Adding Fault Tolerance to OpenCL

Through Redundant Heterogeneous Computing

Robin Alexander Bijl

```
memobjs[2] = clCreateBuffer (context, flags: CL_MEM_READ_WRITE,
                             size: sizeof (cl_uint) * opt.input_size, host_ptr: NULL, errcode_ret: &err);
CHECK_CL_ERROR2 (err);

err = clEnqueueWriteBuffer (command_queue: cmd_queue, buffer: memobjs[0], blocking_write: CL_TRUE, offset: 0,
                            size: opt.input_size * sizeof (cl_uint), ptr: srcA, num_events_in_wait_list: 0, event_wait_list: NULL, event: NULL);
CHECK_CL_ERROR2 (err);

err = clEnqueueWriteBuffer (command_queue: cmd_queue, buffer: memobjs[1], blocking_write: CL_TRUE, offset: 0,
                            size: opt.input_size * sizeof (cl_uint), ptr: srcB, num_events_in_wait_list: 0, event_wait_list: NULL, event: NULL);
CHECK_CL_ERROR2 (err);

program = clCreateProgramWithBuiltInKernels (
    context, num_devices, device_list: devices, kernel_names: "pocl.add.i32", errcode_ret: &err);
CHECK_CL_ERROR2 (err);

err = clBuildProgram (program, num_devices, device_list: devices, options: NULL, pfn_notify: NULL, user_data: NULL);
CHECK_OPENCL_ERROR_IN ("clBuildProgram");

kernel = clCreateKernel (program, kernel_name: "pocl.add.i32", errcode_ret: &err);
CHECK_CL_ERROR2 (err);

err = clSetKernelArg (kernel, arg_index: 0, arg_size: sizeof (cl_mem), arg_value: (void *) &memobjs[0]);
CHECK_CL_ERROR2 (err);

err = clSetKernelArg (kernel, arg_index: 1, arg_size: sizeof (cl_mem), arg_value: (void *) &memobjs[1]);
CHECK_CL_ERROR2 (err);

err = clSetKernelArg (kernel, arg_index: 2, arg_size: sizeof (cl_mem), arg_value: (void *) &memobjs[2]);
CHECK_CL_ERROR2 (err);

global_work_size[0] = opt.input_size;

err = clEnqueueNDRangeKernel (command_queue: cmd_queue, kernel, work_dim: 1, global_work_offset: NULL, global_work_size,
                              local_work_size: NULL, num_events_in_wait_list: 0, event_wait_list: NULL, event: NULL);
```

TU Delft

Adding Fault Tolerance to OpenCL

Through Redundant Heterogeneous Computing

by

Robin Alexander Bijl

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday June 30, 2023 at 2:00 PM.

Student number:	4360192	
Project duration:	May 3, 2022 – May 1, 2023	
Thesis committee:	Assoc. prof. dr. ir. Zaid Al-Ars,	TU Delft, supervisor
	Assist. prof. dr. Christoph Lofi,	TU Delft
	Assoc. prof. dr. Pekka Jääskeläinen,	Tampere University

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

This thesis is the final part of my efforts to get a master's degree in Computer Engineering at the Delft University of Technology. It has been an exciting journey and I never imagined that it would take me to Finland. First off, I would like to thank my supervisor Dr. Ir. Zaid Al-Ars for his guidance, feedback and most of all critical questions during the process of writing my thesis. Secondly, I would like to thank Dr. Christoph Lofi for being on the Thesis committee. Thirdly, I would like to thank DR. Ir. Pekka Jääskeläinen for the opportunity to come to Finland and do my thesis under him at Tampere University. Under his guidance, I have been able to experience working on new and open-source technologies. Fourthly, I would like to thank Jan Solanti and Topi Leppänen for their insight and opportunities to brainstorm together. I am also grateful for the European Union's Horizon 2020 research and innovation program Grant Agreement No 871738 (CPSoSaware), through which my salary at Tampere University was paid. Finally, I would like to thank my friends and family for their support and encouragement. In particular, I would like to thank Elina Marttila-Tornio for always supporting me and being my guide in Finland and into Finnish culture.

Robin Alexander Bijl
Tampere, Finland
May 2023

Contents

Acronym	Description
ABFT	Algorithm Based Fault Tolerance
AlmaIF	Almarvi hardware InterFace
API	Application Programming Interface
AXI	Advance eXtendable Interface
BRAM	Block Random Access Memory
CPC	Customized Parallel Computing
CPU	Central Processing Unit
DMA	Direct Memory Access
ECC	Error Correcting Codes
FPGA	Field Programmable Gate Array
GCC	GNU C Compiler
GDB	GNU DeBugger
HDL	Hardware Description Language
HLS	High Level Synthesis
HPC	High Performance Computing
IDE	Integrated Development Environment
ILA	Integrated Logic Analyzer
LLVM	Low Level Virtual Machine
MAD	Multiply and ADd
OpenCL	Open Computing Language
PoCL	Portable Computing Language
PL	Programmable Logic
pthread	Posix thread
QoS	Quality of Service
RCQ	Replicated Command Queue
RAM	Random Access Memory
RTL	Register Transfer Level
SBC	Single Board Computer
TTMR	Temporal Triple Modular Redundancy

1 Introduction

The ever-increasing demand for computing has led to the need for specialized heterogeneous hardware, and the frameworks required to utilize them. Besides the traditional central processing units, more and more programs will make use of specialized hardware such as GPUs, FPGAs and DSPs to accelerate computations. There are a number of frameworks that provide a way to utilize this hardware. And while some frameworks only specialize in one type of device, Open Computing Language (OpenCL) specifications provide a shared specification for a wide range of hardware.

Contrary to everyday belief, computers can in fact make errors. And while the chance of an error happening is generally small, the increase in computing lowers the mean time between errors occurring. In this thesis, we address this problem for the range of devices supporting the OpenCL specifications.

1.1. Context

The Customizable Parallel Computing [1] (CPC) group, is a research group that is part of Tampere University (TAU) located in the city of Tampere, Finland. The CPC group focuses on parallel applications ranging from energy-efficient hardware architectures with their OpenASIP tools to high-performance computing with PoCL. PoCL is an open-source implementation of OpenCL and has support for a range of different hardware. The hardware it supports includes: CPUs, GPUs and FPGAs.

The CPC group participated in the CPSoSaware [2] project with a focus on heterogeneous computing and fault tolerance. CPSoS in CPSoSaware stands for Cyber-Physical System of Systems and the project included many different aspects from computer-controlled physical machines to machine learning and software computing stacks. The organizational structure of the CPSoSaware project is to have participants work closely together on different aspects of the project. The CPC group was only one of the thirteen organizations participating and focused particularly on heterogeneous computing, machine learning and fault tolerance. At the start of this thesis, work on heterogeneous computing and machine learning had already been delivered. The only thing left to complete was work on achieving fault tolerance.

1.2. Problem definition

The CPSoSaware project focuses on large, complex systems. The more computations a system does, the shorter the time between errors occurring. If this error is not caught in time, it can propagate throughout the system. One example of such an occurrence happening to cyber-physical systems happened on Qantas Flight 72 in 2008. According to an investigation by the Australian Transport Safety Bureau [3], an error in one of the plane's sensors caused the flight control computer to make an abrupt nose dive.

The CPSoSaware project is broad in its range of applications and this makes PoCL suitable since PoCL is vendor independent and has been ported to a number of different platforms, including ARM and recently partially to RISC-V [4]. This aspect also poses a challenge as any solution that depends on platform-specific properties will be limited in its applicability. The vendor independence of PoCL is also attractive to the CPSoSaware project as it looks at the total lifecycle of a system and parts may be changed or swapped out during that lifecycle.

Another aspect with regard to the CPSoSaware project is that any implementation will only be one part of a larger, more complex system. Any solution that is difficult to use or hard to apply faces the possibility of not being used. A programmer's time is valuable and if a solution requires them to put significant time into one detail of a much larger system, that said solution might not be applied in favor of another.

The problem we focus on in this thesis project is related to adding fault tolerance to large, interacting, complex, heterogeneous computing systems. In order to enable fault tolerance in such systems, we choose to use PoCL as an implementation platform. Therefore, the problem statement in this thesis can be formulated as follows: How can we add fault tolerance to PoCL?
This leads us to formulate three research questions and which are listed below:

1. How can fault tolerance be added to PoCL in a way that is easy to use for the end user?
2. How can the runtime overhead of this implementation be kept to a minimum?
3. How can we preserve the flexibility of OpenCL while adding this fault-tolerant capability?

1.3. Thesis layout

The layout of the thesis is as follows. Chapter 2 discusses the background of fault tolerance as well as the used technology stack that we build on. This includes concepts such as triple modular redundancy as well as key concepts of OpenCL and by extension PoCL. We also discuss previous work and provide a hierarchical overview of different methods for fault tolerance.

This is followed by Chapter 3, in which we discuss a number of criteria by which we evaluate a number of potential solutions. We provide a number of potential solutions which we discuss and score according to the criteria. Finally, we select the most suitable solution for our problem based on how it satisfies the given criteria.

In Chapter 4, we discuss the implementation details of the chosen solution. We go into detail on the design and extensions of PoCL. We also discuss the flow of data and computations done by the chosen solution.

This is followed by Chapter 5, where we evaluate the implemented solution according to the criteria described in Chapter 3. We compare variations of the same solution in order to find the most optimal variant. We also evaluate the solution on different hardware architectures and explain any phenomena shown in the benchmark results.

Finally, we close off with Chapter 6, in which we reflect back on the research questions and the work done. We also discuss potential future work.

2

Background

In order to discuss in detail how we will add fault tolerance to PoCL, it is important to know some background information on the used technologies and fault tolerance in general. We will start off by describing the technology stack used, which among other things includes key concepts of OpenCL and PoCL. Fault tolerance in computing has been tackled before in several different ways. And therefore, in this chapter, we will also discuss the theory of fault tolerance and discuss previous research on different implementations.

2.1. Used technology stack

In this thesis, we build on top of existing technologies and therefore it is important to know the terms and concepts related to these technologies. We will now go into detail on said technologies.

2.1.1. Open Computing Language

Open computing Language [5] (OpenCL) is an open standard that allows a programmer to write code for a multitude of heterogeneous hardware. The standard is maintained by the Khronos group and receives extensions from both industry and academia. Examples of hardware that can be utilized by OpenCL include central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field programmable gate arrays (FPGAs) and custom accelerators. The code written for these devices is executed in a parallel or accelerated way. The standard does not provide an implementation just a description of the framework and runtime needed to run code on this diverse set of hardware. Often it is up to a hardware vendor (e.g. AMD, Intel and Xilinx) to provide the required resources to run on their hardware. Since the standard is open, vendors will often add extensions to their implementation that can exploit features unique to their hardware.

For a programmer, the use of OpenCL follows a number of steps. First, they install the required runtime and framework for their machine. OpenCL is a C-based API with bindings to Popular languages such as C++ and Python. Then the programmer writes a special piece of code in a language called OpenCL C [6] called a kernel. This kernel is different from the program that the programmer writes calls OpenCL. Such a program is called a host program. Figure 2.1 shows a timeline of the typical execution of an OpenCL program. First, the host program queries OpenCL to see the available compute devices and then selects the desired device. The host program then calls the OpenCL runtime to compile and execute the kernel on the hardware device that has been selected. The reason the kernel is not compiled beforehand is that it is only known during runtime what the exact hardware is that the kernel will be run on. However, a programmer can also provide an explicit binary beforehand if they know what hardware it will run on. Before the kernel is executed on the hardware device, there needs to be data that the kernel executes on. This is done by creating buffers and filling them with data. When it is time to execute the kernel, the buffers are moved to the compute device where the kernel then executes. After the kernel is done executing, the result buffer is moved back to where the host program is running and the results can then be read. Some devices also allow the result buffer to be mapped so that the host program can directly read the results.

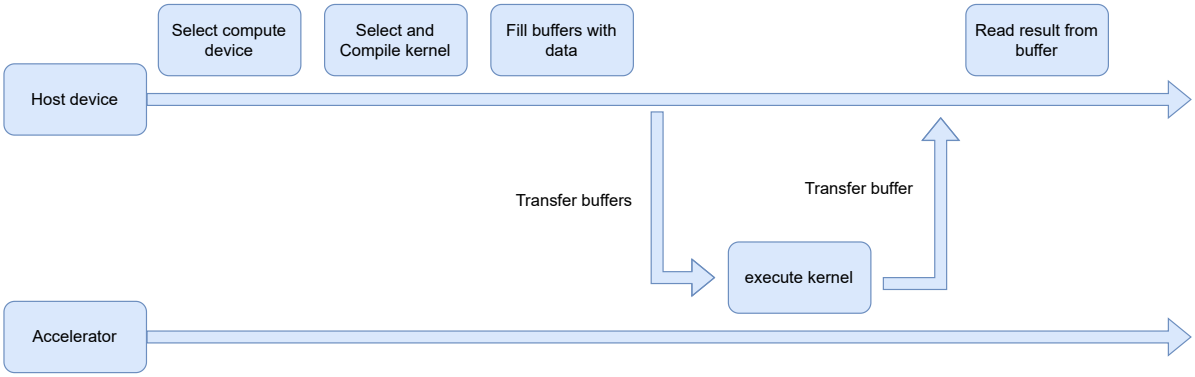


Figure 2.1: The typical steps in the execution of an OpenCL program.

This example shows how OpenCL takes away some of the complexity of writing code for heterogeneous systems. The same kernel code can be run on say a CPU as well as a GPU without modifying it. The programmer does not have to describe when or how data is moved between devices, just what data they want to be moved for a computation to be done on. It should however be noted that some knowledge of how the OpenCL model works and the specific physical hardware can improve the overall performance during execution of the program.

To understand the inner workings of OpenCL, it is important to know a number of objects and terms it uses. To start off, OpenCL differentiates between a host and a device. A host is a system where the program that calls OpenCL is run on, i.e. a traditional CPU with its own RAM. A device is a piece of hardware that executes the kernel code. This device typically has its own RAM to which a buffer is transferred to before execution. The device writes the result to a different buffer in RAM before this buffer is then transferred back to the host where it can be read. On the software level, OpenCL has a number of important objects, as described below.

Device
A device is an abstract object that describes the physical hardware that a kernel will be run on. For example, device-specific compilation parameters as well as a description of the type of device (e.g. CPU, GPU, etc...) can be found here. Kernels are compiled for a specific device. To find all the available devices on the system, one can query the platform.

Platform
A platform is the top layer of information about the system that is running OpenCL. It can be used to get information on device configurations as well creating a context. Every device or context only has one platform.

Context
A context describes the context in which a kernel will be run. This includes what buffers are available, the program running the kernel, the command queue of operations that have to be done and events that can be used for synchronization and callback during execution. A context also has at least one device.

Program
A program is a collection of compiled kernels for a number of devices. It is specific to one given context of which the devices are also part of.

Memory object
A memory object is an abstract datatype that can be either an image or a buffer. It acts like an array of data and can be used as an argument for a kernel. It also serves as the bridge between a host and a device, allowing the host to send and receive data from a device. It is specific to a context and not a device. This means that if a context has multiple devices, the memory object can exist on one or more devices. A memory object can have a number of properties such as being read- or write-only. These properties can be useful hints for the runtime to get good performance.

Command queue
This object is used by the host to issue commands to a device. These commands fall under three different categories: kernel enqueue commands, which are used to enqueue work; memory commands used to read and write to buffers and finally synchronization commands which can be used explicitly to create an order of work.

Event
Events are closely associated with command queues and serve a number of purposes. They signal the status of a command. This allows events to be used to wait for synchronization and as a trigger for a callback function. This can for example be used in scenarios where one kernel needs the results of another kernel.

2.1.2. Portable computing language

Portable computing language [7] [8] (PoCL) is one of the few open-source implementations of OpenCL. It is developed and maintained by Tampere University (TAU) with contributions from the open-source community. The goal of PoCL is to be hardware independent and also maintain performance when changing hardware. It addresses in part the issue that a programmer will often need to optimize kernel code to specific hardware to get the best performance. It does this by delegating the work group vectorization to the LLVM compiler. Having to optimize for specific hardware negates the appeal of having a single interface for multiple devices.

At its core, PoCL uses Clang [9] and LLVM [10] to compile kernel code to a binary that is able to run on the targeted hardware. Due to the wide range of architectures that LLVM supports it is also possible to run PoCL itself on many different architectures. The original paper introduced four different devices of which the pthread device is the most widely used one. But since then, a number of different devices have been added. The presence of all these different devices allows a programmer to mix different devices and select the optimal one for the right stage. Some of these devices are described below.

Pthread device
The pthread device is one of the original devices of PoCL. It makes use of POSIX threads[11], often shortened to pthreads, to execute computations in a parallel fashion. POSIX stands for portable operating system interface and is a standard maintained by the IEEE Computing Society. The pthread device runs on the CPU of the system and makes use of its RAM. The fact that it relies on pthreads means that it is very portable. Pthread devices can work on any system that has an operating system (OS) that supports the pthread application programming interface (API), regardless of what the underlying CPU architecture is[7].

