$[5] is be switched to *High* (Read).
  - After a specified required amount of time, $\overline{\text{RAS}}$ is switch to *Low*
  - An internal counter determines the row to be refreshed
  - After a specified required amount of time, $\overline{\text{CAS}}$ is switch to *High*
  - After a specified required amount of time, $\overline{\text{RAS}}$ is switch to *High*

  The main difference between CBR and ROR is the way for keeping track of the row address, respectively an internal counter or externally supplied.
- **Self Refresh (SR)**
  Also known as **Sleep Mode** or **Auto Refresh**. SR is a unique method of refresh. It uses an on-chip oscillator to determine the refresh rate and, like the CBR method, an internal counter to keep track of the row address. This method is frequently used for battery-powered mobile applications or applications that uses a battery for backup power.
  The timing required to initiate SR is a CBR cycle with $\overline{\text{RAS}}$ active for a minimum amount of time as specified by the manufacturer. The length of time that a device can be left in *sleep mode* is limited by the power source used. To exit, $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ are asserted *High*.

3.3.5 DRAM - Reading

To read from DRAM the following eight steps are needed:

1. Disconnect the sense amplifiers
2. Pre-charge the bit lines (differential pair) to exactly equal voltages that are midrange between logical *High* and *Low*. (E.g. 0.5V in the case of '0'$= 0$V and '1'$= 1$V)
3. Turn off the pre-charge circuitry. The parasitic capacitance of the "long" bit lines will maintain the charge for a brief moment
4. Assert a logic 1 at the word line WL of the desired row. This causes the transistor to conduct, enabling the transfer of charge to or from the capacitor. Since the capacitance of the bit line is typically much larger than the capacitance of the capacitor, the voltage on the bit line will slightly decrease or increase. (E.g. 0.45V$=$'0' or 0.55V$=$'1'). As the other bit line will stay at 0.5V there is a small difference between the two bit lines

[3] RAS: Row Address Select
[4] CAS: Column Address Select
[5] WE: Write Enable

5. Reconnect the sense amplifiers to the bit lines. Due to the positive feedback from the cross-connected inverters in the sense amplifiers, one of the bit lines in the pair will be at the lowest voltage possible and the other will be at the maximum high voltage. At this point the row is *open* (the data is available)
6. All cells in the *open* row are now sensed simultaneously, and the sense amplifier outputs are latched. A column address selects which latch bit to connect to the external data bus. Reading different columns in the same (*open*) row can be performed without a row opening delay because, for the open row, all data has already been sensed and latched
7. While reading columns in an *open* row is occurring, current is flowing back up the bit-lines from the output of the sense amplifiers and recharging the cells. This reinforces ("*refreshes*") the charge in the cells by increasing the voltage in the capacitor (if it was charged to begin with), or by keeping it discharged (if it was empty).
   Note that, due to the length of the bit lines, there is a fairly long propagation delay for the charge to be transferred back to the cells capacitor. This takes significant time past the end of sense amplification and thus overlaps with one or more column reads
8. When done with reading all the columns in the current *open* row, the word line is switched *Off* to disconnect the cell capacitors (the row is "*closed*") from the bit lines. The sense amplifier is switched *Off*, and the bit lines are pre-charged again (Next read starts from item 3).

3.3.6 DRAM - Writing

To store data, a row is *opened* and a given column sense amplifier is temporarily forced to the desired *Low* or *High* voltage, thus causing the bit line to discharge or charge the cells capacitor to the desired value. Due to the sense amplifiers positive feedback configuration, it will hold a bit line at a stable voltage even after the forcing voltage is removed. During a write to a particular cell, all the columns in that row are sensed simultaneously (just as during reading), so although only a single columns cell capacitor charge is changed, the entire row is *refreshed*.

3.4 Dual In-Line Memory Module

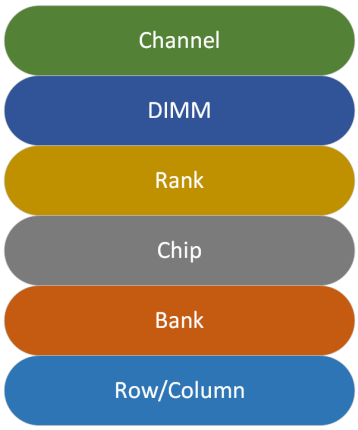


Figure 3.8: Dual In-Line Memory Module memory subsystem organisation

Nowadays, CPU architectures have integrated memory controllers. The controller connects to the top of the memory subsystem through a *channel* (see figure 3.8). On the other end of the *channel*

are one or more DIMMs (see figure 3.9). The DIMM contains the actual DRAM chips that provide 4 or 8 bits of data per chip.

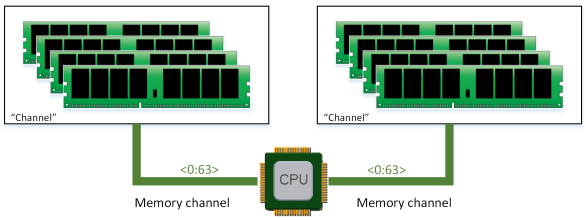


Figure 3.9: DIMM Channels

Current CPUs support triple or even quadruple channels. These multiple, independent channels increase data transfer rates due to the concurrent access of multiple DIMMs. Due to interleaving, latency is reduced when operating in triple-channel or in quad-channel mode. The memory controller distributes the data amongst the DIMMs in an alternating pattern, allowing the memory controller to access each DIMM for smaller bits of data instead of accessing a single DIMM for the entire chunk of data. This provides the memory controller more bandwidth for accessing the same amount of data across channels instead of traversing a single channel when it stores all data in one DIMM.

The Dual Inline of the DIMM refers to the DRAM chips on both sides of the module. The ”group” of chips on one side of the DIMMis called a *Rank* (see figure 3.10). Both *Ranks* on the DIMM can be accessed simultaneously by the memory controller. Within a single memory cycle 64 bits of data is accessed. These 64 bits may come from the 8 or 16 DRAM chips, depending on the data width of a single chip (see figure 3.11).

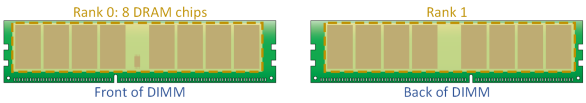


Figure 3.10: DIMM Ranks

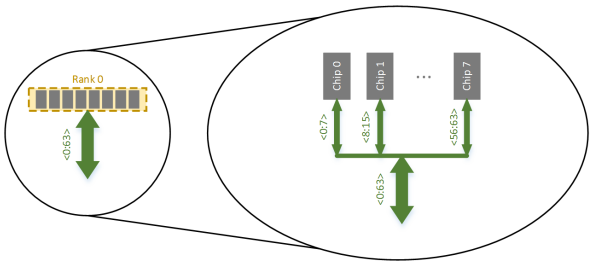


Figure 3.11: DIMM Rank breakdown

DIMMs come in three rank configurations: single-rank, dual-rank or quad-rank configuration. Ranks are denoted as (xR). Together the DRAM chips grouped into a rank contain 64 bit of data. If a DIMM contains DRAM chips on just one side of the PCB, containing a single 64-bit chunk of data, it is referred to as a single-rank (1R) module. A dual rank (2R) module contains at least two 64 bit chunks of data, one chunk on each side of the PCB. Quad ranked DIMMs (4R) contains four 64 bit chunks,

two chunks on each side. To increase capacity, combine the ranks with the largest DRAM chips. A quad-ranked DIMM with $4Gb$ chips equals $16GB$ DIMM ($^{4Gb \times 8 \text{ chips} \times 4 \text{ ranks}}/_{8 \text{ bits}}$).

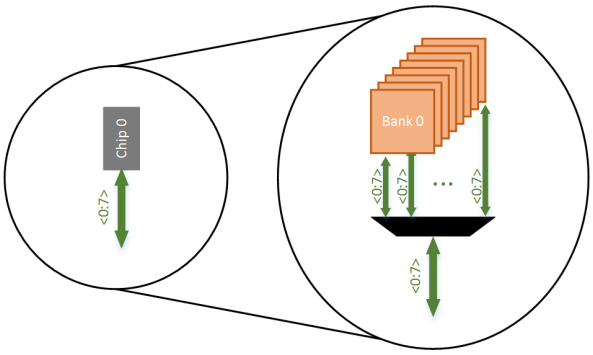


Figure 3.12: DIMM Chip

Finally, the DRAM chip is made up of several *banks* (see figure 3.12). These banks are independent memory arrays which are organised in rows and columns (see figure 3.13).

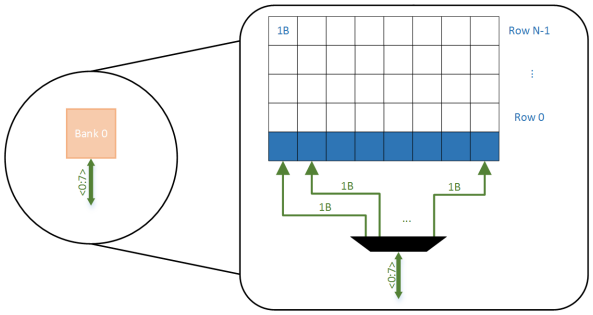


Figure 3.13: DIMM Bank Rows and Columns

For example, a DRAM chip with 13 address bits for *Row* selection, 10 address bits for *Column* selection, 8 *Banks* and 8 bits per addressable location, this chip has a total density of $2^{Rows} \cdot 2^{Cols} \cdot Banks \cdot Bits_{Addressable} = 2^{13} \cdot 2^{10} \cdot 8 \cdot 8 = 512$Mbit ($64$M x 8 bits).

3.5 Double Data Rate type 5 Synchronous Graphics Random Access Memory

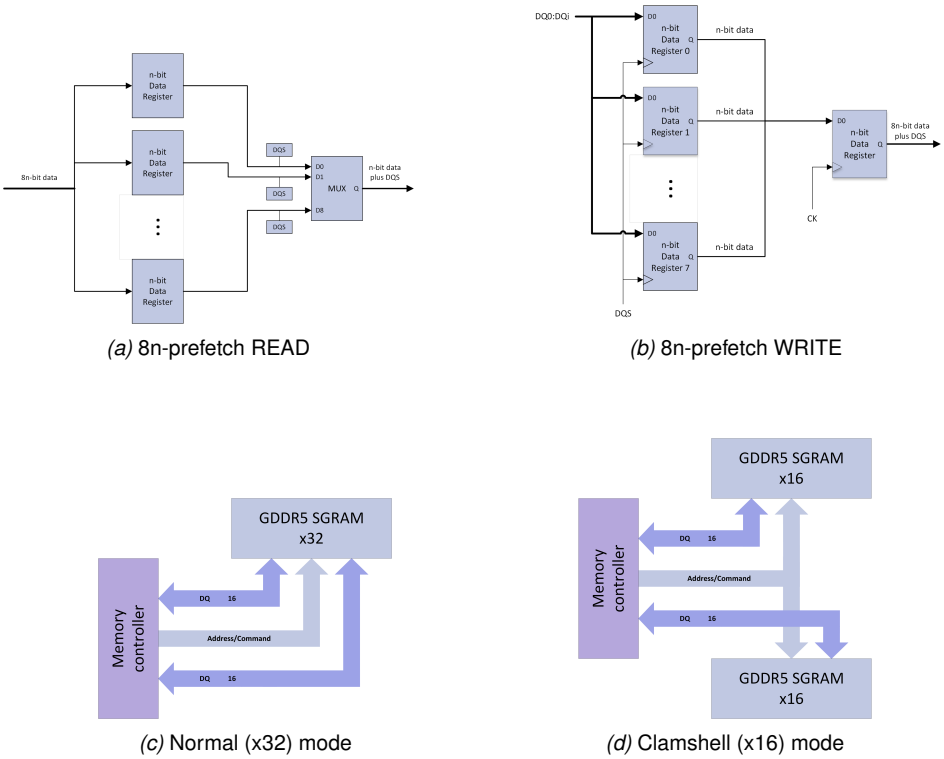
GDDR5 is the most commonly used type of Synchronous Graphics Random Access Memory (SGRAM) at the moment of writing this thesis. This type of memory has a high *bandwidth* (Double Data Rate SDRAM (DDR)) interface for use in HPC and graphics cards.

This SGRAM is a specialised form of Synchronous Dynamic Random Access Memory (SDRAM), based on DDR3 SDRAM. Functions like *bit masking*, i.e. writing a specified set of bits without affecting other bits of the same address, and *block write*, i.e. filling a block of memory with one single value. Although SGRAM is single ported, it can open two memory pages at once, simulating the dual ported nature of other video RAM technology.

GDDR5 uses a DDR3 interface and an 8n-prefetch architecture (see figures 3.14a and 3.14b) to achieve high performance operations. The prefetch buffer depth (8n) can also be thought of as the ratio between the core memory frequency and the Input/Output (I/O) frequency. In an 8n-prefetch architecture, the I/Os will operate $8\times$ faster than the memory core (each memory access results in a burst of 8 datawords on the I/Os). Thus a $200MHz$ memory core is combined with I/Os that each operate eight times faster (1600 megabits per second). If the memory has 16 I/Os, the total read bandwidth would be $200MHz \times 8datawords/access \times 16I/Os = 25.6Gbit/s$, or $3.2GB/s$. At Power-up, the device is configured in *x32 mode* or in *x16 clamshell mode*, where the 32-bit I/O, instead of being connected to one IC, is split between two ICs (one on each side of the PCB), allowing for a doubling of the memory capacity.

Just by adding additional DIMMs to the memory channels is the traditional way of increasing memory density in PC and server applications. However, this dual-rank configuration can lead to performance degradation resulting from the dual-load signal topology (The databus is share by both ranks). GDDR5 uses a single-loaded or Point-To-Point (P2P) data bus for the best performance.

GDDR5 devices are always directly soldered on the PCB and are not mounted on a DIMM. In *x16 mode*, the data bus is split into two 16-bit wide buses that are routed separately to each device (see figure 3.14d). The *Address* and *Command* pins are shared between the two devices to preserve the total I/O pin count at the controller. However, this Point-To-Two-Point (P22P) topology does not decrease system performance because of the lower data rates of the address or command bus.



(a) 8n-prefetch READ

(b) 8n-prefetch WRITE

(c) Normal (x32) mode

(d) Clamshell (x16) mode

Figure 3.14: Double Data Rate type 5 Synchronous Graphics Random Access Memory (GDDR5)

GDDR5 operates with two different clock types. A differential Command Clock (CK) as a reference for address and command inputs, and a forwarded differential Write Clock (WCK) as a reference for

data reads and writes, that runs at twice the CK frequency. A Delay Locked Loop (DLL) circuit is driven from the clock inputs and output timing for read operations is synchronised to the input clock. Being more precise, the GDDR5 SGRAM uses a total of three clocks:

1. Two write clocks associated with two bytes: WCK01 and WCK23
2. One Command Clock (CK)

Over the GDDR5 interface 64-bits of data (two 32-bit wide data words) per WCK can be transferred. Corresponding to the 8n-prefetch, a single write or read access consists of a 256-bit wide two CK clock cycle data transfer at the internal memory core and eight corresponding 32-bit wide one-half WCK clock cycle data transfers at the I/O pins.

Taking a GDDR5 with $5\,Gbit/s$ data rate per pin as an example, the CK runs with $1.25\,GHz$ and both WCK clocks at $2.5\,GHz$. The CK and WCKs are phase aligned during the initialisation and training sequence. This alignment allows read and write access with minimum latency.

3.6 Hybrid Memory Cube

As written in [20] the Hybrid Memory Cube combines several stacked DRAM dies on top of a Complementary Metal Oxide Semiconductor (CMOS) logic layer forming a so called **cube**. The combination of both DRAM technology and CMOS technology dies makes this a **hybrid** chip, hence the name Hybrid Memory Cube. The dies in the 3D stack are connected by means of a dense interconnect mesh of Through Silicon Vias (TSVs), which are metal connections extending vertically through the entire stack (see figures 3.15 and 3.16).

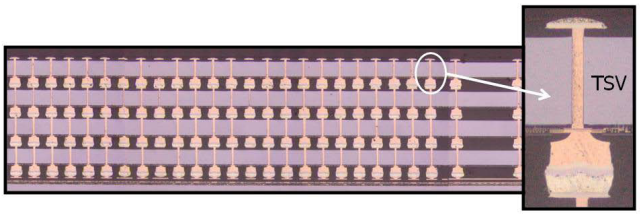


Figure 3.15: Cross Sectional Photo of HMC Die Stack Including TSV Detail (Inset) [1]

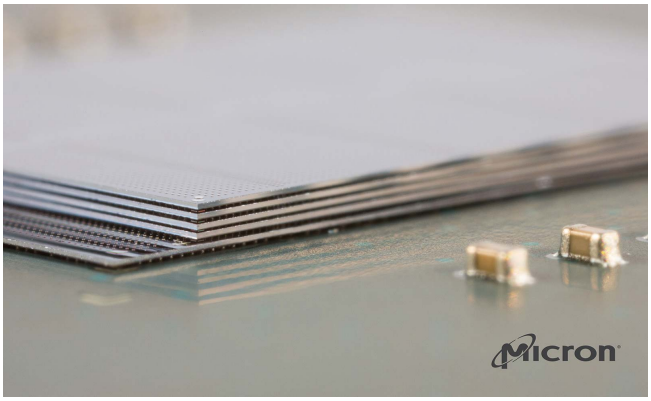


Figure 3.16: HMC Layers [2]

Unlike conventional DDR3 DIMM which has the electrical connection through the pressure of the pins in the connector, the TSVs form a permanent connection between all the layers in the stack. The TSV connections provide a very short (less then a mm up to a few mm) with less capacitance than the long PCB trace buses which can extent to many cm, hence data can be transmitted at a reasonable high data rate through the HMC stack without the use of power hungry and expensive I/O drivers [21].

Within each HMC [3], memory is vertically organised. A partition of each memory die is combined into a vault (see figure 3.17). Each vault is operationally and functionally independent. The base of a vault contains a vault controller located in the CMOS die. The role of the vault controller is like a traditional memory controller in that it sends DRAM commands to the memory partitions in the vault and keeps track of the memory timing constraints. The communication is through the TSVs. A vault is more or less equivalent to a conventional DDR3 channel. However, unlike traditional DDR3 memory, the TSV connections are much shorter than the conventional bus traces on a motherboard and therefore have much better electrical properties. An illustration of the architecture can be seen in figure 3.17.

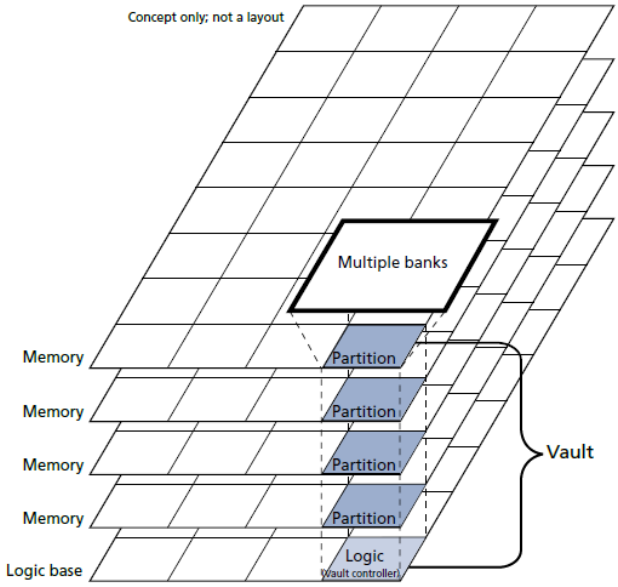


Figure 3.17: Example HMC Organisation [3]

The vault controller, by definition, may have a queue to buffer references inside that vaults memory. The execution of the references within the queue may be based on need rather than the order of arrival. Therefore the response from the vault to the external serial I/O links will be out of order. When no queue is implemented and two successive packets have to be executed on the same bank, the vault controller must wait for the bank to finish its operation, before the next packet can be executed, potentially blocking packet executions to other banks inside that vault. The queue can potentially optimise the memory bus usage.
Requests from a single external serial link to the same vault/bank address will be executed in order of arrival. Requests from different external serial links to the same vault/bank address are not guaranteed to be executed in order. Therefore the requests must be managed by the host controller (e.g. an FPGA or CPU).

The functions managed by the logic base of the HMC are:

- All HMC I/O, implemented as multiple serialised, full duplex links
- Memory control for each vault; Data routing and buffering between I/O links and vaults
- Consolidated functions removed from the memory die to the controller
- Mode and configuration registers
- BIST for the memory and logic layer
- Test access port compliant to JTAG IEEE 1149.1-2001, 1149.6
- Some spare resources enabling field recovery from some internal hard faults.

A block diagram example for an implementation of a 4-link HMC configuration is shown in figure 3.18.

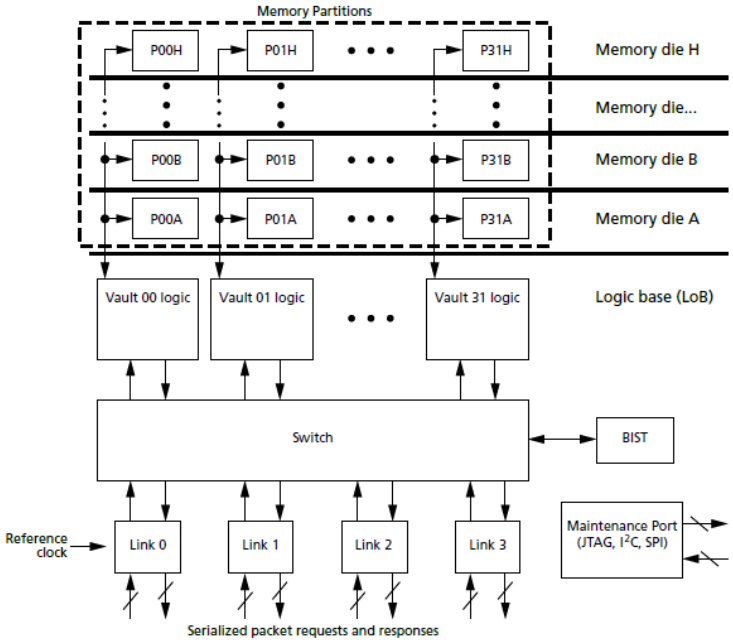


Figure 3.18: HMC Block Diagram Example Implementation [3]

Commands and data are transmitted in both directions across the link using a packet based protocol where the packets consist of 128-bit Flow Units (FLITs). These FLITs are serialised, transmitted across the physical lanes of the link, then re-assembled at the receiving end of the link. Three conceptual layers handle packet transfers:

- The physical layer handles serialisation, transmission, and de-serialisation
- The link layer provides the low-level handling of the packets at each end of the link.
- The transaction layer provides the definition of the packets, the fields within the packets, and the packet verification and retry functions of the link.

Two logical blocks exist within the link layer and transaction layer (see figure 3.19):

- The Link Master (LM), is the logical source of the link where the packets are generated and the transmission of the FLITs is initiated.
- The Link Slave (LS), is the logical destination of the link where the FLITs of the packets are received, parsed, evaluated, and then forwarded internally.

The nomenclature below is used throughout this report to distinguish the direction of transmission between devices on opposite ends of a link. These terms are applicable to both host-to-cube and cube-to-cube configurations.

Requester: Represents either a host processor or an HMC link configured as a pass-through link. A requester transmits packets downstream to the responder.

Responder: Represents an HMC link configured as a host link (See figure 3.20 through figure 3.23). A responder transmits packets upstream to the requester.

