Tampere University

Topi Leppänen

SCALABILITY OPTIMIZATIONS FOR MULTICORE SOFT PROCESSORS

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Prof. Pekka Jääskeläinen
M.Sc. Joonas Multanen
February 2021

ABSTRACT

Topi Leppänen: Scalability Optimizations for Multicore Soft Processors
Master of Science Thesis
Tampere University
Master's Degree Programme in Electrical Engineering
February 2021

The growth of single core performance and energy efficiency have been stagnating for decades. Multicore systems are an efficient way to do parallel computing at the level of threads. Additionally, specializing the processor architecture to better fit the application at hand is one approach to achieving the required energy and performance improvements.

TCEMC is a toolset under development at Tampere University which generates multicore application-specific instruction set processors. This thesis evaluates and improves the toolset. The toolset's ability to scale up the number of cores is tested by seeing how many cores can be fitted on a small field-programmable gate array device.

An 8-fold increase in performance is achieved with 24 cores, compared to the equivalent single core system. The external memory bandwidth can be utilized with 11.4% efficiency. The remaining bottlenecks of the multicore soft processors are highlighted which remain to be solved to unleash the full potential of customized multicore systems. The most important of these being the scalar access to the external memory, which is shown to be an inefficient way to utilize the external memory bandwidth.

Keywords: parallel computing, multicore, OpenCL, soft processor

TIIVISTELMÄ

Topi Leppänen: Moniytimisten pehmeiden prosessorien skaalausoptimointi
Diplomityö
Tampereen yliopisto
Sähkötekniikan DI-ohjelma
Helmikuu 2021

Yksiytimisten prosessorien suorituskyvn ja energiatehokkuuden kasvu ovat hidastuneet merkittävästi viime vuosikymmeninä. Moniytimiset järjestelmät ovat tehokas tapa suorittaa rinnakkaislaskentaa säikeiden tasolla. Prosessoriarkkitehtuurin räätälöiminen tiettyyn sovellukseen on toinen tapa saavuttaa vaaditut energia- ja tehokkuusvaatimukset.

TCEMC on Tampereen yliopistolla kehitettävä työkalu, jolla voidaan generoida moniytimisiä räätälöityjä prosessoreja. Tämä työ tarkastelee ja kehittää edelläan kyseistä työkalua. Työkalun kykyä kasvattaa ytimien määrää testataan pienellä kenttäohjelmoitavalla porttimatriisi-laitteella.

Työssä saavutetaan 8-kertainen suorituskyvun kasvu 24:llä ytimellä, verrattuna vastaavan yksiytimiseen prosessorin suorituskykyyn. Ulkoista muistikaistaa pystytään hyödyntämään 11.4% tehokkuudella.Työssä etsitään jäljelle jääneet moniytimisen pehmeän prosessorin pullonkaulat, jotka tulee ratkaista, jotta räätälöityjen moniydinprosessorien potentiaali saadaan hyödynnettyä paremmin. Tärkein näistä on ulkoisen muistin käyttö skalaarioperaatioilla, minkä näytetään olevan tehoton tapa hyödyntää ulkoista muistikaistaa.

Avainsanat: rinnakkaislaskenta, moniytimellisyys, OpenCL, pehmeä prosessori

PREFACE

The research for this work was done in Customized Parallel Computing research group at Tampere University. I would like to thank my coworkers for making the workplace a pleasant place to work. Special thanks to my supervisor Pekka Jääskeläinen and to a former coworker Kati Tervo for their consistent and excellent help during this research. Also, thanks to Joonas Multanen for providing useful feedback for the drafts of this thesis.

Finally, I would like to thank my wife Maria for graciously supporting me during this thesis and my studies. Also, thanks to my son Ruben for constantly reminding me of the importance of regular breaks.

This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 783162 (FitOptiVis). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Netherlands, Czech Republic, Finland, Spain, Italy. It was also supported by European Union's Horizon 2020 research and innovation programme under Grant Agreement No 871738 (CPSoSaware).

In Tampere, Finland, 12th February 2021

Topi Leppänen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction set Processor
AXI	Advanced eXtensible Interface
critical path	Combinatorial logic path that limits the clock frequency of the digital circuit.
DLP	Data-Level Parallelism
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
FU	Function Unit
GPP	General-Purpose Processor
HLS	High-Level Synthesis
ILP	Instruction-Level Parallelism
latency	The delay from the initiation of the operation to the completion of it.
LSU	Load-Store Unit
LUT	Lookup Table
RF	Register File
RTL	Register-Transfer Level
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SoC	System-on-Chip
SPMD	Single Program, Multiple Data
TCEMC	TTA-based Codesign Environment, Multicore
throughput	The amount of work done per a unit of time.
TLP	Thread-Level Parallelism
TTA	Transport Triggered Architecture

1 INTRODUCTION

The breakdown of Dennard scaling and the slowdown of the rate of progress predicted by Moore's law limit single-threaded performance of computer hardware [1]. Increasing the clock frequency and the performance of a single core is not as easy as before, so more attention has been focused towards more parallel computer architectures. Parallelism is a general concept which means that multiple computations are happening simultaneously. This means that a larger application is split into parallel parts which can be performed at the same time for the sake of efficiency.

There are a number of different approaches to creating computing hardware, some of which are presented in Figure 1.1. Choosing the correct approach is a trade-off between flexibility and performance. One way to evaluate the differences between these approaches is based on the level of *specialization* they utilize. Specialization means that the hardware is optimized for a certain task or a domain of similar tasks. At one end of the spectrum are the *fixed function accelerators*, which are often custom-made *application-specific integrated circuits* (ASIC) designed to perform certain task really efficiently in terms of energy and the amount of silicon area used. At the other end are the *general-purpose processors* (GPP). These are generally used as *central processing units* (CPU) in e.g. traditional desktop and mobile computers. They can execute many kinds of programs efficiently. In between these two approaches are the specialized processors. These represent a more hybrid approach, where the processor has been designed with a certain application or an application domain in mind, although it could actually execute any kind of program. The instruction set and the microarchitecture of the processor can be specialized to best accomplish a certain task. This level of specialization is called *application-specific instruction set processor* (ASIP). GPPs could also be thought to be specialized for their intended tasks, such as running an operating system or multimedia applications. However, ASIPs are generally specialized for a much narrower domain.

Another way to evaluate these different approaches is to look at how easily the task running on them can be changed. Processor-based approaches like GPP or ASIP make this really easy since only the program source code has to be changed and then recompiled. At the other end of the spectrum, an ASIC's task can't really be changed without redesigning the accelerator. The functionality is 'fixed' into the silicon and it can only be configured based on the configurability implemented at the time of manufacturing. However, a fixed-function accelerator doesn't always have to be physically implemented in silicon.

Field programmable gate arrays (FPGA) make it possible to change the logic design after the chip has been manufactured. In Figure 1.1 the FPGA box corresponds to these types of fixed-function accelerators implemented on an FPGA device. Fixed function accelerators (for ASICs and FPGAs) have traditionally been designed using *Hardware description languages* (HDL), such as Verilog or VHDL, to describe the circuit at *register-transfer level* (RTL). Lately there has been a lot of interest in so-called *High-level synthesis* (HLS), which can be used to create the hardware description directly from the program written in high level programming languages such as C or C++.

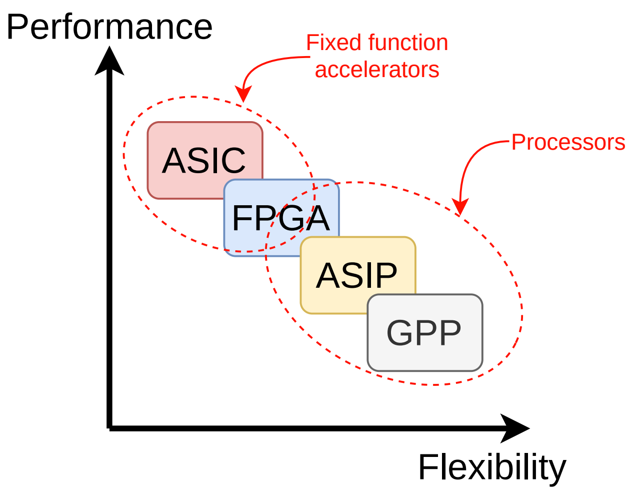


Figure 1.1. *Trade-off between flexibility and performance in digital processing systems.*

A *soft processor* is a processor running on the FPGA fabric, as opposed to a *hard processor*, which is a processor implemented in silicon as ASIC. The soft processor is used as a overlay between the user application and the FPGA hardware. This makes it possible to easily utilize the FPGA device with very limited hardware design expertise. Compared to HLS, porting complex software from GPPs to FPGA might be easier to do using a soft processor overlay, since some complicated program structures might be difficult to implement with HLS. Implementing ASIPs as soft processors makes it possible to iterate and easily change the hardware based on the specialization. So, in this way the ASIP can include only the resources that are most useful for a certain domain or application.

Having only a single core in the soft processor -system comes with limitations, even if the core itself has been highly specialized for the application. Even though the core might be able to execute multiple instructions in parallel, often there aren't that many independent instructions available to execute [2]. Additionally, critical signal paths inside a single core can become difficult to manage if the core is expanded to cover larger silicon area. Simply increasing the clock frequency is not always possible due to the generation of excess heat.

In addition, the memory performance has not increased as much as the performance of the processors [3]. This has lead to the memory bandwidth often becoming the limiting factor for performance. Having multiple threads of execution on the same chip can be an efficient way to utilize external resources, such as memory. 'Copy-pasting' cores in parallel is one way to create a multicore system with multiple threads of execution, each having its own separate resources.

Increasing the numbers of cores in a multicore system doesn't come without a cost. This work estimates the soft processor scalability of a *Multicore ASIP*-template (MCASIP) by increasing the number of cores. It can be assumed that some parts of the system become more complex as the core count increases. Some of these scale up easily with a basic 'copy-pasting things in parallel'-method. However, some can become performance bottlenecks by limiting the clock frequency of the system, or by excessively stalling the cores. The soft processor system is evaluated on an FPGA-device to find out how efficiently it can utilize the external memory bandwidth.

Chapter 2 of this thesis describes the basics of parallel computing architectures and programming. Chapter 3 describes how the *TTA-based co-design environment, multicore*-template (TCEMC) can be used to generate multicore systems. During this work, a few optimizations were made to the template. These are discussed in Chapter 4. The scalability of the TCEMC-template and the effectiveness of the optimizations are evaluated using a few benchmarks in Chapter 5. Chapter 6 describes two potential improvements to the TCEMC-template left for future research. Finally, Chapter 7 sums up the findings of this thesis.

2 PARALLEL COMPUTING

Types of parallelism are commonly categorized to three different types: *instruction-level parallelism* (ILP), *data-level parallelism* (DLP) and *thread-level parallelism* (TLP). Instruction-level parallelism means that the multiple instructions of a function can be evaluated at a same time using parallel hardware. Most commonly this is exploited by pipelining the instructions, so that the next instruction can start when the previous one hasn't yet finished. In addition to pipelining, the instructions can execute in parallel function units of a *multi-issue* processor. Parallelism is at the data-level when the same control flow can be applied to multiple data items at a same time. Many processors have vector instructions, which define some basic operation for an entire vector of data (e.g. vector addition for 8 integers at a time). Thread-level parallelism describes a coarser kind of parallelism, where there are multiple independent computation tasks executing in parallel. [2]

Computer architectures can be categorized based on how many instruction and data streams they have [4]. *Single instruction, single-data* (SISD) is a basic single-processor architecture, where a single processor executes one instruction stream operating on a single stream of data. *Single instruction, multiple data* (SIMD) still has a single instruction stream, but can operate on a multiple pieces of data (vectors) at a same time. This term nowadays most often refers to vector instructions of an instruction set [5]. *Multiple instructions, single data* (MISD) is often used to describe either streaming processors where a single data stream passes through multiple processors, or in redundant computing, where the same thing is computed multiple times for reliability reasons [6]. *Multiple instructions, multiple data* (MIMD) is an architecture with many independent instruction streams, and the streams can process different data streams. This is a broad category which includes both multicores and multithreaded processors. [7] A processor architecture is not limited to utilizing only one of these principles at a time. For example, many high-performance processors utilize both MIMD and SIMD, by having support for multiple threads and cores, and including SIMD vector instructions [8].

Single Program Multiple Data (SPMD) is a parallel programming concept which means that multiple pieces of the same program are operating on different pieces of data in parallel [7]. The programmer can describe some work (for example, the insides of a loop) as a *kernel*, which is then applied to an arbitrary amount of data.

This chapter presents the basics of multithreaded and multicore systems, and how they can be programmed. Finally, FPGA is presented as a way to implement parallel logic circuits.

2.1 Thread-level parallelism

A program can be split into tasks, which can then execute independently using their own instruction streams as *threads* [7]. Each thread must keep track which part of the program it is executing. It does this by having a program counter, which is an index to the instructions of the program. Threads often share the same memory space with each other, so they can communicate with each other by reading and writing to a piece of shared data. A program can be split into threads either by the programmer or the compiler. The threads can execute in parallel, taking advantage of MIMD hardware, either inside a single core or between multiple cores in a multicore system. [2]

Accessing a shared resource such as a shared memory creates resource contention between the threads. This resource contention leads to the interleaved access pattern shown in Figure 2.1. The red MEM boxes in the figure represent the external memory access and the white COMPUTE boxes represent a computation done using the values fetched from the memory. The threads could even be executing an identical program and be slightly offset from each other because of the resource contention. At any given time some threads can be accessing the memory while others are computing the results. This phenomenon is called *latency hiding*, where the memory latency of one thread is covered by another thread's computation.

