Evaluating Interconnection Networks for Exposed Datapath Architectures

Sebastian Schumb
Embedded Systems Group - University of Kaiserslautern

June 2018

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Kaiserslautern den,	Date	Signature

Acknowledgements

I would like to use this opportunity to thank all the kind people, without whom this master thesis would not have been possible.

First of all I want to thank my supervisors Tripti Jain and Prof. Klaus Schneider, for their patience, encouragement, support and their continued faith in my abilities.

Furthermore I would also like to thank my family for being understanding and supportive as well as all my friends for their continuous participation in my rubber duck debugging efforts.

Abstract

The Synchronous Control Asynchronous Data (SCAD) architecture is an exposed-datapath architecture, which relies on a having a fast data transport network in order to exchange data between its various function units. Interconnection networks developed from radix-based self-routing sorting networks are particularly well suited for this task. This thesis intends to evaluate two types of interconnection networks for a SCAD machine implementation using the Intel HARP hardware platform and the OpenCL runtime library. To that end a network generator is developed, which allows to translate existing hardware circuit description for interconnection networks into OpenCL code, making use vector operations to leverage data parallelism. The consumption of FPGA hardware resources by the different network types is determined using hardware synthesis with the Intel OpenCL SDK for FPGAs and the networks are evaluated with respect to their potential application within a SCAD architecture.

Zusammenfassung

Die Synchronous Control Asynchronous Data (SCAD) Architektur ist eine exposed-datapath Architektur, welche auf ein schnelles Datentransportnetz angewiesen ist, um Daten zwischen ihren verschiedenen Funktionseinheiten aus zu tauschen. Interconnection Netzwerke, basierend auf radixbasierten, selbstroutenden Sortier-Netzwerken sind für diese Aufgabe besonders geeignet. In dieser Arbeit sollen zwei Arten von Interconnection Netzwerken, für eine SCAD-Maschine, die auf der Intel HARP-Hardwareplattform und der OpenCL-Laufzeitbibliothek aufbaut, evaluiert werden. Dazu wird ein Netzwerkgenerator entwickelt, er es erlaubt bestehende Schaltungen für Interconnection Netzwerke in OpenCL-Code, der Vektoroperationen verwendet um Datenparallelität zu nutzen, zu übersetzen. Der Verbrauch von FPGA-Hardware-Ressourcen durch die verschiedenen Netzwerktypen wird durch Hardwaresynthese mit dem Intel OpenCL SDK für FPGAs bestimmt und die Netzwerke werden hinsichtlich ihrer Einsatzmöglichkeiten innerhalb einer SCAD-Architektur bewertet.

Contents

1 Introduction

1.1 Motivation

For decades improvements in semiconductor production techniques allowed to double the number of transistors on a processor die every year, which in combination the rapidly increasing speeds of single transistors resulted in the performance of processors doubling every 18 months. Up until the 2000s Moore's law seemed like a self-fulfilling prophecy for processor designers. Advancements in semiconductor technology slowed down and new ways to improve the performance of new processor generations had to be explored. A trend leading to the introduction of multicore processors. While combining multiple processor cores on the same chip allowed programs to execute multiple tasks in parallel, it also introduced new complexity in the form of synchronization, cache consistency issues and race conditions between the different cores. Since not all software scales well over multiple cores, new ways of boosting a single cores execution speed were developed. Most state-of-the-art processors are superscalar architectures, allowing them make use of instruction level parallelism by reordering instructions to execute them out-of-order and partially parallel to archive higher executing speeds. However, from the programmers and also from the compilers point of view the processor is still sequentially executing instructions and storing data in a central register file. Having to maintain the central register file still remains a performance bottleneck for processor architectures, even with more advanced techniques such as register renaming.

Exposed-datapath architectures do not share this problem. Typically, a program executed on an exposed-datatpath architecture mainly consists of instructions which move data from one function unit to another function unit using some form of bus or interconnection network. Each function unit has its own set of internal registers and can operate independently once its input data has been delivered. While some exposed-datapath architectures still provide a register file to simplify programming it is not necessarily required. Synchronous Control, Asynchronous Data machines function completely without a central register file. Instead, data is stored in buffers attached to the function units and the program executes data transfers from output to input buffers, which appear to be synchronous to the program. However, the actual execution is performed asynchronous. This is achieved by using tags attached to the data that arrives in the input buffers and reordering the contents of the input buffers such that the functional unit can perform its computations as soon its input buffers contain a complete set of inputs with matching tags. Utilizing this form of asynchronous execution enable the SCAD machine to exploit any instruction level parallelism present in the program.

At the same time SCAD machine are also highly customisable, since, given the enough hardware resources, the number of function units of a certain type can be easily adjusted to fit the requirements of certain application. This makes SCAD machines a particularity well suited architecture for FPGA bases accelerators.

The Datat Transfer Network between the function units' buffers is of the major components of a SCAD machine. It has to be highly optimized for the SCAD machine to achieve the best possible performance. At the same time it also needs to flexible enough to scale with different set function units, if the architecture is to be customised for special applications. It is known that self-routing sorting networks can be used as efficient asynchronous interconnection networks, making them an ideal starting point to develop a DTN for use in SCAD machines.

This thesis aims to evaluate two different radix based sorting networks, which could potentially be used as a basis for a DTN for SCAD machine. The networks will be translated from existing hardware circuits into OpenCL and use the Intel Heterogeneous Architecture Research Platform (HARP) system as underlying hardware platform. The HARP system, which is intended to accelerate typical datacenter applications, provides the means to employ FPGAs as OpenCL computed units, which can execute OpenCL kernels, making it an interesting target platform for SCAD based accelerators.

1.2 Goals

The first goal of this thesis is to develop a network generator, which can generate OpenCL code from hardware circuits for interconnection networks. The hardware circuits are provided in the form of levelled circuits by a FSharp library, which has been developed at the Embedded Systems group at the University of Kaiserslautern for research and teaching purposes. Since these networks generated by this library are thoroughly tested and also formally verified in some cases, it was decided to use levelled circuits as a starting point instead of implementing the networks from scratch in OpenCL. The main idea behind this generator is to evaluate whether is it possible to efficiently implement the switches, which make up a major portion of each networks circuits, using OpenCLs shuffle instruction. Moreover, each network should be self-contained in a single OpenCL kernel to make it easy to integrate it into lager designs, such as SCAD machines. To still achieve a certain level of parallelism in the resulting OpenCL code, it was decided to utilize vector operations whenever possible. While this decision may prove beneficial in terms of performance of the FPGA bitstreams compiled from the OpenCL, it also limits the networks to at 16 inputs, since this is the largest vector size supported. This thesis will mainly consider the Koppelman-Oruç network and Narasimha's network, therefore these networks are the primary focus for the network generator. These networks do not support partial permutations, which are inputs to the network where only a portion of the target address are used. This is an intentionally imposed restriction to reduce the complexity of the code generation.

The second goal of this thesis is to evaluate the networks with respect to exposed datatpath architectures. To that end tests have been conducted to ensure that the resulting networks in OpenCL work correctly when compiled to be executed on a GPU or CPU using the standard OpenCL runtime. After the networks have been tested to ensure no incorrect behaviour is introduced in the translation process, they have been compiled for simulation using the Intel OpenCL SDK for FPGAs, which can simulate the execution on the HARP hardware platform. Finally, the network have been synthesised into FPGA bitstreams and the hardware resources consumed by each network have been evaluated.

1.3 Structure

This thesis is structured as follows: After this introduction section 1.4 will provide an overview of related publications and thereby establishing the scientific context. Section 2 presents the necessary preliminaries, by first introducing the concept of sorting networks and then discussing the two networks considered in this thesis in greater detail. Following that section 2.2 provides a short introduction to OpenCL and section 2.2 presents the Intel HARP platform.

Next section 3 describes the translation process to OpenCL. The levelled circuits, which provide the starting point for the code generation are explained in section 3.1. Section 3.2 introduces the first intermediate representation used in the process, the integer gate circuit. Afterwards the process of vectoring the gates is discussed in section 3.3, which is followed by section 3.4 explaining the actual OpenCL code generation.

The evaluation of the generated networks is presented in section 4. Potential future work is discussed in section 5, before section 6 concludes this thesis.

1.4 Related Work

This section aims to provide a short overview of other publications related to this thesis.

The intended field of application for the interconnection networks discussed in this thesis are accelerators employing exposed data path architectures. Specifically architectures based on the Synchronous Control Asynchronous Dataflow paradigm, SCAD for short, which has been introduced in [3]. This type of networks is an ongoing field of research at the Embedded Systems group at the University of Kaiserslautern. The SCAD machine architecture presented in [21] has been implemented in OpenCL to enable it to be deployed on the Intel HARP platform. As of now this implementation still lacks an efficient OpenCL implementation of an interconnection network, which is one of the core components of the SCAD architecture.

Interconnections between different function units was one of the first applications suggested for sorting networks in [1]. The basic idea of radix-based interconnection networks has already been investigated in [5] and [2]. The Koppelman-Oruç network introduced in [15] and Narasimha's network introduced in [18] refined this ideas further towards the concept self-routing permutation networks. Based on this results the Embedded Systems group has developed the SelectorTree network [12]. Inspired by these results [13] presents a radix-based self-routing and non-blocking interconnection network and compares its capabilities as an on chip interconnect to networks, based on a hardware design in 65nm CMOS chip technology. A common problem to most interconnection networks based on radix sorting are partial permutations, which occur if not every input of the network is used in each cycle, leaving some target addresses unused. [14] presents a general approach to convert a binary sorter into a tenary split module, allowing any radix-based interconnection network to cope with partial permutations.

Introductory work on the advantages of using OpenCL as a high-level alternative to more traditional hardware description like Verilog or VHDL for FPGA application can be found in [22]. A compiler toolchain and workflow to generate Verilog code from OpenCL kernels is presented in [6]. The proposed solution uses FPGA hardware by Altera, now acquired by Intel, which similar to the HARP hardware platform discussed in section 2.3. [20] discusses a medium scale deployment of FPGA based accelerators in a Mircosoft datacenter which is used to accelerate the Bing Search Engine, reducing the datacenters power consumption at the same time.

2 Preliminaries

The following section contains the necessary preliminaries for the interconnection network generator presented in the next section. Section 2.1 introduces the basic principle of sorting networks and explains how they can be used to dynamically interconnect function units. It also discusses the structure and the functionality of the two networks used in this thesis in greater detail. Section 2.2 contains a short introduction into the OpenCL programming language as well as the OpenCL runtime environment, while it also establishes the necessary nomenclature to discuss OpenCL applications. Finally, Section 2.3 presents an overview over the Intel Hardware Accelerator Research Program, its underlying hardware components and its integration with the OpenCL runtime.

2.1 Sorting networks

The performance of dataflow architectures is heavily dependent on an efficient dynamic interconnection between the different processing elements. Likewise, exposed-datapath architectures like the SCAD machine also require a fast interconnect for their function units. While this could potentially be solved using buses interconnecting the components, the necessary bus arbitration usually introduce wait times and therefore blocks the execution of processing elements, which is highly undesirable for a partially asynchronous architecture like scad. This can be addressed by using a crossbar, which allows transferring multiple inputs to multiple outputs in parallel. However, for a crossbar connecting n inputs to n outputs not only requires n^2 switching elements, in addition to that a global conflict free configuration for these switching elements has to be computed every time, the routing between inputs and outputs is changed. Self-routing Interconnection networks offer a better solution as they can be constructed in using fewer switching elements than a full crossbar and the routing decisions can be determined locally at each switching element, while the data is transported through the network.

The basic principle behind self-routing interconnection networks is closely related to sorting networks. A sorting network is usually described as a number a compare and swap units interconnected by wires. The units have two inputs and two outputs. They compare the values at their inputs and either forward them directly to the outputs if the first input is greater the second one, or swap them, outputting the first input to the second output and the second input to the first output if the first input is less or equal to the second one. Different network structures can be constructed from compare and swap units, which take a fixed number of values as input and sort them while transporting them to the output. The most useful aspect of sorting networks is that the routing decision, whether a compare and swap unit swaps the inputs, is entirely local to the unit and can be determined on the fly only considering the two inputs. Therefore, the data can be routed through the network without a global configuration specifying the routing. At the same time a sorting networks for n inputs can be constructed from less than n^2 compared and swap units, thus also providing a smaller alternative to crossbars in terms of hardware elements required. The principal of sorting networks has been widely adopted in the field of parallel computing as depending on the networks exact structure, a significant portion of the compare and swap can be executed in parallel.

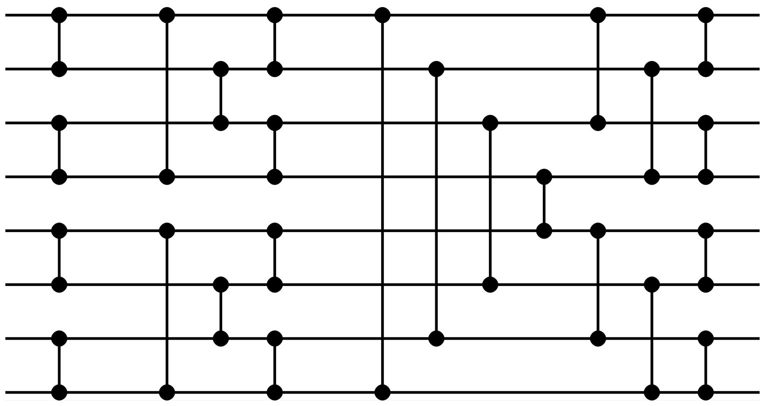
Figure 1: A bitonic sorting network for 8 inputs

The bitonic sorting network first introduced by Batcher [1] can be constructed recursively for 2^k many inputs, for an arbitrarily large k. Figure 1 shows a bitonic sorting network for eight inputs. The compare and swap units are denoted as vertical lines with dots at each end, whereas the horizontal lines denote the wires carrying the data through the network. Batcher also proposed to implement interconnection networks for computer hardware based on the principle of sorting networks.

To use a sorting network as an interconnect between different components of a hardware architecture, the values transported between the inputs and outputs have to be extended to contain an address and as well as data. The sorting and therefore the routing is based on the address portion of the values, whereas the data is simply transported alongside the addresses through the network. For this thesis two kinds of self-routing interconnection networks are considered. Both networks implement a form of radix based binary sorting, which means both sorting networks split their inputs based on the most significant bit of the address before forwarding it to recursively generated networks with half as many inputs.

The Koppelman-Oruç network has been introduced in [15]. A ranking circuit generates local addresses, which are used to route the data through a cube network [17]. The circuit computes the prefix sum over the most significant bit of the actual destination addresses: $r_i = \sum_{j=0}^{i} msb(a_j)$ To reduce the number of adders required for the prefix sum, the ranking circuit is implemented using a tree of adder gates. For a network with n inputs the greatest possible rank occurs at the last output of the ranking tree circuit and can range from 0 in case all the most significant bits are 0 to n in case all most significant bits address bits are set to 1. This results to the local addresses being one bit wider than required for routing. For example in a network with 8 inputs, the ranks will range from 0 to 8, thus requiring 4 bit wide integers.

However, for the routing 3 bit wide addresses ranging from 0 to 7 would be sufficient. To resolve this issue the ranks can be computed as $r_i = -1 + \sum_{j=0}^{i} msb(a_j)$ instead.