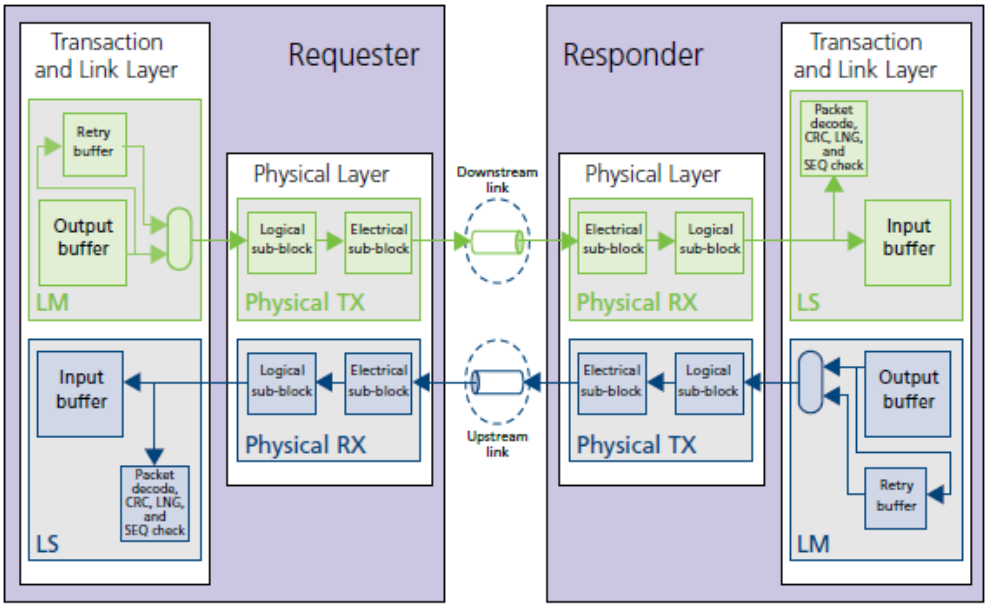


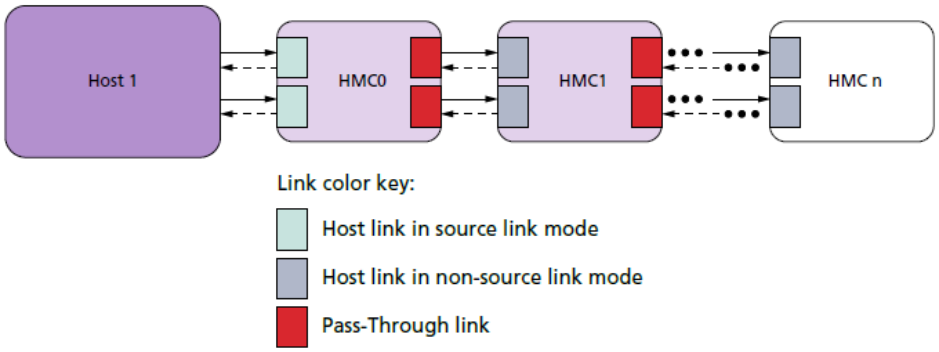
Figure 3.19: Link Data Transmission Example Implementation [3]

Multiple HMC devices may be chained together to increase the total memory capacity available to a host. A network of up to eight HMC devices and 4 host source links is supported, as will be explained in the next paragraphs. Each HMC in the network is identified through the value in its CUB field, located within the request packet header. The host processor must load routing configuration information into each HMC. This routing information enables each HMC to use the CUB field to route request packets to their destination.

Each HMC link in the cube network is configured as either a **host link** or a **pass-through link**, depending upon its position within the topology. See figure 3.20 through figure 3.23 for illustrations.

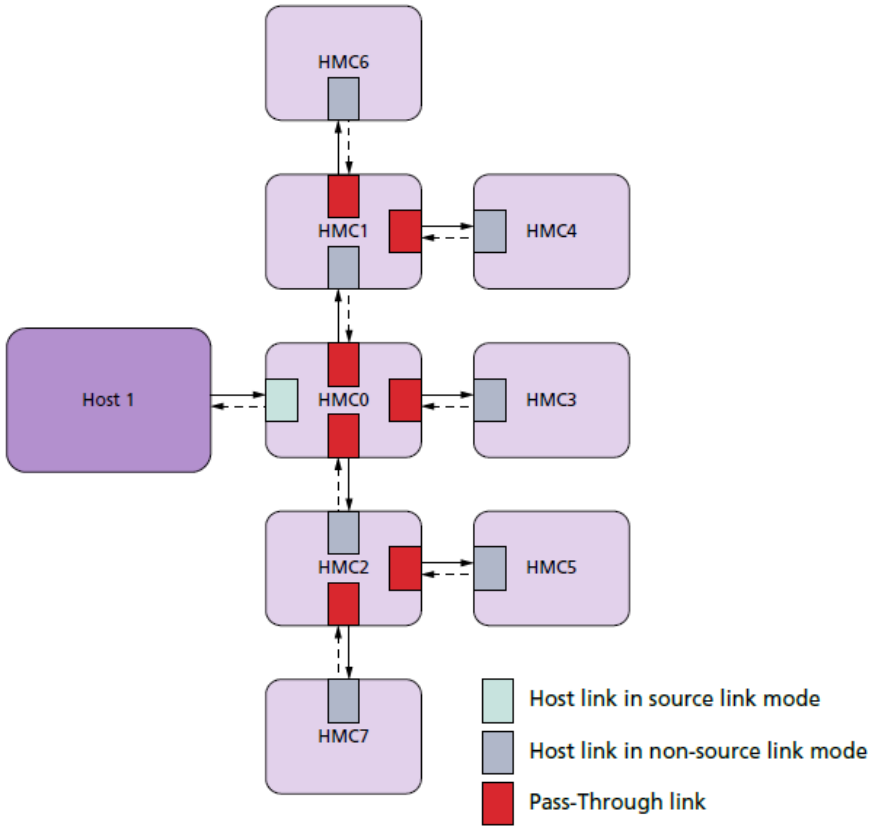
A **host link** uses its Link Slave to receive request packets and its Link Master to transmit response packets. After receiving a request packet, the host link will either propagate the packet to its own internal vault destination (if the value in the CUB field matches its programmed cube ID) or forward it towards its destination in another HMC via a link configured as a pass-through link. In the case of a malformed request packet whereby the CUB field of the packet does not indicate an existing CUBE ID number in the chain, the request will not be executed, and a response will be returned (if not posted) indicating an error.

A **pass-through link** uses its Link Master to transmit the request packet towards its destination cube, and its Link Slave to receive response packets destined for the host processor.



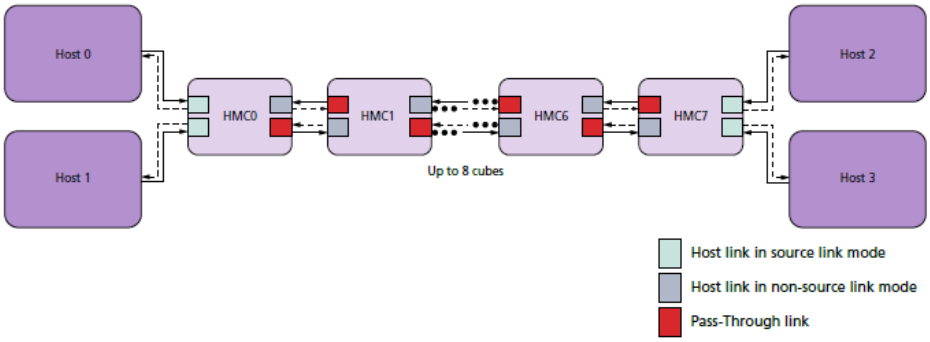
Notes: 1. For this topology, n ≤7.
2. The solid arrow indicates the command request flow; the dashed arrow indicates the response flow.

Figure 3.20: Example of a Chained Topology [3]



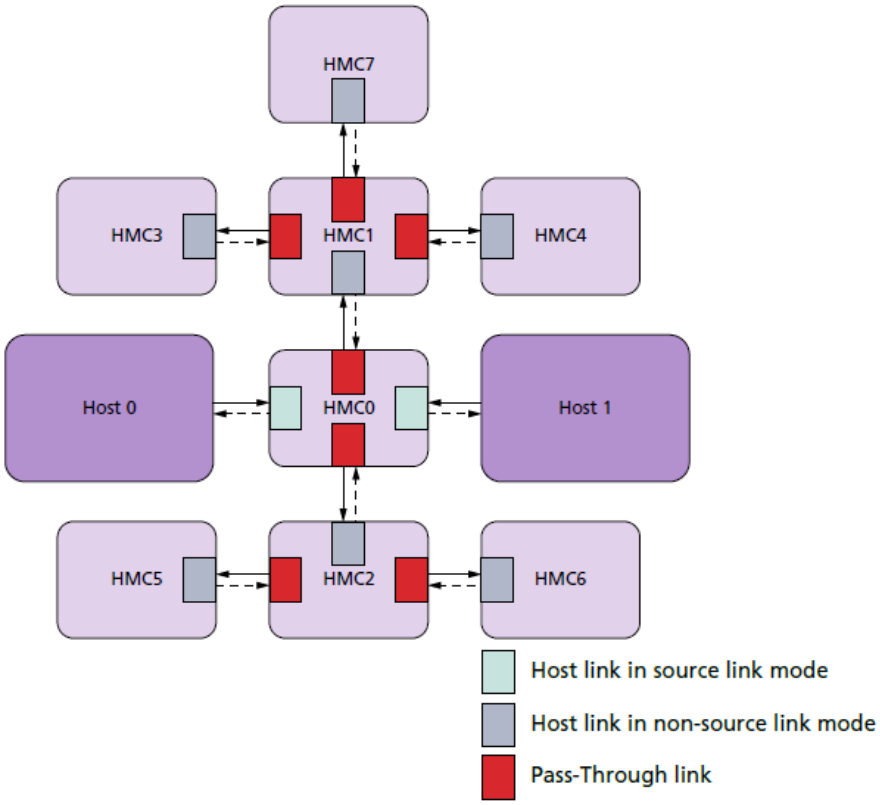
Note: 1. The solid arrow indicates the command request flow; the dashed arrow indicates the response flow.

Figure 3.21: Example of a Star Topology [3]



Note: 1. The solid arrow indicates the command request flow; the dashed arrow indicates the response flow.

Figure 3.22: Example of a Multi-Host Topology [3]



Note: 1. The solid arrow indicates the command request flow; the dashed arrow indicates the response flow.

Figure 3.23: Example of a Two-Host Expanded Star Topology [3]

An HMC link connected directly to the host processor must be configured as a host link in source mode. The Link Slave of the host link in source mode has the responsibility to generate and insert a unique value into the Source Link Identifier (SLID) field within the tail of each request packet. The unique SLID value is used to identify the source link for response routing. The SLID value does not serve any function within the request packet other than to traverse the cube network to its destination vault where it is then inserted into the header of the corresponding response packet. The host processor must load routing configuration information into each HMC. This routing information enables each HMC to use the SLID value to route response packets to their destination. Only a host link in source mode will generate an SLID for each request packet. On the opposite side of a pass-through link is a host link that is not in source mode. This host link operates with the same characteristics as the host link in source mode except that it does not generate and insert a new value into the SLID field within a request packet. All LSs in pass-through mode use the SLID value generated by the host link in source mode for response routing purposes only. The SLID fields within the request packet tail and the response packet header are considered Don't Care" by the host processor. See figure 3.20 through figure 3.23 for illustrations for supported multi-cube topologies.

3.6.1 HMC Bandwidth and Parallelism

As mentioned in the previous section, a high bandwidth connection within each vault is available by using the TSVs to interconnect the 3D stack of dies. The combination of the number of TSVs (the density per cube can be in the thousands) and the high frequency at which data can be transferred provides a high bandwidth.

Because of the number of independent vaults in an HMC, each build up out of one or more banks (as in DDR3 systems), a high level of parallelism inside the HMC is achieved. Since each vault is roughly equivalent to a DDR3 channel and with 16 or more vaults per HMC, a single HMC can support an order of magnitude more parallelism within a single package. Furthermore, by stacking more dies inside a device, a greater number of banks per package can be achieved, which in turn is beneficial to parallelism.

The overall build up of the HMC, depending on the cubes configuration, can deliver an aggregate bandwidth of up to $480GB/s$ (see figure 3.24).

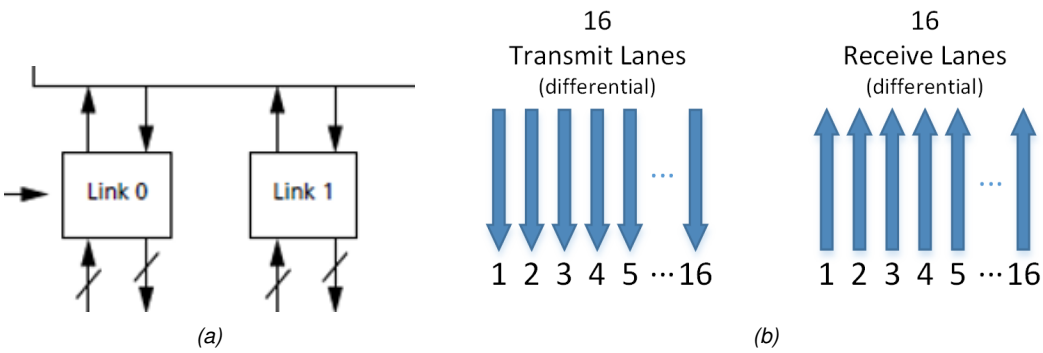


Figure 3.24: HMC implementation: (a) 2 Links; (b) 32 Lanes (Full width link)

The available total bandwidth on an 8 Links, Full width HMC implementation is calculated as follows:

- 15 Gb/s per Lane
- 32 Lanes = 4 Bytes
- 8 Links / Cube = 480 GB/s / Cube

For the AC-510 [16] UltraScale-based SuperProcessor with HMC, providing 2 half width links, the total available bandwidth is equal to $15Gb/s \times 2 \times 2 = 60GB/s$

3.6.2 GDDR5 versus HMC

To reduce the power consumption of GDDR5 many techniques have been employed. A POD signalling scheme, combined with an On-Die Termination (ODT) resistor [22] will only consume static power when driving *LOW* as can be seen in figure 3.25.

Lowering the supply voltage ($\approx 1.5V$) of the memory, Dynamic Voltage/Frequency Scaling (DVFS) and the usage of independent ODT strength control of the command, address and data lines, are some of the other techniques used in GDDR5 memory. The DVFS [23] technique reduces power by adapting the voltage and/or the frequency of the memory interface while satisfying the throughput requirements of the application. However, this reduction in power results in the degradation of the throughput.

Using all these techniques, to get a maximum throughput or bandwidth of $6.0Gbps$ a relatively high clock frequency is needed. For example, the memory clock of the AMD Radeon(tm) HD 7970 GHz edition GPU is equal to $1500MHz$.

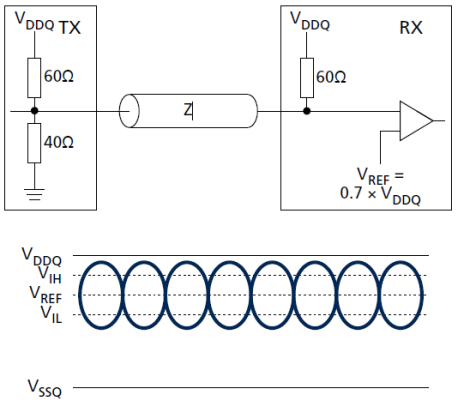


Figure 3.25: GDDR5 - Pseudo Open Drain (POD)

The architecture of the HMC, in contrast to GDDR5, uses some different techniques to reduce the power consumption. The supply voltage is reduced even more to only $1.2V$, and the clock frequency is reduced to only $125MHz$, which is 12 times lower then that of the GDDR5 memory. To keep the throughput equal (or even greater) to that of GDDR5 the HMC package uses multiple high speed links to transfer data to and from the device. These links are all connected to the I/O of the HMC package, resulting in a package with 896 pins in total (GDDR5 packages only have 170 pins).

The above optimisations result in the following comparison. Note that in table 3.2 [6] not only GDDR5 and HMC memory, but also other memory architectures are included. The claim of the Hybrid Memory Cube (HMC) consortium [8] that the HMC memory is $\approx 66.5\%$ more energy efficient per bit than DDR3 memory, is concluded from this table. However, comparing the energy efficiency of the HMC to GDDR5 memory, the HMC is only $\approx 17.6\%$ more energy efficient per bit.

Technology	V_{DD} (V)	I_{DD} (A)	**Data rate** (MT/s)[7]	**Bandwidth** (GB/s)	**Power** (W)	**Energy**	
						pJ/Byte	pJ/bit
SDRAM PC133 1GB Module	3.3	1.50	133	1.06	4.95	4652.26	581.53
DDR-333 1GB Module	2.5	2.19	333	2.66	5.48	2055.18	256.90
DDRII-667 2GB Module	1.8	2.88	667	5.34	5.18	971.51	121.44
GDDR5 - 3GB Module	1.5	18.48	33000	264.00	27.72	105.00	13.13
DDR3-1333 1GB Module	1.5	1.84	1333	10.66	2.76	258.81	32.35
DDR4-2667 4GB Module	1.2	5.50	2667	21.34	6.60	309.34	38.67
HMC, 4 DRAM 1Gb w/ logic	1.2	9.23	16000	128.00	11.08	86.53	10.82

Table 3.2: Memory energy comparison

In table 3.2 the relation between the data rate and the bandwidth is:

$$\text{bandwidth} = \text{DDR clock rate} \times \text{bits transferred per clock cycle}/8$$

Memory modules currently used are 64-bit devices. This means that 64 bits of data are transferred at each transfer cycle. Therefore, 64 will be used as *bits transferred per clock cycle* in the above formula. Thus, the above formula can be simplified even further:

$$\text{bandwidth} = \text{DDR clock rate} \times 8$$

[6] The values for V_{DD}, I_{DD} and Data rate can be found in the datasheets of the given memories.
[7] Megatransfers per second

Chapter 4

Benchmarking

In order to compare GDDR5 and HMC memory, it is important to look at the performance and power consumption of these memory types. Because, in a complete system, not only the GDDR5 or HMC will be used, but also a GPU or FPGA, the comparison is made on a graphical card versus the HMC system [15] (see figure 4.1).

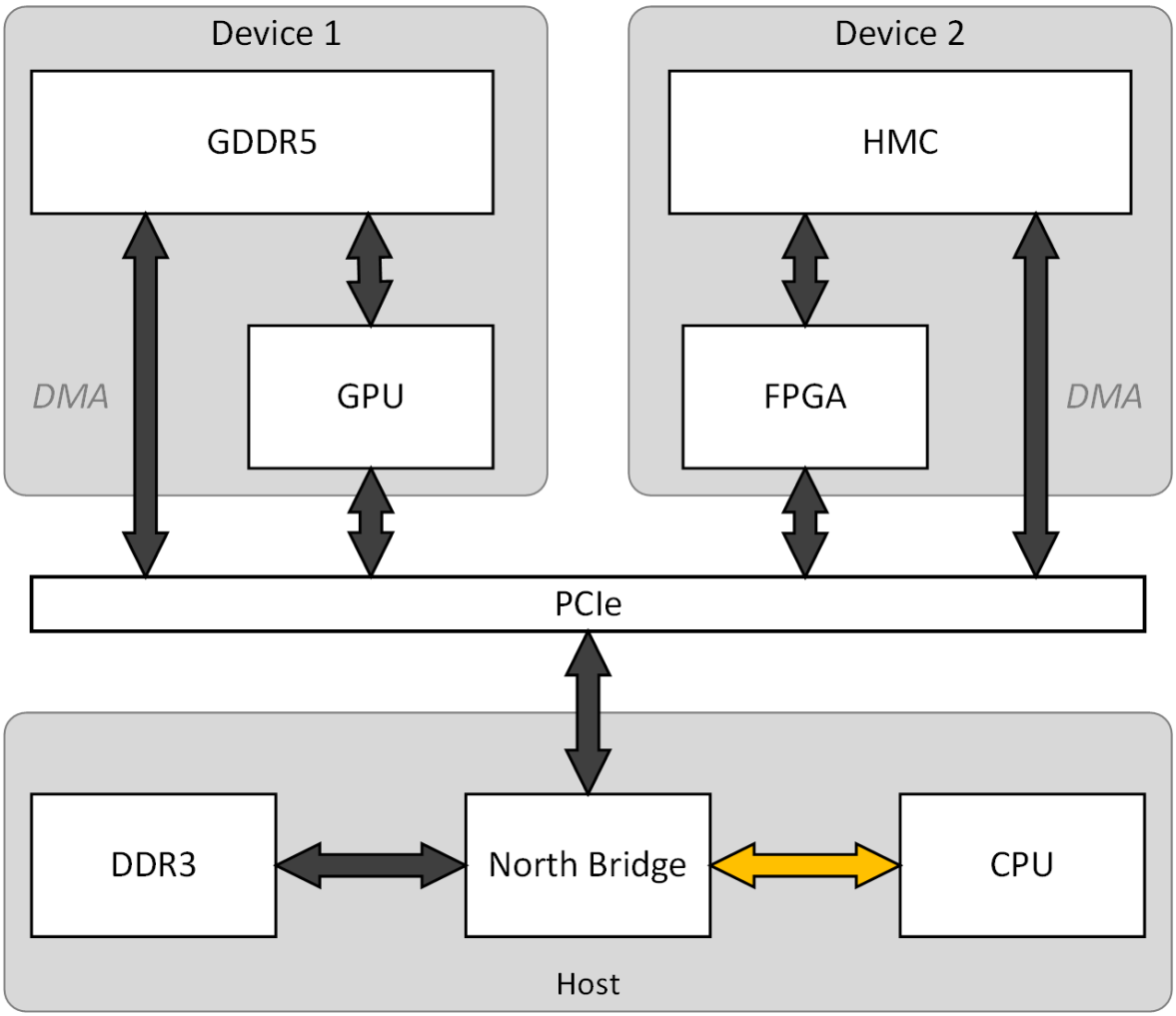


Figure 4.1: Functional block diagram of test system

In the next sections a description of some of the key performance characteristics of a HMC and GDDR5 are given. This should give insight into the relative merits of using either an FPGA in combination with the HMC memory or a GPU in combination with the GDDR5 memory.

There is a vast array of benchmarks to choose from, but for this comparison this is narrowed down to three tests[1], e.g. there will be no gaming and virtual reality benchmarking needed (see figure 4.1):

- How fast can data be transferred between the Personal Computer (PC) (host) and the graphical card (device 2) or HMC card (device 2)?
- How fast can the FPGA or GPU read and write data from HMC or GDDR5 respectively?
- How fast can the FPGA or GPU read data from, do computations and write the result to HMC or GDDR5 respectively?

In these benchmarks, each test is repeated up-to a hundred times to allow for other activities going on on the host and/or to eliminate the first-call overheads. During the repetition of the tests the overall minimum execution time per benchmark is kept as result, because external factors can only ever slow down execution. In the end this results in a maximum bandwidth. In order to get as close to an absolute performance measurement, it is important the host system execute as little tasks as possible.

4.1 Device transfer performance

This test measures how quickly the host can send data to and read data from the device (either HMC or GDDR5) (see figure 4.1). Since the device is plugged into the PCIe bus, the performance is largely dependent on the PCIe bus revision (see table 4.1) and how many other devices are connected to the PCIe bus. However, there is also some overhead that is included in the measurements, particularly the function call overhead and the array allocation time. Since these are present in any "real world" use of the device, it is reasonable to include these overheads.

Note that the PCIe rev 3.0, is used in the test equipment, it has a theoretical bandwidth of $1.0GB/s$ per lane. For the 16-lane slots (PCIe x16) used by the devices a maximum theoretical bandwidth of $16GB/s$ is given. Although an x16 slot is used, most of the devices only use 8 lanes and therefore the maximum theoretical bandwidth is only $8GB/s$.

PCI Express version	Transfer rate[2]	Throughput			
		x1	x4	x8	x16
1.0	2.5 GT/s	250 MB/s	1 GB/s	2 GB/s	4 GB/s
2.0	5 GT/s	500 MB/s	2 GB/s	4 GB/s	8 GB/s
3.0	8 GT/s	984.6 MB/s	3.938 GB/s	7.877 GB/s	15.754 GB/s
4.0 (expected 2017)	16 GT/s	1.969 GB/s	7.877 GB/s	15.754 GB/s	31.508 GB/s
5.0 (far future)	25/32 GT/s	3.9/3.08 GB/s	15.8/12.3 GB/s	31.5/24.6 GB/s	63.0/49.2 GB/s

Table 4.1: PCIe link performance

[1] For all performance tests, also power consumption will be measured.
[2] Gigatransfers per second

4.2 Memory transfer performance

While not all data is sent-to or read-from the PCIe bus, but can be processed 'locally' using a GPU or FPGA (see figure 4.1), a separate test for measuring the Memory transfer performance should be executed. As many operations performed will do little computation with each data element of an array, these operations are therefore dominated by the time taken to fetch the data from memory or write it back to memory. Simple operators like `plus (+)` or `minus (-)` do very little computation per element that they are bound only by the memory access speed.

To know whether the obtained *memory transfer* benchmark figures are fast or not, the benchmark is compared with the same code running on a CPU reading and writing data to the main DDR3 memory. Note, however, that a CPU has several levels of caching and some oddities like "read before write" that can make the results look a little odd. The theoretical bandwidth of main memory is the product of:

- Base DRAM clock frequency. $(2133MHz)$
- Number of data transfers per clock. (2)
- Memory bus (interface) width. $(64bits)$
- Number of channels. (1)

For the used test equipment the theoretical bandwidth (or burst size) is $2133 \cdot 10^6 \times 2 \times 64 \times 1 = 273024000000$ $(273.024$ billion) bits per second or in bytes $34.128GB/s$, so anything above this is likely to be due to efficient caching.