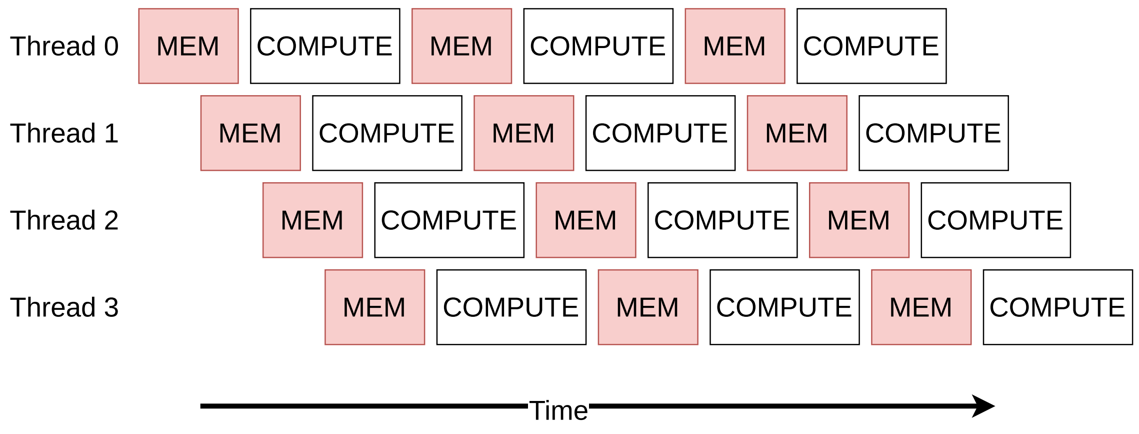


Figure 2.1. *Memory latency hiding in a system with multiple threads.*

2.1.1 Multithreading

All the resources of the processor core aren't utilized all of the time. To help with that, the core can support execution of multiple threads inside of it. This allows for most of the processor’s hardware resources to be shared between the threads. The threads have their own dedicated instruction streams, and therefore must have their own context registers (e.g. program counter, stack pointer), as there are instructions that depend on the previous instruction modifying the processor state. So, these parts of the core are duplicated and others can be shared between the threads. This is called *multithreading*. [7]

The threads can take turns to execute their instructions in the processor (temporal multithreading) The granularity of the temporal multithreading can be fine (cycle-level) or coarse (switch threads at certain high-latency operations, e.g. cache misses). Because the instructions from different threads have no dependencies with each other, they can fill up each other’s ’empty spots’ or ’bubbles’ (empty red boxes in Figure 2.2) caused by the long latencies of certain instructions or dependencies inside a single thread’s program. This helps to utilize the function units of the processor more efficiently. [7]

Modern superscalar processors have the ability to execute multiple scalar instructions in parallel inside a single core. This can also be used to execute instructions from different threads in a truly parallel fashion instead of just temporal multithreading (Figure 2.2). This is called *simultaneous multithreading* (SMT). [7]

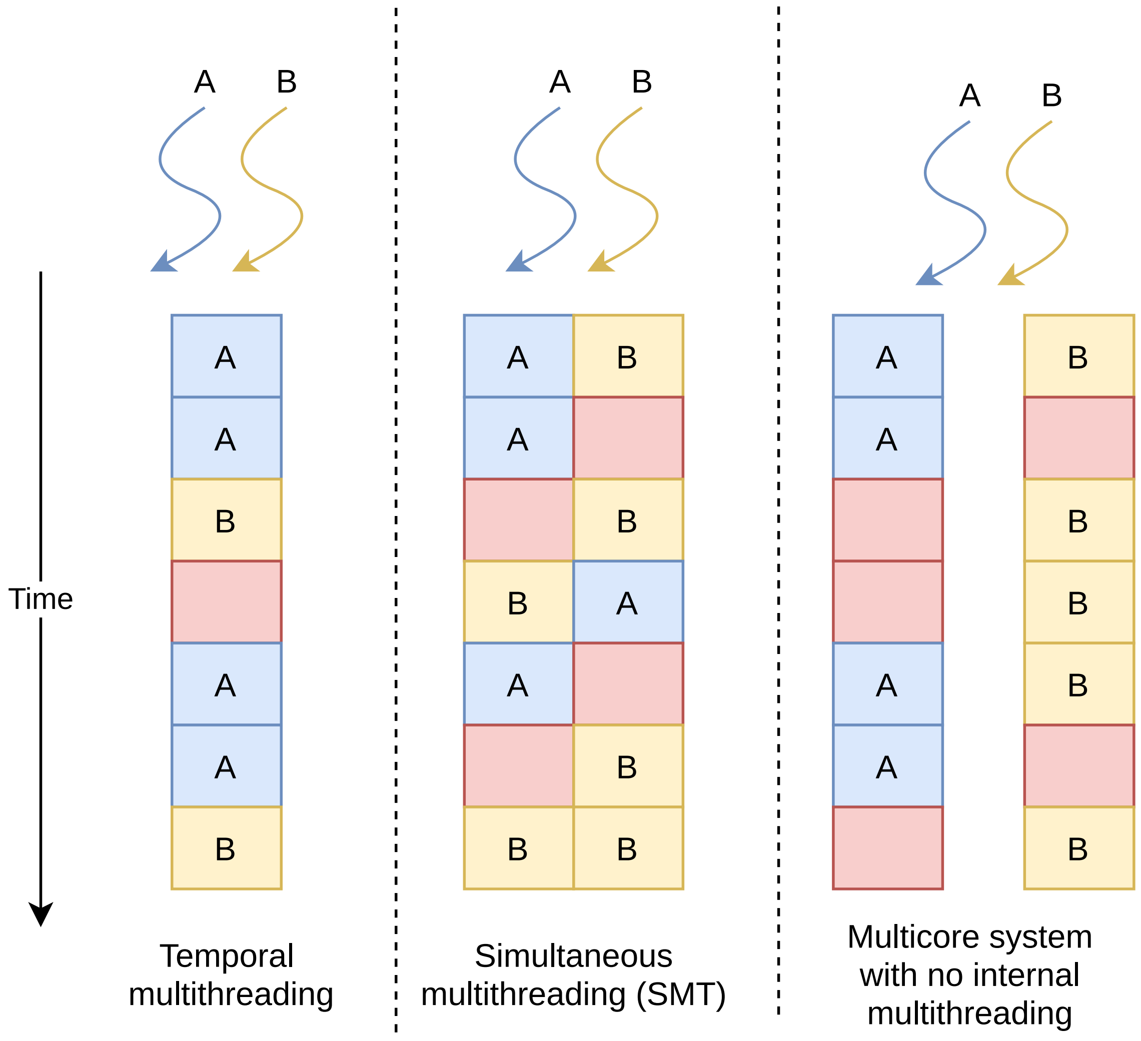

A
B
Time
A
A
B
A
A
B
Temporal multithreading
A
B
A
B
A
B
B
A
A
B
B
B
Simultaneous multithreading (SMT)
A
B
A
B
A
B
A
B
A
B
B
Multicore system with no internal multithreading

Figure 2.2. *Threads A and B running concurrently on different processor systems. Adapted from [7].*

2.1.2 Multicore systems

Multicore systems exploit the thread-level parallelism similarly to the multithreading, but now the core hardware is no longer shared between the threads. The cores are completely separate from each other with each having its own data and control paths. As can be seen from Figure 2.2, the multicore systems by themselves can be less efficient in terms of hardware utilization, since there are ’empty spots’ in their execution. Naturally the multicore systems could still utilize multithreading inside the cores.

The main challenge in developing the multicore system is the memory system [2]. The threads are most often defined and programmed to share the same address space, which means that all the cores must be able to access the same memory areas. Multicore systems with shared address space can be divided into two categories based on their memory system. *Symmetric multiprocessors* (SMP) have a single centralized memory that all the cores can access equally. The other configuration is *distributed shared memory* (DSM), where the memory components are physically distributed near each of the cores. All the cores can still access any location of the memory, but the memory access times are now non-uniform, since the desired location can physically exist far away in some different core's memory component. Because of this feature, the configuration is also called *Non-Uniform Memory Access* (NUMA). Both of these shared-address-space systems must have a complex cache coherency logic to ensure that the each of the cores still sees the same memory areas similarly, since the programs are defined to work with that assumption. [2] Memory address spaces don't necessarily have to be shared between cores, but it helps the programming effort, as all the threads see the same memory.

Maintaining the coherency of the shared memory system becomes more and more difficult as the number of cores is increased [2]. To combat this, in large core count systems the memory address space is no longer shared, which simplifies the hardware but requires a different kind of software to run on the system. This is often called distributed memory. As opposed to the DSM, now the memory is no longer shared by default. The distributed memory forces the programmer to think carefully which data should be shared and which parts can remain in local memories. This is fundamentally different to the DSM-system, where all the memory was available, but some parts of it just worked slower. Traditionally message-passing protocols have been used to communicate between the cores. [2] These kinds of systems have been used in cluster computing where the individual computers have been connected to each other as a network, and can communicate via traditional networking protocols such as Ethernet [7].

2.1.3 Heterogeneous multicore systems

Traditionally, all the cores of the multicore system have been identical copies of each other (homogeneous) to simplify the design and programmability of the system. However, multicore systems can also be heterogeneous, which means that some of the cores of the system are specialized to execute a specific task more efficiently (in terms of area, energy or execution time). An example of this is a *graphics processing unit* (GPU), which is specialized for graphics-related tasks. GPUs have also been made generally-programmable, meaning that they can be used to perform any computation. However, their architecture is optimized for data-parallel programs due to their main use being in graphics processing. [9]

Recent advances in *System-on-Chip* (SoC) design make it possible to have several different processors sharing the same physical chip (*Multi-Processor SoC* i.e. MPSoC). This allows parts of the memory system (e.g. caches) to be directly shared with very different kinds of components, such as traditional CPUs, GPUs, *digital signal processors* (DSP) and FPGAs. Heterogeneous systems require a lot from the compiler and the programmer to efficiently generate program code for these components. In a homogeneous system, programs could be compiled for every core with the same compiler, while often assuming the shared memory address space. In a heterogeneous system, all the cores often need their own compiler, and quite often even their own domain-specific programming language. [7] There are programming standards such as OpenCL, which try to solve this problem for the application developers by natively supporting all these different types of processing architecture in the same *application programming interface* (API) [10].

2.1.4 Single instruction multiple threads

Single instruction multiple threads (SIMT) is an execution model introduced by Nvidia to efficiently program GPUs [2]. SIMT consists of data-parallel lanes, similarly to SIMD, but it also has more advanced per-lane control than SIMD. It's a still *single instruction*-model, so one instruction is used to control all the lanes. On the other hand every lane is actually a restricted dependency-free thread, which can have independent control flow. But since all the lanes execute in lock-step according to the single instruction, some lanes are just automatically masked off based on the per-lane control flow. For example, if there are multiple lanes in the same SIMT-operation that take on different branches from an if-condition, the program executes both branches and the hardware takes care to automatically mask the lanes off that weren't supposed to take that branch.

GPUs utilizing the SIMT programming model can hide the external memory latencies by having much more data-items to compute than there are computing elements. Since every SIMT-lane is supposed to be completely independent from each other, the GPU can dynamically schedule any of these SIMT-instructions to execute. This can be used to hide the long external latencies to memory. SIMT-processing is efficient only for data-parallel programs, since there must be enough tasks to oversubscribe the processor this way. [2]

2.2 Parallel programming

The concurrent execution of threads means that they progress independently. In actual hardware they might even be executing on the same physical CPU, just separated in time, as described in Section 2.1. Even a single-threaded processor can execute multiple concurrent threads with the operating system's assistance. The operating system can take care of switching between the threads (*context switch*) to give the illusion of them both making progress at the same time. The parallel execution of threads can be thought of as a special case of concurrency where the threads are literally executing at the same time on parallel hardware.

2.2.1 Synchronization

The concurrent execution of threads presents synchronization problems when the threads ’cross over’ each other (e.g. accessing shared data). The parallel threads have similar synchronization problems which are solved with similar solutions than the more general concurrent threads-case.

Concurrent threads operating on different data with no dependencies between each other might not need to communicate with each other at all. However, practically all applications require that the threads can communicate or share data with each other. Concurrent modifications to the shared data can cause ’race conditions’, situations where the outcome depends on the semi-random order in which the instructions are executed in the concurrent threads. Because of this, *synchronization* is needed.

Figure 2.3 shows a simple example of how things can go wrong, when two concurrent threads operate on the same piece of data. In the figure, both threads are trying to increment the shared variable *value*. Incrementing a value has three steps: load the value from memory into a register; increment the value in *arithmetic logic unit* (ALU); store the incremented value back to the memory. Because the relative order of instructions between the threads is undefined, multiple outcomes are possible. If the threads happen to execute completely after one another, the *value* gets incremented twice, which was the intended outcome. However, it is possible that the execution of the threads is interleaved as shown in Figure 2.3, which leads to the *value* being incremented only once. Since the ordering is undefined, it’s possible that sometimes the program works correctly, and sometimes, seemingly at random, it can fail. To solve this problem, synchronization is needed.

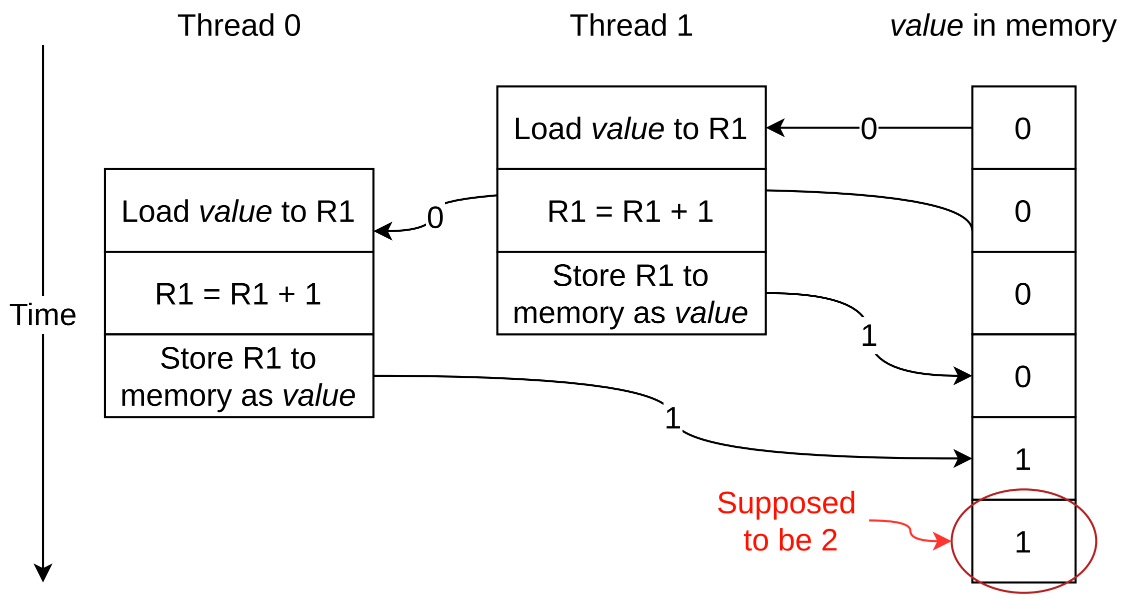


Figure 2.3. *Race condition when two threads try to increment the same variable.*

Atomicity means that a sequence of instructions executes in an 'indivisible' way from every other threads' points of view [7]. In Figure 2.3 this would mean that the sequence of load, increment and store would happen sequentially in such a way that another thread wouldn't be able to execute these same instructions at the same time. A sequence of instructions that only certain number of threads are allowed to execute at a time is called *critical section* [11].

Synchronization can be implemented using *synchronization primitives* such as locks or barriers. A lock can only be acquired by a one thread at a time, so all the other threads wanting to acquire it must wait until the initial thread releases it. This can be used to reserve a shared resource exclusively for a single thread at a time. [11]

A barrier is a point in the program that all threads must reach before any of them can continue. This can be used to ensure that some task is completely finished before any of the threads goes forward. To implement locks, barriers or other synchronization methods, a thread must be able to read and modify some shared state of the system atomically. [12]

Modifying the shared state of the system is not trivial and requires at least some support from the processor's instruction set and the underlying hardware. Basic load- and store-operations don't work, as they can't modify the memory atomically. Many modern processors include atomic memory-editing operations, which can read and edit a value in the memory atomically (atomic read-modify-write), meaning that during the execution of this instruction, no other thread can successfully operate on that same piece of memory. After executing the instruction, the thread can deduce whether it managed to modify the memory. Even if multiple threads try to execute this operation at the same time on the same memory address, only one of the threads can succeed. This is a feature guaranteed by the hardware. [2]

A similar method can be used that works with the combination of two instructions. The first instruction (load-link) reads some value from the memory, then immediately the second instruction (store-conditional) tries to save something to the same address. If the value in the memory had changed the meanwhile from the value it initially read, the store will fail. [2] All the described methods to implement locking operate on a shared memory, which can put a lot of stress on the shared memory system, including the cache.

2.2.2 Parallel programming methods

Parallelism can be extracted from a sequential program by the compiler or the developer. The compiler can analyze the program, and create parallel sections from certain parts of it. However, this can be problematic as the compiler must be able to prove that the semantics of the program don't change. In other words, it must still produce the correct results after parallelization. Some technical details of the programming languages might prevent this automatic extraction. Often, the developer must make changes to the program to help the compiler with the parallelization. The developer can also utilize pre-made parallel libraries of the language to utilize the parallel hardware. Even then, there are certain program constructs where the compiler is not able to extract parallelism, or there might not be ready-made libraries to use. In some cases, changing the programming language to one specifically designed for parallel computing might be the only way forward. [7]

There are at least three standard methods to create parallel programs. *Message passing* is a method where the threads are executing completely independently with their own separate memories. To share data with each other, they must follow a specific protocol. An example of a *message passing*-based programming method is *message passing interface* (MPI). [7]

The second one is fork-join parallelism, where the main program thread can spawn more threads to compute tasks in parallel. The threads can communicate with each other freely using a shared memory. When the threads finish their work, they will rejoin to their parent threads. [7] When creating the threads, the program must be split into clear tasks for each of the threads. The splitting can be done automatically (using e.g. OpenMP [13]) or manually by the developer who creates the threads.

