MaxWare
Dataflow Acceleration Process and Optimisation Book

Version 2021.1

Contents

1 Introduction

The MaxWare product provides a very powerful toolchain to target a wide variety of platforms. These include Maxeler's DFE hardware, the Xilinx Alveo cards and the Amazon AWS EC2 F1 instance. Its IP core generator is able to generate target independent code which can even be used for Application-Specific Integrated Circuits (ASICs). There are many examples showing the successful usage of the MaxWare package to achieve outstanding performance which greatly exceeds the capability of conventional control flow based systems. However, we at Maxeler know that the development of dataflow systems is fundamentally different to the development of control flow designs. This document is intended to bridge the gap between the knowledge available to average software engineers and the knowledge required to become a dataflow expert.

This document does not explain MaxJ fundamentals. For this we recommend the MaxCompiler Dataflow Tutorial, which provides many educational examples to teach the MaxJ syntax and API. Instead we focus on *what* to program and *how* to optimise dataflow designs. The first part of the document describes the fundamental knowledge required to successfully design and implement a dataflow system. The second part is intended to provide in depth knowledge for different components of MaxWare toolchain.

The document starts with an introduction of fundamental hardware concepts in section 2. Section 3 describes the philosophy behind the MaxCompiler toolchain. It explains what a user can expect the compiler to achieve and what he has to do by himself. The process of accelerating an application using MaxWare toolchain is described in section 4. Here we describe the process which we use to perform acceleration projects, which has been continuously refined based on more than a decade of experience. The document is written from the perspective of a High-Performance Computing (HPC) developer, but many parts of it can also be applied to low latency networking applications.

The second part of this document start with section 5 which describes how the low level hardware units can be used efficiently. Afterwards section 6 discusses the most important aspects of the Manager. It also discusses the functionality of Maxeler's memory controller and how to write applications which can be deployed on multiple platforms. Section 7 discusses the pipelining of Dataflow kernels. In the sections 8 and 9 we discuss common methods to reduce area usage or achieve timing closure more successfully. Section 10 describes how the SLiC Application Programming Interface (API) can be used in the most efficient way and we discuss good practices for the integration of dataflow components into existing software applications. In section 11 we discuss datatypes in MaxJ and how one can use MaxWare to help with the move to fixed point numerics. Section 12 will cover some more advanced functions to use in kernel development. Finally section 13 provides a detailed system overview of all current hardware targets.

2 The Reconfigurable Fabric

In most cases the core of the targeted device, which is used to lay out the dataflow graph in space, is an FPGA (Field-Programmable Gate Array). As such we want to first give an introduction into how these devices work and which components they entail.

FPGAs are semiconductor devices able to implement arbitrary hardware circuits while also being reconfigurable. Initially one of the main use cases for FPGAs was the emulation and verification of ASICs. Since FPGAs can implement the same circuits as ASICs it is possible to completely test and verify the design reducing design turnaround times significantly. There is no need to generate a complicated netlist with specific technology - meaning a cheaper and faster design time.

However, the usage of FPGAs was quickly expanded to other use cases, most notable signal processing and more recently HPC. The main motivation for this is the massive parallelism supported by the FPGAs combined with the capability to develop highly customised architectures for the intended use case. Compared to Graphics Processing Units (GPUs) and especially Central Processing Units (CPUs), FPGAs support a higher level of parallelism at an order of magnitude lower clock frequency which results in often higher or at least comparable application level performance with lower energy usage. In comparison to ASICs the performance and energy efficiency are often worse using the same technology node, however, the costs of ASICs production are often prohibitive.

Modern FPGAs often follow a column based architecture as shown in fig. 1. This is the case for the current FPGAs of both major chip vendors, Intel and Xilinx. Within these columns different resources are present which can be grouped into three major classes:

1. freely usable reconfigurable fabric;
2. hardened Intellectual Property (IP) and transceivers;
3. clocking infrastructure.

These resources are explained in more detail in the following subsections.

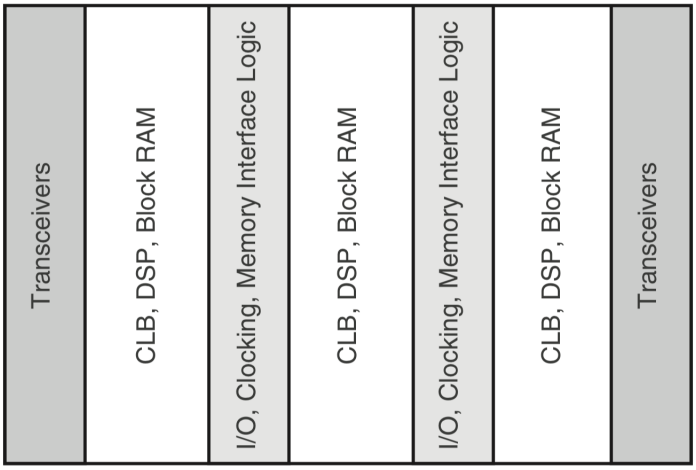

Figure 1: Column based architecture of the Xilinx Ultrascale FPGA family[38].

2.1 Reconfigurable Fabric

The reconfigurable fabric of the FPGA itself consists of three main parts, which can be configured by the user. These are logic resources, memories and Digital Signal Processors (DSPs).

The logic resource consist of Look-Up Tables (LUTs) and Flip-Flops (FFs). A LUT is a structure which is usually implemented as a small memory table. It has multiple inputs and depending on these inputs an output value is created. The input bits form the address for the memory of the LUT and as a result for every combination of input signals a corresponding output will be generated. This allows for the implementation of every possible logic and arithmetic functions. A FF can store a single bit of information based on a clock edge.

The second resource is dedicated on-chip memory arrays. These resources have a limited number of bits and read and write ports to access the data within the memory. However, in most cases they are highly configurable in terms of depth and width. For example the same memory might be used as a 36 bit wide and 512 entries deep or as a 18 bit wide and 1024 entries deep memory.

The last class of resource directly used by the user is the DSPs. These blocks usually consist of hardwired multipliers and additional units e.g. pre- and post-adders. As a result DSPs are less versatile than most other FPGA resources, but enable a significantly more efficient multiplication implementation than achievable when using directly LUTs.

In addition to these resources directly accessible by the programmer there are also routing resources, which are required to connect the different units in the reconfigurable fabric. These are usually hidden from the programmer and implemented using horizontal and vertical connections across the FPGA. At points where the wires cross switches can be used to configure the signal flow.

2.2 Hardened IP and Transceivers

In addition to the configurable parts of the FPGA most modern devices also contain hardened IP cores. These are usually used to interact with high speed interconnects, e.g., double data rate synchronous dynamic random-access (DDR) memory or Peripheral Component Interconnect Express (PCIe) interfaces. The transceivers are used to perform the connection between external signals and the FPGA fabric via the pins of the package. On modern devices these are usually capable of achieving data transfer rates of multiple gigabits per transceiver.

2.3 Clocking Infrastructure

Most FPGAs contain dedicated clocking networks. These are similar to the normal routing networks but are supposed to be used specifically to distribute the FPGA clock across the chip. To complete the clocking infrastructure there are hardware units to derive different clocks from a single base clock and help with the clock distribution across the FPGA.

2.4 Trends in Modern FPGA Architecture

Apart from the fundamental building blocks described above a few recent trends in modern FPGAs deserve special attention. In this section we will specifically focus on changes which were introduced in the last three generations of Xilinx FPGAs.

2.4.1 Super Logic Region (SLR)

In order to increase chip area, while keeping yield and production costs in check Xilinx has introduced devices consisting of multiple dies, called Super Logic Regions (SLRs), in recent FPGA generations. This is achieved by a technology called Stacked Silicon Interconnect (SSI) by Xilinx, where multiple FPGA dies are mounted on a single silicon interposer [33].

As a result the inter-die communication can only be performed using a limited number of slower wires on the silicon interposer. For example the Xilinx VU9P has just more than 20,000 inter-die connections in total. However, SLR crossings are only available between neighbouring SLRs. As a result routing between different and especially non adjacent SLRs is challenging and requires special attention.

2.4.2 Memory Resources

With recent advancements in silicon technology the overall memory capacity of FPGAs has increased massively, leading to multiple heterogeneous memory resources. The Xilinx UltraScale+ FPGAs contain three different physical memory types [41, 39]:

1. Distributed RAM;
2. BlockRAM;
3. UltraRAM.

One logic slice of the Xilinx Ultrascale+ FPGAs contains eight 6-input Look Up Tables (LUTs) that are used to construct a single 512 bits distributed RAM. In the Xilinx documentation this is referred to as SLICEM. Multiple SLICEM can be combined together to form deeper memories, however, this comes with a significant overhead. Individual SLICEMs can be tiled in a multitude of different configurations in terms of depth, width and number of read and write ports.

BlockRAM (BRAM) modules are separate physical hardware memory units. Each BRAM of the UltraScale+ architecture can store up to 36 Kbits of data and can be used as one or two independent memory units. In both cases they consist of two read and two write ports and it is possible to tile them into different depth and width configurations. As an example a 68 bit wide and 850 deep single port logical memory would occupy two BRAM modules with a tiling of 36x1024. The number of tiling options is further increased considering the number of supported read and write port combinations.

Finally, the UltraRAMs (URAMs) represent an additional dedicated memory resource. One URAM module can store up to 288 Kbits of data, but has only one single write and one read port. Additionally, URAMs can be used only as 72 bit wide and 4,096 deep memories and some specific functions, e.g., dual clock FIFO implementations, are not supported. To summarise, URAMs are the least flexible from all available memory types but usually contribute most to the overall on-chip memory capacity of the Xilinx Ultrascale+ devices.

3 Dataflow Programming Approach

Maxeler's MaxCompiler uses a dataflow-oriented paradigm to accelerate large-scale applications. This computational model is an evolution of dataflow and systolic array concepts [10, 20]. In conventional processor-based systems, computations are carried out by performing a sequence of operations on data which must be fetched from and returned to memory each time. In contrast to this, dataflow computing carries out computations by streaming data over a structure of operators. Here, operations do not happen in a time-sequence, but rather in a highly customised deep pipeline of closely-coupled and specialised operators. Developers describe large dataflow pipelines using a high-level language. This results in application-specific, statically interconnected compute structures in the form of dataflow graphs that can be easily optimised and mapped to FPGAs.

As with CPU based programming abstractions there is a trade-off between ease-of-use and achievable performance. While it might be very easy to describe a statistical problem in a language like R, performance conscious developers would probably move to a Fortran or C/C++ implementation of the problem in a HPC environment.

In the world of hardware design the two major concepts for programming languages are hardware description languages such as VHDL or Verilog and High-level Synthesis (HLS). HLS tries to fully automate the generation of hardware circuits from a behavioural algorithm description like, e.g., C/C++ implementation.

Hardware description languages suffer from the required level of expertise as well as from the low level of abstraction. For certain circuits even transistor level description is required. These leads to high development effort. Additionally, often one gets lost in low level optimisation details, while losing the ability to perform high level system optimisations and changes.

On the other hand HLS systems suffer from the inability to fully describe or understand the mapping from algorithm to hardware and an inability to influence low level implementation details. The HLS tool is often not able to implement more advanced architectures and is often limited to kernel acceleration which suffers from Input/Output (I/O) bandwidth restrictions. Additionally, the degrees of freedom available to the HLS tools are simply too high to automatically generate efficient hardware structures. For comparison, on CPU systems the order of loops can have a significant impact on system performance, even though the target architecture is fixed and a lot simpler and the compilers are more mature. How can we expect to produce optimal structures to an architecture with even more freedom?

MaxCompiler attempts to find the sweet spot between those two abstraction levels. The target is to automate as much as possible, without impacting the ability of the designer to control the actual hardware generation. In order to address this balance, usually there is a direct relation between every line of MaxJ and the generated hardware (as can be seen, e.g., in the resource annotation). On the other hand the programmer has to actually design a hardware architecture. That means, that one has to understand hardware design fundamentals to achieve maximum performance. Additionally, we need to understand what we can expect the tool to automate, and what the tool expects the user to do. The remainder of this section will provide a brief overview on this topic. Many of the aspects mentioned here will be discussed in more detail in later sections.

The most important abstraction used in the compiler is the kernel. While the kernel is already introduced in the Dataflow tutorial we will recap its most important abstractions here.

One kernel maps a single dataflow graph onto the Dataflow Engine (DFE) fabric. The kernel uses the abstraction of logical ticks. On every tick one item of data is fully processed, e.g., flows through the complete dataflow graph. To get data from previous or future ticks stream offsets can be used. Naturally, in hardware not all operations can be executed at the same time. Instead the compiler automatically schedules the dataflow graph and inserts FIFO and registers as needed to delay data.

Scheduling is used to assign the nodes of the graph to specific cycles. The purpose of this is, that

the signals are delayed sufficiently to create the described behaviour while considering the latency of the nodes. The latency of the node is determined by the operation it represents, the specific hardware resources and timing constraints targeted. MaxCompiler usually tries to build a pipeline as deep as possible in order to facilitate timing closure at higher frequencies resulting in nodes with high latency.

Fig. 2 shows an unscheduled dataflow graph implementing the function $out = (x+y+z)+(x+y)$. The equivalent scheduled graph is shown in fig. 3. A FIFO is inserted to delay the result of $x + y$ to add to the partial sum of $x + y + z$. Depending on the latency of the addition operation the depth of the FIFO will change. For floating point additions this will usually be in the order of ten cycles, while fixed point additions normally only have a latency of a single cycle. The FIFO is usually implemented in on-chip memory or Flip Flops and the schedule is automatically created and applied by MaxCompiler.

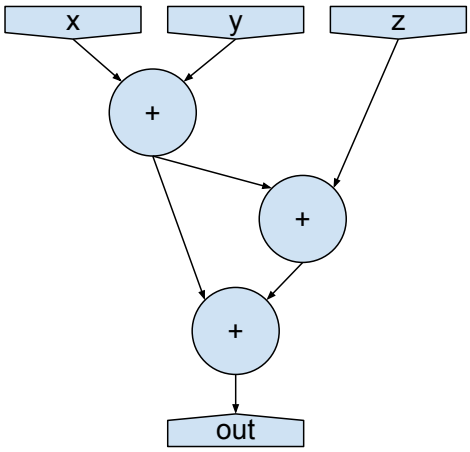

Figure 2: Unscheduled simple dataflow graph.

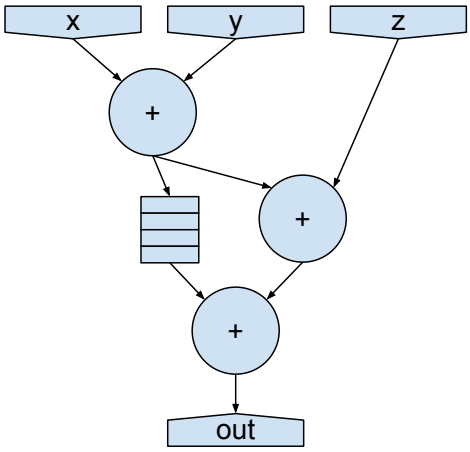

Figure 3: Scheduled simple dataflow graph.

The compiler ensures that the data arrives at the correct cycle to achieve functional correctness, but the user has to handle the potential problems arising from latency. E.g. it is not possible to simply implement an accumulator around a floating point addition using a stream offset with an offset of minus one, since the latency of the floating point addition is more than one cycle.

For the dataflow graph shown in 3 it is by the way not necessary to add a FIFO between the input z and the addition. Instead the input is only activated with the required delay to ensure data integrity.

The tick simplification also allows programmers to instruct the kernel to run for a given number of ticks. This is based on the logical abstraction of ticks and the compiler ensures that the required number of cycles are actually executed. Fill and flush counters keep track of filling and flushing the pipeline to ensure data are only modified during valid execution cycles.

Additionally, the kernel has a stalling mechanism. If data is not available at one of the inputs or if one of the outputs is full the execution of the complete kernel is halted. This ensures local synchronicity and simplifies the complexity of the programming task. On the other hand this also means that if for example, a group of kernels is connected in a loop, the user has to ensure that deadlocks can not occur. E.g. if *Kernel1* needs data from *Kernel2* and *Kernel2* needs data from *Kernel1* a deadlock can easily occur if internal execution latencies are not considered. Manager synchronisation will be discussed in more detail in section 6, and pipelining as well as scheduling is discussed in section 7.

One of the most important topics in the development of every Dataflow system is the selection of suitable datatypes. While MaxCompiler supports the usage of floating point, implementing the dataflow graph using floating point operations is strongly discouraged. The reason for this is the area overhead relative to fixed point arithmetic. For example a floating point addition uses more than ten times the resources than a fixed point addition.

MaxCompiler has a rich support for fixed point types. All non floating point types, including integers, fall under the category of fixed point types. Fixed point arithmetic by default uses the round to nearest even rounding mode. This is the same rounding mode commonly used by IEEE floating point and avoids bias, which would be introduced by more simplified rounding modes like truncate. However, one has to consider that, since this rounding mode also applies to integers, by default the result between integer divisions on CPUs and DFEs differs.

The casting between different types is expected to be managed by the user. This means that performing operations on different fixed point types causes a compilation error. The reason for this is to raise awareness to potential problems. If the compiler is supposed to automatically select a suitable type on some predefined rules *bitgrowth* can be used as described in section 11.3.

Optimisations such as selection of rounding modes, pipelining factor or bitgrowth are configured using push pop methods. This is a fundamental concept of MaxCompiler, which automatically applies a set of rules to a part of a dataflow graph. Most settings which apply to operations of the dataflow graph are configured using these push pop methods and often impact internal compiler functions such as more advanced mathematical functions for example exponent calculation.

The optimisations that the compiler applies automatically are often limited to a single node of the graph and rarely on the complete dataflow graph. For example the compiler heavily optimises single nodes to tailor them to certain datatypes and ranges, but does not implement Common Subexpression Elimination (CSE) elimination. While CSE is a standard optimisation implemented in most contemporary compilers it is not easily applicable to hardware. For example a kernel might entail two identical counters. A CSE pass would fold those two counters into a single operation and as a result reduce the logic usage. However, if the counters are scheduled thousands of cycles apart we would need to schedule the output of the counter accordingly. As a result, the compiler would ”save” tens of LUT at the cost of significant on-chip memory overhead. Similarly, the programmer might duplicate some functions in order to assist timing closure.

To avoid these issues and prevent a constant fight against the toolflow, MaxCompiler uses a very conservative approach when it comes to these kind of optimisations. The two most notable exceptions to this are Constant Folding and Dead Code Elimination. As such it is usually recommended to perform all these optimisations automatically.

Kernels are optimised and intended for the creation of complex computational structures. However, usual Dataflow programs also contain buffers and control logic. The control logic is used to keep track of the current execution status and configure the computational graph accordingly. While it is possible

to use counters, more complex cases often require the usage of kernel state machines.

Buffers in Dataflow programs are not only used to keep data on-chip for fast access, but also for data reordering. In many cases it is necessary to access data in a very specific pattern to fully utilise the parallelism offered by the DFE. For example data might need to be accessed in a strided pattern, within a window of surrounding data or transposed. In all these cases on-chip memory is usually used to transform data between different data formats, as they might be required by different stages of the computational pipeline.

In the experience of the author of this document the time required to implement this control logic and buffering often exceeds the time required to implement the actual compute. As such they have to be considered throughout the design process.

If the abstractions of Kernels are limiting the ability of the programmer to implement the desired architecture, MaxCompiler also offers less restrictive abstraction levels. KernelLites remove the stalling component from normal Kernels. The programmer can define the conditions under which individual parts of the dataflow graph stall. ManagerStateMachines remove all abstractions and allow a direct implementation of logic and arithmetic functions, without any kind of automatic optimisations or transformations. Details on both components can be found in the Networking Tutorial.

The Manager does not contain actual logic. Instead it simply connects different parts of the design, configures them and routes signals between them. While Kernels are fully synchronous the components in the manager are asynchronous to each other. Usually, a DFELink connecting two Manager components is implemented as FIFO. This FIFO is by default 512 elements deep, even though in hardware usually the last few elements do not fill up to accommodate stalling latencies[1].

The Manager provides highly convenient interfaces for the I/O available on the card. As such it is possible to create PCIe or Large Memory (LMEM) interfaces using a single function call. While the compiler offers standard address generators for the LMEM interface it also provides the option to generate fully custom address commands. The reason for this is that while it might be possible to automatically infer addressing patterns from array access code, it would be nearly impossible to ensure that efficient address schemes are used. This would limit the bandwidth obtainable from the LMEM. This is another example for a design decision which follows the fundamental philosophy of always letting the user decide, if an automatic optimal decision is not possible and the added complexity is bearable.

The final component of a dataflow system is the CPU side. While the application calling the DFE is fairly normal, one has to consider the design of the code directly interfacing with the DFE. The main purpose of this code is twofold. It has to configure the DFE for correct execution by, e.g., setting the number of ticks kernels are supposed to run as well as PCIe transfers. Secondly, it is important to transform data that is being sent to or received from the DFE. As described above, computational structures on the DFE often require a very specific pattern to deliver full performance. As such data often requires reordering before it is send to the DFE or once it is received. More details on this topic are discussed in section 10.

In this section we provided an overview of the most important components and design decisions behind MaxCompiler. The intention is to create the correct mindset and expectation to maximise MaxCompiler's potential. The next section will expand on this, by providing a detailed discussion of the design process for Dataflow applications.

[1] This is not fully simulated within the CPU simulator and can be the cause for stalls in hardware. More details in the Debugging tutorial as well as in section 6.

4 The Maxeler Acceleration Process

As described in the last section Maxeler's MaxWare tools do not fully automate the implementation of efficient accelerators but only support the programmer as much as possible. As a result, it is necessary to also provide a methodology on how to use these tools in an efficient manner. In order to achieve it, the different aspects of analysis, architecture, implementation and debugging have to be united in a comprehensive design methodology to program FPGA accelerators using MaxWare rigorously.

This section presents a methodology for the development of complex, DFE based systems. Our methodology results from more than 15 years of experience and has been validated by the deployment of DFE based applications at multiple Tier 1 institutions in different industries such as finance and oil and gas [11, 31, 30, 21, 29].

The aim of the Maxeler dataflow acceleration process is a first-time-right design process, delivering state of the art performance for highly complex applications. Our methodology provides guidelines orthogonal to any algorithm or dataset specific optimisations which steer towards an optimal architecture, maximising the utilisation of all system resources while emphasising additional possibilities for improvements.

It is important to stress that it does not promise the best achievable performance, but it is the best possible, given a set of known optimisations. New optimisations might be discovered or designed and older ones might become obsolete. Nevertheless this methodology is not based on individual optimisations but only provides guidance in the design process. As such no changes to the methodology are needed as the set of considered optimisations changes. While new techniques are being developed an update of the design might still be useful. In those cases the methodology will assist the designers to rapidly evaluate the impact of the new techniques and enable a cost benefit analysis. Similarly, the methodology enables designers to evaluate the system characteristics for different hardware platforms.

Traditional CPU development methodologies usually follow an incremental approach, where a first implementation is gradually improved step by step with the help of different profiling and debugging tools. However, there are multiple problems with applying this methodology to DFE based designs. The main problem is that incremental hardware changes can easily involve a complete redesign of the existing application, which might render nearly all previously undertaken development efforts useless. This is especially the case if the fundamental handling and storage of data is dependent on bandwidth or memory capacity limitations. Additionally, the process to generate a bitstream[2] for an FPGA is extremely time consuming. Combined with the higher complexity of DFE programming this results in prohibitive development times.

It is however possible to avoid these problems altogether. Given that the performance of DFEs is fully predictable, it is possible to save a lot of time and development effort by performing a large part of the design process a priori before even a single line of code is written. The Maxeler acceleration process formalises this procedure and, to the best of our knowledge, is the most efficient way to design an architecture for dataflow systems.

4.1 Methodology Overview

The Maxeler acceleration methodology consists of four main parts:

1. accurate analysis of the targeted application;
2. design of a representative software model;
3. performance modelling of architecture candidates; and

[2] The configuration loaded onto the FPGA

4. rigorous development of an application specific architecture based on the results of the above parts.

Fig. 4 shows how these four parts interact with each other. All individual steps are explained in more detail later in this sections. The methodology is an iterative process, which requires incremental refinement until the final result cannot be further improved. The natural starting point is the analysis of the original application (Part 1). Based on it the user can perform an initial partitioning in hardware and software components (Step a). The parts selected to be moved onto hardware are modelled in software (Part 2). Additionally, an initial architecture is designed (Part 4) which updates the representative performance model (Step d, Part 3). The performance model predicts the achievable performance as well as area and bandwidth requirements. It relies on the analysis of the initial application (Step c), which is further improved by making additional measurements of, e.g., data movements and numerical properties. Both are hard to obtain in the original application, using the software model (Step b). The performance model is used to refine the architecture by identifying bottlenecks, e.g., bandwidth limitations which can be addressed by changes to the architecture (Step e). However, the architectural choices determine what exactly to model, since the performance model is calculated based on a given architecture (Step d). Finally the software model can be used to verify algorithmic and numeric changes in the architecture (Step f).

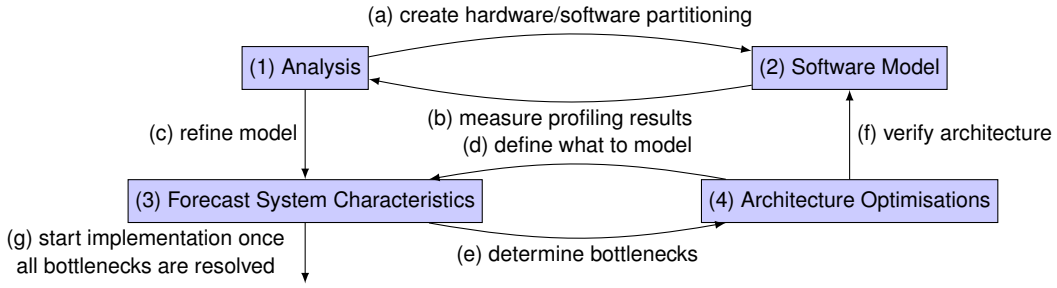

Figure 4: Design methodology breakdown and its four parts and seven steps

Usually all these steps are iterative, where for example a change of the architecture causes changes to the software model and the performance model. Similarly the performance model might highlight that additional measurements of the initial application are necessary. This iterative process is also shown in alg. 1. The programming of the FPGA only starts, once the bottlenecks highlighted by the performance model are resolved and full utilisation of all system resources is achieved (Step g).

Alg. 1: Iterative refinement using the methodology.

```
1  begin
2    perform initial analysis of original application (1)
3    create initial partitioning in hw and sw (a)
4    create first initial software model, performance model and architecture (2, 3, 4)
5    while bottleneck existing
6      identify bottlenecks using performance model (e, 3)
7      resolve bottlenecks with new architecture or hw/sw partitioning (4, a)
8      verify new architecture in updated sw model (f)
9      perform further analysis as necessary (b, c)
10     refine performance model based on updated analysis and architecture (d, c)
11   end
12   start implementation (g)
13 end
```

As a result, cases in which redesigns late in the design process are required will be avoided, since the architecture is based on the performance model, moving the solution to design time. This, combined

with the availability of the MaxCompiler simulator for verification, also removes the need to perform place and route for multiple different design points.

The process is usually only applied to a specific system target at a time. However, Maxeler DFEs are designed following a common pattern. As a result, it is often possible to apply the same architecture and performance model to different DFEs with little changes. This process is explained in more detail in section 6.4.

4.2 Application Analysis

The first step, as depicted in fig. 4, is the analysis of the original CPU application in order to identify its compute and data intensive parts, and achieve better understanding of the challenges which will be faced in the architecture stage. If no CPU implementation is available, it is necessary to create a runtime optimised implementation at this stage in order to enable this analysis. However in most real-world use cases there will be an existing reference application, which has been developed for CPUs, before operational requirements created the need for further acceleration.

In classical general purpose computing the view on the performance characteristics is often limited to analysing instructions and their order of execution as well as the assignment to execution units. This view, however, is not optimal for applications on massively parallel architectures like DFEs. Instead, it is crucial to also understand the required data movements in the system. As a result a strategy where data placement and movement are optimised at all levels of the system has to be developed. Only then is it possible to get the best of the available computing resources, by providing direct access to the working dataset in order to mitigate limited I/O bandwidths.

The target of the control/dataflow analysis is twofold. First this step will help to make an initial assessment of which parts of the application should be ported to DFEs, since in most cases it is not feasible and/or desirable to port the entire application. However, following Gustafson's Law [13] the designer needs to be careful to address a significant part of the execution time, in order to make sure that the code remaining on the CPU will not dominate the runtime. For example, if the part of the application that contributes 90% of the execution time is ported onto the FPGA the maximum theoretical achievable speedup is 10x, due to the remaining CPU parts of the application.

The second target is to collect all the information needed to model the performance of the accelerated application and consequently make correct design decisions. This includes the amount of data that is needed and when is it needed as well as which operations are executed and in what order.

4.2.1 Static and Dynamic Code Analysis

In order to identify the parts of the application that can benefit most from hardware acceleration, various profiling tools like gprof [12] or Intel Advisor [16] can be used. With their assistance, it is possible to annotate the execution time to all functional blocks within the application. These measurements should be made using multiple sets of realistic data in order to approximate the envisioned real world use case as accurately as possible. Erroneous test data selection can invalidate the entire acceleration process, since the envisioned speedup cannot be achieved in the real operational setting.

However, not only the computational complexity, but also the data requirements are essential to decide on an optimal partitioning of the algorithm onto the available computing resources. One should keep in mind that the bandwidth between the different subsystems, e.g, CPUs and FPGAs, is often limited and frequently becomes a major bottleneck. As a result it might be beneficial to also port parts of the code with small contributions to the overall runtime onto the FPGA, in order to reduce the amount of data that needs to be transferred.

An example of this is shown in figure 5. Profiling of the original application has shown, that function 1 has a runtime of 1000 seconds and function 2 1 second. This suggests, that the acceleration function 1 has to be the main target and function 2 can remain on the CPU. However, 10GB of data have to be transfered from function 1 to function 2, which in this example would take 5 seconds via the available interface. If function one can be accelerated on the DFE to only take 5 s the total runtime can be reduced from 1001 to 11 seconds. Now the data transfer is one of the major factors contributing to the overall runtime. Instead, if function 2 is also ported and only the final result is sent back, the runtime can be reduced to 6 seconds, even if no acceleration of function 2 is achieved. This example shows, why the impact of bandwidth limitations is so important.

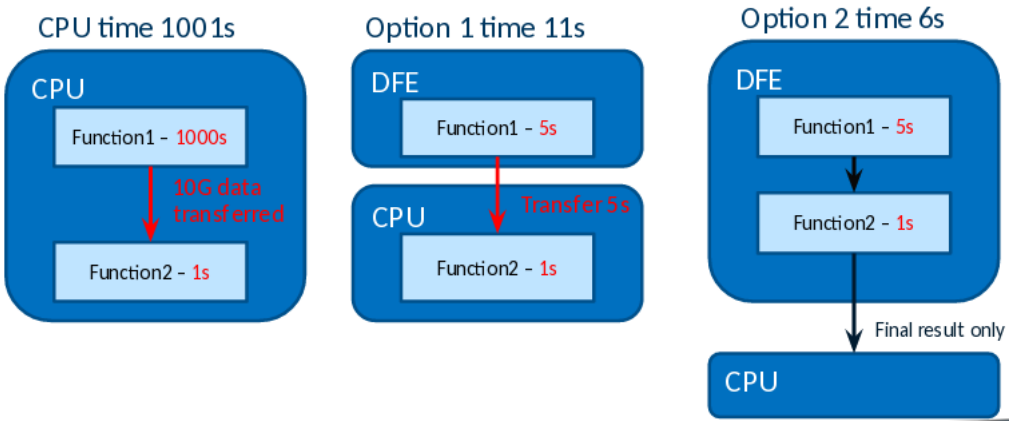

Figure 5: Example visualising the advantage of also moving functions with smaller runtime impact to the DFE.

As such, the size of the data structures and their accessing patterns need to be well understood. In the case of statically allocated memory this might be trivial, but in many cases the size of the allocated data structures depends on the input dataset. However, most profiling tools only have a very limited support to monitor the data structure size and as a result it is often only possible to add debugging statements into the original application. Since those statements might have a significant impact on execution times it is necessary to perform the runtime and data analysis separately. Similarly, access patterns to those data structures should be analysed. Sometime only a tiny fraction of the data is accessed, even though a reference to the complete data structure is provided.

Based on the findings of this analysis it is possible to create the initial partitioning between the CPU and DFE. However, once the performance is modelled as it will be described in section 4.4 it is highly recommended to revisit and reconsider this partitioning. Especially if bandwidth limitations between components of the system exist or individual components are under- or over-utilised.

The parts of the application selected for porting to the DFE require further analysis to provide sufficient information for the later performance modelling stage. This analysis stage can either be performed on the original application or on a software model as described in section 4.3.