AlmaIF device
The concept of using FPGAs for accelerated computing is becoming more and more popular. But in order to use an FPGA for such applications, an interface is needed. the Almarvi hardware interFace (AlmaIF) device is a software driver that allows hardware that has an AlmaIF v2 interface to be controlled by PoCL. As shown in the paper by Leppänen et al. [12], the interface can be used to control hardware realizations on FPGAs such as softcore processors as well as modules created by high-level synthesis (HLS). These realizations can then be used as accelerators in PoCL. The interface is designed to be hardware vendor independent. In OpenCL, this driver is classified as a custom device that abstracts away more traditional steps like the online compilation of kernels. Instead, it provides a set of built-in kernels that are implemented on the FPGA fabric.

The AlmIF device also provides an emulation device. It implements the interface and allows for built-in kernels to be programmed into it. This device can then be used to simulate the FPGA realization on hardware that does not have an FPGA. This feature can be quite useful in the design process as any changes can more easily be applied before settling on the final specifications for the FPGA.

Vulkan device
One of the more recent additions to the list of devices available on PoCL is the Vulkan driver. Vulkan is an open API [13] standard maintained by the Khronos group. Vulkan is known as a platform-independent graphics API, but it also provides the ability to do computing. The Vulkan driver uses the CLSPV compiler to compile CL code into SPIR-V. SPIR-V stands for standard portable intermediate representation where the ”V” indicates the version. The compiled SPIR-V code can then be used

as input for any available Vulkan runtime to do the desired computation. There exist Vulkan runtimes for hardware from vendors like AMD, Nvidia and Intel, allowing the Vulkan driver to use any of the hardware available.

2.1.3. Vitis high-level synthesis

High-level synthesis (HLS) is the process of taking a program written in a high(er) level language such as C and generating a description of the hardware that implements the same functionality. This Description is on the register transfer level (RTL) and is in a hardware description language (HDL) such as Verilog or VHDL. RTL level is a very low level and describes how the basic building blocks of digital circuits are connected to each other. This HDL file can then be used by other programs to synthesize bitstreams that in turn are used to configure FPGA devices.

The FPGA vendor Xilinx provides a software suite called Vitis HLS for this purpose. For this thesis, we are using a Xilinx FPGA, therefore it makes sense to use the tools provided by Xilinx. However, it is not the only one to provide such HLS tools. Through the use of pragmas in the C code, Vitis can be instructed about what interfaces and optimizations are desired. By creating a test bench also in C, the code can be checked for desired functionality. There are two ways of doing this testing: C or RTL simulations. With a C simulation, both files are compiled into binaries and run as regular C programs. This is fast but not accurate with regard to the eventual hardware that will implement it. The benefit of this test is that it allows one to check for functional programming mistakes. For a better analysis, an RTL simulation can be used. Here speed is traded in for better accuracy by running a simulation using the generated RTL code.

It is not necessary to run the C simulation tests in Vitis. Since The source code is in C, it is possible to compile it with a conventional compiler outside of Vitis. It also allows one to use a debugger of choice such as the GNU debugger (GDB). There are some conventions one should adhere to when writing C code for HLS, but overall the barrier to entry is lower than for HDL languages. And since programming errors can easily be found in the early stages of development, it is also quicker to develop.

Writing code for HLS is different from writing other accelerated code. When writing parallel code to run on say a GPU, one tries to break the problem up into many smaller blocks. These blocks can then be run in parallel on the many processors of the GPU. When writing HLS code, one tries to instead split the problem up into a sequential order. This is so that the HLS tools can apply pipelining. Pipelining is the concept of breaking a task up into multiple stages and starting computations on a new piece of data before the previous entry is done. While this does not reduce the time to complete one task, it does increase throughput since data can be processed concurrently. This makes HLS very suited for streaming applications.

2.2. Fault tolerance theory

In order to prevent faults, we first need to know what kind of faults there are. The book ”Distributed Systems: An Algorithmic Approach” [14] by Ghosh classifies faults into seven different categories. Solutions for these differ and there is no catch-all solution. Transient errors are particularly interesting for us.

Transient errors are classified as errors that disturb a system temporally. The source can be from outside the system and examples include unstable power delivery; (cosmic) radiation (often referred to as single event upset (SEU)) and mechanical stress. These errors are sometimes also referred to as soft errors since they do not physically damage the system, but the execution is still disrupted in some manner. An example of such a disruption is a bit flip in which a bit changes from one to zero or visa versa.

Some errors not looked at in this thesis are errors caused by the end programmer accidentally introducing bugs. This is classified as a software error. Another such error that was not looked at is a security error, where there is deliberate tampering with the system from inside or outside.

There are two aspects of dealing with errors: fault detection and fault recovery. A system cannot mitigate an error that is not detected and in some scenarios, merely knowing an error has occurred is already enough. In other scenarios, it is also necessary that the system continues to function and that is where recovery plays a role.

Double Modular Redundancy (DMR) is a way of detecting failures. In this setup, two computations are run independently and if at any point an error occurs, be it in transferring data, performing operations

on it or a bit flip in a register, the results will end up being different. Comparing the two results will show that there has indeed been a fault but is not possible to tell which result is correct.

Expanding on this concept, Triple Modular Redundancy (TMR) adds a third independent computation. If one fails, a majority vote can be used to determine the correct result. However, this concept is still vulnerable to a scenario where two errors occur at the same time. A generalized version of this is called N-modular redundancy (NMD) and allows for more errors to occur. Equation 2.1 describes the number of errors that can be recovered from with NMD, in this equation, n is the number of independent devices and m is the number of errors that can safely occur. It should be noted that n needs to be an odd number, otherwise it is possible for scenarios to happen where there is no majority vote.

$$n \geq 2 * m + 1 \tag{2.1}$$

2.3. State of the art

There are many approaches to ensure reliability in heterogeneous systems [15]. In Figure 2.2 we provide a hierarchical overview of a number of different approaches to achieving fault tolerance. We discuss the idea behind each approach and identify any benefits and shortcomings for our use case. This thesis is focused on developing a fault-tolerant approach that is both compatible with the various components of PoCL and can be easily implemented and used by developers using PoCL.

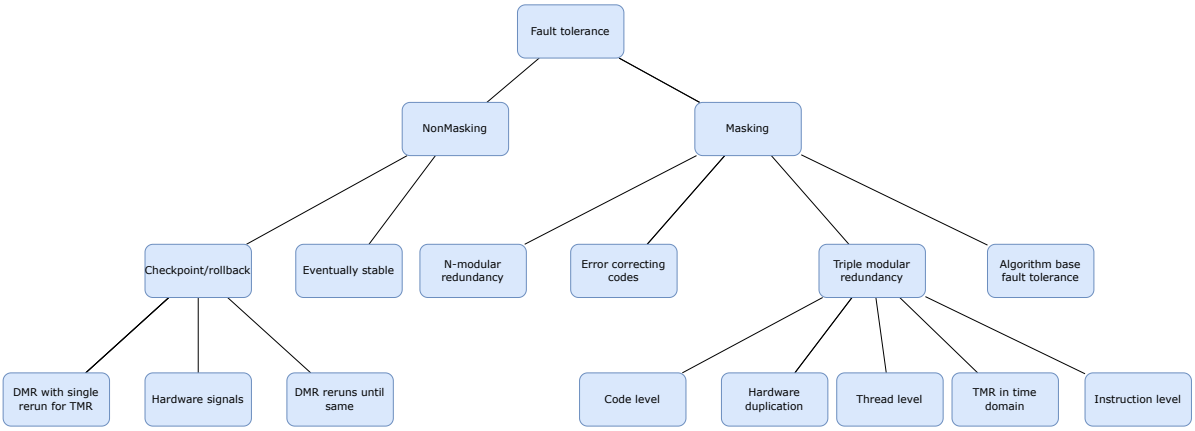


Figure 2.2: A comparison of different methods for fault tolerance.

2.3.1. Nonmasking fault tolerance

In nonmasking fault tolerance, the presence of an error can cause a system to temporarily be in an invalid state but eventually reach a valid state again. Often this results in delays in time. There are two common forms of nonmasking fault tolerance: checkpointing, also referred to as rollback, and eventually stable fault tolerance.

A system that uses checkpointing will periodically save its state. When it makes this checkpoint, the state is known to be error-free. This can be done in multiple ways, data can for example be written to permanent storage or registers can be copied somewhere else. When an error is detected somewhere before the next checkpoint is made, the system can revert back to the known good state.

Checkpointing has been implemented multiple times in literature. An example of this is the paper by Mushtaq etc al. [16]. The authors introduce a library that exploits the multicore nature of today's processors to execute a process redundantly. If at some point an error is detected, the system uses checkpointing to recover. Checkpointing is often used in High-Performance Computing (HPC) environments [17], [18], [19]. Due to the large number of computations done in such large systems, errors are common.

The examples given so far fall in the category of applying DMR and rerunning until both results are the same. It is also possible that a rollback is triggered by specialized processors that have special hardware for the detection of errors. Some systems in the case of a difference in results in DMR execution opt to execute the computation a third time in order to apply TMR and resolve the errors

encountered with a majority vote. This is called Temporal Triple Modular Redundancy (TTMR) and has been applied in work by Czajkowski et al. [20][21].

The checkpointing paradigm has proven to be very useful in scenarios such as HPC. However, in the case of an error occurring, the total runtime of the system is increased, something not desired for our use case. The library by Mushtaq et al [16] sounds promising if applied to the pthread device of PoCL. This is due to the library not depending on special hardware and therefore being portable and like the pthread device, making use of pthreads. However as mentioned previously, the pthread device is only one of the available devices PoCL provides, thus limiting the applicability of the library as a whole to PoCL.

Some systems are able to tolerate small errors. Even though there is an error, the system is able to correct itself back to a valid state without having to redo a computation. Or in some cases, the error can outright be ignored without a noticeable loss to the output of the system. This kind of fault tolerance is what the ”Eventually stable” node indicates. This kind of fault tolerance is situational and can not be applied to every use case.

2.3.2. Masking fault tolerance

With masking fault tolerance any error occurring during the operation of the system is not visible from the outside. This form of fault tolerance is used in safety-critical scenarios where any errors or delays are unacceptable. Whereas nonmasking fault tolerance will repeat the computation in order to resolve an error and thereby increase the total runtime, masking fault tolerance will resolve the error.

One such form of masking fault tolerance is called algorithm-based fault tolerance (ABFT). The concept of ABFT is to exploit mathematical properties of the being done to both detect and correct any errors that might occur. One such method was introduced by Vainstein [22] and is called a checking polynomial. Vainstein’s approach is to use algebraic methods to both detect and correct errors in computations. It does this by finding a so-called checking polynomial for a function or number of functions and using this polynomial to check the results. While this method does not require any modification to hardware, it requires the person writing the code to find these polynomials which is not an easy task. This significantly increases the barrier to entry and is not desirable for our use case.

Another form of masking fault tolerance is to use error-correcting codes (ECC). There are a number of different error-correcting codes of which Hamming code [23] is a well-known example. A Hamming code works by calculating a number of parity bits over a range of bits. Usually, this range is double digits in size. When the data is accessed, a new set of parity bits are calculated over the read data, and if this diverges from the previously stored parity bits, the error can be localized and corrected. Error-correcting codes are often used in server-grade memory and some GPUs also come with them. Running PoCL on ECC-capable memory reduces the number of errors visible, however, ECC only catches errors related to moving and storing data. ECC will not catch a calculation error as when the data is written back, it calculates the parity over the erroneous data.

Triple modular redundancy (TMR) is a form of masking fault tolerance that has been implemented on many levels of computing. By doing a calculation three times and using a majority vote on the results, TMR is able to correct errors.

An implementation of TMR on the lower end of the computing stack is shown by Li et al. [24]. They provide a mechanism to execute work on three separate redundant processor cores. A fourth core acts as a voter and manages the execution of the other cores. While this implementation provides low runtime overhead due to the use of four cores, an implementation in PoCL will be limited in scope to custom devices only.

Going a step higher in the computing stack is SWIFT-R [25]. SWIFT-R adds TMR on an instruction level. SWIFT-R is implemented into a compiler. During compilation, SWIFT-R will add extra instructions to execute work three times and then compare the results. By being a software-only implementation, There are no special hardware requirements. This would work well on any of the devices of PoCL that make use of compilation, but not on devices such as the AlmaIF devices that only have fixed function hardware accelerators.

Instead of adding redundancy to instructions, it is also possible to add redundancy in code. A programmer can write their code to do the same thing three times and then compare the results. However, compilers such as for example GCC and Clang have gotten to a point where they can detect this redundant execution and will optimize it away. This optimization can be overcome in the C programming language by using the ”volatile” keyword for variables but also prevents the compiler from applying any

other optimizations.

Triple modular redundancy can also be achieved on a hardware device level. In such a scenario, there are multiple hardware devices that share little to no resources. The devices do however have some form of data exchange. Each device acts independently to compute a task redundantly after which the result is sent for comparison. Such an implementation works well in PoCL since the level of TMR is high enough that it can be applied to any device supported by PoCL.

2.3.3. Comparison methods

Since most forms of fault tolerance rely on some form of comparison of results, it is also interesting to categorize the options available. These can be seen in Figure 2.3. These levels each differ in granularity. On the buffer level, the entire result is compared. There is very little granularity, either all contents are correct or it is marked as erroneous. In such a case, it can also make sense to compute some form of hash (checksum or identity hash) over the entire buffer and compare those. This can reduce data transfer times and simplify the actual comparison but requires some work to actually compute the hash. Finding a suitable hash that works well in the parallel environment of OpenCL is an interesting topic that falls outside the scope of this thesis.

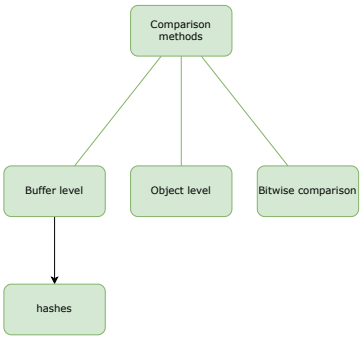


Figure 2.3: Different methods for finding differences in computed results.

On the object level, individual variables are compared. In this case, the maximum granularity goes up to the size of the register of the processor that the comparison is running on. This finer granularity allows for more fault tolerance since more errors can be corrected as long as the errors do not happen at the same variable index. Depending on how this granularity is used, it is also possible to correct errors earlier. A bitwise comparison is even more granular than an object-level comparison. Using bitwise comparisons, individual bit errors can be detected and even corrected.

3

Methodology

In this chapter, we will discuss a number of different possible solutions that can bring fault tolerance to PoCL. To evaluate each solution, we first present a number of criteria. These criteria assist in assessing each possible solution and finding the solution that best fulfills the research questions we described in Section 1.2. We end this chapter by picking a solution and discussing our reasoning for doing so.

3.1. Criteria

In order to narrow down the best solution, a number of criteria have been selected. These have been based on the context and research questions described in Section 1.2. These criteria are: Ease of use for the end user; runtime overhead; hardware overhead; applicability and robustness. We will now discuss each criterion in further detail.

3.1.1. Ease of use for the end user

For our solution, we consider the end user to be the programmer that writes an application that uses the PoCL library. This is because we do not expect the user of that application to know anything about the internals of the application and instead consider the application a black box. If the solution is to be used, it should not bring a higher barrier to entry. Heterogeneous computing is already considered more difficult than programming for one processor. Therefore ideally it should be easy to use once the foundation of OpenCL is laid out in the code. This non-functional criterion is hard to quantify, so we will provide some good and bad characteristics that a solution can have.

An example of a good characteristic of a given solution is the number of lines of code that need to be added to make a given OpenCL application fault tolerant. If there are only a few extra function calls or there is a flag that can be passed along to a function to enable fault tolerance, the programmer does not need to make many modifications to existing code. Therefore it can be considered easy to add.

If a solution requires an entirely new set of objects and functions to be used instead of the standard OpenCL ones, the programmer will first need to familiarize themselves with the programming guidelines of these new functions. After familiarizing themselves, the programmer will have to modify many sections of the code. Therefore, such a solution can not be seen as easy to use.

Deviating from the OpenCL specifications might be necessary for some solutions. However, it is preferable to stay close to the standard as that is what most users will be familiar with. Some solutions might not even need any modification to the code. For example, this could be because the tolerance is applied on a hardware level, or the compiler handles it.

Depending on the solution, it might also be beneficial to be able to enable or disable the fault tolerance. This feature can be useful during development for quick testing of new features. Some solutions might also require hardware or resources not available during all stages of development, so being able to easily disable it is beneficial.

A bad characteristic would be if the end user must rewrite large quantities of code to apply fault tolerance to an existing application. This can be especially a hindrance if it involves kernel code as this will make the code only compatible with the solution and therefore less portable. A solution can also be

hard to use if it requires the programmer first to spend a significant amount of time to first understand the solution before it can be used.