4.3 Computational performance

For operations where computation dominates, the memory speed is not very important. In this case, how fast the computations are performed is the interesting part of the benchmark. A good test of computational performance is a matrix-matrix multiplication. As above, this operation is timed on both the CPU, GPU and the FPGA to see their relative processing power.

Another computation dominated operation is the removal of impulse noise [24], also known as salt-and-pepper noise, from an image. For the removal of this noise a Median Filter can be used. The advantage of this type of filter on an image is that there are no Floating or Fixed Point operations involved. Therefore, the FPGA implementation is much more straight forward, needs less area in the FPGA and can designed and implemented quicker, keeping in mind the time constraints of this project.

4.3.1 Median Filter

In signal processing, it is often desirable to be able to perform some kind of noise reduction on an image or signal. The median filter is a non-linear digital filtering technique, often used to remove noise. Such noise reduction is a typical pre-processing step to improve the results of later processing (see figure 4.2).

The main idea of the median filter is to run through the signal entry by entry, replacing each entry with the median of neighbouring entries. The pattern of neighbours is called the *window*, which slides, entry by entry, over the entire signal. For one dimensional signals, the most obvious window is just the first few preceding and following entries, whereas for two dimensional (or higher-dimensional) signals such as images, more complex window patterns are possible (such as *box* or *cross* patterns). Note that if the window has an odd number of entries, then the median is simple to define: it is just the middle value after all the entries in the window are sorted numerically. For an even number of entries, there is more than one possible median. Therefore, most median filters use a window with an odd number of entries.

The median filter is an *order statistics filter* (see Appendix A), where the filtered output image $\hat{f}[x, y]$ depends on the ordering of the pixel values of the input image g in the window $S_{[x,y]}$. The Median Filter output is the 50% ranking of the ordered values:

$$\hat{f}[x, y] = median\left\{g[s, t], [s, t] \in S_{[x,y]}\right\}$$

For example, a 3×3 Median Filter $g_{S_{[x,y]}} = \begin{pmatrix} 1 & 5 & 20 \\ 200 & 5 & 25 \\ 25 & 9 & 100 \end{pmatrix}$, the values are first ordered as follows: $1, 5, 5, 9, 20, 25, 25, 100, 200$. The 50% ranking (in this case the 5^{th} value) is 20, thus $\hat{f}[x, y] = 20$.

The Median Filter (see Algorithm 1) introduces even less blurring then other filters of the same window size. This filter can be used for Salt noise, Pepper noise or Salt-and-pepper noise. The Median Filter is a non-linear filter. A simple 2D median filter algorithm could look like algorithm 1.

Algorithm 1: 2D median filter pseudo code

Input:
f: Noisy input image with dimensions $N \times M$
w: Window size of the median filter
Output:
$\hat{f}$: Denoised output image with dimensions $N \times M$
Initialisation:
edge$_x = \lfloor S/2 \rfloor$ x offset due to filter window size
edge$_y = \lfloor S/2 \rfloor$ y offset due to filter window size
for *x=edge$_x$:N* **do**
  for *y=edge$_y$:M* **do**
    for *fx=0:w-1* **do**
      for *fy=0:w-1* **do**
        $S(fy \cdot w + fx) = f[x + fx - edge_x][y + fy - edge_y]$ % Fill median filter window
      end
    end
    Sort entries in S
    $\hat{f}(x, y) = S(\lceil w \times w/2 \rceil)$ % Store filtered output pixel
  end
end

Note that this algorithm processes one colour channel only without boundaries.

(a) *(b)* *(c)*

Figure 4.2: Denoising example: (a) Original; (b) Impulse noise; (c) Denoised (3×3 window)

Chapter 5

Power and Energy by using Current Measurements

Power consumption and energy efficiency is becoming more and more important in modern, smaller-sized electronics systems with increasing functionality. The equations for *Power* (Equation 5.1) and *Energy* (Equation 5.2) are as follows:

$$P[t]_{LOAD} = V[t]_{LOAD} \times I[t]_{LOAD} \tag{5.1}$$

$$E = \sum_{i=1}^{T} P\big[t(i)\big] \times \big[t(i) - t(i-1)\big] \tag{5.2}$$

As can be seen in Equation 5.1, to calculate the (instantaneous) power the instantaneous voltage over the load and the instantaneous current through the load must be measured. While the voltage can be easily measured without almost no effect on the load, this is not the case for the measurement of the current through the load. Correctly picking the method to monitor the current for the given system is critical in measuring system efficiency. First the correct method must be decided on, that is whether to use direct or indirect techniques (see table 5.1 at the end of the section).

Indirect current sensing is based on Maxwell's 3^{rd} (*Faraday's law* [25]) and 4^{th} (*Ampere's law* [26]) equations. The basic principle is that as a current flows through a wire, a magnetic field is produced proportional to the current level. By measuring the strength of this magnetic field (and knowing the materials properties), the value of the load current can be calculated.

The sensing elements are commonly called Hall-effect sensors [27]. This non-invasive method is inherently isolated from the load.

Indirect current measurement does not cause any loss of power into the load. It can be used in systems with only a few milliamperes of load up to high currents (> 1000A), up to high voltages (> 1000V), dynamic loads and any area requiring isolation. Using indirect sensing is typically more expensive due to the sensors required and historically these sensors required a fairly large footprint, but nowadays these sensors are available in chip-size up to many tens of amperes, e.g. the ACS715 [28].

On the other hand direct sensing is based on Ohm's law (Eq.5.3). This law simply states that the current flowing though a resistor is directly proportional to the ratio of the voltage across the resistor and it's value. This so called shunt resistor is placed in series with the load. This can be either between the supply and the load (high side) or between the load and ground (low side). This resistor adds power dissipation to the system and therefore this method is invasive. Both isolated and non-isolated variants are available.

$$I_{LOAD} = \frac{V_{LOAD}}{R_{LOAD}} \tag{5.3}$$

For currents and voltages less than $100A$ and $100V$, direct sensing offers a low-cost method. However, the system must be able to tolerate a small loss in power due to the shunt resistor.

Due to the invasiveness of this measurement technique, the main design goal is to minimise the amount of load the measurement system adds to the system. To achieve this goal, the sense-element or shunt resistor must be very small. The typical shunt resistor has a value of less then $50m\Omega$ and in some cases even less then $1m\Omega$. The small value of the shunt resistor results in a fairly small voltage drop, hence signal conditioning is required.

To amplify the small signal measured on the shunt resistor to usable levels, a differential amplifier should be used. There are four main differential amplifier configurations that can be used (see table 5.1) to measure the current:

- Operational Amplifier
- Differential Amplifier
- Instrumentation Amplifier
- Current-Sense Amplifier

The Operational Amplifier (Op-amp) is the most basic implementation. To set the gain and precision levels, the Op-amp relies on external discrete components, hence the Op-amp is only used in low accuracy, low cost systems. The Op-amp requires a feedback path, the input is single-ended and can only be applied in low-side configurations. Because the Op-amp is single-ended, it becomes susceptible to parasitic impedance errors. To increase the precision of the Op-amp high-accuracy components can be used which will increase the cost of the system.

Differential Amplifiers are special Op-amps that integrate gain networks. Differential Amplifiers are used to eliminate parasitic impedance errors. However, the Differential Amplifier is designed to convert large differential signals to large single-ended signals, typically with unity gain. The very small input signals delivered by the shunt resistor requires high gain amplifiers, therefore the Differential Amplifiers architecture isn't suitable for most current-sensing systems. Since these amplifiers have a very high Common-mode Voltage (CMV) range on the inputs, these amplifiers can be used as buffer stages.

An Instrumentation Amplifier is a special three-stage device. This amplifier has two buffers, one on each input of the Differential Amplifier. This architecture creates a high input impedance, resulting in a device for measuring very small currents (actually the voltage drop over the shunt resistor is measured). The Instrumentation Amplifier also eliminates the parasitic impedance error of the Operational Amplifier. The limitation of this amplifier is that the Common-mode Voltage must fall within the supply voltage range.

The last device to measure currents is the Current-sense Amplifier, also called a current-shunt amplifier or current-shunt monitor, that features a unique input stage (Figure 5.1), with:

$$V_O = (V_{SHUNT} + V_P) \times \left(1 + \frac{R_F}{R_G}\right)$$

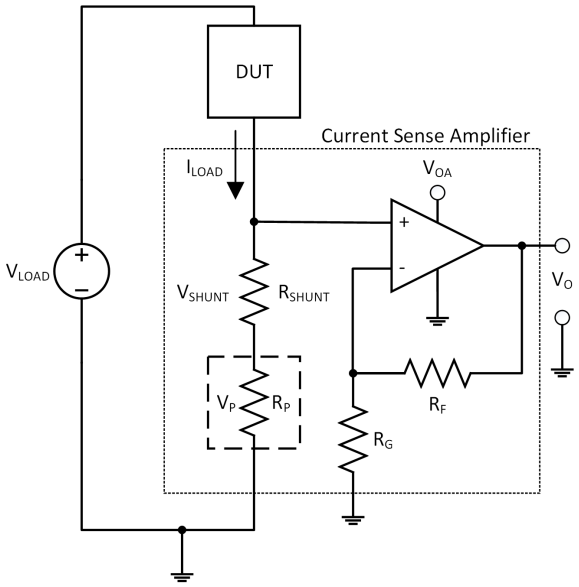


Figure 5.1: Current Sense Amplifier

The amplifier can be exposed to significantly higher Common-mode Voltage than the supply voltage, thanks to its topology. The higher CMV range is obtained by attenuating the non-inverting input by a factor of x times, using a resistor divider network. On the inverting input, the resistors are chosen such that both sides of the amplifier are in balance. The noise gain of the circuit is thereby providing unity gain for differential input voltages. Laser wafer trimming of the thin film resistors yields a minimum Common-mode Rejection (CMR).

Additionally, the low-drift, high precision gain network integration maximises measurement accuracy. For current-sense amplifiers, the input architecture of these amplifiers limits the I_{LOAD} to a minimum of approximately $10\mu A$. Through the input circuitry an input bias current will be flowing introducing a great uncertainty in the measurement.

To determine which implementation to choose, several questions must be answered:

- Will the measurement be on the high or low side?
- What is the Common-mode Voltage to be measured?
- What is the current range to be measured?
- Is the current bidirectional or unidirectional?
- How will the current value be used?

A low-side implementation has the shunt resistor between the load and the system ground, like in figure 5.1. This is the most common method for current monitoring, because the Common-mode Voltage is near ground. However, the shunt resistor disturbs the ground seen by the load and this

prevents special circuitry in the load to detect any shorts-to-ground. On the other hand, a high-side implementation allows for good system grounding, but the shunt resistor is placed between the supply and the load. This results in the Common-mode Voltage being near the load supply voltage. In certain cases this will be well above the amplifiers supply rail making this a no-go.

In the case of a PCIe card, direct sensing will have too much impact on the proper operation of the digital hardware. Therefore, indirect sensing will be applied.

	Indirect	Direct amplifier alternatives			
		Operational	Difference	Instrumentation	Current Sense
Current Range	$mA \rightarrow kA$	$mA \rightarrow A$	$mA \rightarrow A$	$nA \rightarrow kA$	$mA \rightarrow A$
CMV Range	kV	$+V_{cc} \rightarrow -V_{cc}$	$\gg V_{cc} \rightarrow \ll = V_{cc}$	$+V_{cc} \rightarrow -V_{cc}$	$> V_{cc} \rightarrow \leq V_{cc}$
Strengths	No insertion loss Isolated	Low cost Ease of use	High side, high common-mode range	Very low currents	High-precision low and high side high gain
Challenges	Higher cost Larger footprint	Accuracy	Low gain requires $\gg V_{SHUNT}$	Common-mode range limited to volts	Low current limitations

Table 5.1: Current sensing techniques

Chapter 6

Power Estimations

According to Bircher, et. al [29], there is a correlation between the operating systems performance counters of the different sub systems in a system and the power used by the different sub systems in a system. It is shown that using well known performance events within the CPU, such as Direct Memory Access (DMA) transactions and cache misses, are highly correlated to the power consumption of subsystems outside the CPU. Their models are shown to have an average error of less than 9% per subsystem across the considered workloads.

For the main memory in a system it is possible to estimate the power consumption by using the number of *read* or *write* cycles and the percentage of time the operation is within the *precharge*, *active* or *idle* state [30]. Since these performance events are not directly visible to the CPU, an estimation is done by using the performance events of the memory bus accesses by the CPU and other subsystems. It is also necessary to account for memory utilisation caused by subsystems other than the CPU, namely I/O devices performing DMA accesses.

For the power estimation of the *Chipset*[1] and I/O-devices in the system, the specific disk and memory system metrics are not used. The estimation is done by using metrics of the number of Interrupts, DMA and uncacheable accesses in the system.

I/O and Chipset subsystems are composed of rather homogeneous structures and the estimation of their power is done by traditional CMOS power models. These models divide power consumption into *static* and *dynamic* power. The *static* power represents current leakage, while *dynamic* power accounts for the switching currents of CMOS transistors. Since the *static* power does not vary in a system, due to a relatively constant V_{cc} and temperature, only the *dynamic* power is estimated in the I/O and Chipset subsystems.

Power consumption of a system is divided into two components: *Static* and *Dynamic* power, as shown in equation 6.1. The static power is determined by the used chip technology, chip layout and the operating temperature. The dynamic power is determined by events during runtime, mainly the switching overhead in the transistors in the chip.

$$P_{System} = P_{Dynamic} + P_{Static} \tag{6.1}$$

[1] A chipset is a set of electronic components in an IC that manages the data flow between the processor, memory and peripherals. It is usually found on the motherboard. Because the chipset controls the communication between the processor and external devices, the chipset plays a crucial role in determining system performance.

An empirical method to build a power model is proposed by Isci and Martonosi [31]. Equation 6.2 shows their Counter-based Component power Estimation model. They use the component access rates to weight component power numbers. In particular, they use the access rates as weighting factors to multiply against each components maximum power value along with a scaling strategy that is based on micro-architectural and structural properties. In general, all the component power estimations are based on Equation 6.2, where $MaxPower$ and $NonGatedClockPower$ are estimated empirically during implementation. The C_i in the equation are the hardware components, like $ALU_{integer}$, ALU_{FP}, Register File and Instruction Fetch Unit. The total system power (P_{System}) is equal to the summation of the power all individual components.

$$\begin{aligned} P_{System} = \sum_{i=0}^{n} \big[& AccessRate(C_i) \times ArchitecturalScaling(C_i) \\ & \times MaxPower(C_i) + NonGatedClockPower(C_i) \big] + IdlePower \end{aligned} \tag{6.2}$$

6.1 Power Consumption Model for AMD GPU (GCN)

Basically, the power consumption of a GPU is divided in the same way into two components: *Static* and *Dynamic* power, as shown in equation 6.3. The dynamic power is build up out of the dynamic power of the individual components in the GPU, as shown in equation 6.4. The architecture (Graphics Core Next) of the tested GPU is shown in appendix E.

$$P_{GPU} = P_{Dynamic} + P_{Static} \tag{6.3}$$

$$P_{Dynamic} = \sum_{i=0}^{n} (P_{Component})_i = P_{GCNs} + P_{Memory} \tag{6.4}$$

To model the dynamic power of a single GCNs, the GCN is decomposed into the main physical components as shown in equation 6.5.

$$\begin{aligned} P_{GCN} = & P_{ALU_{Int}} + P_{ALU_{FP}} + P_{Cache_{Texture}} \\ & + P_{Cache_{Const}} + P_{Memory_{Global}} + P_{Registers} \\ & + P_{FDS} + P_{GCN_{Const}} \end{aligned} \tag{6.5}$$

$$P_{GCNs} = Number_{GCN} \times \sum_{i=0}^{n} (P_{GCN})_i \tag{6.6}$$

However, as Hong [32] states that, like equation 6.2, the equation 6.5 can be changed into the equation 6.7 , since GPUs do not have any speculative execution. Therefore the estimated hardware access rates are based on the number of instructions and the execution time.

$$P_{GCN} = MaxPower_{Component} \times AccessRate_{Component} \tag{6.7}$$

$$AccessRate_{Component} = \frac{DynamicInstructions_{Component} \times Wavefronts_{GCN}}{Cycles_{Execution}/4} \tag{6.8}$$

$$DynamicInstructions_{Component} = \sum_{i=0}^{n} Instructions\_per\_Wavefront_i(Component) \tag{6.9}$$

$$Wavefronts_{GCN} = \left(\frac{\#Threads_{SIMD\text{-}VU}}{\#Threads_{Wavefront}} \times \frac{\#SIMD\text{-}VU}{\#GCN_{Active}} \right) \tag{6.10}$$

6.2 Power Consumption Model for HMC and FPGA

Similar to the power model for the GPU, a model can be specified for the AC-510 (HMC and FPGA) module.

$$P_{AC510} = P_{Dynamic} + P_{Static} \tag{6.11}$$

$$P_{Static} = P_{HMC_{Static}} + P_{FPGA_{Static}} \tag{6.12}$$

$$P_{Dynamic} = P_{HMC_{Dynamic}} + P_{FPGA_{Dynamic}} \tag{6.13}$$

To model the dynamic power of the FPGA, the FPGA is decomposed into the main physical components as shown in equation 6.14.

$$\begin{aligned} P_{FPGA_{Dynamic}} = {} & P_{Clock} + P_{Logic} + P_{BRAM} + P_{DSP} + P_{PLL} + P_{MMCM} \\ & + P_{IPs} + P_{I/O} + P_{Others} \end{aligned} \tag{6.14}$$

P_{Clock} is all dynamic power in the different clock trees, P_{DSP} is all dynamic power of all used DSPs, P_{PLL} is all dynamic power of the Phase Locked Loops (PLLs), P_{MMCM} is all dynamic power of the Clock Manager, and P_{GTH} is all the dynamic power of the used GTH transceivers in the FPGA implementation.

$$P_{Logic} = P_{Registers} + P_{LUTs} \tag{6.15}$$

$$P_{LUTs} = P_{Combinatorial} + P_{ShiftRegisters} + P_{DistributedRAMs} \tag{6.16}$$

$$P_{I/O} = P_{On-chipI/O} + P_{Off-chipI/O} \tag{6.17}$$

$$P_{On-chipI/O} = P_{LogicI/O} + P_{BufferI/O} + P_{Other/IO} \tag{6.18}$$

$$P_{BRAM} = P_{RAMB18} + P_{RAMB36} \tag{6.19}$$

$$P_{Other} = P_{SYSMON} + P_{Config} + P_{eFUSE} \tag{6.20}$$

To model the dynamic power of the HMC, the HMC is decomposed into the main physical components as shown in equation 6.21.

$$\begin{aligned} P_{HMC_{Dynamic}} &= P_{Links} + P_{Vault} + P_{NetworkOnChip} + P_{Memory} \\ &\quad + P_{BIST} + P_{MaintenacePort} \end{aligned} \tag{6.21}$$

$$P_{Links} = \sum_{i=1}^{N} (P_{Link})_i \quad \text{where } N \in [1, 8] \text{ (Number of Links used)} \tag{6.22}$$

$$P_{Vaults} = \sum_{i=0}^{N} (P_{Vault})_i \quad \text{where } N \in [0, 31] \text{ (Number of Vaults in HMC)} \tag{6.23}$$

Part II

Realisation and results

Chapter 7

MATLAB Model and Simulation Results

7.1 Realisation

7.1.1 Image Filtering

Before creating the image processing kernel in software and hardware a simulation in MATLAB is created. First the algorithm described in section 4.3.1 is extended in order to process the boundaries as well. When filtering pixels at the boundary of an image the filter requires also the neighbouring pixels and some of these pixels will be outside the boundary of the image as shown in figure 7.1a.

Trying to filter the pixel at the (x, y)-coordinate $(0, 0)$ using a 3×3 filtering window would need 5 pixels outside of the image array (see figure 7.1a). These 'pixels' will not have a valid value and when trying to read these values from memory using a program will definitely result in a runtime error. A solution to this problem is to extend the input image with extra rows and columns on the outside. The number of rows and columns needed is calculated as follows: $Window = m \times m \rightarrow Pixels_{extra} = m - 1$. Hence, for a 3×3 window, there will be two extra rows and two extra columns; one on each side of the image. The value of the extra pixels is set to zero (see figure 7.1b).

After adding the extra rows and columns to the image the first pixel $(0, 0)$ in the original image is shifted to coordinate $\left({}^{(m-1)}/_{2}, {}^{(m-1)}/_{2}\right)$.

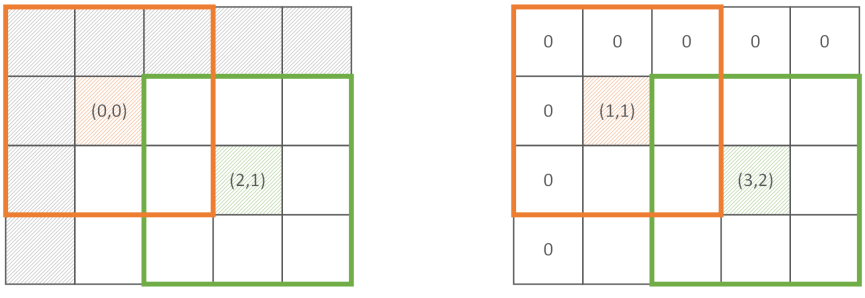


Figure 7.1: Median Filtering: (a) Boundary exceptions; (b) No Boundary exceptions

This intermediate image is larger than the original size, but depending on the original image size and the widow size, the intermediate image is using approximately the same amount of memory or uses a lot more (see table 7.1). This should be taken into account when designing the hardware or software solution.

Window Size	**Image size**		**Difference (Colour image)**	
$(m \times m)$	(original)	(extended)	(pixels)	(Bytes)
3×3	128×128	130×130	516	1.548
5×5	128×128	132×132	1.040	3.120
7×7	128×128	134×134	1.572	4.716
9×9	128×128	136×136	2.112	6.336
17×17	128×128	144×144	4.352	13.056
19×19	128×128	146×146	4.932	14.796
25×25	128×128	152×152	6.720	20.160

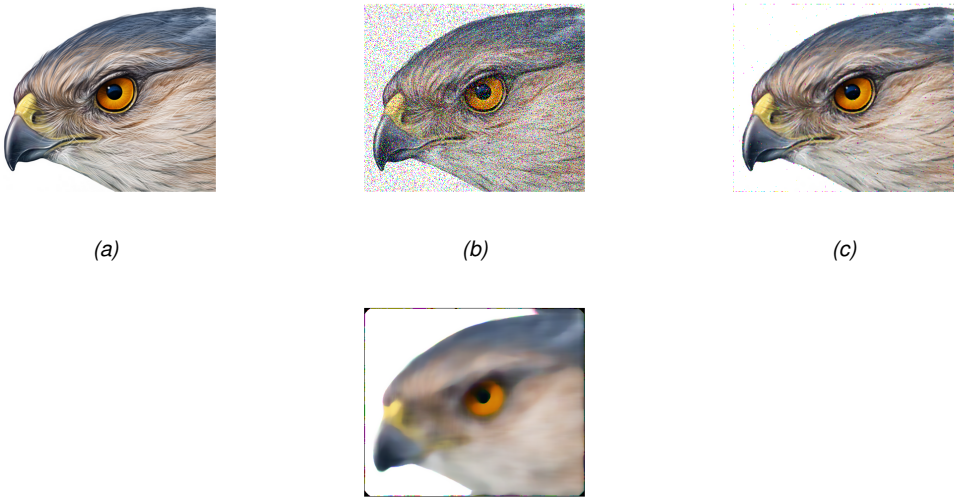
Table 7.1: Median Filter Image Sizes

7.2 Results

The algorithm presented in section 4.3.1 results in MATLAB code to filter an image (see Appendix I). First an image is read and some Impulse Noise is added to this image. Afterwards, the Median Filtering is applied to the noisy image with a window size from 3 up to 25, resulting in an denoised image.

7.2.1 MATLAB results

Running the simulation results in the following filtered images. A result of the median filter is that the larger the window size the more blur is introduced into the filtered image (see figure 7.2).



(a) *(b)* *(c)*

(d)

Figure 7.2: MATLAB Simulation: (a) Original; (b) Impulse Noise; (c) Filtered (3x3); (d) Filtered (25x25)

Chapter 8

Current Measuring Hardware

In order to measure the power consumption of the HMC [16] and GDDR5 memory it is necessary to measure the voltages and currents supplied to the memory chips in the system. This is a cumbersome task because the individual memory chips are not easily accessible for measuring voltages and currents.
Also extra hardware is always needed to use the memory. This extra hardware is already included on the graphics card (power management, PCIe interfaces, and video interfaces) or on the HMC backplane (power management, PCIe interface, and FPGA) [33]. Therefore, measuring the voltages and currents on the PCIe connector, instead of measuring the voltages and currents on the individual memory chips, will be more easy during the project.