The third approach is using a *data-parallel language*. These languages are specifically designed to express highly data-parallel programs, where there isn't too much dependency between the data items. An example of this would be an image-processing algorithm which operates on every pixel of the image separately. [7] The data-parallel languages map well to many-threaded GPUs but aren't exclusive to them. A common idea of these languages is to have a very large amount of relatively simple threads, which can execute in the hundreds of parallel threads available in GPUs. CUDA and OpenCL include ways to create these data-parallel programs, and therefore enable general-purpose computing on the GPUs. [9]

All of these methods are generally useful, but some are much more efficient on a specific hardware setup. The message passing is efficient when working with processors that don't have easy physical access to each other's memory components. Fork-join parallelism is quite easy for the programmer to understand and use, and it maps well to many programs consisting of both serial and parallel parts. Data-parallel languages have a high initial learning curve, but are often the most efficient (or only) way of programming GPUs for general computation. However, they also require for the problem itself to naturally be very data-parallel.

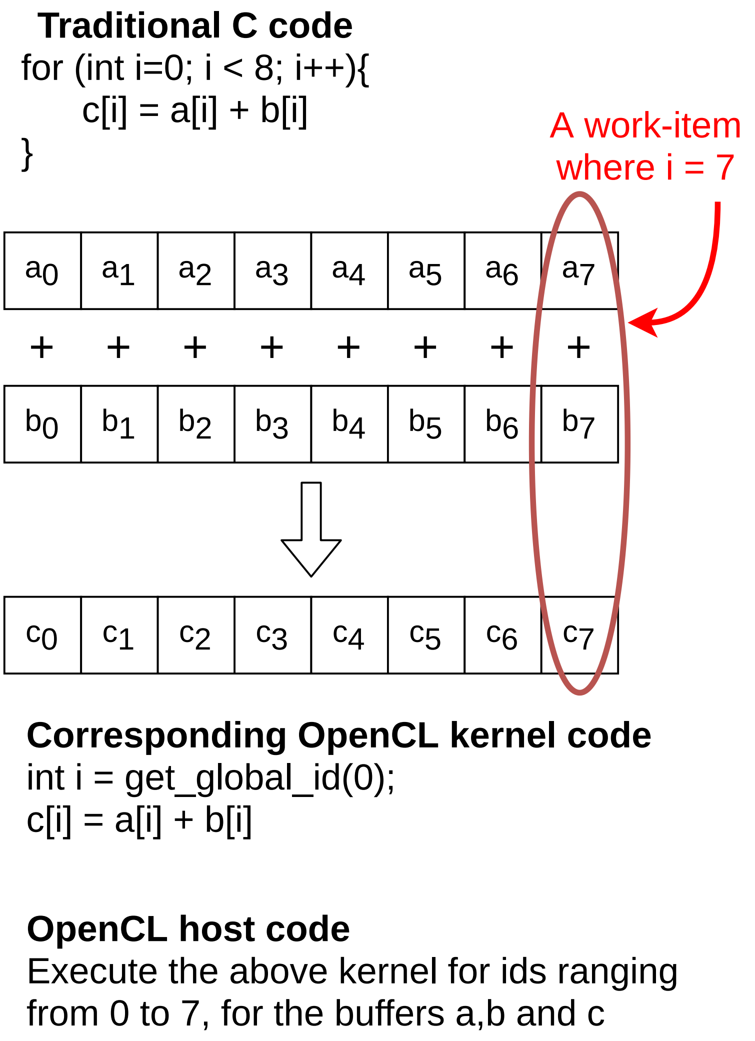
2.2.3 Open computing language

Open computing language (OpenCL) is a framework to program heterogeneous systems [10]. OpenCL allows the same program to run on many different kinds of systems, for example on CPUs, GPUs, DSPs and FPGAs. OpenCL programs consist of two parts, the host and the device code. The device code, also called the kernel, executes the heavy computation on the data and the host code manages the execution of these *compute kernels*. [12]

The OpenCL kernels are written in OpenCL C, which is an extension of C programming language to support the SPMD-style programming in OpenCL. Kernels written in OpenCL C can then be executed for many pieces of data. These different instances of the kernel operating on different data are called *work-items*. All work-items execute the same kernel code program but use different data for the computations. The work-items of the same kernel are independent of each other by default, any synchronization between them needs to be manually implemented using *barriers*. Barrier is a point in a program that all work-items must reach before any of them can continue forward.

Executing a single work-item per core one at a time is not very efficient. On the other hand, manually splitting the work-items for the usable computing units is not very portable. Therefore, it can be left for the OpenCL runtime to determine how these work-items map to the underlying hardware. They might execute all in parallel, or all sequentially, or something in between, depending what's optimal for given hardware. Work-items are grouped into *work-groups* which are then executed together on a single compute unit. There cannot be any dependencies between the work-groups, so the grouping is only possible if there are no synchronization needs between the work-items to be split. The splitting can be done manually by the programmer, or it can be left for the OpenCL runtime to determine the optimal split for the specific hardware. [12]

Figure 2.4 shows a vector addition done in OpenCL. The figure shows which parts of the program are put into the kernel code, and which parts are left for the host code. The kernel code describes the actual computation and the host code tells how often and where to perform it. The function *get_global_id* in the example is a special function that returns the id of that particular work-item. It helps the work-item to know which part of the data it should operate on. Every work-item has a different consecutive id depending on the range set for the kernel in the host code. This example has 8 work-items. The OpenCL runtime compiler could, for example, optimize this code into a single vector instruction of a CPU if it is available.


A work-item
where i = 7
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
+ + + + + + + +
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7
c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7

Traditional C code

```
for (int i=0; i < 8; i++){
     c[i] = a[i] + b[i]
}
```

Corresponding OpenCL kernel code

```
int i = get_global_id(0);
c[i] = a[i] + b[i]
```

OpenCL host code

Execute the above kernel for ids ranging from 0 to 7, for the buffers a,b and c

Figure 2.4. *Vector addition in OpenCL.*

OpenCL has separate memory regions for the host and the device. The OpenCL memory regions can be thought of as an abstraction on how the different parts of the program see the memory. An OpenCL implementation can then map them to optimal memory components. The device memory region is further split into four logically disjoint parts. First, *global memory* can be written and read by all the work-items. Second, *constant memory* is a read-only memory for all the work-items initialized by the host code. Third, *local memory* is only accessible to work-items of a same work-group. Fourth, *private memory* is exclusive to each work-item. [10]

The language naturally supports data-level parallelism, because every work-item computes the same function for different pieces of data. For example, a single work-item could map to an SIMD lane of a processor or a lightweight thread of a GPU. OpenCL also supports task-level parallelism since multiple work-groups can execute freely in parallel threads as there cannot be dependencies between them. Also, the separate kernel launch commands are not always dependent on each other and can therefore be executed in parallel. [14]

2.3 Parallel computing on field programmable gate arrays

FPGA is a device which can be used to implement any logical circuit and the circuit can be changed after manufacturing by *configuring* it. FPGA consists of mainly logic cells arranged as a matrix and the interconnections around them. Both of these can be configured to create the optimal circuit for specific application [15]. FPGA contains a lot of inherent parallelism in its hardware due to its matrix-like structure. However, utilizing it is still quite difficult, since it requires designing a logic circuit with appropriate parallelism.

FPGA tools take care of mapping the designed circuit to the FPGA device's resources. This is called synthesis. Placing the logic elements and routing the interconnections between them is a complicated process and can take hours to complete. Configuring an FPGA is functionally closer to designing an ASIC than programming a CPU since FPGAs can be used to implement any logic circuit. However, usually the clock frequency is much lower than would be on a corresponding ASIC. On the other hand, developing FPGA designs is much faster, since the design can be tested and be ready to use in just a few hours compared to ASICs, where the manufacturing process is very expensive and can take months. Therefore, FPGAs can be used to quickly develop application-specific fixed-function accelerators. Additionally, FPGAs are also used for prototyping ASIC designs. [16]

Generally, the common digital design principles from ASIC world apply also to the FPGAs. However, there are certain technical features of FPGAs that should be taken into account. For example, FPGAs have a lot of available flip-flop registers to break up critical paths [16]. Registers can be added in the middle of these paths to split the logic into two cycles. This can allow the clock frequency to be increased which directly increases the performance of the entire system. Therefore the critical paths should always be examined closely, and any low-hanging optimizations should be made. In ASIC-design adding extra registers to data path would also increase the resource usage, but in FPGA these registers are often there anyway just waiting to be configured. However, in any logic design adding registers in these paths does increase the latency of all the signals going through them. In some cases this can create problems if external components rely on the timing of that path. Also, having the state of the circuit extended over multiple clock cycles adds complexity in the control logic.

In addition to logic elements, the FPGAs usually include dedicated resources for certain commonly used pieces of circuit. Almost all designs use some amount of internal fast RAM, so there are dedicated RAM blocks inside the FPGA. Implementing hardware multiplier in FPGA logic can be quite expensive, so FPGAs often include hardened DSP-blocks to perform multiplication or floating point operations. [16]

While designing circuits for FPGA is much faster than on ASIC, it still requires a lot of knowledge of both digital design and the FPGA device. High-level synthesis tools aim to generate the circuit based only on the high level language description of the program. Even then, the mapping from software domain to a digital circuit is far from being a solved problem. Traditional HLS tools still require long synthesis times, since they have to perform the heavy placing and routing. In addition, typically the source code of the program has to still be adjusted for the HLS, so the same CPU-optimized code can't be used directly.

2.3.1 Soft processors on FPGA

Another approach to the HLS problem is to use soft processor overlays on the FPGA. A soft processor overlay is a premade processor system running on the FPGA and the high level program is compiled to it using the regular software compiling techniques. This is one way to utilize the FPGA resources without having to design or generate (with HLS) a logic circuit. The soft processor might be an ASIP, so it might need to have its own compiler for that special instruction set architecture.

Traditionally FPGA soft processors have only been used in controller-type of applications [16]. However, the flexibility they offer can be useful in even more data-heavy applications. The datapath of a soft processor can be customized to best fit to a certain application and to fully take advantage of the resources present on the FPGA device.

There are at least two distinct approaches for creating an efficient soft processor system for heavier computing. Parallel structure of the FPGA makes it possible to have processors with very wide data lanes. This approach is called *soft vector processor* (SVP). The second approach is to create a relatively simple scalar core, and duplicate it on the the FPGA tens or hundreds of times. Both of these methods can utilize *specialization* to create the most suitable configuration for certain application. Naturally the use of one approach doesn't exlude the other, the multicore design can still utilize wide SIMD inside the core. However, at some point the available FPGA resources start to constrain the size of the system.

Two of the more popular soft processors are the ones created by FPGA manufacturers themselves. Xilinx has Microblaze [17] and Intel Altera has Nios II [18]. These soft processors are cleanly integrated into their FPGA development tools so they are easily usable for the developer. Both of these can also be specialized to either be a light-weight and simple controller-type core or a more specialized core with wide SIMD operations.

MXP [19] is a scalable matrix processor, this means that it operates with wide SIMD vectors, with hardware supported loops, which make it able to natively operate with matrices of data. It doesn't include regular register files, but instead connects the function units directly to the scratchpad memories to enable a smooth dataflow through the FUs.

Another example is PipeArch [20], which is something between a programmable accelerator and a soft processor. It uses heavily specialized accelerator function units with software controlled threads and context switches. They utilize deep pipelines and wide SIMD provided by the FPGA device to accelerate various machine learning applications.

Octave [21] is a heavily pipelined customizable processor with fine-grained multithreading. It's well optimized in terms of clock frequency and can reach up to 550 MHz on a Stratix IV FPGA, which is impressive for a soft processor.

IPPro [22] is a very light-weight RISC-based core utilizing the hard DSP-block of an FPGA as an ALU. It can be used to create a multicore system with up to 120 cores running at 530 MHz on Zynq-7020 FPGA.

The toolset created by Cartwright et al. [23] can generate a multicore system out of Microblaze core based on an OpenCL application. The tools generate a fitting OpenCL memory system, and compile code for it using their own threading library.

OpenRCL [24] is a multicore system utilizing simple MIPS processors with fine-grained multithreading. The multicore can be programmed with OpenCL and has been prototyped as a system with 30 cores running on Virtex-5 FPGA.

MARC [25] is an asymmetric multicore system with a single control processor running the control program and multiple highly specialized DSP-cores running the compute kernels. The cores are based on RISC-architecture and can be programmed with the OpenCL framework.

Hoozemans et al. [26] present a way to create an efficient OpenCL streaming setup for image processing using multiple VLIW cores. The cores are connected to each other as a pipeline utilizing on-chip block RAMs.

2.3.2 Advanced extensible interface

A bus protocol defines a standard interface to connect different parts of a circuit together. This simplifies the system design, when only conforming interface is required to ensure that two components can communicate with each other. This makes it possible to develop parallel systems where each component is developed independently and can then be easily connected to each other. In a multicore system, this could mean that the memory interface for each of the cores is implemented as some standard bus interface.

Standard interfaces are also used to connect FPGA logic designs to external hardware circuits. These can be used to e.g. connect to an external DRAM memory. Examples of bus protocols include ARM's AXI [27] and open-source Wishbone [28]. On FPGA logic itself, any of these can be used. When connecting to the external resources from the FPGA device the correct bus protocol of the hardware port must be used. This might require adding an adapter, or changing all the interfaces to conform to the port's interface.

Advanced microcontroller bus architecture advanced extensible interface (AMBA AXI) is a protocol for on-chip communication developed by ARM [27]. It can be used create high-performance interconnections between parts of a circuit. AXI has completely separate read and write channels with their own address, data, and control signals to allow for full-duplex communication. It is a burst-based protocol, which means that it can do transactions where it only specifies a starting address and how many consecutive transfers it wants to do following that address. These sub-transactions which use the full data width of the data signal are called 'beats'. For example when wanting to load 128-bit value through a 32-bit wide interface, it can start the transaction by specifying the address for the first 32-bit value to the slave and signaling that it wants 4 total values from it. Then the transfer happens as 4 consecutive 'beats' without the master having to specify the address for the other 3 values. Because the AXI is a burst-based protocol, it's optimized for throughput instead of latency. This means that it's able to move lot of data through the interface per unit of time, but that the latency of any single transaction can be quite high, depending on the interconnect size and complexity.

In the simplest configuration, there is one master interface and one slave interface. The master interface initializes the transfer using control signals and the slave must be able to respond to according to the protocol. AXI isn't limited to single-master, single-slave configurations, since it supports both multiple slaves and multiple masters. However, these bring complexity, since in multi-master configurations the control of the interconnect must be arbitrated between them. In multi-slave configurations, each slave must be memory mapped to their own address space. Therefore in practice, only single-master single-slave-transactions can happen at a time in one interface.

AMBA AXI has a few different revisions which are compatible each other. The AXI3 protocol was released in 2003. The AXI4 was released in 2010 together with the AXI4-Lite and AXI4-Stream protocols for special use cases. The main difference moving from AXI3 to AXI4 was the increase in maximum burst size from 16 beats to 256 beats. The AXI4-Lite protocol is a subset of the AXI4 protocol with some signals removed but it is still able to interoperate with the complete AXI4 interface. Most importantly, it only supports bursts of length one (1 beat). AXI4-Stream is a low-overhead interface for streaming consecutive data from master to slave without using addresses. [27]

3 GENERATION OF MULTICORE SYSTEMS USING TCEMC-TEMPLATE

TCEMC is a toolset to generate MCASIPs. The design flow includes a way to duplicate a single *transport triggered architecture*-core (TTA) many times and connect it to a memory system. In addition, it includes a distributed threading library to easily distribute the computation between the cores. [29] Multicore systems generated using the TCEMC-toolset exploit the thread-level parallelism similarly as in the right-most diagram in Figure 2.2. So there is no hardware support for multithreading inside the core.