Within the parts of the application selected for porting, the operation count for each operation type needs to be obtained. This information is needed in order to estimate the area usage of the circuit which will be implemented (see section 4.4.1 for more details). The process of counting operations can be performed manually or with performance analysis tools.

The second part of this analysis aims at the identification of data movements and memory access patterns. In case the data used on the FPGA cannot be generated on the FPGA itself, it has to be

sent over from the host[3]. For these transfers it is important to determine the amount of data that will be transferred, but also if this data stays constant or changes on regular basis, since in the first case it might be possible to store the data in ROMs while in the second case a regular re-transmission might be necessary depending on the application control flow.

In most real-world use cases it is often not possible to store the entire working dataset in the very high bandwidth on chip memory of the FPGA. As a result even the data which is only generated and used on the FPGA, including temporary results, needs to be analysed. In these cases again not only the size of the data structure but also its access pattern needs to be examined. The access pattern is of special interest, since it has a significant impact on the read and write rates the on board DDR memory can achieve, if the data needs to be stored there. This will be further discussed in section 4.4.4.

4.2.2 Loopflow Graphs

Counting operations for a given set of functions is typically easier than the observation of data movements, which are often harder to analyse, represent and understand. For this reason it is beneficial to create a loopflow graph, which will help to visualise some of the most important results of the previous analysis. Loopflow graphs focus on the loops in the program and how they interact with each other. Since in general most of the execution time is spent in loops this provides a good level of abstraction. Each loop is represent by a rectangle and the number of nested loops is annotated by a factor. Additionally, the operation count can be annotated inside each rectangle.

The dataflow between the loops is depicted using directed arrows, where the width of the arrow hints on the amount of data that needs to be transferred. Consequently, the computational and data requirements can be easily visualised together, which helps to identify the portions of the code which should be moved to the DFE.

The loopflow graph for the VGG-16 Convolutional Neural Network (CNN) [28] in fig. 6, provides a good example on how this graph can help to perform the code split between CPU and DFE. One can easily spot that the operations needed to compute the fully connected layers, depicted in the box at the bottom, are two orders of magnitude less than for the convolutional layers, depicted by the remaining boxes. Additionally, only a tiny amount of data needs to be transferred between the convolutional and the fully connected part of the network. As such, splitting between the convolutional and the fully connected part of the network becomes the obvious choice.

4.3 Software Model

The software model is a simplified software implementation of the parts of the application which are supposed to be ported onto the DFE. It is not intended to be optimised for speed, but instead it should be an accurate representation of the intended DFE implementation. This means that it should be as close as possible to the planned DFE architecture and use the same API. We recommend the usage of C++. It serves three different purposes.

1. Insight into the selected code and easier measurement of profiling results;
2. Testbed for verifying numeric and algorithmic changes; and
3. Debugging reference for DFE implementation.

In order to achieve the first goal of the software model the algorithm can be implemented as a simple, not optimised code, helping with the analysis described in section 4.2.1, and as a result the lessons

[3] The CPU side of the system.

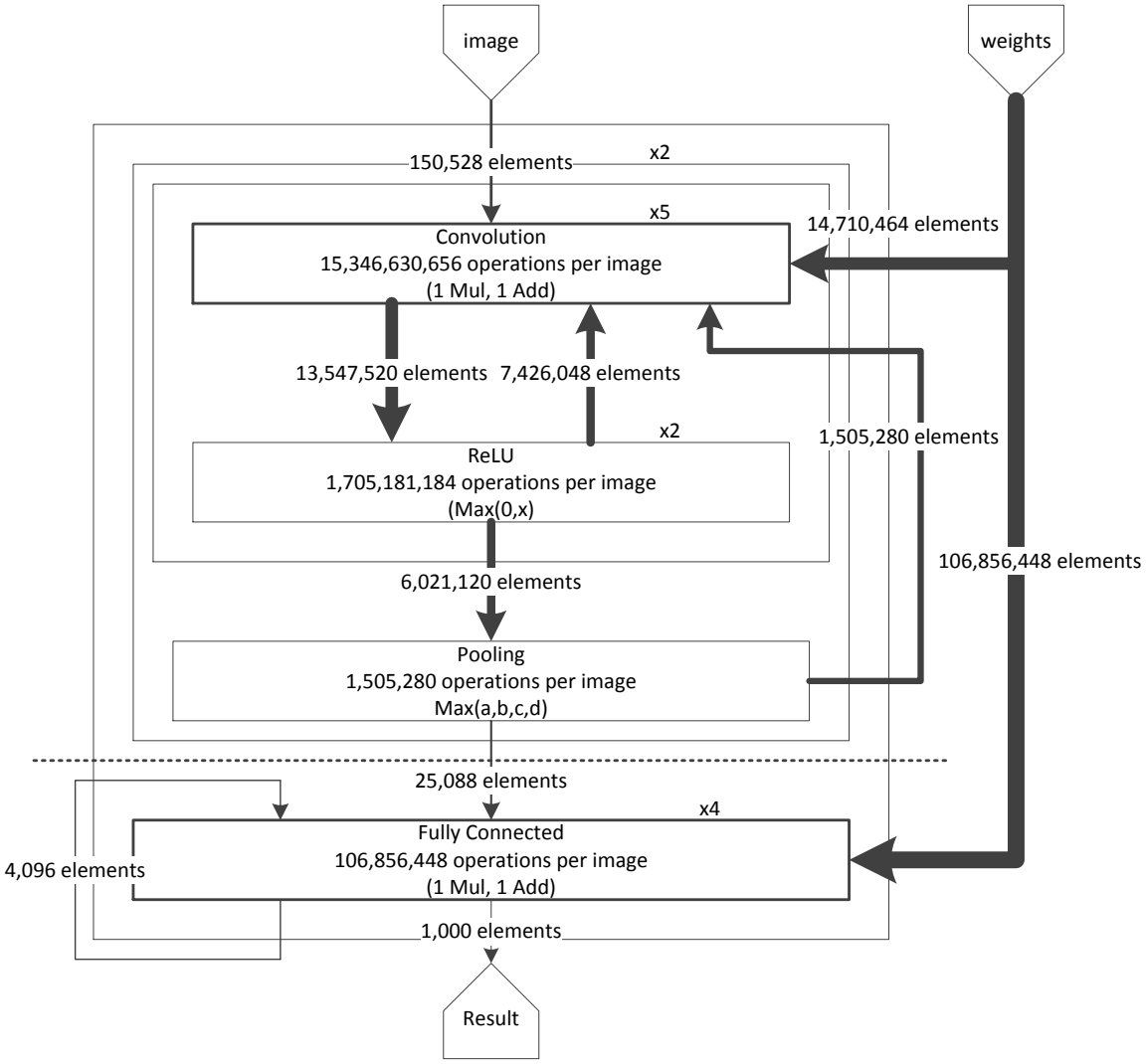

Figure 6: VGG-16 CNN Loopflowgraph. Boxes represent loops and their internal operation types and counts. Arrows show data movements, thickness hints transferred data amounts

learned from this implementation are used to refine the performance model and architecture. Similarly the software model will be used to quickly evaluate different algorithmic and numerical options, which then feed back into the architecture and performance prediction. As a result the software model is typically co-developed with the performance model and the architecture.

It is important to integrate the software model back into the original application. This ensures, that the planned API between the original application and the accelerated modules is sufficient to recreate the original functionality. Additionally, it enables the verification of the software model by comparing the results between the original software and the application using the software model. More details on this are provided in section 10.

Using the same API for the software model as for the DFE implementation also creates an easy to use debugging tool for the DFE implementation. The ability to seamlessly switch between the software model and the DFE application provides a convenient verification method.

The reason to not only verify against the original application is that the software model is supposed to undergo the same numeric and algorithmic changes as the DFE implementation. As such it is easy to check, if an unexpected result for a given input dataset is due to an error in the DFE design, side effects or fundamental problems with the selected algorithm or its numeric properties. Numeric and algorithmic problems are significantly easier to resolve in the software model than in an DFE implementation. Additionally this provides a DFE mock-up implementation for earlier integration into the host application.

While the creation of the software model and especially its continuous improvement during the design process represents an overhead in terms of programming effort it avoids the need to make these design iterations in hardware. Even using MaxJ it is still the case that software development, especially if code is not optimised for performance, is easier to accomplish than DFE development, meaning that it is easier and cheaper to perform these design iterations in software rather than hardware.

4.3.1 Numeric Analysis

One of the most crucial and often also hardest steps in the process of porting an application onto DFEs is the change from floating-point to custom fixed-point arithmetic. The reason for this is that fixed point number formats have a very limited dynamic range, which is fixed at compile time. As such the numeric properties of the algorithm to implement need to be well known, which is especially complicated around operations that have a big impact of the order of magnitude of processed data, e.g., multiplications.

However, in order to maximise the usage of the available hardware resources this is usually necessary. The motivation comes from the fact that for example a floating-point addition needs roughly one order of magnitude more logic resources than a fixed-point addition[34, 36]. The reason for this is that handling of the exponent requires expensive barrel-shifters before and after each floating point operation.

As such the usage of fixed-point operations will in most cases allow significantly more operations to be placed on the DFE. However, if a design is mostly constrained by the I/O bandwidth and there is no option to remove this bottleneck, floating-point might still be sufficient. This can be determined using the performance model.

The first step in order to generate a fixed-point implementation is to understand the numeric properties. This can be achieved using tools which collect data during the execution of the software. Usually the easiest way to integrate these tools is again by instrumenting the software model which is supposed to use the same numerics as the intended design. As in section 4.2.1, it is crucial to select a representative input data set, in order to ensure representative results. More details on this are provided in section 11.

4.3.2 Bit-level Accurate Fixed Point Simulation

Using the results of the numeric analysis the software model is transformed to use fixed point arithmetic in order to derive a functional fixed point implementation. In general a reduction in data width leads to a smaller resource usage. If FPGA logic resources are used this dependency is usually linear, whereas for other functions the resource usage usually changes in a step function, where certain widths, depending on the underlying hardware architecture, provide the best trade-off. The functional correctness is verified by comparison with the original software using representative data.

The problem of floating-point to fixed point conversion is widely covered and there are many tools available addressing this problem [4, 5, 6, 7, 17, 18, 19, 27]. Performing the fixed point conversion at such an early step in the design process enables much larger acceleration structures, a more precise performance model and avoids highly complex debugging of numeric problems in hardware.

If the dynamic range of the algorithm is so large that the required data width causes problems further strategies can be used. One option is the usage of shared exponents, which is for example often used to solve partial differential equations for the oil and gas industry, where a wavefield decays over time, but all values in the wavefield can share the exponent [26]. Alternatively the CPU handles rarely occurring overflows.

4.4 Forecasting System Characteristics

With the results of the code analysis and the initial version of the software model an accurate performance model capturing the implementation of the planned architecture on the selected platform can be built. In order to achieve this the hardware and bandwidth usage as well as the execution time for a problem of a defined size are estimated. A preliminary speed-up expectation is thus obtained.

The performance model enables the design space exploration but also emphasises potential problems and bottlenecks of the architecture before the implementation is started. Since the architecture tries to alleviate the bottlenecks highlighted by the performance model an iterative improvement of performance model and architecture is often necessary. As such it is possible to perform design space exploration early on in the acceleration process, enabling cost-benefit analysis and resulting decision making.

Since DFEs only consist of essentially predictable building blocks, it is possible to estimate the performance for a given architecture very accurately using only a few simple equations. In contrast, on a CPU or GPU performance improving components like caches and branch predictors greatly limit the ability to predict the performance of a given application.

The easiest way to create a performance model is a spreadsheet. Different sheets can be used to capture different aspects of the design, e.g., different functional components. Equations should be used as far as possible, to enable the design space exploration. For example the unrolling factor of a loop can be changed, directly indicating the impact on resource usage and execution time.

The time it takes to process a given workload on a DFE (T_{tot}) can be represented as the sum of the time it takes to initialise the DFE (T_{init}), for example to set up DMA requests, registers or to fill the computational pipeline and the time the actual execution takes (T_{exec}). This is shown in equation 1.

$$T_{tot} = T_{init} + T_{exec} \tag{1}$$

If the workload is sufficiently large, which is normally one of the prerequisites for DFE acceleration, the execution time is supposed to dominate over the initialisation time so that it can be safely ignored. The precise initialisation time depends on the used platform. The normal SLiC action overhead is between 1 and 100 ms. However, if many mapped elements are present in a maxfile, this time can increase further. Additionally it is possible to take the time for reconfiguration into account.

In a streaming dataflow design the execution time is the maximum of the time it takes to perform the computations performed by the algorithm (T_{comp}, section 4.4.2), the time it takes to transfer the data between host and FPGA (T_{comm}, section 4.4.3) and the time required to transport data between FPGA and on board memory (T_{mem}, section 4.4.4). As a result the longest latency dominates the overall execution time of the accelerated task and becomes the primary focus as shown in eq. 2.

$$T_{exec} = max(T_{comp}, T_{comm}, T_{mem}) \tag{2}$$

It should be noted that this is only the case if all resources are utilised at a constant rate with respect to time. This, however, may not always be the case, e.g., it might be that all communication to the host has to happen before the compute can start. When the utilisation is non-constant, but its moving average is, contiguous flow can be mimicked by sufficiently large buffers smoothing the effect over time. If this is not possible the execution time of these tasks has to be treated as an additive term to the overall runtime.

4.4.1 Predicting Area Usage

In order to determine the time requirements for the compute one first needs to find out what degree of parallelism is achievable for a given device. This is mainly limited by the amount of hardware resources available on the FPGA fabric.

In general, the FPGA hardware resources are used for three different purposes. First, to implement the arithmetic operations, second the scheduling of operations and finally other IP modules, e.g., the PCIe, memory controllers, etc.

In order to predict the area usage for arithmetic operations it is first necessary to find out the amount of hardware resources required to implement a simple operation on the target FPGA device. Those figures can be determined either by using automated tools, finding appropriate tool and FPGA vendor documentation (e.g., [1, 2, 34, 35, 36, 37]) or by creating microbenchmarks and performing place and route for them. The overall area usage can be roughly estimated as the area cost of a single operation multiplied by the number of operations to be implemented on the fabric.

MaxCompiler also offers an API to provide area usage predictions, which also considers the current configuration of the kernel. It is possible to query some area usage prediction in any kernel using:

__int__ getResourceInfo().getLatency(OperatorType op, __DFEType__... type)
__int__ getResourceInfo().getFmemLatency()
ResourceEstimate getResourceInfo().getResourceEstimate(OperatorType op, __DFEType__... type)
ResourceEstimate getResourceInfo().getFmemResourceEstimate(__DFEType__ type, __int__ depth)

More information can be found in the JavaDoc of these functions. Additionally, one should consider that the predictions of these functions is never as accurate as a placed and routed microbenchmark using the combination of local environment, MaxCompiler version and vendor tool version. As such this prediction is only a first order estimate. Furthermore, for MAX5 generation devices section 13 provides additional information on expected area usage.

The scheduling is implemented using FIFOs, which usually use on chip memory resources, which will be discussed in more detail in section 4.4.1, or registers.

For IP modules usually the resource requirements can be predicted using micro benchmarks. In general, it is recommended to assume a slightly higher memory and logic usage, in order to keep some safety margins especially for scheduling but also additional control logic.

Conditional statements On CPUs it is only necessary to execute the selected branch of a conditional statement. However, in hardware it is necessary to implement logic which can deal with all possible

branches. The final result is then selected from all computed alternatives using a multiplexer. As such the resource requirements for the different branches need to be accumulated together with the cost of the multiplexer.

In cases where only one branch is active at a time, resources can be time shared. If for example two branches perform a multiplication one can use multiplexers on the inputs of the multiplier instead of on the output. This can be done manually or automatically as described in 8.3.

Dealing with loops In general, loops either have to be fully unrolled, partly unrolled or not implemented at all. If the loop is unrolled, the operation count has to be multiplied with the unrolling factor. Unrolling is especially recommended, if dependencies between loop iterations exist. If the loop cannot be fully unrolled due to area limitations, the loop normally is implemented in a pipelined fashion. This means that the loop is only partially unrolled and then the full loop is computed iteratively across multiple cycles. Again, the operation count has to be multiplied by the number of unrolled loop iterations to generate an accurate resource model. While in the case of for-loops it is usually straight forward, to determine how many loop iterations need to be implemented on the surface of the reconfigurable fabric, for while-loops this is not the case. If possible all while-loops should be transformed into loops with a fixed number of iterations. If this is not possible the loop cannot be completely unrolled. In order to estimate the computation time one can assume the average number of iterations for the average case and the maximum number of iterations for the worst case.

More details can be found in the loop tutorial.

Predicting on chip Memory Usage On chip memory is normally used for three different main purposes:

1. in operations and IP-blocks, e.g., DDR memory or PCIe;
2. for on chip buffering or reordering of data; or
3. for the FIFOs used to schedule the kernel's dataflow graph.

Predicting the area usage for the first case is very straight forward, since it can be estimated with micro benchmarks. The same can be done for the second case, however it is also possible to model it using eq. 3, where the number of memory blocks required (n_{mem}) is calculated as the product of the width (w) required divided by the native width of the on chip memory read and write ports, the depth (d) required divided by the possible depth of the hardware unit as well as the number of read ports (p) required divided by the number of read ports present in the hardware.

$$n_{mem} = \left\lceil \frac{w_{req}}{w_{hardware}} \right\rceil \times \left\lceil \frac{d_{req}}{d_{hardware}} \right\rceil \times \left\lceil \frac{p_{req}}{p_{hardware}} \right\rceil \tag{3}$$

FPGAs offer different on chip memory types with different aspect ratios that can be represented by splitting up the required memory into tiles with a size which fits better into the supported memory configurations (see section 13). The total hardware costs of the memory is the sum of the costs for each individual tile.

Predicting operation scheduling resources accurately is nearly impossible, since this would require deep knowledge of the scheduling algorithm used by the toolchain or manual scheduling (see section 7.1). However, it is possible to identify the largest FIFO needed to access data from previous cycles and treat them as normal buffers. The amount of memory resources required by the smaller FIFO is highly dependent on the application. As such only a rough estimate is possible.

4.4.2 Predicting the Compute Performance

After having estimated the area usage, one can predict the achievable level of parallelism and the number of data items processable per clock cycle, ignoring bandwidth limitations.

The time needed to compute a set of data items can be calculated as shown in eq. 4, by dividing the number of items that need processing by the product of the targeted frequency and the number of items processed per cycle.

$$T_{comp} = \frac{n_{total}}{n_{cycle} \times f} \tag{4}$$

The frequency cannot be precisely predicted, however if sufficiently deep pipelining is applied, as done by MaxCompiler automatically, and not more than 80% of the chip resources are used the frequency for different designs using the same FPGA can be safely estimated based on previous experience or experiments using artificial designs to fill up the chip. A MAX4 based DFE usually achieves 200 MHz if the chip is only filled up to 80% and a MAX5 based DFE 300-350MHz.

4.4.3 Predicting I/O Bandwidth Usage

The bandwidth between the host and the FPGA depends on the physical interconnect used. For most acceleration platforms the interconnect is PCIe. PCIe is built by bidirectional links, meaning that the full bandwidth can be used in both read and write direction at the same time. For example PCIe of the second generation can transport up to 4 GB/s over an eight lane link.

As a result, the communication time between host and accelerator can be estimated using the maximum data sizes transferred in either direction (S_{in} and S_{out}) divided by the bandwidth (BW), as written in eq. 5.

$$T_{comm} = \frac{max(S_{in}, S_{out})}{BW} \tag{5}$$

4.4.4 Modeling On Board Memory Behaviour

On board memory for FPGA accelerators is mostly based on DDR memory. The bandwidth to this memory is unidirectional, meaning that the bandwidth is shared between both directions (read and write). As such the time it takes to transfer data between on board memory and the FPGA can be calculated as shown in eq. 6.

$$T_{mem} = \frac{S_{write} + S_{read}}{BW \times efficiency} \tag{6}$$

The parameter *efficiency* represents the proportion of the theoretical DDR bandwidth achievable for a given data set. Only large linear access patterns allow efficient memory access. Fig. 7 shows the achievable memory efficiency based on the amount of data which are accessed in one linear read for DDR4 memory as used on a MAX5 generation DFE card. Different DDR based memory technologies behave in a similar manner.

The location in memory at which the data are stored also has significant impact on memory bus efficiency. The address space is divided into multiple ranks[4]. Due to the fact that if one rank is currently accessed the next rank can already be prepared, accessing data from different ranks will make sure that efficiency can be improved. If only one burst is accessed at a time the efficiency will be halved, but if more bursts are accessed this penalty will be significantly reduced. For this reason, it is beneficial to distribute the data uniformly across the available address space.

[4] All DRAM chips sharing the same chip select [22]

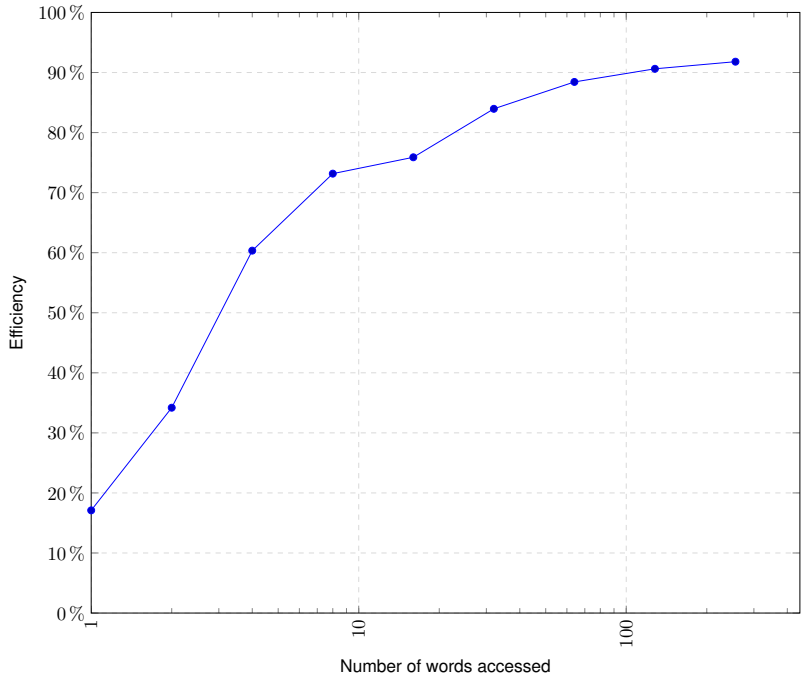

Figure 7: DDR4 memory efficiency as measured on an Alveo U250

4.5 Architectural Optimisations

The details of developing an efficient architecture go beyond the scope of this document. However, some general concepts should be explained here, since only the development of a good architecture promises optimal results. Additional information can be obtained in sections 6 and 8.

In general, the target is to achieve a balance, where all system resources are fully and equally used, and all bottlenecks of the system as indicated by the performance model are removed. This means that in the best case the memory and PCIe bandwidth as well as all of the hardware resources of the DFE are fully used and none of the above becomes a bottleneck. If, for example all the memory bandwidth is utilised but only 50% of the chip resources are in use, a better architecture should try to save memory bandwidth. This will allow an increase of the degree of parallelism, using more of the chip resources and reducing the overall execution time.

4.5.1 Improving Bandwidth Utilisation

Reducing the memory- or host-to-accelerator bandwidths are very similar. There are three main strategies to achieve this. First, is to buffer more data on chip, reducing the need for streaming it back and forth. Secondly, customised compression can be used. This could be as simple as changing the data width (e.g., from 32 bit to 12 bit integers). However, more advanced compression algorithms e.g., run length encoding can also be a good choice. Additionally, DFEs provide unique opportunities for completely custom compression algorithms. Another widely used option is to generate the sequences with predictable values on the FPGA, avoiding transmission altogether.

In the case of on board memory it is often also possible to increase memory efficiency, e.g., by reordering data on the chip just before it is written or after it is read in order to create linear access patterns. Reordering operations to reduce the amount of data that needs to be moved is also an

option. Additionally, it is possible to optimise the access into the different ranks of the DDR memory, as discussed in section 4.4.4, by using an optimised memory layout.

4.5.2 Reducing Area Usage

Usually the most important step, in order to make efficient use of the available hardware resources, is the selection of the datatype and width. For example it is usually very inefficient to use floating point arithmetic, but even if fixed point arithmetic is used, smaller data widths use less resources (see section 11).

Otherwise it is possible to replace area expensive operations with cheaper ones. One option to achieve this is by considering alternative implementations of a given algorithm. Even if those implementations are less efficient on a CPU, they might be cheaper to implement in hardware. For example sorting networks are often used in hardware, while on CPUs other algorithms, e.g. merge sort, are usually preferred. Moreover, one could consider to move calculations to the CPU or precompute some values, if possible. More strategies are discussed in section 8.

4.5.3 Overlapping Host and Accelerator Execution

Usually applications are not fully ported onto FPGAs, but instead less compute- and more control-heavy code parts remain on the CPU. The simplest way of integrating the FPGA into the CPU implementation is to call the FPGA as a function. However, as shown in fig. 8 a), this leads to an inefficient usage of the available hardware resources, where only one of the resources is fully used at any time.

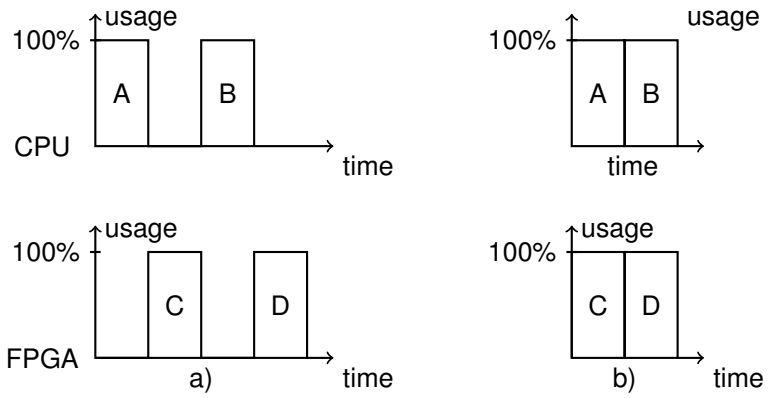

Figure 8: Suboptimal vs overlapped use of resources

A better way to handle this integration is to overlap the execution as shown in fig. 8 b). One way to achieve this is by calling the FPGA in a separate thread. If the CPU and FPGA calculations are independent then it is easy to parallelise them. Otherwise it is recommended to have the CPU and the FPGA work either on different parts of the same dataset or completely disjunct tasks. More details on this are described in section 10.1.

4.6 Usage Examples

In this section the application of the methodology is shown on two real life projects which were performed by Maxeler. One is the acceleration of the inference of the VGG-16 [28] CNN discussed in section 4.6.1 and the second is the acceleration of BQCD which is discussed in section 4.6.2.

In both cases we used Maxeler MaxCompiler version 2017.1 and Vivado 2017.1. We target Maxeler's MAX5C Dataflow Engine (DFE) as our FPGA platforms. The compute device of the MAX5C DFE as described in section 13.

4.6.1 Convolutional Neural Network

The details of the analysis and software model for the CNN are not described here, since their results are already encapsulated in fig. 6. The loopflow graph also motivates the split of the application. In this case the convolutional layers are ported onto the FPGA, while the fully connected layers stay on the CPU.

Architecture In this section we first define an initial architecture to determine what to model in the performance model. There are two fundamentally different options to implement a CNN on FPGAs. The first option is the implementation of a fully streaming architecture, in which hardware is dedicated to each layer separately and all layers are computed in parallel. However, this architecture often requires a prohibitive amount of on chip memory, since on board memory does not provide the required bandwidth. The second option is the usage of a more generic hardware unit, often called Processing Elements (PE), which is able to perform the necessary computations for all layers. In this case only parallelism within a layer is exploited.

To estimate the on chip memory usage of the fully streaming architecture it is possible to calculate the memory needed to store the results of the first layer. If a resolution of 224x224 pixels is assumed the 64 output channels of the VGG-16 CNN produce in total 3,211,264 elements of data. This means that ~ 3MB of on chip memory are required if each element occupies one byte of memory.

As a result it is not feasible to implement the fully streaming architecture. Instead the PE based architecture shown in fig. 9 is selected. Each PE performs the convolution operations, accumulation over input channels and the Rectifier Linear Unit (ReLU) activation functions [42]. To save on board memory bandwidth all PEs operate on the same input channel but produce different output channels, but the hardware unit performing pooling and writing data back to on board memory is shared between PEs.

The following sections will provide the detailed performance model for this architecture.

Modelling Host to Accelerator Communication First the I/O communication time will be predicted, using eq. 5. For this the amount of data that needs to be transferred to and from the DFE needs to be determined. This consists of the weight and input image data.

The weight data are constant across iterations and as a result only needs to be transmitted once. As such they can be neglected since the one-time transmission is of no consideration as soon as thousands of images are processed.

In cases when it is not possible to compute all output channels of the first layer in parallel, it is necessary to retransmit the input or buffer it in memory. Eq. 7 shows the situation when the data is retransmitted. n_{out} is the number of output channels of the first layer, n_{PE} the number of processing elements and n the number of images transferred.

$$T_{comm} = \frac{n \times max(S_{in} \times \frac{n_{out,L1}}{n_{PE}}, S_{out})}{BW} \tag{7}$$

Eq. 7 becomes eq. 8, to estimate the number of items that can be processed in a given time. To obtain the number of input items per second T_{comm} is set to one second.

$$n = min\left(\frac{BW \times T_{comm}}{S_{in} \times \frac{n_{out,L1}}{n_{PE}}}, \frac{BW \times T_{comm}}{S_{out}}\right) \tag{8}$$

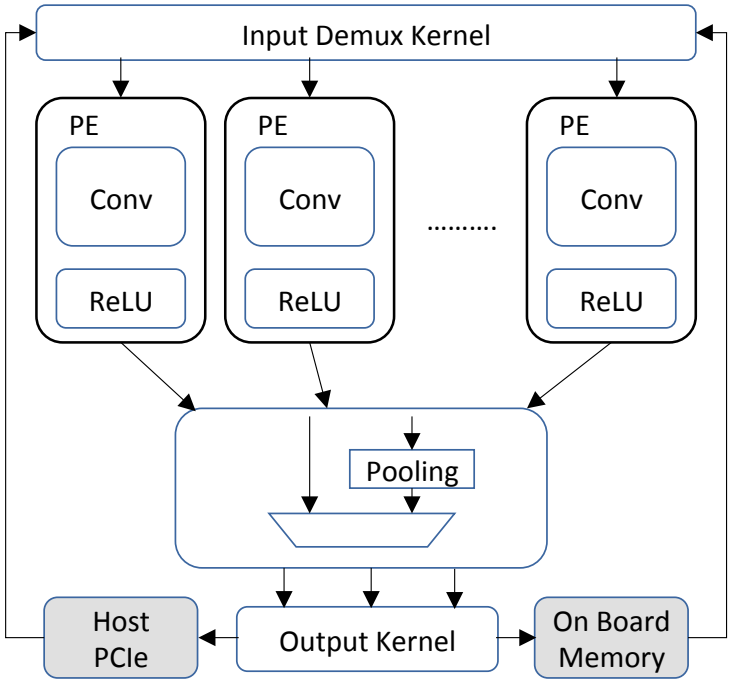

Figure 9: Convolution design architecture, PE connectivity

Forecasting Accelerator Computational Latency Similarly, the accelerator runtime can be estimated as shown in eq. 9, where n again denotes the number of input images, OPS_{needed} the number of operations performed per input image and $OPS_{available}$ the number of operations that can be performed on the FPGA for the used architecture within a given amount of time.

$$T_{comp} = \frac{n \times OPS_{needed}}{OPS_{available}} \tag{9}$$

The maximal number of processed elements as limited by the on chip computational capabilities is given by eq. 10.

$$n = \frac{OPS_{available} \times T_{comp}}{OPS_{need}} \tag{10}$$

Since the convolutional layers mainly consist of Multiply Accumulate (MAC) operations, one needs to calculate the number MAC operations implemented and needed in order to estimate T_{comp}. This assumes the implementation of the ReLU and pooling functions directly on the data streams as proposed in this architecture.

Estimating On Chip Memory Usage Depending on the on chip memory size it might be feasible to buffer all the input and output channels to each layer on chip. However, on smaller chips, or for layers having either more or bigger output channels, this is not feasible. Similarly there are networks like ResNet-101 or other residual networks, which reuse data from earlier layers [14], and therefore require even more buffer space. In those cases it is necessary to store the results produced by the computation of each layer in on board memory. Only this case will be modelled here.

Each processing element works on a separate output channel, hence all the data produced by each processing elements needs to be written back to memory.

Based on eq. 3, eq. 11 describes the amount of required on chip memory, where n_{PE} is the number of processing elements, p the number of output pixels created per PE per cycle and w the dat-

apath width. Furthermore $elem_{out,layer}$ presents the number of elements in a given output channel. MEM_{width} and MEM_{depth} are the memory port width and memory depth.

$$n_{mem} = \left\lceil \frac{n_{PE} \times p \times w}{w_{hardware}} \right\rceil \times \max_{layer} \left(\left\lceil \frac{2 \times elem_{out,layer}}{p \times d_{hardware}} \right\rceil \right) \tag{11}$$

The first term in eq. 11 represents the requirement to write all results produced by the PEs into memory on every cycle, while the second term represents the need to store all output channels of each layer on chip, using double buffering. Only one read port is needed, removing the last term of eq. 3.