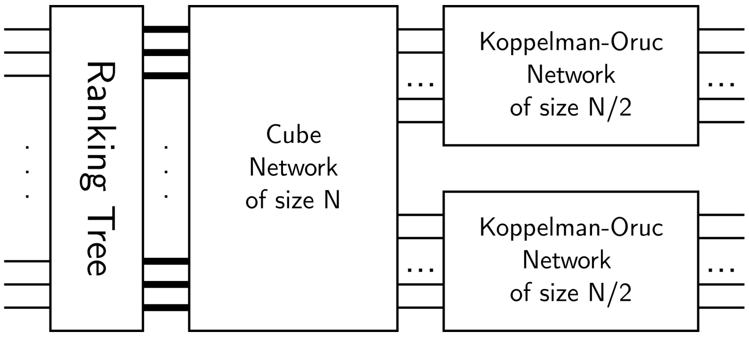
Ranking Tree
Cube Network of size N
Koppelman-Oruc Network of size N/2
Koppelman-Oruc Network of size N/2

Figure 2: Recursive structure of the Koppelman-Oruç network

The recursive structure of the Koppelman-Oruç network is shown in figure 2. After the ranks are computed a cube network is used to route the ranks, the addresses and the data to their correct position according to the ranks. Each switch in the cube network use the most significant bit of the address and the least significant bit of the rank to determine its routing decision. After each switch the least significant bit of the rank is no longer relevant to the routing process and therefore it does not need to be forwarded to the next switch. At the output-side of the cube network the ranks have been consumed completely. The upper and lower half of the cube networks output are fed into two Koppelman-Oruç networks half the initial size. Similar to the least significant bit of the rank not being forwarded to the next switch, the most significant bit of the address is dropped before the two smaller networks. Hence, the ranks inside the smaller network will be calculated based on the second most significant bit of the original addresses.

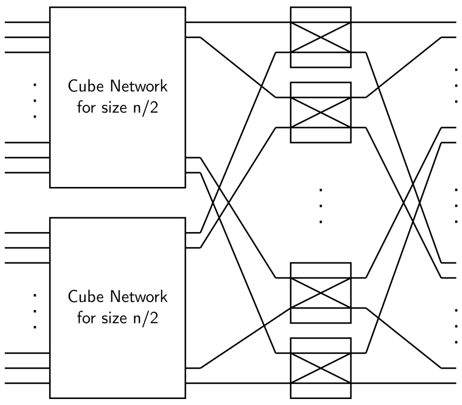

Figure 3: Recursive structure of the cube sorting network

The recursive structure of the cube network is shown in figure 3. In the beginning the upper and lower half of the input are each sorted by cube network with half the input size. The outputs of these networks are then processed by a final column of switches, where the lower input of each switch comes from the lower network and the upper input comes from the same output of the upper network.

r_u	r_l	a_u	a_l	Position
0	0	0	0	don't care
0	0	0	1	transpose
0	0	1	0	identity
0	0	1	1	impossible
0	1	0	0	don't care
0	1	0	1	identity
0	1	1	0	identity
0	1	1	1	identity
1	0	0	0	don't care
1	0	0	1	transpose
1	0	1	0	transpose
1	0	1	1	transpose
1	1	0	0	don't care
1	1	0	1	identity
1	1	1	0	transpose
1	1	1	1	impossible

Table 1: Switch positions in Koppelman-Oruç networks

The position of the switches in the cube network is determined by the most significant bit of the addresses and by the least significant bit of the ranks according the function specified in table 1, where r_u and r_l refer to the least significant bit of the ranks at the upper and lower input and where a_u and a_l refer to the most significant bit at the upper and lower input. Transpose is the crossed switch position, where the inputs are swapped and identity is the straight-through position where both inputs are directly forwarded to the outputs. This function can be simplified to some extent if the two impossible combinations are added to the don't-care-set. The resulting boolean function is $p = (a_u \land \neg r_l) \lor (\neg a_u \land r_u)$ or $p = a_u ? \neg r_l : r_u$ if a multiplexer is used. It is also possible to use the lower address bit resulting in the complementary functions: $p = (a_l \land r_u) \lor (\neg a_l \land \neg r_l)$ and $p = a_l ? r_u : \neg r_l$. p being true sets the switch in the transposed or crossed position and p being false sets the switch in the identity or straight-through position.

All networks used in this thesis are provided by a network generator library, which is actively used in both teaching and research in the Embedded Systems working group. The Koppelman-Oruç network generated using this library slightly differs from the original version presented here as it computes the ranks using the negated most significant bits of the addresses. It then uses a reverse-banyan/flip-shuffle network with reversed outputs instead of the cube network to route the data and addresses to their local destinations. The reverse-banyan/flip-shuffle network splits the inputs in half, where one half will have the most significant bit of the addresses set and the other half will not. Each half can then be sorted further by a recursively constructed network with half as many inputs. This is different to the cube network, which first uses two half-size networks to sort each half of its inputs and then merges their outputs using the final column of switches. In order to simplify the generator library the alternative structure has been chosen, which is closer to that of the other networks being generated, thus allowing for a more generalized generator.

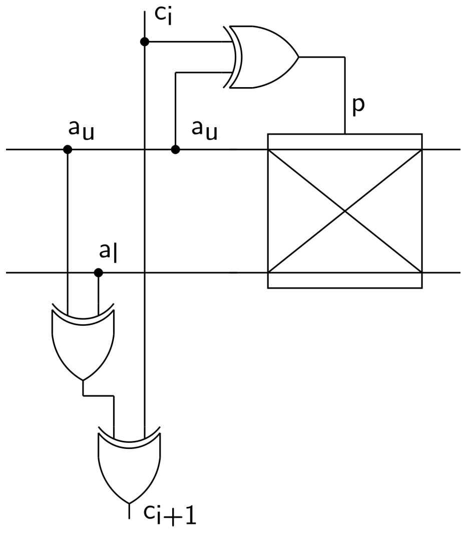

Figure 4: Circuit for determining a switch position in Narasimha's network

The second networks is Narasimha's network, which was introduced in [18]. In contrast to the Koppelman-Oruç network, it does not require ranks or any other form of local addresses to be routed alongside the data and the destination addresses. Instead, the circuit shown in figure 4 is used to determine the switch positions. It operates on the two most significant bits a_u and a_l of the addresses at the upper and lower input as well as the additional signal c_i coming from the switch directly above it in the same column. The switch position itself is computed using a XOR gate as $p = c_i \oplus a_u$ and the signal c_{i+1} for to the switch below the current one is computed as $c_{i+1} = (a_u \oplus a_l) \oplus c_i$. If the gate is the topmost switch in a column the signal c_0 is initialized as false. Likewise, the bottommost gate does not need the gates computing c_{i+1} as there is no switch below it, which would use the signal.

Like the Koppelman-Oruç network, Narasimha's network can be constructed recursively. Figure 5 shows the structure of the network, without the boolean logic for each switch. The first column of switches with N switches in total is connected two smaller networks, with $\frac{N}{2}$ inputs. Each odd input of the network is connected to an upper input of a switch in this first column, whereas each even input is connected to a lower input. The lower outputs of the switches are connected the lower half-sized network, while the upper network is connected to the inputs of the upper

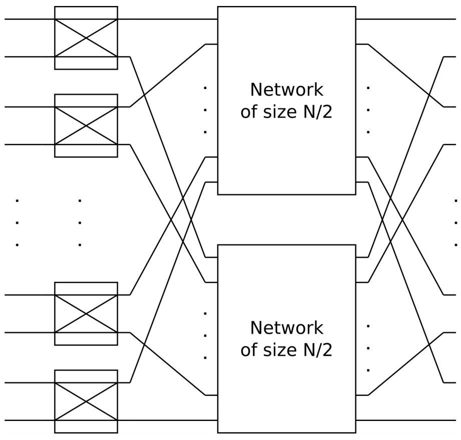

Figure 5: Recursive structure of Narasimha's network

smaller network. The outputs of the two smaller networks are connected to the overall network outputs in an interleaved fashion. Each even output originates from the lower network and each odd input is connected to the upper network. Similar to Koppelman-Oruç network, the most significant bit is stripped from the addresses after it has been used for routing amd the address and data pairs are fed into the smaller networks. Hence, the second most significant bit of the addresses is used to determine the switch position for the first column in the smaller networks.

2.2 OpenCL

The Open Compute Language [19] is a software framework for parallel computing which is developed and maintained by the Khronos Group. Its architecture is split into the host system, which can distribute work units to one or more compute devices. The term is OpenCL is used to refer to actual language used on for writing programs for compute units as well as to refer to the supporting libraries for the host system that provides the communication and synchronization between the host and the compute device. Programs written for a compute unit are called kernels in the OpenCL terminology. As one of OpenCLs core features is the support of heterogeneous architectures, implementations and tools for various different types of compute devices exist. The most common use case for OpenCL is x86-system as host, that offloads calculations to a graphics card which serves a compute unit. In addition to GPUs, OpenCL can also employ CPUs, DSPs, various types of dedicated hardware accelerators and FPGAs as compute units. Each compute unit can offer multiple processing elements which execute kernels. In case of a CPU each core can be considered a processing element, while in case of GPUs a processing will consist of a group of shader units.

Using OpenCL parallelism can be achieved in several different ways. First of all it is possible to execute kernels in parallel on different compute units or on different processing elements of the same compute unit. While this task based parallelism is trivial to implement, it is often desirable to have a more granular type of parallelism. To that end OpenCL offers different ways to make use of data parallelism. If the input data can be split up in smaller work units, which are basically chunks of data that can be processed independently of each other, it is possible to execute multiple instances of the same kernel in parallel, each one processing the work items that make up one work unit. Furthermore, it is also possible to exploit data parallelism from within the same kernel. For this the OpenCL programming language provides vector operations for vectors of up to 16 elements. These operations include arithmetic operations, boolean logic and the permutation of elements inside a vector. The operations are mapped to vector instructions or hardware units if the underlying hardware of the compute unit provides them.

The OpenCL language is based on the C programming language, more specifically the C99 standard [11], with additional datatypes and restrictions to facilitate parallel execution and vector operations, while maintaining support for the various different types of compute units. Since each OpenCL kernel needs to be expressible as a hardware circuit for FPGA based compute units, recursive functions are forbidden in OpenCL. For the same reason dynamic memory allocation is not allowed. The memory requirements for kernel have to be determined via static analysis at compile time in order to generate code for some architectures. Similarly, function pointers are not allowed as they not only can be used to create recursion, but also since they can result in complex control flows, which are difficult to analyse statically during compile time. An additional restriction for function is that parameters are only allowed to be of primitive types or vector types, prohibiting the programmer from passing pointers or arrays to functions as the size of data structure accessible via a pointer might not be determinable at compile time. Further restrictions are necessary, where ever the behaviour of the C language becomes hardware dependent. This is for example the case for bit fields, a feature of structs. For those the compiler usually inserts padding to make efficient use of the width of a memory word of the underlying architecture. Depending on how these data structures are accessed the program might read parts of this padding or write to it. Since OpenCL kernels can be compiled for any of the supported architectures, this may lead to undefined or at least unexpected behaviour of the program and is therefore forbidden.

The memory accessible from within a kernel is split into four different regions. Data stored in the global region can be read and written by all kernels currently executed. It is therefore necessary to utilize synchronization mechanisms such as memory barriers to ensure that a consistent state is read from this region and no write conflicts occur. The second region is the read-only region, which can be written by the host system, but is read-only from the processing element's and therefore from the kernel's points of view. It can be used to store small amounts of data, such as constants, which are shared among all kernels. Since the synchronization required to safely modify the globally shared memory affects all running kernels, it can be a rather costly operation in terms performance. For many applications it is sufficient if a smaller groups of kernels can share data locally only requiring the kernels inside a group to synchronize their modifications. To this end OpenCL provides so called local memory, which is only shared between a small group of processing elements. Lastly there is also the per element region which is only accessible from the current processing element. This region can be used to store temporary data such as the local variables of a kernel.

2.3 Intel HARP

The Intel Hardware Accelerator Research Program [7] provides a hardware platform and software framework to combine FPGA based accelerators with Intel Xeon CPUs. The main goal of the program is to provide a flexible universal accelerator architectures that can improve the performance and power efficiency of many applications typically found in a datacenter environment. It aims to provide a base for heterogeneous architectures, which can achieve a significantly better performance and scalability than homogeneous approaches distributing workloads over multiple CPU cores or CPUs.

In other FPGA based accelerator architectures exchanging data between the CPU and the FPGA is often one of the major performance bottlenecks. To solve this HARP provides the FPGA direct access to the system memory, allowing the FPGA to work on the same data alongside the CPU, without the overhead of data transfers between the two. This is made possible by the Quick Path Interconnect, a low-latency interconnect between the CPU, the FPGA and the systems' memory. QPI enables a fast exchange of routed messages between the different components, via multiple bidirectional point-to-point links. When accessing the memory via QPI, an extended version of the MESI protocol ensures that the CPU and the FPGA with their various caches retain a coherent view of the memory.

The API necessary for software to interface with the FPGA is provided by the Intel Quickassist Architecture and the Accelerator Abstraction Layer. While the IQA is an interface to synchronize and exchange data with the FPGA via shared memory regions, the drivers necessary to load bitstreams and configuration data into the FPGA are made avialable by the Accelerator Abstraction Layer. Intel offers an SDK [9] consisting of the IQA, the Accelerator Abstraction Layer, a version of Quartus Prime, a tool for logic synthesis formerly developed by Altera, and aocl, a compiler used to generate register transfer level logic from OpenCL kernels.

Using the SDK, bistreams for the FPGA can be generated in two ways. First it is possible to use VHDL or Verilog to implement the algorithms and use IP cores provided Intel for the QPI communication. While this path offers the most degrees of freedom in terms of design, it is also more complex since it does not only require experience in circuit design for FPGAs, but in addition to that it also requires some degree of knowledge about QPI and the necessary IP cores.

The second path is a lot less demanding as it based around an OpenCL compiler, which can compile kernels to hardware circuits that can be used on the FPGA. OpenCL is already widely used in distribute and parallel computing, which allows developers to tap into an existing ecosystem of documentation, examples and libraries. In contrast to working in hardware description languages, which have semantics

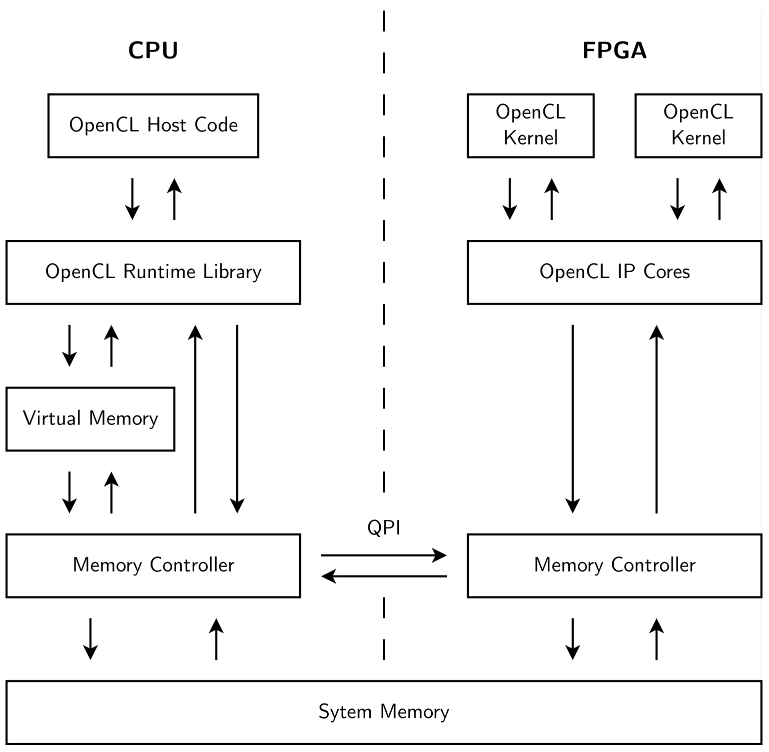

Figure 6: Intel HARP architecture overview

based on circuits, OpenCL has semantics close to most imperative programming languages and therefore a much lower barrier of entry for developers coming from a software background. It is also worth mentioning that existing software written in OpenCL, could in theory be ported to HARP accelerators with relatively few modifications. Likewise, the host application also only requires minimal modifications and can be compiled with any C++-compiler using the headers for the OpenCL runtime included in the SDK. Furthermore, the OpenCL compiler has the added benefit of automatically generating the necessary lower level interfaces, which simplifies the development process significantly. On the host-side the FPGA appears as a standard OpenCL compute unit, which can be accessed like any other compute unit. This becomes especially useful when testing and debugging kernels. Generating bistreams for an FPGA is a computationally expensive process that can take up to several hours for complex designs. Since OpenCL can be compiled quickly for CPUs or GPUs it is possible to use these platforms for testing instead. For these reasons this thesis will primarily focus on the OpenCL part of the HARP platform.