Figure 3.1 shows the memory connectivity of a typical TCEMC system. Each core has a small local memory next to it. The instruction memories are also located close to the cores. Accessing the shared memory is perhaps the most difficult part of the system. The cores have to share the physical signals which connect them to the shared memory with each other. The shared memory component can be implemented using either on-chip memory or an external memory component. Another shared resource is the shared mutex unit, which can be used to lock specific memory addresses exclusively for a single core to use. The system can be accessed from outside through an AXI slave interface, which allows for a host CPU to control the execution.

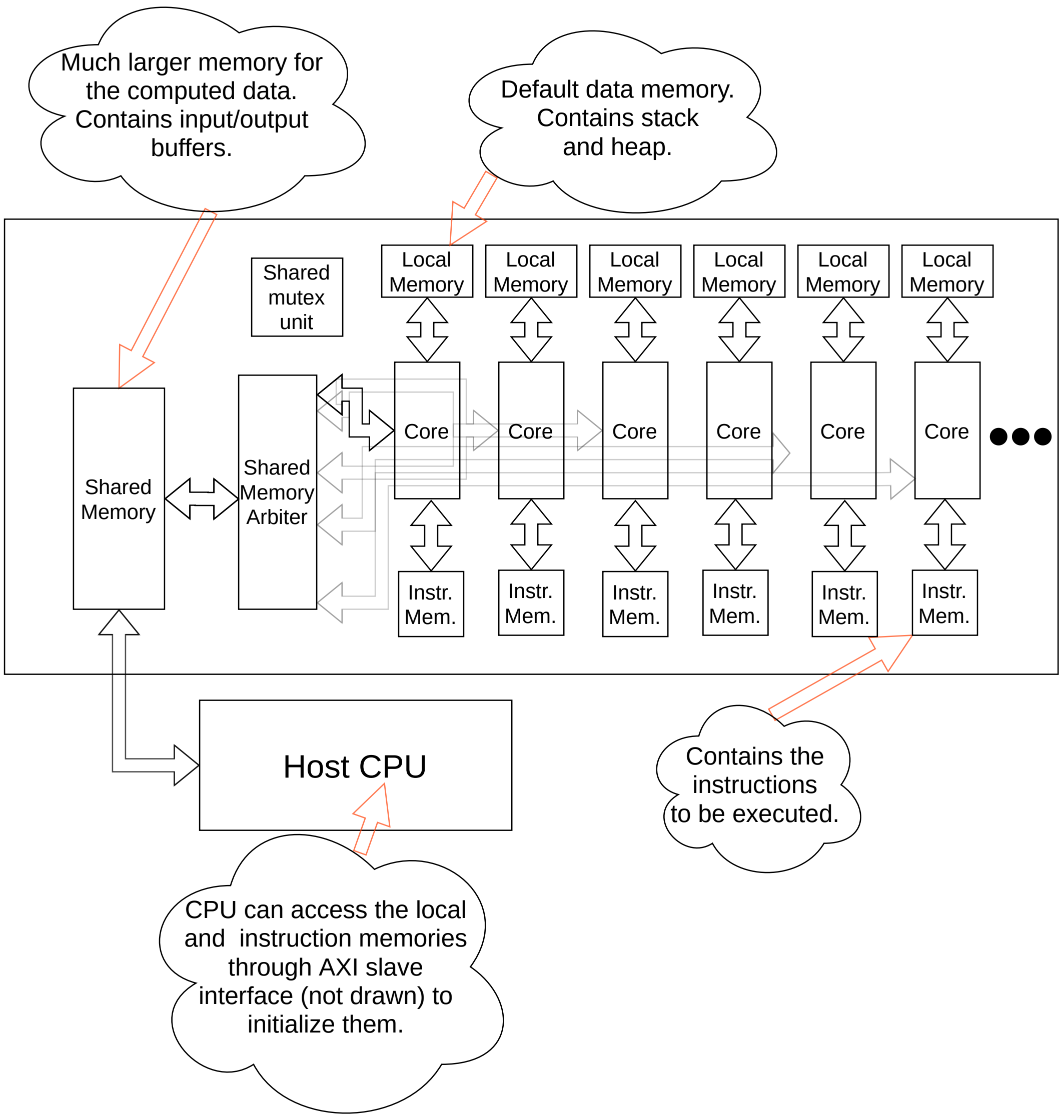

Much larger memory for the computed data. Contains input/output buffers.
Default data memory. Contains stack and heap.
Shared mutex unit
Local Memory
Local Memory
Local Memory
Local Memory
Local Memory
Local Memory
Shared Memory
Shared Memory Arbiter
Core
Core
Core
Core
Core
Core
●●●
Instr. Mem.
Instr. Mem.
Instr. Mem.
Instr. Mem.
Instr. Mem.
Instr. Mem.
Host CPU
Contains the instructions to be executed.
CPU can access the local and instruction memories through AXI slave interface (not drawn) to initialize them.

Figure 3.1. *The original TCEMC-template.*

3.1 Transport triggered architecture

Transport triggered architecture is a processor architecture, where the internal datapath of the processor is exposed to the programmer and the compiler. It's often implemented as a static multi-issue processor with parallel function units executing different tasks which must be utilized statically during compile-time [30]. Operations are defined as moves between the *function units* (FU) or *register files* (RF) of the processor. Operations trigger as a side-effect of this move-operation. Each internal bus of the processor can be used to perform a move once in a cycle. The connectivity of the processor doesn't have to be complete, even the register files don't have to be connected to every function unit.

A very typical comparison to TTA is the *very long instruction word*-processor (VLIW), which is another static multi-issue processor. One of the issues with VLIW is the complex register file connectivity, caused by the implicit RF access of the operations. The VLIW processor has to be designed to account for every possible RF access to happen on every operation field of the instruction on the same cycle. This means that the hardware must have many RF read- and write-ports. TTA does the accessing of RFs with explicit moves defined at compile-time. This helps to reduce the amount of RF-ports, as every operation doesn't need the full access to the RF [31].

The processor has an internal pipeline, but the structure of it differs quite a bit from e.g. a RISC-pipeline. After the instruction is decoded, the pipeline splits into each function unit. This lets every operation to have its own optimal pipeline length.

TTA-based Co-Design Environment (TCE) -toolset allows rapid design and customization of ASIP TTA-processors. It can be used to design TTA-processors with specialized resources to best fit the given application. The tools generate either VHDL or Verilog description of the processor, which can then be synthesized to FPGA or ASIC. The customization changes the instruction set, so the toolset has its own compiler in order to generate code from high-level languages (C, OpenCL) to all possible TTA-configurations. In addition to the FUs, also the register files and the interconnection network between the FUs and RFs can be customized to fit the flow of data in a given application. [30] Almost any kind of RTL-code can be fitted inside processor's function units, which further enlarges the TTA design space. Automated approaches to choose the correct FUs and to optimize the interconnection network of a TTA processor are being researched [32].

3.2 Memory connectivity

TCEMC-template supports multiple disjoint address spaces. One address space can be reserved for each core's local memory which is located physically close to the core to ensure a fast and private access to it. The another address space can either be on-chip or on an external shared component. In both of the cases the cores can access the shared memory through an arbiter.

The local memory of the core is the default data memory for the threads. This means that the stack and heap of the threads running on a particular core are located on the local memory component next to it. This improves the performance of the stack accesses, as they can happen simultaneously on the distributed components of the system and don't have to go through the shared memory arbiter. The shared memory can be accessed, but the access has to be explicitly defined in the program source code with an address space qualifier. This model complicates the programming somewhat, since now the stacks or heaps of other threads are no longer visible to all the cores. So, passing a simple pointer pointing to one of these structures to another core doesn't work. Therefore all the shared structures must be explicitly created in the shared memory component. [29]

Shared memory must be accessed through a shared arbiter. Shared memory is a single component and parallel access to it is limited by the number of ports it has. Generally memory components have multiple ports, so all of the ports should be used to minimize the needed arbitrating logic. The TCEMC-tools allow creating a full arbiter which arbitrates every memory interface of the cores together to get a single external memory interface which is then connected to the memory component. This is possibly inefficient depending on how many ports are available on the shared memory component.

The TCEMC-system describes by default a homogeneous multicore processor, where each core executes exactly the same instructions. Therefore, all the cores could share a single instruction memory. Having all the cores accessing the same component creates resource contention, which should be dealt with efficiently so that each core can still access the instruction memory in a single cycle. This can be achieved by each core having its own small cache or a loop buffer to store the most often used pieces of the program. Another approach is to include a read-only RTL array of the instructions for each of the cores. This is quite efficient, but makes it impossible to reprogram the processor. The simplest approach is to duplicate the instruction memory for every core. This is quite wasteful, since every memory component has the same content. However, it allows for better physical distribution of the cores on the chip as it reduces the number of signals connecting the cores together.

3.3 Shared mutex unit

By default, the data memories of the cores are not shared so they cannot be used for communication or data sharing. The shared memory is freely shared for every core to access. Using the shared memory for synchronization would require implementing atomic read-modify-write memory operations which could be used to create mutexes. However, using the shared memory bandwidth for lock queries could fill up the memory bandwidth unnecessarily. The shared memory bandwidth should be reserved for the moving of input/output data related to the main computation task itself. [33]

This is the reason why TCEMC-template includes a separate mutex unit to handle the inter-core synchronization. Each core is connected to the mutex unit by a few control signals that can be used to request and release a lock on a specific memory address (signals not shown in Figure 3.1). Each core must have a special function unit which can implement the locking operations. The instruction set includes the special instructions for this function unit. The function unit communicates with the shared mutex unit using a handshake. It can return a value to inform the core whether the lock operation succeeded or failed. It can also be used to stall the entire core until a lock is available. [33]

3.4 Executing OpenCL on TCEMC-system

TCEMC-template allows running OpenCL kernel code on the multicore processor. OpenCL host code is running on an external CPU and manages the kernel execution. The OpenCL support is built on top of *Portable Computing Language* (PoCL) [14].

3.4.1 Portable Computing Language

PoCL is an open source implementation of the OpenCL standard [14]. It can support multiple different devices including CPUs, GPUs, TTAs, and other experimental architectures. PoCL works together with the TCE toolset, and can be used with both the instruction-set simulator and the actual synthesized cores on an FPGA.

PoCL can generate regular functions from a range of OpenCL work-items. Work-items are split into optimal number of groups, and then loops are generated to execute this group of work-items, taking into account any possible synchronization barriers. These loops are then wrapped in a function, so it can be inserted to a regular program to make it possible to use traditional compilation pipeline. The compiler can then start with the assumption of parallel loop iterations, so it can generate efficient unrolled and interleaved code from the loops. The kernels are only compiled once the work-item range is known, which allows for compile-time static loop boundaries to further improve the performance. [14]

On the multicore system itself, there are two separate software components, OpenCL kernel code and a wrapper for it, which are both compiled to a single program binary. PoCL tools compile the kernel code and the wrapper automatically according to the just-in-time compilation model of OpenCL. The kernel code is given by the user and the wrapper is provided by PoCL. The wrapper has two responsibilities, split the given workgroups to the cores and communicate with the driver code to manage the execution.

PoCL takes care of allocating memory for the TTA. It fills the buffers with the input data, and then launches the TTA cores to compute the kernel. The shared memory of TCEMC-template maps to the global OpenCL memory. The global input and output buffers for the kernels are located there. The local memories of the cores map to OpenCL local and private memories. The PoCL host code driver doesn't need to access them during runtime, but it can initialize them.

3.4.2 Dthread

Dthread is a distributed threading library specifically designed to work with systems where the cores can't access each other's default data memories. It works similarly to more traditional threading library POSIX Threads [34]. The main difference between them is that Dthread doesn't assume shared address space between the threads. Dthread is written in C, and compiled from sources for each application. This is mainly because the TCE-toolset generates ASIPs, so the instruction set changes as the architecture is specialized. Also, any functions not used in a certain application are not included in the final program binary, which saves instruction memory bytes. [29]

A thread table is a book-keeping structure for the threads. When a thread is created, an entry containing the thread's arguments is created in the thread table. In Dthread, this thread table is split into two parts which are both implemented as linked lists: *shared thread table* (STT) and *local thread table* (LTT). When the threads are created, they are put into the STT, which resides in the shared memory that all the cores can access. Then the cores can independently fetch threads from STT and copy them to their LTT. The LTT contains the stacks of all threads running at that time on a certain core. When the thread finishes, the STT is notified that the thread is done. [29]

The STT is initialized right after reset by a single core. This includes creating the main thread of the application. All the cores, including the aforementioned one, initialize their LTTs after reset by creating an idle thread. The core can switch to its idle thread at any point if it has no other threads to execute.

Work-stealing is implemented with *ready queues* located in the shared memory. Every core has their own *ready queue* (RQ), which is a queue of threads waiting to start executing. A thread is (softly) allocated into one of the RQs at the creation-time. If some core finishes all the threads they have in their RQ, their idle thread can go look at other thread's RQs. If they find an unstarted thread, they can grab it for themselves and start executing it. The benefit of the RQs is to distribute the locking pressure in shared memory. The newly created threads will get split across the ready queues, and if the load is naturally quite balanced, then most of the time the core will only execute threads from its own RQ. This means that each core will most often be only contending for its own RQ lock, which they should therefore be able to get immediately. [29]

Figure 3.2 shows a simple case with only two cores. Each core has some threads obtained in their LTTs, so those are not available for any other core. On the other hand, thread R has been created lately and hasn't been obtained by anyone. During its creation, it was initially allocated for core 1. This is visible in RQ1, which has the thread R initially set for that core. However, if core 0 manages to finish both threads X and Z before core 1 goes to look for more work, it is able to steal the thread R from RQ1.

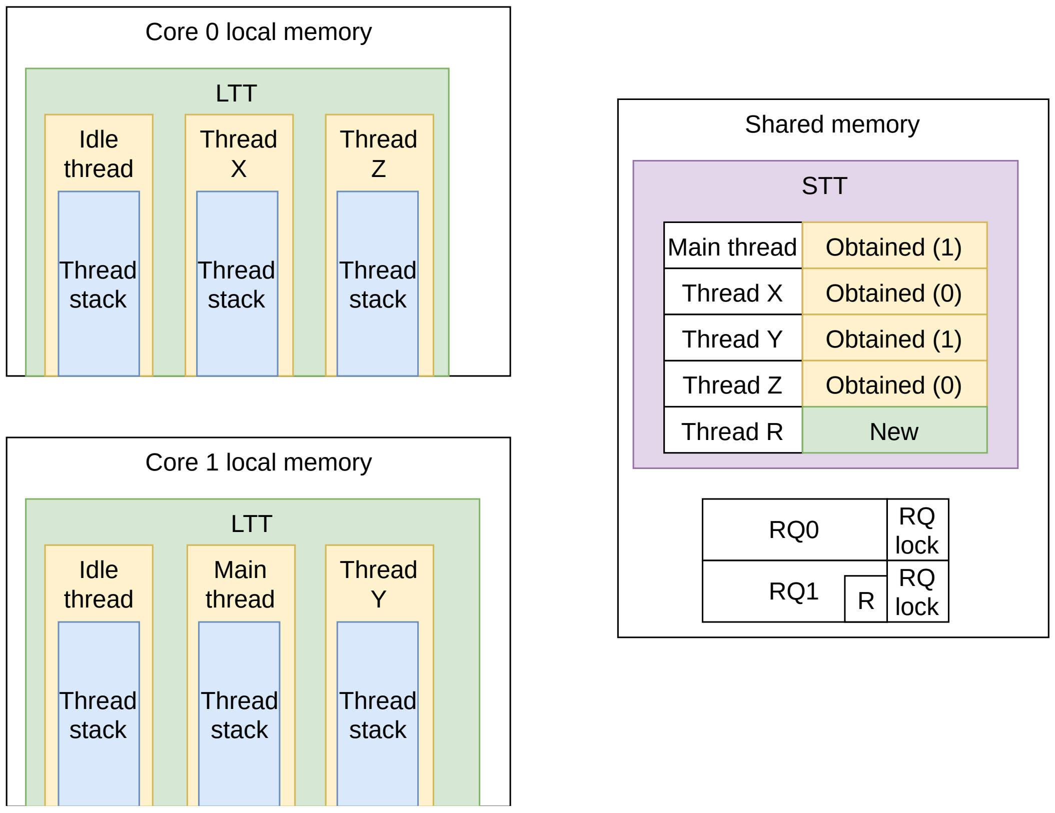


Figure 3.2. *Dthread data structures.*

When executing OpenCL code with PoCL, the Dthread code executes on the device. In the main thread, the work-groups are statically split to the cores and new threads are created to the STT. The cores can then fetch and claim threads for themselves, execute them, and repeat this until there are no threads left. The threads execute the work-groups by calling the work-group function described in Section 3.4.1. The main thread then waits for each of the threads to join back to the main thread. Then it can inform the driver that the kernel was successfully executed.