Double buffering is needed, to avoid writing all outputs to external memory at the same time. The latter would require a prohibitively large buffer to the on board memory in order to smooth over this burst while consuming enough data at each cycle. Double buffering additionally improves the efficiency of the DDR memory if each output can be written back separately, since it enables data access in long continuous bursts.

Estimating On Board Memory Bandwidth Utilisation The amount of data that needs to be read to load the weights is shown in eq. 12, where n_{in} and n_{out} represent the number of inputs and outputs per layer and $S_{weights}$ the size per weight filter.

$$S_{weights,total} = \sum_{l=0}^{layer} n_{in,l} \times n_{out,l} \times S_{weights,l} \tag{12}$$

The amount of memory that needs to be accessed per input element for one layer can be estimated by eq. 13, where n_{in} represents the number of input channels to the layer and n_{out} the number of output channels. S_X represents the size of one input channel, S_Z the size of one output channel and n_{PE} the number of processing elements and therefore outputs processed in parallel.

$$S_{layer} = S_X \times n_{in} \times \left\lceil \frac{n_{out}}{n_{PE}} \right\rceil + S_Z \times n_{out} \tag{13}$$

The boundary can be estimated as shown in eq. 14, where again the efficiency will be close to the maximum since all memory accesses are linear and most of them will use the maximum number of bursts.

$$n = \frac{BW_{mem} \times efficiency \times T_{mem}}{\sum_{layer} S_{layer} + S_{weights,total}} \tag{14}$$

Design Space Exploration Using these equations from the performance model it is straight forward to perform a design space exploration. The FPGA used in the MAX5C DFE consists of three separate dies, called SLRs, which are mounted on the same silicon interposer. The communication between those dies is a significant bottleneck.

However, to get around this problem, all three SLRs are treated as individual FPGAs. Each SLR is connected to one separate DDR Dual In-Line Memory Module (DIMM) and the PCIe bandwidth is shared.

The size of the images fed into the CNN is fixed to 224×224 images and as a result $150,528$ pixels need to be sent into the DFE and $100,352$ elements need to be received. Assuming that 16 PEs are used the number of images that can be processed within one second can be calculated using eq. 8 as shown in eq. 15.

$$n = min(\frac{4GB/s \times 1s}{401{,}408B \times \frac{512}{16}}, \frac{4GB/s \times 1}{602{,}112B}) = 334 \tag{15}$$

The design is heavily constrained by the number of on chip multipliers. As such only modelling of the multipliers will be presented here. If each of the 16 PEs processes 14 pixels per cycle the number of multipliers used can be calculated as shown in eq. 16, where n_{filter} represents the number of elements in each weight filter. This fits within one SLR, which has 2,280 multipliers.

$$n_{mul} = n_{filter} \times n_{PE} \times n_{pixelspercycle} = 9 \times 16 \times 14 = 2{,}016 \tag{16}$$

For the whole network 1,705,181,184 convolutions need to be computed for each image. The proposed architecture can perform, as just calculated $16 \times 14 = 224$ convolutions per cycle. Using eq. 10, we can calculate the compute boundary based on the frequency as shown in eq. 17. For a frequency of 250 MHz this would mean that 32.8 images can be processed per cycle.

$$n = \frac{224\frac{OPS}{cycle} \times f \times 1s}{1{,}705{,}181{,}184OPS} \tag{17}$$

It was decided, that the data between layers should be buffered on board. As a result the DDR bound performance can be estimated using eq. 14. Per image 355 MB of pixel data and 33 MB of weights need to be transported. The memory efficiency used for this estimation is 0.85.

$$n = \frac{14GB/s \times 0.85 \times 1s}{388MB} = 33 \tag{18}$$

As a result the overall expected performance of all three SLRs is 99 images per second. Fig. 10 shows other possible design points, where the equations are evaluated for the different degrees of parallelism and the area usage is predicted to discard all invalid design points. Blue dots in the figure are considered in the DSE (chosen design point in green). Red and grey dots where not considered because of resources and complexity constraints.

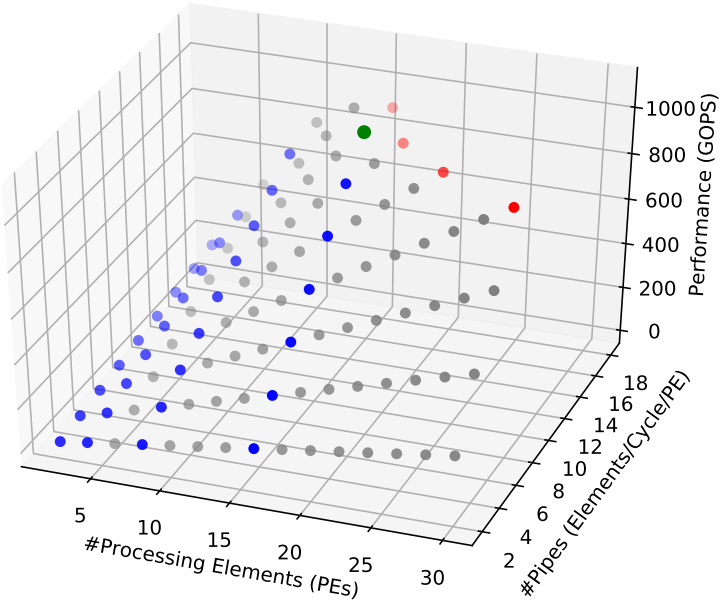

Figure 10: Design Space of the proposed architecture for the VGG-16 network.

Experimental Results and Discussion Following the results of the design space exploration the actual implementation was performed. Benchmarking of the CNN shows that the implemented design delivers 84.5 images per second at 240 MHz. This means that the error of the performance model is less than 15%. If a frequency of 240 MHz is used in the model the error decreases to less than 10%. This remaining difference can be explained by an overestimation of the on board memory efficiency.

4.6.2 Berlin Quantum Chromodynamics

Our methodology was applied to a widely used Quantum Chromodynamics (QCD) application. QCD is the physical theory of strong interactions between subatomic particles, and Lattice QCD (LQCD) is an approach to computationally simulate such particles, based on discretising space and time into a 4D lattice [32]. Berlin QCD (BQCD) [23] is a popular implementation of LQCD, and was part of the PRACE-PCP challenge [25] workloads. BQCD natively supports timing its compute steps and a generated output file includes a detailed timing report that made utilities like gprof not necessary. Since the majority of the BQCD compute time was spent in its Conjugate Gradient (CG) solver, we focussed on porting this portion.

BQCD [23] is a popular implementation of LQCD. It natively supports timing its compute steps and a generated output file includes a detailed timing report that made utilities like gprof not necessary (Part 1). Since the majority of the BQCD compute time was spent in its CG solver, we focused on porting this (Step a).

In terms of code complexity, the complete code base of BQCD contains roughly $200,000$ lines of code, 63% of which is Fortran-90, 37% C, and 0.1% C++, though not all of this code is accessed for the input files and compiler options considered. The portion accelerated consists of roughly $1,500$ lines of actually used (accessed) Fortran code. Our implementation (at the time of writing) consists of a total of roughly $10,500$ lines of MaxJ[5] and Java, plus about $4,200$ lines of C and and 130 lines of Fortran for the interface between BQCD and the DFE (most of which consists of data reordering and fixed-point conversion). As was the case for the full BQCD project, the number of lines is reduced when only code is considered that is accessed as part of a BQCD run: in that case the interface represents roughly $1,600$ lines of code, and the MaxJ portion about $4,800$.

The results presented below show 20x improvement over a BlueGene/Q on a compute-per-volume basis.

A CG algorithm [15] iteratively solves an $M\mathbf{x} = \mathbf{b}$ equation for the unknown vector $\mathbf{x}$, where M and $\mathbf{b}$ are known. The algorithm relies on repeated multiplication of M by so-called conjugate vectors. In the case where M is sparse, rather than performing an explicit matrix-vector multiplication, it is more efficient to instead infer the effect of M when applied to a vector $\mathbf{p}$ by referencing only selected elements of M.

In the case of BQCD, each lattice point contains a so-called spinor which, for our purposes, is a block of 12 complex numbers, and the vectors are enumerations of the spinors of all the lattice points. The matrix M consists of repeated application of so-called Wilson $\not{D}$ (d-slash) and clover operators (both sparse) to the lattice of spinors. Rather than explicitly constructing M and computing a matrix multiplication, BQCD applies $\not{D}$ and clover to a lattice of spinors four times in succession for each CG iteration. For the application of $\not{D}$ the program needs to make reference to the current output site's neighbouring spinors (in 4D) and, furthermore, one so-called gauge matrix (3×3 matrices with complex elements) per neighbour.

[5] Maxeler's hardware programming language

Architecture During performance modelling , it became clear that the QCD application would be bound by on-board memory bandwidth and, hence, most design decisions were taken in order to minimise on board memory I/O. First of all, a CG algorithm variant was chosen that removes synchronisation points from within CG iterations, thus ensuring that an iteration requires only a single pass over the matrix and vector data [9]. Second, an inner product arising in the CG algorithm, $\langle \mathbf{p}, M\mathbf{p} \rangle$, was rewritten as a square norm $(W\mathbf{p})^2$, where $M = W^H W$, and $W\mathbf{p}$ is an intermediate result that naturally occurs during the computation of $M\mathbf{p}$. This transformation saves a large amount of on chip buffer space because the computation of $M\mathbf{p}$ has a very high latency and re-streaming $\mathbf{p}$ from on board memory would hurt the overall performance severely. A third optimisation exploits the mathematical properties of the 3×3 gauge matrices to express 3 elements in terms of the other 6, hence reducing the required off board memory bandwidth by 33%. Lastly, the gauge streams needed for the four successive $\not{D}$ operations applied during a CG iteration are dependent on one another; specifically, the stream for the first and third $\not{D}$, as well as for the second and fourth $\not{D}$, are identical, and the second gauge stream can moreover be computed from the first, albeit at the cost of a significant amount of on chip memory. Thus, a gauge initialisation kernel was added to the design.

The resulting architecture is displayed in fig. 11. Spinors, gauges and clover matrices are read from on board memory and streamed into the CGKernels (in the case of gauges, via the InitGauge kernel). Each CGKernel performs a $\not{D}$ and clover operation, and some perform additional calculations of the form $a\mathbf{x} + \mathbf{y}$ operations (known as "AXPY"; here a is a scalar and $\mathbf{x}$ and $\mathbf{y}$ are vectors) required by the CG algorithm. Some spinor vector results are output by CGKernel0, whereas the result of the matrix multiplication $M\mathbf{p}$ is output by CGKernel3. These results are streamed back to on board memory to be used by the next CG iteration. CGKernels 0 and 1 accumulate the square norm of certain spinor vectors, which at the end of a CG iteration are fed to the CGControlKernel, which uses these numbers to compute the scalars required for the next CG iteration. Note that to save chip space the CGKernels process a lattice site only every 4 cycles. Since a CGKernel needs to reference all neighbours in 4D, its latency is effectively $2L_X L_Y L_Z \times 4$, where L_i denotes the number of sites in direction i,[6] and the factor 4 represents the number of cycles needed per site.

The resulting architecture is displayed in fig. 11. Spinors, gauges and clover matrices are read from on board memory and streamed into the CG algorithm. Each CGKernel performs a $\not{D}$ and clover operation and some perform additional $a\mathbf{x} + \mathbf{y}$ operations required by the CG algorithm. Some spinor vector results are output by CGKernel0, whereas the result of the matrix multiplication $M\mathbf{p}$ is output by CGKernel3. These results are streamed back to on board memory to be used by the next CG iteration. CGKernels 0 and 1 accumulate the square norm of certain spinor vectors, which at the end of a CG iteration are fed to the CGControlKernel, which uses these numbers to compute the scalars required for the next CG iteration. Note that to save chip space the CGKernels process a lattice site only every 4 cycles. Since a CGKernel needs to reference all neighbours in 4D, its latency is effectively $2L_X L_Y L_Z \times 4$, where L_i denotes the number of sites in direction i, and 4 the cycles needed per site.

Performance modelling The performance of the architecture can be modelled as follows.

Compute-bound time to solution Each kernel in the diagram presented in fig. 11 needs to process a number of items as shown in eq. 19

$$n_{items} = (L_X + 2H_X)(L_Y + 2H_Y)(L_Z + 2H_Y)(L_T + 2H_T) \tag{19}$$

[6] Strictly speaking, the real number of sites simulated by BQCD is doubled by a property of the $\not{D}$ operator known as even-odd preconditioning. We can avoid this subtlety here by defining L_i to be number of sites in direction i as actually processed by the CG algorithm. As such, any dimensions mentioned in this article are given from the point of view of the CG algorithm.

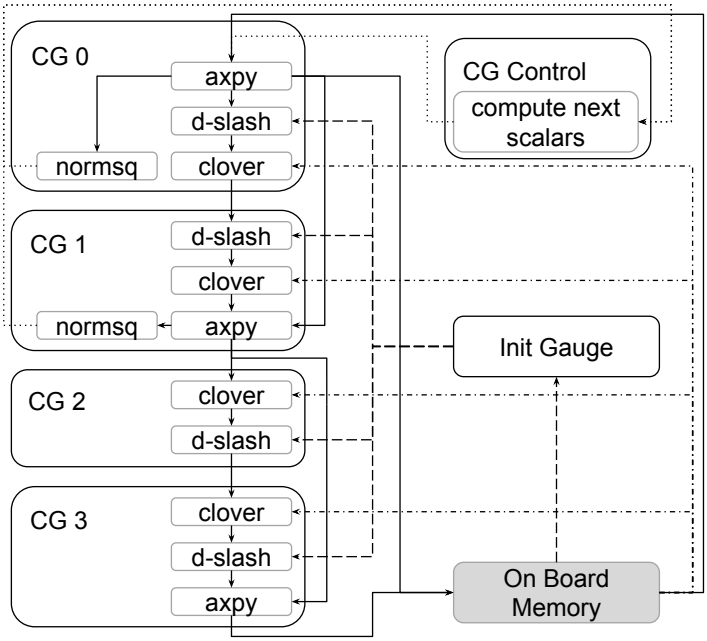

Figure 11: BQCD design architecture

where H_i denotes the number of halo sites in direction i ($i \in \{X, Y, Z, T\}$). In our example, $H_X = H_Y = H_Z = 0$ for all kernels, and the maximum value taken by H_T is 4 (this occurs both in the gauge initialisation kernel as well as CGKernel0). The number of items processed per cycle is $1/4$, as 4 cycles are required to process a single item. In addition to these considerations, we need to account for the pipeline latency, because a new CG iteration cannot start until the previous one has finished outputting. The number of extra cycles incurred this way is equivalent to $3L_XL_YL_Z$ elements[7], which gets added to n_{items}. Thus, using eq. 4, the compute-bound time to process a single CG iteration is shown in eq. 20.

$$T_{comp} = \frac{4L_XL_YL_Z(L_T + 11)}{f} \tag{20}$$

The compute-bound time can be plotted as a function of inverse frequency (i.e. clock period), for a $16 \times 16 \times 16 \times 32$ problem size and a CG run that took 53 iterations to converge, as shown fig. 12.

Memory-bound time to solution To gain the memory-bound time to solution T_{mem}, first the total data size read from, and written to, on board memory has to be computed. For each ordinary lattice point 8 gauge matrices are needed which each consist of 12 real numbers, 2 clover matrices of 72 numbers, and 5 spinors of 24 numbers. For halo lattice points, instead only 3 spinors per lattice point are needed. The number of halos to read differs per data type: the gauges and 3 spinor streams are read with 4 halos, whereas one clover stream requires 3 and the other 2 halos. Hence the total number of values read from on board memory is computed in eq. 21.

$$\begin{aligned} N_{num,r} &= L_XL_YL_Z\left(L_T\left(5 \times 24 + 8 \times 12 + 2 \times 72\right)\right. \\ &\quad \left. + 8\left(3 \times 24 + 8 \times 12\right) + 6 \times 72 + 4 \times 72\right) \\ &= L_XL_YL_Z\left(2064 + 360L_T\right) \end{aligned} \tag{21}$$

[7] The factor 3 (rather than 4) arises from the fact that the next CG iteration can start even if CG kernel 3 has not finished flushing (but has finished outputting) the previous iteration

At ordinary lattice points, 5 spinors of 24 numbers each are written to on board memory, and no data is written during halos as shown in eq. 22.

$$N_{num,w} = L_X L_Y L_Z L_T \times 5 \times 24 = 120 L_X L_Y L_Z L_T \tag{22}$$

Since for this example we stream all data in 24-bit (3B) fixed point format, Eq. 6 can thus be evaluated as shown in eq. 23.

$$T_{mem} = \frac{L_X L_Y L_Z \left(2064 + 480 L_T\right) \times 3\text{B}}{BW \times \textit{efficiency}} \tag{23}$$

In fig. 12, the on board memory bound time is shown as a horizontal line (it is independent of kernel clock frequency), for a problem size of $16 \times 16 \times 16 \times 32$, a bandwidth of 49 GB/s and efficiency of 80% and, as before, 53 CG iterations give $T_{mem} = 270.98$ ms.

Modelled time-to-solution The previous two ingredients (compute-bound and on board memory bound time to solution) can already provide a good estimate of the expected time-to-solution. Two additional factors are the PCIe I/O time and the initialisation overhead. Both are additive (because data has to be written and read strictly before and after the CG is ran) to the total time-to-solution of all CG iterations needed to converge to a solution. Due to implementation details, only a pair of spinor vectors needs to be streamed before and after the FPGA run, whose size is computed by multiplying the number of values by 2 (to account for input and output) as shown in eq 24. For a problem size of $16 \times 16 \times 16 \times 32$, the estimated PCIe time is 11.7 ms. The action overhead is estimated to be 10ms for the QCD bitstream.

$$N_{num,pcie} = L_X L_Y L_Z L_T \times 2 \times 24 = 48 L_X L_Y L_Z L_T \tag{24}$$

The resulting theoretical time-to-solution as shown in 25 is plotted in Fig 12 as a yellow line.

$$\max\left(T_{comp}, T_{mem}\right) + T_{PCIe} + T_{overhead}, \tag{25}$$

Experimental Results and Discussion We target the same hardware platform as described in section 4.6.1. Fig. 12 depicts the experimental results. Generally the estimated results represent measurements very closely. However, at high frequencies, a discrepancy in time-to-solution of around 10% is observed. This discrepancy is due to the assumption that data accesses to on board memory can be averaged over the run. In reality, on board memory I/O varies during a CG iteration, being lower for halo sites than when computing ordinary sites. This means that there will be a transitional frequency range where part of the execution is on board memory bound and the other part is compute bound. By treating the halo and core compute separately whilst assuming the "flushing" cycles after each CG iteration to be compute bound, a more accurate model is created (the grey line in fig. 12). It should be noted, that more simplified models, e.g., the roofline model, will not be able to accurately predict this behaviour.

4.7 Implementation

Even though the methodology itself does not cover the implementation but also describes the stages before implementation can be started we also want to cover best practices of MaxJ development. As such this section will discuss some best practices on how to most efficiently work through the actual implementation phase.

There are two major principles, which are based on the difficulty of debugging in hardware and making changes late in the design process.

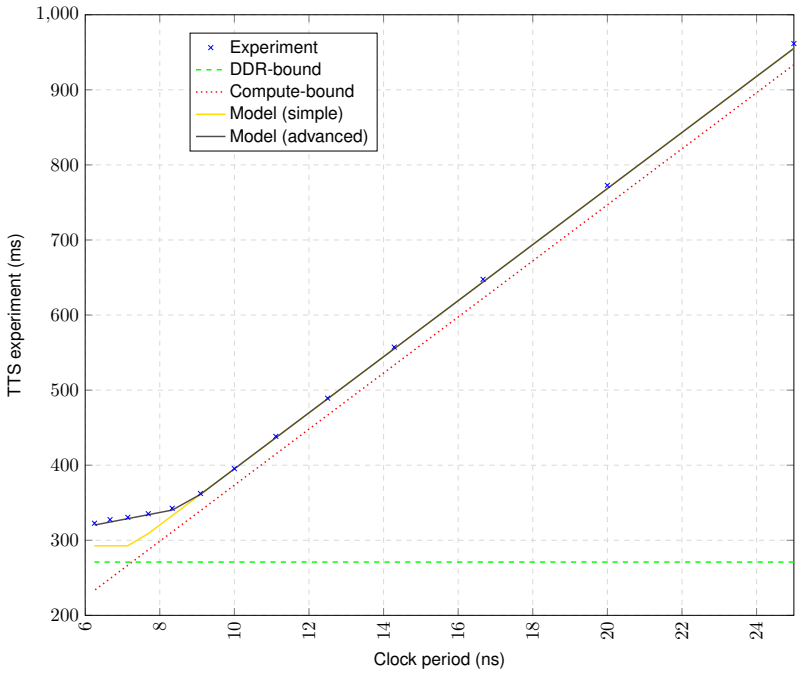

Figure 12: BQCD CG time-to-solution

1. Test as many components as possible in isolation.

2. Avoid as many surprises as possible.

On the testing side, it is often a good decision to test components by functionality in isolation as well as integrated. E.g. one should test an arithmetic function, the buffers as well as the control logic for a specific functionality all isolated once and then once integrated.

These test can use the normal Java JUnit unit testing framework to be easily checked throughout the development process (or even from a Continuous Integration Framework). The debugging tutorial provides an introduction into how one can achieve this. By isolating the individual parts it is significantly to track down if a problem is caused by a specific component or an integration problem. Additionally, these small components can then be fully tested and debugged from within the Java project.

Additionally, it is recommended to start building for hardware as soon as possible. Even if the bitstream does not not provide any functionality and the CPU code does not exist it is important to double check if the area requirement predictions are accurate compared to the actual place and route results. If they are not it is better to discover potential problems or errors as soon as possible to address them. In the best case the error is on the MaxJ development side and can be easily fixed and in the worst case a fix to the performance model and even the architecture is required.

By building the design regularly it is also possible to get an idea of the timing properties. This again helps to improve the performance prediction on the one side, but also highlights problems early on. Timing improvements are often a time consuming issue, due to the iterative process of it. One has to discover a problem, try to address it and see if that works. By starting as soon as possible, timing closure can overlap with the actual development avoiding potential delays due to timing paths.

5 Low level resource usage and MaxJ

As described in section 3 the mapping between MaxJ code and hardware structures is usually very direct. This section will describe which resources are used for what. Additionally we will describe how many resources are used in what cases to provide the reader fundamental understanding for the amount of resources required by individual operations. To measure the precise resource requirements of a specific operation are a code section we recommend the usage of micro benchmarks. This means that one runs place and route for the individual code section and uses the resource annotation to gather the resource requirements.

5.1 Logic

Logic resources are used by pretty much everything. For example a FIFO as well as a multiplier will need some additional logic e.g. to control the remaining component or provide additional features not supported by the dedicated hardware as well. Logic resources are also used for short term buffering of data. Physically it consists of two individual resources, which we will discuss separately.

5.1.1 FlipFlops (FFs)

A single FF can store one bit. This can be used to for example delay a signal. As such FFs are usually used to form a register after every operation. The amount of pipelining used will increase the amount of FFs required. On top of this FFs can also be used as dedicated memories using the *stream.hold* API.

5.1.2 LookUp Tables (LUTs)

LUTs are used by nearly all components of a MaxJ design. This includes interfacing components like the PCIe or memory controller but also the control logic of the kernel. Additionally, all memory and multiplier operations use some LUTs. All control logic, including counters and state machines, is implemented using LUTs as well. However, all in all the contribution of these blocks are limited and dedicated LUT usage usually dominates the area usage.

If floating point datatypes are used each floating point operation will use a significant amount of LUTs. The precise number will depend on the selected platform but a 32 bit float addition will require around 180 LUTs. For double this increases to roughly 580 and even further if DSP usage is avoided (see below).

In contrast, fixed point operations require less area. The area usage scales linearly and for each bit of the datatype width one LUT is used. This means that a 32 bit fixed point addition requires 32 LUTs (and the same amount of FFs). A 64 bit addition will require 64 LUTs.

Divisions require a lot more resources. A 32 bit division will need more than one thousand LUTs. This is the reason why divisions should be avoided if possible. One standard strategy is to replace them by multiplications.

A Multiplexer (MUX) requires roughly the same resources as an addition, but resource usage increases if many inputs are used. Variable rotate and shift operations belong to the most expensive operations since they consists of many MUXs internally. The hardware usage increases stronger than linear, but micro benchmarking is recommended to gather precise predictions.

It is possible to also perform multiplications in logic. This can be useful if all hardware multipliers are used, but a lot of logic is still available. The MaxJ API *optimization.pushDSPFactor()* can be used to manually target logic instead of DSPs. However, only very few multiplications will fit into the chip, since the resource usage is significant.

Logic operations like and, or, xor and equals require one LUT for each bit. By default MaxCompiler will insert a register after each individual logic operation. However, the LUTs used in the fabric of the FPGA usually has more than two inputs. As such it is beneficial to reduce the pipelining for these operations to squash multiple operations into a single LUT. More details on this will be described in section 7.

On top of all of this LUTs can also be used to form small memories. Since the capacity is fairly limited they are usually only used for the scheduling of the dataflow graph. The usage of memories is described in more detail in section 5.3.

5.2 DSPs

DSPs are used only for arithmetic. In fixed point designs this means only for multiplications and more complicated functions like trigonometric functions or the exponential. The DSP usage depends on the port width of the multiplier. On MAX5 a DSP has input port widths of 18 and 27 bit. As such a DSP can perform one 18 by 27 bit multiplication. For a 27 by 27 bit multiplication 2 DSPs are needed, which could also perform a 36 by 27 bit multiplication. The usage can be determined by ceil dividing the input data widths by the port widths and multiplying the results. A 32 by 32 bit multiplication will require 4 DSPs as a result.

On the MAX5 platform it is not possible to save DSP resources by reducing the data width further than the port width. However, on the MAX4 platform it is possible to tile multipliers more efficiently. By default the MAX4 multipliers are 27 by 27 bits. One of those can also perform two 18 by 18 or three 9 by 9 bit multiplications.

If floating point operations are used the amount of DSPs used follows similar rules. However here the DSP usage can be calculated based on the mantissa and not the exponent. Additionally, DSPs are not only used for multiplications but dependent on the platform and other settings also for additions.

Finally parts of the Manager infrastructure might use DSPs. For example each DIMM used on the MAX5 platform requires three DSPs. As one can see the usage is very small compared to the available resources.

5.3 Memory

Memory is used for two purposes:

1. Dedicated memories on the chip
2. Data buffering in the form of FIFOs

Dedicated memories are allocated using the MaxJ memory API. As such it is straight forward to understand the size of the instantiated memory. MaxCompiler will map those memories to BRAMs on MAX4 or older devices. On MAX5 a more complicated algorithm as described in section 5.3.1.

It is important to remember, that adding more read ports increases the hardware usage. On MAX4 memories have one read port while on MAX5 memories have two read ports. Each time the *read* function in MaxJ is called a new read port is created. If more read ports are used then available, the memory gets duplicated.

Memory usage for scheduling is sadly really hard to predict or impact. It will be discussed in a lot more detail in section 7.1. For now it is important to understand that scheduling resources are automatically allocated by the compiler. It uses FF, LUT and all dedicated on-chip memory resource. In general one can work under the assumption, that more pipelining stages will increase the memory usage. If *stream.offset* calls are used it is possible to make more accurate predictions of the memory usage.

However, since this is basically a scheduling constraint, the behaviour will usually not be completely as expected.

More complicated functions like square root and exponential also use memories. These are either used as internal look up tables or as FIFOs to delay data within the function, which takes multiple cycles to compute. Finally, the Maxeler infrastructure also requires memories. This includes the PCIe and memory controller but also the FIFOs between the components in the Manager. FIFOs in the manager have a depth of 512 by default and have the width as required by the connected components.

5.3.1 Allocating URAMs and BRAMs on MAX5 devices

On MAX5 devices multiple different on chip memory resources are present. Additionally to FFs, LUT memories and normal BRAMs there are also URAMs. URAMs have a significant higher capacity, but are less flexible. Additionally, even individual URAMs have a very large capacity and it is important to maximise utilisation to avoid hardware wastage. Since the manual selection of hardware resources would be very complicated and a significant developer overhead we decided to automate it. This section describes the motivation for the selected mapping algorithm and how it works.

Generally a good memory mapping algorithm should:

1. use as few resources as possible; and
2. facilitate timing closure.

In order to address the first objective the utilisation of the available memory resources needs to be considered. As shown in eq. 26, the utilisation of a given physical memory resource (BRAM or URAM) is the maximum utilisation of its valid tilings. The utilisation of each tiling is the ratio between the user defined logical memory size and the product of the physical memory unit size (both in #bits) and the required number of physical memories. To minimise hardware wastage the memory type with the best utilisation is selected.

$$\max \left(\frac{logical\ memorysize}{unitsize * \#units} \right)_{tiling=1}^{N} \tag{26}$$

We call a group of hardware resources with high interconnectivity placed in close proximity on the FPGA fabric a *design unit*. Design units can be specified explicitly by the user, implicitly by using language structures or by creating a high-level floorplan model. As such a design unit could, for example, contain all resources of a single computational unit, e.g., a kernel. Alternatively, if a floorplan is used, a design unit could also contain all the resources which are mapped to a certain SLR. In order to reduce SLR crossings and as a result aid timing closure, it is of major concern to balance the allocation of memory resources between different design units. Consider a case where a specific design unit requires more BRAMs than a single SLR capacity. As a result BRAMs from neighbouring SLRs have to be allocated increasing SLR crossings and routing congestion. As such redirecting some of the memories to URAMs is beneficial even though this introduces overheads in terms of allocated memory bits. In short, balanced allocation between BRAMs and URAMs is expected to improve SLR locality of individual design units. In the authors experience SLR crossings and the related routing congestions limit timing closure. Our experience is based on thousands of place and route attempts for dozens of real applications on the Xilinx VU9P. As a result the major goal for a timing optimised multi-die aware memory mapping algorithm is avoidance of unnecessary SLR crossings.

To address the above Maxeler developed the following algorithm. Our balanced mapping algorithm runs per design unit and its input is the list of logical memories. In few cases a logical memory can only be mapped to a specific physical memory, due to specific hardware features, e.g., dual clock

domain support. Such special logical memories are handled first to ensure mapping to the appropriate hardware resources.

Afterwards another preprocessing stage decides which of the remaining logical memories should be mapped to distributed RAM. Since the logic resources used to implemented distributed RAM are less scarce, this mapping can be based on a simple heuristic and does not need to consider SLRs. This heuristic is based on the BRAM utilisation calculated using eq. 26. It is not needed to test URAM utilisation, since URAMs are significantly larger than BRAMs and because of the tiling restrictions will never achieve a better utilisation for a given logical memory than BRAMs. The decision on mapping to distributed RAM uses a simple BRAM threshold. When logical memory BRAM utilisation is lower than $1/8$ it will be mapped to distributed RAM. Otherwise, the algorithm decides between URAMs and BRAMs on a later stage. The BRAM threshold value was deduced by considering the BRAM tiling with the smallest possible depth of 512 and a width of 36 bits. The above datapath width was chosen based on our experience that 18 bits is typically not sufficient for small memories implemented in distributed RAM. If we consider a 36 bits wide logical memory matching BRAM widths, it will be mapped to distributed RAM when its depth is 64 or less. It should be noted that a depth of 64 matches a distributed RAM tiling. Comparing utilisation provides a simple metric which accounts for both, width and depth. The reason for this low utilisation requirement is that the amount of distributed RAM available is very small in comparison to the other memory resources. Additionally logic resources are also used to implement arithmetic and other user design features. As such saving logic resources is in general considered beneficial. However, in the context of static dataflow designs with many shallow FIFOs we have to use distributed RAM. For those FIFOs BRAM utilisation will be very low.

When beneficial, BRAM threshold value can be customised for each specific design depending on the overall logic utilisation. For example, when a design requires an exceptional amount of logic resources it is possible to skew the balance towards BRAMs by decreasing the threshold. As a result BRAM threshold best value is application specific; however, the setting used here achieved good results for all use cases.

After the first memory mapping stage all memories not mapped to distributed RAM are grouped by design units. Design units' memories are stored in a global list, sorted by decreasing URAM utilisation. This list is used by the memory allocation algorithm to decide if a particular logical memory should be implemented as BRAMs or URAMs.