An overview over the HARP architecture for OpenCL is shown in figure 6. On the host-side an applications is running on top of the OpenCL runtime provided by the Intel SDK. The OpenCL runtime takes care of loading a precompiled bistream for the applications kernels into the FPGA and configures it to act as a compute unit for the application. The application can then setup memory regions to pass data to the FPGA or retrieve data from it, which are accessible trough the operation systems regular virtual memory interface. If required, e.g. for synchronization or setup purposes, the runtime can also bypass the virtual memory layer and access the memory directly via the CPUs memory controller. The memory controller attached to the CPU will use a QPI connection to the FPGAs memory controller to ensure that the caches on both sides are kept in a coherent state. On the FPGA-side the higher level parts of the memory controller, QPI connectivity and a hardware circuit version of the OpenCL runtime are provided by so called IP cores. IP cores are precompiled chunks of FPGA bistreams, which are usually shipped without their source code in order to protect the intellectual property contained in them. The cores are inserted automatically by the compiler while compiling the FPGA bistreams. The hardware circuit can contain multiple instances of the same kernel for parallel execution, or instances of different kernels operating in sequence on the same data, giving the FPGA similar capabilities to a regular compute unit.

3 Generating Permutation Networks in OpenCL

The permutation networks used in this thesis are generated by a previously existing software library. This library has been developed for research and teaching purposes at the embedded systems working group at the Technical University of Kaiserslautern. It can generate various different types of networks, which have been extensively tested and in some cases formally verified. The library is designed to generate hardware circuits and therefore a netlist format, describing logic gates and the wires interconnecting them, is used as its output format.

From a semantic point of view this format is vastly different from OpenCL code. In a hardware circuit the every gate will read its inputs and compute its corresponding output instantaneous and in parallel. In contrast to that OpenCL kernels are executed instruction after instruction sequentially, like regular C programs. The parallelism in OpenCL comes from vector operations or from multiple kernels running in parallel. Mapping the hardware circuits to OpenCL can be achieved by finding a sequential schedule for the operations performed by the circuit gates and expressing these operations as sequential code. Unfortunately this approach also removes any inherent parallelism in the process. To reintroduce some degree of parallelism, instructions can be grouped together into vector operations, using vectorisation techniques similar to those employed by modern compilers to make efficient use of vector extensions to CPU instructions sets. This results in a single OpenCL kernel containing the entire network, which uses vector operations for parallelism. An additional benefit of this approach is that no synchronization between different kernels is required to run the network, resulting in a self-contained module that can easily be used as a part of larger programs.

A further difference between the hardware circuit and an OpenCL kernel is that in the hardware circuit the operations of gates are defined on the bit level, whereas OpenCL mostly works on a complete integer or floating-point numbers. Therefore, multiple bit-level operations have to be merged into one integer operations if possible and any remaining bit-level operations have to translated directly to appropriate integer operations.

The steps necessary to transform a hardware circuit, into an OpenCL kernel will be explained in detail in following sections of this document. Section 3.1 explains the netlist format generated by the library in detail. The translation of the bitwise operations into integer operations will be presented in section 3.2 before section 3.3 will show how the resulting stream of operations is first scheduled and then vectorized. Finally, section 3.4 will explain how OpenCL code is generated from the stream of vector operations.

3.1 Levelled Circuits

The networks generated by the FSharp library are in the levelled circuit format. It is based on the principle of circuit gates interconnected by wires. Wire are only a single bit wide and do not retain any state on their own.

```
{
    name = "NET_BIN_HC0_Nara94_1_4";
    inputs = [|[|"x0_0"; "x0_1"; "x0_2"|];
            [|"x1_0"; "x1_1"; "x1_2"|];
            [|"x2_0"; "x2_1"; "x2_2"|];
            [|"x3_0"; "x3_1"; "x3_2"|]|];
    outputs = [|[|"w23"|]; [|"w24"|];
            [|"w25"|]; [|"w26"|]|];
    levels = [
        [(XOR ("x0_2","x1_2"), [|"w1"|]);
        (SW ("x0_2",[|"x0_0"; "x0_1"; "x0_2"|],
        [|"x1_0"; "x1_1"; "x1_2"|]),
        [|"w2"; "w3"; "w4"; "w5"; "w6"; "w7"|])];

        [(XOR ("x2_2","w1"), [|"w8"|])];

        [(SW ("w8",[|"x2_0"; "x2_1"; "x2_2"|],
        [|"x3_0"; "x3_1"; "x3_2"|]),
        [|"w9"; "w10"; "w11"; "w12"; "w13"; "w14"|])];

        [(SW ("w4",[|"w2"; "w3"|],[|"w9"; "w10"|]),
        [|"w15"; "w16"; "w17"; "w18"|]);
        (SW ("w7",[|"w5"; "w6"|],[|"w12"; "w13"|]),
        [|"w19"; "w20"; "w21"; "w22"|])];

        [(SW ("w16",[|"w15"|],[|"w19"|]),
        [|"w23"; "w24"|]);
        (SW ("w18",[|"w17"|],[|"w21"|]),
        [|"w25"; "w26"|])]
    ];
}
```

Listing 1: Levelled circuit for a Narasimha's network

Listing 1 shows an example of a levelled circuit. The circuit is represented by a struct containing inputs, outputs and a nested list of gates. The inputs and outputs are given as a list of lists of strings. In the inner list, the strings represent the name of the input or output wires. The surrounding list groups wires into bitvectors, which contain the data and the destination address for each input or output. The example in listing 1 shows a network with 4 inputs where each input consists of a one bit wide message `xn_0`, and a two bit wide address `xn_1, xn_2`. The outputs are four one bit wide bit vectors containing only the message bits, as the address bits are dropped once they are no longer required for the routing process.

The gates in of the circuit are stored in separated levels. A gate inside a level only uses inputs generated by gates in previous levels, while conversely the output generate by a gate is only connected to gates in following levels, implying that there are no data dependencies between gates of the same layer. This property becomes especially useful when evaluating a levelled circuit sequentially, e.g for simulation purposes. Since the gates within the same level are independent of each other their outputs can be computed in an arbitrary order, as all of their inputs will already be available from previous level and their outputs will not be used until the next level is evaluated. Due this restriction it is also impossible to construct a levelled circuit containing loops, further simplifying sequential evaluation.

Type	Inputs	Output	Operation
NOT	x	$[\neg x]$	Boolean negation
AND	x, y	$[x \wedge y]$	Boolean conjunction
OR	x, y	$[x \vee y]$	Boolean disjunction
XOR	x, y	$[x \oplus y]$	Exclusive disjunction
HA	x, y	$[x \oplus y,\ x \wedge y]$	Half adder
FA	x, y, z	$[x \oplus y \oplus z,\ x \wedge y \vee z \wedge (x \oplus y)]$	Full adder
PG	g_1, p_2, g_2, p_2	$[p_1 \wedge p_2,\ g_2 \vee (p_2 \wedge g1)]$	Propagate-generate gate
MX	$c,\ [x_0, x_1,\ ...\ x_n]$, $[y_0, y_1,\ ...\ y_n]$	$[c?y_0{:}x_0,\ c?y_1{:}x_1,\ \ ...\ c?y_n{:}x_n]$	2:1 multiplexer
SW	$c,\ [x_0, x_1,\ ...\ x_n]$, $[y_0, y_1,\ ...\ y_n]$	$[c?y_0{:}x_0,\ c?y_1{:}x_1,\ \ ...\ c?y_n{:}x_n,$ $c?y_0{:}x_0,\ ...\ c?y_n{:}x_n]$	Switch gate

Table 2: Gate types in levelled circuits

Each level is stored as a list of gates with inputs and outputs. Depending on the gate the inputs can either be single wires, which is the case for the boolean logic gates, or a list of wires representing a bit vector. The outputs are always in bit vector form, which for the simpler gates only contains a single element. Similar to the inputs and outputs of a circuit, wires are stored as strings, containing the name of the wire, whereas bit vectors are stored as list of wires. A complete list of all gate types with their inputs and outputs can be found in Table 2.

The AND, OR, XOR and NOT gates compute the boolean conjunction, disjunction, exclusive disjunction and negation of two inputs. Additional HA, FA, PG gates provide a way to express half adders, full adders and propagate generate in a more compact and readable fashion. Similarly, the MX gate acts as a syntactically compact 2:1 multiplexer for bit vectors of arbitrary length.

The SW gate forms the core of every permutation network. It takes a control wire and two bit vectors of equal size as inputs and outputs a single bit vector with twice the size of the inputs. In case the control input is false the output of a switch gate is the concatenation of the two input bit vectors: $[x_0, x_1, \; ... \; x_n, y_0, y_1, \; ... \; y_n]$. If the control input is set to true the order of the two input vectors in the output is swapped: $[y_0, y_1, \; ... \; y_n, x_0, x_1, \; ... \; x_n]$. Using additional logic to generate the control signal, the SW gate provides a primitive to implement the compare and swap units used as basic building block for permutation and sorting networks.

3.2 Integer Gates

The levelled circuits discussed in the previous section operate on the level of single bits, whereas in OpenCL code the basic data structures are integers or floating-point numbers.

While the C99 standard [11] specifies the `bool` type for boolean values, for efficiency reasons this type is usually implemented as a type alias for an integer type of the same width as the registers of the underlying architecture. In order to avoid any issue caused by potential integer overflows, and to allow implicit narrow type conversions the C compiler generates additional code to limit assignments to boolean variables to the values 0 or 1. This means of 0 will be stored as 0, while assignments of any non-zero value will be stored as one. Due to the implicit conversion the same applies whenever one of the boolean operations `!`, `&&` or `||` are used on integers types or whenever an integer is used in place of a boolean, for example as condition in an if-statement.

A levelled circuit can be translated directly into an OpenCL kernel, by mapping every wire on a single to a boolean variable. However, the resulting code would be far from efficient and might also prove difficult to optimize for a compiler. A more elegant approach should leverage the integers which are already present in the levelled circuit in the form of bit vectors and translate the operations applied to them into integer operations in the OpenCL kernel. To perform this translation efficiently, it is beneficial to split the input and output bit vectors, which contain both the data and its destination address, into separate integers.

The translation also needs to address cases where only a smaller portion of a bit vector is used as an input to a gate, e.g. when the most significant bit of an address is extracted. In the levelled circuit this case is handled simply by addressing the single wire by its name. Levelled circuits also can represent arbitrary permutations of a bit vector by rearranging the wire names inside it. This is not trivially possible with an integer in OpenCL. Instead, one possible way to implement this is to mask out the necessary bits and rearrange them using shift operations. Therefore, an intermediate representation for the circuit was developed, that uses integers instead of bits and bit vectors. It introduces a special SUBSIG gate, which can extract bits from an integer and can store them in arbitrary permutations in its output integer.

In addition to that the translation process also separates the integers in two different groups: the ones used in address and rank calculations and the ones exclusively used to transport data within the circuit. This distinction allows to efficiently split SW gates during the process of translating the levelled circuit to the integer circuit, while also providing additional information for later stages of the OpenCL code generation. Furthermore, the data type used to represent the integer signals connecting the integer gates also keeps track of which signal contains a smaller portion of a larger signal. Keeping this reference allows to delay inserting the necessary SUBSIG gates until after the translation to the integer circuit is completed, which greatly simplifies the process.

Table 3 contains an overview of gates available in the integer circuit. In compar-

Type	Inputs	Output	Operation
NOT	x	$\neg x$	Boolean negation
AND	x, y	$x \wedge y$	Boolean conjunction
OR	x, y	$x \vee y$	Boolean disjunction
XOR	x, y	$x \oplus y$	Exclusive disjunction
MX	c, x, y	$c?y{:}x$	2:1 Multiplexer
SW	c, x, y	$c?y{:}x,\ c?y{:}y$	Switch gate
SUBSIG	x	y with $y \subseteq x$	Extract the bits in y from x
RANKINGTREE	$x_0, x_1 ... x_n$	$r_0, r_1, ... r_n$	Compute ranks of inputs

Table 3: Gate types in integer circuits

ison with table 2 it becomes apparent that the gates necessary for building adders and adder trees have no corresponding gates in this intermediate representation. This is an intentional design decision as they could in theory all be replaced by a single gate for the addition of two integers. To make use of this gate, additions of bit vectors would have to identified in the levelled circuit and the gates would have to be merged into an addition gate for integers. The required structural analysis for this is rather complex for a generic levelled circuit. A much simpler solution can be found by narrowing the problem down to half adders and full adders used within the circuits for permutation networks. The Nara94 networks do not make use of adders at all, so for translating networks of this type no special handling is required.

The Koppelman-Oruç networks on the other hand do make use of half and full adders, but only inside their ranking trees. From a mathematical point of view the ranking trees basically compute a prefix sum over their inputs, which can be expressed as vector operations. Once the inputs and outputs of ranking tree are known, OpenCL code to replace it can be generated trivially. The remaining problem of inferring the inputs and outputs of the tree from the circuit structure, has been avoided by modifying network generator function to return a list of the bit vectors, which act as the tree's inputs and outputs, alongside the circuit itself.

Alogrithm 1 is used to eliminate ranking trees from a levelled circuit based on their inputs and outputs. Starting from two sets of wires I_{trees} containing all inputs wires of ranking trees and O_{trees} containing all output wires of ranking trees, it iterates over all levels L_{full} inside the levelled circuit C_{full}. For each gate it encounters there are two possible cases: The first one is that input wires of the gate are all part of the set of tree inputs, and none of them are part of the set of tree outputs. In that case the gate belongs to a ranking tree and does not need to be kept in the circuit. Instead, its outputs are added to I_{trees} to identify gates further down the ranking tree. The other possible case is that one of the inputs is not contained in I_{trees} or that the inputs are all part of the trees output. In this case the gate is either completely independent of the actual ranking tree, or it only uses the trees output without being part the tree itself. These gates have to be kept inside the circuit and

Algorithm 1 Stripping ranking trees from a levelled circuit

```
 1: $C_{strip} \leftarrow [\ ]$
 2: for all $L_{full} \in C_{full}$ do
 3:     $L_{strip} \leftarrow [\ ]$
 4:     for all $g \in L_{full}$ do
 5:         $i < -inputs(g)$
 6:         if $i \subseteq I_{trees} \wedge i \nsubseteq O_{trees}$ then
 7:             $I_{trees} \leftarrow I_{trees} \cup outputs(g)$
 8:         else
 9:             $L_{strip} \leftarrow append(L_{strip}, g)$
10:         end if
11:     end for
12:     $C_{strip} \leftarrow append(C_{strip}, L_{strip})$
13: end for
```

are therefore added to the list L_{strip}, which contains the current level of the circuit without any gate from a ranking tree. Finally, the stripped level are concatenated to form a circuit without the ranking trees. This leaves some unconnected wires which are missing their source or destination gate, but which also are no inputs and outputs to circuit. To resolve this a ranking tree gate is added to the circuit after completing the translation to integer gates. Even-though this introduces a few edge cases in the translation process, it was preferable to adding a ranking tree gate to the levelled circuit beforehand as it does not require any modifications to the network generator library. Furthermore, condensing the ranking tree into a single layer, will also produce a number of empty or only sparsely populated levels. This problem is addressed by a rescheduling step after the translation, which will rearrange the integer gates in a valid ASAP schedule again. Scheduling the completed circuit again, also allows adding new gates and levels in the translation process, without having to conserve a valid schedule.