Dthread gives a lot of flexibility to create even complex kernel structures. It would also be possible to implement dynamic load-balancing and work-stealing using Dthread. The dynamic load balancing would be very useful if the work-group execution times vary significantly depending on the data. This way, some cores might be able to execute their work-groups faster than others. This becomes more important the less work-groups there are, since with enough work-groups it can be assumed that this discrepancy would even out. Another case for dynamic load balancing would be the concurrent execution of entirely different kernels. Different kernels are even more likely to have very different execution time, so static work allocation becomes even harder. On the other hand, for the simpler OpenCL kernels having a full threading library might be a bit too complex. The context switches, shared memory accesses and the locking using mutexes are all quite heavy operations that should be minimized as much as possible.

4 IMPLEMENTED OPTIMIZATIONS TO THE TEMPLATE

During this work, a few optimizations and improvements to the original TCEMC-template were implemented. Figure 4.1 shows the final connectivity of the system after the optimizations. The template is also slightly specialized to be more suitable for the used device (ZYNQ-7020 SoC). Instruction memories are now shared between two cores, and there are four separate AXI interfaces for the shared memory. In this setup, the onchip and *dynamic random-access memory* (DRAM) memory are mapped to separate address spaces with their own LSUs. Alternatively, the TCEMC-template could also have them in the same address space, but that feature was not explored during this research.

The local memories are implemented as on-chip memories and the shared memory is configured to be on the DRAM memory of the device. This is to make it possible to estimate the *external memory bandwidth utilization* with large amounts of data. Putting the entire shared memory to the external chip might not be the best idea in the long term because the commonly accessed shared data structures, such as OpenCL command queues, should be kept as close to the cores as possible. However, the OpenCL data buffers themselves might be too large to directly fit on the limited-size on-chip memory.

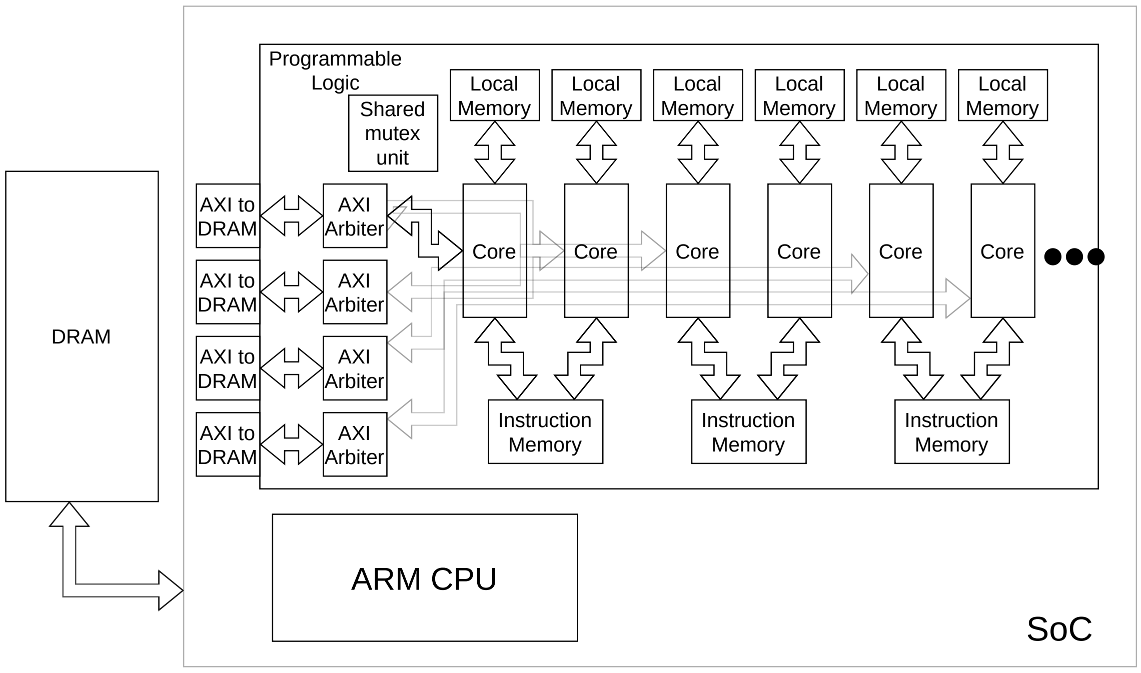


Figure 4.1. *System configuration after the optimizations and specialization.*

4.1 Optimizing the shared mutex unit

The shared mutex unit is connected to each of the cores. This makes it a possible bottleneck that should be investigated when increasing the number of cores. The unit must be designed carefully to minimize the combinatorial signal paths between the cores.

During this study one such combinatorial path was found. It was noticed that the shared mutex unit had unregistered signals from its input coming from one core going directly to its output to a different core. This was solved simply by registering all the incoming signals to the shared mutex unit. In Figure 4.2 the clouds represent combinatorial logic and the path formed by the red arrows is the original critical path. FF-blocks represent flip-flop registers. Since the lock function unit inside the core doesn't rely on the shared unit's timing, no other changes are necessary.

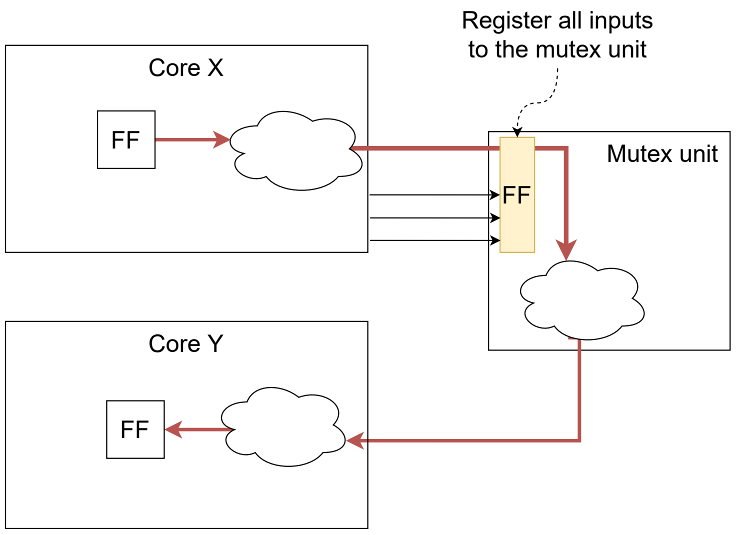


Figure 4.2. *Registering the inputs to the mutex unit.*

4.2 Accessing the local memories from outside

On the FPGA the local memories are implemented using the block RAM resources of the device. To initialize the global and static variables of the program, the local memories should still be accessible from outside. The local memories could be set to certain values during the synthesis. However, that would limit the programmability of the system, when the initial values for the local memories couldn't be changed after synthesis.

Another approach would be to handle the initalization of the local memories in software. A simple memory initialization routine could be added to the final program by the compiler. When the core is then launched, it would start by initializing the local memory by itself. Unfortunately, this would increase the instruction memory usage since the initialization values would be packaged together with the program.

Instead, the local memory initialization was left for the external ARM CPU to handle. There was a ready-made functionality in TCEMC to access the local memory through AXI slave interface, but it only worked for single core. During this research, that was extended to a simple hardware broadcast operation where the ARM CPU can write the same value to each of the local memories at the same time. This has the advantage of being very programmable (host CPU controls it), while also not increasing the hardware complexity too much (adds a few wires). Having more fine-grained access to each of the local cores could be useful in the future, but in this setup it was found to be unnecessary, since the local data memory is completely private for each core.

4.3 Sharing the instruction memories between two cores

In this case, the system is executing SPMD-programs, which are implemented so that each core executes exactly the same instructions. This regularity can be exploited by sharing the instruction memories between the cores. The accesses to the instruction memory are still independent between the cores as they usually execute a different part of the program at any given time. This would require arbitrating if the memory component was shared. In the Xilinx's ZYNQ-7020 Soc, the block RAM resource has a dual-port operating mode, which allows two independent accesses to happen at the same time without any missed cycles. This allows the RAM blocks to be shared between two cores. This configuration is shown in Figure 4.3. This fix saves the block RAM resources of the FPGA and therefore allows more cores to be fitted on the device.

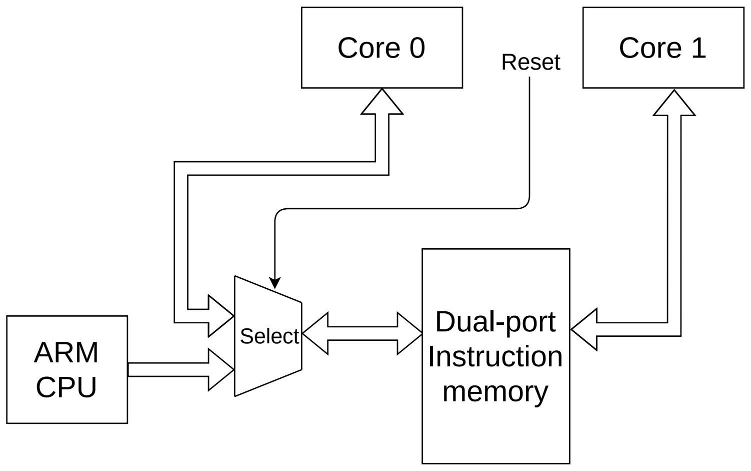


Figure 4.3. *Dual-port instruction memory connectivity*

However, there is also a third component which has to access the instruction memory. The ARM CPU has to initialize the memories to the desired program. Previously, this was done using the second port of the block RAM component. So now that the both ports are used during runtime, this access has to be redesigned. The instruction memory initialization happens before the cores are started, so it doesn't need to rewrite or read the instruction memories during the execution. This constraint allows an arbiter to be controlled by the reset signal of the processors. This is important, because now there are no missed cycles during runtime for either of the cores connected to the memory. When the cores are under reset, the ARM CPU can write to the instruction memory. When the reset is deasserted, both of the cores are connected to the two ports of the instruction memory and the ARM CPU's connection is disconnected.

4.4 Arbitrating the shared memory access

In this setup, an external DRAM chip is used as the shared memory for the multicore system. A physically continuous area of the DRAM memory is allocated by the PoCL driver to be the shared memory of the system. The ARM CPU can access this memory as a normal non-cacheable part of its own memory. The multicore system must go through the FPGA's memory port interfaces to access it. The address for the reserved DRAM memory area is given to the multicore system as a configuration value while it's under reset.

The shared memory access is one of the critical points of this system. Every core needs to access the same shared memory. To minimize the needed arbitrating logic, all the ports of the memory component should be utilized. On the ZYNQ-7020 FPGA [35], there are 4 *High Performance* (HP) AXI3 ports going directly to the DRAM controller and bypassing the central memory interconnect, which is shared between all the components of the SoC. To take the maximum advantage of this hard-coded parallelism, all four of the HP-ports are used equally. Four AXI arbiters are needed to connect every cores' AXI interface to the device's AXI ports. The example machine used for this research is a 32b scalar machine, so the maximum access width used is 32 bits. The AXI interfaces of the cores are implemented as AXI4 lite ports. The device's High Performance-ports are 64-bit wide AXI3 ports so protocol and data width conversion from the 32-bit signals is needed. Xilinx has provided premade IPs to do this conversion. However, using them and getting the maximum performance still requires some work. Figure 4.4 shows a few of the available configurations.

ZYNQ-7020 SoC allows the HP-ports to be configured to either 32-bit or 64-bit mode, so the data width in bits is configurable. However in this work, the 32-bit mode couldn't be made to work correctly, so the 64-bit mode had to be used. It is possible that the HP-ports need to be configured during the device boot sequence. This is something that needs to be tested and investigated further. Using the 32-bit mode directly would make the most sense, since the cores use 32-bit data width. In Figure 4.4 this corresponds to the configuration "True 32b".

Using the 64-bit mode of the HP-ports requires data width conversion from 32 bits to 64 bits. Xilinx's AXI Interconnect and SmartConnect IPs can be used to perform this conver-

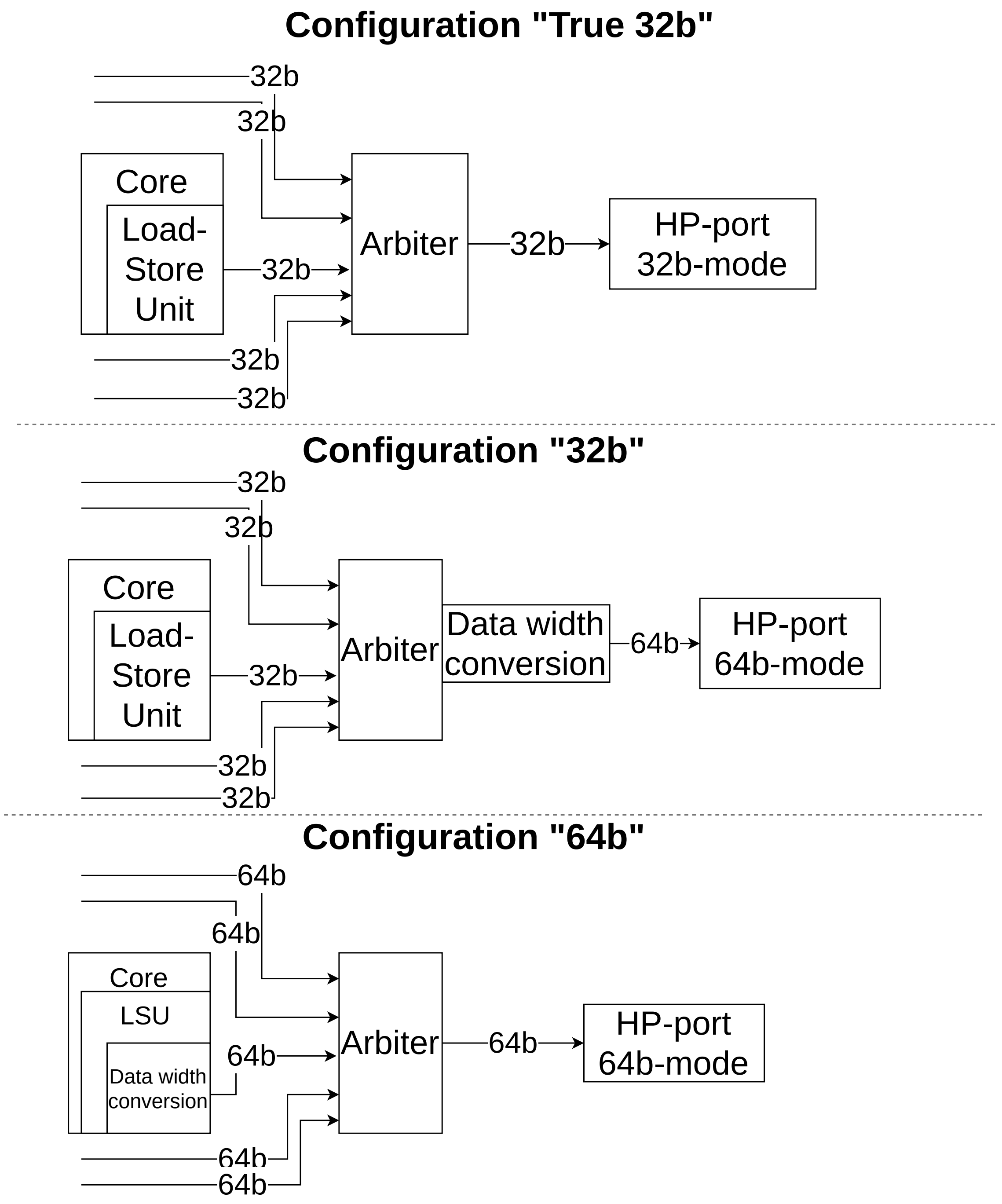


Figure 4.4. *A few of the different shared memory arbiter configurations*

sion automatically. However, the SmartConnect IP was quickly found to not be usable in this case. It consumes massive amounts of area because of its poor implementation of the data width conversion. The IP is required to do both arbitration and data width conversion. It is much more efficient to do the arbitration with 32 bit-wide signals and do the data width conversion only once to the output of the arbiter. However the SmartConnect first does the data width conversion to 64 bits and then arbitrates the 64-bit wide signals. This is not efficient in terms of area utilization, since it does the conversion many times more than necessary. Configuring the SmartConnect to do the data width conversion only once did not work. This huge area consumption ruled out the usage of SmartConnect IPs.