To sum the ease of use for the end user criterion up, the less time and effort an end user spends to apply the solution to their (existing) application, the higher the ease of use can be considered.

3.1.2. Runtime overhead

A fault-tolerant solution often adds some overhead in the form of delay before the execution is done. This can vary greatly from one form to the other. In some solutions, the delay can also be affected by the occurrence of an error. This extra delay that is caused by an error is what differentiates masking from nonmasking fault tolerance. Nonmasking tolerance will still eventually reach a correct state, but the moment of uncertainty can be unacceptable in applications where time is critical.

The importance of these delays can vary per use case, in some cases, the overhead does not matter as much, as long as the result is correct. However, one of our research questions is to find out how we can keep the runtime overhead as low as possible. Therefore, when comparing different solutions, a lower runtime overhead is always desirable.

3.1.3. Hardware overhead

Besides an overhead in time, many solutions can also add an extra requirement in the domain of hardware. Many masking solutions require more hardware than nonmasking solutions. Sometimes, a tradeoff can be made between runtime overhead and hardware overhead. An example of this can be seen with TMR. Three devices could compute the required redundant results at the same time, or one device could compute the same result three times and the results are compared after that.

The range of hardware overhead can vary from extra logic on a chip to separate processing cores on the same CPU to entire separate devices that share no resources with each other. Monetary costs of a solution are often reflected to an extent in this extra hardware. However, costs are not a concern in our concerns. This same logic also applies to power consumption.

While our research questions do not put a limit on hardware requirements, it is still a factor that is worth keeping in mind. A solution that has lower hardware requirements than another is preferable. A solution that has steep hardware requirements can lose out on some of the flexibility of OpenCL.

3.1.4. Applicability

The applicability criterion captures how well a given solution can be applied to different applications in OpenCL and more specifically PoCL. Of course, a solution should be able to be applied to be even considered, however, a solution might not be applicable to a particular device type. An example would be a solution that targets compilers. This solution would work well on pthread devices, but not if a GPU device has a closed source compiler or a fixed function accelerator. The same goes for a solution that adds extra logic to a chip or FPGA.

For this comparison, we will limit ourselves and make a grade based on three different types of devices. These are: CPU devices, such as pthread devices; GPU devices such as Vulcan devices and finally custom devices such as AlmaIF devices. In Table 3.1, the number of "**+**" indicates on how many devices the given solution works. Since the pthread driver is very much a staple of PoCL and works on a wide variety of architectures, it is preferable that a given solution works on this, but this is ultimately not taken into account in grading the applicability.

The applicability criterion is a way to evaluate how well we preserve the flexibility of OpenCL. PoCL has a wide range of supported devices and limiting a solution to a subset of these devices would make it less flexible.

3.1.5. Robustness

Robustness gives an indication of how well a solution is resistant to errors happening. This criterion helps evaluate the research question of how fault tolerance can be added to PoCL that is easy to use for the end user. The robustness gives an indication of how fault tolerant a solution is. A point of interest is where the points of failure are. A solution where only one device is critical for good operation is better than one that relies on multiple aspects to go well in order to guarantee good operation.

Another characteristic is how many errors can happen while continuing to be fault tolerant. A solution that uses TMR with computed hashes over an entire buffer will be able to tolerate any number of

errors in one device as long as the other two devices do not have any errors. This will work well in a scenario where the chance of errors is very small, but if the chance is larger, the chance of getting three different hashes is a lot larger and thereby not a good application of a solution. A more robust implementation would be TMR while looking at objects, for example, 32-bit integers. In this case, at least one error needs to occur in the results of two different devices before fault tolerance fails. This dramatically improves the robustness compared to the previous solution. An even bigger improvement to robustness would be to do a comparison on the bit level since the given solution will fail if two or more bits at the exact same buffer index experience a flip.

Some of the previous work mentioned in Chapter 2 do not aim to catch all errors, but rather only the critical errors. Such a solution will score lower in this criterion compared to a solution that catches all errors.

3.2. Alternative solutions

In Chapter 2, we created a hierarchical overview of different fault-tolerant methods, the result of this can be seen in Figure 2.2. For many entries of this overview, we also discussed previous work that tackled fault tolerance using a similar method. Based on this information, we have selected a number of possible solutions that we can apply to PoCL to add fault tolerance. Table 3.1 shows an overview of the possible solutions. The first column provides a name of the possible solution while the other columns grade the solution based on the criteria described in Section 3.1. We make a distinction between runtime overhead with or without errors occurring since depending on the implementation, these can be different.

Solution	Ease of use for end user	Runtime overhead no errors	Runtime overhead errors	Hardware overhead	Applicability	Robustness
Hardware-level checkpointing	+ + +	+ +	+	-	+	+ +
Thread-level checkpointing	+ + +	+	+	+	+ +	+ +
Code-level checkpointing	- -	+	+	+	+ +	+ +
Bitwise TMR on runtime level	+ +	+ +	+ +	- -	+ + +	+ + +
Hashed TMR on runtime level	+ +	+ +	+ +	- -	+ + +	-
checkpointing on runtime level	+ +	+ +	-	-	+ + +	+ +

Table 3.1: A table with different possible solutions and their scores according to criteria.

3.2.1. Hardware-level checkpointing

The Hardware-level checkpointing solution applies DMR on a functional unit level of a chip. By having two functional units doing the same operation twice, and comparing the result, faulty operations can be detected. Once a fault has been detected, a rollback can be done so that the correct result is achieved and errors are not propagated. This has a number of benefits.

The first benefit is the ease of use for the end user. by having the hardware handle all redundancy, the fault tolerance is abstracted away from the programmer. The programmer can use the device as any normal OpenCL device.

The second benefit is the runtime overhead. Since the operations are done in parallel, the only major overhead comes from the logic of comparing the results. And since the comparisons are done in hardware on a fine-grain level, a rollback in the case of an error will only require a couple of operations to be repeated.

This solution also has some drawbacks. The first drawback is the hardware overhead. Extra area of an FPGA or chip is required for the duplicate hardware and comparison logic. This is needed to keep the runtime overhead low.

A larger drawback is the applicability of hardware-level checkpointing. Since this solution requires special hardware, it is limited to only accelerators like the AlmaIF device. This is due to the freedom

that the AlmaIF device provides with regard to how to implement accelerators in hardware. GPU and pthread devices have fixed hardware and can not provide such fine configuration of the hardware.

3.2.2. Thread-level checkpointing

Instead of providing checkpointing on a hardware level, thread-level checkpointing applies checkpointing on a higher level. Similar to the work by Mushtaq et al. [16], This solution works By having two threads computing the same work and regularly comparing the results. In the case of an error, a rollback can be done.

This solution differs from the work of Mushtaq et al. by adding this fault tolerance as part of the compiler instead of a library. Like this, the end user does not need to make any changes to existing code or use new functions. An end user can continue to write OpenCL kernels without having to consider fault tolerance.

By implementing this fault tolerance in the compiler, we limit ourselves to only CPU and GPU devices. Pthread devices make use of the LLVM compiler and this compiler can be modified to apply such fault tolerance. Something similar could also be achieved for GPU devices through the Vulkan device back-end of PoCL. The CLSPV compiler also makes use of LLVM and therefore could have similar modifications applied to it.

By having the checkpointing on thread level and therefore one device, there is no extra hardware overhead required. The downside of this is that there is a significant runtime overhead since one device is essentially doing twice the number of computations. On top of that, there is also some overhead with regard to comparing results and possible rollbacks.

3.2.3. Code-level checkpointing

Instead of having the compiler apply fault tolerance, code-level checkpointing instead requires the end user to apply fault tolerance to the code they write. This could be done by extending the OpenCL C specifications and adding special functions that allow a programmer to add checkpoints and possible rollback options. This adds a lot of control to the programmer to add fault tolerance where they deem necessary but also requires them to have a better understanding of the code in question and fault tolerance in computing.

Similar to thread-level checkpointing, the solution does not have any extra hardware overhead requirements. But similarly, this solution also suffers from a higher runtime overhead due to the extra redundant computing. However, a programmer can choose to not apply fault tolerance to every part of the code in an attempt to lower overhead. This can come at the cost of lower robustness.

Since this solution extends the OpenCL C specification, the compiler will need to be modified. Therefore, this solution is also limited to only CPU and GPU devices.

3.2.4. Bitwise TMR on runtime level

This solution makes use of the OpenCL concepts to implement TMR. By executing the kernel redundantly on three different devices, the runtime is kept low regardless of whether errors occur or not. By applying TMR on a runtime level, we are assured that this solution can be applied to any device PoCL provides. And by extending the PoCL code, it is possible to make it easier for the end-user to apply the solution.

To create a high level of robustness, a bitwise comparison and possible correction of the buffers can be made. This implementation also provides a form of redundancy outside of computation. Because the data is sent to a fourth voter device, any errors encountered during storing or transportation of the data are also corrected. This voter device is a single point of failure and can be run on hardware that has been hardened against faults.

There are however downsides to this implementation. The largest downside is the large hardware requirement of three redundant devices and a voting device. This makes it very expensive from a hardware perspective. Another downside of this implementation lies in the comparison method. In order to do bitwise comparisons, all buffers need to be moved to the voting device. This can cause some time overhead due to the large amount of data that needs to be moved. Overall, this solution maximizes robustness while not requiring the end user to have intricate knowledge of the subject.

3.2.5. Hashed TMR on runtime level

With hashed TMR, instead of transferring the entire result buffers of each redundant device to vote, a hash is sent instead. This hash is computed of the result buffers and is smaller than the buffer. This reduces the time spent transferring buffers to the voting device. The voting device compares the hashes and picks a buffer that is correct. This buffer is then read by the host program to get the correct result.

This solution is similar to the bitwise TMR on runtime solution and has similar properties. It has the same hardware overhead and is applicable on all types of devices. The runtime overhead is the same regardless of errors. Since this method computes a hash, some time is required to compute it. Depending on the size of the workload, computing hardware and transfer speeds, this method can have an advantage in speed compared to the bitwise TMR solution.

One major difference between the bitwise TMR and hashed TMR is the level of robustness. Because hashed TMR votes on entire buffers, it can not correct errors in scenarios where two devices experience an error. This is due to the fact that an error will change the hash and the hash does not give any indication where the error occurred. Therefore, in a scenario with two devices experiencing errors, there will be three different hashes and the voter will not be able to pick a majority.

3.2.6. Checkpointing on runtime level

The high hardware requirement costs of the previous TMR runtime level solution can not always be feasible in every scenario. In such scenarios, double modular redundancy can be a viable solution. By having one less device, errors can still be detected, however, the masking ability is not possible anymore. Instead, the computation can be redone from a point in which the known state was good until the results match.

By implementing DMR on a runtime level, we keep the benefits such as the guarantee that it will work on any device supported by PoCL and the possibility of detecting errors that happen during storing or transportation of data.

However, the downside of applying DMR on a runtime level is that the checkpoints are a lot more coarse grain. The previously discussed solutions implemented DMR on a much lower level, therefore the checkpoints could be a lot smaller. Now the entire computation needs to be done again. The end user could try to mitigate this to a certain extent by breaking up the work into smaller pieces so that there are more intermediate buffers that can be used as checkpoints. However, this is not possible in all scenarios.

In scenarios where there is no hard time requirement, a delay is acceptable. It can also be the case that the chance of an error is so low that a redo is rare ([26],[27]). However, our goal is to keep the overhead as low as possible and we can not make any assumptions about if this delay is acceptable in the end-use applications. Therefore, in a worst-case scenario, this overhead is unacceptable.

3.2.7. Chosen solution

Of the six possible solutions we presented, we will now pick a suitable solution that we will implement. Hardware-level checkpointing provides a solution that is easy for the end user and has low runtime overhead. However, it requires specific hardware requirements that make it only suitable for accelerators like the AlmaIF device. Thread- and code-level checkpointing can not be applied to the AlmaIF device but can be applied to CPU and GPU devices. These two solutions, however, do introduce more overhead compared to other solutions. Since one of our goals is to have the solution work on the range of hardware devices supported by PoCL, one could make the argument to implement both Hardware- and thread-level checkpointing. However, this would require more time to implement than there available in this thesis.

This leaves us with the possible solutions that work on the runtime level. Checkpointing on runtime level provides an easy way for an end user to apply fault tolerance that works on every device supported by PoCL. It also has lower hardware requirements compared to the TMR solutions. However, while it scores well on runtime overhead when there are no errors, this is not the case when errors occur. While there are use-case scenarios where errors are rare or such delays are acceptable, we can not make this assumption for all use cases. This leaves the bitwise and hashed TMR on runtime solutions. These both provide an easy way to implement fault tolerance on a wide range of devices. On top of that, they both can be implemented with minimal changes to the OpenCL specification. Both solutions also have the same steep hardware requirements, however, that is required to achieve a low runtime overhead.

Of these, the hashed TMR has the potential to be faster than the bitwise TMR since there is less data transferred. However, the bitwise TMR allows more errors to occur across all redundant devices and is able to correct these. Therefore, we choose Bitwise TMR on runtime level as the solution that best accomplishes the goals set forth with our research questions.

4 Implementation

In this chapter, we cover the details of the implementation. We will go into detail on how we integrated our chosen solution into PoCL from a runtime level as well as details on the voting devices and their kernels.

4.1. Solution architecture

The bitwise TMR solution chosen in Chapter 3 is on the runtime level. This means that the solution is implemented in the higher levels of PoCL. Our implementation of TMR involves three redundant devices and one voter device. These redundant devices can be any device supported by PoCL and do not need all be the same type. The voter device can also be any device supported by PoCL, but as we will show in Section 4.5, we also provide a fixed function accelerator to perform voting using the AlmaIF interface.

In order to apply fault tolerance, a number of modifications to the code of PoCL had to be made. These modifications consist of the following parts: a modified command queue called a replicated command queue (RCQ); a new function to initialize the replicated command queue; new properties that can be passed to specific functions and modifications to the function used to queue work.

4.1.1. Replicated command queues

PoCL is written in the C programming language, therefore the command queue object is implemented as a struct type in PoCL. In order to implement TMR, this struct has been extended with new members. We call this modified command queue a replicated command queue (RCQ). The RCQ can be used the same as a normal command queue, the only difference is that it is initialized via a different function. This means that an existing application making use of PoCL only has to change the initialization function for a command queue to one for an RCQ. Other functions taking a command queue as an argument do not have to be changed.

Name	Type	Function
is_replicated	int	Used to check if the command queue is replicated
num_rep_devices	unsigned int	A count of the number of redundant devices
rep_devices	cl_device_id *	An array of devices on which the user kernel will run
rep_queues	cl_command_queue *	An array of command queues that are used to schedule work on the devices.
voter_program	cl_program	The program required to create the voting kernel
voter_kernel	cl_kernel	The kernel that is used to determine the correct result

Table 4.1: The new members added to command queue struct type.

The new members to the command queue struct can be seen in Table 4.1. It now becomes clear where the RCQ gets its name from. The RCQ has references to a number of other command queues that are associated with each redundant device. The RCQ also contains a reference to the voting kernel which is run on the voting device. Every device needs a command queue in order for work to be scheduled on it and in the case of the voting device, this is the RCQ. The end user can still schedule

work on the RCQ as they would normally, but as we will discuss in Subsection 4.1.4, behind the scenes, the work is not actually scheduled on the voting device.

4.1.2. Replicated command queue creation function

In order to properly initialize an RCQ, a new function has been added to PoCL. This function is called **clCreateReplicatedCommandQueuePoCL**. The arguments of this function can be seen in Table 4.2.

This function is an extension of the OpenCL API and this fact is reflected in the name. The ”cl” prefix indicates that it is an OpenCL function, ”CreateReplicatedCommandQueue” indicates the purpose, and finally, the ”PoCL” suffix indicates that it is an API added by PoCL.

Name	Type	Description
context	cl_context	The context with the information needed to create the command queue
num_devices	cl_uint	A count of the number of redundant devices.
devices	cl_device_id*	An array of devices that will act as redundant devices
voter_device	cl_device_id	The designated device for voting
properties	cl_command_queue_properties	A list of flags that can be used to change the behavior of the command queue
errcode_ret	cl_int*	A value that can be used to signal something went wrong. If this is nonzero, an error has occurred

Table 4.2: The arguments of the *ClCreateReplicatedCommandQueuePoCL* function used to create a RCQ.

The function does a number of things. The first thing it does is to make sure that all the inputs are suitable in order to create an RCQ object. After that, the new members of the RCQ are initiated. For each device passed to this function via the *devices* argument, a new command queue is initialized. The next step is to select and build the right kernel. By passing flags via the *properties* argument, a specific kernel can be selected. The list of available properties specific to the RCQ can be found in Subsection 4.1.3. The *properties* argument can also be used to pass normal command queue properties to the replicated command queues. Finally, after everything goes well, the RCQ object is returned. If something goes wrong, the *errcode_ret* will return a number associated with the type of error that occurred.