After removing the ranking trees from the Koppelman-Oruç network, both types of networks can be translated to integer gates circuits. A pseudocode version of the translation is shown in algorithm 2. In a first step, the inputs of the levelled circuit are split up into address and data inputs. This is done simply by splitting the input bitvector according to the data and adress width of the circuit, using the $split()$ function in the pseudocode. The wires in the two resulting bitvectors are then stored in two sets I_{data} and I_{addr}. These sets can be used later to check efficiently if a bitvector contains only address, or data wires, or a mixture of both types. Furthermore, the outputs of the now removed ranking tree have to be added I_{addr}, in order for the algorithm to be able keep track of data and address flows. In the next step the gates in all levels have to translated into integer gates. First a new empty circuit C_{int} is created to store the translated gates, then a nested set of loops iterates over all levels and over all gates. For each gate the set of input wires are retrieved in order characterize it as either a data flow gate, an address flow gate or a

Algorithm 2 Splitting address and data flow

```
 1: $I_{data} \leftarrow \emptyset$
 2: $I_{addr} \leftarrow \emptyset$
 3: for all $i \in I$ do
 4:     $i_{data}, i_{addr} \leftarrow split(I_{data})$
 5:     $I_{data} \leftarrow I_{data} \cup i_{data}$
 6:     $I_{addr} \leftarrow I_{addr} \cup i_{addr}$
 7: end for

 8: $C_{int} \leftarrow [\,]$
 9: for all $L \in C_{strip}$ do
10:     $L_int \leftarrow [\,]$
11:     for all $g \in L$ do
12:         $i \leftarrow inputs(g) \setminus control(g)$
13:         if $i \subseteq I_{data}$ then                          ▷ g is an data-only gate
14:             $g_{int} \leftarrow translateDataGate(g)$
15:             $L_{int} \leftarrow append(L_{int}, g_{int})$
16:             $I_{data} \leftarrow I_{data} \cup outputs(g)$
17:         else if $i \subseteq I_{addr}$ then                  ▷ g is an address-only gate
18:             $g_{int} \leftarrow translateAddressGate(g)$
19:             $L_{int} \leftarrow append(L_{int}, g_{int})$
20:             $I_{addr} \leftarrow I_{addr} \cup outputs(g)$
21:         else                                                   ▷ g is a SW gate
22:             $c \leftarrow g.c$
23:             $x_{data}, x_{addr} \leftarrow split(g.x)$
24:             $y_{data}, y_{addr} \leftarrow split(g.y)$
25:             $o_1, o_2 \leftarrow splitOutputs(g)$
26:             $o_{data1}, o_{addr1} \leftarrow split(o_1)$
27:             $o_{data2}, o_{addr2} \leftarrow split(o_2)$
28:             $g_{int1} \leftarrow makeDataSWGate(c, x_{data}, y_{data}, o_{data1}, o_{data2})$
29:             $g_{int2} \leftarrow makeAddressSWGate(c, x_{data}, y_{data}, o_{data1}, o_{data2})$
30:             $L_{int} \leftarrow append(L_{int}, g_{int1})$
31:             $L_{int} \leftarrow append(L_{int}, g_{int2})$
32:         end if
33:     end for
34:     $C_{int} \leftarrow append(C_{int}, L_{int})$
35: end for
```

mixed gate having both data and address wires as its inputs. The control inputs are ignored for this purpose since control inputs of multiplexers and switches are always address signals in both types of circuits considered in this thesis. The three types of gates are handled separately. If a gate is a pure dataflow gate, the output of the gate will also be a data wire or bitvector made up of data wires. Similarly, the second type of gates which has only address wires as input will have only address wires as ouputs. For these two cases the translation procedure is simple: The inputs are and outputs are turned into an integer signal of the correct type, either address or data, and since all gates without a direct integer pendant have been removed from the circuit, the gate can be direct translated into an integer gate. This translation is abstracted by the *translateDataGate* and *translateAddressGate* functions in the pseudocode. The ranks generated by the ranking trees inside in a Koppelman-Oruç network are treated as part of the address logic as well.

In the actual implementation a hashmap mapping lists of wires names to integer signals is used to translate the inputs. As soon as a gate has been translated its output integer signals are added to the hashmap. Maintaining this additional data structure provides the ability to quickly find the integer signal for each bitvector. If an input bitvector can not be found in the hashmap, this can either mean the bitvector has to be translated as a subsignal, containing a smaller portion of the bits in an integer signal, or that for some reason there is no bitvector containing all the required wires. The later can be seen as in indication that inputs circuit was malformed in some way, thus providing an additional sanity.

The third possible type of gate is a mixed gate which has inputs containing both data and address wires. Fortunately, the only gates with this property are switches. As discussed initially, splitting these gates into to separate switches, one for data and one for addresses or ranks, greatly simplifies the code generation in later stages. To split a switch gate, first its inputs have to be split in their data and address portion. Since the network generator function has been modified to not remove the used address bits, this can be done by simply splitting the bitvectors according to the data and address width. As shown in table 2 the output bitvector contains both outputs, which then in turn contain the data along with its destination address. Therefore, the output has to be split after half of its bits into the two outputs of the original gate o_1 and o_2. Then the outputs can be split into addresses and data using the *split* function analogous to the inputs. Using the split inputs x_{data}, y_{data}, x_{addr}, y_{addr} as well as the split outputs o_{data1}, o_{data2}, o_{addr1} and o_{addr2} two seperate switch gates g_{int1} and g_{int2} can be constructed. The switch g_{int1} routes only data, where as g_{int2} takes care of routing the addresses. Both gates are then added to the current level of the integer circuit. Since both new gates use the same control input as the original one for routing, the splitting does not affect the overall behaviour of the circuit, as long as the interconnections between each split gate and its split predecessors are kept the intact.

As discussed previously integers signals used as inputs by the gates in the translated circuit may only use smaller portion of another integer signal. In that case the smaller integer signal maintain a reference to its larger parent signal. The next step is to add SUBSIG gates to the circuit in order to generate these subsignals.

Alogrithm 3 shows how the SUBSIG gates are inserted into the integer circuit C_{int}. It starts creating a new empty circuit C_{subsig} and an empty set S_{done}, which will contain all the subsignals for which a SUBSIG gate has been inserted already. The algorithm then continues to iterate over all inputs of all gates of all levels of the

Algorithm 3 Adding subsignals to an integer circuit

```
1: $C_{subsig} \leftarrow [\,]$
2: $S_{done} \leftarrow \emptyset$
3: for all $L \in C_{int}$ do
4:     $L_{subsig} \leftarrow [\,]$
5:     for all $g \in L$ do
6:         $I < -inputs(g)$
7:         for all $i \in I$ do
8:             if $isSubSignal(i) \wedge (i \notin S_{done})$ then
9:                 $p \leftarrow parentSignal(i)$
10:                $g_{subsig} \leftarrow makeSUBSIGGate(p, i)$
11:                $L_{subsig} \leftarrow append(L_{subsig}, g_{subsig})$
12:                $S_{done} \leftarrow S_{done} \cup \{i\}$
13:            end if
14:        end for
15:    end for
16:    $C_{subsig} \leftarrow append(C_{subsig}, L_{subsig})$
17:    $C_{subsig} \leftarrow append(C_{subsig}, L)$
18: end for
```

circuit. While doing so it checks for each input signal i if it is a subsignal of a larger integer signal and if it is not already an element of C_{subsig}. In case both conditions are met a SUBSIG gate has to be added to generate the subsignal by extracting its bits from its parent signal. The gate can not be added to the same circuit level as the gate requiring the subsignal as input, since this would break the invariant that gates on the same level are independent. It can also not be added to previous level, as the gate outputting its parent signal might be part of that level. Therefore, a new level L_{subsig} is introduced between the current level L and the previous level such that the subsignal can added to it, without breaking the invariant. Finally, both levels are added to the newly created circuit C_{subsig}. The necessary RANKINGTREE gates can be added to integer circuit in a similar way, by creating a separate level for them as soon as all their input signals are available.

After adding both the SUBSIG and the RANKINGTREE gates, the resulting circuit possibly contains a few empty levels and the schedule represented by the levels is no longer necessarily an as-soon-as-possible-schedule. This is caused partly by the removal of the gates making up the ranking tree, but also by adding the intermediate levels for the SUBSIG gates. While the latter could be added without breaking the schedule with little additional effort, the removing the ranking tree while maintaining the existing schedule would add a lot of complexity. Overall it is simpler to reschedule the integer gates once the translation steps are completed, which has the additional advantage that the translation operations do not need to maintain the schedule of the initial levelled circuit.

Alogrithm 4 is a simplistic ASAP scheduler that removes the unnecessary empty

Algorithm 4 Rescheduling an integer circuit

```
1: $G \leftarrow flatten(C_subsig)$
2: $S_{ready} \leftarrow I_{data} \cup I_{addr}$
3: $C_{re} \leftarrow [\,]$
4: while $G \not\subseteq \emptyset$ do
5:     $L \leftarrow [\,]$
6:     $T_{ready} < -\emptyset$
7:     for all $g \in G$ do
8:         $i \leftarrow inputs(g)$
9:         if $i \subseteq S_{ready}$ then
10:            $o \leftarrow outputs(g)$
11:            $T_{ready} \leftarrow T_{ready} \cup o$
12:            $L \leftarrow append(L, g)$
13:            $G \leftarrow G \setminus g$
14:        end if
15:    end for
16:    $S_{ready} \leftarrow S_{ready} \cup T_{ready}$
17:    $C_{re} \leftarrow append(C_{re}, L)$
18: end while
```

levels and puts each gate into a circuit level, as soon as all of it inputs are available. It starts by flattening the entire circuit into an unordered set of gates G and setting up a set of integer signals S_{ready} that will contain the outputs of any gates, which have already scheduled. It then creates a new empty level L, which is subsequently filled with all gates whose inputs are elements of S_{ready}. In the process these gates are also removed from G and their outputs are added to the set T_{ready}. Once there are no more gates left to add, the new level is added to the circuit C_{re} and the elements of T_{ready} are added to S_{ready}. This process is repeated until all gates are part of C_{re} and there are no more gates left in G. Due to the property that switch gates are either directly connected to inputs of the circuit, or to a previous switch, SW gates that are part of the same column in the permutation network, will be put into the same level. Similarly, gates depending on the same input signals directly or on a subsignal of the same signals, will also be scheduled to the same level, which simplifies the following code generation stages, significantly as gates of the same type with similar inputs which are good candidates for vectorization will be scheduled into the same level.

3.3 Vector Operations

The integer gate circuit produced by the translation procedure in the previous section, could be already be translated into OpenCL code. However, the resulting OpenCL code would not make use of any of the inherent parallelism of the circuit. Even if the code was compiled for a hardware platform that has features for vector operations, such as GPUs or CPUs with instruction set extensions such as AVX [10], it would be up to the compiler to infer vector operations from the sequential OpenCL code. In case of a permutation networks compilers like clang, which is used as a frontend compiler for OpenCL by HARP, can only rely on automatic superword vectorization [4], as most other vectorization optimizers depend on the presence of loops in the code. Similarly, if the code is compiled for an FPGA platform, the compiler has to analyse the code in order to find operations which are independent of each other and can be executed in parallel in the synthesized hardware circuit. Experience has shown that providing additional information to the synthesis, such as explicit vector operations in the code, can greatly improve the efficiency of the resulting circuit. This is especially true for circuits generated from OpenCL code, as the synthesis tool will most likely have highly optimized building blocks available, to generate the circuit equivalent of common OpenCL primitives such as arithmetic operations on vectors or the shuffle operation. To make full use of these and to be independent of compiler optimization support, it is favourable to generate OpenCL code, which already contains explicit vector operations.

```
// --------------- Level 0 ---------------
sig_sub_0[a0_1] = SUBSIG(sig_addr_in_0[a0_0, a0_1])
sig_sub_1[a1_1] = SUBSIG(sig_addr_in_1[a1_0, a1_1])
sig_sub_7[a2_1] = SUBSIG(sig_addr_in_2[a2_0, a2_1])

// --------------- Level 1 ---------------
sig_addr_2[w1] = XOR(sig_sub_0[a0_1], sig_sub_1[a1_1])

// --------------- Level 2 ---------------
sig_addr_8[w14] = XOR(sig_sub_7[a2_1], sig_addr_2[w1])

// --------------- Level 3 ---------------
sig_data_9[w15, w16, w17, w18], sig_data_11[w21, w22, w23,
    w24] = SW(
    sig_addr_8[w14],
    sig_data_in_2[x2_0, x2_1, x2_2, x2_3],
    sig_data_in_3[x3_0, x3_1, x3_2, x3_3])

sig_addr_10[w19, w20], sig_addr_12[w25, w26] = SW(
    sig_addr_8[w14],
    sig_addr_in_2[a2_0, a2_1],
    sig_addr_in_3[a3_0, a3_1])

sig_data_3[w2, w3, w4, w5], sig_data_5[w8, w9, w10, w11] =
    SW(sig_sub_0[a0_1],
    sig_data_in_0[x0_0, x0_1, x0_2, x0_3],
    sig_data_in_1[x1_0, x1_1, x1_2, x1_3])

sig_addr_4[w6, w7], sig_addr_6[w12, w13] = SW(
    sig_sub_0[a0_1],
    sig_addr_in_0[a0_0, a0_1],
    sig_addr_in_1[a1_0, a1_1])

// --------------- Level 4 ---------------
sig_sub_13[w13] = SUBSIG(sig_addr_6[w12, w13])
sig_sub_18[w7] = SUBSIG(sig_addr_4[w6, w7])

// --------------- Level 5 ---------------
sig_data_14[w39, w40, w41, w42], sig_data_16[w45, w46, w47
   , w48] = SW(
    sig_sub_13[w13],
    sig_data_5[w8, w9, w10, w11],
    sig_data_11[w21, w22, w23, w24])

sig_addr_15[w43, w44], sig_addr_17[w49, w50] = SW(
    sig_sub_13[w13],
    sig_addr_6[w12, w13],
    sig_addr_12[w25, w26])

sig_data_19[w27, w28, w29, w30], sig_data_21[w33, w34, w35
   , w36] = SW(
    sig_sub_18[w7],
    sig_data_3[w2, w3, w4, w5],
    sig_data_9[w15, w16, w17, w18])

sig_addr_20[w31, w32], sig_addr_22[w37, w38] = SW(
    sig_sub_18[w7],
    sig_addr_4[w6, w7],
    sig_addr_10[w19, w20])
```

Listing 2: Integer circuit for Narasimha's network

Listing 2 shows the first 6 levels of integer circuit for Narasimha's network with four inputs. A two input version of this network was shown in listing 1. Identifiers starting with `sig_` are integer signals. The wires contained in them denoted in square brackets after the identifier. The start of each new level is denoted by a comment, containing the level number.