Fig. 13 shows the algorithm in more detail. The algorithm uses a score, which is initialised to zero and is used to decide in which order the memories are allocated. The score is based on BRAM cost and used to track of the balance between BRAMs and URAMs. As a result when the algorithm selects to map a given logical memory to BRAMs the score is increased by the number of BRAMs allocated. When URAMs are selected, the score will be decreased as explained next. Since the score is based on BRAM cost we need to find a factor relating BRAM to URAM cost. This is achieved by using the ratio between the available BRAM and URAM modules. Consequently, if a logical memory is mapped to URAMs the score will be decreased by the product of this ratio and the number of URAMs needed to implement the logical memory resource. This procedure is repeated for each design unit until the corresponding list of unmapped memories is emptied. As a result our algorithm will perform best in the case of single design unit per SLR. This will avoid segmentations and the consequent suboptimal mapping resulting in slightly increased memory utilisation. However, when the HLS toolflow does not support floor planning, the above is highly unlikely and resources have to be grouped into design units based on other properties. The evaluation presented here assumes the less advantageous later option, where design units are implicitly inferred by language structures.

Within the mapping loop of the algorithm there are three possible cases which are handled separately. If the score is close to zero the allocation between BRAMs and URAMs is considered as balanced. In this case the next unmapped logical memory is picked from the beginning of the list.

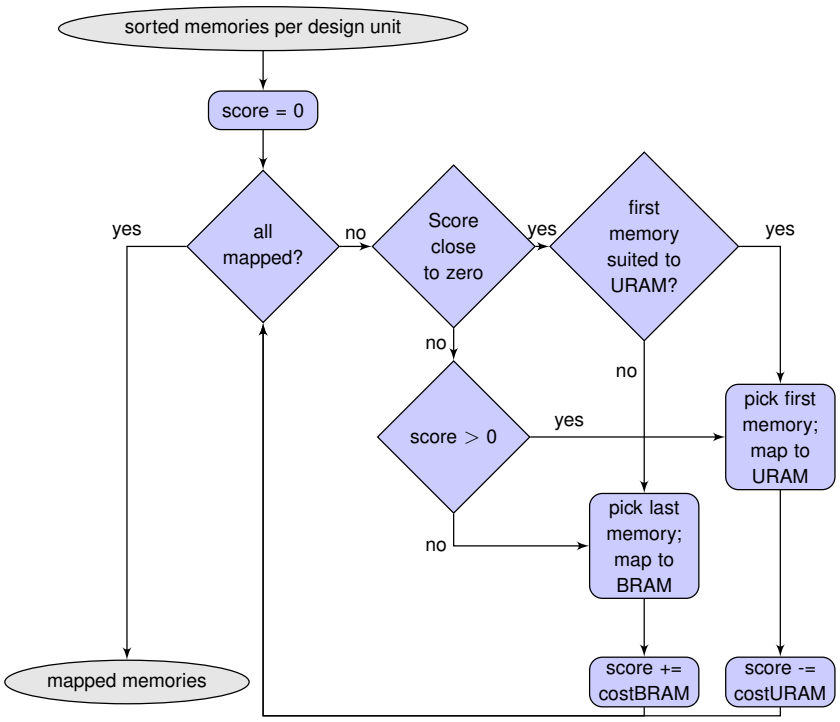

Figure 13: Maxeler's balanced memory mapping algorithm with greedy, score-based proportional mapping

Since the list is ordered by URAM utilisation, highest URAM suitability of the selected unmapped logical memory is ensured. The memory will be actually mapped to URAMs only when its utilisation is bigger than the URAM threshold. Otherwise a new memory from the end of the list, hence suited to BRAMs, is selected and mapped to BRAMs. This URAM threshold together with the score enforce when a logical memory is mapped to URAMs and can be modified by the designer. In the authors experience a URAM threshold of 0.6 provides good results. This means that logical memories with at least 60% URAM utilisation are directly mapped. The URAM threshold ensures that design units with only a single logical memory benefit from the best suited physical memory resource.

It should be mentioned that the URAM threshold in most cases has a very limited impact, due to the algorithm capability to balance allocation between BRAMs and URAMs within the same design unit. However, the URAM threshold becomes important for certain rare edge cases. These consist of designs composed of multiple design units with only one or two memories of significant size. The default value of 0.6 tries to find a good balance with the target of decreasing the overall memory wastage. However it might be necessary to adapt the URAM threshold manually in those rare edge cases.

In the case of a positive score more BRAMs than URAMs are used and again the logical memory from the beginning of the list is mapped to URAMs without considering the URAM threshold. Finally, in the last case of a negative score a memory from the list end will be mapped to BRAMs.

Each time a memory is mapped to a specific resource the score is adjusted as described above. It will be increased in the case of BRAM mapping or decreased in the case of URAMs. As a result BRAMs and URAMs are allocated at a comparable rates with respect to the overall availability.

By ordering the memories based on URAM utilisation it is ensured that always the memories most suited to URAMs or BRAMs are mapped first. This greedy strategy ensures that as many memories as possible are mapped to their best suited physical memory resource therefore saving area and addressing the first objective of the algorithm. Additionally, the second algorithm objective is fulfilled by

allocating memories to URAMs and BRAMs at the same rate and hence minimising SLR crossings, which causes routing congestion reduction and aid timing closure.

The scaling between URAMs and BRAMs as well as logic and dedicated memories can be tuned using the following functions in the `BuildConfig` of the manager:

*setBramMappingScaleFactor(**double** scale)*
*setLogicMappingScaleFactor(**double** scale)*

6 The Manager

Even though the manager is only used to instantiate and connect the different high level modules of a dataflow design, it is important to understand the underlying abstractions and paradigms to avoid wasteful resource usage and performance problems. Additionally a well designed manager can improve the maintainability of a design significantly, by allowing the easy support of other DFEs. In this section we will discuss the rules for synchronisation within a manager, discuss the memory control in detail, present solution strategies for aspect changes and finally provide suggestions for portable design practices.

6.1 Synchronisation

All components of the Manager are asynchronous. As such if a component is currently running or stalling[8] is purely a function of its inputs and outputs. Data is sent between different Manager components using FIFOs. The depth of a FIFO is 512 by default and can be modified using the *setFifoDepth()* function, which every *DFELink* object supports.

Basically Manager components start to stall if inputs are empty while they are read or outputs are fill while they are written to. However, the precise behaviour depends on the specific component. For Kernels it is enough if one input or output is empty or full, while for example the PCIe controller only stalls the stream which is empty/full. Additionally FIFOs are never filled up completely in hardware.

From a user perspective it is important to ensure, that if any data dependencies between components exist FIFOs are deep and enough to hide the latencies of the individual components. If a Kernel detects, that one input is often empty or often full it enters a so called trickling mode, which only executes every fourth cycle. As such it is important to ensure that FIFOs are never to empty or to full to avoid slower than expected performance. In the worst case a inter component latency which has been ignored might lead to a deadlock, completely stalling the execution.

The position of inputs and outputs of a Kernel are determined by the scheduler. As such, avoiding synchronisation problems sometimes requires the usage of specialised features of MaxCompiler. To identify the latency between every input and output combination of a Kernel MaxCompiler creates a *<KernelName>_IODistances.h* file in the scratch subdirectory of the build directory. Additionally, one can manually configure the latency between an input output pair using the *io.forceExactIOSeparation()* function.

6.2 Accessing LMEM

DFEs have a large memory (LMEM) resource available. This memory is implemented using DDR modules and allows to keep large amounts of data local to the board so that it can be easily accessed by the DFE. In this section we will provide a quick overview of the LMEM API and discuss best usage practices.

There is no virtual memory management or similar. Instead the user is addressing the physical memory itself. This means that the memory controller creates one monolithic memory block which is a single address space. This space represents all the memory it is physically connected to. As such the size of the memory space is defined by the amount of DDR DIMMs a memory controller is handling.

This means that the creation of the *LMemInterface* is of special importance. This interface does not only provide the functionality to connect streams to and from the LMem but it also defines the amount of connected DIMMs. This means its creation also defines capacity, port width and bandwidth. It is not

[8] Waiting to read or write data.

possible to use the same DIMM with multiple memory interfaces. The number of available DIMMs per card is board specific and can be checked in section 13.

The LMem interface to LMem in monolithic mode can be obtained via the Manager Compiler API as follows:

LMemInterface addLMemInterface()

LMem interfaces for independent LMem modules can be obtained as follows:

*LMemInterface addLMemInterface(**String** name, **int** dimmCount)*

where:

`name` is the name of the LMem module.

`dimmCount` is the number of DIMMs to be used for this LMem module.

If more than one DIMM a wider virtual DIMM is used. This means that both DIMMs are used in parallel, practically increasing the port width and therefore the bandwidth. A single DIMM has normally a port width of 64 bits. However, since the FPGA fabric can not keep up with the frequencies of the memory multiple DDR bus words are combined into a single *burst*. This burst width is 512 bits for DDR4 memory as used on the MAX5 DFEs. If multiple DIMMs are selected the combined burst width is the number of DIMMs times 512 bits. It is not possible to address memory in a smaller unit. The burst width can also be queried using the `LMemInterface.getBurstSizeBits()` API.

6.2.1 Memory Controller Architecture

The Maxeler memory controllers uses a command-based interface to the LMem. A stream to or from LMem in a Manager has a **command queue** and a **data buffer**. The memory controller reads the commands and either reads data from the data buffer and writes it to the appropriate location in memory or reads data from the appropriate location in memory and writes it into the data buffer. Each time a *LMemInterface* is created a new, independent, memory controller is created.

Figure 14 shows the architecture of the memory controller for a simple example. The example contains a kernel with two inputs and an output which are all connected to LMEM. Additionally, a LMEM read and write stream are connected to the CPU. Each of the five streams has their own memory command generator, command queue and data buffer. The **command generator** generates commands instructing the memory controller to read or write a specified amount of data to or from a specified location in the memory, and places these commands in the command queue for the stream. The **memory controller** reads commands from the queues, and, if the data buffer for the referenced stream is ready (not full for a read stream, not empty for a write stream), executes the memory operations.

Growing the number of streams to the LMem requires additional logic and can make it harder to meet timing constraints: consider multiplexing streams to and from LMem where there are streams not active at the same time.

Stalling Behaviour Each command executed by the memory controller is atomic. This means that a command is only executed if the data FIFO is ready for the complete execution of the command (full enough in the case of write, empty enough in the case of read). As a result it is possible to generate

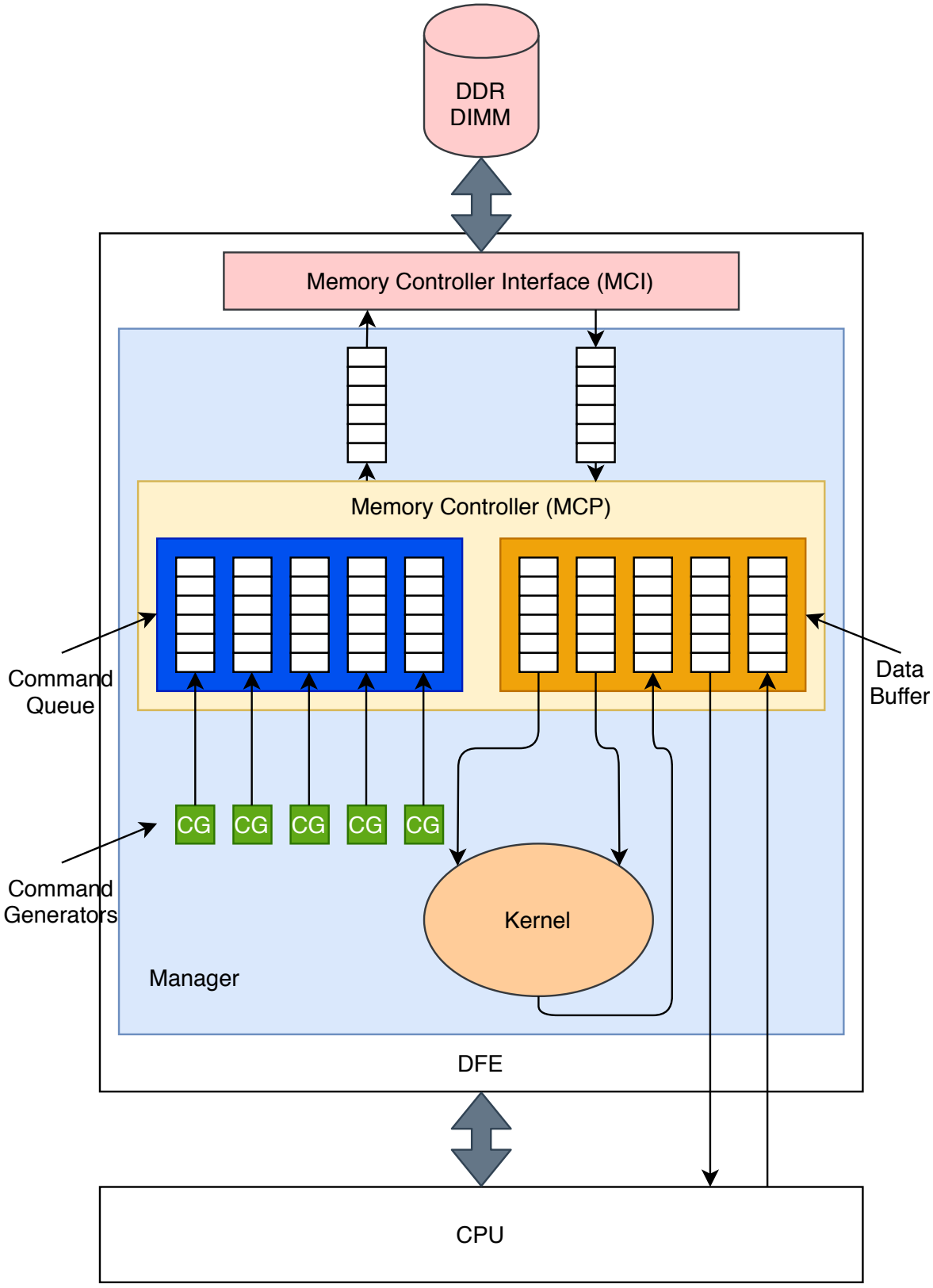

Figure 14: Memory controller architecture

all the commands that might ever be needed immediately, since they will only be executed once the accompanying data is available.

On the other side this means, that for example read commands are executed immediately (at least until the data FIFO is filled up), even if the data that is supposed to be read is not written yet. To solve these read before write and write before read issues echo streams can be used as explained in section 6.2.6.

Arbitration The memory controller supports multiple arbitration schemes to decide which commands to execute first, if multiple commands are ready for execution. The available arbitration modes are:

- `ROUND_ROBIN` Use a round robin scheme to decide which command to execute next. This is the default arbitration mode.
- `DYNAMIC` Schedule the command for the fullest write or the emptiest read FIFO first.
- `PRIORITY` The user provides an additional priority list for all the streams (using the *LMemInterface.getLMemConfig().setPriorityArbitrationList()* function. The commands are then executed based on the provided priority.
- `CUSTOM` Custom Arbitration scheme provided by the programmer.

By default the `ROUND_ROBIN` arbitration mode is used. Custom Arbitration modes can be added by extending the *CustomArbitrationImplementation* interface. The arbitration mode is selected using the *setArbitrationMode()* function on the *LMemConfig* object.

6.2.2 Memory Command Generators

By default MaxCompiler supports a few standard access patterns:

- `LMEM_LINEAR_1D` addresses the LMem module with a simple linear address.
- `LMEM_STRIDE_2D` addresses the LMem module either linearly or with a "stride", to address data at a fixed interval through the LMem.
- `LMEM_BLOCKED_3D` addresses the LMem module with a 3D address.

By default, all of the arguments for memory address generators appear in the Basic Static and Advanced Static SLiC Interface. Alternatively, the parameters can be set up through the Advanced Dynamic SLiC Interface.

The SLiC Interface automatically enables interrupts for all memory command stream generators, which are raised once all of the data has been read from or written to an LMem module. The blocking API calls will return when all the interrupts have been received. When using the non-blocking API calls, `max_wait` will similarly return when all the interrupts have been received.

It is often recommended to write custom address generators. This has two reasons:

1. They are easier to debug.
2. They provide more flexibility and can adapt to more complicated addressing patterns.

Custom address generators will be discussed in section 6.2.4.

6.2.3 Memory Command Groups

Memory streams can be grouped into a memory command group. A memory command group can either be associated with a SLiC or a custom address generator. Either way, a memory command group is always associated with a specified LMEM interface.

A memory command generator associated with a memory command group generates commands for each stream in the group that have the same access pattern but different start positions. This mode of operation is termed “pantograph” mode as the identical access pattern, with the same settings, is transcribed to multiple memory streams at different offsets.

A memory command group is created by calling the `addCommandGroup` method on the desired LMemInterface with the desired access pattern:

LMemCommandGroup LMemInterface.addCommandGroup(String name, LMemCommandGroup.MemoryAccessPattern pattern)

Memory streams are added to the control group by using versions of the `addStreamFromLMem` and `addStreamToLMem` methods on the desired memory command group:

DFELink LMemCommandGroup.addStreamFromLMem(String name)
DFELink LMemCommandGroup.addStreamToLMem(String name)

Alternatively, they are created by using versions of the `addStreamFromLMem` and `addStreamToLMem` methods on the desired LMemInterface, that take a memory command group as the second argument, rather than a memory access pattern:

DFELink LMemInterface.addStreamFromLMem(String name, LMemCommandGroup control_stream)
DFELink LMemInterface.addStreamToLMem(String name, LMemCommandGroup control_stream)

In CPU code the memory command generator settings are set per stream, with all of the same settings except a different offset into the LMem.


SLiC does not verify that all calls to `max_lmem_*` in CPU code use the same common settings.

6.2.4 Custom Memory Command Streams

In addition to the pre-configured memory command generators, a custom memory command stream can be generated. The input stream is a normal stream that can come from any source, for example from a Kernel, the CPU or even an LMem (accessed via a different command stream).

The advantage of custom command generators compared to the SLiC command generators are mostly two fold:

1. They are far more customisable. Applications that make full use of modern DFEs often have access patterns which are far more complicated then supported by the default address generators.

2. It is far easier to understand and debug the behaviour of custom address generators. For example it is possible to print the commands using *simPrintf()* in order to find out what is happening precisely.

Additionally, writing something like a linear address generator manually can be done in only a few lines of code. As such it is often recommended to only use custom memory command generators.

For custom memory command streams originating from Kernels, an API (see section 6.2.5) allows memory commands to be generated without explicit reference to their binary format. Its use is strongly recommended for new projects as it abstracts the detail of the command stream implementation, offers better performance and enables MaxCompiler to perform more compile-time analysis. Commands from other sources should conform to the specifications given in section 6.2.4.

There are several rules that apply to commands:

- Commands are not executed until the respective stream data buffer has either enough space for read streams or enough data for write streams.
- Commands are atomic operations: once a command starts, it runs to completion before the next command can be started.
- For a specific memory read or write stream, commands to that stream are executed in the order they enter their respective command queue. Commands that are waiting for space or data block the stream.
- No ordering is guaranteed between different streams either from the same control group or different control groups.

It is often simpler to generate a command stream from a separate Kernel to a computation Kernel as managing the stream control within the same Kernel can create convoluted Kernel code and has the potential to create a circular dependence between data streams in or out of the Kernel to LMem and the command stream output. For example the compute kernel might stall because the command queue to the MCP is full. The normal rule is to have one separate kernel for each command stream. If resources are scarce it is possible to use a *KernelLite* or *ManagerStateMachine* to save resources, however normally the resource usage of address generators is very small.

To associate a memory command stream with a single memory stream on the desired LMemInterface, there is a variant of the `addStreamToLMem` method:

*DFELink LMemInterface.addStreamToLMem(**String** name, DFELink control_stream)*

To create a memory command group associated with a memory command stream on an LMemInterface, there is a variant of `addCommandGroup`:

*LMemCommandGroup LMemInterface.addCommandGroup(**String** name, DFELink control_stream)*

In the design of custom memory address generators efficiency is of uttermost importance. The motivation for this can be seen in figure 15. The logarithmic x-axis in this figure describes how many bursts of memory are accessed with one command. The y-axis shows the percentage of memory bandwidth achieved compared to the theoretical maximum bandwidth. This percentage is usually referred to as memory efficiency. One can see that if only a single burst is accessed less, than 20% of the possible memory bandwidth is achievable. On the other side if one hundred bursts are accessed the efficiency increases to more than 90%. It is never possible to use the full theoretical memory bandwidth, since the DDR controller has to use some of the bandwidth to for example refresh memory cells.

To achieve good memory performance it is crucial to make any memory access as linear as possible. This is also one of the usual motivations to use multiple memory controllers (apart from potential timing closure improvements on MAX5[9]). If the architecture requires independent access to memory chunks of 3,072 bits one would need to access 2 bursts using a memory controller handling all 3 DIMMs

[9] Since DIMMs are placed on different SLRs, addressing each DIMM separately removes a lot of SLR crossings from the design

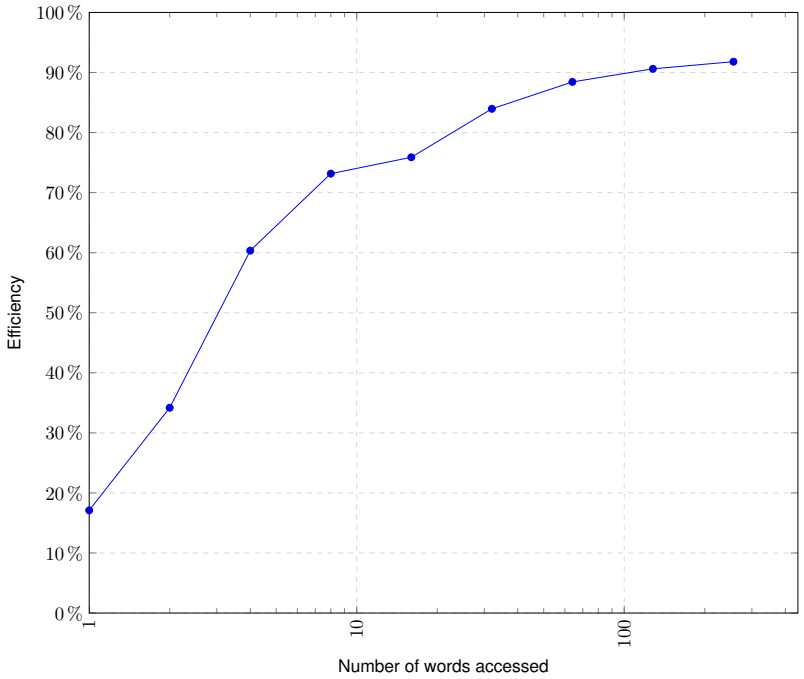

Figure 15: DDR4 memory efficiency as measured on an Alveo U250

on a MAX5C. By only using one DIMM, and therefore limiting the burst size to 512 bits, this increases to 6 bursts of data accessed at a single time. As a result, the memory efficiency is increased from roughly 35% to nearly 70%. This means that the memory bandwidth achievable can be doubled in some use cases. The projects Optimisation-chap06-example01 and Optimisation-chap06-example02 show bad and good practice regarding LMem access and bandwidth utilisation. It can not be stressed enough how important it is to do everything possible to use linear access patterns into DDR if memory bandwidth is of any concern.

A second technique to improve memory bandwidth is to distribute the accessed memory regions across the memory space. If different memory regions are accessed concurrently it is beneficial to move them apart from each other. Additionally, it accessing addresses which are multiples of powers are two is usually slightly more efficient. While these more advanced techniques further improve the achieved memory bandwidth the benefits are more limited and usually stay below 10%.

If each command issued accesses multiple bursts of memory, it is possible to run the command generate at significantly lower frequency than the remaining design. On the current MAX5 generation card the maximum memory frequency support is 1200MHz. This results in a frequency of roughly 300 MHz for the memory controller. Since the memory controller needs to run for one cycle for each burst accessed it would be more than sufficient to run the command generator at 2MHz without impacting performance if 255 bursts are accessed with each command. However, place and route tools often struggle not only with high, but also low frequencies, so setting the target frequency to less than 50 MHz rarely improves timing closure. Reducing the frequency of address generators is often especially helpful if they contain large, complicated state machines, which are not pipelined automatically.

6.2.5 Kernel API

`LMemCommandStream.makeKernelOutput` can be used within a Kernel to create an output stream that can be used as a custom memory command generator.

There are two versions of this method. The first uses separate streams for each part of the command:

void *LMemCommandStream.makeKernelOutput(**String** output_name, **DFEVar control**, **DFEVar** address, **DFEVar** size, **DFEVar** inc, **DFEVar** stream_num, **DFEVar** tag)*

The second uses a single `DFEStruct` stream containing the complete command command:

void *LMemCommandStream.makeKernelOutput(**String** output_name, **DFEVar control**, **DFEStruct** command_struct)*

The required `DFEStruct` type can be retrieved using:

LMemCommandStream.getLMemCommandDFEStructType()

The names of the arguments in the first version of `LMemCommandStream.makeKernelOutput` match the names of the fields in the `DFEStruct` for the second version. Each argument/field is described in *Table 1*. The field widths in the `DFEStruct` must exactly match those shown in *Table 1*, whereas the previous version automatically adjusts to narrower stream widths by padding as necessary. A normal kernel output can also be used to send the `DFEStruct` to the memory controller.

A command may specify multiple bursts to read or write, with either contiguous or dispersed locations. A command includes arguments for the **Command Address** (`address`), **Command Size** (`size`) and **Command Increment** (`inc`) of the operation.

The value of `stream_num` should always be 0 (1 for one-hot encoded struct) if the command is associated with an individual stream or a control group containing only one stream.

The `tag` argument instructs the memory controller to raise an interrupt once the memory command has been executed. This allows the CPU to synchronise with the DFE at the point when all the data for a stream has been written to or read from an LMem. The interrupt is only raised to the CPU once all streams, which are configured by SLiC to raise interrupt, have executed a command where tag is set to true.

It is recommended to tag only the final command in the memory command stream since only the first occurrences of tag is evaluated.

The CPU code can be set to wait on an interrupt raised by a custom memory command stream generator via the SLiC Interface in the CPU code itself:

void *max_lmem_set_interrupt_on (max_actions_t ∗actions, **const char** ∗mem_stream_name)*

A memory control command has a `stream_num` field because a memory command stream can be associated with either a single memory stream or a memory command group that can contain one or more streams.

For each memory command group, `stream_num` numbering is independent and in the order: read streams followed by write streams. The numbering of streams is determined when the Manager graph is constructed. For a memory command stream associated with a single memory stream, the stream number should refer to the 0th stream (this may require an integer value of 1 depending on whether the interface used accepts this value using one-hot encoding, which is the case for the struct).

Table 1: Memory Command Stream Components

Type	Argument/Field Name	Description	Valid Range
`String`	`output_name`	Name for the stream	n/a
`dfeBool()`	`control`	Control to enable or disable the stream in a cycle	`false`=Disabled `true`=Enabled
`dfeUInt(32)`	`address`	Start address in *bursts* for read or write command	Range depends on the size of the DIMM. Can be requested using LMemInterface.getTotalSizeBursts()
`dfeUInt(8)`	`size`	Number of *bursts* to read or write	0 to 255
`dfeInt(8)`	`inc`	Number of *bursts* added to address after each burst	-128 to 127
`dfeUInt(4)` / `dfeUInt(15)` for the struct	`stream_num` /`stream` for the struct	Selects stream to apply command to Command type (read or write) determined by memory stream type	X=0 to 14 In the case of using the struct has to be one-hot encoded
`dfeBool()`	`tag`	Tags the command to raise an interrupt if this command is executed	`false`=No Interrupt `true`=Interrupt

The stream select ID for a stream can be determined in the Manager using `getIndexForStream` with the stream name:

int *LMemCommandGroup.getIndexForStream(**String** name)*

The stream select ID is read by the memory command group to place the command in the command queue for the correct stream. The memory command group does not read any more commands from the incoming command stream if any of the command queues for the group are full.

A single command stream for multiple memory streams should only be used where the data throughput is similar for all memory streams in the group, otherwise there is a risk of the design stalling when a command buffer becomes full.

The project Optimisation-chap06-example03 shows how command buffer becomes full when single command stream is used for different throughput streams. The project Optimisation-chap06-example04 shows proper command stream usage in this case.


In general it is advisable to use custom command streams by creating the struct and output manually to get access to the full feature set of the memory controller including echo streams.

6.2.6 Echo Streams

As mentioned before, commands are always executed once they are ready. This means that once a command is executed and enough data is present in a write FIFO or a read FIFO is sufficiently empty the MCP might execute the command. In many cases this is not the desired behaviour since another stream is supposed to finish reading or writing first.

In order to provide synchronisation between multiple streams connected to the memory controller echo streams, also called command completion streams can be used. There are two modes of echo streams. In the default mode the memory controller adds an additional output stream, which replays the executed commands. This output is added once per command group. Alternatively the memory controller only sends a single bit, a token, and not the complete command to save resources.

In order to use echo streams one always has to use memory command groups. If a memory command group is created for example by using

*LMemInterface.addCommandGroup(**String** name, DFELink commandStream)*

a echo stream can be added with

LMemCommandGroup.addCommandCompletionStream()

This function returns a DFELink containing the echo streams. By setting the optional boolean parameter to true, the single bit echo stream is created.

The memory controller needs to be explicitly instructed to output something on the echo stream. This is done by setting bit 31 in the memory command. An example of setting up a command stream and enabling echo bits is shown in *Listing 2*.

Listing 2: Example of setting up a command struct and enabling echos

```
1  DFEVar address, size, interrupt, sendEcho, outputEnable;
2  // some control logic to set the previous variables
3  DFEStruct cmd = cmdType.newInstance(this);
4  cmd["address"] = address;
5  cmd["size"]    = size;
6  cmd["inc"]     = constant.var((DFEType) cmd["inc"].getType(), 1);
7  cmd["stream"]  = constant.var((DFEType) cmd["stream"].getType(), 1);
8  cmd["tag"]     = interrupt;
9
10 DFEVar cmdRaw = cmd.pack();
11 final int cmdBits = cmd.getType().getTotalBits();
12 DFEVar echoBit = sendEcho.cast(dfeUInt(cmdBits)) << 31;
13 cmdRaw |= echoBit.pack();
14 io.output("CmdOutput", cmdRaw, cmdRaw.getType(), outputEnable);
```

The resulting echo stream can be used for example in another address generator to evaluate which data has been accessed by another command generator. It is often sufficient to exploit stalling mechanics, e.g., as shown in *Listing 3*. In this case the input *EchoInput* waits to receive an echo and if no echo is present in the `DFELink` connected to this input the kernel will stall and therefore also not submit the next command. By enabling this input conditionally it is possible to synchronise the command generation with this wait condition. It is crucial to use the

*io.forceExactIOSeparation(**String** input, **String** output, **int** distance);*

method to ensure that the scheduler puts inputs and outputs at the correct position within the kernel schedule. The distance is determined by the latencies of the input and output. The example uses kernel functions to determine this, however, if a user makes a lot of changes to the kernel configuration and for examples disables input or output registering it might be worthwile to manually measure the distance. This can be done by setting it to zero, disable the generation of echos and using a maxdebug graph to identify how many ticks the memory command generator was able to do before it stalled.

Listing 3: Example of using an echo stream to stall the address generation based on echos.

```
1 DFEVar echoEnable, outputEnable;
2 DFEStruct cmd;
3 // some control logic to set the previous variables
4
5 io.input("EchoInput", dfeBool(), echoEnable).setReportOnUnused(false);
6 io.forceExactIOSeparation("EchoInput", "CmdOutput", 2 * getKernelConfig().
      getPhotonKernelConfiguration().getCEPipeling());
7 io.output("CmdOutput", cmd, cmd.getType(), outputEnable);
```

The project Optimisation-chap06-example05 shows an example when synchronization between kernels accessing LMem is needed but not implemented. The project Optimisation-chap06-example06 shows how to properly synchronize LMem access using echo streams.

6.2.7 LMem Configuration

Although the default LMem configuration is typically suitable for most users, it is also possible to configure various LMem settings for application-specific optimisations. Parameters that are commonly tuned include LMem frequency and LMem channel granularity, which influences the burst size. It is important to note that some settings can only be configured on a global level, i.e. these settings affect all LMemInterfaces in the design, while the remaining settings are configurable on a per LMemInterface level.

The global LMEM configuration object can be obtained via the Manager Compiler API as follows:

LMemGlobalConfig getLMemGlobalConfig()

This configuration object allows to configure global settings which are then applied to all memory controllers. These include the frequency to run the DIMMs and the memory controller at as well as the arbitration mode.

Similarly, a per LMemInterface configuration object can be created as follows:

LMemConfig makeLMemConfig()

LMemInterfaces which use custom configurations can be created as follows:

LMemInterface addLMemInterface(LMemConfig config)

*LMemInterface addLMemInterface(**String** name, **int** dimm_count, LMemConfig config)*

Using this configuration it is possible to configure the depth of the FIFOs used inside the memory controller, change the stall latencies, enable additional debug options but also to disable parity error correction (ECC).

In general most users will only need to adjust the frequency of the memory controller to increase performance or timing closure in some cases. However, some of the other options, e.g. deeper FIFOs

or changes to the stall latencies, can be handy to customise the memory controller to a specific design to, e.g., remove unnecessary FIFOs in the manager.