To measure the needed signals, a special PCIe riser card [34] needs to be developed. On the card the signals to be measured are made easily accessible. (see Appendix D)

8.1 Realisation

The function of the riser card is to make the power lines on the PCIe interface available. All the other signals will just pass through to the other end of the card. This is shown in figure 8.1.

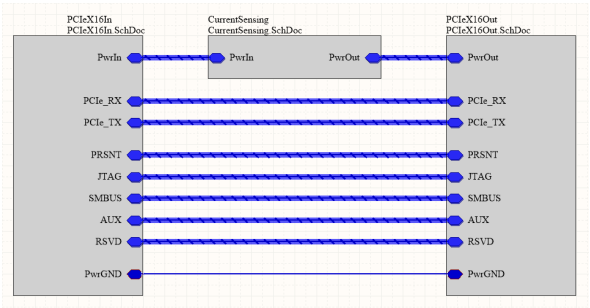
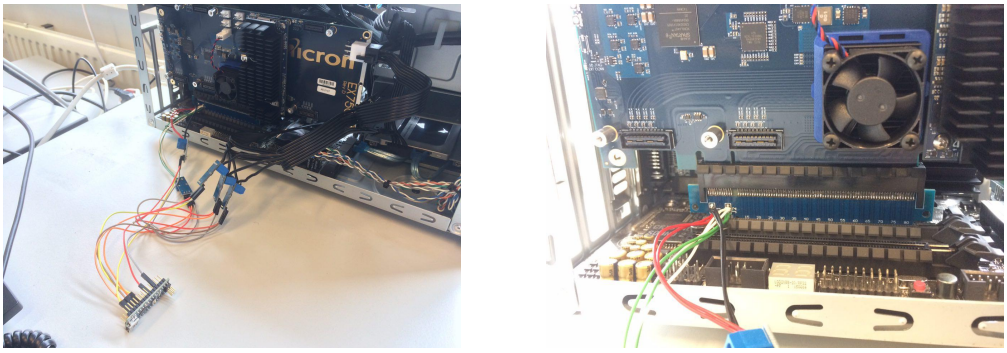


Figure 8.1: Riser card block diagram

The complete schematic (see appendix F) results in a riser card of dimensions 99.80×22.00mm which can be placed in a PCIe slot of a main board. In turn, the device to be measured, can be inserted on top of this card.

8.2 Results

Figures 8.2a and 8.2b show the test set-up and a close-up of the riser card. The connectors `J1` and `J2` are connected to the ACS715 [28] Hall sensor (see figure 8.3).



(a) *(b)*

Figure 8.2: Test set-up: (a) Overview; (b) Riser card Printed Circuit Board (PCB) - with HMC Backplane inserted

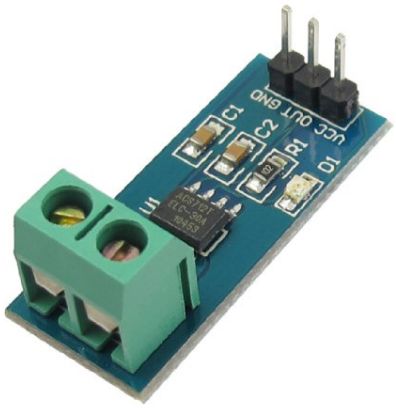


Figure 8.3: Hall Sensor board ACS715 - Current Sensing

The Hall sensor used in this experiment is susceptible to external magnetic influences. Therefore, it is necessary the create a baseline measurement before measuring currents on the HMC and GPU systems. This baseline will be used to correct the measurements on the different systems. All measurements are collected using an Arduino Nano micro controller [35] (see figure 8.4).

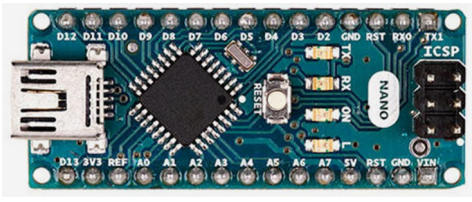


Figure 8.4: Arduino Nano with an ATmega328p AVR

Measuring the voltage from the Hall sensor and converting it to currents is actually not as simple as doing the math. First of all the measured voltage is converted to a digital value using an Analog to Digital Converter (ADC). The digital value is then divided by the maximum ADC value and multiplied by the supply voltage.

$$Voltage = \frac{ADCValue}{1023} \times V_{dd}$$

This surely looks straight forward. The Arduino Nano is plugged into the USB, which, by definition, supplies $5V$. However, this depends on the load of the USB port and therefore the $5V$ is a rough approximation. Even using an external, more accurate and stable power supply, it is necessary to know the supply voltage at the moment of measurement, to make more accurate readings.

Measuring the $5V$ connection on the Arduino while plugged in to the USB and connected to the Hall sensors is actually reading $4.35V$. That makes a big difference to the results of the conversion from ADC to voltage value. And it fluctuates. Sometimes its $4.35V$, sometimes its $4.63V$. So, the supply voltage must be known at the time of reading the ADC value.

However, having a known precise voltage for measurement using the ADC, it is possible to calculate the supply voltage. Fortunately, the AVR chip used on the Arduino Nano has just such a voltage available (V_{ref}), and can be measured with the ADC.

Using the above for a more accurate voltage reading resulted in the Arduino firmware as can been seen in Appendix B.

The results of reading the voltage outputs of the Hall sensors, when the complete system under test is switched off, are shown in figure 8.5. For each sensor the average current is calculated over a period of approximately 60 seconds. These averages are subtracted from the measured current during the other experiments. This results in an average offset shown in table 8.1. Due to the fact that the Hall sensor functions on the magnetic field created by the current flow through the sensor, when no current flows the sensor still picks up the magnetic field of the surroundings. Hence, these errors are present.

Hall Sensor	Avg Current (mA)
PCIe Main connector ($12V$)	28.038
PCIe Main connector ($3.3V$)	7.469
PCIe connector (pin 1) ($12V$)	32.309
PCIe connector (pin 2) ($12V$)	7.419
PCIe connector (pin 3) ($12V$)	20.622

Table 8.1: AC715 Hall sensors average current - Power off

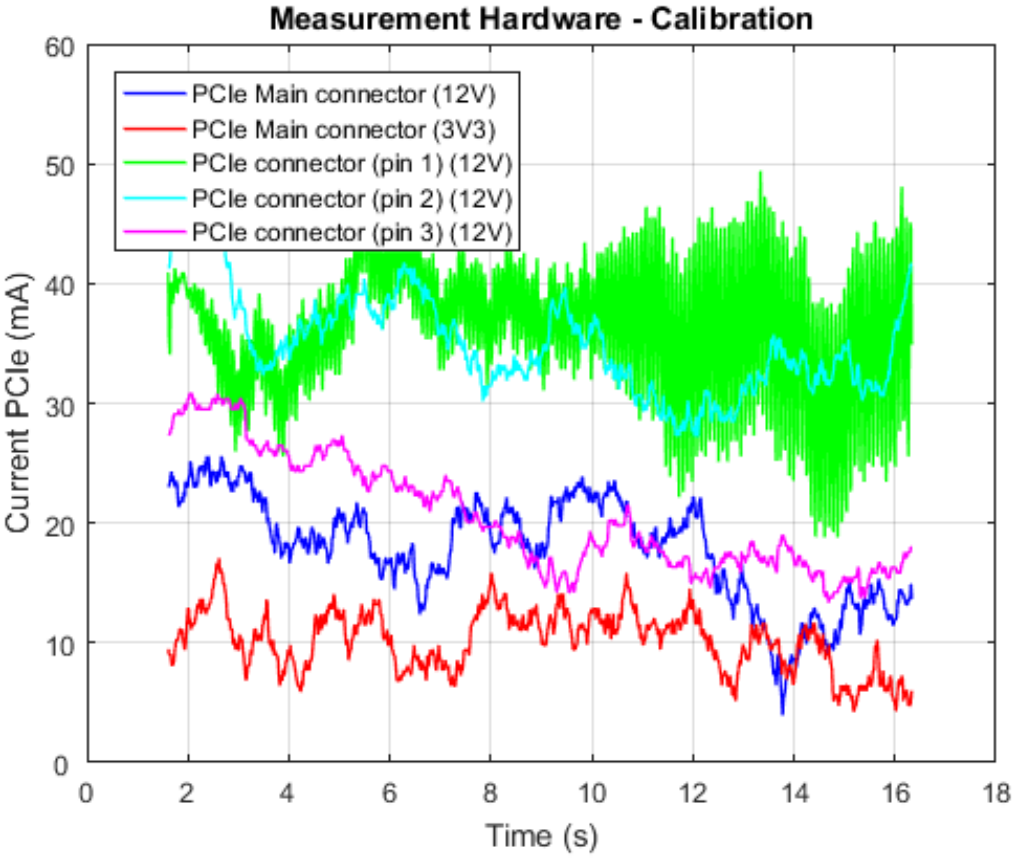


Figure 8.5: AC715 Hall sensors readout - Power off

Chapter 9

GPU Technology Efficiency

In this chapter the results on performance and energy efficiency for the Graphics Processing Unit (GPU) system in combination with GDDR5 memory are presented.

9.1 Realisation

The benchmark performs the following tests:

- Host $\Leftrightarrow$ GPU system transfer performance test.
- GPU system local memory (GDDR5) transfer performance test.
- GPU system energy efficiency test.

9.1.1 Host/GPU system transfer performance

As described in section 4.1, this test measures how quickly data can be sent-to and read-from the GPU. The maximum theoretical bandwidth is $8GB/s$. For retrieving the actual bandwidth an OpenCL program according to algorithm 2 is executed. The complete benchmark program code is found in appendix G.

Algorithm 2: Bandwidth calculation based on $Bandwidth = SendBytes/Sendtime$

Output:
$Bandwidth$: Calculated bandwidth array with length 15
for *S*=2^{12}:2^{26} **do**
  Allocate host buffer $D(S)$ % Host array with dimension S
  Allocate device buffer $H(S)$ % Device array with dimension S
  repeat
    Reset and Start Counter
    $H = D$ % Copy host array to device array
    Stop Counter and Retrieve $Time_{Elapsed}$
    if $Time_{Elapsed} < Time_{Stored}$ **then**
      Store $Time_{Elapsed}$
    end
  until D *is copied* $100\times$;
  Calculate $Bandwidth(S)$
end

9.1.2 GPU system local memory (GDDR5) transfer performance

This test is for the memory transfer bandwidth between the GPU and the GDDR5, because the data is only sent once over the PCIe bus before using this data 'locally', and afterwards is only read once over the PCIe bus. As many operations performed will do only little computation with each data element of an array, these operations are therefore dominated by the time taken to fetch the data from memory or write it back to memory. A simple operator, `plus (+)`, is used while it does a little computation per element that it is bounded only by the memory access speed.

For retrieving the actual bandwidth an OpenCL program according to algorithm 3 is executed. In this algorithm the instruction **EnqueueKernel** indicates that a specific OpenCL kernel is loaded, initialised and executed. The complete benchmark code, including the OpenCL kernel, is found in appendix G.

Algorithm 3: Calculate local $Bandwidth = SendBytes/Sendtime$

Input:
N: Number of OpenCL kernels
Output:
$Bandwidth$: Calculated bandwidth array with length 15
for S=2^{12}:2^{26} **do**
  Allocate host buffer $D(S)$ % Input array with dimension S
  Allocate device buffer $H(S)$ % Output array with dimension S
  repeat
    Reset and Start Counter
    EnqueueKernel(H, N) % OpenCL function to copy arrays
    Stop Counter and Retrieve $Time_{Elapsed}$
    if $Time_{Elapsed} < Time_{Stored}$ **then**
      Store $Time_{Elapsed}$
    end
  until D *is copied* $100\times$;
  Calculate $Bandwidth(S)$
end

9.1.3 GPU energy efficiency test

This test is used for measuring the GDDR5 memory energy consumption. To perform a more computationally complex operation, the OpenCL kernel used in the local memory transfer performance measurement in the previous section, is replaced with a kernel which removes the impulse noise (salt-and-pepper noise) from an image, by implementing a median filter (see Appendix A).

For measuring the actual power usage, an OpenCL program according to algorithm 3 is executed. The OpenCL kernel is found in appendix H.

9.2 Results

This section presents the results on performance and power for the GPU system.

9.2.1 Host/GPU transfer performance

The benchmark (see Algorithm 2) will transfer data of different sizes from the host to the GPU. After executing the Host/GPU transfer benchmark, the following results are gathered with a peak bandwidth of $\approx 2.7GB/s$ (see table 9.1). The same data is shown in figure 9.1.

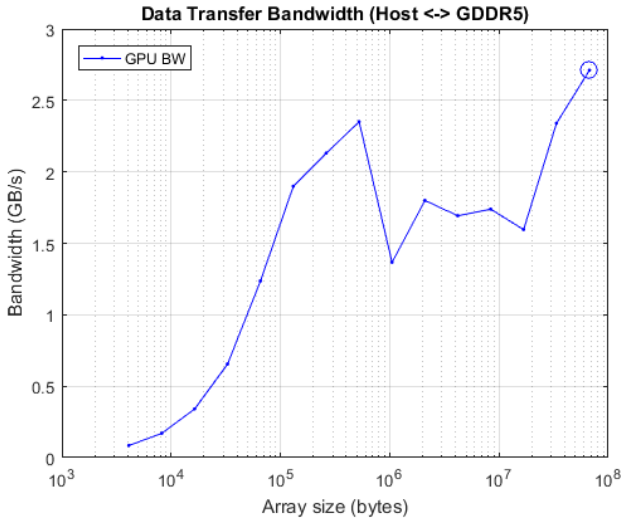


Figure 9.1: Host to GPU Bandwidth

Data size (B)	Bandwidth[1] GB/s
4,096 (4kB)	0.0853333
8,192 (8kB)	0.1706670
16,384 (16kB)	0.3413330
32,768 (32kB)	0.6553600
65,536 (64kB)	1.2365300
131,072 (128kB)	1.8995900
262,144 (256kB)	2.1312500
524,288 (512kB)	2.3510700
1,048,576 (1MB)	1.3671100
2,097,152 (2MB)	1.8001300
4,194,304 (4MB)	1.6933000
8,388,608 (8MB)	1.7392900
16,777,216 (16MB)	1.5964600
33,554,432 (32MB)	2.3405700
67,108,864 (64MB)	2.7112500

Table 9.1: Host to GPU Bandwidth

[1] The circled values indicates the peak value.

In this benchmark the data is basically transferred through the CPU caches. When accessing some memory that is not in any cache, it triggers a cache line load from the closest cache, *L1*, which will cascade through the *L2* and *L3* caches, until main memory is reached. At this point an *L3* cache line sized chunk of data from main memory is fetched to fill the *L3* cache. Then an *L2* cache line sized chunk of data from the *L3* cache is fetched to fill the *L2* cache, and a similar fetch from the *L2* cache to fill the *L1* cache.

The program can now resume processing of the memory location that triggered the cache miss. The next few iterations of copying data to GDDR5 memory will proceed with data from the *L1* cache, because a cache line size is larger than the data size that is accessed in each iteration. After some small number of iterations, the *L1* cache line end is reached, and this triggers another cache miss. This may or may not go all the way out to main memory again, depending on the cache line sizes of the *L2* and *L3* caches.

The *L1*, *L2* and *L3* total cache sizes in the test system are: $384KiB$[2] (2-3 clock cycle access), $1536KiB$ (≈ 10 clock cycle access) and $15MiB$[3] ($\approx$ 20-30 clock cycle access) respectively.

The drop in bandwidth at a data size of $1MB$, seen in table 9.1 and figure 9.1, is caused by the first cache miss in the *L2* cache. At this point the *L2* cache needs to retrieve data from the much slower *L3* cache, which causes the data transfer towards the GDDR5 memory.

9.2.2 GPU/GDDR5 transfer performance

With the execution of the Local memory transfer benchmark (algorithm 3), the following results are gathered with a peak bandwidth of $\approx 3.74GB/s$ and $\approx 169.125GB/s$ using a single kernel or 128 kernels respectively (see table 9.2). The same data is shown in figure 9.2. As expected the GDDR5 bandwidth is much larger compared to the PCIe bandwidth. Furthermore, a design that is more focused on the GPU and GDDR5 memory also gives a more constant rise in bandwidth over the total range measured.

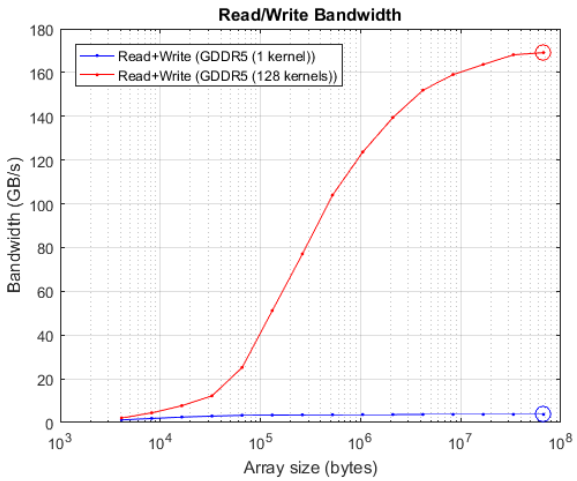


Figure 9.2: GPU Local memory Bandwidth

[2] 1 kilobyte (kB) is 1000 bytes. 1 kilobyte (KB) is 1024 bytes. To address the confusion between kB and KB, 1024 bytes are called a kibibyte (i.e., kilo binary byte) or 1 KiB. [36]

[3] 1.048.576 bytes are called a Mebibyte (i.e., Mega binary byte) or 1 MiB. [36]

[4] The circled values indicates the peak value.

Data size (B)	Bandwidth (GB/s)[4]	
	1 kernel	128 kernels
4,096	1.24878	2.13334
8,192	1.89630	4.45218
16,384	2.54410	7.72830
32,768	2.97891	12.22686
65,536	3.33009	25.20613
131,072	3.42404	51.20000
262,144	3.55788	77.10113
524,288	3.60186	104.02538
1,048,576	3.68128	123.65288
2,097,152	3.69088	139.43875
4,194,304	3.71375	151.96750
8,388,608	3.72516	159.11625
16,777,216	3.73076	163.71250
33,554,432	3.73423	168.14250
67,108,864	3.73591	169.12500

Table 9.2: GPU Local memory Bandwidth

9.2.3 GDDR5 energy efficiency results

The initial measurements taken are those for the idle power used by the tested GPU (Radeon HD7970). For a timespan of approximately 65 seconds, the currents towards the GPU are measured, using the AC715 Hall sensors [28]. This results in a static power of approximately $15.05 J/s$, as shown in figure 9.3.

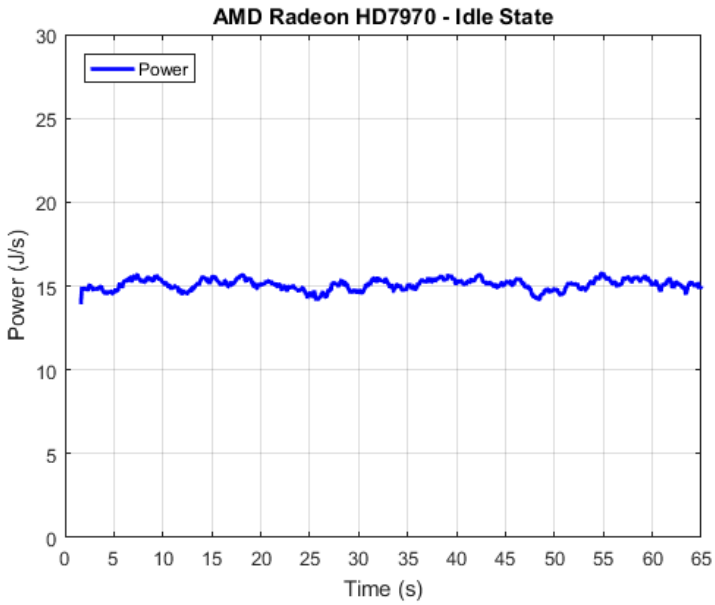


Figure 9.3: GPU Idle Power

A benchmark program is created using the pseudo code for the median filter presented in section 4.3.1. This benchmark applies a 3×3 median filter to an image with dimensions 5208×3476 (width $\times$ height). This results in the following metrics:

Description	Value
Image Width	5208 pixels
Image Height	3476 pixels
Total # pixels	18.103.008 pixels
# Colour channels	3 (RGB)
# Memory Loads per pixel	9 (3×3 window size)
# Memory Stores per pixel	1

Table 9.3: GPU Median Filter Metrics

The execution time of the denoising kernel on the AMD Radeon HD7970 GPU is approximately $522.85ms$. Running the kernel repeatedly for $120\times$ results in a power measurement running for approximately 62.74 seconds. The resulting power measurement is shown in figure 9.4. The average power is equal to $21.5J/s$.

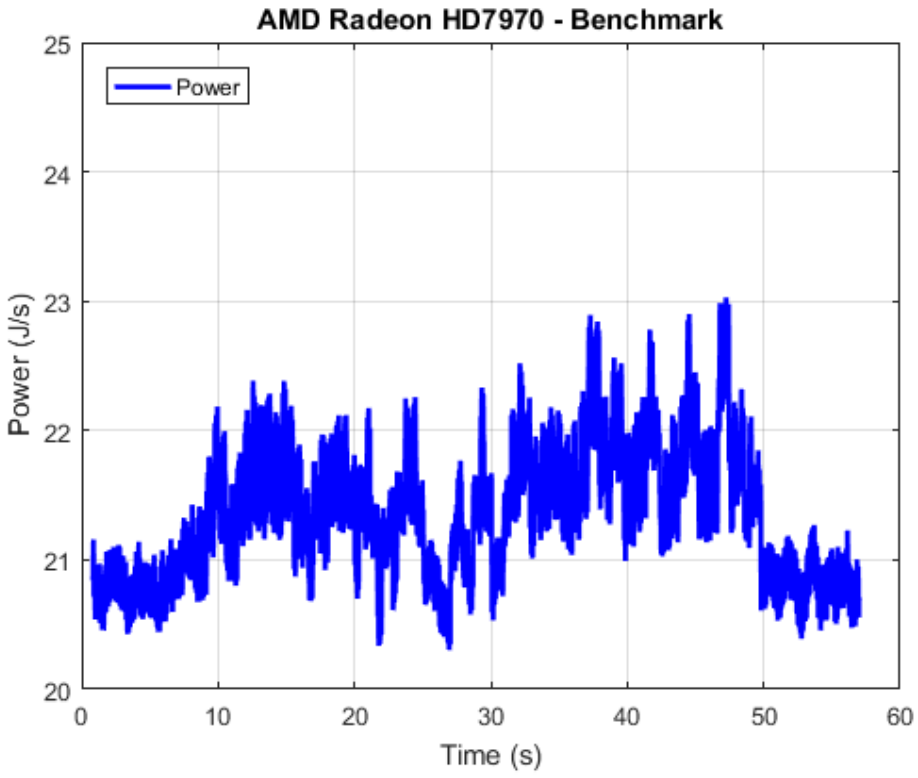


Figure 9.4: GPU Benchmark Power

Performing the fairly simple task of denoising the given image on the test set-up results in an extra power usage, defined as the difference of the average benchmark power (see figure 9.4) and the average Idle power (see figure 9.3), of approximately $\approx 21.5J/s - \approx 15.05J/s \approx 6.45J/s$.

Chapter 10

Hybrid Memory Cube Technology Efficiency

In this chapter the results on performance and energy efficiency for the Hybrid Memory Cube (HMC) system, i.e. an FPGA (see Appendix C) in combination with HMC memory, are presented.

10.1 Realisation

In contrast to the GPU system with the GDDR5 memory, for the HMC system it is fairly easy to measure the power usage and throughput. During the implementation of the memory controller it became clear that the read latency of the HMC memory itself could also be measured. By adding performance counters inside the memory controller these measurements are not very difficult although these measurements are not part of the initial set of criteria.

For the measurement of the read latency the memory controller provided by Micron will be extended with the HPC challenge (HPCC) RandomAccess benchmark [37].

10.1.1 Memory Controller Timing Measurements

The structure of the provided memory controller makes it possible to include Application Specific Integrated Circuit (ASIC) like hardware inside the controller.

Inside the user wrapper of the provided controller, 9 *User Modules* are implemented (see figure 10.1). The top level of a single *User Module* implements the HPCC RandomAccess benchmark [37]. This benchmark is typically reported in terms of Giga-Updates Per Second (GUPS), so this *User Module* is referred to as the GUPS module.