This example shows that switch gates, which can be vectorized relatively easy using OpenCL's shuffle instruction, make up a large portion of the interconnection network. Therefore, vectorising them is the primary focus of the process presented in the following. The address and ranking logic is only vectorized opportunistically. Moreover, the example also illustrates that due to the scheduling of the levels, switches for data and addresses that belong to the same column of the network, can be found grouped to together in the same level. This mainly due to the fact, that switches of the same column use the same address signals as control inputs, which because of the similar address logic of the network become available at the same level. This fortunate property allows to vectorise the circuit one level at time, greatly simplifying the search procedure for gates which can be merged into one vector operation.

Type	Inputs	Output	Operation
NOT	$(x_0, x_1, ..x_n)$	$(\neg x_0, \neg x_1, ..\neg x_n)$	Elementwise boolean negation
AND	$(x_0, x_1, ..x_n)$, $(y_0, y_1., .y_n)$	$(x_0 \wedge y_0, x_1 \wedge y_1,$ $..x_n \wedge y_n)$	Elementwise boolean conjunction
OR	$(x_0, x_1, ..x_n)$, $(y_0, y_1., .y_n)$	$(x_0 \vee y_0, x_1 \vee y_1,$ $..x_n \vee y_n)$	Elementwise boolean disjunction
XOR	$(x_0, x_1, ..x_n)$, $(y_0, y_1., .y_n)$	$(x_0 \oplus y_0, x_1 \oplus y_1,$ $..x_n \oplus y_n)$	Elementwise exclusive disjunction
MUX	(c) $(x_0, x_1., .x_n)$, $(y_0, y_1., .y_n)$,	$(c?y_0{:}x_0,$ $c?y_1{:}x_1, ..$ $c?y_n{:}x_n)$	2:1 Multiplexer
SWITCH	$(c_0, c_1, ..c_n)$ $(x_0, x_1., .x_n)$,	$(y_0, y_1., .y_n)$,	Switch gate
SUBSIG	$(x_0, x_1..x_n$	$y_0, y_1..y_n$ with $y_i \subseteq x_i$	Extract the bits from elements
RANKINGTREE	$(x_0, x_1...x_n)$	$(r_0, r_1, ...r_n)$	Compute ranks of inputs

Table 4: Overview of vector operation types

In the vector operation representation of the circuit vectors are represented as lists of integer signals. This allows keeping track of which integer signals are grouped into a vector and which wires are part of each integer signal.

An overview over the vector operations generated in this step can be found in table 4. The NOT, AND, OR and XOR operations are direct representations of the booelan operators `!`, `&&` `||` and `^^`, a regular part of the C programming language which is extended on vector data types by OpenCL. Their semantics are discussed in detail at the beginning of section 3.2. When used on vector data types in OpenCL, these operations simply apply their scalar counterparts element-wise on the input vectors. The multiplexer uses single element vector as its control input and two vectors with the same number of elements as it data inputs. Depending on the boolean value of the element in the control vector, either the first or the second input is forwarded to the output vector. SUBSIG operations extract bits from elements of their input vector and rearrange them to fit the bits in the elements of their output vector. The actual masking and shifting operations are generated at the final stage of code generation based on the wires contained inside the integer signal. RANKINGTREE operations take a single vector of addresses and compute their ranks. The SWITCH operation has slightly different semantics then the SWITCH gate of the integer circuit. Instead of interchanging two inputs based on the control signal, the SWITCH operation alternates between different permutations of its single input vector. The SWITCH operation internally stores a list of five tuples (c, i_0, i_1, o_0, o_1) consisting of a reference to an integer signal in the control vector, two references to signals inside the input vector, and two references to signals in the output vector. If the control input c is 0, then the input i_0 is written to o_1 in the output vector and the input i_1 is written to o_0, representing a crossed switch. If the control input is 1 the value from i_0 is written to o_0 and i_1 is forwarded to o_1, representing a switch in straight through configuration. This allows for arbitrary arrangements of the elements inside the input and output vectors, making it easier to merge SWITCH gates into one vector operation. Every vector operation will also work on single element vectors, except for the SWITCH and RANKINGTREE operations, which would not serve a meaningful purpose on single element vectors. This provides a trivial way to directly map single integer gates to vector operations, in case the gates can not be grouped into lager vector operations.

Depending on the type of integer gate different vectorisation strategies are applied. The circuit is traversed level-wise. For each level the gates are grouped by type and handed over to the vectorisation procedure for the particular gate type, along with a list containing all output vectors generated in previous levels. For the first level the list of output vectors only contains the data and address inputs grouped into two vectors. The procedure then tries to vectorise the gates and outputs a list of vector operations, along with a list of new output vectors generated by these operations.

For the unary NOT gates the vectorisation procedure is comparatively simple. First the gate's inputs are matched against the vectors inside the output vector list, in order to group the gates by the vector their inputs belong to. If one of these groups contains a NOT gate for every element of its corresponding vector, the gates are merged into a vector operation. Its output vector is derived from outputs of the gates and the order of elements in the input vector. The resulting vector operations and new output vectors for all groups are collected into lists and will be added to the overall lists of vector operations and output vectors after the vectorisation of other gate types in the same level is completed. The unary SUBSIG gates can be vectorised using a similar strategy with the major difference, that in addition to the input vectors, the bit patterns in the output also need to be considered when grouping the gates for merging. In order to merge a group of SUBSIG gates the bit masking and shifting required to form the outputs, needs to be the same across all gates.

The same principal can also be applied to the binary gates AND, OR, XOR. First the two potential input vectors containing each input integer signal are retrieved. Then the gates are sorted into groups depending on the two tuple of input vectors. The gates in a group are merged, if the group contains one gate for on each pair of input vector elements. If this is not the case, the gates inside the group are translated into separate vector operations on single element vectors. For the merged gates output vectors are constructed using the order given by the input vectors and the outputs of the gates inside the group. It is worth noting that this strategy is rather conservative, as it only works if for the input vectors $(x_0, x_1..x_n)$ and $(y_0, y_1..y_n)$ there is a gate using each pair x_i and y_i as inputs. With some additions it could also work vectors which are only partially used as inputs by a group of gates, or in cases where all pairs are used but the elements of one vector are arranged in the wrong order. However, at the time the vectorisation strategy was implemented it was still unclear, how much impact additional operations to permute vectors or to extract partial vectors would have on the overall resource consumption and performance for the FPGA platform. Hence, a conservative approach, which does not introduce to much additional code and in turn leaves some room for further improvement, was implemented.

Alogrithm 5 shows the vectorisation strategy for integer switch gates in pseudo code. It starts with the list S of all SW gates inside a level, and the list V of all vectors which are outputs of previous levels. As a first step, the SW gates are grouped by their input vector into lists inside the hash map G.

The algorithm assumes that both inputs of a switch are part of the same vector and terminates with a failure if this is not the case. Since switches are going to be implemented using as shuffle instructions, which can only permutes the elements inside of one vector, it is not possible to use it on inputs coming from two different vectors. This should not pose a problem for Koppelman-Oruç network or Narasimha's network, as the values for addresses, data, and ranks can routed through the network as a single large, between the shuffle operations. If for example the data input is represented as a single vector, all switches in one level will be merged into a single SWITCH vector operation, producing a single output vector, which in term facilitates the generation of a single SWITCH operation for the next level. The control inputs do not introduce a similar problem, as they will be generated by logic in previous levels, which can not be vectorized in all cases. This means that control inputs will most likely be found as multiple single element vectors. The OpenCL code generation will need to generate additional code, which constructs the control vector from these single element vectors while computing the shuffle mask.

After all SW gates are grouped by their input vector, there is one group for

Algorithm 5 Vectorising SW gates

```
1: $G \leftarrow HashMap()$
2: for all $s \in S$ do
3:     $x \leftarrow findVector(V_in, s.x)$
4:     $y \leftarrow findVector(V_in, s.y)$
5:     if $x \neq y$ then
6:         abort()
7:     end if
8:     if $x \notin keys(G)$ then
9:         $G[x] \leftarrow [\ ]$
10:    end if
11:    $G[x] \leftarrow append(G[x], s)$
12: end for

13: $V_{out} \leftarrow []$
14: $O_{out} \leftarrow []$
15: for all $i, g \in G$ do
16:     $c \leftarrow [\ ]$
17:     $o \leftarrow [\ ]$
18:     $p \leftarrow [\ ]$
19:     for all $s \in g$ do
20:         $q \leftarrow (s.c, s.x, s.y, s.o_0, s.o_1)$
21:         $p \leftarrow append(p, q)$
22:         $c \leftarrow append(c, s.c)$
23:         $o \leftarrow append(o, s.o_0)$
24:         $o \leftarrow append(o, s.o_1)$
25:     end for
26:     $t \leftarrow SWITCH(c, i, p, o)$
27:     $O_{out} \leftarrow append(O_{out}, t)$
28:     $V_{out} \leftarrow append(V_{out}, o)$
29: end for
```

all gates operating on data, one for all gates operating on addresses and also one group for all gates routing the ranks in case of the Koppelman-Oruç network. The gates inside each group can be merged into a single SWITCH vector operation. To do so all control inputs used in the group are collected in a control vector c, and all outputs are collected in a vector o, both of which are just a list of the integer signals contained in the vectors. The order of the elements inside the output vector is arbitrary, which does not matter while the data is routed inside the network, as long as the interconnection of the switches themselves is kept intact. Since the shuffle instruction is able to generate arbitrary permutations of the input vector, the only point when the order of the input and output elements becomes critical is when the shuffle masks are being generated. The interconnection between the inputs and outputs is stored as a list of five-tuples p which is used later to compute the shuffle mask.

Finally, a SWITCH vector operation is created for each group using the control, input and output vectors as well the list of tuples collected earlier. The operation is the added to the list of vector operations O_{out} and its output vector is appended to the list of output vectors V_{out} which will be added to V once all gates of the current level a vectorised.

The ranking tree gate does not need to be vectorized. As shown in table 3 its inputs and outputs are a list of integer signals and there already a vector. This gate can be directly replaced with the RAKINGTREE vector operation.

Once all levels are vectorized, the resulting stream of vector operations is no longer split into different levels. Instead, it can be regarded as one sequential stream of instructions, similar to an OpenCL program. However, the order of these instructions still resembles the levelled structure of the initial levelled circuit. This due to the fact that all gates inside one level are independent of each other and that therefore the vector operations for this level do also not depend on each other in terms of dataflow. Hence, the vector operations inside each level can be executed in an arbitrary order as long as each one of them is executed after all operations of previous levels and before all operations in the next levels. Since the vectorisation happens level for level, and the vector operations are appended to the overall operation stream after each level is completed, this condition is trivially fulfilled.

3.4 OpenCL Code Generation

The last step in translating a levelled circuit for a interconnection network into OpenCL, is generating the actual OpenCL source code for the stream of vector operations created by the previous vectorisation stage.

First the vectors used by the operations need to be translated in local variables for OpenCL. This can be achieved by simply assigning an identifier and datatype to each vector. The identifier for each vector is stored inside a hashmap using the vector as key, while the datatype can be inferred on the fly using the vector size. OpenCL provides vector types for most of its primitive data types such as integers, floats or doubles, in sizes of 2,4,8 and 16 elements. Fortunately most vectors required for interconnection networks match these sizes, due to symmetries in the networks structure. However, if other sizes are encountered the next larger vector size can be used. Since most operation work on pairs of elements located at the same indices the input vectors, any additional elements can simply be ignored or treated as a don't-care-value. While this may waste some amount memory or registers on a CPU or GPU based platform, the additional elements can in theory easily be optimized out by hardware synthesis tools for FPGA platforms. Therefore, it should not have a significant impact in terms of resource usage.

`Int` is used as the primitive data type for the elements in each the vector, which means that the number of wires in the integer signal making up the vector is ignored. This design choice was primarily motivated by the fact that the computations for the shuffle mask require a signed integer type. It would also have been possible to use unsigned integers types and add the necessary conversions before computing the shuffle masks. However, the main advantages of using an unsigned integer type is the greater range of positive values for the same number of bits. Since the `int` type should be at least 32 bits wide on most modern architectures, it would also be necessary to switch to a smaller data type such as `char` for this optimization to a have a significant impact on the resource consumption. In addition to this, it would introduce the problem that shuffle is not available for all types of vectors on certain platforms, leading to additional type conversions. To keep the OpenCL simple and preserve at least a minimum level of readability for debugging purposes, the `int` type was chosen, which is the commonly supported data type for shuffle on all platforms. Moreover, there is a high likelihood that the hardware synthesis will be able to remove most of the unused bits from the integers during optimization, mostly eliminating their negative resource impact.

Since each vector is directly translated to a new variable, each variable in the resulting OpenCL code will be written by exactly once by a single operation and will retain this value for the whole execution of the kernel. As a result the code is basically in a single static assignment form, which is often used as in intermediate format by compilers as it greatly simplifies the optimization of register and memory usage, or in case of an FPGA synthesis workflow the optimization of the number of required registers and block RAMs in the circuit.

A common problem for code generation across all different types of vector operations are operations on single elements of vector or in case of the control vector for the SWITCH operation a vector composed of elements of other vectors or scalars. The first case often occurs as part of the address logic which generates the control inputs for the switches. In contrast to the integer gate circuit where SUBSIG gates have been introduced to extract wires from integer signals, no explicit operation is used to extract elements from a vector. Instead, it is indicated implicitly by using a single element vector containing only one integer signal, or in case of the SWITCH control input containing multiple integer signals from different vectors. For any vector encountered during code generation, there always three possible cases: The first on is the trivial case that the vector is an output vector of a previous operation. In that case the vector is a key of the identifier hashmap introduced earlier and the identifier assigned to the vector can be retrieved from it.

The second case is a vector containing only a single element, which has to be extracted from an other vector. This case usually occurs when single bits are extracted from elements of the address or rank vector, to compute the control variables for MUX and SWITCH operations. In that case there is no key matching the input vector inside the hashmap, as there is no previous operation which outputs the exact same vector. Instead, the integer signal contained in the input vector can be found inside exactly one of the vectors used as keys for the hashmap. Once the full vector containing the required element is known, the vectors identifier and the elements index inside the vector can be determined. After that an accessor to retrieve the element is generated as OpenCL code and used instead of a plain identifier. For example if the identifier is `vec_42` and the index is 5, the accessor code generated for it is `vec_42[5]`.

The last case is a vector containing elements of different vectors, which only occurs for the control vector of SWITCH operations. Similar to the case discussed previously these vectors also can't be found directly as keys in the hashmap. Instead, the same technique is used to find the identifiers and indices for each element and a new vector is constructed using the accessors to retrieve the elements. For example a vector containing two elements, the second element from `vec_1` and the fifth one from `vec_2` results in the following OpenCL code: `(int2) (vec_1[1], vec_2[4])`. The resulting vector can either be assigned to a temporary variable, or used directly in an operation. Depending on the optimizations done by the hardware synthesis tools, constructing this temporary vector might consume additional resources, but it can't be avoided in all cases especially since the structure of the address logic does not allow to fully vectorise it.

```
int4 x = (int4) (0,0,1,1);
int4 y = (int4) (0,1,0,1);

// z1 = NOT(x)
int4 z1 = !x;      // (-1,-1,0,0)
// z2 = AND(x,y)
int4 z2 = x && y;    // (0,0,0,-1)
```

```
// z3 = OR(x,y)
int4 z3 = x || y;    // (0,-1,-1,-1)
// z4 = XOR(x,y)
int4 z4 = x ^^ y;    // (0,-1,-1,0)
```

Listing 3: Examples for NOT, AND, OR and XOR in OpenCL

After having addressed the issues of accessing vectors and their elements, the vector operations themselves have to be translated into OpenCL code. For the basic boolean logic operations NOT, AND, OR and XOR this process is rather straight forward. The identifier ar accessor for the vectors can be retrieved from the hashmap or generated and the necessary operations can be implemented using the `!`, `&&`, `||` and `^^`. An example for all four operations can be found in listing 3.