It is important to note that the config for an LMemInterface instance is fixed at the point of creation. Subsequent changes to the config do not affect previously created LMemInterfaces. This allows users to make minimal changes to an LMemConfig while being able to configure as many LMemInterfaces as possible.

For more details on available settings, see LMemGlobalConfig and LMemConfig JavaDoc documentation.

6.3 Aspect Changes

In many cases the physical port width of an interface does not match the width of the data which is sent through this port. For example one my write a single 27 bit value to LMEM on each cycle while the LMEM word with is 512 bits. If a kernel creates now an output which has a width of 27 bits which is connected to the memory controller having a port width of 512 bits an aspect change is required. This behaviour is shown in the project Optimisation-chap06-example07.

By default MaxCompiler will complain about such an aspect change and cancel the build process. By calling `setAllowNonMultipleTransitions(true)` the compiler will automatically insert an aspect change. This aspect change will find a common multiple (for this example 13,824 bits) and build a FIFO of that size. The project Optimisation-chap06-example08 shows behaviour when this solution to the aspect change problem is used. 512 27 bit words will be collected until a word is written into that FIFO and on the read size 27 512 bits words can be read from one FIFO entry. Obviously, this solution has two significant disadvantages:

1. The hardware usage is very high, due to the very wide FIFO. This will also have a negative impact on timing.

2. Only if enough data is written through this part (amount of data multiple of the large FIFO width), the design will not stall. If for our example only 10 27 bits word are supposed to be written to DDR, the aspect change will never get to a complete word which will be trasfered through the FIFO.

If the aspect change is between multiples (e.g. 64 bit to 128 bits), the compiler automatically inserts an aspect change by default. In this case a FIFO of the first port width is connected to an aspect change which is connected to a FIFO of the second width. While the hardware overhead is significantly smaller in this case, stalling conditions again have to be considered. Only if the amount of data transfered is a multiple of the bigger FIFO no stall will occur.

All of these solutions have some disadvantages. Another problem in the context of aspect changes is independent data access. For example, one might want to independently address 50 27 bit words, which are written to LMEM with a burst width of 512 bits. This means that it has to be possible to address individual data chunks of 1350 bits. Since 1350 bits is not a multiple of 512 bits if the above mentioned aspect changes would be used all data would be written directly after another leading to a continuous memory block in the memory. Accessing the required subset of data, and especially writing to it, without erasing the remaining data within that memory region will be very challenging. As a result padding would be needed, to write 1536 bits (e.g. 3 times 512 bits) to memory instead of the 1350 bits. In that case individual data access would be possible. Fig. 16 shows how the data can be packed

into 512 bit words including padding. This will allow to address different memory blocks individually in LMEM.

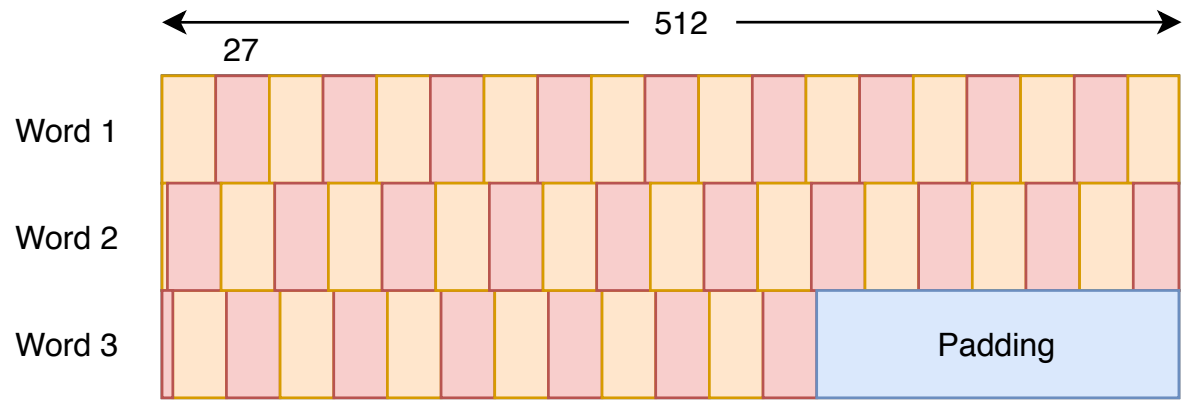

Figure 16: Packing of 50 27 bit words into three 512 bit words.

To solve all of these problems AspectChangeIO (ACIO) from MaxPower can be used. ACIO can pack or unpack arbitrary kernel types into or from frames of higher width. Data types are automatically split across multiple frames as shown in fig. 16. Additionally, it is possible to add and remove padding as required.

The API is very similar to normal kernel inputs and outputs. An ACIO object can be created using the constructor

public *AspectChangeIO(KernelLib owner,* ***int*** *portWidth)*

The port width passed in is the size of the kernel port that will be created. It should be either exactly or an multiple of the width of the port the kernel is supposed to be connected to. If, as in the previous example, the kernel is supposed to be connected to LMEM with a width of 512 bits in the best case the port width of the constructor would be 512 bits. If the data that has to be send or received in a single cycle is wider than 512 bits it is also possible to use multiples of 512 bits (e.g. 1024, 1536, ...). It is not recommended to use a smaller type (e.g. 256 bits), since this would conflict with the addition of padding.

It is now possible to create inputs and outputs by calling `input` or `output` on the ACIO object, basically replacing the normal `io` functionality of the kernel. The complete functions for input and output are

public *<T* ***extends KernelObject****<T>>* ***void*** *output(****String*** *name, T output,* ***DFEVar*** *enable,* ***DFEVar*** *finaliseIoFrame) {*
public *<T* ***extends KernelObject****<T>> T input(****String*** *name,* ***KernelType****<T> inputType,* ***DFEVar*** *enable,* ***DFEVar*** *flush)*

The `enable`, `finaliseIoFrame` and `flush` arguments are optional. `enable` has exactly the same meaning as for normal inputs and outputs, enabling or disabling the stream on a given cycle.

The padding logic is handled by the other optional parameters. For outputs padding is added on the cycle when `finaliseIoFrame` is set to true. Basically the current frame that is filled up is finished up with padding and pushed out of the kernel. On the input `flush` is used to load a new frame on the cycle where flush is set to true. This has the effect of basically discarding all the data that might be remaining in the previous frame, declaring it as padding. If the previous frame got finished on the last cycle setting `flush` high has no effect.

The project Optimisation-chap06-example09 shows proper usage of ACIO solving the problems from Optimisation-chap06-example07 and Optimisation-chap06-example08.

The Java documentation of ACIO is very detailed and will provide more details than this motivational overview. ACIO should be used every time a kernel is connected to an I/O interface of the DFE (e.g. LMEM, PCIe).

6.4 Design For Portability

In recent years the amount of platforms which can be targeted by MaxCompiler has increased significantly. For many use cases it is important to target multiple platforms at the same time, for example to scale from on-premise Alveo or MAX5C cards to cloud based Amazon EC2 F1 instances or to the edge using miniMAX5. Additionally, as new FPGAs and DFEs become available, it will be of interest to support these newer and probably faster and more capable devices.

Traditionally, portability between different hardware platforms is a significant problem, due to a multitude of challenges, including inflexible implementations, changes to vendor toolchains and low level implementation differences. With the addition of the MAX5 product line Maxeler made major changes to MaxCompiler to specifically support the portability between different platforms. In this section we will explain some of the features of MaxCompiler which assist with the usage of different platforms but also present some best practices that we validated on dozens of different use cases.

6.4.1 Best Practices for Performance Scalability

There are two different targets when porting an DFE based application from one platform to another. The first is *performance portability*, which only requires functional correctness, while the second target is what we define here as *performance scalability*. The latter aims at achieving performance close to what is theoretically possible on the new platform. Since performance portability is less demanding we will discuss it first and move to performance scalability later. It should be noted that performance portability is a necessary condition to achieve performance scalability.

In the following, eight best practices are proposed, numbered BP1 to BP8. Their use will be illustrated in Section 6.4.3.

Performance Portability MaxCompiler is in general free from any platform dependent code. Only some of the configuration set in the manager is specific to the selected platform, while in kernels and state machine the syntax does not change at all. This means that if the code is structured as suggested in the next section no significant challenges should arise from that side.

As a result if the area and memory capacity of the newly targeted DFE increases or at least stays the same it is possible to port an application to the new device without significant changes. In this case it is only necessary to deal with the changes to the I/O interfaces and introduce the required aspect changes if hardware port widths change (BP1). This is one of the reasons why it is usually recommended to always use ACIO on ports connected to the DFE I/O as mentioned in section 6.3. However, this only provides functional correctness and not good performance. It might even worsen performance compared to older devices, if I/O can not be used efficiently or high clock frequencies are not longer achievable due to architectural changes, e.g. new multi die architectures.

Moving to smaller or otherwise more restricted devices requires changes to the application to reduce the area usage. As a result techniques used for performance scalability have to be applied as described in the next section.

Performance Scalability As mentioned in section 4 it is a good practice to construct a performance model and use it to guide the development of a specific dataflow application (BP2). It can then be used to evaluate if a given target platform can facilitate the planned implementation in terms of available DFE

area, memory capacities and interconnect capabilities and lead the design space exploration. As such parts of the acceleration process might need to be repeated for each individual platform. However, the process can also help with the creation of a cost benefit analysis, if a more fundamental redesign for a new platform provides a significant advantage or not.

Most DFE applications contain a set of standard optimisations, including loop unrolling or the instantiation of multiple copies of the same computation. In these cases it is easily possible to parameterise the degree of parallelism and therefore steer hardware usage (BP3). This should be achieved by parameterising the kernel and setting the level of parallelism using the EngineParameters. By using the EngineParameter functionality it is easily possible to configure the build and saving the parameters within the *build.log* as well as adjusting the build name. Similarly data path widths can be adjusted based on the underlying hardware substrate for efficient use of e.g., hardware multipliers (BP4). By using an accurate performance model it is possible to predict the resulting area usage, memory and I/O requirements of a design point. This enables rapid design space exploration of architectural options without waiting for time consuming place and route jobs.

If the computational abilities increase faster than the available I/O bandwidths it is necessary to make further changes to the design to take advantage of this. In most designs on-chip memories are used to buffer bigger chunks of data for fast access. An example for this is tiling[10] (BP5), which is, e.g., often used in the context of linear algebra. In these cases a bigger tile size can reduce the pressure on the I/O system. If the on-chip memory capacity increased similarly to the compute capability the additional on-chip memory capacity can be used to resolve I/O bottlenecks (BP6). For example, considering the MAX4C DFE and the Alveo U200 cards, the memory bandwidth of the U200 card only increased by roughly 10% while the on-chip memory capacity increased nearly seven fold. As such it is important that the application designer makes the size of these on-chip memories configurable.

It should be noted that there are also cases where such a simple solution is not applicable, e.g., if data are truly streamed and not buffered at all. In those cases changes to the implementation are required. In the best case additional compression and decompression blocks are sufficient, which can be added without requiring significant changes to the remaining code base. As a result, it is possible to still maintain code sharing between different platforms. In the worst case a fundamental change to the architecture is required which will limit code sharing options between platforms.

Similarly, changes to the overall memory capacity including on card memory have to be considered. If the memory capacity of the new target device is big enough to hold the same amount of data as the current device no further changes are necessary. However, it might be possible to transfer a bigger working data set to the DFE if the memory capacity increases, potentially reducing communication overheads. If the capacity decreases and falls below the required capacity a domain decomposition has to be introduced (BP7). In that case the dataset is split into smaller parts which are processed individually.

An issue often encountered with big, modern Xilinx FPGAs is the internal split into separate SLRs. If SLRs are ignored, timing closure and routing congestion can become challenging. As a result it is important to limit the amount of connections between different SLRs.

In MaxCompiler, the easiest way to mitigate this is by making sure that every kernel can fully reside within a single SLR and only the connections between different kernels or kernels and I/O cross SLRs. There are two straight-forward ways to achieve this. First, it is possible to split a design into multiple kernels, where each individual kernel is small enough to not require more than one SLR (BP8.1). A second option is to create a copy of the design for each SLR (BP8.2).

In the first case it is often challenging to ensure that all individual kernels can be scalable in terms of size at a similar rate to fill up all individual SLRs. Especially if the size or count of SLRs might change

[10] Splitting one data structure into multiple smaller data structures

between current or future targeted platforms this might introduce a significant challenge to performance scalability.

However, in most cases it is possible to simply create copies of the same design for each SLR. This is especially recommended if it is possible to split the overall computation into smaller either completely or partial independent units, which require no or only limited intercommunication. This practice provides a very easy way to achieve performance scalability between different FPGAs.

Usually the size of different SLRs is very similar, however, in some cases certain SLRs might have less usable space then others, e.g., due to the need to instantiate certain IP cores close to a physical pin out. In these cases the different copies of the design can be instantiated with differing degrees of parallelism. Alternatively the first suggested approach of splitting the design into multiple separate units might be more promising. Since the use of chiplets and multiple dies becomes more prevalent in the semiconductor industry, one can expect that future platforms will also include multiple dies.

6.4.2 Tool Supported Performance Scalability

While most functional MaxCompiler code is platform independent, isolating the user from other device specific characteristics like the I/O capabilities and the FPGA architecture is not that easily achievable and often also not desirable. This includes the system architecture of the underlying device (e.g., number of logic resources and multiplier architecture) and target platform (e.g., number of memory channels and I/O bandwidth). For example, a user should be able to change data types to make use of the specific port widths of the hardware multipliers or use additional networking ports present on one platform and not the other.

In few cases the tool can efficiently hide these low level details from the designer, if efficient usage can be automated or the advantage of exposing them to the designer is minimal. One example is the mapping of logical to physical memory resources as discussed in section 5.3.1.

If user input is required to mitigate changes between devices it is crucial to limit the scope of changes and isolate these platform specific code blocks from the remaining code base. We tried to isolate these platform dependent code blocks as much as possible to the Manager.

We developed a feature-specific APIs for functionality that depends on a specific platform. Fundamentally, the API provides generic functions to create, configure and use all possible device specific functionality. This means that the device independent code can use generic API functions and platform dependent configuration can be handled separately. By providing functions to query hardware properties, e.g. port widths, it is possible to automatically adapt the platform independent code base to all possible configurations.

The new APIs for example contains the functions required to instantiate networking ports, PCIe connections and DDR interfaces but also more generic functionality for example the instantiation of computational kernels, state machines or routing blocks. While the former are highly specific to the specific platform the latter will be supported by all devices. This API definition can also deal with platform specific differences and for example report port widths or, e.g., in the case of DDR, the number of physical DIMMs available. A simplified example for the PCIe API can be seen in listing 4. The API provides platform independent functions to create PCIe interfaces to stream data to and from the host.

Listing 4: Simplified example for the ManagerPCIe API

```
1 public interface ManagerPCIe extends ManagerIO {
2   public DFELink addStreamToCPU(String name);
3   public DFELink addStreamFromCPU(String name);
4 }
```

Each platform supported in MaxCompiler implements these interfaces so that the same functions can be used. Due to this object oriented approach it is possible to write platform agnostic functions which only operate on the common APIs. An example of this is shown in listing 5. A Java interface for platform specific managers is defined which provides access to the functions required to create a compute Kernel and PCIe connections. Within this interface a default implementation is created which instantiates a kernel and connects it to PCIe and LMEM. Since the interface to LMEM itself is platform specific the interface only defines a function which provides access to the required hardware interface, which has to be implemented by the platform specific manager.

Listing 5: Isolating platform specific code using object orientation

```
1  public interface ExampleManager extends ManagerKernel, ManagerPCIe {
2
3    default void createDesign() {
4      final KernelBlock kernel = addKernel(new ExampleKernel(makeKernelParameters("
         ExampleKernel")));
5      kernel.getInput("x") <== addStreamFromCPU("x");
6      addStreamToCPU("y") <== kernel.getOutput("y");
7      LMemInterface iface = createLmemInterface();
8      kernel.getInput("z") <== iface.addStreamFromLMem("z", MemoryAccessPattern.
         LINEAR_1D);
9    }
10
11   public LMemInterface createLmemInterface();
12
13   public class ExampleU200Manager extends XilinxAlveoU200Manager implements
       ExampleManager {
14     public ExampleU200Manager(EngineParameters params) {
15       super(params);
16       createDesign();
17     }
18
19     @Override
20     public LMemInterface createLmemInterface() {
21       getLMemGlobalConfig().setMemoryFrequency(LMemFrequency.LMEM_1200MHZ);
22       return addLMemInterface();
23     }
24   }
25
26   public class ExampleMax4CManager extends MAX4CManager implements ExampleManager {
27     public ExampleMax4CManager(EngineParameters params) {
28       super(params);
29       createDesign();
30     }
31
32     @Override
33     public LMemInterface createLmemInterface() {
34       getLMemGlobalConfig().setMemoryFrequency(LMemFrequency.LMEM_800MHZ);
35       return addLMemInterface();
36     }
37   }
38 }
```

In the lines 13 to 23 a platform specific manager for the Xilinx Alveo U200 card is created. It calls the interface function to create the design and implements the function to create a DDR interface. Similarly a manager for Maxeler's MAX4C DFE can be created. It should be noted that the implementation for both cards are nearly identical and mainly consists of Java boilerplate code in this simplified example. However, the creation of the DDR interface is actually different. The reason for this is that the MAX4C DFE is based on DDR3 technology while the Alveo U200 uses DDR4. As such different memory frequencies are supported. Within the platform specific manager only supported configurations are offered, but by exploiting the object orientated API based approach to isolate these functionalities

the programmer is able to share as much code as possible between the different platforms to ease maintainability.

A similar, but slightly more manual, approach can be used to isolate other architecture specific features like native hardware multiplier sizes and I/O port widths. It is possible to define the appropriate types for each platform within the platform manager and pass them to the kernels. Similarly the kernels can query port widths from the managers and instantiate ACIO accordingly.

6.4.3 Usage Example

The techniques described in sections 6.4.1 and 6.4.2 are now applied to a financial application which was originally developed for a MAX4C DFE to show their usefulness.

Asian Option Pricing As case study, we use a financial Asian Option pricing application, which computes the expected option payoff by averaging the asset price over a given time horizon.

The evaluation of the option payoff requires an approximation, as the probability distribution of the variable defining the option payoff at valuation date has no closed-form solution. The application computes the value of an Asian Option through Curran's approximation algorithm [8] which is widely used in the financial industry due to its high accuracy. However, this is compute intensive and acceleration is desirable as financial institutions often carry out option pricing repeatedly and over very large datasets.

A dataflow architecture of the Curran approximation algorithm was previously developed for a MAX4C DFE [24] which involves accelerating the calculation of a Normal Cumulative Distribution Function (NCDF), the most compute-intensive part in the approximation model. It was found that mapping this computation to an FPGA is very resource intensive due to the reliance on floating-point computations and non-elementary functions such as exponential, square roots, and logarithms. As a first step, several algorithmic and numeric optimisations were carried out which include reordering of computations, changing the approximation model for the NCDF, and replacing floating-point with fixed-point calculations, such that an application-specific accuracy requirement of 9 decimal places at the output is satisfied.

Next, a dataflow architecture was developed which consists of a chain of five asynchronous kernels, each implementing different stages of the Curran's algorithm. It was found that stages two and four could benefit from parallelism, as each stage carries out loop computations related to the number of averaging points *n* in the option. However, attempting to fully unroll these loops is challenging because the number of averaging points is a runtime parameter that changes for different option settings. Also, for large numbers of *n*, loop unrolling will be limited due to FPGA resource constraints. Hence, we developed a parametric design where loops are partially unrolled by a factor of *k* and an arbitrary number of averaging points is supported at runtime.

The degree of parallelism *k* is a design parameter in the dataflow implementation that can be easily tweaked to maximise performance and resource utilisation according to the targeted device. When targeting a Maxeler MAX4C DFE, parallelism of $k = 13$ was found to be the best value. The accelerated Asian Option pricer was benchmarked in a Value-at-Risk application that evaluates 5,000 scenarios with 10,000 options each. When using a single MAX4C DFE, speed-ups of 278.3x and 9.2x were reported when comparing to baseline software versions that were executed on a 48 core CPU either single-threaded or multi-threaded, respectively [24].

Application Porting This application is now ported from the MAX4C DFE to four additional platforms, following three individual steps:

1. Adapt the application to the proposed manager API;

2. Apply the discussed best practices; and

3. Add platform managers for all new platforms.

In order to add platform portability to the application we first migrate the existing manager code to the new Manager API. A single platform generic function, called `createDesign`, can be created which instantiates the design by creating kernels and wiring them up with each other and PCIe. Additionally, a platform dependent function, `setupMax4`, is created which configures the Quartus toolchain for the Intel Stratix V based MAX4C DFE. This function contains up to ten lines of additional code, which can be shared between all platforms using the same generation of FPGA devices. The resulting platform manager for the MAX4C DFE is shown in listing 6.

Listing 6: The new platform specific Manager code for the MAX4C DFE

```
1 public class AsianMax4Manager extends MAX4CManager implements AsianManager {
2     public AsianMax4Manager(AsianParameters params) {
3         super(params);
4         setDefaultStreamClockFrequency(params.getFrequency());
5         setupMax4(params, getBuildConfig());
6         createDesign(params);
7     }
8 }
```

It is now possible to achieve application portability for a new platforms with minimal effort. For example the manager shown in listing 7 adds support for the Xilinx Alveo U200 card. This only requires the addition of one additional function to configure the Xilinx Vivado toolchain. In this case no further optimisations for the specific platform are performed. As a result the MAX4C and U200 design will both achieve the same performance at the same design frequency. It should be noted that apart from these few lines of manager code and the toolchain configuration the remaining code base is completely shared and no further changes are required.

Listing 7: A Manager for the Alveo U200 which achieves performance portability.

```
1 public class AsianU200Manager extends XilinxAlveoU200Manager implements AsianManager
      {
2     public AsianU200Manager(AsianParameters params) {
3         super(params);
4         setDefaultStreamClockFrequency(params.getFrequency());
5         setupUltrascale(params, getBuildConfig());
6         createDesign(params);
7     }
8 }
```

The build target selection between both platforms can be handled as a command line argument for the build script. We now build both designs unrolling the same number of loop iterations by setting the parallelism factor to $k = 13$. The results can be seen in tab. 2. At the same frequency the design on the MAX4C and the Alveo U200 achieve the same performance, fulfilling the target of performance portability. Furthermore, it is possible to increase the frequency from 200 MHz to 350 MHz which results in a 1.74x speedup.

As a next step we apply the previously discussed best practices to the code base to achieve a further speedup. The first step is to create the performance model for the application (BP2). For this application no DDR memory access is required and we can fully focus on the PCIe bandwidth and area usage.

The area usage can be predicted by counting the number of operations for the design and running micro applications to get figures for the area usage for each individual operation. In this case all additional platforms are based on the MAX5 platform so the area usage will be constant across all of them.

Table 2: Comparison of the designs for the different target platforms. Speedup is provided relative to the MAX4C implementation. The resource usage percentage is based on the available resources of the given FPGA. In the case of F1 the AWS shell is excluded.

Platform	Design Copies	Parallelism k	Frequency	Time (s)	Speedup	Logic (%)[11]	DSPs (%)	BRAMs[12](%)	URAMs (%)
MAX4C	1	13	200	15.13	1x	258,209 (98.40%)	1,436 (73.15%)	1,205 (46.94%)	-
U200	1	13	200	15.13	1x	214,569 (18.15%)	2,628 (38.42%)	933 (21.60%)	59 (6.15%)
U200	1	13	350	8.69	1.74x	214,611 (18.15%)	2,628 (38.42%)	933 (21.60%)	59 (6.15%)
U200	3	10	350	3.79	3.99x	535,374 (45.28%)	6,372 (93.16%)	2,157 (49.93%)	189 (19.69%)
U250	4	14	350	2.04	7.42x	874,035 (50.58%)	11,184 (91.02%)	3,471 (64.56%)	323 (25.23%)
MAX5C	3	10	350	3.91	3.87x	536,143 (45.35%)	6,372 (93.16%)	2,164 (50.09%)	189 (19.69%)
F1	1	18	250	8.62	1.75x	423,501 (35.82%)	3,471 (50.75%)	1,506 (34.86%)	170 (17.71%)

[1] In the case of the MAX4C (Intel Stratix V) we count ALM usage, while in all other cases (Xilinx Ultrascale+) we count LUT usage.
[2] For MAX4C we count M20Ks usage, while in all other cases (Xilinx Ultrascale+) we count BRAM usage. There is no URAM equivalent for MAX4C.

After creating the performance model we determined, that we can calculate the DSP and LUT usage as shown in eq. 27 and eq. 28 respectively.

$$dsps = 444 + k \times 168 \tag{27}$$

$$luts = 63{,}482 + k \times 9{,}181 \tag{28}$$

In terms of I/O bandwidth we can analyse the amount of data that has to be transmitted for each scenario and each option. Per option 67B and per scenario 654KB have to be sent from the host to the FPGA. Additionally, 8B have to be read from the accelerator per option. Since we run 5,000 scenarios on 10,000 options this means that in total 3.12GB of data have to be transmitted. This can be achieved in roughly one second using PCIe Gen2 x8 or in a quarter of that time using PCIe Gen3 x16 at slightly higher hardware costs.

The number of cycles required for the calculation can be computed as shown in eq. 29. Using this equation it is straight forward to derive the compute time for a given frequency.

$$n_{cycles} = n_{scenarios} \times \left\lceil \frac{n_{average\_points}}{k} \right\rceil \times n_{options} \tag{29}$$

Using this information we can further optimise the design for the additional platforms. As a first step we can ensure that all data are properly aligned, if the width of the PCIe bus changes. To accomplish this we pass the PCIe port width to the kernels communicating with the CPU and add an automatic aspect change (BP1). Additionally we change the CPU code to allocate data accordingly.

The second step is to make efficient use of multiple SLRs. As discussed in section 6.4.1 there are two fundamental strategies to deal with this problem. One can either try to split the design into equally sized parts (BP8.1) or replicate the design multiple times (BP8.2).

In the case of the Asian Option pricing application there are good arguments for the application of both solution strategies. Since there is a constant amount of resources used, which is independent of the unrolling factor keeping a single design and only unrolling the loop more is more resource efficient. However, a very high unrolling factor results in low hardware utilisation, if only a limited number of average points are used. The reason for this is that each loop iteration deals with one average point. As an example if k is set to 29 and we calculate 30 average points we would need to make two passes where only one of the 29 hardware pipelines is used in the second pass. Also unrolling is only applied in two out of the five kernels. As a result of this it will not be possible to distribute the design proportionally across the bigger DFEs, e.g., the Xilinx Alveo U250.

Creating multiple copies of the design can be considered as an overhead. Since DSPs are the limiting resource, in this case each additional design instance uses hardware resources which could

otherwise be used to unroll 2.5 additional loop iterations. We decided to create multiple design instances (BP8.2), since the overhead is small in comparison to the expected benefits in terms of achievable frequency and ease of development.

To implement this optimisation we have to further modify the manager code. We need to call the `createDesign()` function in a loop to create multiple instances of the design. Additionally we have to change the naming of kernels and interfaces accordingly, to ensure that all names are unique. As a result it is now possible to configure the number of copies via a simple command line argument for the build environment. The result of these changes can be seen in listing 8.

Listing 8: The manager for the Alveo U200 achieving performance scalability.

```
1  public class AsianU200Manager extends XilinxAlveoU200Manager implements AsianManager
       {
2      public AsianU200Manager(AsianParameters params) {
3      super(params);
4          setDefaultStreamClockFrequency(params.getFrequency());
5          setupUltrascale(params, getBuildConfig());
6          for (int i = 0; i < params.getDesignCount(); i++) {
7              createDesign(params, i);
8          }
9          addMaxFileConstants(params, 16);
10     }
11 }
```

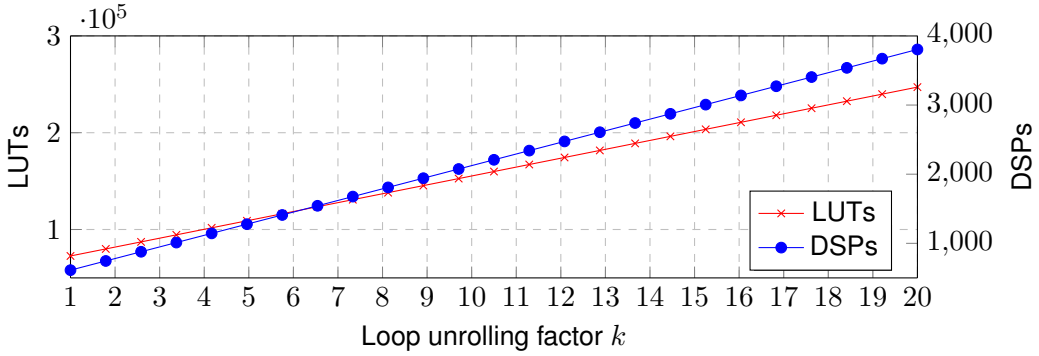

Figure 17: LUT and DSP usage predicted by the performance model for one design instance implemented on Xilinx Ultrascale+ technology.

In this case it is also necessary to modify the CPU code to split the overall workload into multiple parts, which are equally distributed onto the different copies of the design. Using MaxCompiler it is possible to add C/C++ preprocessor defines to an automatically generated header file which is needed to integrate the bitstream into the host application. This allows us to synchronise the design count automatically between bitstream and CPU code, by passing the number of design instances to the CPU code which can then automatically distribute the workload accordingly. In listing 8 the `addMaxFileConstants` function fulfills this role. As a result the same CPU code base can be used for all platforms.

At this point we are able to build performance scalable designs for the current platforms targeted, MAX4C and Alveo U200. We can expand this to the other targets, Alveo U250, MAX5C and Amazon F1, by adding the managers accordingly. No further changes to the code base are required.

To rapidly perform design space exploration we can use the previously developed performance model (BP2). Fig. 17 shows the area usage in terms of LUTs and DSPs for a single design instance dependent on the loop unrolling factor k (BP3). Eq. 27 and 28 show the area usage in terms of DSPs

and LUTs for a single design instance dependent on the loop unrolling factor k (BP3). This can be seen as the area usage for a single SLR and since SLRs have approximately the same size it can be used to determine the most reasonable values for k. It is now only necessary to try these values and figure out the highest achievable frequency by running place and route. Even though we still have to run multiple place and route runs per targeted platform, overall the possible number of required runs is decreased significantly by using the performance model. For the valid design points for this application and the targeted platforms the I/O bandwidth is no bottleneck, so we do not need to apply further optimisations to change the compute to communication ratio.

The achieved performance and area utilisation for all designs are shown in tab. 2. We use MaxCompiler 2019.1, Vivado 2018.3 and Quartus 13.1 and run 5,000 scenarios on 10,000 options as our benchmark. In most cases we manage to use a significant part of the FPGA resources while still achieving timing closure at high frequencies. In the case of the MAX4C DFE more than 98% of the logic resources are used and timing is met at 200 MHz. For the MAX5C DFE as well as the Alveo U200 card the DSP usage is the highest at more than 93% and the design meets timing at 350 MHz achieving a speedup of nearly 4x compared to the MAX4C design. Porting from the MAX5C DFE to Alveo U200 does not require any architecture changes and delivers almost the same performance, since the platform architecture is very similar. In the case of the Alveo U250 we manage to use more than 91% of the DSPs at a frequency of 350 MHz resulting in a speedup of 7.4x compared to the MAX4C baseline.

In the case of the Amazon EC2 F1 instance the resource usage and performance improvement is smaller. This is due to the presence of the AWS shell which takes up some of the chip resources in two of the three SLRs. There are two potential options to achieve good performance in this case. It would be possible to use different degrees of parallelism on the individual SLRs. However, the overhead introduced by the part of the application which does not benefit from further parallelisation is unjustifiable large for the SLRs containing the AWS shell. The second option is to not create multiple instances of the design, but only one with a higher parallelism (BP8.1). In the cases of the other cards it is difficult to split the five dataflow kernels of the Asian Option pricing application across three or four SLRs of the same size. However, here we can make use of the fact that most of the area is used in two kernels while the others remain smaller. As a result the tool can spread the design a lot better across the three available SLRs. We can utilise our existing code to create this bitstream by instantiating only a single design instance with a parallelism factor of $k = 18$. The resulting design achieves a speedup of 1.75x compared to the initial MAX4C design. The lower frequency achieved for this design can be traced back to the fixed interconnect to the AWS shell.

An example of the EngineParameter class for this application is shown in *Listing 9*. By overriding the `getBuildName()` function one can modify the build name based on the parameters selected. This also enables to build multiple different configurations of the same application in parallel and easily identify the resulting builds by the directory name. Additionally, *Listing 10* shows the configuration for the individual platforms as well as how one can select the target to run using EngineParameters.

It should be stressed that we did not need to modify any kernel code to achieve these results and only the manager code was changed. Furthermore, the required code size per individual platform is in the order of tens lines of code. As such maintaining this application for current and future platforms is greatly simplified without degrading performance. Building for different platforms is achieved by simply passing different command line parameters to the build system which can be evaluated by the EngineParameters.