4.1.3. Properties

As mentioned before, when calling **clCreateReplicatedCommandQueuePoCL**, it is possible to pass along a number of properties to configure the RCQ. We will now go over the new options added. As can be seen in Table 4.3, a total of six new properties have been added.

Property name	Description
CL_QUEUE_HARDENED_VOTER	Select a hardened voting kernel
CL_QUEUE_VOTER_GRANULARITY_8	Create kernel with 8-bit datatype arguments
CL_QUEUE_VOTER_GRANULARITY_16	Create kernel with 16-bit datatype arguments
CL_QUEUE_VOTER_GRANULARITY_32	Create kernel with 32-bit datatype arguments
CL_QUEUE_VOTER_GRANULARITY_64	Create kernel with 64-bit datatype arguments
CL_QUEUE_VOTER_GRANULARITY_512	Create kernel with 512-bit datatype arguments

Table 4.3: Properties that can be passed to the *ClCreateReplicatedCommandQueuePoCL* function to select the desired voting kernel.

The first property is *CL_QUEUE_HARDENED_VOTER*. Passing this property to **clCreateReplicatedCommandQueuePoCL** will cause the function to create an RCQ with a hardened voting kernel, the details of which can be found in Section 4.4. This hardened kernel is only enabled when this property is passed along and is not mutually exclusive with any other property.

The other five new properties are however mutually exclusive. The various *CL_QUEUE_VOTER_GRANULARITY_** properties are used to select the size of the inputs for the voter kernel. Since the voter does a bitwise comparison, the result will always be the same regardless of the property chosen. The size of the property also does not need to match the size of the datatypes that are arguments to the redundant kernel. The only requirement is that the result buffer is a multiple of the chosen granularity. The use of these properties is to be able to select an input size that runs optimally on the given hardware. In Chapter 5, we show the effect on performance these different granularities have in different configurations.

4.1.4. Execution function modifications

When everything has been set up the same way an end user would normally set up an OpenCL program, it is time to enqueue the kernel to the RCQ. To do this, the user has to call the same function they would normally call. This function is the *clEnqueueNDRangeKernel*[1] function.

The function has been changed to handle the RCQ. The first thing it does is analyze the arguments of the kernel. Read-only memory objects and scalar objects can be kept unmodified. However, writable memory objects need to be different per redundant device. This prevents devices from overwriting buffers intended for other devices. *clEnqueueNDRangeKernel* creates intermediate buffers per device and enqueues a command per device to execute the kernel with the writable buffers pointing to the intermediate buffer assigned to it.

The function also issues commands to transfer the intermediate buffers to the voter device and start voting once the redundant devices are finished. This scheduling happens by means of events. The voter device itself is set up in a manner that it writes the result to the original buffer that was assigned as output to the redundant kernel. In the case that there is more than one output buffer, the process of moving buffers and voting is enqueued and repeated. The reason to vote in such a stepwise manner is to use less total memory. The memory requirement can become quite large since there are minimally 3 buffers the size of the original output buffer on the voting device.

4.2. Data flow

In order to get a good overview of what happens during execution, the diagram of Figure 4.1 can be consulted. It follows the data from a high level per device. This figure also highlights some of the differences between the physical hardware and software representations. A good example is the step of copying the input buffers to devices. In software, there exist only input buffers zero and one; however, each physical device has a copy of this in memory.

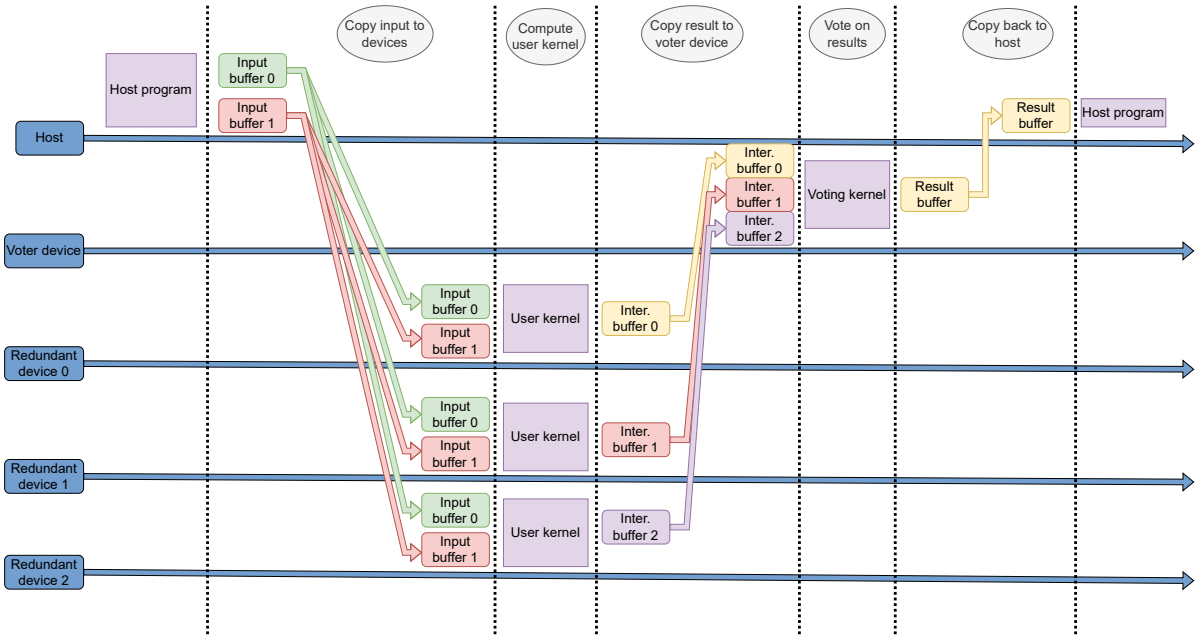


Figure 4.1: The flow of data and computations during TMR execution.

4.3. Concurrency modifications

Our implementation for fault tolerance in PoCL brings with it some inherent concurrency. As an example, whereas before typically each device would be executing its own kernel, now one kernel is replicated on all redundant devices. To get this concurrency working, some modifications needed to be made.

[1] https://registry.khronos.org/OpenCL/sdk/3.0/docs/man/html/clEnqueueNDRangeKernel.html

4.3.1. Pthread device

The most obvious change to the pthread driver was to move its scheduler from a device-wide one to a scheduler per instance of a device. It has been possible before to get multiple instances of a pthread device by passing an environment variable, but it seems until now a shared scheduler was not an issue.

Less obvious was the case where the pthread driver prepares a built-in kernel for execution. PoCL uses the built-in kernel feature of OpenCL as a way to share a common set of kernels that do not need to be declared by a programmer. Before a built-in kernel can be executed on a pthread, the source has to be found. This requires the kernel name to first be sanitized and later restored. Since all redundant devices share one kernel, this can lead to some unexpected heisenbugs [28] if this is not done in a thread-safe manner.

The third modification is related to memory, specifically global memory. By default, pthread devices share one global memory. this is very useful between devices since one device can read the same buffer that another wrote to. This is however not very representative when it comes to scenarios where the redundant devices have their own global memory. In order to get a better representation, a build option has been added to PoCL to disable this shared memory.

4.3.2. Reference counts

Instances of objects in OpenCL keep count of how many references there are. Once this count reaches zero, the object is freed. These counts are vital for resource management and objects interacting with each other. When writing for the PoCL runtime, this count will need to be explicitly incremented. failing to do so will cause an object to be freed prematurely and cause a segmentation fault later when the program tries to access it.

In our implementation of TMR, the redundant devices share as many resources as possible. This includes the input buffers and kernel. In order to prevent an input buffer or kernel from being freed too soon, each redundant device increments the reference count of that object and decrements it when done.

A good test to see if the reference-count bookkeeping and concurrency are working correctly is to repeatedly enqueue the kernel. Some issues will not be noticeable when executing a kernel once. However, bugs will often start to manifest when repeatedly calling *clEnqueueNDRangeKernel* with new data in a loop. This kind of execution is common in video processing, where work the same work is done on a frame-by-frame basis. The reason this kind of application is such a good test is that it requires the program state to be valid before and after executing the kernel. Resources not freed will eventually fill up the available memory and resources freed too early will cause the program to crash when accessed.

4.4. Voting kernel

So far we have discussed the system so that voting can happen, but not how the voting happens. For this there are a number of different ways of doing this. The benefit of TMR is that it reduces the vulnerability of the whole system to one point. This means that the voting needs to be more resistant to faults happening. By making sure the system that is executing the voting kernel is resistant, we can be sure that the result is also correct. Ghosh [14] also makes a point that the host needs to be fault resistant because if that fails, computing the fault-corrected result has been for nothing. However, this falls outside the scope of this thesis.

Regardless of the specific device, there needs to be some form of the kernel to be run. On a bitwise level, the voting decision can be represented in a truth table, as can be seen in Table 4.4. It turns out that the logic for the correct result is simple, As can be seen in Equation 4.1. It is however harder to determine where the error is coming from.

$$\begin{aligned} R &= AB \cup AC \cup BC \\ error_A &= AB'C' \cup A'BC \\ error_b &= A'BC' \cup AB'C \\ error_c &= A'B'C \cup ABC' \\ error &= \neg(ABC \cup A'B'C') \end{aligned} \tag{4.1}$$

The fact that keeping track of where an error is coming from has an effect on the performance of

A	B	C	R	error	$error_A$	$error_B$	$error_C$
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	1
0	1	0	0	1	0	1	0
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	0
1	0	1	1	1	0	1	0
1	1	0	1	1	0	0	1
1	1	1	1	0	0	0	0

Table 4.4: Truth table for TMR. A, B and C are inputs while R is the desired result. $Error_*$ indicates which input deviated from the rest.

voting. Not only is there more logic to determine where the fault is coming from, but there also needs to be atomic increments of error counts. Since our goal is to keep the overhead as low as possible, we have opted to only implement the correcting logic.

Due to the simple nature of this voting, it is also possible to create a hardened kernel that is more resistant to errors. This version of the kernel executes the voting twice and then compares the result. If the results of the voting differ, then the voting is done again until both results are the same. This hardens the kernel against errors happening while voting. To make sure the comparison of the voting results goes well, there is a second check to make sure that it is still the same, only then will the result be written to the output buffer.

Due to the optimizations that modern compilers do, this double voting is normally optimized out. This makes sense from a perspective where no errors can happen as the results will be the same then. But the hardened kernel has been designed for scenarios where this is possible. Therefore, to stop the compiler from applying optimizations, we declare that our data is volatile.

4.5. FPGA voter

Instead of running a voting kernel on a device, it is also possible to have a device with dedicated hardware for voting. By using the AlmaIFv2 device driver of PoCL, it is possible to do the voting on an FPGA. The high IO bandwidth and the possibility of Direct Memory Access (DMA) make FPGAs an interesting platform to do voting. The task of voting is very suitable to be streamed since data access can happen in a linear fashion. This is beneficial since FPGAs generally do not have much on-chip memory forcing buffers to be small if they are to fit in said memory.

By using DMA, It is possible to read data from RAM, bypassing any CPU overhead. This also allows any FPGA design not to be constrained by the small on-chip memory. The AlmaIFv2 driver has the possibility to dedicate some RAM on the host that the device can use as its own memory. Using DMA, this data can then be fed to the logic that implements the kernel. The DMA controllers Xilinx provides in their Vivado design suite are fast [29]. It allows for data to be fetched in bursts instead of individually. And by using a dedicated DMA controller, we decouple the data retrieval from the computation. This allows the next set of data to be fetched while the current data is being used.

The downside to using these DMA controllers is that the data from RAM needs to be physically contiguous. This is different from how OSes typically allocate memory. Typically an OS will allocate a number of virtual addresses that are contiguous but are physically spread around on RAM. Fortunately, Xilinx does provide an API for their Linux images to allocate memory contiguously. The AlmaIF device driver can then work with this to create buffers in this region which can then be read and written to by the DMA controllers.

The PoCL repository provides an example of C code that can be used by Xilinx Vitis HLS to create an Intellectual Property (IP) block that can be used in Xilinx Vivado in a block design. This example demonstrates how the AlmaIFv2 interface works and the packet design used for communication. While this example works, it does have some shortcomings for our use case. The first is that it doesn't use DMA, but instead reads values using an AXI bus. This is not a problem for applications with more complex access patterns, but this does not allow us to leverage the benefits of DMA. The second issue is that Vitis is not able to pipeline the main kernel loop, leaving performance on the table. And finally, since the control mechanism is in the same IP block as the kernel execution code, the clock speeds are

tied together. This forces both the control and kernel execution to run at the lower of the two speeds. This again can leave performance on the table if the kernel code can run faster.

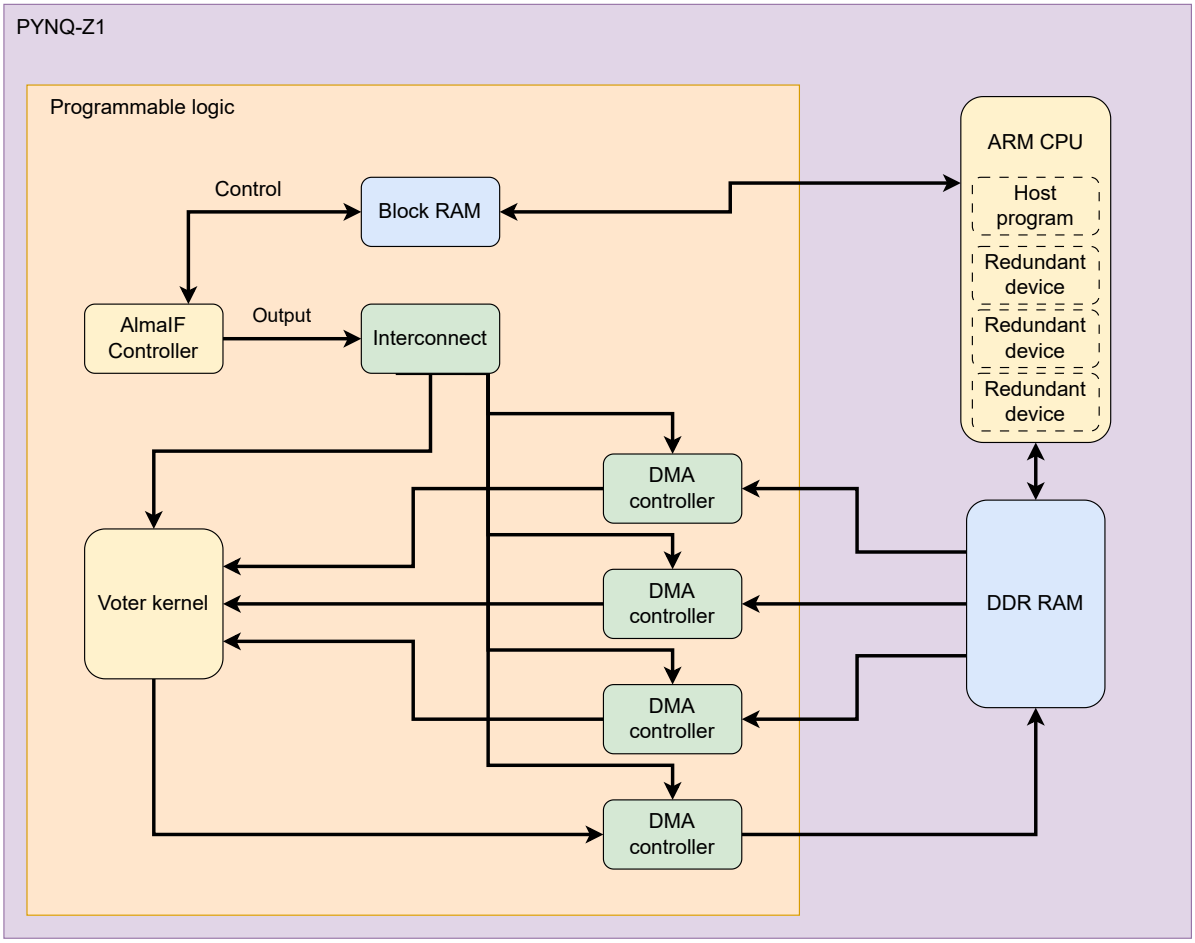


Figure 4.2: A Hardware design of an AlmaIFv2 voter device on a PYNQ-Z1 SBC.

To address the issues identified with the example code, we come up with a new design that can be seen in Figure 4.2. We have implemented this on a Digilent PYNQ-Z1 single-board computer (SBC). This SBC contains an SoC with both two ARM cores and an FPGA. This FPGA is referred to as programmable logic (PL) by Xilinx. In the figure, components that are on the FPGA are shown in the rectangle marked with ”programmable logic”.

Figure 4.2 also shows the full TMR setup of our solution. In this case, the ARM CPU runs the redundant devices as well as the host code. All the devices read and write buffers to the shared DDR RAM.

In this design, the control (AlmaIF controller) and kernel logic (Voter kernel) have been separated into two different IP blocks. By having the kernel in its stand-alone IP block, Vitis is able to pipeline the voting mechanism. For the TMR voting kernel, Vitis is able to synthesize a pipelined implementation that takes three clock cycles to compute the result while also starting a new comparison every cycle. This greatly increases the throughput compared to a nonpipelined version.