The listing also illustrates a difference between the vector and scalar operations in OpenCL. Boolean vector operations in OpenCL will set all bits in the output if the result is true, yielding -1 for integers, whereas the scalar operators will return 1 as specified in the C99 standard. For most cases this behaviour does not pose a problem, as any value other than 0 will be treated as true and only 0 will be interpreted as false. Therefore, boolean logic will still work as expected, but if the value is used in arithmetic operations, extra steps may be necessary to accommodate for the difference in signs between the vector and scalar versions.

To generate the code shown in listing 3 for the logic operations, first of all the data type of the resulting value has to be determined. Since the logic operations operate element-wise on vectors of the same size or scalars, an integer vector of the same size or the scalar `int` type is used. After writing the data type to the source code, the identifier or accesor for the result variable can be written. For the unary negation this is followed by an assignment using the `!` operator and the identifier for the input variable. In the binary case it is followed by an assignment of the corresponding operator applied to the two input vectors.

The code generated for a MUX operation needs to be slightingly more complex. While a 2:1 multiplexer for the inputs x, y and the control input c could be implemented using bitwise logic operations in the form of `(c & x) | (!c & y)`, this approach was deemed problematic, as it works fine for the case where `c` was generated using vector operations, as it will have the either the value 0 (no bit set), or -1 (all bits set), but does not work if c was computed using scalar operations where `c` will have the value 0 or 1.

```
// z = MUX(c,x,y)
int4 z;
if(c) {
  z = y;
}
else {
  z = x;
}
```

Listing 4: Example code for MUX in OpenCL

Listing 4 shows the OpenCL code which is generated instead. Since the control input `c` is a scalar regardless of the size of the other input vectors, an if-statement can be used to assign either the first or the second input to the output variable. This avoids the problem explained previously as the if-statement will tread any non-zero values of `c` as true and only zero as false, which works for both cases. In addition to avoiding a special case for the control variable this code is also more readable than the version using bitwise logic.

The SUBSIG operation extracts bits from the elements of its input vector and rearranges them to match the output vectors elements. In contrast to every other operation the integer signals of the output vectors contain the same wires as the signals of the input vectors. This can be exploited to identify at which position inside the input signals a wire can be found, by simply comparing wire names. Assuming the wire at position a inside an element i of the input vector is found at position b in an element o of the output vector, the value of this wire can be extracted using the bit-wise conjunction with the integer 2^a, which will be represented as `(1 << a)` in OpenCL to improve readability. The next step is to shift the extracted bit to its target position b. While this could be achieved using a single shift instruction with the absolute value of the difference between the two positions as shift amount and its sign to indicate the direction, the code generation becomes simpler if it is split into two operations. First the bit is shifted right from its position a to position 0, from which it can then shifted left to its destination position b: `o = ((i & (1 << a)) >> a) << b`. This makes it easier to find the OpenCL code for specific SUBSIG operation during debugging, while the performance should not be affected negatively, as the additional shifts should be optimized out by the compiler using constant folding. If the outputs contains multiple wires of the input this process can be repeated for each wire and the results can be combined using the bit wise conjunction: `o = (((i & (1 << a0)) >> a0) << b0) | (((i & (1 << a1)) >> a1) << b1)...`

```
1  int sub_0 = (((addr_in[0] & (1 << 2)) >> 2) << 0);
2
3  int4 sub_4 = (((v_4 & (1 << 1)) >> 1) << 0)
4               | (((v_4 & (1 << 2)) >> 2) << 1);
```

Listing 5: Example code for SUBSIG in OpenCL

A complete example of the resulting OpenCL code can be found listing 5. The first part at line 1 shows how the most significant bit of a three bits wide element of an address vector is extracted and placed at position 0 in the integer `sub_0`. The scheme discussed earlier is used to generate an accessor for the vector element `in_addr[0]`. Similarly, line 3 shows code for a completely vectorized SUBSIG extracting the third and second bits and placing them at positons 0 and 1 inside the elements of the output vector. Unfortunately the case where the same bits are extracted from all elements of a vector, does never occur in the circuits for the two networks considered here, therefore the code generated for SUBSIG operations usually falls back to multiple scalar operations.

The RANKINGTREE operation on the other hand can never occur in the scalar case, as it needs at least two elements in its input vectors to perform a meaningful computation. It is also a special operation as it will be generated already in vectorized form at the integer circuit stage. The output vector of the ranking tree y is basically the prefix sum of the input vector x, which means that the element at index i in the output vector is the sum of all elements 0 to i of the input vector: $y_i = \sum_{j=0}^{i} x_i$. Since the elements of the input vector are either 0 or 1, the last element of the output vector would range between 0 and the number of input elements in case all inputs are 1. This introduces an issue for the hardware circuits, as for example for an 8 input ranking tree the 8th output could reach the value 8 and would therefore require 4 bits, whereas all other elements would fit in 3 bits without problems. Also, the additional bit would only be used for one of the 2^8 possible inputs. To avoid using an extra bit for the output, while still preventing an overflow, the first element can be decremented by 1, reducing the range of the last element to 0 to 7 in this case. Therefore, the output of the ranking tree is a prefix sum over the input vector, with the first element of the input vector decremented by 1.

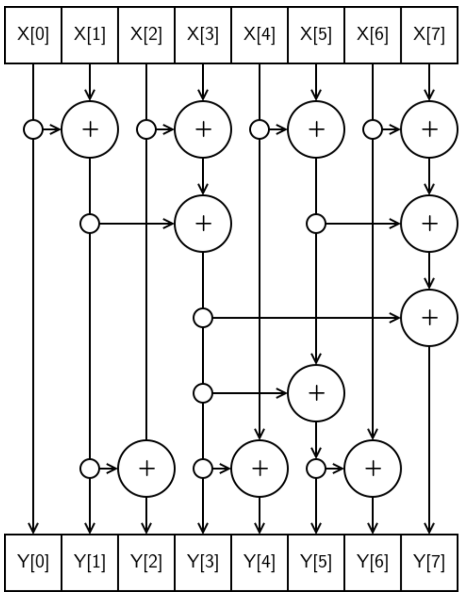

Figure 7: Parallel prefix sum computation

To computed to prefix sum efficiently in OpenCL a parallel algorithm is required, which preferably can be implemented using vector operations. There are two well-known algorithms for parallel prefix sum computation, one by Hillis and Steele [8] and one by Ladner and Fischer [16]. While both have a runtime of $\mathcal{O}(log(n))$, the second one has the advantage of requiring fewer additions for them same input size, which is advantageous for a hardware implementation.

Figure 7 shows the structure of the additions performed by the second prefix sum algorithm for eight inputs. It first adds all the input values pair-wise at odd indices to the values at the even indices, storing the results at the even indices. The next step applies a similar pair-wise addition, but only considers the elements at odd indices from the previous step instead of the whole input vector. This is repeated until there is only one pair left to add. After this step the addition pattern continues with the elements from the step prior to the single addition step and it again adds elements pair-wise, each element the next element, but this time the first and last element are ignored. This pattern ascends going through the sets of elements from the descending phase of the algorithm, until the pattern addition is applied to the entire input vector again.

While this algorithm could be implemented elegantly in a recursive fashion, OpenCL does not allow recursive procedures, and recursion is also a problematic concept when it comes to creating hardware circuits. Instead, a recursive generator procedure is used to produce a sequence of vector additions, performing the same operations as shown in figure 7.

```
//--- start ranking tree ---
int8 rank0 = (int8) (sig_1, sig_3, sig_5, sig_7,
                     sig_9, sig_11, sig_13, sig_15);
rank0 = rank0 + (int8) (-1, 0, 0, 0, 0, 0, 0, 0);
int8 rank0_add_0 = (int8) (0, rank0[0], 0, rank0[2],
                           0, rank0[4], 0, rank0[6]);
int8 rank0_0 = rank0 + rank0_add_0;
int8 rank0_add_1 = (int8) (0, 0, 0, rank0_0[1],
                               0, 0, 0, rank0_0[5]);
int8 rank0_1 = rank0_0 + rank0_add_1;
int8 rank0_add_2 = (int8) (0, 0, 0, 0,
                           0, 0, 0, rank0_1[3]);

int8 rank0_2 = rank0_1 + rank0_add_2;
int8 vec_24_add_1 = (int8) (0, 0, 0, 0,
                            0, rank0_2[3], 0, 0);
int8 vec_24_1 = rank0_2 + vec_24_add_1;
int8 vec_24_add_0 = (int8) (0, 0, vec_24_1[1], 0,
                    vec_24_1[3], 0, vec_24_1[5], 0);
int8 vec_24 = vec_24_1 + vec_24_add_0;
//--- end ranking tree ---
```

Listing 6: Example code for RANKINGTREE in OpenCL

Listing 6 shows the code generated for a ranking tree with 8 inputs. First it decrements the first element of the vector by one, which is unnecessary for in terms of overflow avoidance in OpenCL, but since the RANKINGTREE operation has to compute the same output as the circuit it replaces, it is nevertheless still necessary. After that, the pair-wise additions are applied using the patterns discussed previously. The additions are performed as vector additions, using a summand vector constructed from elements of the rank vector. The vectors are constructed by initializing a new vector using the vector element accessor principal introduced earlier. Theoretically these vectors could also be constructed using shuffle instructions, but the shuffle instruction assigns each element in the output vector an element of the input vector. It is not possible to construct vectors with elements set to zero unless zero is contained in the input vector. This would require additional operations to zero out unwanted elements. In contrast to the that, the approach used in listing 6 can be translated into hardware circuit, which only reroutes certain signals to generated to summand vector without consuming any additional hardware resources.

The final operation to be translated to OpenCL is the switch operation, which is implemented using OpenCL's build in shuffle instruction. The Shuffle instruction takes to parameters, an input vector `v` and a mask vector `m` of the same size as the input vector to compute an output vector `o` which also has the same size as the two inputs. Each element in the output vector has a corresponding mask vector element at the same position in the mask vector. The output vector is generated by assigning each output element an element from the input vector, using the output elements corresponding mask element as an index to the input vector: `o[i] = v[m[i]]`. This allows generating arbitrary permutations of the elements in the input vector.

The SW gate in the initial levelled circuit either forwards its two inputs directly to its two output or swaps them depending on its control input. Since the SWITCH vector operation is only a number of SW gates, operating on data inside the same vector, each element of the operations input vector will be forwarded to one of two possible locations in the output vector. Conversely, each element of the output vector will be assigned one of two elements of the input vector, depending on the control vector element for the output. Additionally, due to the SW gates swapping pairs of inputs, there will be pairs of outputs depending on the same element of the control vector, where the two possible input elements for each output are the inverse of each other. For example a one output of the pair can be assigned the input vector element at index 4 if the control input element is set and the input vector element at index 5. The other output of the pair will be assigned input vector element from index 5 if the control input is set and 4 otherwise. Hence, each entry in the shuffle mask can have two different values, and there are pairs of elements in the shuffle mask, which will swap their values depending on the value of one control vector element.

These pairs of input and output elements are represented by the five-tuples stored within the SWITCH vector operation. Each tuple contains the integer signal for the control input, the two data inputs and the two data outputs. To generate the code that compute a shuffle mask, it is beneficial to reorganize these tuples to obtain the control input signal and both data input signals for each output. The signals can be mapped to indices in the control, input and output vector of the SWITCH operation by looking up the position of the signal inside the vector. Using the indices, the shuffle mask can be constructed. Looking at a single output at index o controlled by the signal c_o in the control vector with the two inputs at indices x_o and y_o the corresponding element in the shuffle mask vector can be computed as shown in equation (1).

$$\begin{aligned} m_o &= c_o \cdot y_o + (1 - c_o) \cdot x_o \\ &= c_o \cdot y_o + x_o - c_o \cdot x_o \\ &= x_o - c_o \cdot (y_o - x_o) \end{aligned} \tag{1}$$

This works by using c_o as decision variable which is allowed to have either the value 0 or 1, causing a problem if c is generated using logic operations on vectors. As discussed previously the boolean logic operators for vectors produce the values 0 and -1 instead. Fortunately these two cases can be harmonized by performing a bit-wise conjunction with the value 1, on the entire control vector which will result in 1 for -1 as well as 1 and 0 for 0.

Constructing two constant vectors $\vec{x}$ and $\vec{y}$ containing the two input indices for each output allows to compute the shuffle mask using vector arithmetic as shown in equation (2).

$$\vec{m} = \vec{i_0} + \vec{c} \cdot (\vec{y} - \vec{x}) \tag{2}$$

The control vector $\vec{c}$ can be created either by applying shuffle on an existing vector, containing all the necessary control signals or it has to be constructed from single signals.

```
//--- start switch ---
int8 switchCrtlVec0 = (int8)(sig_22, sig_15,
                              sig_13, sig_0, 0, 0, 0, 0);
switchCrtlVec0 = switchCrtlVec0 & 1;
int8 shuffleInputVec0 = (int8)data_input;
int8 preShuffleVec0 = (int8)(0, 0, 1, 1,
                              2, 2, 3, 3);
int8 shuffleCtrlVec0 = shuffle(switchCrtlVec0,
                         convert_uint8(preShuffleVec0));
int8 shuffleOffsetVec0 = (int8) (6, 7, 4, 5,
                              2, 3, 0, 1);
int8 shuffleCoeffVec0 = (int8) (1, -1, 1, -1,
                              1, -1, 1, -1);
int8 shuffleVec0 = shuffleOffsetVec0 +
                shuffleCoeffVec0 * shuffleCtrlVec0;
int8 vec_1 = shuffle(shuffleInputVec0,
                         convert_uint8(shuffleVec0));
```

```
18 //--- end  switch ---
```

Listing 7: Example code for SWICTH in OpenCL

Listing 7 shows the OpenCl code generated for a SWITCH operation used in a Koppelman-Oruç interconnection network. The control vector is constructed from single signals on line 2. Line 4 limits its values to either 1 or 0. Line 8 uses a first shuffle operation to duplicate each element of the control vector to the correct pair locations for the shuffle mask. It is used in line 14 to compute to the final shuffle mask using the principle demonstrated in equation (2). The subtraction $\vec{y} - \vec{x}$ is already performed during code generation and its result is stored in the vector `shuffleCoeffVec0`. Finally, in line 16 shuffle is applied to actual data vector and the result is assigned to `vec_1`.

```
int4 data_output = (int4)(vec_2[2], vec_2[3],
                            vec_2[0], vec_2[1]);
int4 address_output = (int4)(vec_7[2], vec_7[3],
                               vec_7[0], vec_7[1]);
```

Listing 8: Example code for rearranging network output vectors

As the last step of code generation, the elements in the output vectors of the network might have to rearranged, as the vectorization for SW gates does not ensure a particular order of the elements in the output vector. This can be resolved by comparing the output vector of the last shuffle operation for addresses and data with the output vectors of the network and rearranging the elements if required. Listing 8 show how the same technique which was used to construct the control vector for the switch operation is applied to ensure that the four outputs of an instance of the Koppelman-Oruç network are in the expected order.

A complete OpenCL example code for Narasimha's network with 4 inputs, which demonstrates how the translated operations interact with each other can be found in listing 11 in the Appendix to this thesis.

4 Evaluation

The first step in evaluating the networks, was to ensure that the generated code works correctly, when compiled and executed using the regular OpenCL runtime. To that end a simple host program has been developed in C++. The program uses the OpenCL SDK provided by Intel for their x86 CPUs, and should run on any recent consumer computer without any special hardware. It loads the OpenCL kernel for the network, compiles it into x86 machine code and executes it for a number of test inputs. For networks with up to 8 inputs all of the up to 14833 different permutations for the inputs are tested exhaustively, ensuring that there are no edge-case with incorrect behaviour. On recent hardware the exhaustive tests only take couple of minutes. In contrast to that the networks with 16 inputs can not be tested exhaustively, as testing all the 7697064251745 possible input permutations would take about 100 years at the same rate. Instead, only a few thousand randomly chosen inputs have been tested for the 16-input networks.