This *User Module* comprises 3 main pieces:

- **gups_reader**: if enabled, this generates read requests to the memory. This takes care to not generate a new request if there is no available tag[1].

[1] Read request packets are uniquely identified with a tag field embedded in the packet. Tag fields in packets are 11 bits long, which is enough space for 2048 tags. The tag range is independent for each host link.

- **gups_writer**: if enabled, this generates write requests to the memory. This also creates some write data to send to the memory. If reads are also enabled, this waits until a read completes before sending a write to the recently completed read address. If reads are not enabled, then this just writes to the memory as fast as possible.
- **gups_arbiter**: this is a passive observer that measures what's happening in the system. This measures things like the number of reads and writes (which we can convert to bandwidth) and latency for reads.

Most of the setup of the GUPS modules is done in software. The software is using the *PicoBus*, which is part of the Pico Framework by Micron [38]. This includes giving the *gups_reader* the amount of tags to use, resetting the user module, enabling reads versus writes, etc. The software also passively monitors the performance of the system by reading counters and status registers via the PicoBus.

A feature of the HMC interface is that the user can select inside the FPGA whichever clock they want to run on, e.g. the HMC TX, HMC RX or the slower PicoBus clock. In this case, the HMCs TX clock is selected. Being able to select the HMCs interface clock has the advantage to avoid having to cross back and forth into the memory clock domain, in the case of using the slower PicoBus clock, and run into problems with the synchronisation of the two clock domains.

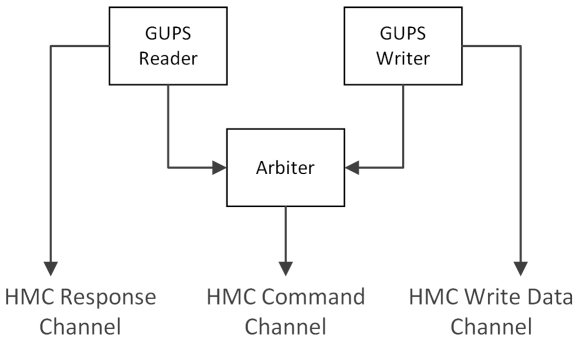


Figure 10.1: Hybrid Memory Cube Memory Controller - User Module Top Level

The HMC memory controller, with the included GUPS *User Modules*, is used to perform the following test:

- Host $\Leftrightarrow$ Device transfer performance test.
- Device local memory transfer performance test.
- Device energy efficiency test.

10.1.1.1 User Module - Reader

This module handles the issuing of read requests to the memory (see figure 10.2). It takes care to not issue a read if it does not have a free tag for the request. Tags are managed by the tag manager. When the read data comes back, the tag is used to identify which read the data corresponds to. This is because reads may come back out of order. An Linear Feedback Shift Register (LFSR) in the address generator is used to generate a random read addresses.

When the read completes, the write datapath is provided the address that is just used for the read (if performing a read-modify-write). This is done to write back to the same address. Therefore a memory is used to store the read address. This memory is indexed by tag. When the read data comes back, the tag is used to look up the original read address. That original read address is asserted on the completed_address signal.

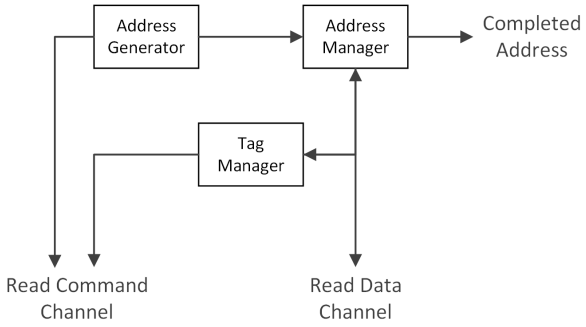


Figure 10.2: User Module - Reader

10.1.1.2 User Module - Writer

This module handles the issuing of write requests to the memory (see figure 10.3). By using posted writes, there is no need to worry about a response for each write. Therefore the same tag can be used for each write request. In order to parallel the design of the GUPS Reader , a tag manager is instantiated, but ONLY 1 tag is loaded into the tag manager. In this way, there will always be another tag available for the next write request, and this tag for the write requests will always be 0.

In the event that the module is in write-only mode, an LFSR is used to generate a random write addresses. If in read-modify-write mode, the module waits for an address on the `rd_addr` input before sending out the next write request.

In a true GUPS, the read data is accepted and than XOR some value into that read data before writing it back to the memory. This is viewed as being (performance-wise) equivalent to just writing some new data to that same address in memory. Therefore 2 64-bit LFSRs are used to cook up the write data.

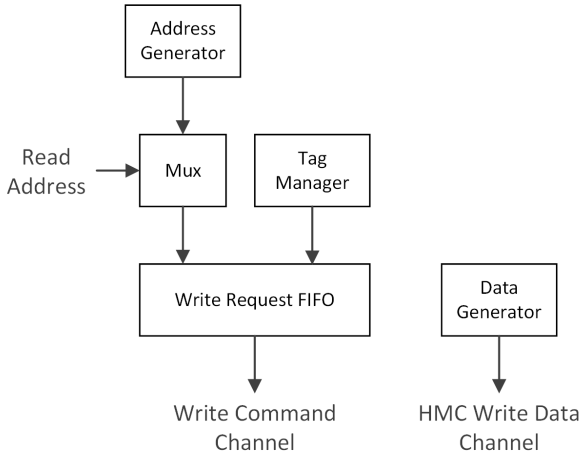


Figure 10.3: User Module - Writer

10.1.1.3 User Module - Arbiter

This is a passive observer that measures the following things:

- number of read or write requests
- number of read or write data
- minimum and maximum read latency
- sum of all read latencies

There is also the ability to bring the entire system to a screeching halt using the `stop` output. Currently this feature is not used (stop = 0), but this feature can be added in the future.

Note that care is taken to not affect the timing on any of the HMC communication channels. This is done by registering all inputs before observing them.

10.1.2 Memory Controller Energy Measurements

For the measurements of the energy consumption of the HMC device a DSP application should be created like the image processing kernel used for the GPU system (see Appendix G). For this the Micron OpenCL accelerator framework can be used to simply convert the used GPU kernel into Verilog HDL.

From one moment to the other, the HMC hardware stopped functioning. It became impossible to program the FPGA. Together with the manufacturer, Micron, the complete system was debugged. After two weeks of troubleshooting , Micron shipped a replacement EX-750 backplane and AC-510 HMC module. Unfortunately, these replacement parts exhibited similar behaviour. After an additional 2 weeks of troubleshooting, no solution was found to solve the problem with the hardware.

At that moment, the same problem arose with similar hardware used by Micron, and a decision was made to stop troubleshooting and just to finish without the power measurements on the HMC. Fortunately, before the malfunction of the HMC hardware, timing measurements and an idle power measurement were taken .

10.2 Results

10.2.1 HMC performance results

Using the extended HMC memory controller, using all 9 user modules, *read*, *write* and *read-write* operations per second are measured. The test is performed repeatedly to average out one time anomalies during the measurements. Finally this results in the number of GUPS. As expected, it takes longer to transfer data to or from the memory, when the packet size[2] increases. This is clearly shown figure 10.4.

[2]Number of bytes transferred in a single read or write request.

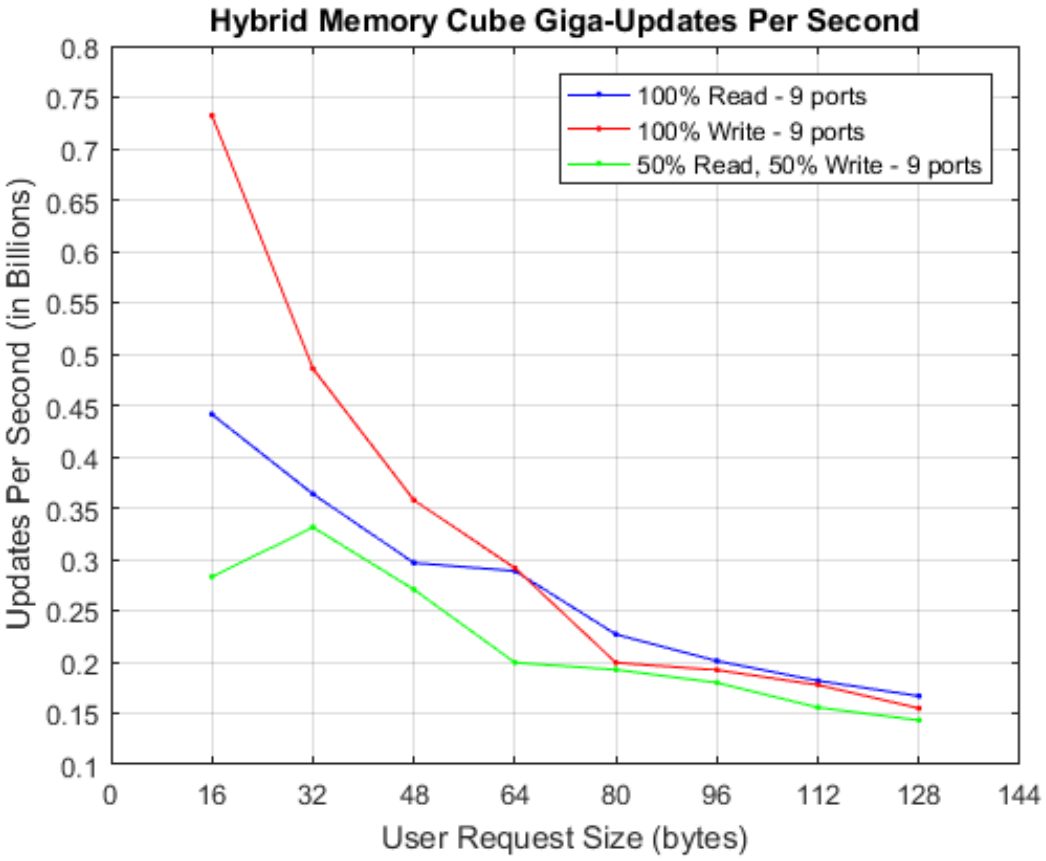


Figure 10.4: Hybrid Memory Cube Giga-Updates Per Second

Although the number of operations decreases as the packet size increases, the throughput or bandwidth increases as the packet size increases. This is calculated using the following equation:

$$Bandwidth_{Max} = GUPS_{Max} \times Packet_{Size} \times 10^9$$

This results in the maximum bandwidth as shown in table 10.1 and figure 10.5. It is clear that the measured bandwidth of approximately $20GB/s$ is only $^1/_3$ of the theoretical maximum of $60GB/s$.

Data size (B)	Bandwidth (GB/s)		
	HMC Read	HMC Write	HMC Read/Write
16	7.06	11.72	4.53
32	11.64	15.54	10.60
48	14.23	17.16	12.99
64	18.48	18.66	12.76
80	18.15	15.95	15.40
96	19.30	18.45	17.26
112	20.36	19.90	17.45
128	21.36	19.83	18.33

Table 10.1: Hybrid Memory Cube Bandwidth (9 user modules)

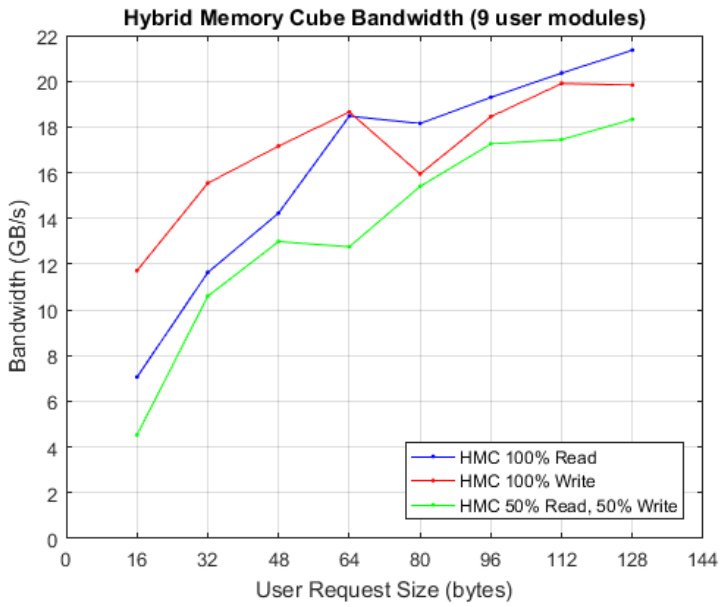


Figure 10.5: Hybrid Memory Cube Bandwidth (9 user modules)

The bandwidth of a single user module HMC system is shown figure 10.6.

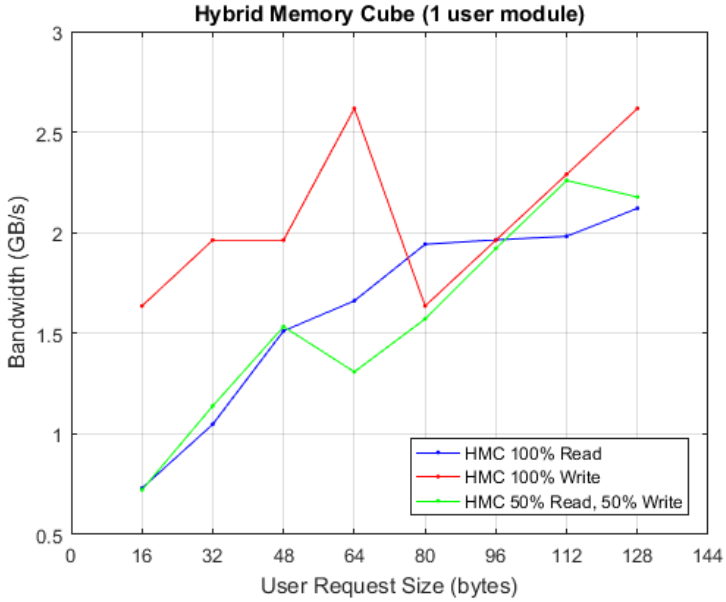


Figure 10.6: Hybrid Memory Cube versus DDR3 Bandwidth

It is clearly shown that the HMC system performs better when utilising the HMC features to the full extent, that is, using the maximum data bus width of 128 bits and using the parallelism by creating multiple *User Modules* in the FPGA.

Another extra feature measured is the minimum and maximum latency for reading from the HMC memory. The latency is measured in number of clock cycles and can be easily converted into seconds. The results are shown in figure 10.7.

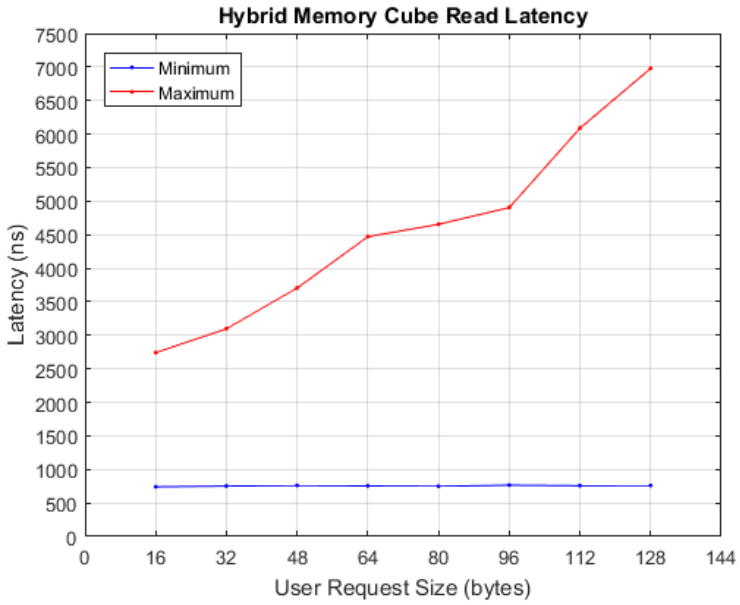


Figure 10.7: Hybrid Memory Cube Read Latency

10.2.2 HMC energy efficiency results

Due to the failure of the hardware and not being able to resolve the issue after four weeks, there are no results for the dynamic power of the HMC system.

However, just before the failure an idle measurement was performed and this resulted in a static power usage of approximately $9.82 J/s$ as shown in figure 10.8.

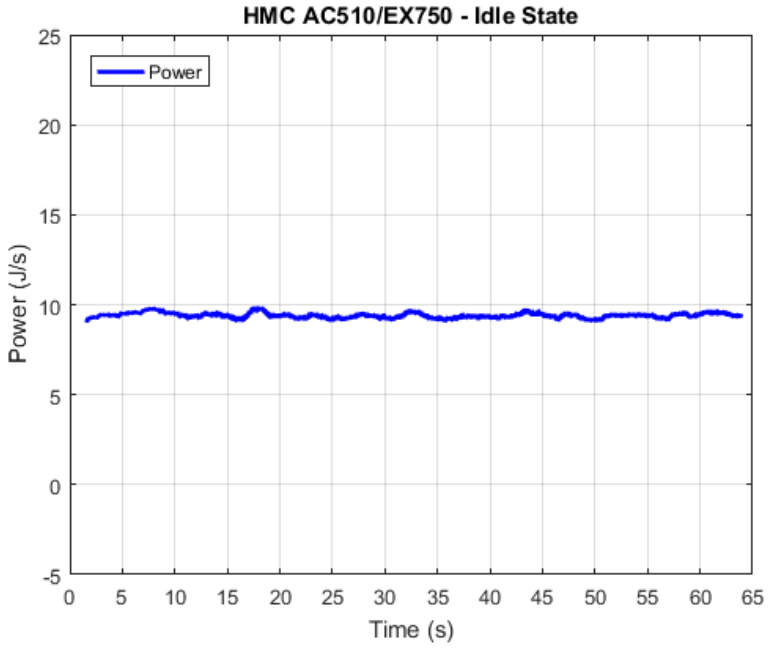


Figure 10.8: HMC Idle Power

Part III

Conclusions and future work

Chapter 11

Conclusions

The problem statement of this thesis is formulated in the introduction (Section 1.2) as follows:

To evaluate the efficiency of the HMC, it is necessary to use a benchmark which can also run on other memory architectures, like GDDR5. The benchmark will provide the following metrics:

- *Achievable throughput/latency (performance);*
- *Achievable average power consumption.*

This research will attempt to answer the following questions:

- *What is the average power consumption and performance of the HMC?*
- *How does the power consumption of the HMC compare to GDDR5 memory?*
- *How does the performance of the HMC compare to GDDR5 memory?*
- *What are the bottlenecks and (how) can this be improved?*

11.1 General Conclusions

What is the average power consumption and performance of the HMC? For the used HMC memory - a 2, half width, link memory chip - the measured average performance of the HMC is $\approx 20.595GB/s$ (see section 11.2). The average power consumption could not be measure because the HMC hardware stopped functioning (see section 11.3).

How does the power consumption of the HMC compare to GDDR5 memory? A comparison of the idle power usage of the HMC system compared to the GPU system resulted in a $\approx 34.75\%$ more energy efficient system in favour of the HMC (see section 11.3).

How does the performance of the HMC compare to GDDR5 memory? The performance of the HMC system with a bandwidth of $\approx 20.595GB/s$ is approximately $5.5\times$ better compared to for the GPU system (see section 11.2).

What are the bottlenecks and (how) can this be improved? The main bottleneck is to create a high performance, energy efficient DSP and memory controller in the FPGA fabric to connect the HMC memory to the rest of the system. Using the OpenCL ability of the Xilinx FPGA it is more easy to create such a DSP (see section 11.4).

11.2 HMC performance

The Hybrid Memory Cube (HMC) consortium [8] claims that the performance of HMC memory can be $15\times$ better than DDR3 memory. This claim is, approximately, supported by the given theoretical bandwidth for the HMC and DDR3 memories, $128GB/s$ and $10.66GB/s$ respectively (see table 3.2; section 3.6.2), which indicates that the HMC is $\approx 12\times$ faster. Comparing the performance of GDDR5 memory to the used HMC memory, using the numbers from table 3.2, $264GB/s$ and $128GB/s$ respectively, GDDR5 memory is $\approx 2.06\times$ faster. Replacing the HMC memory from table 3.2 by an HMC memory with 8, full lane links ($480GB/s$), the HMC has a theoretical performance gain of $\approx 1.8\times$ compared to GDDR5.

The experiments in this thesis show that a single Hybrid Memory Cube can provide a higher performance than GDDR5 memory, by comparing the maximum GPU local memory bandwidth (see Table 9.2) to the maximum HMC bandwidth (see Table 10.1). Taking the average bandwidth (see Table 11.1) of the HMCs read and write maximum bandwidths and compare this with the maximum bandwidth of the GPU, the HMC has a performance gain of approximately $5.5\times$ compared to GDDR5.

Bandwidth (GB/s)				**Gain (X)**
HMC[1]			**GPU**[2]	
Read	**Write**	***Average***		
21.36	19.83	20.595	3.73423	5.515

Table 11.1: Memory Bandwidth Gain

The difference in theoretical $(1.8\times)$ and measured $(5.5\times)$ performance of ${}^{5.5}/_{1.8} \approx 3.1\times$ is caused by the experimental set-up. Not only the memory is tested, but the complete system around the memory is taken into account, that is the complete graphical card for the GDDR5 memory and the FPGA for the HMC memory. The extra performance gain is a result of a better performance of the DSP application created specifically for the task performed.

11.3 HMC power consumption

Based on the data-sheets, using the data rate and the supply voltage and current[3], the average energy per bit can be calculated. The energy per bit for HMC, GDDR5 and DDR3 is $10.82pJ/bit$, $13.13pJ/bit$ and $32.35pJ/bit$ respectively. This results in the HMC being $\approx 66.553\%$ and $\approx 17.593\%$ more energy efficient than DDR3 and GDDR5 respectively.

Unfortunately, the available HMC hardware stopped functioning before dynamic power measurements could be done. The malfunction of the HMC hardware could not be resolved, even with help from Micron, in a reasonable time. It was not possible to provide a new bitstream to the Xilinx FPGA. Although the HMC Linux kernel driver recognised the hardware, it was impossible to stream the new

[1] Values from Table 10.1

[2] Values from Table 9.2

[3] Operation Burst Test (IDD4): one bank activated; continuous read/write burst with 50% data toggle on each data transfer; random bank and column addresses

bitstream. After reinstalling the complete system from scratch with Ubuntu 14.04.2 (with kernel version 3.13), the hardware was still not functioning. Even a replacement of the complete HMC hardware did not result in a functioning set-up.

Due to the malfunctioning hardware, no energy measurements other than an idle power measurement is performed on the HMC system. A comparison of the idle power usage of the EX-750/AC-510 HMC system ($\approx 9.82J/s$ (see figure 10.8)) compared to the AMD Radeon HD 7970 GPU system ($\approx 15.05J/s$ (see figure 9.3)) resulted in a $\approx 34.75\%$ more energy efficient system in favour of the HMC.

11.4 HMC improvements

Due to the fact that the HMC hardware stopped functioning, it becomes hard to provide any solid improvements to the HMC memory controller. Nevertheless, it seems useful to develop an application-specific, ASIC-like memory controller (such as the GUPS User Modules) and include this controller into the HMC die, to get the best performance on the HMC architecture for the specific task to perform. Using the OpenCL ability of the Xilinx FPGA it is more easy to create such a DSP.

Another issue is the Pico Framework [38] provided by Micron. The framework release 5.6.0.0 is only supported on Ubuntu 14.04.2 (with kernel version 3.13) or CentOS 6.7. Both operating systems have an End Of Life (EOL) date set on August 2016 and May 2017 respectively. This may cause problems with the HMC or other hardware in the future, while developing new applications with the HMC.

Chapter 12

Future work

12.1 General future work

As mentioned in this thesis, the Hybrid Memory Cube (HMC) system failed to function and therefore no dynamic power measurements could be retrieved. To really finalise the evaluation and make a fair comparison between GDDR5 and HMC memory, the dynamic power must be measured.

12.2 Hybrid Memory Cube Dynamic power

Due to the fact that the HMC system failed, no further development is done to create the image processing hardware in the FPGA implementation of the memory controller. Micron also provides an OpenCL accelerator to implement any OpenCL kernel into the FPGA. By using this accelerator it will be fairly easy to implement the same image processing kernel, used by the GPU system, into the FPGA.

12.3 Approximate or Inexact Computing

Another interesting field to explore the usage of the HMC architecture, is the research area of inexact or approximate computing [39]. The main research question would be how well an HMC device performs when lowering the supply voltage while keeping the frequency fixed.