It's also possible to use an older Xilinx IP called AXI interconnect, since it can perform similar functions as the SmartConnect. It is used similarly to perform the arbitration and the data width conversion automatically. This arbiter uses significantly less area than the SmartConnect. This suggests that the AXI Interconnect IP is correctly arbitrating the signals as 32 bits wide, and only doing the data width conversion once. However, there are problems with the performance of this IP that are evident in Figure 5.2 and will be discussed in more detail later. This configuration will be called "32b".

Another approach to the arbiting and the data width conversion is to first manually do the data width conversion at each core's load-store -function unit. This is somewhat equivalent to the way SmartConnect IP is doing it, but writing it manually produces much better area utilization results. This way the Xilinx's AXI Interconnect IP is only used to handle the arbiting with the 64-bit wide signals. This configuration is called "64b" in Figure 4.4. In the end, this configuration provided the best performance.

4.5 Simplifying the kernel launcher

The kernel launcher is the part of the OpenCL system which handles the execution of the work-groups. It's running on the OpenCL device, which in this case is the TCEMC-multicore system. Using Dthread to create new threads for each of the cores comes with slight overheads, since there is synchronization and shared memory accesses needed to create the threads. In addition, the Dthread functions must be compiled and included in the final program binary, which increases the instruction memory usage. Additionally, many OpenCL programs are so simple (single kernel executed at a time) that there is no need to use a full threading library to launch the kernels.

Therefore, a simpler kernel launching wrapper was created. This wrapper has only one locking operation at the start of the program to generate a unique ID for each of the cores. Then the core with the ID 0 becomes the manager for the other cores. All the cores are waiting for the host to issue a command to start the execution, after which they independently start executing the work-groups. The work-groups are (statically) split based on the core's ID and the total amount of work-groups. There is no communication needed between the cores to split the work-groups because the cores can all figure out based on the work-group dimensions and the total core count which work-groups are their responsibility to execute. The work-groups are split equally to all the cores, including the core ID 0. If the number of work-groups is not divisible by the core count, the leftover work-groups are spread out evenly to only some cores. After cores finish executing their work-groups, they set a flag that the core ID 0 polls for. Once the core ID 0 notices that all the other cores are finished, it can communicate the finished execution to the driver.

5 EVALUATION

To be able to evaluate the effectiveness of the optimizations and the multicore scalability of the TCEMC-template, it has to be specialized to include the internal architecture of a single core. The chosen single core architecture is kept as simple as possible to focus the attention to the multicore-specific problems. Fitting as many cores as possible to the device exposes the parts of a template that don't scale up as well as the others. In fact, this research technically doesn't even require using TTA-processor as the internal core, as long as the memory intefaces of the LSU would be similar, any kind of soft processor could in theory be used in the place of it.

The system is benchmarked on ZYNQ-7020 SoC, which has both an FPGA and a dual-core ARM CPU on the same chip [35]. It has a large DRAM memory on an external chip, which is used for the shared memory. The performance of the multicore system is evaluated by four simple OpenCL benchmarks. The external memory bandwidth utilization is measured to estimate the efficiency of latency hiding.

5.1 Architecture of a single core

The architecture of a single core is designed with the TCE toolset. TCE toolset allows the function units and the operations and connections between them to be edited using a graphical interface. The designed architecture is shown in Figure 5.1. Since the architecture is kept as simple as possible, only the absolutely required function units are present. Starting from the left in Figure 5.1 the blue ALU block is the unit that computes all the arithmetic operations. It uses a very minimal subset of arithmetic operations. The hardware multiplication operation is included because it can utilize the available DSP blocks of the device. The generic shifting functionality is removed and replaced with only a one-bit shift. Therefore the compiler must emulate all other shifts by repeating the shift-by-one as many times as necessary. The green LOCAL and DRAM blocks are the load-store-units that access the local and DRAM memories. The green LOCK block is the function unit implementing the lock/mutex-operations. It is connected to the shared mutex unit. The yellow BOOL and RF blocks are 1-bit and 32-bit wide generic register files. The 1-bit register file is used as a guard variable to allow predicated execution of instructions. The red IU block can be used to output 32-bit long immediate values to any other function unit's input. The violet GCU block is the global control unit which is used to implement function calls and jumps by manipulating the program counter. More comprehensive details of the instruction set are listed in Appendix A.

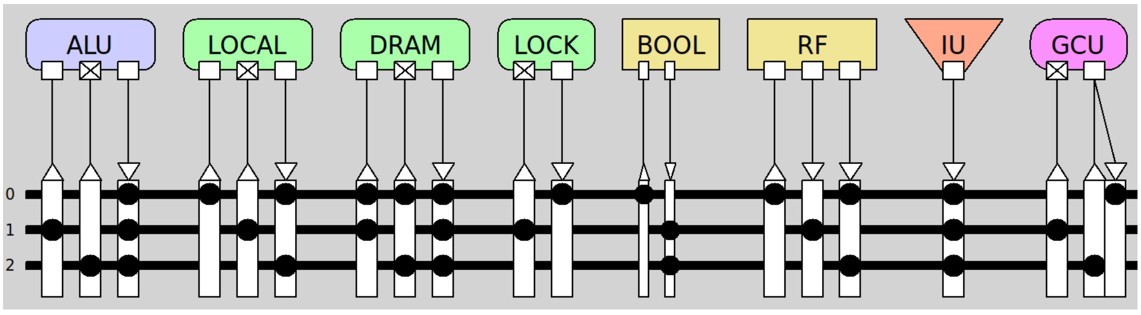


Figure 5.1. *Architecture of a single core*

The connectivity between the function units is designed manually to be a balance between the interconnect complexity and smooth movement of data between the function units. Full connectivity of the buses would allow the most free movement of data, but would be really costly in terms of interconnect area utilization and instruction width. On the other hand using only a single bus would allow for even more minimal hardware but would hinder the performance too much, since the operands would have to be moved to the function unit inputs sequentially. The core is designed to sustainably execute 1-2 operations in parallel.

5.2 Benchmarks

Simple OpenCL kernels are written to benchmark the performance of the multicore systems. Benchmarks are chosen to be very simple to focus on the communication with the DRAM memory. The kernels have just a little bit of computation to add small latencies between the memory accesses. The benchmarks are: vector addition, matrix multiplication, polynomial and copying. The amount of data in each benchmark is chosen to be as large as possible, but still staying within the practical limitations of the system (~10 MiBs).

The size of the local memories has been minimized as much as possible to the point where the system still functions with the tested benchmarks. 2 KiB was found to be enough for the local memories. The instruction memory size was similarly minimized to be 16 KiB per core.

5.3 Comparison of AXI arbiters

Since the shared memory arbiting is such a critical point of the system, a few different arbitrating methods are evaluated. In Section 4.4 a few of the possible arbiter configurations are shown in Figure 4.4. The two working configurations "32b" and "64b" are now

Table 5.1. *A comparison of FPGA LUT utilization between the 16-core machines with 32b DRAM AXI interface and the 64b DRAM AXI interface*

Utilization (LUT)	32b	64b	Increase
Total	29516	34406	1.2x
AXI interconnect	2892	6049	2.1x
Load-Store-units	3360	5024	1.5x

compared against each other.

The difference in area utilization between these last two configurations is shown in Table 5.1. The difference in logic utilization means that the 32b configuration allows up to 28 cores to be fitted on the FPGA whereas the 64b configuration only allows 24 cores. This is caused by the location of the data width conversion block. If the conversion is done before the arbiter, it has to be done for every core separately and the arbitrating is done with the wider signals.

Figure 5.2 shows how the performances of these configurations scale when increasing the number of cores. It is clear that the 32b configuration already plateaus at the core count 4. This points to the AXI Interconnect not working as intended. Since there are four (4) AXI HP-ports on the device connecting the multicore system to DRAM, it can be deduced that each AXI Interconnect only allows one memory access to happen at a time. This causes the performance of the entire system to be saturated at 4 cores.

Why would changing the width of the AXI interface cause the AXI Interconnect to perform so poorly? This could be due to a poor implementation of the data width conversion element needed in the AXI Interconnect. When loading the data in the processor, the data width conversion element has to remember the ongoing load operation's address to be able to select the correct 32 bits out of the 64 bit data coming from the DRAM. If the AXI Interconnect allows only one load-operation to happen at a time, it only has to remember one value. This limitation could be overcome by remembering the addresses in a FIFO, which would then be used to control the selection of the loaded data. In fact, not even the entire address would have to be remembered. One bit would be enough to remember whether the most or least significant 32 bits would need to picked out from the 64 bits of data.

Because of the performance saturation of the 32b-configuration, the 64b-configuration has been selected for the rest of this work. Further tests are required to optimize this critical piece of the system. Getting the High Performance ports' 32-bit mode working would allow the "True 32b"-configuration to be used, which would probably yield the best performance. Another approach would be to leave out the Xilinx's premade interconnection IPs and rather implement a simple arbiter manually. That would allow the IP to be the most optimized for this particular use case

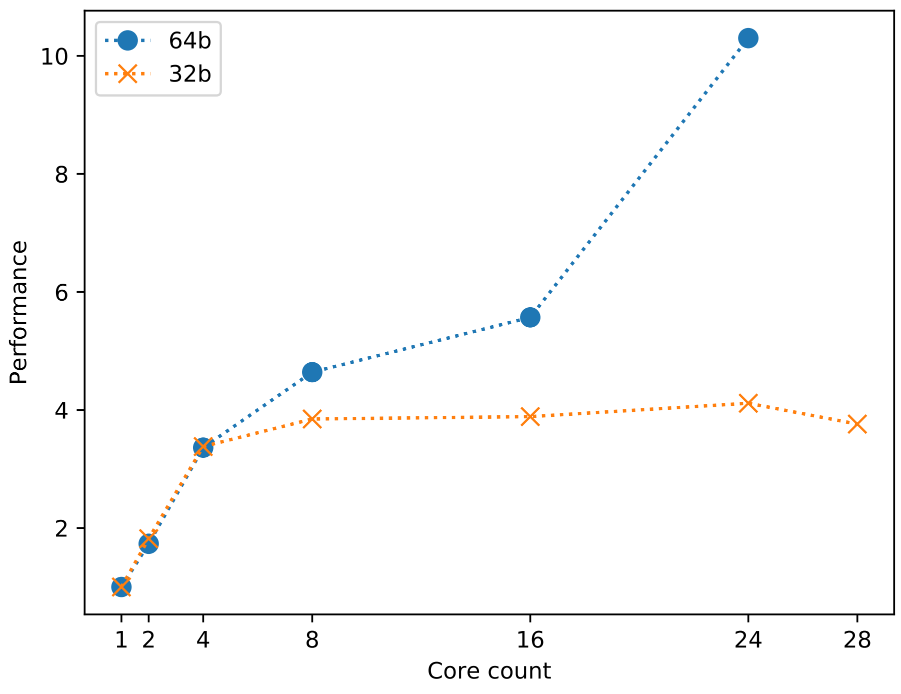


Figure 5.2. *Performance comparison between the 32b arbiter and the 64b arbiter (Normalized to the single core performance)*

5.4 Comparison of kernel launching methods

The minimal kernel launching runtime presented in Section 4.5 can be compared to the Dthread runtime. With the large amounts of data, and using the same workgroup splitting algorithm there is very little difference in performance. However, the simpler approach has a significant advantage in terms of memory usage. The Dthread must have enough space in the local memory to support the stacks of all the local threads. Another place where the overhead of Dthread shows is in the instruction memory usage. Dthread includes quite a lot of functions that are compiled to the instruction memory, even though the flexibility they offer isn't fully utilized in a simple kernel launching task.

The minimal kernel launching method allows memory sizes to be decreased significantly, as shown in Table 5.2. In the table local memory usage is rounded up to nearest power-of-two, since estimating the precise usage of it is difficult since it varies depending on the stack usage of the program. Absolute values of all these numbers are not very important, since they are quite dependent on the used benchmarks. However, minimizing the local and instruction memory sizes is very important, since in the total memory usage these values are multiplied by the core count. So being able to decrease the memory sizes by a factor of four (4) is very important when the number of cores increased.

Table 5.2. *A comparison of maximum memory usage between Dthread kernel launcher and the implemented minimal launcher.*

Memory size (B)	Minimal	Dthread	Increase
Local memory	2048	8192	4x
Instruction memory	12632	52400	4.1x

5.5 Area utilization of the system

Synthesizing the RTL to the bitstream format recognized by the FPGA is done by using Xilinx's Vivado tool [36]. The FPGA used is the Zynq-7020 SoC, which has 53200 LUTs [35]. The LUTs are the limiting physical resource of the board. It could've been possible for the block RAM blocks to become the bottleneck, but the memory sizes were chosen to be small enough to not become a problem (16KiB instruction memory and 2KiB local memory). Additionally the implemented dual-port instruction memories as described in Chapter 3 helps with this by halving the block RAMs needed by the instruction memories.

LUT element utilization of the FPGA increases when the number of cores increases. There are resources other than LUTs, but the LUTs are used for implementing most of the logic, so that's why they are often used for approximate area estimates. The LUT scaling is shown in Figure 5.3. From the figure it can be seen that the utilization increases very linearly with the core count. The bend at the core count 24 is explained by the change of synthesis settings. The synthesis settings had to be changed from performance-oriented optimization to more general optimizations because the 24-core system wouldn't otherwise fit on the FPGA. Performance optimizations trade area for clock frequency, so it is expected that the clock frequency of 24-core system would be further decreased due to this change.

From the logic utilization breakdown in Figure 5.4 it's clear that the AXI arbiting takes a significant portion of the total area. Inside the single core the logic utilization depends heavily on the implemented machine architecture. In this core the ALU is quite simple as there is no generic shifter. Also, there are no application-specific function units, so it can look like the overheads of the TTA architecture (instruction fetch & decode and interconnect) are more significant than they would perhaps be in a more specialized core. Now the only "number-crunching"-function unit is a very basic ALU.