The test software first generates one address-data-pairs for each input of the network. While the addresses are simply enumerated, the data portion of each pair derived from the address in a way that the data and the address form two disjoint ranges of integers. This is done to check that the splitting of the address and data flow during network generation has been done correctly and no bits of the addresses are routed as data an vice versa. The list of data is then permuted and copy into a memory region, where it can be accessed by the OpenCL kernel containing the network. Once the execution of the kernel completes, its outputs are checked. A first checks validates that the addresses in the output pairs are in ascending order at the correct indices. After that a second check ensures that pairing of data and addresses still matches the input, making sure that the data arrived at the correct destination along with its destination address. These initial tests have been passed for all input sizes and by both network types.

As a next step towards evaluating the kernels on the actual HARP hardware platform, the kernels had to be compiled for simulation using the Intel OpenCL SDK for FPGA [9]. Compilation for simulation is a two stage process. First of all a host program is required, which similar to when the kernel is executed on the real hardware, loads the kernel, provides it with its inputs, starts the kernel and then retrieves the results. Since the OpenCL SDK for FPGA mostly conforms to the standard OpenCL API on the host-side, the test program previously written for the tests with the regular OpenCL runtime could be reused. The only major difference to the workflow for regular OpenCL is that instead of loading the kernels source code and compiling it, the host program has to load a precompiled binary of the kernel. For the simulation this binary is a shared library containing an executable model of kernel, whereas for the actual execution on FPGA the binary would be the FPGA's bitstream along with additional metadata. The precompiled binary is created using the aoc compiler for OpenCL, which internally uses the Quartus Pro FPGA synthesis toolchain to create a model for simulation, as well as for creating the FPGA bitstream.

Unfortunately the simulations the output failed for any network with more than four inputs. In all failed simulations consisted of four correctly routed address-data-pairs, whereas the rest of the output was filled with duplicates of these four pairs. Networks with four or fewer inputs simulated correctly for all inputs. Closer investigation revealed that the shuffle instruction for vectors with 8 or more elements did not behave correctly in simulation. For example for the input `int8 i = (int8)(0, 10, 20, 30, 40, 50, 60, 70)` and the shuffle mask `int8 m = (int8)(5, 6, 3, 1, 2, 0, 7, 4)` yields `(int) (10, 20, 30, 10, 20, 0, 30, 0)`. On the other hand there also shuffle mask which work without problems. For example: `int8 m = (int8)(5, 6, 3, 1, 2, 0, 7, 4)`. The failed example, suggests that the elements in the shuffle mask are truncated to the two least significant bits which are then used as an index into the input vector, since 4 and 0 both map to first element of the input vector whereas 7 maps to the third and so on. However, if this was the case then shuffle should always fail for all shuffle masks containing elements larger than three. In any case this behaviour suggests a bug in either the simulation environment or the aoc compiler.

Due to this compiler bug and time spent to investigate it, the kernels for the networks could not be tested on the actual hardware. It was still possible to run the full synthesis workflow on all kernels and the synthesis reports at least contain some basic figures regarding the resource consumption of the kernels on the FPGA. These values are estimates based on the register transfer level representation of the kernel, before the placement and routing stage of the synthesis, which usually performs further optimizations.

The following evaluation assumes that the figures obtained from the full synthesis are either not affected by the compiler bug or that they nevertheless still are representative to some extent, if they are affected.

The reports also contain an estimate of the resources consumed by each individual line of the OpenCL source code. This data is especially interesting as it can be used to evaluate the design decisions with respect to rearranging vector elements and the construction of vectors inside the ranking tree.

```
//--- start ranking tree ---
int4 rankInputVec0 = (int4) (sig_addr_1, sig_addr_3,
                  sig_addr_5, sig_addr_7);
//                  ALUTs: 0  FFs: 0  RAMs: 0
rankInputVec0 = rankInputVec0 + (int4) (-1, 0, 0, 0);
//                  ALUTs: 0  FFs: 0  RAMs: 0
int4 rankInputVec0_add_0 = (int4) (0, rankInputVec0[0],
                    0, rankInputVec0[2]);
//                  ALUTs: 0  FFs: 0  RAMs: 0
int4 rankInputVec0_0 = rankInputVec0 + rankInputVec0_add0;
// 32-bit Integer Add (x2)  ALUTs: 66 FFs: 0  RAMs: 0
int4 rankInputVec0_add1 = (int4) (0, 0,
                    0, rankInputVec0_0[1]);
```

```
14 //                 ALUTs: 0  FFs: 0  RAMs: 0
15 int4 rankInputVec0_1 = rankInputVec0_0 +
       rankInputVec0_add1;
16 // 32-bit Integer Add (x1)  ALUTs: 33   FFs: 0  RAMs: 0
17 int4 vec_8_add0 = (int4) (0, 0, rankInputVec0_1[1], 0);
18 //                 ALUTs: 0  FFs: 0  RAMs: 0
19 int4 vec_8 = rankInputVec0_1 + vec_8_add0;
20 //   32-bit Integer Add   ALUTs: 33   FFs: 0  RAMs: 0
21 //--- end ranking tree ---
```

Listing 9: Synthesis report for a 4 input ranking tree

Listing 9 contains the source of a ranking tree with four inputs annotated with estimates from its synthesis report. The comment below each line contains an estimate on how many resources of each type are required to implement the line on the FPGA hardware. The three types of resources are: ALUTs, which are lookup tables used to implement boolean functions, FFs, short for flip-flops, which are used keep store states over multiple clock cycles and RAMs, which are memory cells to store larger amounts of data. Looking at lines 2, 7 and 12, it becomes evident that constructing new vectors from unmodified elements of existing vectors is basically free. This implies the compiler can simply add wires to route the existing data to the operations on the newly created vectors, without having to introduce new registers. Moreover, the additions in line 10 and 15 show that vector operations are optimized element-wise, which means that no adder is generated for the additions with zero. Overall it is likely that the generated hardware for this piece of code is actually a tree-like structure of 32-bit adders.

```
//--- start switch ---
int4 switchCrtlVec1 = (int4)(sig_addr_13, sig_addr_15,
                 0, 0);
//                 ALUTs: 0  FFs: 0  RAMs: 0
int4 shuffleInputVec1 = (int4)data_input;
//                 ALUTs: 0  FFs: 0  RAMs: 0
int4 preShuffleVec1 = (int4)(0, 0, 1, 1);
//                 ALUTs: 0  FFs: 0  RAMs: 0
int4 shuffleCtrlVec1 = shuffle(switchCrtlVec1,
    convert_uint4(preShuffleVec1));
//                 ALUTs: 0  FFs: 0  RAMs: 0
int4 shuffleOffsetVec1 = (int4) (0, 1, 2, 3);
//                 ALUTs: 0  FFs: 0  RAMs: 0
int4 shuffleCoeffVec1 = (int4) (1, -1, 1, -1);
//                 ALUTs: 0  FFs: 0  RAMs: 0
int4 shuffleVec1 = shuffleOffsetVec1
          + shuffleCoeffVec1 * shuffleCtrlVec1;
// 32-bit Int Subtract(x2)  ALUTs: 66   FFs: 0  RAMs: 0
```

```
18 int4 vec_0 = shuffle(shuffleInputVec1,
19                convert_uint4(shuffleVec1));
20 // Extractelement(x4)   ALUTs: 332  FFs: 128 RAMs: 0
21 //--- end   switch ---
```

Listing 10: Synthesis report for 4 input switch

Listing 10 contains the synthesis report for a switch vector operation with 4 inputs. As discussed earlier, it is implemented using two shuffle instructions. Only two lines of this piece of code have resources annotated to them. Line 15 computes the shuffle mask and requires two subtracters. It can be assumed that the subtracters have additional inputs to enable or disable them, which can be used to implement the multiplication with 0 or 1 efficiently. The flip-flops consumed by the shuffle operation in line 18 imply that the shuffle implementation is more complex than just one multiplexer for each output. Additionally, this can indicate that the shuffle operation needs multiple clock cycles to complete, which might become a performance bottleneck for larger networks, with many shuffle instructions. Other than that the switch does not consume any unexpected resources and all of the newly created vectors are either eliminated by constant folding or by forwarding elements of other vectors to the operations similar to the ranking tree discussed above.

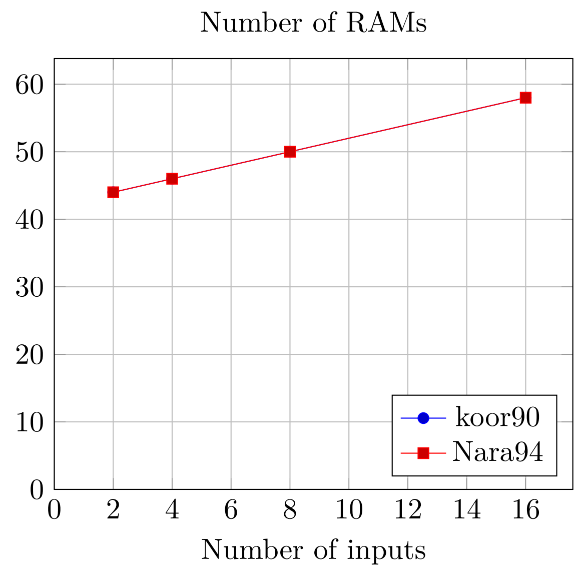

Figure 8: RAM elements consumed by network type and number of inputs

Figure 8 shows the number of RAM cells used by the two kinds of networks over the four different input sizes. RAM cells usually the most scarce hardware blocks on FPGAs, therefore it is in most cases desirable to keep there usage to a minimum. At the same time of a high number of used RAM cells can be good indicator for certain types of design problems. The RAMs cells used for the network kernels are most likely used in the interface to the supporting environment, which connects the kernel to buses and enables it to access the host systems main memory. This hypothesis is further supported by the fact, that the number of required RAM cells increases only slowly and linear with the input size and is the same for both network types. This indicates that the part of the circuit consuming the RAM is actually independent of the number of switches and logic operations.

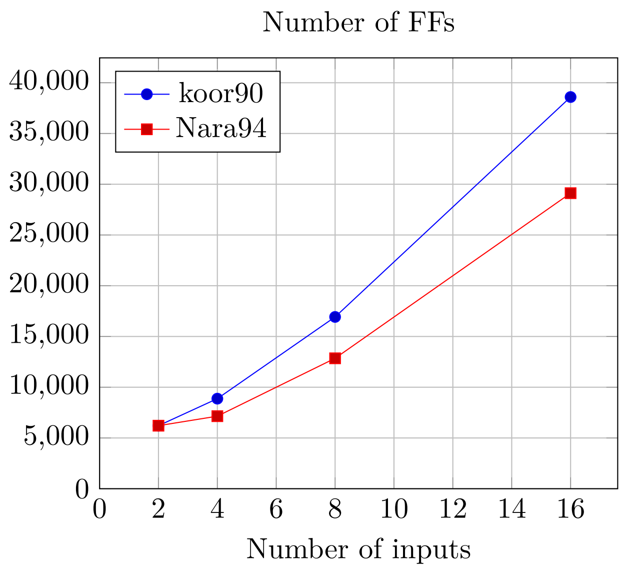

Figure 9: Flip flops consumed by network type and number of inputs

The number of flip-flops used by the kernels is shown in figure 9. As is expected with respect to the network structure and the number of shuffle instructions required, and hence the number of flip-flops consumed grows stronger than linear with the increasing input size. However, it can also be seen that the Koppelman-Oruç network requires more flip-flops than Narasimha's network for all input sizes. This is mainly due to the fact that the Koppelman-Oruç network also routes the ranks through the network, which requires additional shuffle instructions compared to Narasimha's network. In terms of FPGA resources consumed this is a clear advantage of Narasimha's network.

Finally, figure 10 shows the number of lookup tables used to synthesise the kernel. Like the flip-flops the number of ALUTs used increase stronger than linear with the input size. The additional switches required by the Koppelman-Oruç network to route the ranks, and possibly also the ranking trees required to compute the ranks causes a significant overhead over Narasimha's network.

A direct comparison between the different reports can be found in table 5. It shows that the number of ALUTs grows fastest with increasing input size, which means that it will likely become the limiting factor for larger networks. In addition to that even the comparably smaller Narasimha network, in its 16 input configuration, already uses between 15 and 20% of the total ALUTs available depending on the

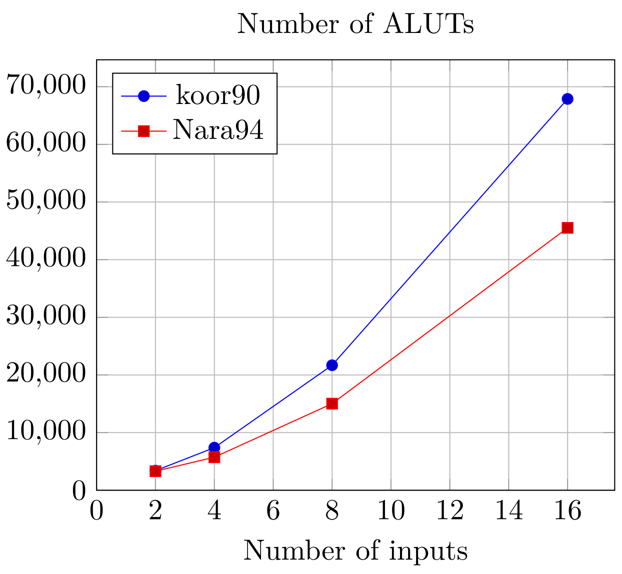

Figure 10: ALUTs consumed by network type and number of inputs

Inputs	ALUTs		Flip Flops		RAM Cells	
	KoOr90	Nara94	KoOr90	Nara94	KoOr90	Nara94
2	3396	3304	6211	6210	44	44
4	7390	5729	8873	7147	46	46
8	21686	15020	16925	12849	50	50
16	67887	45519	38593	29119	58	58

Table 5: Resources consumed by network type and number of inputs

model of FPGA used. As a result, any significantly larger architecture using these interconnection networks will likely be limited in terms of scaling by the amount of FPGA resources that can be devoted to the interconnection network.

5 Future Work

One major area where the network presented in thesis can be improved is the support for networks with more then 16 inputs. One possible approach to this is implementing a replacement for the shuffle instruction with more than 16 inputs. In addition to that, the larger networks will have to work on arrays instead of vectors, as the maximum size of a vector is limited to 16 elements. Depending on the shuffle implementation it might also prove advantageous to use array of vectors to route the data and addresses through the network.

Moreover, the vectorization and OpenCL generation stage of the network generator will have to be modified to handle the larger data types. Replacing the vectors with arrays will introduce new issues when it comes to utilizing parallelism. Vector operations can only be used to limited extent in combination with arrays. While it is possible to cast a part of an array into a vector, this operation can introduce issues with memory alignment and will likely be costly in terms of FPGA resources.

Instead, it might be preferable to use multiple kernels to process different parts of the inputs in parallel. The Intel OpenCL SDK for FPGAs introduces FIFO buffers, the so called channels, as an extension to the OpenCL standard. Channels can be used to pass messages between kernels running in parallel on the FPGA platform. Potentially each kernel could contain multiple rows of switches, subdividing the network horizontally. The channels could then be used to route addresses and data between the different slices of the networks. This approach can potentially avoid the overhead of having to synchronize the memory accesses between the different kernels by minimizing their shared state. The kernels generated by the current network generator could then be utilized as soon as the recursive subnetworks only require 16 or fewer inputs.