12.4 Memory Modelling

There is an open source framework (DRAMPower [40]) to model DDR3 memory. The tool is based on the DRAM power model developed jointly by the Computer Engineering Research Group at TU Delft and the Electronic Systems Group at TU Eindhoven and verified by the Microelectronic System Design Research Group at TU Kaiserslautern with equivalent circuit-level simulations.

This model could be extended to model the HMC and GDDR5 memory as well. By modelling these other memory architectures, not only a more complete design space exploration can be made in the design phase of a project, but also the final implementation can be validated.

Although the DDR3 model is provided, extending this model will be a complete project on it's own.

Part IV

Appendices

Appendix A

Mathematical Image Processing

A.1 Image Restoration

Image restoration is the process of recovering an image that has been degraded, using a-priori knowledge of the degradation process.

Lets assume for the *degradation model* that: $f(x, y)$ is the true ideal image to be recovered and $g[x, y]$ its degraded version. One relation that links the images f and g is the degradation model:

$$g[x, y] = H\,[f]\,[x, y] + n[x, y]$$

where H is the degradation operator (e.g. blur) and n is the additive noise.

Inverse problem: Knowing the degradation operator H and statistics of the noise n, find a good estimate $\hat{f}$ of f.

A.1.1 Denoising

In order to only deal with noise in the image, the assumption is made that the degradation operator is the identity. The linear degradation model becomes $g[x, y] = f[x, y] + n[x, y]$[1].

A.1.1.1 Random noise

The assumption is made that the noise intensity levels are seen as a random variable, with associated histogram or Probability Density Function (PDF) [41] denotes as $p(r)$. It is also assumed that the noise n is independent of the image f and independent of the spatial coordinates $[x, y]$.

The most common types of noise are:

- Gaussian noise [42] (additive) with associated PDF $p(r) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(r-\mu)^2/2\sigma^2}$, where μ is the mean and σ is the standard deviation.
- Uniform noise[2] [43] (additive) with associated PDF $p(r) = \begin{cases} \frac{1}{B-A} & \text{if } A \leq r \leq B \\ 0 & \text{otherwise} \end{cases}$

[1] Not all types of noise are additive.
[2] Also known as White noise

- Impulse noise[3] [24] (not additive) with associated PDF $p(r) = \begin{cases} p_A & \text{if } r = A \\ p_B & \text{if } r = B \\ 0 & \text{otherwise} \end{cases}$

There are several filters for removing random noise. Assume g to be the noisy input image and $\hat{f}$ the denoised output image. Let $S_{[x,y]}$ be a neighbourhood of the pixel $[x, y]$ defined by:

$$S_{[x,y]} = \{[x + s, y + t], -a \leq s \leq a, -b \leq t \leq b\}$$

, of size mn, where $m = 2a + 1$ and $n = 2b + 1$ are positive integers.

1. *Arithmetic Mean Filter*

   $$\hat{f}[x, y] = \frac{1}{mn} \sum_{[s,t] \in S_{[x,y]}} g[s, t]$$

   This filter is used for removing Gaussian or uniform noise. Although removing the noise it introduces blur into the output image.

2. *Geometric Mean Filter*

   $$\hat{f}[x, y] = \left(\prod_{[s,t] \in S_{[x,y]}} g[s, t] \right)^{1/mn}$$

   This filter also introduces blurring as compared to the Gaussian Mean Filter, but tends to lose less detail in the image.

3. *Contra-harmonic Mean Filter of order Q*

   $$\hat{f}[x, y] = \frac{\sum_{[s,t] \in S_{[x,y]}} g[s, t]^{Q+1}}{\sum_{[s,t] \in S_{[x,y]}} g[s, t]^{Q}}$$

   Here the parameter Q gives the order of the filter.
   $Q = 0$ reduces the filter to the Arithmetic Mean Filter.
   $Q = -1$ is the Harmonic Mean Filter. It works well for Gaussian and Salt noise, but fails for Pepper noise.
   $Q > 0$ is used for removing Pepper noise
   $Q < 0$ is used for removing Salt noise.
   As seen above, this filter cannot remove Salt noise and Pepper noise simultaneously.

4. *Median Filter* is an *order statistics filter*, where $\hat{f}[x, y]$ depends on the ordering of the pixel values of g in the window $S_{[x,y]}$. The Median Filter output is the 50% ranking of the ordered values:

   $$\hat{f}[x, y] = median \left\{ g[s, t], [s, t] \in S_{[x,y]} \right\}$$

   For example, a 3×3 Median Filter $g_{S_{[x,y]}} = \begin{pmatrix} 1 & 5 & 20 \\ 200 & 5 & 25 \\ 25 & 9 & 100 \end{pmatrix}$, the values are first ordered as follows: $1, 5, 5, 9, 20, 25, 25, 100, 200$. The 50% ranking (in this case the 5^{th} value) is 20, thus $\hat{f}[x, y] = 20$.
   The Median Filter introduces even less blurring then other filters of the same window size. This filter can be used for Salt noise, Pepper noise or Salt-and-pepper noise. The Median Filter is a non-linear filter.

[3] Also known as Salt-and-pepper, or Bipolar noise

5. *Midpoint Filter*

$$\hat{f}[x,y] = \frac{1}{2}\left[\max_{[s,t]\in S_{[x,y]}}\{g[s,t]\} + \min_{[s,t]\in S_{[x,y]}}\{g[s,t]\}\right]$$

This filter is useful for Gaussian or Uniform noise.

6. *Alpha-trimmed Mean Filter*

Let $\{d \in \mathbb{N} \,|\, 0 \leq d \leq mn-1\}$. First the mn pixel values of the input image in the window $S_{[x,y]}$ are ordered and then the lowest ${}^{d}/_{2}$ and the largest ${}^{d}/_{2}$ are removed. The remaining $mn-d$ values are denotes by g_r.

$$\hat{f}[x,y] = \frac{1}{mn-d}\sum_{[s,t]\in S_{[x,y]}} g_r[s,t]$$

With $d=0$ it is reduced to a Arithmetic Mean Filter and with $d=\frac{mn-1}{2}$ it is reduced to a Median Filter.

This filter is useful for the reduction of all the above mentioned noises. When the noise is stronger, the use of a larger window $S_{[x,y]}$ is required. However, this will introduce more blurring. There are adaptive, more complex in design, filters that will produce less blurring. For example the *Adaptive Median Filter* and the *Adaptive, Local Noise Reduction Filter* [44].

Appendix B

Arduino Current Measure Firmware

```
double IntVref = 1.1;
double vPow = 4.35;
int reference;
double Vcc;
bool calibrated = false;
unsigned int ADCValue0;
unsigned int ADCValue1;
unsigned int ADCValue2;
unsigned int ADCValue3;
unsigned int ADCValue4;
unsigned long msec = 0;

void setup() {
Serial.begin(128000);
Serial.println("CLEARDATA");
Serial.println("LABEL,Time,Timer,Cal,Vref,Vcc,ADC12V,ADC3V3,ADC12V1,ADC12V2,
    ↪ ADC12V3,Time");
Serial.println("RESETTIMER");
}

void loop() {
// Read 1.1V reference against AVcc
ADMUX = _BV(REFS0) | _BV(MUX3) | _BV(MUX2) | _BV(MUX1);
delay(2); // Wait for Vref to settle
ADCSRA |= _BV(ADSC); // Convert
while (bit_is_set(ADCSRA,ADSC));
reference = ADCL;
reference |= ADCH<<8;

Vcc = (IntVref / reference) * 1023;

Serial.print("DATA,TIME,TIMER,");
Serial.print(calibrated); Serial.print(",");
Serial.print(IntVref,4); Serial.print(",");

if (not calibrated) {
if ((int)(Vcc * 100) != (vPow * 100)) {
IntVref = (IntVref * vPow) / Vcc;
} else {
calibrated = true;
}
}
```

```
msec = micros();
ADCValue0 = analogRead(0); delay(0.05);
ADCValue0 = analogRead(0); delay(0.05);

ADCValue1 = analogRead(1); delay(0.05);
ADCValue1 = analogRead(1); delay(0.05);

ADCValue2 = analogRead(2); delay(0.05);
ADCValue2 = analogRead(2); delay(0.05);

ADCValue3 = analogRead(3); delay(0.05);
ADCValue3 = analogRead(3); delay(0.05);

ADCValue4 = analogRead(4); delay(0.05);
ADCValue4 = analogRead(4); delay(0.05);

Serial.print(Vcc,6); Serial.print(",");
Serial.print(ADCValue0); Serial.print(",");
Serial.print(ADCValue1); Serial.print(",");
Serial.print(ADCValue2); Serial.print(",");
Serial.print(ADCValue3); Serial.print(",");
Serial.print(ADCValue4); Serial.print(",");
Serial.println(msec);
}
```

Appendix C

Field Programmable Gate Array

The FPGA used in the Micron AC-510 HMC Module [16] is a Xilinx Kintex Ultrascale 060 of type *FFVA1156*.

GTH Quad 128 X0Y8-X0Y11 (RCAL)	HP I/O Bank 48	HP I/O Bank 68	PCIe X0Y2	GTH Quad 228 X1Y16-X1Y19
GTH Quad 127 X0Y4-X0Y7	HP I/O Bank 47	HP I/O Bank 67	PCIe X0Y1	GTH Quad 227 X1Y12-X1Y15
GTH Quad 126 X0Y0-X0Y3	HP I/O Bank 46	HP I/O Bank 66	SYSMON Configuration	GTH Quad 226 X1Y8-X1Y11 (RCAL)
HP I/O Bank 25	HP I/O Bank 45	HR I/O Bank 65	Configuration	GTH Quad 225 X1Y4-X1Y7
HP I/O Bank 24	HP I/O Bank 44	HR I/O Bank 64	PCIe X0Y0 (tandem)	GTH Quad 224 X1Y0-X1Y3

Figure C.1: XCKU060 Banks

GTH Quad 128 X0Y8-X0Y11 G [L] (RCAL)	HP I/O Bank 48 H	HP I/O Bank 68 K	PCIe X0Y2	GTH Quad 228 X1Y16-X1Y19 E [R]
GTH Quad 127 X0Y4-X0Y7 F [L]	HP I/O Bank 47 G	HP I/O Bank 67 J	PCIe X0Y1	GTH Quad 227 X1Y12-X1Y15 D [R]
GTH Quad 126 X0Y0-X0Y3	HP I/O Bank 46 F	HP I/O Bank 66 I	SYSMON Configuration	GTH Quad 226 X1Y8-X1Y11 C [R] (RCAL)
HP I/O Bank 25	HP I/O Bank 45 E	HR I/O Bank 65 C	Configuration	GTH Quad 225 X1Y4-X1Y7 B [R]
HP I/O Bank 24	HP I/O Bank 44 D	HR I/O Bank 64 R	PCIe X0Y0 (tandem)	GTH Quad 224 X1Y0-X1Y3 A [R]

Figure C.2: XCKU060 Banks in FFVA1156 Package

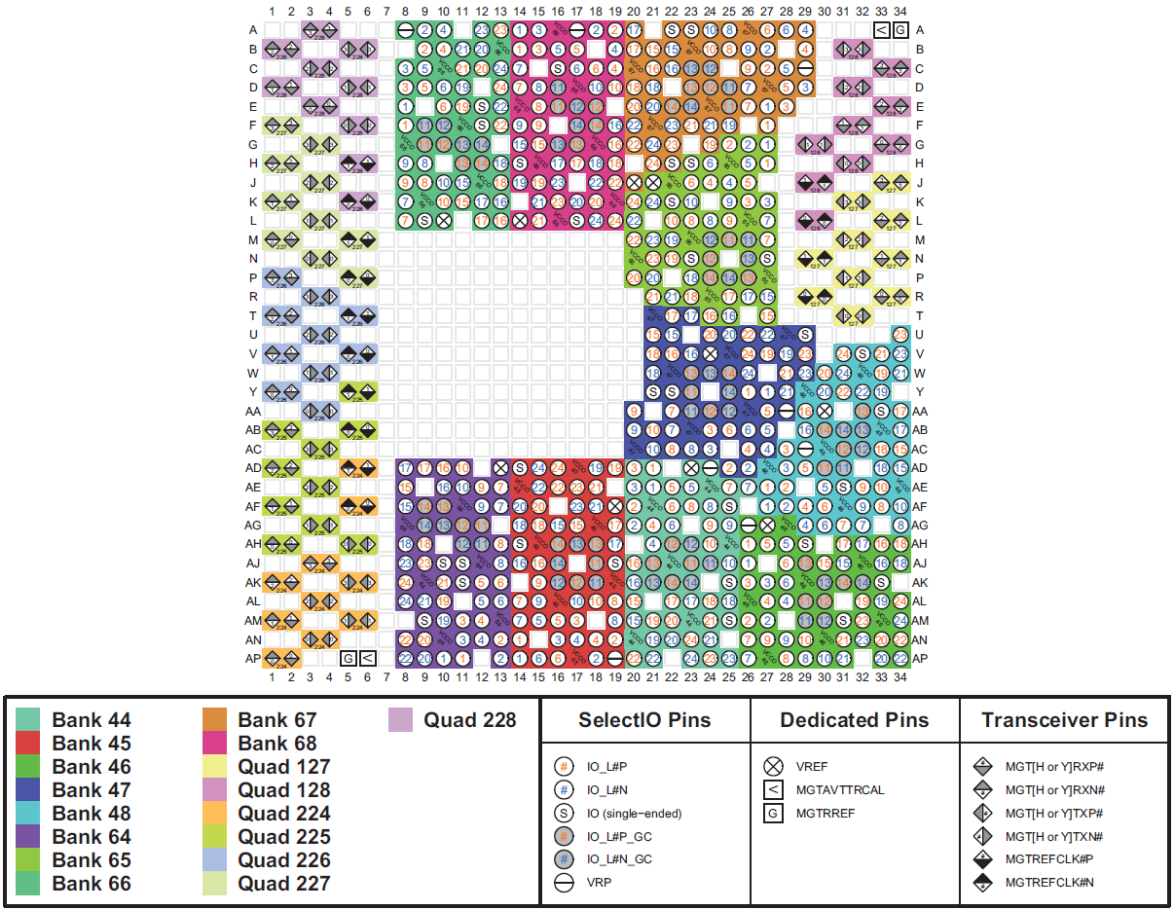
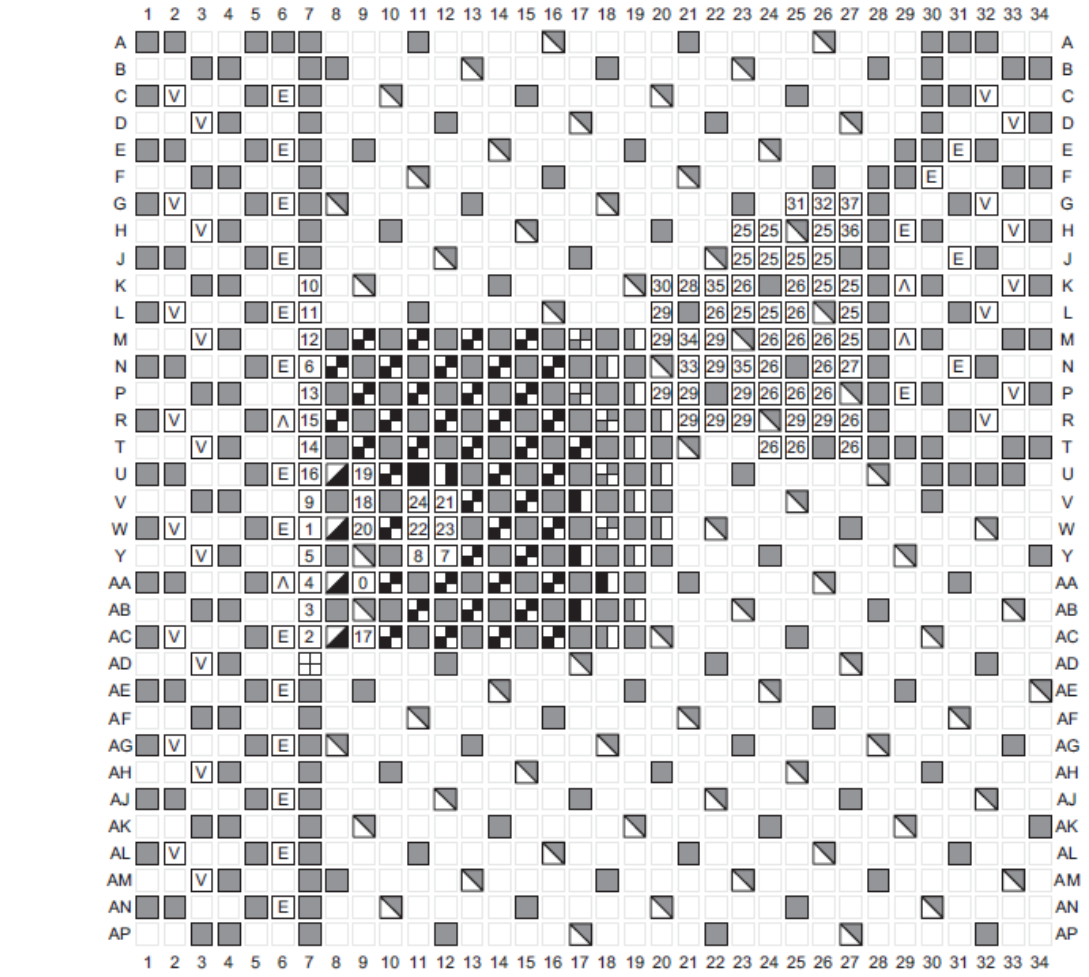


Figure C.3: FFVA1156 PackageXCKU060 I/O Bank Diagram



Power Pins		Dedicated Pins		Multi−Function I/O Pins	
GND		0	CCLK_0	15	PUDC_B_0	25	A[16 to 28]
VBATT		1	CFGBVS_0	16	RDWR_FCS_B_0	26	A[00 to 15]_D[16 to 31]
VCCAUX_IO		2	D00_MOSI_0	17	TCK_0	27	CSI_ADV_B
VCCAUX		3	D01_DIN_0	18	TDI_0	28	DOUT_CSO_B
VCCINT		4	D02_0	19	TDO_0	29	D[04 to 15]
VCCINT_IO		5	D03_0	20	TMS_0	30	EMCCLK
VCCO_[bank number]		6	DONE_0	21	VP	31	FOE_B
VCCBRAM		7	DXP	22	VN	32	FWE_FCS2_B
VCCADC		8	DXN	23	VREFP	33	I2C_SCLK
GNDADC		9	INIT_B_0	24	VREFN	34	I2C_SDA
n	NC	10	M0_0			35	PERSTN[0 to 1]
E	MGTAVCC_[R or L]	11	M1_0			36	RS0
V	MGTAVTT_[R or L]	12	M2_0			37	RS1
Λ	MGTVCCAUX_[R or L]	13	POR_OVERRIDE				
		14	PROGRAM_B_0				

Figure C.4: FFVA1156 PackageXCKU060 Configuration/Power Diagram

	KU060
System Logic Cells	725,550
CLB Flip-Flops	663,360
CLB LUTs	331,680
Maximum Distributed RAM (Mb)	9.1
Block RAM Blocks	1,080
Block RAM (Mb)	38.0
CMTs (1 MMCM, 2 PLLs)	12
I/O DLLs	48
Maximum HP I/Os[1]	520
Maximum HR I/Os[2]	104
DSP Slices	2,760
System Monitor	1
PCIe Gen3 x8	3
150G Interlaken	0
100G Ethernet	0
GTH 16.3Gb/s Transceivers[3]	32
GTY 16.3Gb/s Transceivers[4]	0
Transceiver Fractional PLLs	0

Table C.1: Kintex UltraScale FPGA Feature Summary

Package	**Package dimensions**	**KU060**		
		HR	**HP**	**GTH**
FFVA1156	35x35	104	416	28

Table C.2: Kintex UltraScale Device-Package Combinations and Maximum I/Os

[1] HP = High-performance I/O with support for I/O voltage from 1.0V to 1.8V.
[2] HR = High-range I/O with support for I/O voltage from 1.2V to 3.3V.
[3] GTH [45] transceivers in SF/FB packages support data rates up to 12.5Gb/s. See table C.2.
[4] GTY [45] transceivers in Kintex UltraScale devices support data rates up to 16.3Gb/s. See table C.2.

Appendix D

Test Set-up

The complete test set-up can schematically depicted by figure D.1.

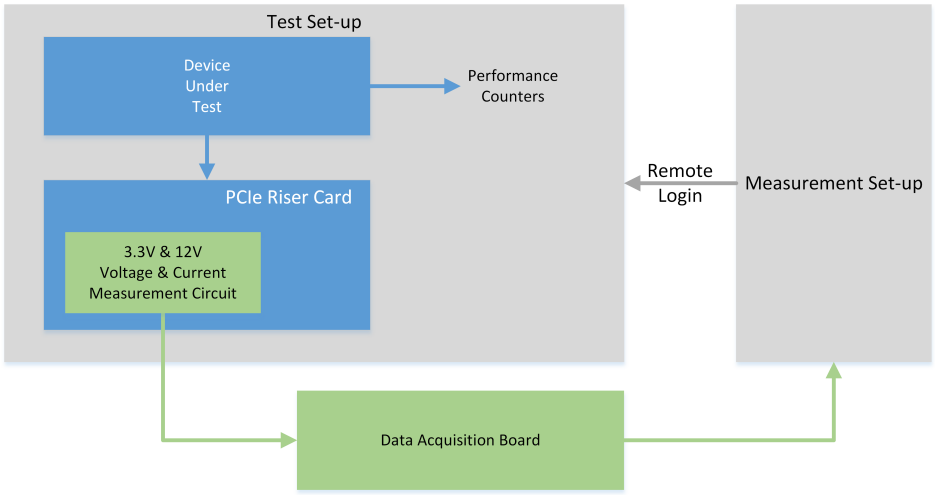


Figure D.1: Test Set-up block diagram

The Device Under Test (DUT) is either an Hybrid Memory Cube (HMC) or a Graphics Processing Unit (GPU). To measure the currents non-intrusively on the custom PCIe riser card the use of Hall Effect-Based Linear Current Sensor ICs [46] is needed. The DUT and the riser card are place inside an industrial PC.

The Data Acquisition Board can consist of many different compositions, i.e.:

- Custom ADC board + Raspberry PI;
- Arduino Nano + PC.

The first can give a higher resolution on the ADC part, but will result in more hardware development during the project. The latter provides an 8 channel 10-bits ADC included in the Micro Controller Unit (MCU) which is already available, but the resolution is limited.

Finally, the Measurement Set-up will be a common desktop PC or laptop in order to communicate with both the Industrial PC to execute the experiments and to read measurement data from the Data Acquisition Board

Appendix E

Graphics Processing Unit

The architecture of the AMD Radeon HD 7900 GPU is schematically depicted by figure E.1.

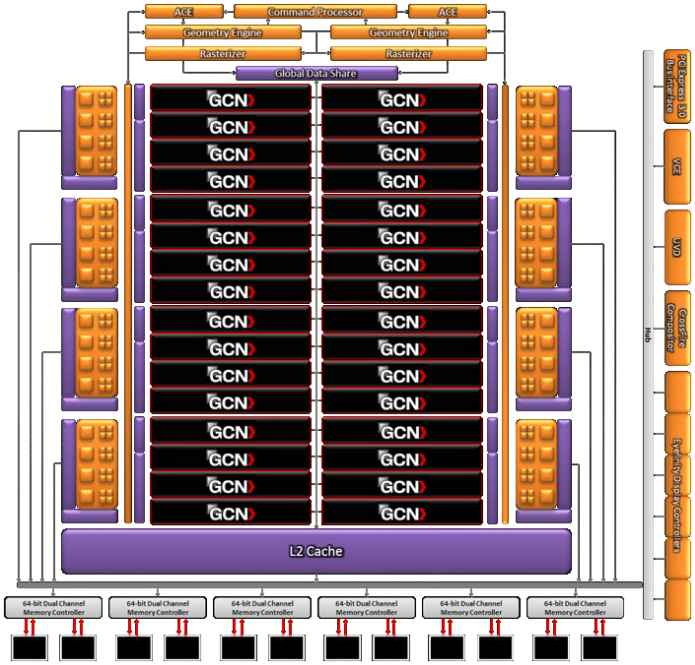


Figure E.1: AMD Radeon HD 7900-Series Architecture

The architecture of the GCN is schematically depicted by figure E.2.