In this design, the voting kernel can continue to run as long as it is fed with data by the DMA controllers. The AlmaIF controller is still in control. It decodes the packets sent to it by the AlmaIF driver. The AlmaIF driver sends these packets by writing them to a memory-mapped address of block RAM. The AlmaIF controller controls the other IP blocks by writing to control registers on the other IP blocks. By using an interconnect IP block, the control packets from the AlmaIF controller are routed to the correct IP block.

The AlmaIF controller turns on the DMA controllers and indicates from which starting address to read the required number of bytes. It also tells the DMA engine that does the writing where to write the

results from the voter kernel IP to. It also turns the voter kernel IP block on. In this design, there is only one kernel implemented, but in a design that implements multiple kernels, being able to turn Kernel IP blocks on or off can be useful. Once the kernel and all the DMA controllers signal that they are done, the AlmaIF controller then signals to the driver that it has completed the work and PoCL can continue to the next stage of computation.

Another benefit of having the kernel logic in a separate IP block is that the clock speed is separated from the control logic. The benefit of this can be seen in the estimated maximum clock speeds that Vitis estimates. In the case of the voter kernel IP, Vitis estimates a maximum clock frequency of 1007 MHz, which is well above the maximum frequency of 250 MHz the Z1 provides by default. The AlmaIF controller IP instead has an estimated maximum frequency of 137 MHz. In practice, the clock frequency is set to the maximum speed that Vivado can successfully synthesize the design for. And in this case, that is the maximum speed the DMA controller that writes to the DDR RAM can go, which is around 142 MHz. This is a decent bump up from 100 MHz which is the speed the AlmaIF controller is running at.

5

Experiments and results

In this chapter, we evaluate the implementation and discuss its performance. When evaluating an implementation, the results also depend on the context, therefore we also discuss the hardware and the parameters used to compile the code. In our case, there are two systems that the experiments have been run on. One is a more conventional x86 computer while the other is an ARM and FPGA-based single-board computer (SBC). We will therefore start by describing the setup, followed by the experimental results, and finish with a conclusion.

5.1. x86 setup

This section describes both the hardware as well as the software setup of the x86 computer system used in the experiments. The hardware has the biggest effect on performance, but the software used can be key in getting the most out of the hardware. We, therefore, start with the hardware and follow this up with the software.

5.1.1. Hardware setup

The x86 computer consists of two server-grade CPUs. This means that the computer has two NUMA (non-uniform memory access) nodes, one for each CPU. Each node is connected to four RAM modules. If one NUMA node wants to access RAM that is connected to the other node, the data will need to be sent over an interconnect. Therefore, this memory access will take longer. To keep the access times consistent between iterations, the tests have been done isolated to one NUMA node. This isolation is done with a program called *numactl*. The *numactl* program allows one to isolate the execution of a program to run on a set of explicitly defined cores as well as only allocate memory locally on the NUMA node that those cores belong to. The final details of components can be seen in Table 5.1.

Component	Part	Details
CPU	Intel Xeon E5-2630 v4	10 cores with SMT at 2.20 GHz
RAM	64 GB DDR4 RAM	2133 MHz, 2 ranks, quad channel

Table 5.1: Details of the x86 computer used in the experiments.

While the Intel Xeon E5-2630 V4 launched in the first quarter of 2016, it still has a considerable number of cores. With 10 physical cores and 20 virtual cores due to Simultaneous Multithreading (SMT), its core count is still comparable to many of the modern-day server processors. Although with a base frequency of 2.20 GHz and a max turbo of 3.10 GHz, it is clocked 0.10 to 0.40 GHz lower than these modern-day processors. And while modern CPUs can benefit from architecture improvements, the E5-2630 has the AVX-512 instruction extensions, allowing data to be vectorized up to 512 bits long. This is useful for parallel programs such as the kernels of PoCL.

The RAM that the CPU has access to consists of 64 GB of DDR4 RAM. The RAM consists of four modules of 16 GB each. It is clocked at 2133 MHz, which is the highest speed supported by the E5-2630. This is slower than the maximum speed supported by modern-day counterparts, which generally

support memory up to around the 3000 MHz mark. However, due to the quad-channel memory configuration, it can still achieve high throughput. Using the STREAM benchmark [30], we were able to get an indication of this throughput. In the array copying task, of STREAM, the system was able to achieve a throughput of 30 GB per second.

5.1.2. Software setup

The experiments were done on the Kubuntu flavor of Ubuntu Linux 22.04. The software was compiled using the GNU C compiler (GCC) version 11.3.0, while the kernels themselves were compiled using LLVM version 14.0.0. When preparing for a compilation we use CMake to configure the project depending on what features are needed, but in every case, we make use of the *release* configuration. This is done to make sure that the code is compiled with high levels of optimization and no debug flags. During development, the PoCL code base was continuously synced. This means that the version of PoCL used is version 3.1.

5.2. PYNQ-Z1 setup

The other hardware used is the Digilent PYNQ-Z1 SBC [31]. The heart of the board is the Xilinx ZYNQ XC7Z020-1CLG400C System on a Chip (SoC). This SoC contains both two ARM cores and an FPGA. It comes with its own Linux image and popular libraries for ARM architectures. This SBC is an effective way to prototype embedded systems. The ARM cores combined with the FPGA make it ideal for low-power, efficient applications. General computations can be done on the ARM cores, while specialized operations can be done on the FPGA.

Compared to the X86 setup, PYNQ-Z1 is on the opposite side of the computing spectrum. While the X86 setup focuses on performance, the PYNQ-Z1 is geared toward efficiency. Therefore, testing on the PYNQ-z1 allows us to get better coverage over the potential deployment configurations of PoCL.

5.2.1. Hardware setup

The hardware details of the PYNQ-Z1 can be found in Table 5.2. The PYNQ-Z1 comes with a dual-core ARM Cortex-A9 CPU. This CPU is of the ARMv7L architecture which is a 32-bit architecture. It is clocked at 650 MHz, but comparing this frequency to the X86 setup does not give any metric of relative performance since the architectures are so different.

The CPU is connected to the memory controller which manages the 512 MB of DDR3 RAM. The RAM chip is connected over a 16-bit wide bus and has a clock frequency of 525 MHz. Since it is double data rate RAM, data can be transferred twice every clock cycle. This means that the clock frequency is effectively doubled and means that the theoretical maximum memory bandwidth is 2100 MB per second. However, in practice, this bandwidth is not reached.

The SoC also contains a low-power FPGA. It consists of 13,300 logic slices that can be configured to provide the requested functionality of the FPGA. It also contains 630 KB of block RAM, referred to as BRAM. This BRAM is fast, but limited in size.

The CPU and FPGA can communicate with each other through AXI interfaces. The PYNQ-Z1 has six AXI3 ports. Two of these are general-purpose master ports that can be used by the CPU to control the programmed FPGA. The other four ports are high-performance slave ports that can be used by the FPGA to interface with other components of the SoC, such as reading data from the DDR3 RAM.

Care has to be taken with the access that the FPGA has. Unlike running code that an OS manages, The FPGA does not have safeguards with regard to memory access. If for example, due to a programming error, the DMA engine writes data to the wrong address, the entire system can crash. Fortunately, on the PYNQ-Z1, a reboot of the system is usually enough to recover.

Componet	Part	Details
Memory	512 MB DDR3 RAM	16 bits wide at 525 MHz
CPU	Cortex-A9	dual-core at 650 MHz (ARMv7L)
FPGA	equivalent to Artix-7	13,300 logic slices and 630 KB of block RAM
Interface	six AXI3 ports	four high-performance slave ports and 2 general purpose master ports

Table 5.2: Hardware details of the Digilent PYNQ-Z1 board.

5.2.2. Software setup

The PYNQ-Z1 comes with its own software stack. To start off, it comes with its own version of Linux called PynqLinux. The version used during the experiments is PYNQ version 2.7.0 which is based on Ubuntu 20.04. The software provided here is older than on the x86 machine. As such, the GCC compiler is version 9.3.0 and the LLVM used here is version 10.0.0. Also in this case the code used in benchmarking was compiled in a release configuration. For PoCL, we use the same code as the x86 setup and is therefore version 3.1.

5.2.3. Cross-compilation to ARM

When compiling the pthread driver on the PYNQ-Z1, memory is the limiting factor. With earlier builds on the PYNQ-Z1, the system would run out of memory and fail. Ironically this happens specifically when compiling the code responsible for compiling kernels. The 512 MB of RAM plus a 512 MB swapfile was enough to compile even while compiling the code single-threaded. Increasing the size of the swapfile can alleviate this problem, however, this will wear out the SD storage card causing it to break after a couple of uses.

The solution we settled on was to cross-compile PoCL on the x86 machine for the ARM architecture. To do this, we used QEMU (Quick EMUlation) [32] and Docker [33] to create a virtual ARM environment on the x86 machine. QEMU was used to emulate an ARM CPU with the same hardware architecture as the PYNQ-Z1, although it had just as many cores as the x86 machine. With QEMU, we were able to solve the difference in hardware architectures.

The software on the x86 machine is not compatible with ARM architectures, therefore we used Docker. Docker allows one to create an isolated container with software that is different from the host that Docker runs on. We used Docker to create a container that contained software that is both compatible with ARM and close enough to what is found on the PYNQ-Z1. As a basis for the container, we used an ARM 32-bit Ubuntu 20.04 image. This is the same Ubuntu version the PynqLinux image is based on as well as the same architecture of the CPU. In the container, we installed the required build tools to compile PoCL. However, it turned out that the CMake program that was available on the software repositories contained a bug that would cause it to fail while running on 32-bit ARM architectures. Therefore, we compiled CMake from the source and this resolved the problem. Using this modified container, we were able to compile PoCL and the pthread driver.

However, when the time came to benchmark the code, this cross-compilation was not used. For whatever reason, possibly due to upstream code changes, the code compiled successfully. While compiling single-threadedly, the memory and the swapfile would fill up significantly, but not enough to hang the system. Cross-compilation is however still useful as it wears out the SD-card less.

5.3. x86 runtime overhead

In order to see what kind of overhead the implementation introduces, we measure the execution times of different setups and we compare normal executions to various fault-tolerant ones. This experiment also allows us to compare different implementations of the bitwise TMR kernels and see which ones are best suited to a given architecture.

5.3.1. MAD experiment setup

In the overhead experiment, we measure the time it takes for data to be written to input buffers, work to be done and finally reading the result back from the host. This process is repeated a hundred times and then the mean value is then taken. The first iteration often takes longer than the subsequent rounds and therefore is considered a warmup round. For this reason, we drop this measurement in the calculations.

The work kernel in the setup is a multiply-add (MAD) operation done on 32-bit integers. This workload includes two arithmetic operations and three 4-byte memory accesses per integer, putting the arithmetic intensity at $\frac{1}{6}$ operations per byte. This low arithmetic intensity will put an emphasis on data transfers while executing the redundant kernel, but not influence the process of bringing the input buffers to the device.

This is essentially a worst-case scenario. In a non-redundant setup, relatively little time is spent doing calculations on the data compared to moving data from and to the host. Since the TMR implementation needs to move data to and from each redundant device, a lot more data is transferred

compared to a non-redundant setup. This means that the relative runtime overhead of a MAD program with TMR is larger than in more computationally intense scenarios.

When it comes to creating TMR voting kernels, one of the design aspects is the size of the data type that a work item works on. Depending on the underlying hardware architecture, this can have an effect on the speed of fetching data and operations. Since the kernel makes a bitwise comparison of the data type, the results will be the same regardless of how many bits the data type has.

By repeating the same experiment on different input sizes, we can see the effect it has on the overall time of TMR execution. These input sizes range from 2^{20} (1,048,576) up to $10 * 2^{20}$ (10,485,760) with increments of 2^{20}. These sizes were chosen so that they were big enough to not be heavily affected by background processes on the system. It should be noted that this input size directly corresponds to the size of each of the two input buffers. So the total sum of input data is twice that of the input size

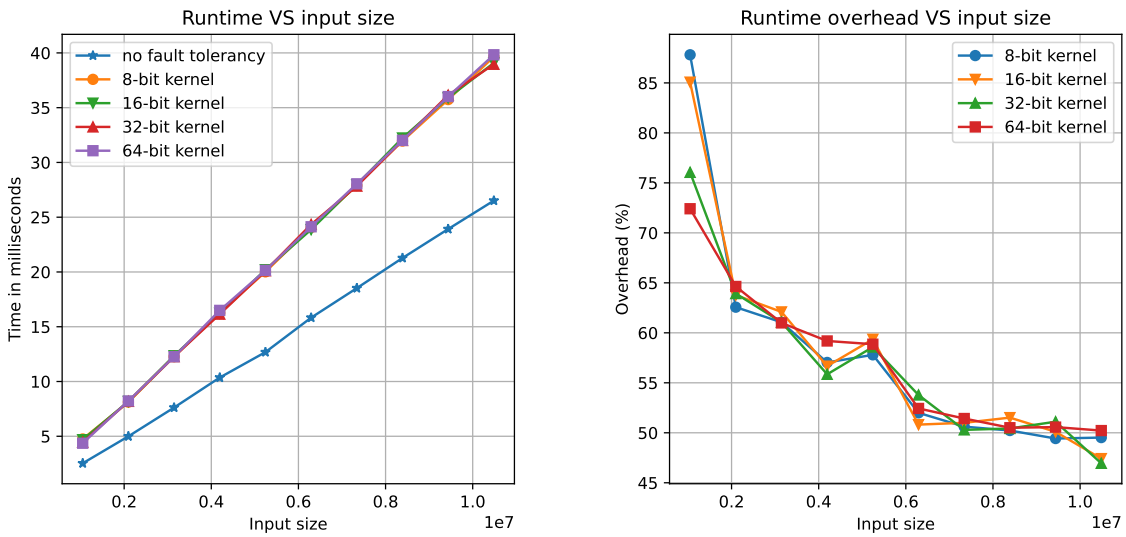
5.3.2. Shared memory TMR

The first voting kernel to be examined is a base TMR kernel that only does bitwise comparisons on the output buffers of the redundant kernels. Both the redundant and voting device are pthread devices and share the same global memory. This eliminates a buffer transfer between the two types of devices.

This setup minimizes data transfer times. Since the redundant devices share the same global memory and input buffers are read-only, it means all redundant devices can read from the same buffer and this buffer does not need to be physically copied to each device. Each redundant device will still write to a different buffer. But since the voting device also shares the same global memory, no buffer transfers between devices are needed and it can read from each redundant output buffer. Compared to a non-redundant setup, there are the same number of buffer transfers. However, since all redundant devices share the same computing hardware, the computational time takes a hit.

The results of this configuration can be seen in Figure 5.1. Subfigure 5.1a shows the average runtime for different setups using variations of the voting kernel compared to normal execution. The difference between each variation is the size of the data type that is the argument to the voting kernel.

At first glance, it looks like there are only two lines plotted, the normal execution denoted by star markers and a fault-tolerant setup. However, this is the result of all variations of the voting kernel being close to the same performance. This is due to PoCL being able to vectorize the work, thereby making the size of the data type irrelevant. This fact is confirmed when disassembling the binaries PoCL generates of the kernel. The assembly code contains AVX-512 instructions which the CPU has support for.



(a) The time it takes for different kernel configurations to execute in a TMR fashion compared to normal execution.

(b) The factor of time spent compared to normal execution.

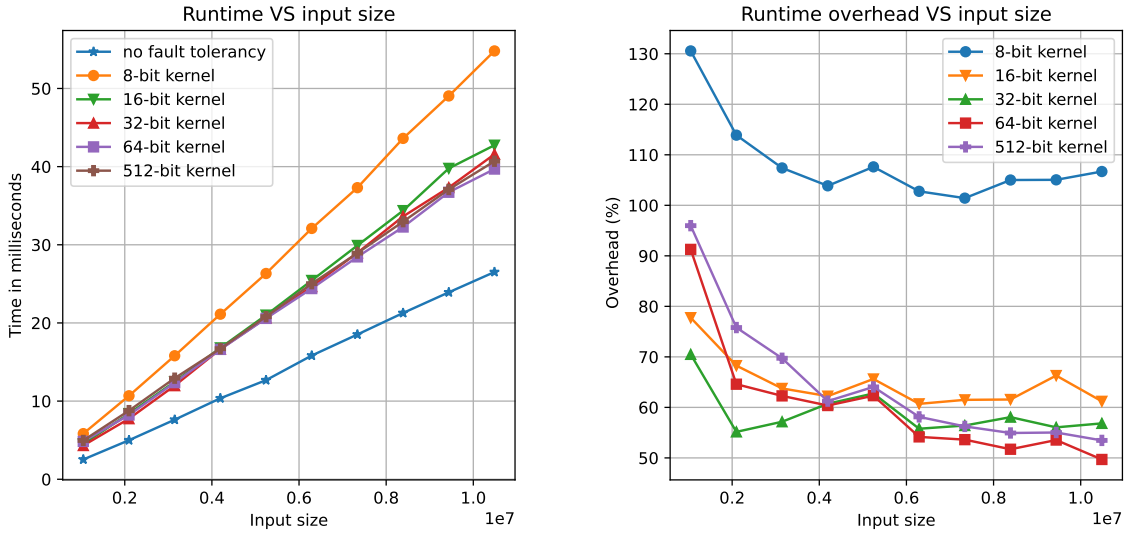
Figure 5.1: Performance metrics of different kernel configurations in the triple modular redundant implementation on an x86 platform.