Listing 9: The EngineParameter class for AsianOptionPricing

```
1 package asian;
2
3 import com.maxeler.maxcompiler.v2.build.EngineParameters;
```

```
4
5  public class AsianEngineParameters extends EngineParameters {
6
7    public enum SupportedBoards {
8      MAX4C,
9      MAX5C,
10     F1,
11     U200,
12     U250;
13   }
14
15   // Design frequency.
16   private static final String s_frequency = "Frequency";
17   // Number of design instances.
18   private static final String s_designCount = "DesignCount";
19   // Number of pipes
20   private static final String s_nPAR = "nPAR";
21   // Automatically add SLR constraints to the manager components.
22   private static final String s_lockToSlr = "LockToSLR";
23   // Pipelining between the PCIe controller and the manager components.
24   private static final String s_pciePipeline = "PCIePipeline";
25   // Insert StreamStatus blocks for debugging.
26   private static final String s_streamStatus = "StreamStatus";
27   // Board targeted. E.g. MAX5C
28   private static final String s_targetBoard = "TargetBoard";
29   // String attached to the maxfile name.
30   private static final String s_tag = "tag";
31
32   public AsianEngineParameters(String[] args) {
33     super(args);
34   }
35
36   @Override
37   protected void declarations() {
38     declareParam(s_frequency, DataType.INT, 150);
39     declareParam(s_designCount, DataType.INT, 3);
40     declareParam(s_nPAR, DataType.INT, 10);
41     declareParam(s_lockToSlr, DataType.BOOL, false);
42     declareParam(s_pciePipeline, DataType.INT, 3);
43     declareParam(s_streamStatus, DataType.BOOL, false);
44     declareParam(s_targetBoard, DataType.ENUM, SupportedBoards.MAX5C);
45     declareParam(s_tag, DataType.STRING, "");
46   }
47
48   @Override
49   public String getBuildName() {
50     String name = getMaxFileName() + getTarget() + "_f" + getFrequency() + "_DC" +
           getDesignCount() + "_nPAR" + getPipes()
51         + "_SLR" + bool2Str(doLockToSlr()) + "_PCIeP" + getPciePipeline() + "_SS" +
               bool2Str(hasStreamStatus())
52         + "_" + getTargetBoard().toString();
53     if (getTag().compareTo("") != 0) {
54       return name + "_" + getTag();
55     } else {
56       return name;
57     }
58   }
59
60   private String bool2Str(boolean val) {
61     return val ? "t" : "f";
62   }
63
64   public int getFrequency() {
65     return getParam(s_frequency);
66   }
67
68   public int getDesignCount() {
69     return getParam(s_designCount);
```

```
70    }
71
72    public int getPipes() {
73      return getParam(s_nPAR);
74    }
75
76    public boolean doLockToSlr() {
77      return getParam(s_lockToSlr);
78    }
79
80    public int getPciePipeline() {
81      return getParam(s_pciePipeline);
82    }
83
84    public boolean hasStreamStatus() {
85      return getParam(s_streamStatus);
86    }
87
88    public SupportedBoards getTargetBoard() {
89      return getParam(s_targetBoard);
90    }
91
92    public String getTag() {
93      return getParam(s_tag);
94    }
95  }
```

Listing 10: The configuration for different platforms and Manager selection

```
1   default void setupMax4(AsianEngineParameters params, com.maxeler.platform.max4.
        manager.BuildConfig buildConf) {
2     setup(params);
3
4     buildConf.setBuildEffort(Effort.VERY_HIGH);
5     buildConf.setOptimizationGoal(OptimizationGoal.SPEED);
6     buildConf.setEnablePhysicalSynthesis(true);
7     buildConf.setMPPRCostTableSearchRange(params.getMPPRStartCT(), params.getMPPREndCT
          ());
8     buildConf.setMPPRParallelism(params.getMPPRThreads());
9     buildConf.setMPPRRetryNearMissThreshold(params.getMPPRRetryThreshold());
10    buildConf.setEnableTimingAnalysis(true);
11  }
12
13  default void setupMax5(AsianEngineParameters params, com.maxeler.platform.max5.
        manager.BuildConfig buildConf) {
14    setup(params);
15
16    buildConf.setImplementationStrategies(getImpStrategies(params));
17    buildConf.setParallelism(params.getMPPRThreads());
18    buildConf.setImplementationNearMissRetries(2);
19    buildConf.setImplementationNearMissThreshold(2000);
20  }
21
22  default void setup(AsianEngineParameters params) {
23    AsianKernel.nPAR = params.getPipes();
24  }
25
26  static ImplementationStrategy[] getImpStrategies(AsianEngineParameters params) {
27    ImplementationStrategy[] strats = {ImplementationStrategy.MAXELER1,
          ImplementationStrategy.MAXELER2,
28        ImplementationStrategy.MAXELER3, ImplementationStrategy.MAXELER4,
29        ImplementationStrategy.PERFORMANCE_EXPLORE, ImplementationStrategy.
              PERFORMANCE_NET_DELAY_HIGH,
30        ImplementationStrategy.CONGESTION_SPREAD_LOGIC_HIGH, ImplementationStrategy.
              CONGESTION_SPREAD_LOGIC_EXPLORE,
31        ImplementationStrategy.AREA_EXPLORE_WITH_REMAP, ImplementationStrategy.
              PERFORMANCE_EXTRA_TIMING_OPT,
```

```
32        ImplementationStrategy.PERFORMANCE_NET_DELAY_LOW};
33    return Arrays.copyOfRange(strats, params.getMPPRStartCT() - 1,
34        params.getMPPREndCT() == 1 ? strats.length : params.getMPPREndCT());
35  }
36
37  default void addMaxFileConstants(AsianEngineParameters params, int pcieWidthInBytes)
        {
38    addMaxFileConstant("DESIGN_COUNT", params.getDesignCount());
39    addMaxFileConstant("N_PAR", params.getPipes());
40    addMaxFileConstant("PCIE_WIDTH", pcieWidthInBytes);
41  }
42
43  public static void main(final String[] argv) {
44    final AsianEngineParameters params = new AsianEngineParameters(argv);
45
46    final AsianManager manager;
47    if (params.getTargetBoard() == SupportedBoards.MAX5C) {
48      manager = new AsianMax5Manager(params);
49    } else if (params.getTargetBoard() == SupportedBoards.F1) {
50      manager = new AsianF1Manager(params);
51    } else if (params.getTargetBoard() == SupportedBoards.U200) {
52      manager = new AsianU200Manager(params);
53    } else if (params.getTargetBoard() == SupportedBoards.U250) {
54      manager = new AsianU250Manager(params);
55    } else if (params.getTargetBoard() == SupportedBoards.MAX4C) {
56      manager = new AsianMax4Manager(params);
57    } else {
58      throw new IllegalArgumentException("Board not supported!");
59    }
60    manager.build();
61  }
```

7 Pipelining

MaxCompiler automatically pipelines all kernels. The motivation for this is, that when dealing with many operations, as they are common for big dataflow graphs, it is very complicated and tedious to manually insert delays into the computational pipeline. Additionally, it would be necessary to adjust the pipeline every time, once changes to the computational structure are made. Pipelining is crucial to reduce the signal propagation delay and therefor increase the achievable frequency.

Fundamentally, when dealing with the amount of pipelining there are two competing objectives. On the one side we want to improve the achievable frequency, by reducing signal propagation delays between registers. These signal propagation delays consist of the routing delay, caused by moving the signal through the FPGA fabric, between hardware units and logic delays caused by the times signals need to propagate through those hardware units itself. On the other side we always want to reduce the area usage. Since the addition of pipelining stages increases the area usage, sometimes it is necessary to remove pipelining stages. The area costs are direct, by the need to add additional registers and increase FIFOs due to the additional pipeline stages, but also indirect through less efficient packing of functions into LUTs.

Since increase area usage can cause increased routing congestion and therefor routing delays it is possible that the addition of more pipelining stages actual hurts timing more than it improves it. These competing advantages have to be balanced in order to achieve maximum performance. Even though MaxCompiler pipelines and schedules kernels automatically there are multiple opportunities for a user to adjust the decisions made by MaxCompiler for a specific design.

The most basic function MaxCompiler offers to change the pipelining is

*<T **extends KernelObject**<T>> T **optimization**.pipeline(T v)*

This function takes a signal as an argument and returns the same signal with an additional pipelining stage. This function is usually used after a timing report (see section 9) is examined and a timing path between specific operations in the dataflow graph is discovered. For example one might need to route a signal from a multiplier into memory. In this case often a long routing delay is required, since DSPs and on-chip memories are in different columns of the FPGA (see section 2). By adding an additional register in this path this part can be split into two, removing the frequency bottleneck.

Another way to impact the pipelining between nodes is graph pipelining. It is possible to change the graph pipelining using the following functions:

***void optimization**.pushGraphPipelining(**double** pipelining, PipelinedOps op)*
***void optimization**.popGraphPipelining(PipelinedOps op)*

The graph pipelining functionality allows to automatically add additional registers within a dataflow graph based on the number of operations the signal passed through. The function creates a scope using push and pop methods in which a settings change is valid. The default graph pipelining factor is always zero. The `PipelinedOps` enum defines to which types of operations the setting applies. The value of the top of the stack modified using the push and pop functions is applied for each of the operations in the dataflow graph. Valid enum values are:

- `FLOAT_ADDSUB` Floating point additions and subtractions.
- `FLOAT_MUL` Floating point multiplications
- `FLOAT_DIV` Floating point divisions
- `FLOAT_COMPARE` Floating point comparisons

- `FLOAT_SQRT` Floating point square root
- `FLOAT_CAST` Floating point casts
- `Fix_ADDSUB` Fixed point additions and subtractions
- `Fix_MUL` Fixed point multiplications
- `Fix_DIV` Fixed point divisions
- `Fix_COMPARE` Fixed point comparisons
- `Fix_CAST` Fixed point casts
- `LOGICAL` Logical operations, e.g., and, or and xor
- `MISC` Remaining operations, e.g., MUXs
- `ALL` All operations

The value of the pipelining factor gets accumulated while traversing through the graph. Every time the accumulated pipelining factor exceeds one a register is added and the accumulated value is reduced by one. All positive values, including zero, are a valid setting for the pipelining factor. If for example a pipelining factor of two is set two registers are set after each operation. If two subgraphs merge the maximum accumulated pipelining factor is carried forwards. An example of this is shown in fig. 18.

Additionally to graph pipelining, there also exists node pipelining, which operates on a per node basis. Node pipelining can be controller using the following functions:

void optimization.*pushNodePipelining(**double** pipelining, PipelinedOps op)*
void optimization.*popNodePipelining(PipelinedOps op)*

Fundamentally, the API works in the same way as for graph pipelining. The idea of node pipelining is that it controlls how deep a single operation in a dataflow graph is pipelined. Many operations, including nearly all floating point operations as well as fixed point multiplications and divisions, have a latency of multiple cycles. The pipelining factor selects the number of pipelining stages selected of a linear scale between zero and one, where zero represents the smallest latency supported and one the maximum. The default pipelining factor is one.

If a operation only has a maximum latency of one, the semantic of the pipelining factor is a bit more complicated. For logic operations any pipelining factor not equals to one means that all pipelining is removed. For more complicated operations, e.g. additions and subtractions, only in the case of a pipelining factor of exactly zero the pipelining is really removed.

It is often beneficial to combine both node and graph pipelining, e.g., to remove all node pipelining on logical operations but automatically add pipelining after every third logical operation. This is often used in logic heavy applications.

One additional method to automatically add registers to the dataflow graph is to limit the fanout. The term fanout describes to how many other nodes in the dataflow graph the output of a single node is connected. The most significant disadvantage of a high fanout is the increased local routing congestion resulting in worse timing characteristics. To mitigate this problem one can limit the fanout to add registers, which can be used to distribute the signals across the chip.

The function

*<T **extends KernelObject**<T>> T **optimization**.limitFanout(T v, **int** maxFanout)*

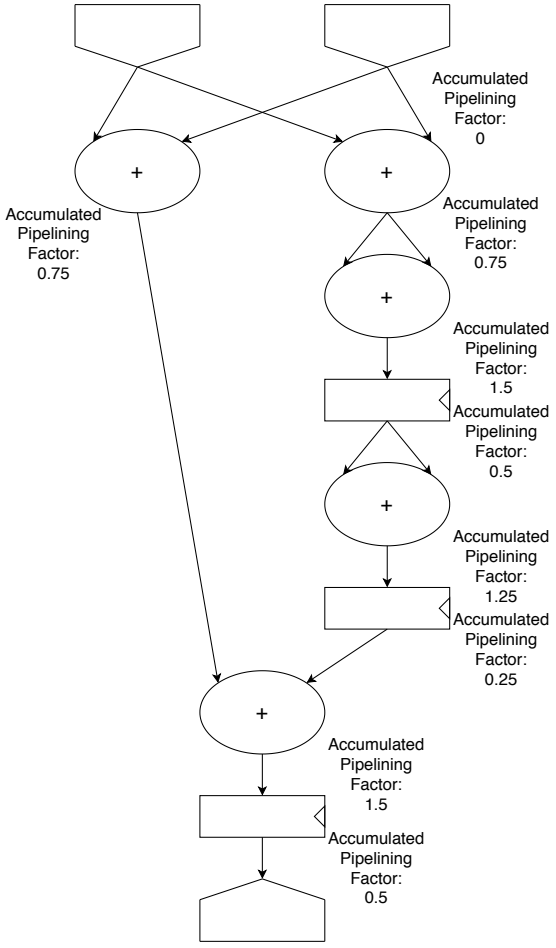

Figure 18: Graph pipelining of a dataflow graph with a graph pipelining factor of 0.75.

can be used to limit the fanout of a specific signal. The limit set using `maxFanout` describes the maximum of outgoing signals to each of the individual registers created by the fanout limiter. If the total fanout is smaller than `maxFanout` no additional register is added.

Additionally the functions

void *pushFanoutLimit(**int** fanoutLimit)*
void *pushFanoutLimit(**int** fanoutLimit FanoutLimitType fanoutType)*
void *popFanoutLimit(**int** fanoutLimit)*

can be used to modify the default fanout limit within a push pop scope. By default the fanout is limited to 32. It is possible to select how the fanout limitation is applied using the `fanoutType` parameter.

Two modes are supported. By default `CHAIN` is used, which builds a chain of registers. This means that, as new outputs are added, a new register is added every time the current register has as many outputs connected as defined as maximum. This leads to a higher latency, but is very helpful, if large distances on the chip have to be crossed and many resources are used. It is for example a natural way to route a signal along multiple DSP columns, if for a example one parameter is used in many multiplications. The second option is called `TREE`. Here the registers are arranged as a balanced tree, resulting in a smaller pipeline depth.

It is important to keep in mind that scheduling is only applied to kernels. State machines have to be manually scheduled, if any kind of additional pipelining is required.

7.1 Scheduling in MaxCompiler

Scheduling is used to assign the nodes of the graph to specific cycles. The purpose of this is, that the signals are delayed sufficiently to create the described behaviour while considering the latency of the nodes. The latency of the node is determined by the operation it represents, the specific hardware and timing constraints targeted. MaxCompiler usually tries to build a pipeline as deep as possible in order to facilitate timing closure at higher frequencies resulting in nodes with high latency.

Fig. 19 shows an unscheduled dataflow graph implementing the function $out = (x+y+z)+(x+y)$. The equivalent scheduled graph is shown in fig. 20. A FIFO is inserted to delay the result of $x + y$ to add to the partial sum of $x + y + z$. Depending on the latency of the addition operation the depth of the FIFO will change. For floating point additions this will usually be in the order of ten cycles, while fixed point additions only have a latency of a single cycle. The FIFO is usually implemented in on-chip memory or Flip Flops and the schedule is automatically created and applied by MaxCompiler.

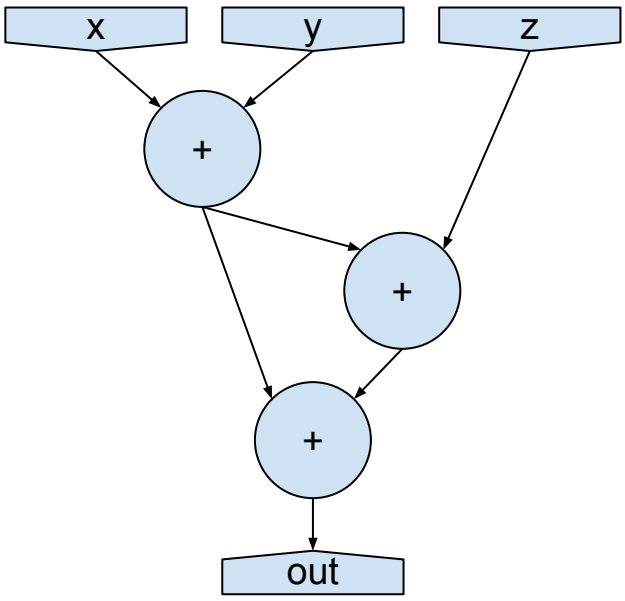

Figure 19: Unscheduled simple dataflow graph.

The scheduler used by MaxCompiler is based on integer linear programming. A set of equations is used to describe latencies and conditions for functional correctness. Additionally a cost function is generated by adding the number of bits required for all FIFOs. By minimising the cost function a schedule is obtained, which can subsequently be applied to the dataflow graph using a graph pass. Alternatively it is possible to use an ASAP scheduler, which places each node as early as possible in the schedule. This scheduler is mainly intended for the use in KernelLites to avoid potential issues arising around stall scopes. However, the ASAP scheduler can also be used in normal kernels.

In some cases scheduling resource usage can occupy up to a third of the total on-chip memory capacity and in general will require a significant amount of memory resources. In order to reduce the resource usage of scheduling the dataflow graph has to be reordered in order to create FIFOs which either are slimmer or shallower. However, it is very hard to decide which changes might accomplish this.

One of the major reasons for this difficulty is, that the dataflow graphs of current state of the art applications can contain multiple hundred thousand nodes. It is not possible for the human mind to fully visualise and process this dataflow graph. To address this problem MaxCompiler contains tool support

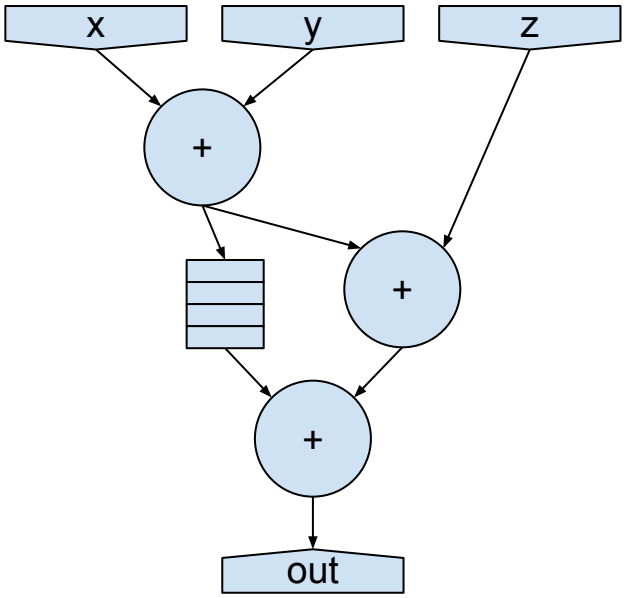

Figure 20: Scheduled simple dataflow graph.

to assist the designer into gaining insight into where FIFOs are placed and why they might occur. Since the amount of FIFOs created in total might also be in the order of multiple thousands it is also crucial to provide an overview of the FIFOs causing the highest resource usage.

7.1.1 Scheduling Visualisation

MaxCompiler creates graphs which capture the most important information about scheduling resource usage. This graph uses source code lines or full stack traces as nodes and the edges represent FIFOs connecting these source code positions. Since the code size is usually very limited and most lines do not create FIFOs, this automatically limits the node count to a significantly smaller number.

The costs for the FIFOs are estimated in terms of BRAMs. Obviously this estimation will not perfectly predict the final resource usage, since it does not consider mapping to other memory resources, like URAMs on newer Xilinx FPGAs or Flip Flops and LUT ram. However, it provides a good indicator for the final resource usage. We additional group all FIFOs which are placed between the same lines of code together and report the resource usage combined. This further reduces the complexity of the graph and helps to gain a quick overview.

The resulting graph is created in two versions. One is providing all the details, e.g., the full stack trace, the depth and width of the FIFO as well as node IDs to allow backtracking in the dataflow graph generated by MaxCompiler. The other version only lists the source line at the top of the stack trace and the accumulated costs of the FIFO. Additionally FIFOs which have higher resource usage are colour highlighted.

The advantage of the more complex graph is that it provides a lot more insight and also keeps track if the same function is called from multiple different origins. In the simplified version of the graph these cases would be collapsed into the same node. As a result we produce a graph for a first, quick overview and a second for deeper insight.

A complex scheduling graph can be seen in fig. 21. It is created for the dataflow graph shown in fig. 22. The node containing the full stack trace contains the function name, which in this case is `<init>` since the FIFO is generated within the constructor of the kernel. Additionally the full kernel name

including package, the source file and the line number is shown. The FIFO representation includes the total costs and a detailed list of all FIFOs. In this simplified case the list only contains one entry.

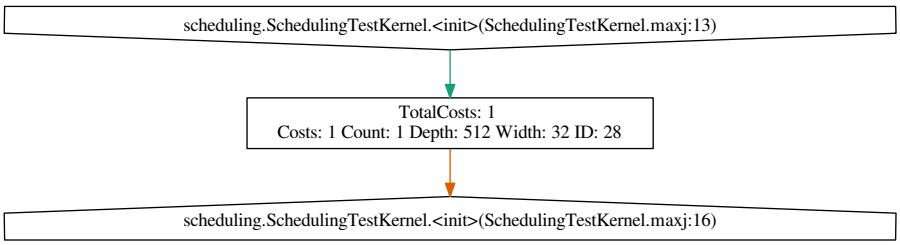

Figure 21: Graph showing the FIFOs in a simple design.

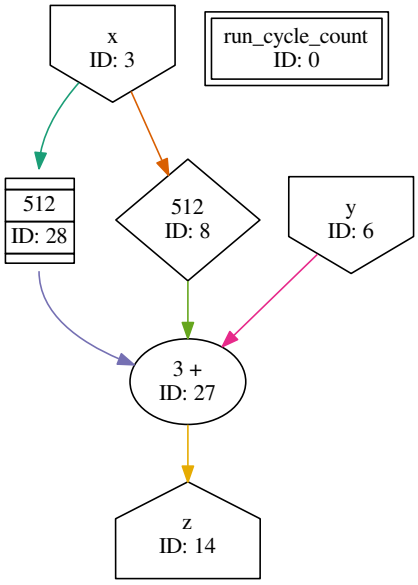

Figure 22: Dataflow graph for the scheduling graph shown in fig. 21.

The more simplified version of the same scheduling graph is shown in fig. 23. In this case only the source file and line number for the location is shown. Similarly for the FIFO only the total cost is reported.

The purpose of this simplified graph might not be obvious for this use case. However, examples like the graph presented in fig. 24 motivate the simplified graph. In this case the simplified graph for a real dataflow kernel is shown. Even though it is still rather big, due to the highlighting of problematic FIFOs as well as sources and sinks the programmer can quickly spot, where optimisations could yield the biggest benefit.

A small section from the more complex scheduling graph for the same application is shown in fig. 25. While this graph contains more information it might still be overwhelming to start with.

The graphs can be found in the `/scratch` directory of the build directory. The name is build from the kernel name as well as a "_FIFOReport.dot" and "_FIFOReportSimple.dot" postfix, where the first file is the complex and the second the simple scheduling graph. These files can be rendered into images using the graphviz command line tools. For example to render them into a vector graphic the following line can be used:

```
dot -Tsvg <filename>.dot > <result>.svg
```

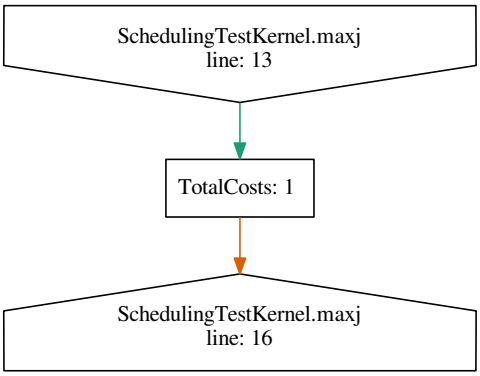

Figure 23: Simplified graph showing the FIFOs in a simple design.

The resulting file can be inspected using normal image viewing programs. Maxeler recommends *Inkscape* since it can cope with large graphs.

Additionally we create a small text report, which contains the top 10 biggest FIFOs as well as sinks and sources. This text report provides a first quick summary to the programmer and can be used to get a quick overview as well as idea which parts of the codebase to tackle first. It should be mentioned, that all of these reports are generated before the vendor toolchain is run. As such all resource costs are only estimations. However, it only takes seconds or a few minutes from the start of the compilation to obtain the results, enabling rapid development. This text report can be found in the same directory with the postfix "_FIFOReportTopHitters.txt".

7.1.2 Scheduling Annotation

To also provide more accurate estimates there also exists an option to enable the annotation of the scheduling resource usage as reported by the vendor tools back into the original user source code. As a result the annotation will be a lot more precise. However, it will require the completion of place and route by the vendor tools, resulting in a significantly longer runtime than the simpler visualisation tools.

Similar as in the case of the visualisation tool it is necessary determine the sources and sinks for each FIFO. This means that the resource annotation needs two columns. One indicates the costs of FIFOs starting from this line and the other the costs of FIFOs going to this line.

It is also possible that FIFOs share hardware resources. The compiler aggressively merges FIFOs to make optimal use of the width of the physical on-chip memory resources. As a result it is possible that the same hardware costs are annotated multiple times if the same hardware is used for multiple FIFOs.

To circumvent this problem we also added columns for unique FIFO costs. This represents FIFOs which do not share any hardware with other FIFOs. Combined with the knowledge of datapath widths it is possible for the user to estimate which costs are more representative and therefor should be addressed.

To enable this resource annotation the line:

```
photon.annotated_scheduling = true
```

has to be added to the `.MaxCompiler_build_user.conf` file.

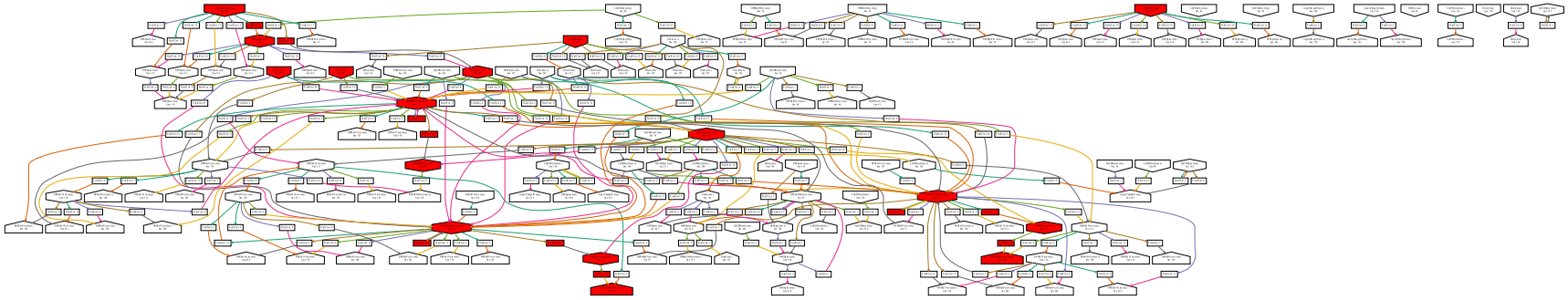

Figure 24: A simple scheduling graph for a big design. Even though the graph itself is not readable the parts of the algorithm that have to be improved to reduce the resources needed for scheduling are visible on the first glance.

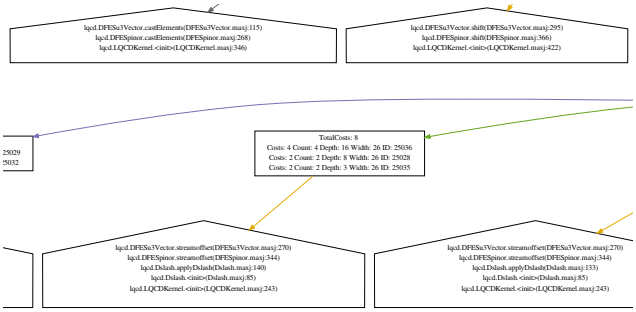

Figure 25: Sub part of the complex scheduling graph for the same design as shown in fig. 24.

7.1.3 Stream Offsets

There are many different stream offsets, all of which are explained in the main tutorial. In this section we want to focus on the ”normal” stream offset, created using the following API

*<T **extends KernelObject**<T>> T offset (T src, **int** offset)*
*<T **extends KernelObject**<T>> T offset (T src, **int** offset, T initVal)*

While this stream offset seems to be the simplest it is often the hardest to understand. The reason for this is, that while other stream offsets are mostly dedicated FIFO implementations this stream offset is actually just a constraint to the scheduler and as a result in some cases needs no hardware resources at all.

Basically, this command just informs the scheduler that a certain edge in the dataflow graph is read with a certain offset. Depending on the position in the schedule of the source and sink of this edge the depth of the resulting FIFO can be evaluated. Additionally, it might be possible to change the position of either the source or the sink in the schedule to remove the FIFO altogether.

An example for this can be seen in figures 26 and 27 which represent the dataflow graphs of *Listing 11* and *Listing 12* respectively. In this case the very simple equation of $(x+x)*x$ is implemented using MaxJ. Since we use fixed point arithmetic the addition has a latency of one cycle and a delay of one has to be inserted. By adding a stream offset taking the input from one cycle in the past as second argument to the multiplier this delay can be removed. Obviously, this results in a different functionality and changes the result of the computation, but it illustrates, that stream offsets do not always increase the hardware usage, but sometimes even reduce it.

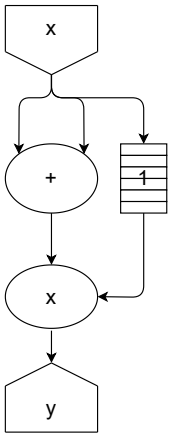

Figure 26: Scheduled dataflow graph for *Listing 11*.

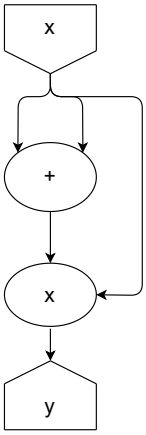

Figure 27: Scheduled dataflow graph for *Listing 12*. The addition of a stream offset removed a FIFO

Listing 11: An example kernel creating the dataflow graph shown in fig. 26

```
1 DFEVar x = io.input("x", dfeUInt(32));
2 DFEVar y = (x + x) * x);
3 io.output("y", y,dfeUInt(32));
```

Listing 12: An example kernel creating the dataflow graph shown in fig. 27. The addition of a stream offset removed a FIFO.

```
1 DFEVar x = io.input("x", dfeUInt(32));
2 DFEVar y = (x + x) * stream.offset(x, -1, 0));
3 io.output("y", y,dfeUInt(32));
```

An additional example for more advanced usage of stream offsets can be found in MaxPower. *Listing 13* shows the implementation of simple 2D transpose which can be found in the `Transpose` class of MaxPower. The transpose is implemented as a buffer which is first written and then read to get the transposed matrix. Since one value is read and written each cycle, it takes $width * height$ cycles to perform the complete transpose. The detailed implementation of the transpose is not discussed here, but can be looked up in the MaxPower source code. In this case it is abstracted by the function call to `transpose`. This function call is used inside a stream offset.

The stream offset in this example is used to hide the latency of the transpose functionality. By getting data from as many cycles in the future as the transpose requires time we get the transposed data immediately from the user perspective. As a result it is not necessary to add additionally control circuitry to enable other parts of the dataflow graph based on the delay of the transpose. This will not increase the hardware usage for the tranpose, since the output can be used immediately. The compiler will modify the flush and fill signal accordingly to delay the execution of other nodes in the dataflow graph. The only potential increase in hardware usage can be caused by using signals before and after the transpose, since those have to be delayed accordingly. Potentially, it can be beneficial to duplicate such signals in those cases.

Listing 13: Implementation of a transpose in MaxPower. Stream offsets are used to hide the latency.

```
1 public static <T extends KernelObjectVectorizable<T,?>>
2 T transpose(T in, int width, int height) {
3   return in.getOwner().stream.offset(transpose(in, width, height, in.getOwner().
        constant.var(true)), width * height);
4 }
```

To summarise stream offsets can be used in very advanced ways to dramatically reduce hardware usage or control complexity.

7.1.4 Arranging Arithmetic

In order to reduce scheduling resource usage and the latency of a design the structure of the dataflow graph is key. MaxCompiler adds operations to the dataflow graph as they are written in the java code. This means for example, that if multiple values are supposed to be reduced, e.g., by adding them all together the order in which they are added matters. The naive implementation of such a reduction would be a simple for loop. This would lead to a chain like structure, where intermediate results are added to new inputs.