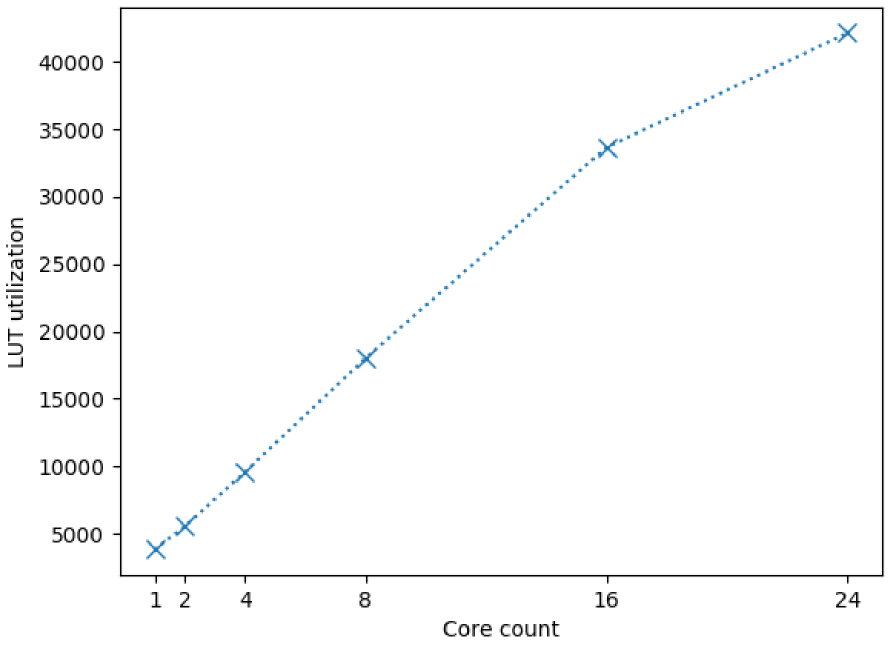


Figure 5.3. *Logic utilization when increasing the number of cores.*

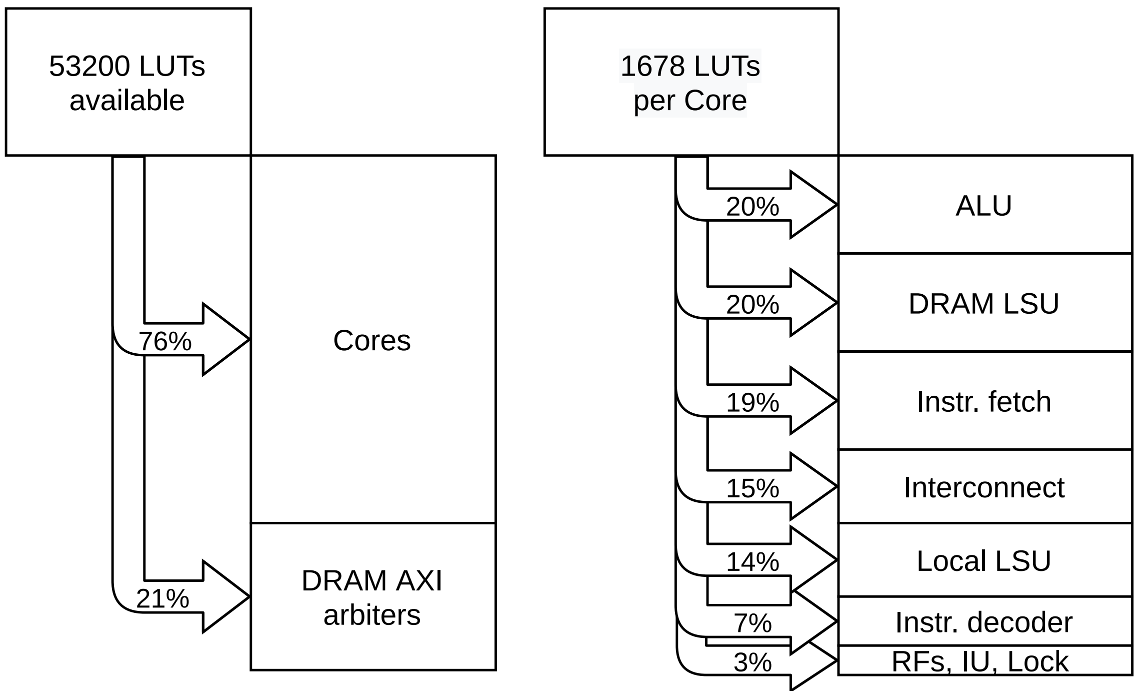


Figure 5.4. *Logic utilization of the 64b configuration (24 core-system).*

5.6 Clock frequencies

The maximum clock frequency for each core count is listed in Table 5.3. It's natural to expect that the maximum clock frequency should decrease when the number of cores is increased. This is because as the design takes up more space on the device, the placing and routing becomes more constrained. Another reason is the unregistered logic paths between the cores. These can be minimized by designing the system more carefully and adding registers wherever needed. Removing all the complex unregistered signal paths between the cores should be possible, because the cores should be able to function independently, excluding the occasional locking operations. Naturally, adding registers to split critical paths increases the area usage and complicates the control logic relying on signal timings. One more reason to explain the non-ideal scaling of the clock frequencies are the components which become more complex when the number of cores is increased. This includes the shared mutex unit and the AXI arbiters, which have to serve more cores, which complicates their internal logic.

Table 5.3. *Maximum clock frequencies of the synthesized machines.*

Core count	1	2	4	8	16	24
Maximum clock frequency	194.5	160.5	169.8	127.9	115.3	99.9

The critical paths of the routed design were analyzed using Vivado's own tools. Ten of the most critical paths of the 24-core system are related to the shared mutex unit. The shared mutex unit has quite a complex address comparison logic, where it checks whether the given address is locked or not. It also has unregistered connections to every cores' lock handling function unit, which causes a lot of routing delay in the signals. These issues should be solvable since adding latency to the lock operations should be possible. Therefore, adding registers to these logic paths would help with the clock frequency of the entire system. The shared mutex unit performance during runtime is not that critical to the performance of the entire system since the locking operations typically aren't used that much. The utilization of the lock unit depends on how many locking operations are needed in the OpenCL kernels. For example, having a system where every locking operation takes 30 cycles isn't a problem, if that allows the mutex unit to be moved off the critical path. A closer look at the shared mutex unit's RTL code is necessary to fix this problem.

5.7 Simulation results

The machines are simulated on the TCE toolset's instruction set simulator ttasim. The simulator doesn't take into account the stall cycles, or the resource contention between the cores. The simulation results' main purpose is to confirm the scalability of the benchmarks and to ensure that the work is split evenly to the cores with no overhead. In Table 5.4 are the geometric means of the benchmarks. The values in brackets are the performance increase per column normalized to the first value of the each column. This shows how the performance scales when increasing the number of cores. From the simulation results shown in the table, it can be clearly seen that the amount of instruction cycles scales linearly when increasing the core count. This is the best possible result that could be expected from the simulation.

The clock frequencies for each of the processors are shown in Table 5.3. The ttasim simulator does not take into account the different clock frequencies. Therefore the simulator performance is based only on the instruction cycle counts. The results received from the simulator can be adjusted by taking into account the real clock frequencies shown in Table 5.3. This gives us a slightly more realistic performance as seen in the column "Simulation scaled" in Table 5.4 but it still doesn't take into account the stall cycles.

5.8 Runtime results

The benchmarks were executed on the multicore system running on the Zynq-7020 FPGA and the results are presented in Table 5.4. The performance scaling in real execution is clearly behind the simulated performance scaling. There are two main factors contributing to this decrease: clock frequency and stall cycles. Relative effect of these two factors can be observed by looking at the column 'Simulation scaled', which takes into account the clock frequency but not the stall cycles. The latency hiding effect of a multicore system should in theory increase the relative performance when the core count increases. However, there's the opposite effect of resource contention caused by having more cores connected to each arbiter causing more dynamic stalls to each core. It is hard to separate the effect of dynamic stalls and the latency hiding. If the memory bandwidth fills up completely at any point, then the number of stall cycles should increase, saturating the performance. To estimate the congestion at the AXI arbiters, the memory interconnect should be profiled to see if it fills up.

Unsurprisingly, stall cycles seem to have a very significant effect on the performance. Cores on the FPGA can be accessed through a debugger interface to fetch the stall cycle counts and executed cycle counts. These values are presented in Table 5.5. In addition to the benchmarks used above, here also a simple copying benchmark is tested. The copying benchmark only copies the data from one buffer to another, so it doesn't have any computation, only communication. As expected, the stall cycle ratio varies between the benchmarks depending on the amount of computation and communication they have. The stall cycles are mostly caused by the Load-Store unit accessing the DRAM memory. The load operation has 4 cycles of programmer/compiler-visible latency. If the memory access takes longer, then the core is stalled until the result arrives. From the table it is

Table 5.4. *Geometric mean of the runtimes (in ms) in simulation and on the FPGA. In brackets is the performance increase normalized to each columns single core value.*

Core count	Simulation (100 MHz clock)	Simulation scaled (Real clock)	Runtime (on FPGA)
1	537.92 (1.0x)	276.55 (1.0x)	1331.77 (1.0x)
2	271.01 (2.0x)	168.88 (1.6x)	774.70 (1.7x)
4	135.56 (4.0x)	79.82 (3.5x)	414.59 (3.2x)
8	67.76 (7.9x)	52.98 (5.2x)	320.94 (4.1x)
16	33.90 (15.9x)	29.39 (9.4x)	284.01 (4.7x)
24	22.62 (23.8x)	22.64 (12.2x)	160.31 (8.3x)

evident that the cores spend most of their time stalled waiting for the external memory access to finish.

The only other possible source of stalls is the mutex unit, which has 8 cycles of visible latency, but can actually take up to N clock cycles, where N is the amount of cores in the system. This is because the mutex unit serves one core in a cycle. However, the effect of these stall cycles is not noticeable since with the simplified kernel launcher there's only one locking operation per core per kernel execution compared to the thousands of DRAM memory operations.

From Table 5.5 it can be seen that the proportion of the stall cycles increases only slightly when the core count is increased. This hints that the memory bandwidth is not completely filled up. If the memory bandwidth would regularly be full of transactions, then perhaps the number of stall cycles would increase more drastically when the core count increases. This effect may start to be visible in vector addition and copying benchmark with the core counts 16 and 24. This makes sense, as those are the two benchmarks with the least computation.

Quite surprisingly, the 24 core count system has consistently smaller proportion of stall cycles compared to the 16 core system. The reason for this is unclear. On one hand, the lower clock frequency of 24-core system allows external memory operations to finish in less cycles. On the other hand, contention for the shared arbiters should increase the amount of stall cycles. Perhaps the effect of clock frequency overpowers the effect of resource contention in this case.

Table 5.5. *Proportion of running cycles (as opposed to stall cycles) to the total cycle count*

	Core count					
Benchmark	1	2	4	8	16	24
matmul	18.4%	20.1%	18.4%	16.8%	10.1%	17.2%
polynomial	28.7%	29.1%	26.3%	23.6%	14.9%	21.7%
vecadd	16.8%	17.7%	13.4%	11.4%	7.3%	9.7%
copy	11.0%	10.7%	8.5%	9.0%	4.3%	4.0%

5.9 Memory bandwidth utilization

External memory bandwidth utilization can be analyzed from the runtime results. The different cores can access the shared resource at different times which creates the interleaved access pattern as shown in Figure 2.1. To see the actual access pattern, the AXI arbiters' inputs and outputs could be monitored to visualize the interleaving effect. Implementing this profiling was not done in this work. The simpler way to estimate the data movement through the DRAM-AXI-interfaces is to compute it directly from the runtime result values. For example, when doing vector addition, 2 values have to be loaded into the core and 1 result value has to be stored back to DRAM. Knowing the total amount of data in a benchmark, the average memory bandwidth of the vector addition benchmark can be computed: $\mathrm{Bandwidth} = \frac{3 * \mathrm{Size\ of\ the\ operand\ array\ (in\ bytes)}}{\mathrm{Runtime\ (in\ seconds)}}$. For the copying benchmark, the calculation is the same except the constant 3 is substituted with 2, since there are only two buffers, one input and one output.

Average memory bandwidths for each core can be seen in Table 5.6. The runtime values of the vector addition and the copying benchmark was used for each of these values. Quite surprisingly, the vector addition benchmark manages to utilize the memory bandwidth better than the simple copying benchmark. There's less address calculations happening in the copying benchmark since it loads only single value, so it would make more sense for it to utilize the bandwidth better.

When analyzing the binaries produced by the compiler using a disassembler, it was clear that the load-store units of the cores aren't used every cycle, even in the simpler copying program. This is because the architecture of the TTA-core is too minimal keep up with the address computation of the input and output arrays. In order to properly estimate the bandwidth utilization, it would be ideal if the cores could make one memory operation every cycle. For this reason, a basic data loading loop was coded manually using TTA assembly. The program loads 64 data words in one loop iteration and stores one of the words to a different output array. Most importantly, since not all values need to be saved back, there is reduced need for address computation. Because of this, the program can initiate a memory operation at almost every cycle. Writing this program in higher level language might've been difficult since the compiler could eliminate the load operations of which the result value is not used. However, when the program is written in assembly, no such optimizations are performed to it when it's being assembled to binary code. The bandwidth utilization reached by this program is listed in Table 5.6 as column 'assembly load'.

As expected, the assembly load-program provides the best bandwidth utilization. However, the vector addition benchmark isn't too far behind, which shows the benefit of latency hiding. The vector addition program can perform a bit of vector computation and proper address calculation for three arrays while not being too far behind the assembly load-program, which only computes address calculations for 1+1/64 arrays.

Table 5.6. *Maximum memory bandwidth of different machines on vector addition, copying and assembly load -benchmarks. The percentage is the utilization compared to the device's theoretical maximum memory bandwidth*

Maximum memory bandwidth (MB/s)		vecadd	copy	assembly load
Core count	1	31.3 (1.5%)	31.1 (1.5%)	41.6 (2.0%)
	2	53.4 (2.5%)	49.9 (2.4%)	67.8 (3.2%)
	4	94.4 (4.5%)	95.3 (4.5%)	132.9 (6.3%)
	8	111.0 (5.3%)	152.4 (7.3%)	170.3 (8.1%)
	16	120.8 (5.8%)	106.2 (5.1%)	215.2 (10.2%)
	24	202.7 (9.7%)	128.4 (6.1%)	238.4 (11.4%)

The maximum memory bandwidth reached was 238.4 MB/s which is 11.4% of the device's theoretical maximum memory bandwidth of 2.1 GB/s [35]. One of the goals of this research was to utilize the external memory bandwidth as efficiently as possible. Therefore, there's still a lot of room for improvement. The main reason why the bandwidth utilization is so low is most likely the use of scalar 32-bit accesses. AXI is a burst-based protocol [27], so the scalar accesses are not optimal. Additionally, since the actual hardware port is 64 bits [35], only half of the interface is used. More ideas to improve the memory bandwidth utilization of the multicore template are presented in Chapter 6.

6 FUTURE WORK

There were some potential ideas for further optimization that could not be pursued within the time limits of a master’s thesis project. Most importantly, the memory access was identified as a critical point of the system, and getting good memory bandwidth utilization might require significant hardware and software redesign.

6.1 Optimizing the external memory access

The current implementation of an AXI arbiter is very inefficient in area as it does all the arbitrating with 64-bit wide signals. The performance of the arbiter is unknown, since it hasn’t been profiled in detail. Careful profiling of the component would show how well the memory bandwidth can really be utilized, and if there are unnecessary stalls in the system. The current setup relies on an AXI arbiter IP, which might be suboptimal for this task. Manual implementation of the arbiter would allow for more thorough analysis of the component.