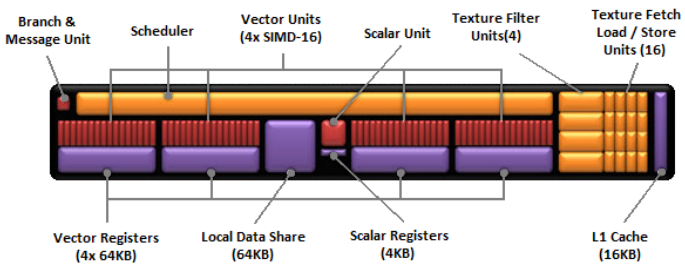


Figure E.2: AMD Radeon HD 7900-Series GCN

Appendix F

Riser Card

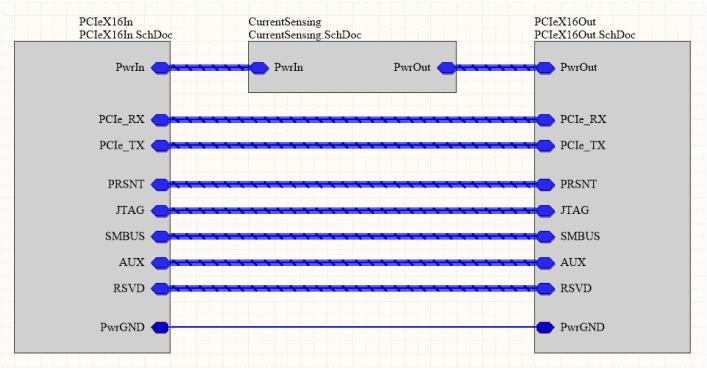


Figure F.1: Riser card block diagram

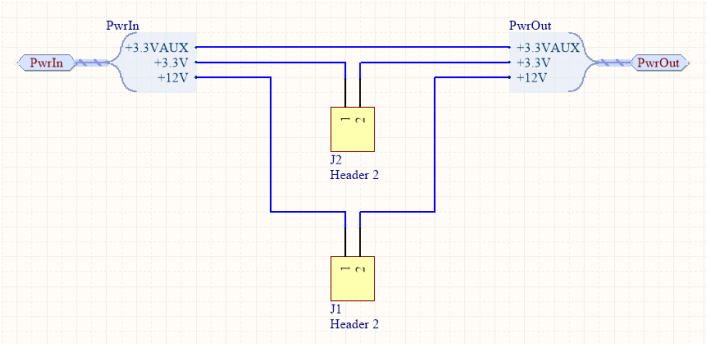


Figure F.2: Riser card schematic - Current Sensing section

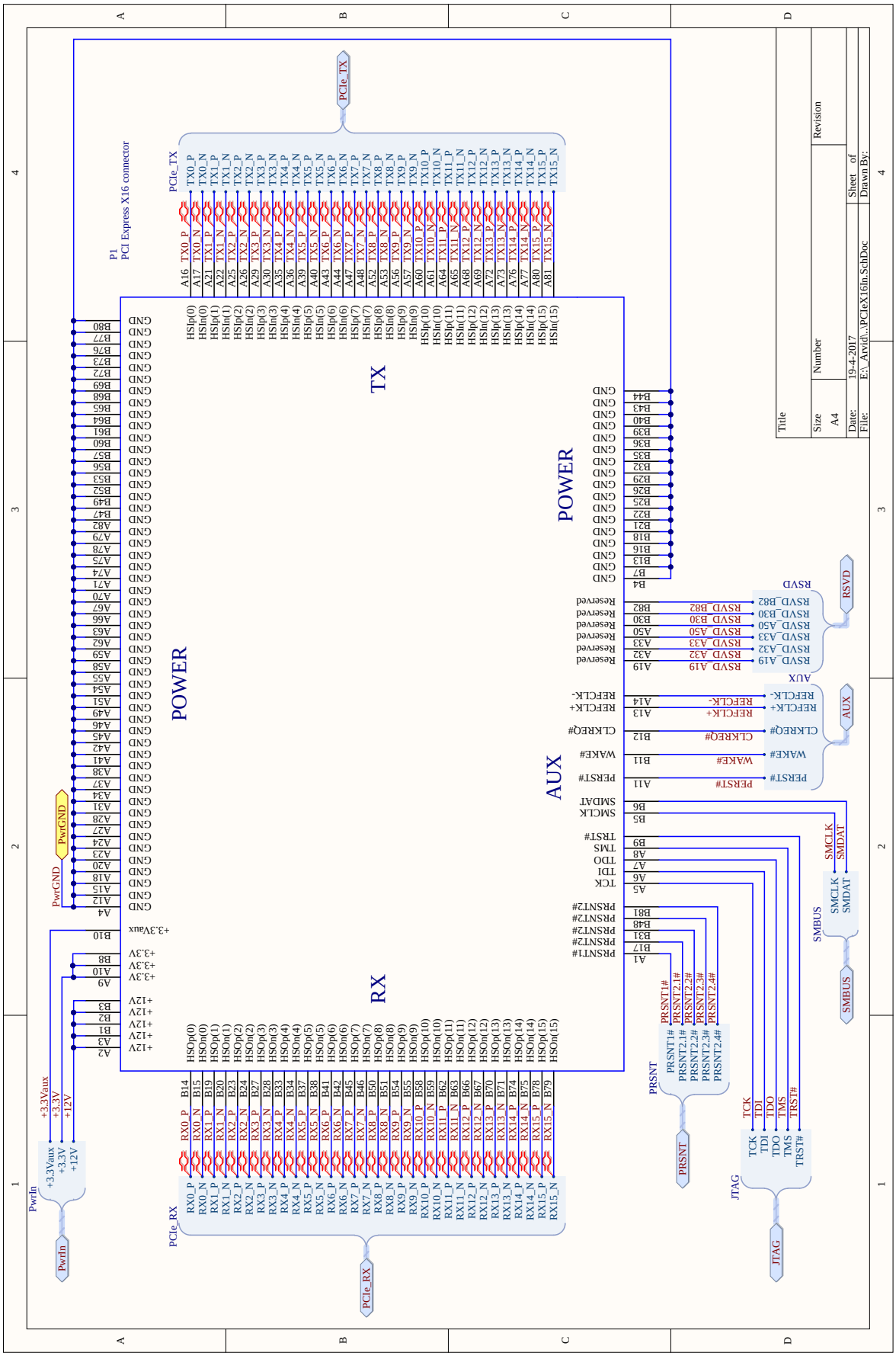


Figure F.3: Riser card schematic - Peripheral Component Interconnect Express (PCIe) Input section

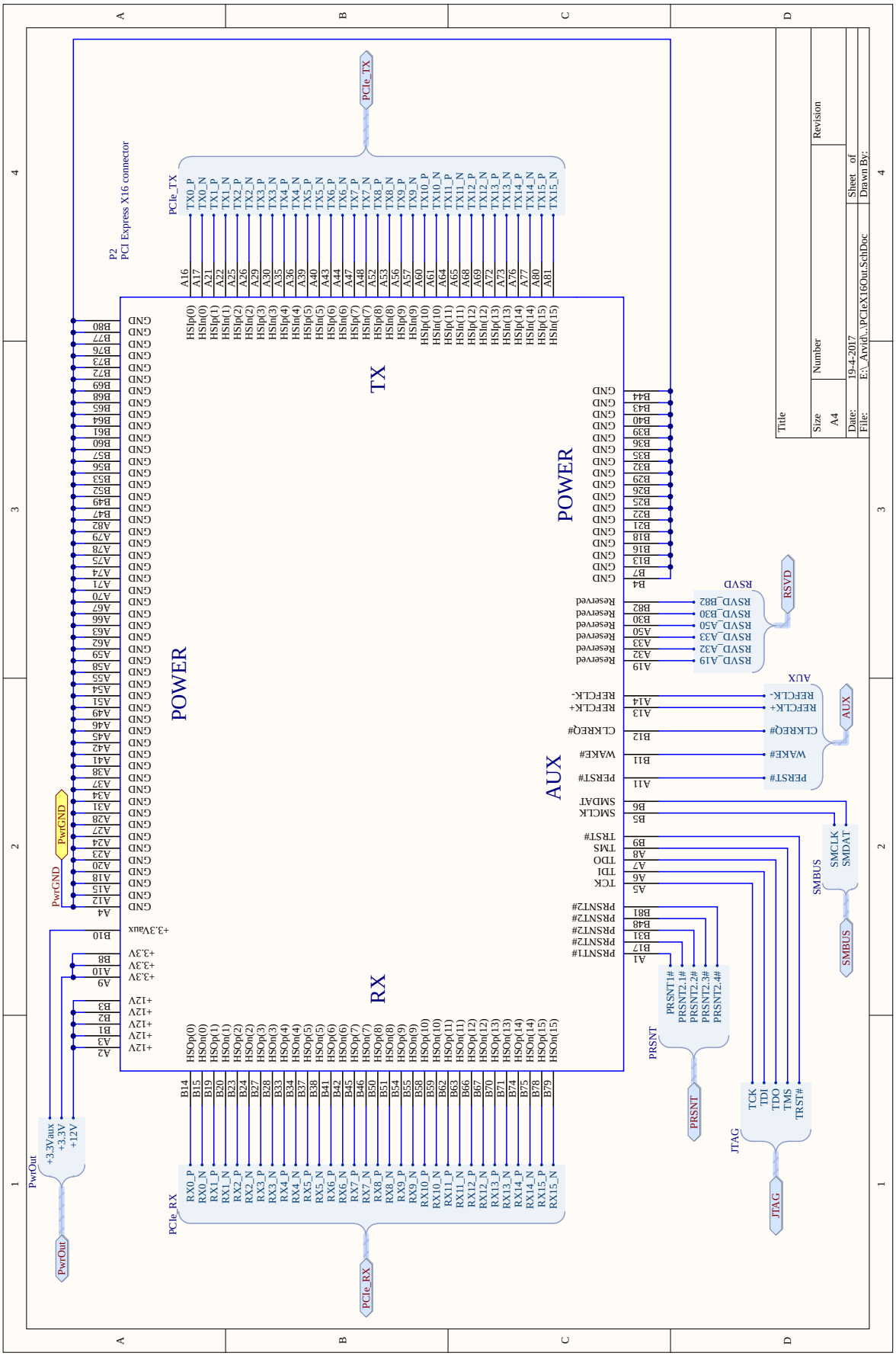


Figure F.4: Riser card schematic - Peripheral Component Interconnect Express (PCIe) Output section

Appendix G

Benchmark C++/OpenCL Code

First the C++ program code.

```
/* ============================================================================
    Copyright (C) 2017 Dinas Software B.V.  All rights reserved
*/

#define DEFAULT_NUM_THREADS 1

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <cstdint>
#include <string.h>
#include <cmath>
#include <vector>
#include "CL/cl.h"
#include "timer.h"
#include <picodrv.h>
#include <pico_errors.h>
#include "gups_reg.h"

using std::cout;
using std::cerr;
using std::endl;
using std::string;
using std::vector;
using std::ifstream;

//////////////////////////////////////////////////////////////////////////////
// MACROS
//////////////////////////////////////////////////////////////////////////////
#define FREE(ptr, free_val) \
if (ptr != free_val)        \
{                           \
free(ptr);                  \
ptr = free_val;             \
}

// we use these macros to make our life a little easier and our code a
// little cleaner some of these macros require certain variables to be
// previously declared in the code where the macro is used e.g. must declare
// u32 as a uint32_t wherever wr(), rd(), or pr() are used.
```

```
#define wr(addr,val) {u32=(val);if((errPico=pico->WriteDeviceAbsolute((addr)
    ↪ ,&u32,4))!=4){fprintf(stderr,"WRITE ERROR4 0x%x: 0x%x(%d)\n",(addr),
    ↪ u32,errPico);}}
#define wl(addr,val) {u64=(val);if((errPico=pico->WriteDeviceAbsolute((addr)
    ↪ ,&u64,8))!=8){fprintf(stderr,"WRITE ERROR8 0x%x: 0x%lx(%d)\n",(addr),
    ↪ u64,errPico);}}
#define rd(addr) {if(0>pico->ReadDeviceAbsolute(addr,&u32,4)){fprintf(stderr
    ↪ ,"READ ERROR\n");}}
#define pr(addr,msg) {if((errPico=pico->ReadDeviceAbsolute(addr,&u32,4))==4)
    ↪ {printf("%20s:  0x%x\n",(msg),u32);}else{fprintf(stderr,"READ ERROR4,
    ↪  %s: 0x%x(%d)\n",(msg),(addr),errPico);}}
#define pl(addr,msg) {if((errPico=pico->ReadDeviceAbsolute(addr,&u64,8))==8)
    ↪ {printf("%20s:  0x%lx\n",(msg),u64);}else{fprintf(stderr,"READ ERROR8
    ↪ , %s: 0x%x(%d)\n",(msg),(addr),errPico);}}

//////////////////////////////////////////////////////////////////////////////
// GLOBALS
//////////////////////////////////////////////////////////////////////////////
struct hostBufferStruct
{
    cl_uint* pInputImage = NULL;
    cl_uint* pOutputImage = NULL;
} hostBuffers;

struct oclBufferStruct
{
    cl_mem pInputCL = NULL;
    cl_mem pOutputCL = NULL;
} oclBuffers;

struct oclHandleStruct
{
    cl_context          context = NULL;
    cl_device_id*       devices = NULL;
    cl_command_queue    queue = NULL;
    cl_program          program = NULL;
    vector<cl_kernel>   kernel;
    cl_event            events[1];
} oclHandles;

struct timerStruct
{
    double dCpuTime;
    double dDeviceTotal;
    double dDeviceKernel;
    CPerfCounter counter;
} timers;

//////////////////////////////////////////////////////////////////////////////
// MAX_ELEMENT
//////////////////////////////////////////////////////////////////////////////
double max_element(double* a, uint32_t num)
{
    double ret = 0.0;

    for (uint8_t aa = 0; aa < num; aa++)
        a[aa] > ret ? ret = a[aa] : ret = ret;

    return ret;
}
```

```
//////////////////////////////////////////////////////////////////////
// UTILITIES
//////////////////////////////////////////////////////////////////////
string FileToString(const string fileName)
{
    ifstream f(fileName.c_str(), ifstream::in | ifstream::binary);

    try
    {
        size_t size;
        char* str;
        string s;

        if (f.is_open())
        {
            size_t fileSize;
            f.seekg(0, ifstream::end);
            size = fileSize = f.tellg();
            f.seekg(0, ifstream::beg);

            str = new char[size + 1];
            if (!str) throw (string("Could not allocate memory. (
                ↪ FileToString)"));

            f.read(str, fileSize);
            f.close();
            str[size] = '\0';

            s = str;
            delete[] str;
            return s;
        }
    }
    catch (std::string msg)
    {
        cerr << "Exception caught in FileToString(): " << msg << endl;
        if (f.is_open())
        f.close();
    }
    catch (...)
    {
        cerr << "Exception caught in FileToString()." << endl;
        if (f.is_open())
        f.close();
    }
    throw (string("FileToString()::Error: Unable to open file "+fileName));
}

//////////////////////////////////////////////////////////////////////
// INITIALIZE AND SHUTDOWN OPENCL
//////////////////////////////////////////////////////////////////////
// Creates Context, Device List and Command Queue
// Loads adn compiles CL file, Link CL Source
// Builds Program and Kernel Objects
//////////////////////////////////////////////////////////////////////
void InitCL()
{
    cl_int errCL;
    size_t deviceListSize;
```

```
//////////////////////////////////////////////////////////////////////
// FIND PLATFORMS
//////////////////////////////////////////////////////////////////////
cl_uint numPlatforms;
cl_platform_id targetPlatform = NULL;

errCL = clGetPlatformIDs(0, NULL, &numPlatforms);
if (errCL != CL_SUCCESS)
    throw (string("InitCL()::Error: Getting number of platforms. (
        ↪ clGetPlatformIDs)"));
if (!(numPlatforms > 0))
    throw (string("InitCL()::Error: No platforms found. (
        ↪ clGetPlatformIDs)"));

cl_platform_id* allPlatforms = (cl_platform_id*)malloc(numPlatforms *
    ↪ sizeof(cl_platform_id));
errCL = clGetPlatformIDs(numPlatforms, allPlatforms, NULL);
if (errCL != CL_SUCCESS)
    throw (string("InitCL()::Error: Getting platform IDs. (
        ↪ clGetPlatformIDs)"));

//////////////////////////////////////////////////////////////////////
// SELECT PLATFORM. (default: first platform)
//////////////////////////////////////////////////////////////////////
targetPlatform = allPlatforms[0];

for (unsigned int i = 0; i < numPlatforms; i++)
{
    char buff[128];
    errCL = clGetPlatformInfo(allPlatforms[i], CL_PLATFORM_VENDOR,
        ↪ sizeof(buff), buff, NULL);
    if (errCL != CL_SUCCESS)
        throw (string("InitCL()::Error: Getting platform info. (
            ↪ clGetPlatformInfo)"));
    if (!strcmp(buff, "Advanced Micro Devices, Inc."))
    {
        targetPlatform = allPlatforms[i];
        break;
    }
}

FREE(allPlatforms, NULL);

//////////////////////////////////////////////////////////////////////
// CREATE OPENCL CONTEXT
//////////////////////////////////////////////////////////////////////
cl_context_properties context_props[3] = { CL_CONTEXT_PLATFORM, (
    ↪ cl_context_properties)targetPlatform, 0 };
oclHandles.context = clCreateContextFromType(context_props,
    ↪ CL_DEVICE_TYPE_GPU, NULL, NULL, &errCL);
if ((errCL != CL_SUCCESS) || (oclHandles.context == NULL))
    throw (string("InitCL()::Error: Creating context. (
        ↪ clCreateContextFromType)"));

//////////////////////////////////////////////////////////////////////
// DETECT OPENCL DEVICES
//////////////////////////////////////////////////////////////////////
errCL = clGetContextInfo(oclHandles.context, CL_CONTEXT_DEVICES, 0, NULL
    ↪ , &deviceListSize);
```

```
if (errCL != CL_SUCCESS)
    throw (string("InitCL()::Error: Getting context info. (
        ↪ clGetContextInfo)"));
if (deviceListSize == 0)
    throw (string("InitCL::Error: No devices found."));

oclHandles.devices = (cl_device_id*)malloc(deviceListSize);
if (oclHandles.devices == 0)
    throw (string("InitCL()::Error: Could not allocate memory."));

errCL = clGetContextInfo(oclHandles.context, CL_CONTEXT_DEVICES,
    ↪ deviceListSize, oclHandles.devices, NULL);
if (errCL != CL_SUCCESS)
    throw (string("InitCL()::Error: Getting device list. (
        ↪ clGetContextInfo)"));

//////////////////////////////////////////////////////////////////
// CREATE OPENCL COMMAND QUEUE
//////////////////////////////////////////////////////////////////
oclHandles.queue = clCreateCommandQueue(oclHandles.context, oclHandles.
    ↪ devices[0], 0, &errCL);
if ((errCL != CL_SUCCESS) || (oclHandles.queue == NULL))
    throw (string("InitCL()::Error: Creating Command queue. (
        ↪ clCreateCommandQueue)"));

//////////////////////////////////////////////////////////////////
// LOAD OPENCL FILE
//////////////////////////////////////////////////////////////////
std::string sourceStr = FileToString("kernel.cl");
const char* source = sourceStr.c_str();
size_t sourceSize[] = { strlen(source) };

oclHandles.program = clCreateProgramWithSource(oclHandles.context, 1, &
    ↪ source, sourceSize, &errCL);
if ((errCL != CL_SUCCESS) || (oclHandles.program == NULL))
throw (string("InitCL()::Error: Loading kernel into program. (
    ↪ clCreateProgramWithSource)"));

//////////////////////////////////////////////////////////////////
// BUILD OPENCL PROGRAM
//////////////////////////////////////////////////////////////////
errCL = clBuildProgram(oclHandles.program, 1, oclHandles.devices, NULL,
    ↪ NULL, NULL);
if ((errCL != CL_SUCCESS) || (oclHandles.program == NULL))
{
    cerr << "InitCL()::Error: clBuildProgram." << endl;

    size_t length;
    errCL = clGetProgramBuildInfo(oclHandles.program, oclHandles.devices
        ↪ [0], CL_PROGRAM_BUILD_LOG, 0, NULL, &length);
    if (errCL != CL_SUCCESS)
        throw (string("InitCL()::Error: Getting Program build info. (
            ↪ clGetProgramBuildInfo)"));

    char* buff = (char*)malloc(length);
    errCL = clGetProgramBuildInfo(oclHandles.program, oclHandles.devices
        ↪ [0], CL_PROGRAM_BUILD_LOG, length, buff, NULL);
    if (errCL != CL_SUCCESS)
        throw (string("InitCL()::Error: Getting Program build info. (
            ↪ clGetProgramBuildInfo)"));
```

```
        cerr << buff << endl;
        free(buff);
        throw (string("InitCL()::Error: Building Program. (clBuildProgram)")
            ↪ );
    }

    ////////////////////////////////////////////////////////////////////////
    // CREATE OPENCL KERNEL OBJECT
    ////////////////////////////////////////////////////////////////////////
    for (int iKernel = 0; iKernel < 1; iKernel++)
    {
        cl_kernel kernel = clCreateKernel(oclHandles.program, "PlusOne", &
            ↪ errCL);
        if (errCL != CL_SUCCESS)
            throw (string("InitCL()::Error: Creating Kernel. (clCreateKernel
                ↪ )"));

        oclHandles.kernel.push_back(kernel);
    }
}

////////////////////////////////////////////////////////////////////////
// SET OPENCL KERNEL ARGUMENTS
////////////////////////////////////////////////////////////////////////
void SetKernelArgs(int iKernel)
{
    cl_int errCL;

    errCL = clSetKernelArg(oclHandles.kernel[iKernel], 0, sizeof(cl_mem), (
        ↪ void*)&oclBuffers.pInputCL);
    if (errCL != CL_SUCCESS)
        throw (string("SetKernelArgs()::Error: Setting kernel argument (
            ↪ input image)"));
}

////////////////////////////////////////////////////////////////////////
// ENQUEUE OPENCL KERNEL
////////////////////////////////////////////////////////////////////////
void EnqueueKernel(int iKernel, const size_t globalNDWorkSize, const size_t
    ↪ localNDWorkSize, bool bBlocking = false)
{
    cl_int errCL;

    errCL = clEnqueueNDRangeKernel(oclHandles.queue, oclHandles.kernel[
        ↪ iKernel], 1, NULL, &globalNDWorkSize, &localNDWorkSize, 0, NULL,
        ↪ &oclHandles.events[0]);
    if (errCL != CL_SUCCESS)
        throw (string("EnqueueKernel()::Error: Enqueueing kernel onto
            ↪ command queue. (clEnqueueNDRangeKernel)"));

    if (bBlocking)
    {
        errCL = clWaitForEvents(1, &oclHandles.events[0]);
        if (errCL != CL_SUCCESS)
            throw (string("EnqueueKernel()::Error: Waiting for kernel run to
                ↪  finish. (clWaitForEvents)"));
    }
}
```

```
////////////////////////////////////////////////////////////////////////////////
// MAIN
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char* argv[])
{
    try
    {
        ////////////////////////////////////////////////////////////////////////
        // GLOBALS
        ////////////////////////////////////////////////////////////////////////
        cl_int errCL;

        PicoDrv* pico;
        int      errPico;

        const uint8_t samples = 15;
        const uint8_t startSample = 12;

        uint32_t sizes[samples];
        int repeats = 10;

        double sendGPUTimes[samples];
        double sendHMCTimes[samples];
        double memoryTimesGPU[samples];
        double memoryTimesHost[samples];
        double memoryTimesHMC[samples];

        double sendGPUBandwidth[samples];
        double sendHMCBandwidth[samples];
        double memoryBandwidthGPU[samples];
        double memoryBandwidthHost[samples];
        double memoryBandwidthHMC[samples];

        InitCL();

        for (int ii = 0; ii < samples; ii++)
        {
            sizes[ii] = (uint32_t)pow(2.0, (startSample + ii));
            sendGPUTimes[ii] = 9999999999.0;
            sendHMCTimes[ii] = 9999999999.0;
            memoryTimesHost[ii] = 9999999999.0;
            memoryTimesGPU[ii] = 9999999999.0;
            memoryTimesHMC[ii] = 9999999999.0;
        }

        ////////////////////////////////////////////////////////////////////////
        // RUN HOST TO GPU BENCHMARK
        ////////////////////////////////////////////////////////////////////////
        cout << "%% Test Host/GPU Bandwidth (" << repeats << ")" << endl;

        for (int ii = 0; ii < samples; ii++)
        {
            uint8_t* data = (uint8_t*)malloc(sizes[ii]*sizeof(uint8_t));

            for (int tt = 0; tt < (int)sizes[ii]; tt++)
                data[tt] = rand() % 256;

            oclBuffers.pInputCL = clCreateBuffer(oclHandles.context,
                ↪ CL_MEM_READ_WRITE, sizes[ii] * sizeof(uint8_t), NULL, &
                ↪ errCL);
```

```
    if ((errCL != CL_SUCCESS) || (oclBuffers.pInputCL == NULL))
        throw (string("InitCLBuffers()::Error: Could not create
            ↪ oclBuffers.pInputCL. (clCreateBuffer)"));

    for (int rr = 0; rr < repeats; rr++)
    {
        timers.counter.Reset();
        timers.counter.Start();

        errCL = clEnqueueWriteBuffer(oclHandles.queue, oclBuffers.
            ↪ pInputCL, CL_TRUE, 0, sizes[ii] * sizeof(uint8_t),
            ↪ data, 0, NULL, NULL);
        errCL = clFlush(oclHandles.queue);
        errCL = clFinish(oclHandles.queue);

        timers.counter.Stop();
        sendGPUTimes[ii] = min(sendGPUTimes[ii], timers.counter.
            ↪ GetElapsedTime());
    }
    sendGPUBandwidth[ii] = (sizes[ii] / sendGPUTimes[ii]) /
        ↪ 1000000000;

    if (oclBuffers.pInputCL != NULL)
    {
        errCL = clReleaseMemObject(oclBuffers.pInputCL);
        if (errCL != CL_SUCCESS)
            cerr << "ReleaseCLBuffers()::Error: oclBuffers.pInputCL.
                ↪  (clReleaseMemObject)" << endl;
        oclBuffers.pInputCL = NULL;
    }

    FREE(data, NULL);
}

cout << "sendGPUBandwidth = [ ";
for (int aa = 0; aa < samples; aa++)
    cout << sendGPUBandwidth[aa] << " ";
cout << "];" << endl;

cout << "% Peak GPU send speed is " << max_element(sendGPUBandwidth,
    ↪  samples) << " GB/s" << endl;

//////////////////////////////////////////////////////////////////////
// RUN HOST TO HMC BENCHMARK
//////////////////////////////////////////////////////////////////////
cout << "%% Test Host/HMC Bandwidth (" << repeats << ")" << endl;

const char* bitFileName = "~/hmc/hmc_benchmark/hmc_benchmark/
    ↪ firmware/hmc_benchmark.bit";

errPico = RunBitFile(bitFileName, &pico);
if (errPico < 0) exit(1);

int myStream = pico->CreateStream(116);
if (myStream < 0) exit(1);

unsigned char* data = (unsigned char*)malloc(sizes[samples - 1] *
    ↪ sizeof(unsigned char));

for (int tt = 0; tt < (int)sizes[samples - 1]; tt++)
```

```
    data[tt] = rand() % 256;

for (int ii = 0; ii < samples; ii++)
{
    cout << "% Sending " << sizes[ii] << endl;
    for (int rr = 0; rr < repeats; rr++)
    {
        timers.counter.Reset();
        timers.counter.Start();
        if ((errPico = pico->WriteStream(myStream, data, sizes[ii]))
            ↪ < 0)
            throw (string("WriteStream: " + errPico));
        timers.counter.Stop();
        sendHMCTimes[ii] = min(sendHMCTimes[ii], timers.counter.
            ↪ GetElapsedTime());
    }
    sendHMCBandwidth[ii] = (sizes[ii] / sendHMCTimes[ii]) /
        ↪ 1000000000;
}

FREE(data, NULL);

cout << "sendHMCBandwidth = [ ";
for (int aa = 0; aa < samples; aa++)
    cout << sendHMCBandwidth[aa] << " ";
cout << "];" << endl;

cout << "% Peak HMC send speed is " << max_element(sendHMCBandwidth,
    ↪ samples) << " GB/s" << endl;

////////////////////////////////////////////////////////////////////////
// RUN HMC BENCHMARK / Memory-Intensive Operations
////////////////////////////////////////////////////////////////////////
cout << "%% Test HMC Bandwidth (" << repeats << ")" << endl;

const uint8_t sizeOfDouble = 8;
const uint8_t readWritesPerElement = 2;

uint32_t u32;
uint32_t gups_base = REG_GUPS_BASE_ADDR(1);

for (int ii = 0; ii < samples; ii++)
{
    cout << "sending " << sizes[ii] << " bytes" << endl;

    for (int rr = 0; rr < repeats; rr++)
    {
        wr(REG_GUPS_BANDWIDTH_EN(gups_base), 0);
        wr(REG_GUPS_RD_BW_COUNTER(gups_base), sizes[ii]);
        wr(REG_GUPS_WR_BW_COUNTER(gups_base), sizes[ii]);
        wr(REG_GUPS_BANDWIDTH_EN(gups_base), 1);

        do
        {
            rd(REG_GUPS_BANDWIDTH_EN(gups_base));
        } while (u32 == 1);

        rd(REG_GUPS_BW_TICKS(gups_base));
        cout << "% Ticks: " << u32 << endl;
```

```
            memoryTimesHMC[ii] = min(memoryTimesHMC[ii], u32);
        }

        double tmpMemTimeHMC = memoryTimesHMC[ii] / 0.0000000008;
        memoryBandwidthHMC[ii] = readWritesPerElement * (sizes[ii] /
            ↪ tmpMemTimeHMC) / 1000000000;
    }

    cout << "memoryBandwidthHMC = [ ";
    for (int aa = 0; aa < samples; aa++)
        cout << memoryBandwidthHMC[aa] << " ";
    cout << "];" << endl;

    cout << "% Peak read/write speed on the HMC is " << max_element(
        ↪ memoryBandwidthHMC, samples) << " GB/s" << endl;

    //////////////////////////////////////////////////////////////////
    // RUN GPU BENCHMARK / Memory-Intensive Operations - 1 Kernel
    //////////////////////////////////////////////////////////////////
    cout << endl << "%% Test GPU Bandwidth - Single Kernel (" << repeats
        ↪ << ")" << endl;

    for (int ii = 0; ii < samples; ii++)
    {
        int numElements = sizes[ii]; // / sizeOfDouble;
        int numOfKernels = 1;

        uint8_t* data = (uint8_t*)calloc(numElements, sizeof(double));
        oclBuffers.pInputCL = clCreateBuffer(oclHandles.context,
            ↪ CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, numElements *
            ↪ sizeof(double), data, &errCL);
        if ((errCL != CL_SUCCESS) || (oclBuffers.pInputCL == NULL))
            throw (string("InitCLBuffers()::Error: Could not create
                ↪ oclBuffers.pInputCL. (clCreateBuffer)"));

        for (int rr = 0; rr < repeats; rr++)
        {
            errCL = clEnqueueReadBuffer(oclHandles.queue, oclBuffers.
                ↪ pInputCL, CL_TRUE, 0, numElements * sizeof(double),
                ↪ data, 0, NULL, NULL);
            errCL = clFlush(oclHandles.queue);
            errCL = clFinish(oclHandles.queue);

            SetKernelArgs(0);
            clFinish(oclHandles.queue);

            timers.counter.Reset();
            timers.counter.Start();

            EnqueueKernel(0, numElements, numOfKernels, true);
            clFinish(oclHandles.queue);

            timers.counter.Stop();
            memoryTimesGPU[ii] = min(memoryTimesGPU[ii], (timers.counter
                ↪ .GetElapsedTime() / 100));

            errCL = clEnqueueWriteBuffer(oclHandles.queue, oclBuffers.
                ↪ pInputCL, CL_TRUE, 0, numElements * sizeof(double),
                ↪ data, 0, NULL, NULL);
            errCL = clFlush(oclHandles.queue);
```

```
            errCL = clFinish(oclHandles.queue);
        }
        memoryBandwidthGPU[ii] = readWritesPerElement * (sizes[ii] /
            ↪ memoryTimesGPU[ii]) / 1000000000;

        if (oclBuffers.pInputCL != NULL)
        {
            errCL = clReleaseMemObject(oclBuffers.pInputCL);
            if (errCL != CL_SUCCESS)
                cerr << "ReleaseCLBuffers()::Error: oclBuffers.pInputCL.
                    ↪ (clReleaseMemObject)" << endl;
            oclBuffers.pInputCL = NULL;
        }

        FREE(data, NULL);
    }

    cout << "memoryBandwidthGPU1 = [ ";
    for (int aa = 0; aa < samples; aa++)
        cout << memoryBandwidthGPU[aa] << " ";
    cout << "];" << endl;

    cout << "% Peak read/write speed on the GPU1 is " << max_element(
        ↪ memoryBandwidthGPU, samples) << " GB/s" << endl;

    //////////////////////////////////////////////////////////////////////
    // RUN GPU BENCHMARK / Memory-Intensive Operations - 128 Kernels
    //////////////////////////////////////////////////////////////////////
    cout << endl << "%% Test GPU Bandwidth - 128 Kernels (" << repeats
        ↪ << ")" << endl;

    for (int ii = 0; ii < samples; ii++)
    {
        int numElements = sizes[ii];
        int numOfKernels = 128;

        uint8_t* data = (uint8_t*)calloc(numElements, sizeof(double));
        oclBuffers.pInputCL = clCreateBuffer(oclHandles.context,
            ↪ CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, numElements *
            ↪ sizeof(double), data, &errCL);
        if ((errCL != CL_SUCCESS) || (oclBuffers.pInputCL == NULL))
            throw (string("InitCLBuffers()::Error: Could not create
                ↪ oclBuffers.pInputCL. (clCreateBuffer)"));

        for (int rr = 0; rr < repeats; rr++)
        {
            errCL = clEnqueueReadBuffer(oclHandles.queue, oclBuffers.
                ↪ pInputCL, CL_TRUE, 0, numElements * sizeof(double),
                ↪ data, 0, NULL, NULL);
            errCL = clFlush(oclHandles.queue);
            errCL = clFinish(oclHandles.queue);

            SetKernelArgs(0);
            clFinish(oclHandles.queue);

            timers.counter.Reset();
            timers.counter.Start();

            EnqueueKernel(0, numElements, numOfKernels, true);
            clFinish(oclHandles.queue);
```

```
            timers.counter.Stop();
            memoryTimesGPU[ii] = min(memoryTimesGPU[ii], (timers.counter
                ↪ .GetElapsedTime() / 100));

            errCL = clEnqueueWriteBuffer(oclHandles.queue, oclBuffers.
                ↪ pInputCL, CL_TRUE, 0, numElements * sizeof(double),
                ↪ data, 0, NULL, NULL);
            errCL = clFlush(oclHandles.queue);
            errCL = clFinish(oclHandles.queue);

        }
        memoryBandwidthGPU[ii] = readWritesPerElement * (sizes[ii] /
            ↪ memoryTimesGPU[ii]) / 1000000000;

        if (oclBuffers.pInputCL != NULL)
        {
            errCL = clReleaseMemObject(oclBuffers.pInputCL);
            if (errCL != CL_SUCCESS)
                cerr << "ReleaseCLBuffers()::Error: oclBuffers.pInputCL.
                    ↪  (clReleaseMemObject)" << endl;
            oclBuffers.pInputCL = NULL;
        }

        FREE(data, NULL);
    }

    cout << "memoryBandwidthGPU128 = [ ";
    for (int aa = 0; aa < samples; aa++)
        cout << memoryBandwidthGPU[aa] << " ";
    cout << "];" << endl;

    cout << "% Peak read/write speed on the GPU128 is " << max_element(
        ↪ memoryBandwidthGPU, samples) << " GB/s" << endl;

    ////////////////////////////////////////////////////////////////////
    // RUN CPU BENCHMARK
    ////////////////////////////////////////////////////////////////////
    cout << endl << "%% Test Host Bandwidth (" << repeats << ")" << endl
        ↪ ;

    for (int ii = 0; ii < samples; ii++)
    {
        int numElements = sizes[ii];

        for (int rr = 0; rr < repeats; rr++)
        {
            uint8_t* hostData = (uint8_t*)calloc(numElements, sizeof(
                ↪ double));

            timers.counter.Reset();
            timers.counter.Start();

            for (int jj = 0; jj < 100; jj++)
            {
                for (int ll = 0; ll < numElements; ll++)
                hostData[ll] += 1;
            }

            timers.counter.Stop();
```

```
                memoryTimesHost[ii] = min(memoryTimesHost[ii], (timers.
                    ↪ counter.GetElapsedTime() / 100));

                FREE(hostData, NULL);
            }
            memoryBandwidthHost[ii] = readWritesPerElement * (sizes[ii] /
                ↪ memoryTimesHost[ii]) / 1000000000;
        }

        cout << "memoryBandwidthHost = [ ";
        for (int aa = 0; aa < samples; aa++)
            cout << memoryBandwidthHost[aa] << " ";
        cout << "];" << endl;

        cout << "% Peak write speed on the host is " << max_element(
            ↪ memoryBandwidthHost, samples) << " GB/s" << endl;
    }
    catch (std::string msg)
    {
        cerr << "Exception caught in main(): " << msg << endl;
        return EXIT_FAILURE;
    }
    catch (...)
    {
        cerr << "Exception caught in main()" << endl;
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Finally the `OpenCL kernel` program code.