Subfigure 5.1b shows a normalized view of the runtime overhead of each of the setups. The Y-axis shows the percentage of extra time it takes for a TMR implementation to complete. This figure shows a bit more variance between different kernel variations in the beginning but less toward the end. The average overhead across all kernels and inputs is 57 percent more than normal execution. The graph also shows the line flattening out at the end where the average of the last four entries is 50 percent.

Even though in this benchmark scenario there is minimal runtime overhead from transfers, the average fault-tolerant implementation is still minimally 50 percent larger than having no fault tolerance. The voting kernel adds some overhead, but this is not the only source. Other sources include the extra computations done on each of the redundant devices. While in PoCL there are three pthread devices executing the redundant kernel, they all share the same hardware as the normal setup, therefore the sum of physical compute hardware is the same for both setups. But since the runtime overhead is not three times as large, it would mean that the normal setup is underutilizing the hardware. The third form of extra overhead comes from scheduling. The TMR setup is more complex and can incur some delay due to that.

5.3.3. Shared memory hardened TMR

Besides normal TMR execution, there are also hardened versions of the TMR kernel. These kernels do not make use of vectorization due to the use of volatile datatypes. These volatile datatypes prevent the compiler from optimizing the code. This lack of optimization leads to some interesting results shown in Figure 5.2. When looking at Subfigure 5.2a, we can see that the setup using the 8-bit voting kernel is significantly slower than the others. It becomes clear that the hardware performs suboptimally when doing operations on 8-bit datatypes compared to other datatypes.



(a) The time it takes for different hardened kernel configurations to execute in a TMR fashion.

(b) The factor of time spent compared to normal execution.

Figure 5.2: Performance metrics of different hardened kernel configurations in the triple modular redundant implementation on an x86 platform.

Compared to the previous benchmark, there is also a kernel making use of a vector datatype. This vector consists of 16 32-bit integers and is therefore 512 bits long (shown in the figure as the ”512-bit kernel” curve). This was done to see if there are any performance gains to be had using a datatype that is as wide as the AVX-512 extension. But as can be seen in the figures, there are no major differences.

Compared to the unhardened kernels, there is also more variance in the normalized graph of Subfigure 5.2b. A surprising result is just how close the average overhead comes to that of the unhardened kernels. the best-performing hardened kernel is the 64-bit one and on average has an overhead of 60 percent, 52 percent if only looking at the last four inputs.

5.3.4. Matrix multiplication experiment setup

The MAD runtime experiments showed a worst-case scenario, where the work of the redundant kernel is straightforward and fast to complete. It can be argued that this work is not suited to be accelerated by frameworks like PoCL, since the data transfer times will likely outweigh any gains in computing time.

A more representative workload that could be computed with PoCL is matrix multiplication. Matrix multiplication is used in many computationally intensive workloads such as computer vision or machine learning. It also is much more arithmetically intense, making the relative runtime cost of moving data to an accelerator less.

Calculating the arithmetic intensity of matrix multiplicating depends on the matrices in question. In these experiments, we are multiplying $N * N$ matrices, so N is the only variable needed to calculate the arithmetic intensity. To calculate one value of the resulting matrix, we need to calculate the dot product of two vectors of length N. This means that there are N multiplications and N additions per element in the resulting matrix. Since the resulting matrix is a $N * N$ matrix, the total number of operations is $2 * N^3$. As for the number of bytes accessed, The input matrices consist of 32-bit floating-point numbers and so does the resulting matrix. Therefore the arithmetic intensity can be calculated as:

$$arithmetic\ intensity = \frac{2 * N^3}{12 * N^2} \tag{5.1}$$

The range of N in the experiments ranges from 128 to 1280. For these values, the arithmetic intensity is 21.33 and 213.33 respectively. This is much higher than the MAD kernel, which is only $\frac{1}{6}$.

as mentioned, the input size for the matrix experiments ranges from 128 to 1280 with increments of 128. However, since the input size is only the dimension of the input matrix, the actual input buffer is the input size squared. This value is reflected in the X-axis of the graphs.

Just like with the MAD experiments, we time the duration that it takes to write data to input buffers, execute work and finally read the results back. The experiments were run one hundred times and the average was taken.

As for the kernel we use in our experiments itself, we use a straightforward multiplication kernel. There are no optimizations done, for example with regard to memory access. Also, we don't pass any local workgroup sizes and instead let PoCL determine them.

5.3.5. Shared memory matrix multiplication

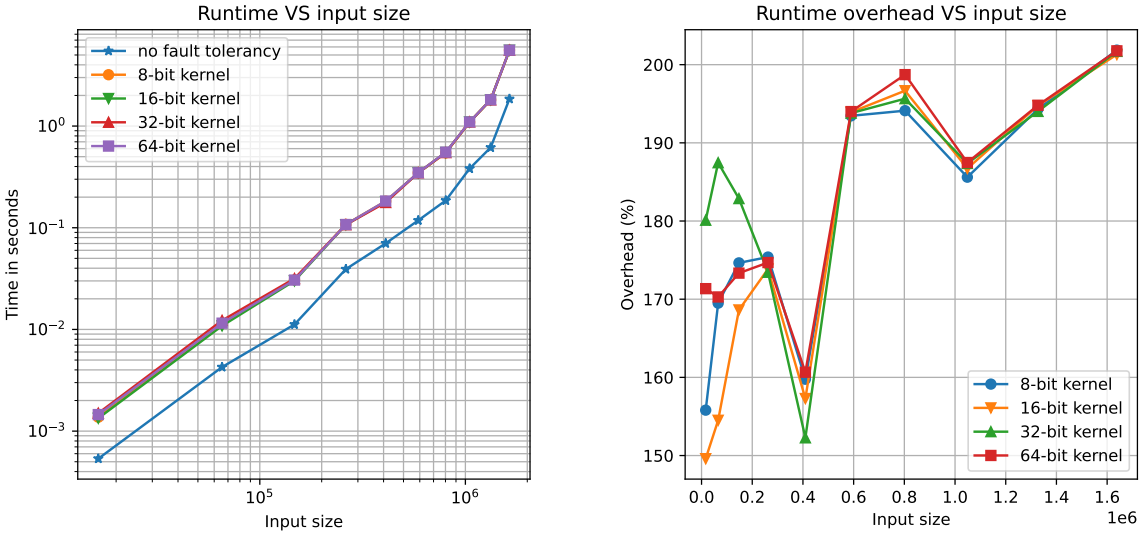
The first matrix multiplication benchmark involves three redundant pthread devices and one pthread voter pthread device. In this scenario, all devices share the same memory, so there is no need for extra data transfers. However, when looking at Figure 5.4a, we can see that the fault-tolerant implementation is noticeably slower. This is to be expected since the normal scenario has access to the same amount of physical hardware as the fault-tolerant scenario.

Subfigure 5.3b shows the extra overhead of the fault-tolerant implementation compared to normal execution. Since the fault-tolerant scenario executes the same work three times on the same physical hardware, one would expect that it takes at least three times the amount of time to execute the fault-tolerant implementation. However, it turns out that this is only the case for the largest input of 1,638,400 (1280^2). It is likely that the buffer transfer times compensate for some of the extra computation required for fault tolerance and that this effect is minimized for larger inputs.

What is surprising, however, is the sharp drop in overhead near marks 0.4 and 1.0. The benchmark runs a variation of the kernel for each input size before moving on to the next kernel. Therefore, these drops do not happen because of an outside disturbance, but rather systematically for each kernel variation. By executing the benchmark with tracing on, we are able to see how much time is spent in each part of the execution. From these traces, it seems that at these points, the buffer transfer times do not seem to increase much compared to those observed at the previous data point. It could be that for these input sizes, the data fits well with the physical hardware and therefore is utilized more efficiently than with other input sizes.

5.3.6. Separate memory matrix multiplication

In the previous experiment, all devices shared the same memory and hardware. In this experiment, the devices do not share any memory, forcing data transfers between devices. This is more representative of different physical hardware running OpenCL.



(a) The time it takes for different kernel configurations to execute in a TMR fashion.

(b) The factor of time spent compared to normal execution.

Figure 5.3: Performance metrics of different voting kernel configurations with a matrix multiplication redundant kernel on the x86 platform.

On top of enabling separate memory for each device, we also limited the number of cores available to the normal execution run. This was done to simulate more hardware for the fault-tolerant scenario. We used *numactl* to allocate 9 cores (18 threads) for the fault-tolerant execution and 3 cores (6 threads) for the normal execution.

The results can be seen in Subfigure 5.4. In Subfigure 5.4a we can see that for small input sizes, there is a noticeable overhead due to relatively much time being spent transferring the data buffers. However, this difference diminishes as the input size increases. This is due to the computing time of the matrix multiplication kernel increasing faster with input size than that of the buffer transfers.

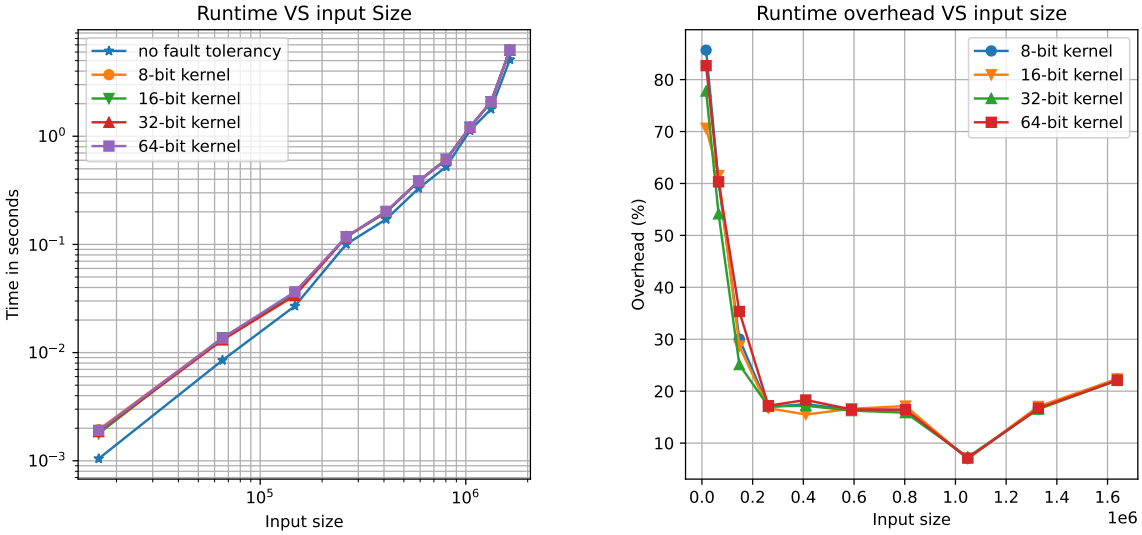
Subfigure 5.4b confirms this trend of transfer times becoming more insignificant as the input sizes increase. From inputs of size 262,144 $(512 * 512)$, the overhead stops dropping off heavily. There is a surprising drop in overhead for inputs of size 1,048,576 $(1024 * 1024)$. Like with the shared memory matrix multiplication experiment, data points for each kernel variation are gathered for the range of inputs before moving on to the next kernel variant, so this is not due to outside influence.

By utilizing the tracing functionality of PoCL, we are able to get some inside into what is happening. By examining traces for inputs of size 802,816 $(896 * 896)$ and 1,048,576 $(1024 * 1024)$, we noticed that while the average execution time of the voting kernel increased with a larger input, it did not increase with the same factor. PoCL aims to be performance portable and it is possible that for the input size of 1,048,576, the hardware was able to be more efficiently utilized while voting.

By taking the average overhead of all the kernels, we can get an indication of what the average overhead is of such a configuration. The overhead graph becomes more horizontal at the fourth data point of 262,144 $(512 * 512)$. The average overhead from the fourth data point is 17.7 percent if we exclude the entry at 1,048,576 and only 16.2 percent if we don't exclude it.

5.4. ARM runtime overhead

Besides profiling the runtime overhead on x86 hardware, experiments were also done on the 32-bit ARM hardware. This was done on the PYNQ-Z1 single-board computer. Doing so gives us the possibility to test and evaluate the performance of other hardware that PoCL may be run on. The PYNQ-Z1 also gives us the possibility to utilize its FPGA with the AlmaIF driver to create a custom voting accelerator.



(a) The time it takes for different kernel configurations to execute in a TMR fashion.

(b) The factor of time spent compared to normal execution.

Figure 5.4: Performance metrics of different voting kernel configurations with a matrix multiplication redundant kernel where the devices do not share any memory on the x86 platform.

5.4.1. MAD experiment setup

The experiments run on the PYNQ-Z1 board are similar to the X86 machine. The same code has been used for both the host and the kernels as with the x86 hardware. The input sizes are also the same and range from 2^{20} (1,048,576) up to $10 * 2^{20}$ (10,485,760) with increments of 2^{20}. The only major difference is the sample size, which has been reduced to fifty instead of one hundred.

5.4.2. Shared memory TMR

Even though both hardware use the same code, the results are slightly different, as can be seen in Figure 5.5. The first noticeable difference can be seen in Subfigure 5.5a. In this subfigure, it can be seen that the 8-bit kernel performs noticeably worse than the other kernels. This is similar to what was observed with the hardened 8-bit kernel on the x86 hardware and is most likely caused by the hardware being suboptimal for such small datatypes.

When looking at the normalized overhead shown in Subfigure 5.5b, we can see that on average the overhead is larger here. In this case, the best-performing kernel is the 64-bit kernel, which on average has an overhead of an extra 113 percent overhead. It is slightly unexpected that the 32-bit kernel performs worse with 119 percent overhead since the CPU has a 32-bit architecture. However, it is still much better when compared to the 8-bit kernel, which on average is 17 percent slower than the 64-bit kernel.

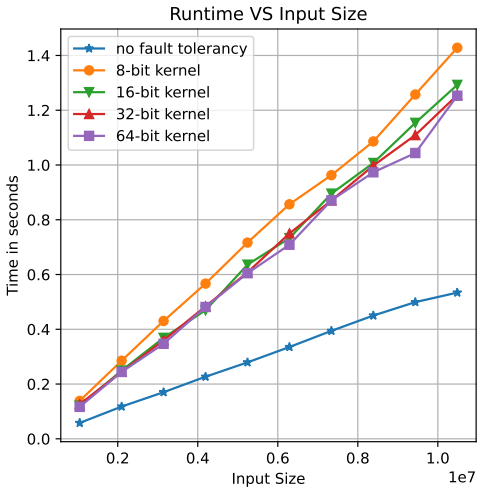
5.4.3. shared memory hardened TMR

Taking a look at the hardened kernels shown in Figure 5.6, we can see that now the 8-bit kernel is not the worst-performing kernel. In fact, the 16-bit kernel is slightly worse than the 8-bit kernel. This is most likely again due to the compiler being prevented from optimizing the code.

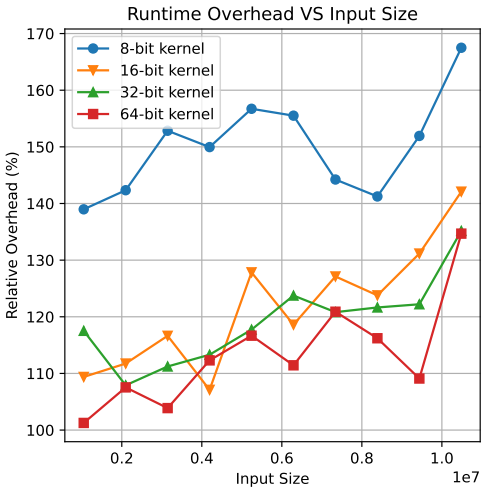
By looking at the normalized overhead in Subfigure 5.6b, we can see just how much the difference in performance is. In this case, the best-performing kernel is the 32-bit kernel, which on average is an extra 154 percent slower. The worst-performing 16-bit kernel is on average an extra 304 percent slower.

5.4.4. AlmaIF TMR

Besides voting using CPU devices on the PYNQ-Z1, it is also possible to utilize the FPGA available. For this, we use the AlmaIF device in PoCL. The hardware layout is described in Section 4.5.



(a) The time it takes for different kernel configurations to execute in a TMR fashion compared to normal execution.



(b) The factor of time spent compared to normal execution.

Figure 5.5: Performance metrics of different kernel configurations in the triple modular redundant implementation on the PYNQ-Z1 platform.