Alternatively, it is often more helpful to perform this reduction in a tree like fashion. Here first all the individual inputs are combined in pairs of two. The resulting intermediate results are combined next and the process is repeated until only the final result is obtained. While the number of operations overall is the same, if a tree is used the latency of the circuitry is $log(n * op\_latency)$ while the more native reduction creates a latency of $n * op\_latency$. Additionally, the usage of trees removes all need for the scheduling of the inputs. This however, is only the case if all inputs arrive at the same time, e.g., because they are received by a kernel input or read from an on-chip memory.

It is important to consider in which operations are arranged in a dataflow graph and to optimise the consecutive parts of the dataflow graph accordingly. This will help to reduce the latency of the overall dataflow graph and reduce the usage of scheduling resources.

The creation of trees can be accomplished using the `reduce()` function in the `TreeReduce` class of MaxPower.

8 Strategies to Save Area

Reducing the area occupied by different functionalities programmed by MaxJ is always a target of every designer. The main motivation is, that more available resources provide more opportunities for parallelism and therefore increased performance.

It is probably an impossible target to collect all the different tips, tricks, methods and design patterns which can be used to save area on a DFE. As such this chapter will first explain the general approach to find opportunities for improvement and then explain some common or especially beneficial examples for optimisation opportunities. Additionally, we want to stress again that the most important way to reduce area usage is to use custom datatypes. This means, that first it is crucial to use fixed point arithmetic as much as possible and secondly every bit which can be removed further reduces the area requirements. Optimising these parameters is the most effective way to reduce the hardware usage as much as possible and should be done as early as possible in the design process.

There are two distinct circumstances in which one might want to reduce the area requirements of a design. The first happens during the design phase, in which different implementation are explored and planned for in the performance model. The second is after an initial implementation is made, which either does not match the predictions of the performance model or which is supposed to be optimised to further improve the characteristics of the design for example to push timing.

In the first case the designer will focus more on different architectures and ways to implement the same functionality. In this section we will present some commonly used design patterns, which can provide ideas on what is possible. Apart from original thinking and experience we can recommend to research implementations presented in the academic literature for similar problems and algorithms. In general most hardware architectures can be implemented using the dataflow approach but a special emphasis should be put upon researching high throughput and high performance designs.

If a initial implementation exists, it is a good practices to first look at the source code resource annotation generated by MaxCompiler. This annotation creates an overview document, called *report.txt*, which can be used to get an overview of the design. This text file contains a list of all components and the attributed hardware usage. One can easily identify which components use the most resources or do not fit expectations. It is usually a good idea to focus on those components first. The same is true for the source code resource annotation, which provides a line by line overview of the hardware usage. It is often already very productive to just quickly scan the annotation and identify the lines requiring most resources. Then one can consider how to change the design to solve these problems.

In the remainder of this section we will focus on some best practices and techniques to reduce the area usage of dataflow designs. This selection is in no way complete but addresses some of the most common mistakes and a few examples for more advanced concepts.

8.1 Manager

The manager consumes resources for three different purposes:

1. Instantiation of Kernels
2. Stream FIFOs (DFELinks)
3. Interfaces

Let's start at the top. The creation of Kernels requires some small amount of resources, to instantiate the default control logic. This logic is required to handle the execution of the kernel including stall behaviour and run cycle count. ManagerStateMachines and KernelLites do not require any static amount of resources. In fact the most basic versions of them, which just connect an input to an output,

can be implemented as a pure wire, removing any potential hardware costs. As such it is important to use the correct tool for the job. Kernels are the perfect environment for large high throughput dataflow graphs, but other elements can be often designed using other environments.

Additionally, the creation of many kernels leads to an increased overhead in control logic. Often it is beneficial to merge functionality into one kernel if possible. However, it is important to also consider the timing impact. Big kernels can have worse timing characteristics than smaller kernels. This is especially true for multi SLR FPGAs as used for most of the MAX5 generation DFEs. As such it is important to limit the size of individual kernels so that they can stay within a single SLR.

Another argument for the combination of as much as possible functionality in a single kernel is the reduction of required manager streams. Each time data are send from one kernel to another a FIFO has to be created. This FIFO is implemented using on-chip memory. Especially if the ports between multiple kernels are wide the amount of on-chip memory required can get significant.

If splitting of kernels is still preferred one should try to limit the width of the data transferred between kernels. This follows the same principles as reducing the required memory or PCIe bandwidth, even though the order of magnitude is very different[13].

If multiple connections exist between two kernels it can be beneficial to bundle them into a single FIFO (e.g. by using a DFEStruct). This can reduce the memory usage by using the port width of the on-chip memories more efficiently. For example sending four 8-bit signals separately requires four BRAMs. If they are bundled together it possible to use only one BRAM18. However, this saving is only possible if the signals bundled together are scheduled similarly. By using the same port they will need to arrive at the same time at this port. This might result in additional overhead within the kernel. A rule of thumb can be if signals are created and processed together. If that is the case they will usually also be scheduled together can be transported together. It should also be noted that creating multiple FIFOs between kernels can result in deadlocks, if for example data are generated at the same position in the schedule of kernel one, but processed at very different positions within kernel two. This is also explained in more detail in the debugging tutorial.

In the context of stream FIFOs it is also important to consider aspect changes. Each aspect change which is performed on the manager level requires the addition of more FIFOs. As a result one should try to use ACIO (see section 6.3) as much as possible. If it is not avoidable to perform an aspect change in the manager, for example because a 128 bit PCIe stream has to be expanded to a wider port, one should always use integer multiples.

Finally, the Manager requires resources for the creation of interfaces to deal with board infrastructure, e.g. the memory or PCIe controller. In general the resources required by these components increase with the number of streams used. Additionally, the resource usage of the memory controller increases with the burst width. As a result it can be beneficial to use less DIMMs, if the full memory bandwidth is not required.

8.2 On-chip memories

Historically, on-chip memory space was usually the scarcest resource. This has been somewhat alleviated with the addition of URAMs to the MAX5 generation DFEs, but an increase in available on-chip memory space can often be used to significantly reduce DDR bandwidth requirements. As such we feel that this resource deserves special attention.

[13] on-chip memory can usually achieve multiple TB/s memory bandwidth

8.2.1 Width and Depth

On-chip memories are usually quite flexible in how they are used. It is possible to configure the trade-off between depth and width. However, they are always an order of magnitude deeper than they are wide. This means that it is possible to store a lot of data, but not to access a lot of data on the same cycle. Whenever possible to trade depth versus width it is usually recommend to go for depth.

Additionally, it is often beneficial to merge multiple memories. If ten memories of depth 512 and width 10 are required it will save resources to use merge all entries together. This can only be done if the read and write addressing is the same. It is also important to remember that huge memories will impact timing negatively.

8.2.2 Read Ports

A commonly made mistake is to call the `read` function of a memory unnecessarily often. Each time this function is called a new read port is added to the memory. Since the number of read ports is very limited (usually to one or two depending on the platform, see section 13) this leads to a duplication of the used memory. Basically the toolchain creates a copy of the memory, filled with the same data, to provide the functionality of accessing the data at more positions in parallel. For this reason the use of the `read` function should be reduced as much as possible and especially avoided within loops.

There are three common solutions for this problem. If the read address does not differ between the function calls a intermediate variable can be used to read the data only once and access it multiple times. If addresses differ one can consider repacking the data and using a wider read port or, if that is also not possible, to hold the data in registers. In this case it is important that the overall amount of data is limited.

To store data explicitly in registers it is possible to use the following API:

*<T **extends KernelObject**<T>> T streamHold(T input, **DFEVar** store)*

The input stream is stored if store is true. It is kept and returned until it is overwritten by a new value.

8.2.3 Double Buffering

Double buffering is widely used to implement data reordering. It relies on a buffer that is duplicated to hold two copies corresponding to the same data structure. Usually, one buffer copy holds the current data set, which is used by the next stage, while simultaneously the second copy is written to, so that it entails the data set for the next step. Consequently, double buffering doubles the amount of memory needed to store the data structure. Double buffering is used when read and write patterns differ, as it allows to read data in a different access pattern without stalling the pipeline. Without double buffering, pipeline stalls would be needed to ensure data are read before an overwrite happens, and this reduces throughput. As a result, the decision whether to use double buffering depends on a trade-off between higher throughput and memory capacity required.

Double buffering is only necessary if the read and write patterns differ. If both patterns are the same, it is possible to read and write to the same address on the same cycle. By changing the sequence of read and write patterns in a systematic way it is possible to avoid double buffering. The central idea is that, even if read and write patterns are different, double buffering can be avoided if the write pattern is adjusted such that data is written to the same location that it has to be read from on the same cycle. However, this will also mean that the read pattern changes on the next iteration. Consequently, we need to derive a scheme to generate modified read and write address patterns, based on the nature of the implemented application.

As an example to introduce this concept we discuss the implementation of a 2D transpose. The source code for this implementation can be found in MaxPower. First, we consider the case where the content of a data stream representing a matrix is transposed with a dedicated transposer function. This means that on each cycle one value of a matrix is written into the transposer and one value is read. The transposer itself consists only of one memory and the necessary logic to generate the addresses.

In a straightforward implementation, there are two options to realise the transposer circuit: Either in single buffered version, where the transposer has separated write and read phases, or double buffered, where the memory is twice as deep. We develop a solution, which combines the advantages of both options and operates at full throughput but only uses one memory of normal depth.

As a simple example, we consider a matrix with a width of three and a depth of four. In the following we will repeatedly transpose a matrix of this size, which is filled with incrementing values from zero to eleven. Fig. 28 shows the matrix and the resulting transposed matrix. Tab. 3 shows the addressing scheme for the 2D transpose of such a matrix on the different cycles. These addresses represent the memory position we write to and read from. In iteration 1 we fill up the memory with the initial content. The content after writing incrementing numbers using the addressing shown in tab. 3 is shown in tab. 4. In iteration 2 we need to modify the access pattern such that the transposed matrix is read. In order to read the transposed matrix, we need to read from the addresses 0, 3, 6, and 9 to receive the first row. Since we write new contents to these addresses at the same time, those will now be filled with the values zero to three as can be seen in table 4. As a result, the addressing pattern needs to be adapted on the next iteration as well to ensure that the correct data is read. This step can now be repeated iteratively until iteration six is reached and the initial data ordering is repeated.

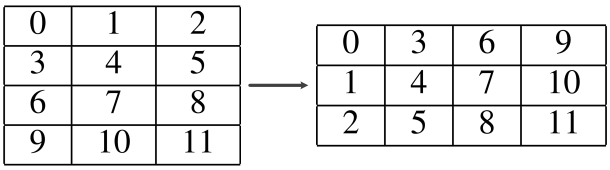

Figure 28: Transpose of a three by four matrix

Table 3: Addresses for a space-efficient 2D transpose of a three by four matrix.

	Iteration					
Cycle	1	2	3	4	5	6
1	0	0	0	0	0	0
2	1	3	9	5	4	1
3	2	6	7	10	8	2
4	3	9	5	4	1	3
5	4	1	3	9	5	4
6	5	4	1	3	9	5
7	6	7	10	8	2	6
8	7	10	8	2	6	7
9	8	2	6	7	10	8
10	9	5	4	1	3	9
11	10	8	2	6	7	10
12	11	11	11	11	11	11

It can be seen that it is possible to create a memory access pattern which performs the 2D trans-

Table 4: Memory content for a space-efficient 2D transpose of a three by four matrix.

	Iteration					
Address	1	2	3	4	5	6
$ 0	0	0	0	0	0	0
$ 1	1	4	5	9	3	1
$ 2	2	8	10	7	6	2
$ 3	3	1	4	5	9	3
$ 4	4	5	9	3	1	4
$ 5	5	9	3	1	4	5
$ 6	6	2	8	10	7	6
$ 7	7	6	2	8	10	7
$ 8	8	10	7	6	2	8
$ 9	9	3	1	4	5	9
$ 10	10	7	6	2	8	10
$ 11	11	11	11	11	11	11

pose in a memory of normal depth while maintaining throughput. However, it is impractical to use hardcoded addresses. As such we now need to derive a formula to calculate the required addresses. The addresses for iteration 2 can be calculated as shown in eq. (30), where $count$ is a simple incrementing counter from zero to the number of elements in the matrix minus one and $size$ represents the number of elements in the matrix to transpose.

$$address = \begin{cases} (size-1) & count == (size-1) \\ (count * width)\%(size-1) & \text{else} \end{cases} \tag{30}$$

However, this equation does not hold for the third iteration. So we need to extend eq. (30) to eq. (31), where $factor$ is defined as in eq. (32). If $factor$ is initialised with one and is updated at the beginning of each new iteration, these equations can be used to calculate all needed addresses. Furthermore, they are generic and work with all matrix input sizes. Even though the memory resource usage is halved using this solution, there is an additional overhead in terms of address generation logic: most notably the constant multiplications, constant modulo operations and one actual multiplication. In the case of the constant operations it is normally possible to replace them with additions and simple bit operations to save area. Additionally, the data path width for all these operations is very limited, reducing the overall costs. However, in cases where logic or multiplication resource are especially scarce and on-chip memory usage is of no concern, this optimisation might not be beneficial.

$$address = \begin{cases} (size-1) & count == (size-1) \\ (count * width * factor)\%(size-1) & \text{else} \end{cases} \tag{31}$$

$$factor_{i+1} = \begin{cases} (factor_i * width)\%(size-1) & count == (size-1) \\ factor_i & \text{else} \end{cases} \tag{32}$$

A more complicated version of the 2D transpose problem is the case where multiple values are written and read in each cycle for increased throughput. In this case it is not sufficient to use a single memory, since values which are written on the same cycle are normally not read on the same cycle as well. As such, as many memories as values processed in parallel are needed to deal with the read port restrictions.

We use the example of a four by four matrix, where two values are written and read in parallel. Again the input data is a stream of incrementing numbers. Tab. 5 shows the data of the two RAMs after it is written linearly into the two memories. In our example the values which have to be read on the first cycle are zero and four. However, one can see that both those values reside within RAM 1. As a result we would need to write and read two values from this memory, while RAM 2 would not be used. In many FPGA architectures, this introduces significant overheads or might even not be possible.

Table 5: RAM content for a 2D transpose of a four by four matrix with a read/write parallelism of two if data is written as it arrives.

Cycle	1	2	3	4	5	6	7	8
RAM 1	0	2	4	6	8	10	12	14
RAM 2	1	3	5	7	9	11	13	15

To circumvent this problem, we rotate the input data before they are written. In this case every second row of the matrix has to be rotated. The addressing pattern for this case can be seen in tab. 6. The $rotate$ value here indicates if the data has to be rotated before it is written. Tab. 7 shows the corresponding memory content. In this case the $rotate$ value indicates if the values read from the memory have to be rotated in order to generate the correct answer. The addressing pattern has only two different states and after two iterations the initial linear address pattern can be used again.

Table 6: Addresses for a space-efficient 2D transpose of a four by four matrix with a read/write parallelism of two.

	Iteration 1			Iteration 2			Iteration 3		
Cycle	RAM 1	RAM 2	rotate	RAM 1	RAM 2	rotate	RAM 1	RAM 2	rotate
1	0	0	0	0	2	0	0	0	0
2	1	1	0	4	6	0	1	1	0
3	2	2	1	2	0	1	2	2	1
4	3	3	1	6	4	1	3	3	1
5	4	4	0	1	3	0	4	4	0
6	5	5	0	5	7	0	5	5	0
7	6	6	1	3	1	1	6	6	1
8	7	7	1	7	5	1	7	7	1

In this case it is also possible to describe a general algorithm to generate the necessary control logic. Four counters are needed. The first one is called in_x and describes how many cycles were spent in the current row of the input matrix. Consequently in_y counts the rows of the input matrix. Finally $trans_x$ and $trans_y$ count the number of cycles in the current row and the rows of the transposed output matrix respectively. Additionally we need to keep track if the linear addressing pattern should be used or not. The variable $state$ is set to zero in the case of the linear addressing pattern and one otherwise. Eq. (33) and (34) describe the linear and transposed address pattern. In this equations, $pipes$ refers to the number of elements processed per cycle. In eq. (34) i refers to the index of the RAM starting at zero.

$$add\_l = in_y * (width/pipes) + in_x \tag{33}$$

$$add\_t_i = (trans_y + i) * (width/pipes) + (trans_x >> \lceil \log_2(pipes) \rceil) \tag{34}$$

Table 7: RAM content for a space-efficient 2D transpose of a four by four matrix with a read/write parallelism of two.

	Iteration 1			Iteration 2			Iteration 3		
Address	RAM 1	RAM 2	rotate	RAM 1	RAM 2	rotate	RAM 1	RAM 2	rotate
$ 0	0	1	0	0	4	0	0	1	0
$ 1	2	3	0	8	12	0	2	3	0
$ 2	5	4	1	5	1	1	5	4	1
$ 3	7	6	1	13	9	1	7	6	1
$ 4	8	9	0	2	6	0	8	9	0
$ 5	10	11	0	10	14	0	10	11	0
$ 6	13	12	1	7	3	1	13	12	1
$ 7	15	14	1	15	11	1	15	14	1

The different words of the input are interpreted as a vector. This vector is then rotated, where the number of rotates is determined by the $\lceil \log_2(pipes) \rceil$ bits of in_y. Similarly the output is rotated by the $\lceil \log_2(pipes) \rceil$ bits of out_x. The later rotate is also applied to add_t. This means that the addresses used to access the memories can be set as shown in eq. (35).

$$add_i = \begin{cases} add\_l & state == 0 \\ add\_t_{i,rotated} & state == 1 \end{cases} \tag{35}$$

This implementation of a parallelised 2D transpose is valid if the matrix is square, each dimension of the matrix is at least twice as large as the grade of parallelism, the dimensions are divisible by the grade of parallelism and the grade of parallelism is a power of two. It also works in some cases for non square matrices, e.g. width of four and height of six, but not for all, e.g. width of 12 and height of 16.

Again, overhead is added in terms of additional logic for the counters and rotates. However, since the rotates are also necessary in the case of double buffering, to avoid the need to write to the same memory multiple times on the same cycle, the overhead in control logic between the double buffered and the optimised version is negligible.

One can see that this optimisation is highly specialised to the memory access pattern. As such for each application the development of a custom addressing pattern might be required. The FFT library of MaxPower contains further examples for this optimisation.

8.3 Kernel Merger

Kernel Merger is a tool which is part of MaxPower and aims at the reuse of hardware blocks between multiple independent computations which are to be implemented on the same chip and never used at the same time.

Let us consider an algorithm that computes in hardware either $a*b$ or $a+b$ depending on a condition c. Both, the adder and the multiplier must be instantiated even though only one of the results will be selected by a multiplexer (MUX) at the output. However, if the computation is either $a * b$ or $a * d$ then instead of creating two multipliers we can use a single multiplier with a MUX selecting between inputs b or d. This is a more efficient hardware implementation as a MUX uses far fewer resources than the second multiplier.

Multiple algorithms are often implemented on the same chip, while only one of them is used at any given time. Since only one branch is active at any given time, the other branch remain idle. However,

since they still have to be implemented on the FPGA surface, a large portion of the chip is not in use most of the time.

A simple method to avoid such waste would be to create different configurations for every kernel and load the correct kernel when it is needed. Since the reconfiguration of an FPGA can take a considerable amount of time (typically longer than $100\ ms$ for larger devices) this method is often not sufficient.

A better solution for this problem is to reuse arithmetic units between branches to minimise the area overhead introduced by simultaneously implementing multiple branches.

Manually identifying the arithmetic units that can be shared and creating the required control logic is very time consuming. For this reason we created a dedicated pre-processing optimisation tool that is able to automatically merge multiple calculations and thereby significantly reduce the hardware area requirements without any additional user interaction.

8.3.1 Motivational Example

To demonstrate why such a tool is useful we provide a simple example. The dataflow graphs in figure 29 represent $(x+y)^2$ and $(x+y)*z$. Assume that only the result of one of the two calculations is used at any given execution cycle.

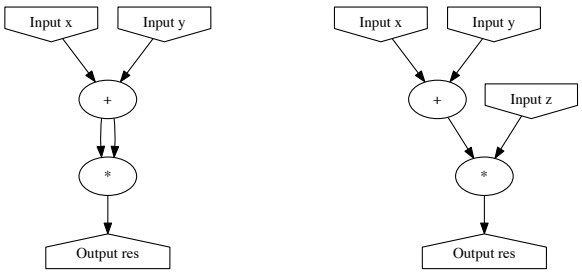

Figure 29: Two dataflow graphs implementing $(x+y)^2$ and $(x+y)*z$.

A naive implementation of both dataflow graphs in the same circuit would look like the dataflow graph shown in figure 30.

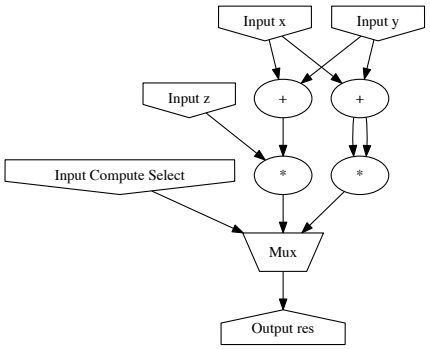

Figure 30: The dataflow graphs shown in figure 29 combined.

It is obvious that this implementation uses nearly double the necessary resources since both additions produce the same result and only one of the multiplication results is going to be used (the other one will be discarded). A more optimal solution in terms of area is shown in figure 31.

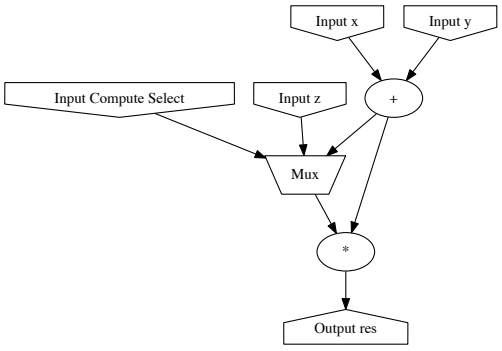

Figure 31: The graphs from figure 29 merged to save hardware resources.

This implementation reduces the hardware requirements significantly since only one addition and one multiplication have to be implemented.

While this seems to be trivial for small graphs of the size presented above, the search for optimal solutions is far more complex when graphs with thousands of nodes are considered.

Kernel Merger purely focuses on optimising arithmetic algorithms. Control flow, memory access and loops in the graph are handled outside of the library by the user.

8.3.2 Algorithmic Optimizations

The tool implements multiple optimisations as distinct graph passes. This means that they operate on a dataflow graph and transform it in order to save hardware resources. While many passes just optimise regular dataflow graphs, some of them are unique to the problem of reusing nodes between separate graphs.

All these graph passes run in a loop until the graph remains unchanged by an iteration. While this increases compilation time, it boosts the number of optimisations applied. We discovered that starting with a more optimised dataflow graph often leads to improved performance in terms of area reduction by the merging algorithms, thus we implemented many optimisations not specific to the problem of merging dataflow graphs to preprocess the graphs. For this reason after each node merging step all optimisation graph passes are repeated.

KernelMerger uses it own dataflow creation library which is very similar to normal MaxJ. The user creates a datflow graph within a unique kernel for each branch of the computation. These individual dataflow graphs are then merged.

The following optimisations are applied:

Dead Code Elimination Eliminate unused parts of the dataflow graph.

Constant Folding Optimise constant operations and compute them at compile time.

Associative Operations Merge all associative operations into a single node to enable other optimisations, like reordering, tree reductions and detection of power computations.

Common Sub-expression Elimination Perform CSE on the dataflow graph. Also uses the associative and commutative property.

Mathematical Optimisation Use distributive law and remove divisions as much as possible.

Improved Floating Point Addition Use the FloatingPointMultiAdder from MaxPower to add multiple floating point numbers more efficiently. Also works for subtractions mixed with addition.

Constant Multiplications and Divisions Change constant divisions to multiplications if possible, optimise constant multiplications.

Merging Merge multiple dataflow graphs automatically. This is based on cost based heuristics and can be modified by changing these heuristics and providing different cost factors.

Add Fanout Limitations Add automatic fanout limitations to each node in the graph.

8.4 Divisions

Divisions are a very costly operation. If possible they should be avoided altogether and, e.g., transformed to a multiplication. This can for example be accomplished if the divisor is either a constant or a datastream coming from the CPU. In that case one can calculate the inverse on the CPU and multiply with the inverse on the DFE.

Implementing a division in fixed point is more expensive then implementing it in floating point. This again provides an opportunity for optimisation if at least one of the numbers is available in float, or other operations that are cheaper in floating point are present directly previously or afterwards. It does not pay of to cast to float for a single division.

8.5 Rotates, Shifts and MUXs

In many cases it is necessary to reorder data. Sometimes this is even based on runtime results, where the order of items depends on the current data itself. There are typically four different building blocks which are used to accomplish this:

1. Buffers in Memory
2. MUXs
3. Rotates
4. Shifts

Buffering and reordering data in memory usually limits the throughput and is mostly required if reordering happens on a large scale. Methods for the optimisation of those blocks were already discussed earlier in this section. As a result we will focus on the other options.

MUXs select one of multiple inputs based on a select signal. The size of them increases faster than linear with the number of inputs and linearly with the size of each individual input. Often inexperienced MaxJ developer create multiple MUXs in a chain with the ternary operator. It is more efficient to create one big MUX instead (`control.mux()`).

It is often possible to use rotates or shifts if many data items are manipulated in the same way. Even though both of these operations are very expensive it is usually more efficient to use one rotate operation instead of many individual MUXs. Since these operations can quickly occupy a significant amount

of the available FPGA logic and additionally require many wires, resulting in high routing congestion, it is recommended to carefully try and evaluate multiple options.

Finally, it is important to note that all constant rotates and shifts are free, since they can be accomplished purely as a reinterpretation of data.

9 Strategies to improve Timing Closure

The compute performance of a Kernel is, broadly, set by its number of pipes (parallelism) multiplied by the clock frequency (see section 4.4. Designs should normally be written to be flexible in the number of pipes, and then the number of pipes could be selected to fit on the targeted chip (see: section 6.4.1).

While a frequency is expected during the development of the performance model and the architecture phase, the real achievable clock frequency can only be determined once the implementation is finished. At that point one can consider a trade-off between frequency and parallelism within the bandwidth constraints as determined using the performance model. This section will describe the general process of achieving timing closure for an design. Additionally, we will provide some special advice for multi die FPGAs and present some useful MaxCompiler functions.

Remember that performance is `pipes*frequency`, so maximum performance may be achieved with a slightly smaller than the maximum number of pipes, if it allows the whole design to run at a considerably higher frequency. E.g., a 10-pipe design at 250MHz will provide better performance than a 12-pipe design at 200MHz (2500 vs 2400 Mpoints computed per second).

9.1 General Timing Closure Process

Frequency optimisation is an iterative process that consists of: test builds, evaluate results, apply optimisations, repeat:

STAGE 1: Explore frequency range: Run a series of maxfile builds at differing frequencies, each one with multiple cost tables/implementation strategies.

- For generations up to MAX4: Use a small number of cost tables, but more then one. Individual cost tables are different random starting points, so you want to avoid looking at a single result because this may just fail randomly. A good number is 4 cost tables for an initial evaluation. With MAX5 implementation strategies replace cost/tables. Since they describe individual optimisation targets a selection of widely different optimisation targets is recommended. We will discuss the different implementation strategies in more detail in section 9.3.
- Choose several frequencies around your target frequency. Depending on the Place and Route compute resources available to you, you may do this in parallel or sequentially. Typical frequency ranges for large designs are 150-200MHz on MAX3, 200-250MHz on MAX4 and 200-350MHz on MAX5. Small designs may run as fast as 350MHz on MAX4 and 500MHz on MAX5 (but, if employing insufficient parallelism, may still deliver lower performance than larger designs running more slowly). A good practice is to use 10MHz intervals.
- Find the frequency at which the design just does not meet timing. If it meets timing at 180MHz, and fails at 190MHz, then take the 190MHz as your optimisation starting point.

STAGE 2: Analyse failing builds

- To make a design meet timing at a higher frequency, we should analyse why it fails at present. To filter out the influence of randomness, this should be done across multiple cost tables/implementation strategies and ideally across several frequency points. E.g., if you

look mostly at a 190MHz build, you may want to cross-check your conclusions against a 200MHz build, as well.

- Open the MaxCompiler timing report for the maxfile. You can do this either using MaxIDE or by entering the path into your web browser (usually: BUILDDIR/timingreport/index.html).
- The top of the timing report has a summary of all cost tables for that build, while the detailed body identifies paths that can not function at the required frequency.
- Generally, you will find that there is a small number of paths that show up repeatedly as problematic.

STAGE 3: Optimize failing paths

- Broadly, there are three possible locations for failing paths:
  - (A) MaxelerOS infrastructure (DDR, PCIe, MaxRing, etc.) or the Manager;
  - (B) Your code in a Kernel;
  - (C) Kernel control infrastructure.
- For any of the above, if you have multiple Kernels in your DFE configuration, confirm that they all need to run on the same high speed clock.
  - You may have one Kernel that can run slower than others, or one that needs to run the fastest. Especially address generators can usually run at lower clock frequencies.;
  - If so, modify your Manager code to generate multiple stream clocks (see section 9.3) and connect each Kernel to the appropriate clock;
  - This allows the Place and Route process to only optimise each Kernel as much as necessary, rather than pushing all Kernels equally.
- In the case of (A), try one or more of the following options:
  - If the failing paths are in the memory controller, try reducing the memory frequency (see: section 6.2.7 and 13);
  - If they are in the MaxRing controllers, try reducing the MaxRing speed or the number of lanes;
  - Optimise your Kernels to reduce the amount of chip area they require; This will free resources and will allow the MaxelerOS infrastructure to compile more easily;
  - Check your Manager graph and consider the width of streams linking Kernels, memory, etc. in the Manager. Minimise stream widths to free resources, if practically possible (see section 8.1). Consider the impact of SLR crossings if your DFE consists of multiple dies.
- In the case of (B), try one or more of the following:
  - Identify the responsible line of code using the timing report. Check that the operations match your expectations from the report, so you definitely have the right line (e.g., if the report says `Add->Mult`, then there should be code like $(a + b) * c$);
  - If you have reduced a pipelining factor for that line of code, consider increasing the node or graph pipelining factor (see: section 7);
  - Consider using `x = optimization.pipeline(x)` to add an additional register into the middle of an expression;
  - Look for fan-out issues. One major challenge for Place and Route is routing the same signal to many places; this can occur in your design easily, if you have a structure like:

```
DFEVar x = ;
for (int i = 0 ; i < 200; i++) {
    output[i] = f(x, input[i]);
}
```

In this case, `x` must be connected to all 200 instances of the function `f`. Check section 7 for more information.

* Replace complex expressions with simpler ones that do the same job, or remove duplicate expressions (MaxCompiler will not perform common sub-expression removal automatically, because the effect on the graph schedule is unpredictable);

– In the case of (C), try one or more of the following:

* Increase the amount of `CE` (*Clock Enable*) pipelining. The `CE` signal is distributed automatically to all the registers in your Kernel, to allow the Kernel to stop and start as necessary; it fans out via a tree, so increasing the depth of this tree should reduce the fan-out on this signal. You can use `Kernelconf.optimization.setCEPipelining(N)` for this (note this is in the Kernel configuration, created in the Manager before the Kernel was instantiated, not in the Kernel itself).
* Use a global clock buffer to distribute the clock and `CE` for a Kernel. Do as above, but `setUseGlobalClockBuffer`. This is mostly relevant for MAX3 and MAX5, particularly if you have very few large Kernels. There is a limited number of global clocks available, so you can not use this approach for 100 Kernels on the same chip.
* Reduce the number of registers in your Kernel, e.g., by reducing the pipelining factor (see section 7), or by removing unnecessary `optimization.pipeline` calls.

STAGE 4: Narrow frequency range

– Each cost table/implementation strategy has a timing score that indicates how close the design was to being able to run at the target frequency. If the timing score is equal to zero, design has met the timing; however, if the timing score is larger than zero, design will fail to meet timing.

– If your minimum timing score was in the range 1-50,000 (MAX4) or 1-1,000,000 (MAX5), this indicates that the design stands a good chance of meeting timing, if you run more cost tables/implementation strategies.

* Try this by re-running the design with a larger number of cost tables/implementation strategies specified, again depending on the level of compute resources available to you. For cost tables a good practical number is 16. In theory, you can run up to 100 cost tables, but running $>$16 is usually only needed for the very last step of optimisation of the most complex designs; In the case of implementation strategies it makes sense to try all the implementation strategies which have similar optimisations targets as the best performing implementation strategies already used.
* If you have not done so, you can tune other build settings (see section 9.3);
* If the design now meets timing, proceed to the next highest frequency, and repeat the previous steps;
* If the design still does not meet timing, proceed to stage 3.