An other area where the network generator can be improved are the vectorisation strategies. As shown in section 4 constructing new vectors from the elements of other vectors does not consume additional hardware resources. Similarly, extracting the elements again is also a cheap operation in most cases. This could potentially allow to employ a more aggressive vectorisation strategy for some gate types and also vectorise operations on partial vectors. On the other hand the synthesis reports also showed that vectors are optimized element-wise anyway during synthesis. Therefore, further investigations are required as to how the use of vector operations influences the hardware resources consumed and the performance of the networks. The complexity of the network generator could be greatly reduced if the vectorization could be restricted to the switch gates.

Currently the networks are limited to routing a pair of two integers, representing the destination addresses and data. However, in a real world application there will likely be additional data, such as status flags or tags to group data from different sources for data flow architectures, which need to be delivered along with the actual data. For small design the unused bits inside the address integer could potentially be used to transport additional information, with minimal changes to the existing implementation. Larger systems will likely require to route additional integers through the network. To this end the data routed through the network can easily be extended in the integer gate circuit. At that stage of the network generation process the data flow is completely separated from the addresses logic and easily be duplicated resulting in multiple data integers being routed alongside each address.

A further limitation of the networks presented in this thesis is that they are unable to route partial permutations, which are inputs that do not use every destination address. A generalized approach to enable radix-based networks, such as the two networks used here, to cope with partial permutations was introduced in [14]. Therefore, it should be attempted to extend the network generator to generate OpenCL code for networks containing tenary splitters.

Finally, the networks generated for this thesis have to be tested on the actual HARP hardware. These test will surely result in new insights with respect to the real world performance of the proposed implementation. If possible the networks should also be assessed as part of a larger architecture, for example a SCAD based accelerator, like the one presented in [21].

6 Summary and Conclusion

In this thesis an approach to translating levelled circuits for interconnection networks into OpenCL has been developed. The proposed approach has successfully been implemented for the Koppelman-Oruç network as well as for Narasimha's network, resulting in an OpenCL network generator tool. Using this generator tool the existing hardware descriptions for interconnection networks can be translated in OpenCL code utilizing vector operations to leverage data parallelism. It has been shown that the switches routing data and addresses through the network can be implemented efficiently in OpenCL use the build in shuffle instruction. Furthermore, the generator also demonstrates how the logic gates in the hardware circuits can be replicated efficiently as vector operations in OpenCL code.

Unfortunately, due to the limited size of vectors in OpenCL this approach is only applicable to smaller networks, with up to 16 inputs. However, it still provides a good starting for further developments, as the network generated using this approach can be employed as recursive subnetworks for larger radix-based interconnects implemented in OpenCL.

The tests performed on the generated OpenCL kernels using the standard OpenCL runtime for GPU and CPU computing showed not only that the kernels work correctly, but also that compiling the kernels for other target architectures is feasible. Running the kernels on a CPU and being able to debug it along with the host software using standard tools for software development has greatly sped up the development process. In addition to that it also demonstrates that the kernels code does not use any features outside the OpenCL standard offered by the Intel SDK and is therefore highly portable within the OpenCL ecosystem.

The compiler bug encountered while simulating the networks using the Intel SDK illustrates that the toolchain and the compiler are still a comparatively new technology and will like be subject to improvements in the future.

While evaluating the networks based on the synthesis reports is less meaningful for the actual performance of the networks, it still provided some insights and directions for further investigation. First of all the data obtained from the reports seems to confirm the assumption that vector elements can statically rearranged and that new vectors can be constructed from existing variables with out consuming additional resources. This is an advantage of the FPGA based platform in general, as these operations can be implemented by simply adding connections in the circuit instead of having to copy values in memory. Additionally, it can also be seen that the compiler can identify shuffle instructions with a static shuffle masks and optimizes them in a similar fashion. This encourages experimentation with a more aggressive vectorisation strategies, which might improve the network's performance.

On the other hand the reports also indicate that vector operations are optimized element-wise, raising the question if vectorising the circuit gates has any impact on performance at all. This subject definitely warrants further investigations as the process of generating the OpenCL kernel could be simplified tremendously, if vectorisation was only required for the shuffle instructions.

Overall both types of networks will need additional optimisations, if they were to be used as a part of a larger system. Even Narasimha's network, being the comparatively smaller network of the two, can use up to 20% of the lookup tables available on the FPGA. Implementations supporting a larger number of inputs will likely scale worse than the implementations presented here as they can no longer rely on the shuffle instruction to route data through the network. Therefore, the interconnection network might quickly become the limiting factor in larger designs.

In contrast to that, the usage of RAM cells and flip-flop by the network is rather unproblematic. The RAM cells used are likely the bare minimum required for the kernel to exchange data with the OpenCL runtime and only a small number flip-flops is consumed for each shuffle operation.

Comparing the two types interconnection networks directly, the Koppelman-Oruç network has clear disadvantage. The ranking tree and the switches required to route the ranks alongside the data and addresses consume a significant amount of additional resources over Narasimha's network. This also causes Narasimha's network to scale significantly better with an increasing number of inputs. While it might be possible to reduce the resource consumption of the Koppelman-Oruç network, by combining the addresses and ranks into the same integer and therefore removing one third of the shuffle instructions, the ranking trees would still introduce an additional overhead compared to Narasimha's network.

In conclusion: It is possible to generate interconnect in OpenCL utilizing the shuffle instruction to route data and while the networks presented in this thesis have not been tested on the actual HARP hardware, other tests indicate that they would work correctly. However, the implementations presented here should be considered prototypes. While there is still a large potential for optimisation and experimentation in terms of resource usage, both networks could potentially be used in a smaller SCAD machine implementation. Choosing from the two implementations presented here it would be favourable to use Narasimha's network, as it does not only consume fewer resources, but also was shown to scale better. Whether the networks for larger numbers of inputs can be implemented in a scalable fashion remains an open question.

References

[1] Kenneth E Batcher. “Sorting networks and their applications”. In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference.* ACM. 1968, pp. 307–314.

[2] Václad E Beneš. “Optimal rearrangeable multistage connecting networks”. In: *Bell Labs Technical Journal* 43.4 (1964), pp. 1641–1656.

[3] A. Bhagyanath, T. Jain, and K. Schneider. *Towards Code Generation for the Synchronous Control Asynchronous Dataflow (SCAD) Architectures.* presentation at MBMV 2016. 2016.

[4] Kuan-Hsu Chen, Bor-Yeh Shen, and Wuu Yang. “An automatic superword vectorization in LLVM”. In: *16th Workshop on Compiler Techniques for High-Performance and Embedded Computing.* 2010, pp. 19–27.

[5] Charles Clos. “A Study of Non-Blocking Switching Networks”. In: *Bell Labs Technical Journal* 32.2 (1953), pp. 406–424.

[6] Tomasz S Czajkowski et al. “From OpenCL to high-performance hardware on FPGAs”. In: *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on.* IEEE. 2012, pp. 531–534.

[7] *Hardware Accelerator Research Program | Intel® Software.* June 2018. URL: `https://software.intel.com/en-us/hardware-accelerator-research-program`.

[8] W Daniel Hillis and Guy L Steele Jr. “Data parallel algorithms”. In: *Communications of the ACM* 29.12 (1986), pp. 1170–1183.

[9] *Intel FPGA SDK for OpenCL - Overview.* June 2018. URL: `https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html`.

[10] *Introduction to Intel® Advanced Vector Extensions | Intel® Software.* June 2011. URL: `https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions`.

[11] ISO. *ISO C Standard 1999.* Tech. rep. ISO/IEC 9899:1999 draft. 1999. URL: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[12] Tripti Jain, Klaus Schneider, and Anoop Bhagyanath. “The selector-tree network: A new self-routing and non-blocking interconnection network”. In: *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2016 11th International Symposium on.* IEEE. 2016, pp. 1–8.

[13] Tripti Jain, Klaus Schneider, and Ankesh Jain. “An efficient self-routing and non-blocking interconnection network on chip”. In: *Proceedings of the 10th International Workshop on Network on Chip Architectures.* ACM. 2017, p. 4.

[14] T. Jain and K. Schneider. “Routing Partial Permutations in General Interconnection Networks based on Radix Sorting”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Ed. by O. Bringmann and A. von Bernuth. ISBN 978-3-00-059317-8. Tübingen, Germany, 2018.

[15] David M Koppelman and A Yavuz Oruç. “A self-routing permutation network”. In: *Journal of Parallel and Distributed Computing* 10.2 (1990), pp. 140–151.

[16] Richard E Ladner and Michael J Fischer. “Parallel prefix computation”. In: *Journal of the ACM (JACM)* 27.4 (1980), pp. 831–838.

[17] Pease MC III. “The indirect binary n-cube microprocessor array”. In: *IEEE Transactions on Computers* 5 (1977), pp. 458–473.

[18] Madihally J Narasimha. “A recursive concentrator structure with applications to self-routing switching networks”. In: *IEEE transactions on communications* 42.234 (1994), pp. 896–898.

[19] *OpenCL Overview - The Khronos Group Inc.* June 2018. URL: `https://www.khronos.org/opencl/`.

[20] Andrew Putnam et al. “A reconfigurable fabric for accelerating large-scale datacenter services”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 13–24.

[21] J. Roob. “OpenCL Implementation of Exposed Data Path Architectures as General Purpose Accelerators”. Master. MA thesis. Department of Computer Science, University of Kaiserslautern, Germany, Dec. 2017.

[22] Deshanand Singh. “Implementing fpga design with the opencl standard”. In: *Altera whitepaper* (2011).

List of Figures

Listings

List of Algorithms

Appendix

```
__kernel void network(__global int4* restrict addresses,
    __global int4* restrict data) {
    int4 address_input = addresses[0];
    int4 data_input = data[0];

    int sig_sub_0 = (((address_input[0] & (1 << 1)) >> 1)
        << 0);
    int sig_sub_1 = (((address_input[1] & (1 << 1)) >> 1)
        << 0);
    int sig_sub_7 = (((address_input[2] & (1 << 1)) >> 1)
        << 0);
    int sig_addr_2 = sig_sub_0 ^^ sig_sub_1;
    int sig_addr_8 = sig_sub_7 ^^ sig_addr_2;
    //--- start switch ---
    int4 switchCrtlVec0 = (int4)(sig_addr_8, sig_sub_0, 0,
         0);
    int4 shuffleInputVec0 = (int4)data_input;
    int4 preShuffleVec0 = (int4)(0, 0, 1, 1);
    int4 shuffleCtrlVec0 = shuffle(switchCrtlVec0,
        convert_uint4(preShuffleVec0));
    int4 shuffleOffsetVec0 = (int4) (2, 3, 0, 1);
    int4 shuffleCoeffVec0 = (int4) (1, -1, 1, -1);
    int4 shuffleVec0 = shuffleOffsetVec0 +
        shuffleCoeffVec0 * shuffleCtrlVec0;
    int4 vec_1 = shuffle(shuffleInputVec0, convert_uint4(
        shuffleVec0));
    //--- end   switch ---
    //--- start switch ---
    int4 switchCrtlVec1 = (int4)(sig_addr_8, sig_sub_0, 0,
         0);
    int4 shuffleInputVec1 = (int4)address_input;
    int4 preShuffleVec1 = (int4)(0, 0, 1, 1);
    int4 shuffleCtrlVec1 = shuffle(switchCrtlVec1,
        convert_uint4(preShuffleVec1));
    int4 shuffleOffsetVec1 = (int4) (2, 3, 0, 1);
    int4 shuffleCoeffVec1 = (int4) (1, -1, 1, -1);
    int4 shuffleVec1 = shuffleOffsetVec1 +
        shuffleCoeffVec1 * shuffleCtrlVec1;
    int4 vec_3 = shuffle(shuffleInputVec1, convert_uint4(
        shuffleVec1));
    //--- end   switch ---
    int sig_sub_13 = (((vec_3[3] & (1 << 1)) >> 1) << 0);
    int sig_sub_18 = (((vec_3[2] & (1 << 1)) >> 1) << 0);
    //--- start switch ---
    int4 switchCrtlVec2 = (int4)(sig_sub_13, sig_sub_18,
       0, 0);
    int4 shuffleInputVec2 = (int4)vec_1;
    int4 preShuffleVec2 = (int4)(0, 0, 1, 1);
    int4 shuffleCtrlVec2 = shuffle(switchCrtlVec2,
       convert_uint4(preShuffleVec2));
    int4 shuffleOffsetVec2 = (int4) (3, 1, 2, 0);
    int4 shuffleCoeffVec2 = (int4) (-2, 2, -2, 2);
    int4 shuffleVec2 = shuffleOffsetVec2 +
       shuffleCoeffVec2 * shuffleCtrlVec2;
    int4 vec_0 = shuffle(shuffleInputVec2, convert_uint4(
       shuffleVec2));
    //--- end   switch ---
    //--- start switch ---
    int4 switchCrtlVec3 = (int4)(sig_sub_13, sig_sub_18,
       0, 0);
    int4 shuffleInputVec3 = (int4)vec_3;
    int4 preShuffleVec3 = (int4)(0, 0, 1, 1);
    int4 shuffleCtrlVec3 = shuffle(switchCrtlVec3,
       convert_uint4(preShuffleVec3));
    int4 shuffleOffsetVec3 = (int4) (3, 1, 2, 0);
    int4 shuffleCoeffVec3 = (int4) (-2, 2, -2, 2);
    int4 shuffleVec3 = shuffleOffsetVec3 +
       shuffleCoeffVec3 * shuffleCtrlVec3;
    int4 vec_4 = shuffle(shuffleInputVec3, convert_uint4(
       shuffleVec3));
    //--- end   switch ---
    int sig_sub_23 = (((vec_4[2] & (1 << 0)) >> 0) << 0);
    int sig_sub_24 = (((vec_4[3] & (1 << 0)) >> 0) << 0);
    //--- start switch ---
    int4 switchCrtlVec4 = (int4)(sig_sub_23, sig_sub_24,
       0, 0);
    int4 shuffleInputVec4 = (int4)vec_0;
    int4 preShuffleVec4 = (int4)(0, 0, 1, 1);
    int4 shuffleCtrlVec4 = shuffle(switchCrtlVec4,
       convert_uint4(preShuffleVec4));
    int4 shuffleOffsetVec4 = (int4) (2, 0, 3, 1);
    int4 shuffleCoeffVec4 = (int4) (-2, 2, -2, 2);
    int4 shuffleVec4 = shuffleOffsetVec4 +
       shuffleCoeffVec4 * shuffleCtrlVec4;
    int4 vec_2 = shuffle(shuffleInputVec4, convert_uint4(
       shuffleVec4));
    //--- end   switch ---
    //--- start switch ---
    int4 switchCrtlVec5 = (int4)(sig_sub_23, sig_sub_24,
       0, 0);
    int4 shuffleInputVec5 = (int4)vec_4;
    int4 preShuffleVec5 = (int4)(0, 0, 1, 1);
    int4 shuffleCtrlVec5 = shuffle(switchCrtlVec5,
       convert_uint4(preShuffleVec5));
    int4 shuffleOffsetVec5 = (int4) (2, 0, 3, 1);
    int4 shuffleCoeffVec5 = (int4) (-2, 2, -2, 2);
    int4 shuffleVec5 = shuffleOffsetVec5 +
       shuffleCoeffVec5 * shuffleCtrlVec5;
    int4 vec_5 = shuffle(shuffleInputVec5, convert_uint4(
       shuffleVec5));
    //--- end   switch ---
    int4 data_output = vec_2;
    int4 address_output = vec_5;


    data[0] = data_output;
    addresses[0] = address_output;
}
```

Listing 11: Full OpenCL code for Narasimha's network with 4 inputs