```
//KERNEL_SIMPLE

////////////////////////////////////////////////////////////////////////////
// PLUS
////////////////////////////////////////////////////////////////////////////
__kernel void PlusOne(__global  uint* pInput)
{
    const int nX = get_global_id(0);

    for (int jj = 0; jj < 100; jj++)
    {
        pInput[nX] += 1;
    }
}
//KERNEL_SIMPLE
```

Appendix H

Image Denoising OpenCL Code

OpenCL kernel code for removing Impulse noise from an image.

```
//////////////////////////////////////////////////////////////////////////////
// MEDIAN FILTER SORT
//////////////////////////////////////////////////////////////////////////////
void medSort(uint* a, uint* b)
{
    if (*a > *b)
    {
        uint temp = *b;
        *b = *a;
        *a = temp;
    }
}

__kernel void MedianFilter(const __global  uint* pInput, __global uint*
    ↪ pOutput, const int MaskWidth)
{
    const int nWidth = get_global_size(0);
    const int nHeight = get_global_size(1);
    const int xOut = get_global_id(0);
    const int yOut = get_global_id(1);
    const int yInTopLeft = yOut;
    const int xInTopLeft = xOut;

    int mask = 0xFF;
    int result = 0;

    for (int channel = 0; channel < 3; channel++)
    {
        int i[9];

        for (int r = 0; r < MaskWidth; r++)
        {
            int idxFtmp = r * MaskWidth;
            int yIn = yInTopLeft + r;
            int idxIntmp = yIn * (nWidth + MaskWidth - 1) + xInTopLeft;

            for (int c = 0; c < MaskWidth; c++)
            {
                int idxIn = idxIntmp + c;
                i[idxFtmp + c] = pInput[idxIn] & mask;
            }
        }
```

```
        // Median Sort
        medSort(&i[1], &i[2]);
        medSort(&i[4], &i[5]);
        medSort(&i[7], &i[8]);
        medSort(&i[0], &i[1]);
        medSort(&i[3], &i[4]);
        medSort(&i[6], &i[7]);
        medSort(&i[1], &i[2]);
        medSort(&i[4], &i[5]);
        medSort(&i[7], &i[8]);
        medSort(&i[0], &i[3]);
        medSort(&i[5], &i[8]);
        medSort(&i[4], &i[7]);
        medSort(&i[3], &i[6]);
        medSort(&i[1], &i[4]);
        medSort(&i[2], &i[5]);
        medSort(&i[4], &i[7]);
        medSort(&i[4], &i[2]);
        medSort(&i[6], &i[4]);
        medSort(&i[4], &i[2]);

        // result is i4
        result |= i[4];
        mask <<= 8;
    }

    const int idxOut = yOut * nWidth + xOut;
    pOutput[idxOut] = result;
}
```

Appendix I

MATLAB Denoise Code

MATLAB code for removing Impulse noise from an image.

```
%% Denoise Function
function denoise

% filter windows size from 3x3 to 25x25
% odd sizes only
window = 3:2:25;

I = imread('hawk.bmp');
J = imnoise(I,'salt & pepper',0.2);

for ii=1:numel(window)
% filter each channel separately
r = medfilt2(J(:,:,1), [window(ii) window(ii)]);
g = medfilt2(J(:,:,2), [window(ii) window(ii)]);
b = medfilt2(J(:,:,3), [window(ii) window(ii)]);

% reconstruct the image from r,g,b channels
K = cat(3, r, g, b);
end
```

Index

Bibliography

[1] PCIe FLASH and Stacked DRAM for Scalable Systems. [Online]. Available: http://storageconference.us/2013/Presentations/Jeddeloh.pdf [Accessed: 29-01-2017]

[2] Micron Technology, Inc. - Image Gallery. [Online]. Available: https://www.micron.com/about/news-and-events/media-relations/gallery [Accessed: 29-01-2017]

[3] Hybrid Memory Cube 30G-VSR Specification, Rev. 2.1. [Online]. Available: http://www.hybridmemorycube.org/ [Accessed: 29-01-2017]

[4] SKA Telescope. [Online]. Available: https://www.skatelescope.org/ [Accessed: 30-12-2016]

[5] ASTRON - The Netherlands Institute for Radio Astronomy. [Online]. Available: http://www.astron.nl [Accessed: 30-12-2016]

[6] Central Signal Processor - SKA Telescope. [Online]. Available: https://www.skatelescope.org/csp/ [Accessed: 30-12-2016]

[7] Science Data Processor - SKA Telescope. [Online]. Available: https://www.skatelescope.org/sdp/ [Accessed: 30-12-2016]

[8] Hybrid Memory Cube consortium. [Online]. Available: http://www.hybridmemorycube.org/ [Accessed: 22-08-2016]

[9] J. Ahn, S. Yoo, and K. Choi, "Dynamic power management of off-chip links for hybrid memory cubes," in *51st ACM/EDAC/IEEE Design Automation Conference (DAC), New York, USA*, 2014, pp. 1–6.

[10] S. Wang, Y. Song, M. N. Bojnordi, and E. Ipek, "Enabling energy efficient hybrid memory cube systems with erasure codes," in *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2015, pp. 67–72.

[11] Paul Rosenfeld, "Performance exploration of the Hybrid Memory Cube," Ph.D. dissertation, University of Maryland, 2014.

[12] Maohua Zhu and Youwei Zhuo and Chao Wang and Wenguang Chen and Yuan Xie, "Performance Evaluation and Optimization of HBM-Enabled GPU for Data-intensive Applications," in *Proceedings of the 2017 Design, Automation and Test in Europe (DATE)*, 2017. [Online]. Available: https://www.date-conference.com/proceedings-archive/2017/html/0018.html [Accessed: 19-04-2017]

[13] High-Bandwidth Memory (HBM) REINVENTING MEMORY TECHNOLOGY. [Online]. Available: https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf [Accessed: 19-04-2017]

[14] AMD Radeon R9 Fury X. [Online]. Available: http://products.amd.com/en-us/search/Desktop-Graphics/AMD-Radeon%E2%84%A2-R9-Series/AMD-Radeon%E2%84%A2-R9-Fury-Series/AMD-Radeon%E2%84%A2-R9-Fury-X/67 [Accessed: 19-04-2017]

[15] Micron Technology, Inc. - Hybrid Memory Cube — Memory and Storage. [Online]. Available: https://www.micron.com/products/hybrid-memory-cube [Accessed: 08-12-2016]

[16] AC-510 UltraScale-based SuperProcessor with Hybrid Memory Cube. [Online]. Available: http://picocomputing.com/ac-510-superprocessor-module/ [Accessed: 08-12-2016]

[17] Micron Technology, Inc. - High Performance On Package Memory. [Online]. Available: https://www.micron.com/products/hybrid-memory-cube/high-performance-on-package-memory [Accessed: 08-12-2016]

[18] Intel Products (Formerly Knights Landing). [Online]. Available: http://ark.intel.com/products/codename/48999/Knights-Landing [Accessed: 08-12-2016]

[19] S. Skorobogatov, "Low temperature data remanence in static RAM," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-536, Jun. 2002. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf [Accessed: 24-01-2017]

[20] A. van den Brink, "Evaluating Energy Efficiency using Hybrid Memory Cube Technology," Drienerlolaan 5, 7522 NB Enschede, Dec. 2016.

[21] M. Facchini, T. Carlson, A. Vignon, M. Palkovic, F. Catthoor, W. Dehaene, L. Benini, and P. Marchal, "System-level power/performance evaluation of 3d stacked drams for mobile applications," in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 923–928.

[22] "GDDR5 SGRAM Introduction," Micron. [Online]. Available: https://www.google.nl/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0ahUKEwjH5unLuI_TAhUJOBoKHeuTCBsQFggaMAA&url=https%3A%2F%2Fwww.micron.com%2F~%2Fmedia%2Fdocuments%2Fproducts%2Ftechnical-note%2Fdram%2Ftned01_gddr5_sgram_introduction.pdf&usg=AFQjCNGRHgeEVD-yey3XKTd--6zWTQooaw&cad=rja [Accessed: 06-04-2017]

[23] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998590 [Accessed: 06-04-2017]

[24] Impulse Noise. [Online]. Available: https://en.wikipedia.org/wiki/Salt-and-pepper_noise [Accessed: 18-01-2017]

[25] Maxwell's 3^{rd} equation. Faraday's law. [Online]. Available: http://www.maxwells-equations.com/faraday/faradays-law.php [Accessed: 20-09-2016]

[26] Maxwell's 4^{th} equation. Ampere's law. [Online]. Available: http://www.maxwells-equations.com/ampere/amperes-law.php [Accessed: 20-09-2016]

[27] Hall Effect Sensor and How Magnets Make It Works. [Online]. Available: http://www.electronics-tutorials.ws/electromagnetism/hall-effect.html [Accessed: 20-09-2016]

[28] ACS715: Hall Effect-Based Linear Current Sensor IC. [Online]. Available: http://www.allegromicro.com/en/Products/Current-Sensor-ICs/Zero-To-Fifty-Amp-Integrated-Conductor-Sensor-ICs/ACS715.aspx [Accessed: 20-09-2016]

[29] W. L. Bircher and L. K. John, “Complete System Power Estimation: A Trickle-Down Approach Based on Performance Events,” in *2007 IEEE International Symposium on Performance Analysis of Systems Software*, April 2007, pp. 158–168.

[30] J. Janzen, Ed., *Calculating Memory System Power for DDR SDRAM*, vol. 10, 2001. [Online]. Available: http://www.ece.umd.edu/class/enee759h.S2005/references/dl201.pdf [Accessed: 22-05-2017]

[31] C. Isci and M. Martonosi, “Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data,” in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 93–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956567 [Accessed: 24-05-2017]

[32] S. Hong and H. Kim, “An Integrated GPU Power and Performance Model,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA ’10. New York, NY, USA: ACM, 2010, pp. 280–289. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815998 [Accessed: 24-05-2017]

[33] EX-750 Backplane for the AC-510. [Online]. Available: http://picocomputing.com/products/backplanes/ex-750/ [Accessed: 22-08-2016]

[34] A PCIe Riser Card. [Online]. Available: https://en.wikipedia.org/wiki/Riser_card [Accessed: 20-09-2016]

[35] Arduino Nano. [Online]. Available: https://www.arduino.cc/en/Main/ArduinoBoardNano [Accessed: 14-06-2017]

[36] The decimal vs binary multiples of bytes. [Online]. Available: https://en.wikipedia.org/wiki/Mebibyte [Accessed: 11-07-2017]

[37] V. Aggarwal, Y. Sabharwal, R. Garg, and P. Heidelberger, “Hpcc randomaccess benchmark for next generation supercomputers,” in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–11.

[38] The Pico Computing Frameworkthe Ghost in the Machine. [Online]. Available: http://picocomputing.com/products/framework/ [Accessed: 23-06-2017]

[39] S. Mittal, “A survey of techniques for approximate computing,” *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2893356 [Accessed: 17-07-2017]

[40] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akessonn, N. Wehn, and K. Goossens. DRAMPower: Open-source DRAM Power and Energy Estimation Tool. [Online]. Available: http://www.es.ele.tue.nl/drampower [Accessed: 19-06-2017]

[41] Probability Density Function. [Online]. Available: https://en.wikipedia.org/wiki/Probability_density_function [Accessed: 18-01-2017]

[42] Gaussian Noise. [Online]. Available: https://en.wikipedia.org/wiki/Gaussian_noise [Accessed: 18-01-2017]

[43] White Noise. [Online]. Available: https://en.wikipedia.org/wiki/White_noise [Accessed: 18-01-2017]

[44] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[45] Xilinx High Speed Serial Transceivers. [Online]. Available: https://www.xilinx.com/products/technology/high-speed-serial.html [Accessed: 19-04-2017]

[46] S. Ziegler, R. C. Woodward, H. H.-C. Iu, and L. J. Borle, "Current sensing techniques: A review," *IEEE Sensors Journal*, vol. 9, no. 4, pp. 354–376, Apr. 2009.