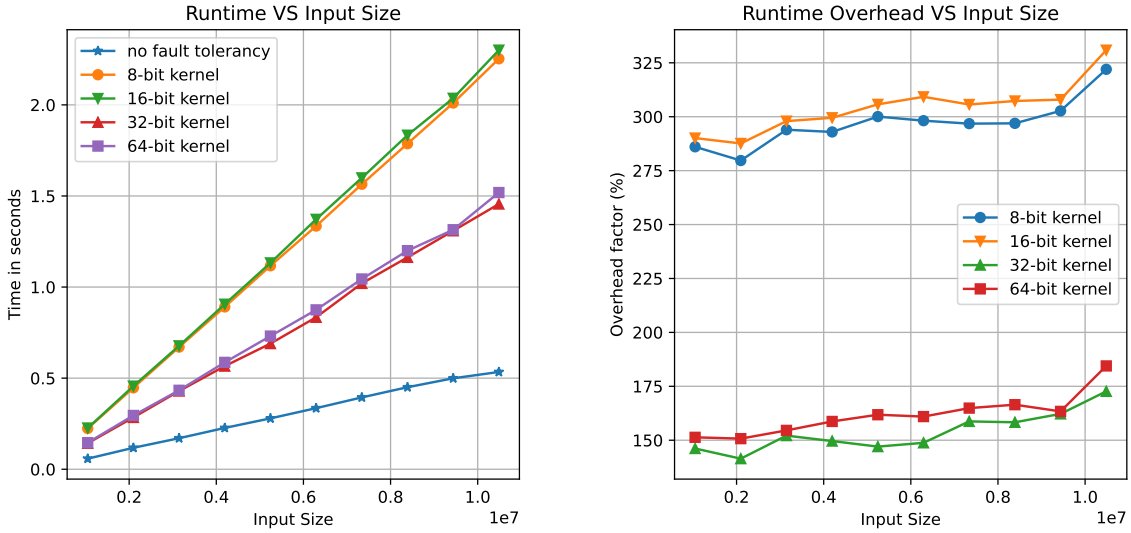
For this experiment, the setup is different compared to the other ARM scenarios. First off, due to memory limitations related to the external region on the AlmaIF device, the maximum input size was set to 2^{19} (524,288). The range of input sizes starts at 2^{11} (2048) and scales exponentially. Furthermore, the run has only been repeated ten times for the AlmaIF voter while the others have been run fifty times. To see how well it performs against voting kernels on the CPU, we have also plotted a 32-bit kernel and a 32-bit kernel with separate global memory.

Looking at the results in Figure 5.7, we can see that the AlmaIF device is actually slower than the pthread voting device with shared memory. This worse performance is due to the overhead associated with transferring result buffers from the redundant devices to the AlmaIF voting device. To confirm this, a setup was also run with a pthread voting device with separate global memory. And as indicated in Subfigures 5.7a and 5.7b with the ”sep mem pthread kernel” plot, this is extra transfer introduces a significant amount of overhead. Overall, the AlmaIF device has an average extra overhead of 176 percent while in this case, the pthread device with or without shared memory has an extra overhead of 96 and 279 percent, respectively.

While overall, the AlmaIF device was not faster than a pthread voter with shared memory, it still is faster at comparing results than a pthread device. To find out how much faster, another run was made with an input size of 524,288 and profiling enabled. Figure 5.8 shows an example of one iteration and how much time is spent transferring data and executing kernels. Using these traces, it is possible to calculate the average amount of time spent voting. For the AlmaIF device, this resulted in an average time of 6.845 milliseconds while the pthread voting device spent an average time of 11.461 milliseconds voting. This means that the AlmaIF device is 67 percent faster than the pthread device.

It is also possible to analyze the IP block design and see figure out the minimum amount of time it would take for the voting IP to process all the data. By adding an integrated logic analyzer (ILA) to the AXI interfaces of the voting IP, we can see how the IP behaves. On a frame of 8192 items of the ILA, the IP will occasionally stall irregularly but otherwise produce a result every clock cycle. This confirms the one-cycle initiation interval reported by Vitis HLS, but also likely indicates that the DMA controllers are oversaturating the available bandwidth. The theoretical minimum time spent by the voting IP can be calculated with Equation 5.2. In this equation, $initial\_delay$ is the number of cycles that are required to process one input item, while $initiation\_intervale$ is the number of cycles before the IP can start processing another input. Given that the IP runs at a frequency of 142 MHz and processes 524,288 packets of data, that puts the minimum time spent at 3,692 milliseconds

$$time = \frac{initial\_delay + (N-1) * initiation\_interval}{frequency} \tag{5.2}$$



(a) The time it takes for different hardened kernel configurations to execute in a TMR fashion compared to normal execution.

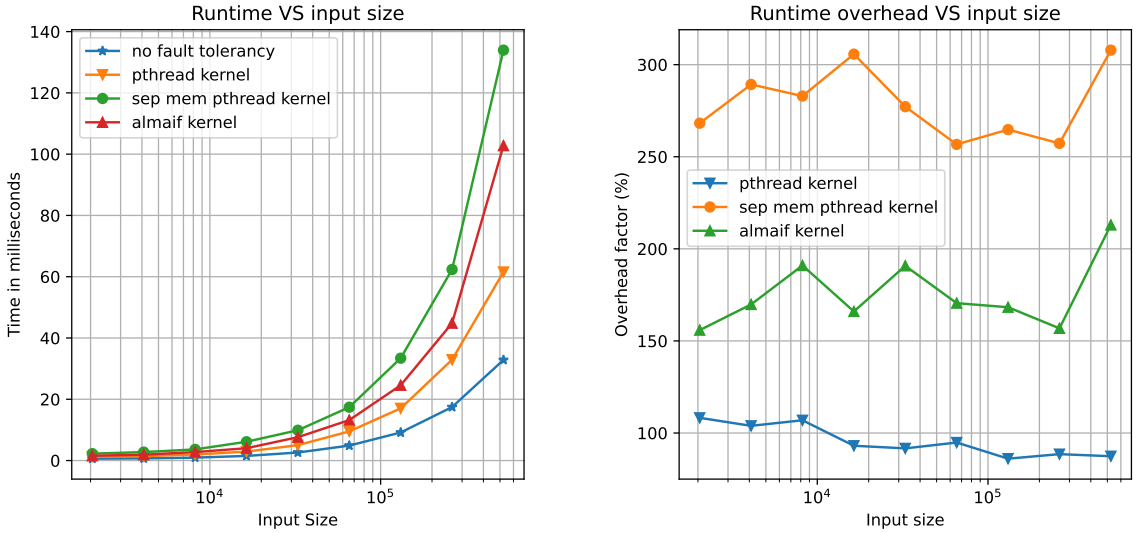
(b) The factor of time spent compared to normal execution.

Figure 5.6: Performance metrics of different hardened kernel configurations in the triple modular redundant implementation on the PYNQ-Z1 platform.

To see what kind of throughput the RAM can provide two tests were run to approximate the limit. The first benchmark was the stream benchmark [30] which benchmarks the throughput of RAM while performing simple tasks such as copying data from one array to another. By enabling multithreading, the best-recorded throughput was 1025.5 MB per second, or 604.1 MB per second if it uses a single thread. This is far below the maximum speed of the memory indicated in the reference manual provided by Digilent [31] which says that the memory has a throughput of 2100 MB per second. Opening the ZYNQ SoC in Vivado shows that the DRAM frequency is 525 MHz and the effective DRAM bus bandwidth is 16 bits, but since it is Double Data Rate (DDR) memory, the frequency is effectively doubled, Thereby reaching 2100 MB per second. The second test to approximate the limit involved using the PYNQ Python API to time the transfer of two DMA controllers writing to and receiving from a FIFO IP block. Even with the overhead associated with Python applications, the highest recorded throughput reached as high as 1113 MB per second, making this slightly faster than the stream benchmark.

Given the minimum time that the voting IP would need to process the buffers, we can calculate the maximum bandwidth required. the AlmaIF device reads three buffers of 524,288 32-bit integers and writes to one buffer; making maximum bandwidth required $\frac{524{,}288*4*4}{3.692*10^{-3}} = 2272MB/s$. This is above the theoretical maximum DDR RAM bandwidth and helps to explain the occasional stalling shown in the wave diagrams captured by the ILA. The PYNQ-Z1 uses a Zynq-7000 SoC and consulting the technical reference manual [34] of this SoC shows that the AXI ports of the FPGA need to go through an interconnect and a memory controller in order to get access to the DDR RAM. This will limit the available bandwidth since the memory controller implements some form of Quality of Service (QoS) preventing the DMA engines from starving other processes of memory access. Nevertheless, the AlmaIF voting device displays a high data transfer. With an average time of 6.845 milliseconds, the AlmaIF device has an average throughput $\frac{524{,}288*4*4}{6.845*10^{-3}} = 1226MB/s$. This time includes all control overhead of the device driver, likely making the actual throughput of the IP block itself even higher.

The technical reference manual [34] may also provide insight into why the stream benchmark is slower than the speeds on FPGA devices. The memory controller has four access ports. One of these ports is connected to the CPU while two others are connected to the interconnect that connects the AXI ports available on the FPGA. It is possible that this access port is a bottleneck while the FPGA does not suffer from this since it has twice the number of access ports.



(a) The time comparison of different 32-bit voting kernels on the PYNQ-Z1 board compared to normal execution.

(b) The factor of time spent compared to normal execution.

Figure 5.7: Performance metrics of different voting kernels in the triple modular redundant implementation on the PYNQ-Z1 platform.

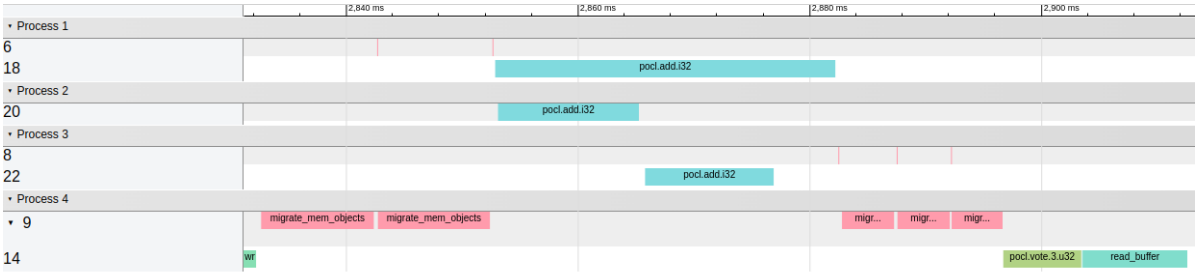


Figure 5.8: A trace from one iteration of a setup with an AlmaIF voting device.

5.5. Ease of use for end-user

In order to determine how easy it is for an end user to use the implementation, we use the number of lines of code in the application as a proxy to the complexity of the application. To compare the number of lines of code, we take one of the example programs provided by PoCL and augment it to also allow for fault-tolerant execution. For this, we chose to use *Example0* which implements a MAD operation in OpenCL and compares the results of that with a set of results calculated on the host.

The analysis of *Example0* can be seen in Table 5.3. To count the lines of code, we excluded brackets, comments and blank lines. The results show that the code specific to fault tolerance is 24 lines. A large percentage is due to error-handling code that is called every time after a function is called. In total, there are 7 function calls for the fault-tolerant code compared to the two function calls for the normal specific code.

The biggest source of lines however is the code that both modes share. This code comes down to 84 percent of the total lines of code and shows just how much code can be reused by a fault-tolerant setup. And when looking at the total code size of the normal code vs fault-tolerant code, the fault-tolerant code is only 11 percent bigger.

5.6. Robustness

In order to test the Robustness of the implementations, we have created a setup that can artificially insert errors. The setup works by inserting an extra kernel between a redundant device and the voter.

Section	Lines of Code	Percentage
fault-tolerant code	24	13%
normal specific code	6	3%
shared lines	153	84%
SUM:	183	100%
Total fault-tolerant code	177	97%
Total normal specific code	159	87%

Table 5.3: Analysis of the code after adding fault tolerance.

This kernel will randomly change a value to simulate an error occurring. OpenCL does not provide any pseudorandom built-in functions, therefore, the kernel does this by implementing a pseudorandom algorithm based on the one described in the book *The Art of Computer Programming, Volume 2* by Knuth [35]. The seed used is defined by $seed = initial\_seed * id^2$ where id is the index of the data currently being examined. By multiplying the id with itself, we create more variance between incremented ids. The $initial\_seed$ is different per redundant device and is used to create differences between each redundant device. By doing the same calculation on the host and comparing that to the result from PoCL, we can determine the number of errors that managed to get past. And by examining the redundant buffers, we can determine how many errors actually happened.

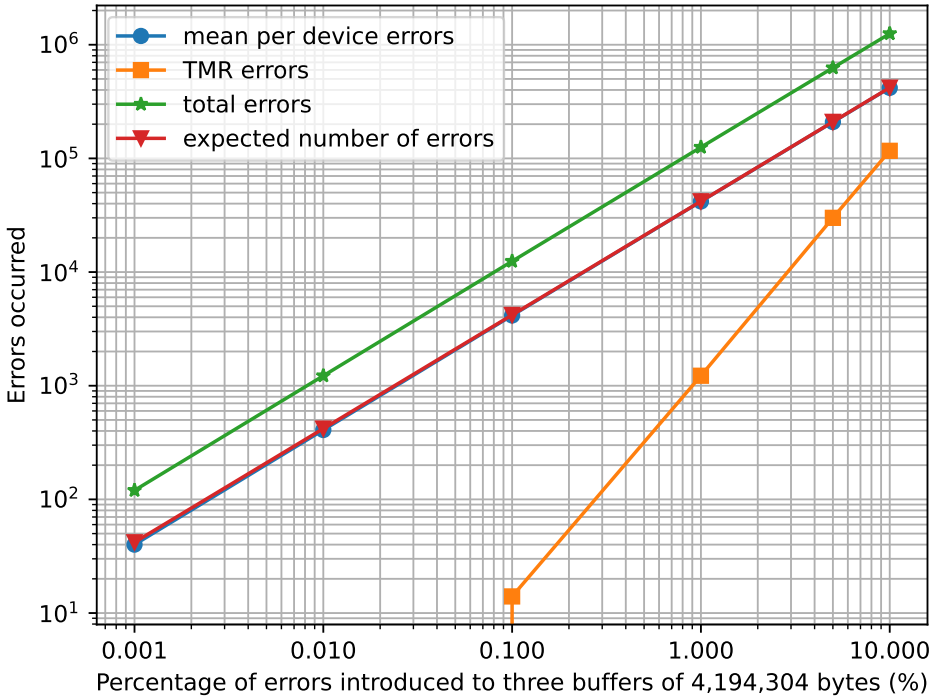


Figure 5.9: The percentage of errors introduced compared to the number of errors occurring when applying the TMR solution.

Figure 5.9 shows the results of applying the TMR implementation to an arbitrary workload. In the experiment, the output buffer contained 2^{20} 32-bit integers and we introduced errors on a byte level, so the total number of points of failure is just over $4.19 * 10^6$. On the X-axis, we show the percentage of errors we introduced to the buffers. As a baseline, we also draw the expected number of errors. This line is generated by multiplying the total size of the buffer by the given percentage. And as can be seen, the mean number of errors per device follows this line closely. The total errors line is the summation of errors for each redundant device. An interesting phenomenon here is that at an error rate that is lower than 0.01 percent, the TMR implementation caught all errors, while each device on average produced

408 errors (all three devices generated a total of 1223 errors at 0.01 percent).

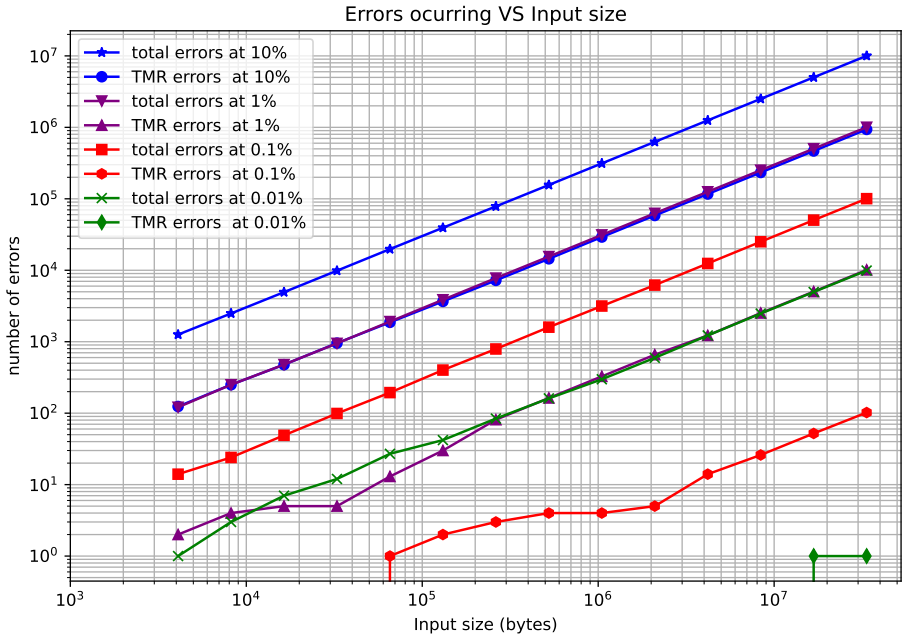


Figure 5.10: The number of errors occurring and being caught by the TMR solution for different buffer sizes and error rates.