AXI is a burst-based protocol [27]. It’s designed to be most efficient when multiple consecutive memory addresses are accessed with a single burst transaction. During this research, all accesses to the DRAM memory were scalar 32-bit wide accesses, which are not optimal. TCEMC-toolset already has the support for SIMD-operations. A SIMD processor with very wide SIMD-operations (e.g. 1024 bits) could do burst transactions to load and store the vectors as whole. This would most likely lead to much better utilization of the external memory bandwidth.

For ZYNQ 7020-device, having just four (4) as wide as possible SIMD-cores might be an efficient way to utilize the FPGA, since that would get rid of all the AXI arbiting logic shown in Figure 5.4. This is because the device has four AXI HP-ports to the DRAM. Naturally, this diminishes the effect of latency hiding, since it reduces the number of threads on the chip.

The current setup is based on the assumption of randomly accessible memory, where the software can address any point it wants. However, the OpenCL programming model applies some restrictions to the memory access, which could be utilized to create more optimized data transfer from DRAM to the multicore system. One approach is doing block transfers at a higher level. A part of an OpenCL buffer is moved to local on-chip memory using the efficient AXI burst transfers. Then the multicore system could quickly access that data. Since the entire OpenCL chain is controlled by the TCEMC/PoCL-tools, it could be possible to ensure that the work-items that need certain part of a buffer would always have it available to them on a fast local memory.

OpenCL pipe is a feature introduced in OpenCL 2.0 which makes it possible to move data between the kernels using FIFOs [10]. These can be used instead of OpenCL buffers to move the data as packets. The standard defines pipes only between the kernels, but the feature could easily be extended to be between the host code and the kernel, as was done by Intel in their FPGA SDK [37]. There would have to be a buffer-to-pipe adapter implemented as a function unit of the processor, which would be able to pull data from the host buffer as packets using AXI burst transfers. Then these packets could be used as regular pipe packets inside the kernel. Then another pipe-to-buffer component would push these packets back to the external memory. The pipes would also support multi-kernel producer-consumer patterns.

One more factor associated with the poor memory bandwidth utilization could be the design of the TTA-core. The memory accesses are initiated by the core. Ideally, the core should initiate a memory transfer every cycle. But since the evaluated design was kept as simple as possible, there weren't enough available resources to keep up with all the address computation required when accessing the arrays. Therefore, the memory access density in the program couldn't be consistently kept at 100%. This can be mitigated by more careful design of the TTA-core and making sure that the compiler can schedule consistent utilization for the LSU. Also, adding *base+offset* memory operations would lessen the address computation load of the ALU. This would mean that when sequential elements of an array are accessed, only the *offset* part of the address would have to be incremented and moved to the LSU.

6.2 Dynamic load balancing

The static work-balancing algorithm works well when the loads are equal in their runtime (as was the case in the benchmarks used in this research). However, often the execution speeds are very data-dependent, so they can vary depending on the work-group. In that case, a more dynamic work-balancing system could offer better performance. Additionally, there could be multiple different kernels executing in parallel. These kernels could either come from independent command queues or they could be dependency-free kernels in an out-of-order command queue. Currently, only a single in-order command queue is supported.

Dthread already has a work-stealing feature built-in. If there are more threads created than cores on the device, some of the threads are left in a queue waiting for their turn.

If some core finishes its threads earlier than others, it can go look for waiting threads in the queue and grab them for itself. This feature could easily be used for dynamic work-balancing. The only change that would be needed is to create more threads than there are cores. The current implementation creates only as many threads as there are cores to minimize the number of context switches. The thread-based approach could also be easily extended to support multiple different kernels executing in parallel in different cores.

Another approach would be to implement a shared queue, or multiple queues, containing the work-groups, where the cores could fetch an entire range of work-groups to execute at a time. This would work similarly to the above idea, but instead of doing full context switches, there would only be a single thread running on each core that would access this shared data structure containing the range of work-groups. However, since the shared data structure is accessed by each of the cores, any modifications to it must be done atomically. This increases the external memory traffic and locking operations. However, if there are enough work-groups available, and each core fetches enough of them at a time, this traffic can be minimized. This approach requires very careful programming to avoid all the concurrency problems, while still supporting as much of the OpenCL specification as possible. The former, Dthread-based approach, would probably be a lot easier to develop even though there is the added penalty of slow context switches.

7 CONCLUSIONS

The performance growth of single-threaded systems has stagnated over the few past decades. Thread-level parallelism is a way to have the program make progress in multiple independent tasks. This way, also external resources can be utilized more efficiently. Multicore processor systems are an attractive way to have multiple threads of execution on the same chip. This thesis evaluated the soft multicore scalability of a MCASIP-template called TCEMC [29]. The TCEMC-template makes it possible to create an OpenCL programmable multicore soft processor system out of a TTA-core. The processor system can be synthesized to an FPGA device and OpenCL code can be compiled for it.

The main purpose of this thesis was to find out if there are any obvious bottlenecks limiting the performance of the multicore system on FPGA. The clock frequency was slightly improved by registering the inputs to the shared mutex unit. However, perhaps a rework of the shared mutex unit is needed to get rid of the critical paths still caused by it.

The shared memory access was identified as a critical point of the system, which clearly affects the performance. Filling up the shared memory bandwidth requires good performance from the arbiter. A few different arbiter IPs from Xilinx were evaluated, but there is clearly still room for improvement. Being able to use burst accesses to the external memory is clearly the next optimization to make. A few possible options for this were presented in Chapter 6.

When the number of cores increases, the resource-usage grows accordingly. Because of this, it's important to take advantage of any redundancy found in the system. This thesis presented a simple way to share the instruction memory between two cores to halve the required instruction memory resources.

Using an entire distributed threading library to launch the kernels was found to be unnecessary in the simple, single-kernel, cases. A simple, more static, kernel launching wrapper was created. This reduced both the instruction and data memory usage, while keeping the sustained performance at least as high as before. It should be possible in the future to extend the wrapper to support more concurrent kernels.

The performance increase of the multicore template was found to be 8x in a 24-core system compared to the single-core performance of the same TTA-core. The maximum external memory bandwidth reached was 238.4 MB/s (11.4% of the system's maximum bandwidth).

The optimizations implemented during this research have been integrated into the TCEMC-toolset. This thesis succeeded in finding the parts of the system that don’t scale up well when the number of cores is increased. These parts should be further researched to achieve competitive performance and better external bandwidth utilization.

REFERENCES

[1] Bohr, M. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007), 11–13.

[2] Hennessy, J. and Patterson, D. *Computer Architecture : A Quantitative Approach*. San Francisco: Elsevier Science & Technology, 2011.

[3] Wulf, W. A. and McKee, S. A. Hitting the memory wall: implications of the obvious. *Computer architecture news* 23.1 (1995), 20–24. ISSN: 0163-5964.

[4] Flynn, M. Very high-speed computing systems. *Proceedings of the IEEE* 54.12 (1966), 1901–1909. ISSN: 0018-9219.

[5] Lomont, C. *Introduction to Intel Advanced Vector Extensions*. Intel. 2011. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/intro-to-intel-avx-183287.pdf.

[6] Spector, A. and Gifford, D. The space shuttle primary computer system. *Communications of the ACM* 27.9 (1984), 872–900. ISSN: 0001-0782.

[7] Padua, D. *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011.

[8] *Product Specifications*. Intel. URL: https://ark.intel.com/content/www/us/en/ark.html.

[9] Kirk, D. and Hwu, W. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 2016.

[10] *The OpenCL Specification*. Khronos OpenCL Working Group. 2019. URL: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf.

[11] Dijkstra, E. Solution of a problem in concurrent programming control. *Communications of the ACM* 8.9 (1965), 569.

[12] Gaster, B. *Heterogeneous computing with OpenCL*. 2nd ed. Amsterdam ; Elsevier/MK, 2013.

[13] *OpenMP Application Programming Interface*. OpenMP Architecture Review Board. 2020. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf.

[14] Jääskeläinen, P., Sanchez de La Lama, C., Schnetter, E., Raiskila, K., Takala, J. and Berg, H. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming* 43.5 (2015), 752–785.

[15] Butterfield, A. and Szymanski, J. A Dictionary of Electronics and Electrical Engineering. Oxford University Press, 2018.

[16] Ball, J. Designing Soft-Core Processors for FPGAs. *Processor Design: System-on-Chip Computing for ASICs and FPGAs*. Ed. by J. Nurmi. Dordrecht: Springer Netherlands, 2007, 229–256.

[17] *MicroBlaze Soft Processor Core*. Xilinx. URL: `https://www.xilinx.com/products/design-tools/microblaze.html`.

[18] *Nios II Processors*. Intel. URL: `https://www.intel.com/content/www/us/en/products/programmable/processor/nios-ii.html`.

[19] Severance, A. and Lemieux, G. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on hardware/software codesign and system synthesis*. CODES+ISSS '13. IEEE Press, 2013, 1–10. ISBN: 1479914177.

[20] Kara, K. and Alonso, G. PipeArch: Generic and Context-Switch Capable Data Processing on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 14.1 (2020).

[21] LaForest, C. and Steffan, J. OCTAVO: an FPGA-centric processor family. *Proceedings of the ACM/SIGDA international symposium on field programmable gate arrays*. ACM, 2012, 219–228. ISBN: 9781450311557.

[22] Amiri, M., Siddiqui, F. M., Kelly, C., Woods, R., Rafferty, K. and Bardak, B. FPGA-Based Soft-Core Processors for Image Processing Applications. *Journal of Signal Processing Systems* 87.1 (2017), 139–156.

[23] Cartwright, E., Ma, S., Andrews, D. and Huang, M. Creating HW/SW co-designed MPSoPC's from high level programming models. *2011 International Conference on High Performance Computing & Simulation*. IEEE, 2011, 554–560. ISBN: 1612843824.

[24] Lin, M., Lebedev, I. and Wawrzynek, J. OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices. *2010 International Conference on Field Programmable Logic and Applications*. IEEE, 2010, 458–463. ISBN: 9781424478422.

[25] Lebedev, I., Cheng, S., Doupnik, A., Martin, J., Fletcher, C., Burke, D., Lin, M. and Wawrzynek, J. MARC: A Many-Core Approach to Reconfigurable Computing. *2010 International Conference on Reconfigurable Computing and FPGAs*. 2010, 7–12. DOI: `10.1109/ReConFig.2010.49`.

[26] Hoozemans, J., Jong, R., Vlugt, S., Straten, J., Elango, U. and Al-Ars, Z. Frame-based Programming, Stream-Based Processing for Medical Image Processing Applications. *Journal of Signal Processing Systems* 91 (2019). DOI: `10.1007/s11265-018-1422-3`.

[27] *AMBA AXI and ACE Protocol Specification*. ARM. 2011. URL: `https://developer.arm.com/documentation/ihi0022/b/`.

[28] *Wishbone B4*. OpenCores. 2010. URL: `http://cdn.opencores.org/downloads/wbspec_b4.pdf`.

[29] Jääskeläinen, P., Salminen, E., Sanchez de La Lama, C., Takala, J. and Ignacio Martinez, J. TCEMC: A Co-Design Flow for Application-Specific Multicores. Inter-

national Conference on Embedded Computer Systems: Architectures, Modeling and Simulation IC-Samos. IEEE, 2011, 85–92.

[30] Jääskeläinen, P., Viitanen, T., Takala, J. and Berg, H. HW/SW Co-design Toolset for Customization of Exposed Datapath Processors. *Computing Platforms for Software-Defined Radio*. Ed. by W. Hussain, J. Nurmi, J. Isoaho and F. Garzia. Springer International Publishing, 2017, 147–164.

[31] Hoogerbrugge, J. and Corporaal, H. Register file port requirements of transport triggered architectures. *Proceedings of the 27th annual international symposium on microarchitecture*. MICRO 27. ACM, 1994, 191–195. ISBN: 0897917073.

[32] Hirvonen, A., Tervo, K., Kultala, H. and Jääskeläinen, P. AEx: Automated Customization of Exposed Datapath Soft-Cores. *22nd Euromicro Conference on Digital System Design (DSD)*. 2019, 35–42.

[33] Jääskeläinen, P., Salminen, E., Esko, O. and Takala, J. Customizable Datapath Integrated Lock Unit. English. International Symposium on System on Chip SoC. IEEE, 2011, 29–33.

[34] *IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7*. 2018.

[35] *PYNQ-Z1 Board Reference Manual*. Digilent. Pullman, WA, USA, 2017. URL: `https://reference.digilentinc.com/_media/reference/programmable-logic/pynq-z1/pynq-rm.pdf`.

[36] *Vivado*. Xilinx. URL: `https://www.xilinx.com/products/design-tools/vivado.html`.

[37] *Implementing OpenCL Pipes*. Intel. URL: `https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html#ewa1425651091639`.

A OPERATION SET OF THE EVALUATED PROCESSOR

A detailed description of the evaluated TTA-processor's operation set for reproducibility. The instruction width of the processor is 44 bits.

ALU	
add (1)	Integer addition. Output 3 is sum of inputs 1 and 2.
and (1)	Bitwise AND. Operands 1 and 2 are inputs and 3 is output result.
eq (1)	Equality comparison. Output 3 returns '1' if inputs 1 and 2 are equal and otherwise returns '0'.
gt (1)	Greater-than signed integer comparison. Output returns '1' if input 1 is greater than input 2 and otherwise returns '0'.
gtu (1)	Greater-than unsigned integer comparison. Output returns '1' if input 1 is greater than input 2 and otherwise returns '0'.
ior (1)	Inclusive OR. Operands 1 and 2 are inputs and 3 is output result.
sub (1)	Integer subtraction. Input 1 is minuend, input 2 is subtrahend and output 3 is difference.
xor (1)	Exclusive OR. Operands 1 and 2 are inputs and 3 is output result.
shr1_32 (0)	Arithmetic shift right by one bit.
shru1_32 (0)	Logical shift right by one bit.
mul (3)	32-bit integer multiplication of the inputs 1 and 2 with lower result bits in the output 3.
LOCAL	Accesses address space **Local**.
ld32 (3)	Loads a 32-bit word in little endian byte order and sign-extend it
ldu8 (3)	Loads a byte and zero extends to 32 bits.
ldu16 (3)	Loads a 16-bit word in little endian byte order and zero extends to 32 bits.
st32 (0)	Stores a 32-bit word to little endian byte order.
st8 (0)	Stores a byte to an absolute byte address in memory (identical to STQ).
st16 (0)	Stores a 16-bit word to little endian byte order.

DRAM	Accesses address space **DRAM.**
ld32 (3)	Loads a 32-bit word in little endian byte order and sign-extend it
st32 (0)	Stores a 32-bit word to little endian byte order.
ldu8 (3)	Loads a byte and zero extends to 32 bits.
ldu16 (3)	Loads a 16-bit word in little endian byte order and zero extends to 32 bits.
st16 (0)	Stores a 16-bit word to little endian byte order.
st8 (0)	Stores a byte to an absolute byte address in memory (identical to STQ).
LOCK	Locks addresses in **DRAM.**
lock_read (7)	Reads the lock status of the given address. 0 == unlocked, 1 == locked.
try_lock_addr (7)	Tries to acquires a lock at the given shared memory address. Returns 0 in case did not manage to get the lock, 1 in case it did.
unlock_addr (7)	Unlocks the lock guarding the given address. Does not perform any ownership checks, just unlocks the address unconditionally.
GCU	Control unit (3 delay slots)
jump (0)	Absolute jump to the given instruction address.
call (0)	Calls a function at the given absolute instruction address. The return address is saved in the return address register (RA).
RF	Register file 32 general purpose 32-bit registers with 1 write port and 2 read ports
IU	Long immediate unit Outputs 32-bit immediate number. (None of the buses have a short immediate)