To get more insight into the relationship between the error rate and input size, we ran a number of iterations with different error rates and input sizes. We counted the total number of errors occurring for all redundant devices and the number of errors still present after voting. The results can be seen in Figure 5.10. In this graph, lines of the same color share the same error rate. Datapoints that do not show up on the graph mean that there was no error occurring. We see this happening for the voter at lower input sizes and error rates.

Regardless of input size, the bitwise TMR voter is able to reduce the final number of errors manifesting. This reduction ranges from several factors for higher error rates, to orders of magnitude for lower error rates.

5.7. Discussion

In this chapter, we put the chosen implementation to the test and saw how well it performs according to the criteria we set forth in Chapter 3. These criteria are: runtime overhead, hardware overhead, ease of use, applicability and robustness. We will now discuss how well these criteria were met.

The first criterion looked at is runtime overhead. And as can be seen from the numerous experiments, the measure of this criterion depends a lot on the context. Factors that can influence this are the underlying hardware, configuration of the fault-tolerant implementation and the workload that is being made fault tolerant.

When considering x86 hardware, we ran a number of different experiments. By taking a MAD workload and a configuration where pthread devices share the same memory, we created the best configuration to minimize overhead in a worst-case scenario. In this case, we were able to achieve a runtime overhead of on average 57 percent. It should be noted that such a workload as only MAD op OpenCL is so straightforward that the biggest contributing factor to runtime is buffer transfers itself.

Devices that share memory are very likely to run on the same physical hardware, therefore we also investigated the overhead of hardened voting kernels. These hardened kernels are designed to still function even on less stable hardware and therefore more suitable to run on the same hardware as the redundant kernels.

We investigated the performance of these hardened kernels on both x86 and ARM architectures. On both architectures, small datatypes should be avoided as they are significantly slower. On x86, this

was the case for 8-bit datatypes while on ARM 16-bit datatypes should also be avoided. Overall on x86, the hardened kernels did not perform much worse than their non-hardened counterparts in the MAD scenario. The best-performing kernel was the 64-bit kernel which on average had an overhead of 60 percent. On ARM, the performance was worse when compared to non-hardened kernels. here too the 32-bit voting kernel performed best, with an average of 154 percent overhead.

For a more representative workload, we also measured the performance of a matrix multiplication workload in different scenarios. These workloads varied in their arithmetic complexity and ranged from 21.33 to 213.33. These experiments were done on x86 hardware. In the first scenario, the memory was shared between all pthread devices doing work. In this scenario, only 1280 by 1280 matrix multiplication caused the runtime overhead to go above 200 percent. Considering that the fault-tolerant scenario is computing the redundant kernel three times, this is better than expected. In the second scenario, the pthread devices do not share any memory and have to transfer data between devices. In this setup, more physical hardware was simulated for the fault-tolerant implementation by only allocating a third of the available cores to the normal execution. Once the matrices became big enough for the transfer time to become comparatively small, the runtime overhead decreased to as little as 17.7 percent.

Part of the runtime overhead experiments also included running a MAD kernel workload on a PYNQ-Z1. This demonstrates the performance of our implementation on an ARM hardware architecture and allows us to use a specialized AlmaIF voting device. Running a MAD kernel in our fault-tolerant fashion without isolating the memory of each pthread device produced more of an overhead compared to the x86 hardware. The best-performing variant was the 64-bit voting kernel variant, with an average runtime overhead of 113 percent. This larger increase in overhead compared to x86 hardware is most likely due to the weaker ARM cores taking relatively longer to complete the redundant computation of the MAD kernel. While other kernel variants are only slightly worse than the 64-bit kernel, the 8-bit kernel is significantly worse and should be avoided if possible. We also ran the MAD workload with an AlmaIF voting device and compared the performance to the pthread devices. It is not possible to share memory between AlmaIF and pthread devices, so buffer transfers are required. Since buffer transfers make up a significant part of the total execution times, the AlmaIF voting device is slower than the shared memory pthread voting device. The AlmaIF voting device has an average runtime overhead of 176 percent. However, a pthread device without shared memory is much worse with an average runtime overhead of 279 percent. Part of our analysis also included examining the bandwidth utilized by the AlmaIF device compared to what is possible from the CPU. Using the stream benchmark, we were able to get a bandwidth of 1050 MB per second, while the AlmaIF device displayed a bandwidth of 1226 MB per second while voting. This increase in voting and bandwidth shows the potential of such custom accelerators.

The runtime overhead is just one criterion of the criteria. the ease of use for the end user criteria aims to quantify how easy it is to apply fault tolerance to a program. For this, we counted the lines of code that were required to add fault tolerance to one of the example programs PoCL provides. This resulted in 24 lines of code specific to fault tolerance being added, which amounts to an increase of 11.3 percent to the total lines of code. This does not give a direct indication of how easy the new functions are to use but does show that the majority of the code can be reused.

The third criterion looked at was applicability. Here, we evaluate how many devices the implementation can be applied to. While there were no explicit tests done to show this, the experiments were run using both pthread and AlmaIF devices. On top of that, the experiments were run not only on x86 hardware but also on ARM hardware. This demonstrates that our implementation can work on multiple devices.

The final criterion was robustness. By injecting faults into the buffers of the redundant devices, we simulated faulty behavior. From our tests, we were able to demonstrate that our TMR implementation is able to significantly reduce the number of errors visible to the host program. Generally, the errors are either totally eliminated (for a lower number of errors), or the number of errors manifesting is orders of magnitude smaller than the actual number of errors happening (for larger number of errors).

6

Conclusions and recommendations

In this work, we set out to add fault tolerance to the open-source implementation of OpenCL called Portable Computing Language (PoCL). We do this by implementing a triple modular redundant (TMR) solution on the runtime level of PoCL. With these solutions, we evaluate the research questions. After that, we follow up with some recommendations for future work.

6.1. Conclusions

In this thesis, we formulated three research questions. We will now evaluate each of these questions.

6.1.1. Research question 1

The first research question we set out to tackle is: *"How can fault tolerance be added to PoCL that is easy to use for the end user?"* In order to accomplish this, we implemented TMR into the PoCL library. TMR involves computing the work three times and using a voting mechanism to correct any errors occurred.

The *clEnqueueNDRangeKernel* function declared by the OpenCL specifications has been extended to execute work kernels in a TMR fashion. From the perspective of the end-user, this function behaves the same with or without fault-tolerant execution. This fault-tolerant execution is triggered by passing a modified command queue, called a redundant command queue (RCQ), to *clEnqueueNDRangeKernel*. This RCQ contains multiple command queues that are used to schedule work to redundant devices. When creating the RCQ, the end user can configure which devices are used redundantly and which device is used for voting. The end user can also configure the voting kernel when creating the RCQ. However, if these voting kernel configurations are not passed, the function will default to presets.

The modifications visible to the end user have been kept as low as possible. This allows for a lot of code reuse. The end user does not have to change each OpenCL function to an equivalent one that is fault tolerant.

6.1.2. Research question 2

The second research question we addressed in this thesis is formulated as: *How can the runtime overhead of this implementation be kept to a minimum?* We have implemented TMR in PoCL and this brings with it a number of performance characteristics. Since TMR is able to correct any errors that occurred during runtime, the computation does not have to be repeated. This makes the runtime overhead constant regardless of the number of errors.

To keep the runtime overhead as low as possible, three redundant devices are needed to compute the same result in parallel. Our solution has been implemented on the runtime level of PoCL, so the redundant computations can be done on any desired device supported by PoCL. However, this generally means that data needs to be exchanged between redundant devices and the voting device.

Depending on the workload, the runtime is either bound by the transfer times or by the computing time. Our implementation can be applied differently to minimize runtime overhead depending on the workload and available physical hardware. In the case of the runtime being bound by the computing time, three devices can be allocated and the data can be moved between the redundant devices and

the voter. In our experiments with matrix multiplication on pthread devices that had isolated memory, this resulted in an average overhead of 18 percent on x86 hardware. If the application is bound by transfer times, this configuration will be significantly slower. However, on some devices, such as the pthread and AlmaIF devices, it is possible to divide the physical hardware into multiple devices. These devices share the same memory and therefore the voting device can directly read the same buffers the redundant devices write to. In our experiments with multiply and accumulate workloads, this resulted in an overhead that on average is 57 percent. We also ran the same test on a PYNQ-Z1 board with an ARM SoC. In this case, the best average performance was 113 percent.

In this thesis, we also implemented a custom accelerator that makes use of the AlmaIF driver in PoCL. By using high-level synthesis (HLS) we were able to design an accelerator that is implemented on the FPGA of the PYNQ-Z1 board. This accelerator is used as a voting device. In our experiments, we have shown that this device is 67 percent faster than the pthread voting device.

6.1.3. Research question 3

The third research question we addressed is: *"How can we preserve the flexibility of OpenCL while adding this fault tolerance capability?"* Our goal here is for our implementation to work with any of the devices supported by OpenCL as well as PoCL. Our implementation works on the runtime level of PoCL and therefore any device supported by PoCL can be used the same way without any modifications.

In our benchmarks, we have demonstrated this by combining redundant pthread devices with an AlmaIF voting device. We also ran our benchmarks on not only x86 hardware but on ARM as well.

We also provide different configuration options for the voting kernel. This allows the end user to tune the voting kernel to execute optimally to the hardware. For example, the end user can choose to use a 32- or 64-bit voting kernel on ARM instead of the 8-bit kernel that we showed performed worse. The end user can also choose to use a hardened voting kernel that employs techniques that mitigate errors happening in the voting process. The tradeoff is that this hardened kernel is slower than the normal kernel.

6.2. Future work

In this thesis, we focused on adding fault tolerance to PoCL in a way that is compatible with all devices that PoCL supports while also minimizing overhead. Part of our goal was to make it easy for an end user to apply this fault tolerance to their application. There is however always room for improvement.

Due to the large amounts of data being transferred, transfer speeds are important to overall performance. Currently, PoCL internally makes use of the *memcpy* function in C to copy data. While *memcpy* is a fast, optimized function, data still goes through the host CPU. This causes overhead and utilizes the CPU. A solution could be to employ direct memory access (DMA) for buffer transfers and thereby avoiding the CPU. This could be done using an FPGA with DMA units. PoCL already has the capability to interface with FPGAs and could control the DMA units with the AlmaIF driver. This could improve the transfer speeds on PoCL in general.

During our experiments, we showed the potential of having a voting device on the FPGA of the PYNQ-Z1. However, this implementation was limited due to the DMA engines only supporting memory transfers of physically contiguous memory. This forced us to make a buffer copy of the output buffers of the redundant pthread devices to the memory region of the AlmaIF device. By either allocating memory contiguously also for the pthread devices or by using DMA engines that do not have this requirement, the runtime overhead using the AlmaIF voting device (and thereby our solution as a whole) would be vastly improved.

Bibliography

[1] "Customized parallel computing (cpc) research group." (Apr. 2023), [Online]. Available: https://www.tuni.fi/cpc/.

[2] "Cross-layer cognitive optimization tools and methods for the lifecycle support of dependable cpsos." (Apr. 2023), [Online]. Available: https://cpsosaware.eu/.

[3] A. T. S. Bureau, "In-flight upset, 154 km west of learmonth, wa, 7 october 2008, vh-qpa, airbus a330-303," Dec. 2011.

[4] J. Uitto, *Offloading computation with a minimized opencl runtime from a nano drone*, eng, Informaatioteknologian ja viestinnän tiedekunta - Faculty of Information Technology and Communication Sciences, 2022.

[5] "The opencl specification." (Feb. 2023), [Online]. Available: https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

[6] "The opencl c specification." (Feb. 2023), [Online]. Available: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_C.html.

[7] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "Pocl: A performance-portable opencl implementation," *International Journal of Parallel Programming*, vol. 43, pp. 752–785, 2015.

[8] "Portable computing language." (Feb. 2023), [Online]. Available: http://portablecl.org/.

[9] "Clang: A c language family frontend for llvm." (Feb. 2023), [Online]. Available: https://clang.llvm.org/.

[10] "The llvm compiler infrastructure." (Feb. 2023), [Online]. Available: https://llvm.org/.

[11] "Ieee standard for information technology–portable operating system interface (posix(tm)) base specifications, issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, 2018. DOI: 10.1109/IEEESTD.2018.8277153.

[12] T. Leppänen, A. Lotvonen, P. Mousouliotis, J. Multanen, G. Keramidas, and P. Jääskeläinen, "Efficient OpenCL system integration of non-blocking FPGA accelerators," *Microprocessors and Microsystems*, vol. 97, p. 104 772, 2023, ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2023.104772.

[13] T. K. G. Inc. "Khronos vulkan registry." (Apr. 2023), [Online]. Available: https://registry.khronos.org/vulkan/.

[14] S. Ghosh, *Distributed Systems: An Algorithmic Approach* (Chapman & Hall/CRC Computer and Information Science Series). CRC Press, 2006, ISBN: 9781420010848. [Online]. Available: https://books.google.fi/books?id=aVjVzuav7cIC.

[15] H. Mushtaq, Z. Al-Ars, and K. Bertels, "Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems," in *2011 IEEE 6th International Design and Test Workshop (IDT)*, 2011, pp. 12–17. DOI: 10.1109/IDT.2011.6123094.

[16] H. Mushtaq, Z. Al-Ars, and K. Bertels, "A user-level library for fault tolerance on shared memory multicore systems," in *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2012, pp. 266–269. DOI: 10.1109/DDECS.2012.6219071.

[17] C. Engelmann, H. H. Ong, and S. L. Scott, "The case for modular redundancy in large-scale high performance computing systems," in *Proceedings of the 8th IASTED international conference on parallel and distributed computing and networks (PDCN)*, 2009, pp. 189–194.

[18] D. Tiwari, S. Gupta, and S. S. Vazhkudai, “Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems,” in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 25–36. DOI: 10.1109/DSN.2014.101.

[19] C.-H. Ho, M. de Kruijf, K. Sankaralingam, B. Rountree, M. Schulz, and B. R. de Supinski, “Mechanisms and evaluation of cross-layer fault-tolerance for supercomputing,” in *2012 41st International Conference on Parallel Processing*, 2012, pp. 510–519. DOI: 10.1109/ICPP.2012.37.

[20] D. Czajkowski, M. Pagey, P. Samudrala, M. Goksel, and M. Viehman, “Low power, high-speed radiation hardened computer flight experiment,” in *2005 IEEE Aerospace Conference*, 2005, pp. 1–10. DOI: 10.1109/AERO.2005.1559559.

[21] D. Czajkowski, M. Pagey, M. Goksel, and D. Bozek, “Single event effects (see) mitigation of reconfigurable fpgas,” in *20th annual AIAA/USU conference on small satellites*, 2006.

[22] F. S. Vainstein, “Error detection and correction in numerical computations by algebraic methods,” in *Proceedings of the 9th International Symposium, on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, ser. AAECC-9, Berlin, Heidelberg: Springer-Verlag, 1991, pp. 456–464, ISBN: 3540545220.

[23] R. W. Hamming, “Error detecting and error correcting codes,” *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[24] J. Li and Y. Wang, “Transient fault tolerance on multicore processor in amp mode,” in *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, IEEE, 2021, pp. 332–337.

[25] G. A. Reis, J. Chang, and D. I. August, “Automatic instruction-level software-only recovery,” *IEEE Micro*, vol. 27, no. 1, pp. 36–47, 2007. DOI: 10.1109/MM.2007.4.

[26] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, “A large-scale study of soft-errors on gpus in the field,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2016, pp. 519–530.

[27] I. S. Haque and V. S. Pande, “Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu,” in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 691–696. DOI: 10.1109/CCGRID.2010.84.

[28] E. S. Raymond, *The new hacker’s dictionary*. Mit Press, 1996.

[29] “Axi dma v7.1 logicore ip product guide (pg021).” (Apr. 2022), [Online]. Available: https://docs.xilinx.com/r/en-US/pg021_axi_dma.

[30] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[31] “Pynq-z1 reference manual.” (Mar. 2023), [Online]. Available: https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual.

[32] F. Bellard, “Qemu, a fast and portable dynamic translator.,” in *USENIX annual technical conference, FREENIX Track*, Califor-nia, USA, vol. 41, 2005, p. 46.

[33] “Docker: Accelerated, containerized application development.” (Mar. 2023), [Online]. Available: https://www.docker.com/.

[34] “Zynq-7000 soc technical reference manual.” (Apr. 2023), [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM.

[35] D. Knuth, *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Pearson Education, 2014, ISBN: 9780321635761. [Online]. Available: https://books.google.fi/books?id=Zu-HAwAAQBAJ.