STAGE 5: Rebuild and compare results

– Test individual changes or small sets of changes independently;

- Not all changes that should help will actually do so. Some may make things worse, so you need be aware of this, and you have to reconsider using them in your case;
- Make sure you clearly name your builds such that you can tell later what optimisations were being tested. It is especially helpful to keep records in a spreadsheet (build name, optimisation set, results, etc.). It is possible to automatically name build directories using EngineParameters by overriding the `getBuildName()` function. It is often helpful to add a String which can be set from the command line to this name.

Once you have succeeded in meeting a certain frequency, you can stop, or you can move onto the next higher clock frequency step. However, expect that, as you progress, you will eliminate the "low hanging fruit" quickly, so eventually you will be left with large amounts of work required for modest gains.

9.2 Multi-Die FPGAs

The MAX5 DFEs uses Xilinx FPGAs which consists of multiple dies. Basically, the VU9P used on the MAX5C is not a single FPGA but instead three VU3Ps "glued together". This means that communication between these individual parts of the chip is limited. It is still significantly higher than for example IO bandwidth, but lower than the normal routing bandwidth within an FPGA.

In order to address this we have developed multiple design patterns, best practices and MaxCompiler features. In this section we want to provide a brief overview on what is most important to achieve timing closure on MAX5 generation DFEs.

The most important property to notice is, that each individual ManagerBlock has to be small enough to fit in one of the dies. This means that it is not possible to have a single huge kernel. Instead one should aim at splitting components which are to big into multiple parts. There are fundamentally two approaches to this. One can either try to replicate the same design for each die or size the individual components and perform some high level floor planning to map them to the individual SLRs. More detail on this is provided in section 6.4.

It is possible to place specific ManagerBlocks, e.g. a Kernel, to specific SLRs using the following function:

void *assignToNamedRegion(ManagerBlock block, NamedRegion region)*

Each of the MAX5 manager offers this function to map ManagerBlocks to their different NamedRegions which map to the individual dies. Section 13 describes the architecture of the individual cards in more detail.

To connect blocks on different dies one can use the normal DFELinks. However, it is possible to add additional registers to those links to relax timing constraints using this function:

void *DFELink.setAdditionalRegisters(**int** number)*

It is recommended to add at least two levels of registering per SLR crossing, since this allows to register the signal directly before and after the signals cross the dies. This should also be used to connect components to the PCIe controller.

Finally one also has to consider the placement of other IO components. This includes the network interfaces and the DIMMs. They are specified in section 13.

In the case of the memory controller it might make sense to split the memory controller to handle DIMMs separately and avoid SLR crossings. While it is possible to use a shared memory controller routing the very wide data stream across the SLRs will occupy a significant portion of the number of available crossings.

Another good design pattern is to use different clocks for blocks which are placed on different SLRs. The creation of different clocks is described in the next section.

9.3 Options to Configure Builds

It is possible to change the way some of the kernel infrastructure is generated. For this one needs to change the `KernelConfiguration`. The current kernel configuration can be loaded by calling `getCurrentKernelConfig()` in a Manager. It is only possible to modify this configuration object in the manager, before the kernel is created.

Using this object it is possible to modify multiple settings on the `optimization` member. These are:

- `setCEPipelining(int pipelining)` Sets the number of pipelining stages to add to the clock enable signal. This allows the enable signal to travel a shorter distance during each cycle. When you have timing problems with the clock enable signal, increase the number of pipelining stages. Default value is 2. For `pipelining` choose a value greater of equal to 2.
- `setCEReplicationNumPartitions(int log2NumPartitions)` Sets the number of partitions for the clock enable signal. Where and when appropriate, increase the number of partitions to reduce the fan-out from the clock enable signal. Default value is 0 (i.e., $2^0 = 1$ partition).
- `setUseGlobalClockBuffer(boolean enabled)` Sets a dedicated, global clock buffer for a Kernel. This can improve the timing for Kernels, but there is a limited number of global clock buffers available. A good practice is to use clock buffering for larger Kernels in a multi-Kernel design. The default value is disabled.
- `setTriAddsEnabled(boolean enabled)` If enabled, the compiler will compress three additions into a single node and share LUTs for the implementation. This will reduce area usage but might have a negative impact on timing closure.
- `setUseAsapScheduler(boolean enabled)` Use an as-soon-as-possible (ASAP) scheduler instead of the normal area optimised scheduler. This might help with compilation MaxJ compilation times (not Place and Route) and change latencies within a kernel. Will usually increase hardware usage.

As described in section 6.2.7 it is possible to configure the frequency of the memory controller and DIMMs using the `setMemoryFrequency()` function of the `LMemGlobalConfig`. Additionally it is possible to also generate different stream clocks within the manager.

The

__void__ setDefaultStreamClockFrequency(__float__ frequency)

function sets the default clock frequency for Manager blocks. The default value is 100 (MHz).

Additionally, it is possible to generate individual blocks by calling one of these two functions:

ManagerClock generateStaticClock(__String__ name, __float__ frequency)
ManagerClock generateDynamicClock(__String__ name, __float__ frequency)

These clocks can then be assigned to a specific ManagerBlock by using the `setClock()` function of that component. The static clock will be fixed at compile time, while it is possible to change the dynamic clock after place and route for individual SLiC actions. However, dynamic clocks require slightly more resources.

Finally, it is possible, to directly configure the place and route tools. For this the `BuildConfig` class is used. An instance of this class can be obtained using the `getBuildConfig()` method in the manager. All functions described in this section are available on this instance.

The vendor tools try to balance the execution time with the quality of results. It is possible to manually increase the effort of the tools using the `setBuildEffort(Effort effort)` function. Increased effort will not necessarily lead to better results, so testing different settings is often a good idea.

Available options are:

- `Effort.LOW`
- `Effort.MEDIUM` (This is the default option)
- `Effort.HIGH`
- `Effort.VERY_HIGH`

Choosing a higher build effort will increase the build time, but should help with timing closure.

On MAX4 DFEs it is possible to allow the toolchain to automatically duplicate registers as needed. This will help especially in cases where a register has a high fanout, or a signal is routed across a large part of the chip. However, this option will increase the build time significantly by a factor of 2-3x, since the tool has to run place and route twice (first to figure out where registers are needed and then with the added registers). The function to enable this register duplication is called

*setEnablePhysicalSynthesisRegDuplication(Switch **switch**)*

The available options are:

- `Switch.AUTO` (This is the default option)
- `Switch.ON`
- `Switch.OFF`

Additionally, it is possible to set an optimisation goal for the place and route tool using the

setOptimizationGoal(OptimizationTechnique technique))

function. Possible optimisation goals are:

- `OptimizationTechnique.SPEED`
- `OptimizationTechnique.AREA`
- `OptimizationTechnique.BALANCED` (This is the default option)

In general, it is recommended to use `SPEED` for small design that are supposed to run at higher frequencies, and to use `OptimizationTechnique.AREA` when a a significant portion of the reconfigurable fabric is used. Although the default is `BALANCED`, once you are using more than 70% of the logic resources, `AREA` tends to produce better results.

MaxCompiler provides a function to run multiple place and route processes in parallel. This does not cover the complete build process, but only the usage of different cost tables/implementation strategies. For DFEs up to MAX4 the build parallelism can be increased using the `setMPPRParallelism(int numberOfThreads)` function. For MAX5 `setParallelism(int numberOfThreads)` is used.

It is possible to automatically retry a specific cost table/implementation strategy with an higher build effort. For this the `setMPPRRetryNearMissesThreshold(int retryTreshold)` function can be used for MAX4 or older and for newer DFEs the `setImplementationNearMissThreshold(int retryTreshold)` function. The threshold defines the maximum timingscore for which the build should be repeated.

For DFEs up to MAX4 place and route is performed using heuristic algorithms, such as simulated annealing, which is in essence a random process. Each cost table is equivalent to running a random process with a different seed - the same cost table will give the same result, but different cost tables may meet timing when other cost tables do not. Since the cost tables are run sequentially, if you meet timing on the first one, then MaxCompiler will not execute the remaining, so the time for the build does not change whether you are using this option or not in that case. If your design fails to meet timing on all cost tables, then it will take longer time. As a general rule of thumb, if running only one or two cost tables has not worked, then just run more (this also helps identify patterns in the timing report). On the other hand, if you have already run 8 cost tables that have not met timing, then your chances of meeting timing are not high, and you should probably consider other options. The function to configure the cost tables to run is `setMPPRCostTableSearchRange(int min, int max)`. The available range: $1 \leq min \leq max \leq 100$

On MAX5 generation DFEs this heuristic approach was replaced with a more objective driven algorithms. This algorithm follows an implementation strategy. It is possible to tell MaxCompiler to automatically try multiple implementation strategies. Once one of the implementation strategies is successful, the toolchain will cancel the other implementation strategy jobs. The implementation strategy to run is set using the

setImplementationStrategies(ImplementationStrategy... strategies)

function. The supported implementation strategies are:

- `VIVADO_DEFAULT`: Default Vivado settings for Implementation. This strategy is usually very fast to run, but struggles to meet timing for high frequencies or bigger designs.
- `PERFORMANCE_EXPLORE`: Uses multiple algorithms for optimisation, placement, and routing to get potentially better results.
- `PERFORMANCE_EXPLORE_POST_ROUTE_PHYS_OPT`: Similar to `PERFORMANCE_EXPLORE`, but enables the physical optimisation step (phys_opt_design) with the Explore directive after routing.
- `PERFORMANCE_WL_BLOCK_PLACEMENT`: Ignore timing constraints for placing Block RAM and DSPs, use wirelength instead.
- `PERFORMANCE_WL_BLOCK_PLACEMENT_FANOUT_OPT`: Ignore timing constraints for placing Block RAM and DSPs, use wirelength instead, and perform aggressive replication of components with high fanout.
- `PERFORMANCE_EARLY_BLOCK_PLACEMENT`: Finalise placement of Block RAM and DSPs in the early stages of global placement.
- `PERFORMANCE_NET_DELAY_HIGH`: To compensate for optimistic delay estimation, add extra delay cost to long distance and high fanout connections. (high setting, most pessimistic).
- `PERFORMANCE_NET_DELAY_LOW`: To compensate for optimistic delay estimation, add extra delay cost to long distance and high fanout connections. low setting, least pessimistic).

- PERFORMANCE_RETIMING: Combines retiming in phys_opt_design with extra placement optimisation and higher router delay cost.
- PERFORMANCE_EXTRA_TIMING_OPT: Includes alternate algorithms for timing-driven optimisation.
- PERFORMANCE_REFINE_PLACEMENT: Increase placer effort in the post-placement optimisation phase, and disable timing relaxation in the router.
- PERFORMANCE_SPREAD_SLLS: A placement variation for multi die devices with tendency to move SLR crossings horizontally.
- PERFORMANCE_BALANCE_SLLS: A placement variation for multi die devices with more aggressive crossings of SLR boundaries.
- CONGESTION_SPREAD_LOGIC_HIGH: Spread logic throughout the device to avoid creating congested regions. (high setting: highest degree of spreading).
- CONGESTION_SPREAD_LOGIC_MEDIUM: Spread logic throughout the device to avoid creating congested regions. (medium setting).
- CONGESTION_SPREAD_LOGIC_LOW: Spread logic throughout the device to avoid creating congested regions. (low setting: lowest degree of spreading).
- CONGESTION_SPREAD_LOGIC_EXPLORE: Spread logic throughout the device to avoid creating congested regions and run route_design Explore.
- CONGESTION_SSI_SPREAD_LOGIC_HIGH: Spread logic throughout multi die device to avoid creating congested regions. (high setting: highest degree of spreading). This strategy can help with very full designs.
- CONGESTION_SSI_SPREAD_LOGIC_LOW: Spread logic throughout multi die device to avoid creating congested regions. (low setting: minimal spreading).
- CONGESTION_SSI_SPREAD_LOGIC_EXPLORE: Spread logic throughout the multi die device to avoid creating congested regions and run route_design Explore.
- AREA_EXPLORE: Uses multiple optimisation algorithms to use potentially fewer LUTs.
- AREA_EXPLORE_SEQUENTIAL: Uses multiple optimisation algorithms to use potentially fewer LUTs and registers.
- AREA_EXPLORE_WITH_REMAP: Adds the remap optimisation to reduce logic usage.
- POWER_DEFAULT_OPTS: Adds power optimisation (power_opt_design) to reduce power consumption.
- POWER_EXPLORE_AREA: Combines power optimisation (power_opt_design) with sequential area reduction to reduce power consumption.
- FLOW_RUN_PHYS_OPT: Similar to the Vivado Implementation Run Defaults, but enables the physical optimisation step (phys_opt_design).
- FLOW_RUN_POST_ROUTE_PHYS_OPT: Similar to the Vivado Implementation Run Defaults, but enables the physical optimisation step (phys_opt_design) before and after routing.

- FLOW_RUNTIME_OPTIMIZED: Each implementation step trades design performance for better runtime. Physical optimisation (phys_opt_design) is disabled.
- FLOW_QUICK: Fastest possible runtime, all timing-driven behaviour disabled. Useful for utilisation estimation.
- MAXELER1: Places with increased estimated delay for high fan-out and long-distance nets (i.e. assumes a congested, difficult-to-route design), aggressive optimisation after place and route.
- MAXELER2: Focuses on maximising post-placement optimisation, with aggressive post-route optimisation.
- MAXELER3: Places with increased estimated delay for high fan-out and long-distance nets (i.e. assumes a congested, difficult-to-route design), using an alternate routing strategy.
- MAXELER4: Attempts to maximise the spread of logic throughout the device, using an alternate routing strategy.

It is furthermore possible to define custom implementation strategies using:

ImplementationStrategy createCustomStrategy(NetlistOpt optNetlist, PowerOpt powerOpt, PlaceOpt placeOpt, PowerOpt postPlacePowerOpt, PhysOpt postPlacePhysOpt, RouteOpt routeOpt, PhysOpt postRoutePhysOpt)

Some of the specific parts of this strategy can also be further customised. It is recommended to have a look into the Vivado documentation to get an in depth explanation on the different implementation strategies.

Additionally to customising the place and route process it is also possible to set optimisation targets for the synthesis. It is not possible to run multiple synthesis strategies in a single build, due to the potential all to all combination with implementation strategies. The synthesis strategy can be set using

setSynthesisStrategy(SynthesisStrategy strategy)

Possible strategies are:

- VIVADO_DEFAULT: Default Vivado settings for Synthesis. This is sufficient for most designs.
- FLOW_AREA_OPTIMIZED_HIGH: Performs general area optimisations including changing the threshold for control set optimisations, forcing ternary adder implementation, applying lower thresholds for use of carry chain in comparators and also area optimised MUX optimisations.
- FLOW_AREA_OPTIMIZED_MEDIUM: Performs general area optimisations including changing the threshold for control set optimisations, forcing ternary adder implementation, lowering multiplier threshold of inference into DSP blocks, moving shift register into BRAM, applying lower thresholds for use of carry chain in comparators and also area optimised MUX optimisations.
- FLOW_AREA_MULT_THRESHOLD_DSP: Vivado default options plus the AreaMultThresholdDSP directive which will lower the threshold for inference of multipliers into DSP blocks.
- FLOW_ALTERNATE_ROUTABILITY: Performs optimisations which creates alternative logic technology mapping, including disabling LUT combining, forcing F7/F8/F9 to logic, increasing the threshold of shift register inference.
- FLOW_PERF_OPTIMIZED_HIGH: Higher performance designs, resource sharing is turned off, the global fanout guide is set to a lower number, final state machine extraction forced to one-hot, LUT combining is disabled, equivalent registers are preserved, shift registers are inferred with a larger threshold.

- FLOW_PERF_THRESHOLD_CARRY: Vivado default options plus the FewerCarryChains directive for less inference of carry chains, turning off the LUT combining, resource sharing off, retaining equivalent registers.
- FLOW_RUNTIME_OPTIMIZED: Trades off Performance and Area for better runtime.

Again it is possible to create custom strategies using:

SynthesisStrategy createCustomStrategy(Directive directive, FlattenHierarchy flattenHierarchy , FsmExtraction fsmExtraction, CascadeDsp cascadeDsp, GatedClock gatedClock, ResourceSharing resourceSharing, ***boolean*** *keepEquivalentRegisters,* ***boolean*** *lutCombining,* ***boolean*** *srlExtract,* ***boolean*** *retiming,* ***int*** *bufg,* ***int*** *fanoutLimit,* ***int*** *controlSetOptThreshold,* ***int*** *shregMinSize,* ***int*** *maxBram,* ***int*** *maxBramCascadeHeight,* ***int*** *maxUram,* ***int*** *maxUramCascadeHeight,* ***int*** *maxDsp,* ***boolean*** *vhdlAssert,* ***boolean*** *oocMode,* ***String*** *options)*

If one wants to dive deeper into these options we recommend to look at the Xilinx Vivado documentation for more information.

10 Interfacing with the CPU

Up until now this document has very much focused on the design of the dataflow part of the complete system, but the importance of the control flow side should not be undervalued. If the dataflow side is not well integrated the overall performance of the system will be impacted significantly. In this section we will provide some advice on how to best integrate a DFE into a software project. We will first focus on the general architecture and then discuss the overlapping between CPU and DFE execution.

It is usually beneficial to add an abstraction layer between the actual CPU application and the SLiC API. This abstraction layer can be implemented for example as a single C++ class fulfilling handling all interactions with the DFE and transforming them into simple to use CPU functions. The functions usually include five topics:

1. Allocate engine and load maxfile.
2. Free engine and unload maxfile.
3. Reorder data to be sent to or received from the DFE.
4. Initialise LMEM and mapped memories as required.
5. Execution of the main task.

Similar functionality can also be provided by using for example the basic static SLiC interface or engine interfaces. However, both of these solutions have inherent limitations and as a result should usually be avoided for more complex applications. The basic SLiC interface does not easily support different types of execution for the same MaxFile (e.g. loading of parameters). The main motivation for the development of engine interfaces is the ability to create a simple API which enables sharing and reuse of the maxfile. However, since these functions are integrated in the maxfile itself they require a recompilation for every adjustment of calculations, to, e.g., the amount of data transferred or the number of ticks executed. As such they can only be used once the function of the maxfile has been fully verified and tested. Furthermore, due to the increased sizes of recent FPGAs architectures tend to become more complicated and as such we recommend nowadays recommend the usage of this custom abstraction layer instead.

To further increase usability potentially even by other users or in other projects using the same maxfile it is recommended to build a shared object from this thin abstraction layer. It also helps with the integration of the DFE project into programming languages other than C or C++ and provides a clear separation between DFE related code and the remaining application. Since the abstraction layer provides a high level interface to all the functionality of the maxfile it can be easily shared and reused. This also has the advantage that it is possible to easily develop tests for the maxfile independent of the main application. A test application can simply link against the shared object to test features in isolation through the same API. This also allows to easily test maxfiles containing different feature sets, build for different frequencies or DFE by simply changing which shared object the main application is using.

Furthermore if the software model uses the same API (e.g. by using a shared header file and only changing the implementation) a shared object build from the software model can be used as a drop in replacement for the DFE implementation further improving testability. It is then also possible to run the software model and the DFE implementation side by side, e.g., by using different namespaces and comparing the results of both implementations directly.

In the design of the API is is beneficial to consider two additional factors to improve performance. First, the amount of allocations should be reduced as much as possible. Often the size of data structures exchanged between host and DFE is constant. In this case it is possible to reuse already allocated

memory regions, to reduce allocation overheads. Secondly, one should consider to limit the idle time of the DFE. To achieve this the function that starts the execution of a SLiC action itself should be made as small as possible. Other functionality, e.g. reordering data, can be implemented in other functions which can then be executed in parallel to the DFE execution.

To implement this layer we recommend to use the advanced dynamic interface. It provides the best flexibility and control, assisting with debugging and achieving best performance. Additionally, it is a good idea to make heavy use of the `addMaxFileConstants()` function of the manager to sync up the state of the maxfile automatically with the CPU code. This includes for example the degrees of parallelism and the PCIe port width. The preprocessor constants which are added to the maxfile by this function can be used to automatically adjust tick counts, padding and the sizes of streams.

10.1 Overlapping Execution

In order to maximise performance it is crucial to overlap CPU and DFE execution times as much as possible. The main idea is shown in fig. 32. Instead of either executing on the CPU or on the DFE, concurrent execution on all processing units minimises runtime.

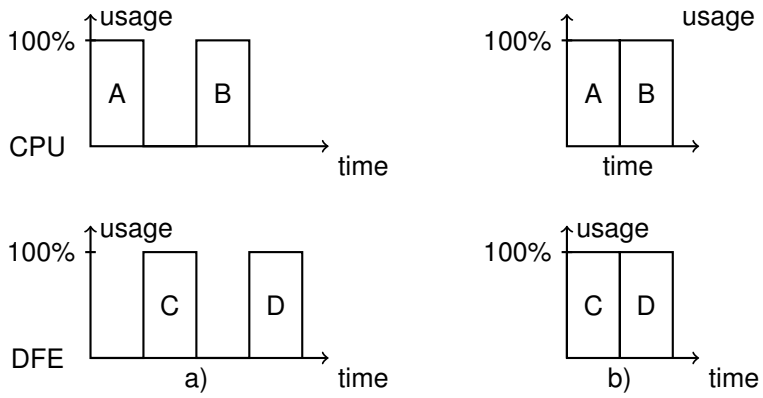

Figure 32: Suboptimal vs overlapped use of resources

There is a wide range of different strategies in overlapping execution times and the best strategy is highly application dependent. We want to mention a few strategies which are often helpful, but depending on the accelerated use case other solutions are possible:

- **Different working sets:** Often applications operate on many working sets consecutively. For example multiple images or transactions are processed one after another. In that case it is possible to prepare the next work item while the current one is still processed. Additionally, further pre- or post-processing steps can be executed on future or past working items. The main target here is to keep the DFE busy at all times.

- **Exploit application level parallelism:** In many applications different independent tasks are executed which are accumulated in a final result. It is unusual to accelerate all these tasks using DFEs. The tasks which remain on the CPU can be executed in parallel to the tasks ported onto the DFE.

- **Work splitting:** Another often seen execution pattern is to repeat the same kind of functionality on different initial conditions. For example a monte carlo simulation calculates an outcome for different starting scenarios and the final solution is the accumulation of all those scenarios. In that case the total working set can be split between CPU and DFE. For example if the DFE is

20x faster it can handle 95% of the working set and the CPU the remaining 5%. While the overall speed-up by this is relatively small, there is also no point in keeping the CPU idle.

- **Special case handling:** Sometimes it is not possible or very inconvenient to guarantee fault free execution on the DFE in all cases. For example in rare cases a loop requires significantly more iterations than can be unrolled or the fixed point number format causes overflows. If these kind of cases can be detected at runtime the CPU can be used to handle all these cases the DFE can not correctly process. This can happen in parallel to the remaining DFE execution.

It is possible to monitor the overlap between CPU and DFE execution using the event logger built into SLiC. In order to use this tool the following steps have to be executed in two terminal sessions. In the first session the following command has to be run:

```
$ maxeventlog --name <name for the eventlogger> -v4
```

In the second session the following is required:

```
$ export SLIC_CONF="eventlog_enable=true;default_eventlog_server=
    <name for the eventlogger>;default_eventlog_process_name=
    <your application>;eventlog_ignore_errors=true;"
$ ./<your application>
$ maxeventlogrendersvg -n <name for the eventlogger>
```

The last command will create an svg image in the current directory, which will show the execution on the DFEs and the CPUs

11 Datatypes

In many cases the hardest step in the creation of a high performance dataflow design is the selection of the datatypes for the implementation. The reason for this is, that floating point requires an unjustifiable large amount of area and fixed point has a very limited dynamic range. As such the designer has to spend significant effort to determine possible fixed point datatypes.

The challenge in this task is to select datatypes which both prevent an overflow as well as an underflow. In the case of an overflow the value of the calculation increases above the maximum representable value. For signed fixed point numbers this will lead to a sign change and in the case of unsigned values all bits higher than the maximum representable are ignored. An overflow will lead to a catastrophic error, which normally changes the result completely. A underflow occurs if not precision is present to represent the result of an calculation and the result gets rounded to zero. This error is usually less catastrophic, but can still result in a significant loss of precision for the final result.

As a result it is crucial to identify the value ranges for all individual values processed on the DFE to select the fixed point types which can represent them. There are three major questions that have to be answered in order to solve this challenge:

1. Is there a physical or mathematical process my program is simulating? If so, what are expected value ranges within the process, and what is the accuracy of the model?

2. What input data sets are going to be used? Do we expect significantly different value ranges?

3. What is the precision I require? How accurate is my program anyway?

In most cases especially scientific and financial applications are implemented using double precision floating point. However, the underlying model includes significantly larger errors than the precision errors introduced by the data types. There is no point in having an absolute accurate numerical implementation of an algorithm with significantly higher error.

Additionally, it should be noted that in many cases floating point numbers also introduce significant errors into a calculation. For example, floating point additions are not associative and the subtraction and addition of values with very different orders of magnitude also introduces large rounding errors.

In the most extreme case one might want to reproduce the result of the CPU implementation bit by bit. However, even if different compilers and CPUs are used this might not be possible due to a changed execution order or the used of vector operations, which often have different rounding and normalisation properties. As such, if one tries to reproduce bit accurately the results of double precision floating point algorithms one basically tries to reproduce the noise introduced by the floating point arithmetic unit.

Once the questions above are answered for the application to be ported are answered one can process with the technical process of creating a fixed point implementation. This process is again build from three stages.

1. Value Profiling

2. Fixed Point Simulation

3. Verify DFE fixed point implementation.

First, the original application, preferably in the form of the software model, is instrumented to perform value profiling. Usually one would use a library, which automatically records exponent values throughout the execution of the application. Figure 33 shows the value distribution of the result of the convolution operation in a CNN during training. Each training batch is recorded as a separate iteration and 20,000 iterations are then aggregated to generate the presented heat-map, showing the value distribution over

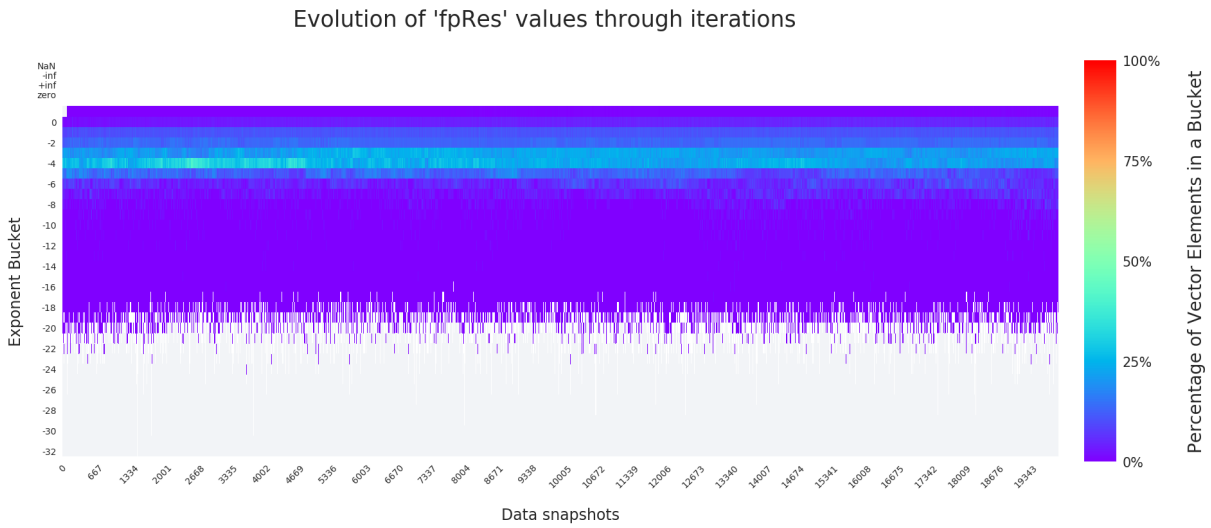

Figure 33: Heatmap of the exponent distribution for a convolutional layer during training.

the complete training period. Using this data educated decisions can be made regarding possible fixed-point types.

Afterwards a fixed point simulation library can be used to verify the lessons learned from the value profiling. Since these libraries are typically quite slow it is not possible to run all data sets through this simulation, but only the most significant ones. Once the results of this simulation are satisfying one can continue with the development process. At the end of the process the DFE implementation can be used to test using larger datasets. If errors are discovered on this stage it is often still possible to make small changes to the datatypes to resolve issues.

It is not necessarily possible to find a datatype which works for all cases. Instead it is possible to also introduce shifting blocks into the DFE design which enables re-scaling of values during execution. For example all input values can be normalised or within an iterative algorithm values can be normalised on every iteration. In those cases it is only important to keep track of the scaling at the input and correctly scale the outputs as well.

The remainder of this section will focus on some properties of the fixed point type implementation of MaxCompiler, which can help to implement working designs and save area.

11.1 Rounding Modes

Rounding is usually applied if the number of least significant bits is reduced. In this case the value of the new least significant bit has to be determined based on the removed bits. MaxCompiler supports three rounding modes:

1. `TRUNCATE`: This rounding mode removes all bits less significant than the new least significant bits. It is equivalent to rounding towards negative infinity or integer rounding in C.

2. `TONEAR`: This rounding mode rounds towards the nearest value. The value of the bits to remove is evaluated and checked if it is closer to rounding up or down. In a decimal system this is equivalent to rounding a value of 2.6 to 3. If the value of the bits to removes is precisely between both possible rounding targets (e.g. 2.5) the compiler always rounds upwards.

3. `TONEAREVEN`: Same as to near, but in the case of being between halfway between both possible rounding targets the compiler rounds towards the nearest even number. This removes to bias of

always rounding upwards and is equivalent to the rounding mode used in IEEE floating point. It is the standard rounding mode used by MaxCompiler.

The rounding mode can be configured using:

optimization.*pushRoundingMode(RoundingMode mode)*
optimization.*popRoundingMode()*

In MaxCompiler integer types are just a special case of fixed point types (e.g. no fraction and only integer bits). As a result the rounding mode also applies to all integer values. This means that if one replicates C integer arithmetic one-to-one in MaxJ the result can differ, since MaxCompiler will use round to nearest even and C will use truncate.

11.2 Saturating Arithmetic

Saturating arithmetic is a technique to limit the catastrophic effects of overflows. Instead of wrapping around in the case of an overflow a calculation using saturating arithmetic will set the result to the biggest representable value of the same sign. This still introduces an error, but it is usually a smaller one.

Saturating arithmetic can be enabled using:

optimization.*pushEnableSaturatingArithmetic(**boolean** enable)*
optimization.*popEnableSaturatingArithmetic()*

This implementation can produce wrong results if the result of an operation wraps multiple times or if in the case of unsigned fixed point types the result of an operation would normally be a negative value. If this can happen in a design the user has to implement saturating arithmetic manually.

11.3 Bitgrowth

MaxCompiler also provides functionality to automatically adjust datatypes. One often used configuration is:

optimization.*pushEnableBitGrowth(**boolean** enable)*
optimization.*popEnableBitGrowth()*

In this case the resulting datatype is automatically selected to represent all possible results without over- or underflow. This means for example that in the case of an multiplication of a n bit wide number by a m bit wide number the result will have n + m bits. In the case of an addition an additional bit is added to handle potential overflows.

Another side effect of this is, that constants are heavier optimised and types are selected in a way to only represent the chosen constant values. These changed datatypes propagate into constant expression optimisations.

Bitgrowth should only be used for small code segments, since otherwise the size of datatypes and therefor also the costs increase constantly. But bitgrowth offers three significant advantages:

1. All operations in MaxCompiler are first calculated in full precision and then rounded to the final result. Bitgrowth returns the full precision result and the user can decide to what new type should be rounded. This also enables to move rounding to a later point, since, e.g, the costs of wider adders are often acceptable.

2. Bitgrowth enables the usage of operations on variables of different types, since the compiler can automatically determine result types.

3. It is possible to only determine sensible types for the inputs and outputs of a functional block using value profiling, since all intermediate calculations are performed without any precision loss.

It is possible to also change the strategies used to determine result type using the following functions:

***optimization**.pushFixOpMode(FixOpBitSizeMode bit_size_mode, FixOpOffsetMode offset_size_mode, MathOps... ops)*
***optimization**.popFixOpMode(MathOps... ops)*

MathOps is an enum containing different operation types (`ADD`, `SUB`, `MUL`, `DIV`, `NEG`, `ALL`, `ADD_SUB`, `MUL_DIV`). The following functions in the `optimization` member can be used to generate different bit size modes:

- `bitSizeAll()`: A bit size strategy that will adjust fixed point types to fit the exact result of addition, subtraction, negation and multiplication, and uses a bit size for division that is as wide as it would be for multiplication. Using this strategy is thus similar to enabling bitgrowth.
- `bitSizeExact(int num_bits)`: A strategy that picks a `num_bits` wide fixed point type for all results.
- `bitSizeLimit(int max_bits)`: A bit size strategy that will adjust fixed point types to fit the exact result of operations as long as that does not exceed `max_bits`. For any operation it picks a bit size that is the smaller one of those picked by `bitSizeAll` and `bitSizeExact`.
- `bitSizeLargest()`: A bit size strategy that picks the largest size of the input types as the size of the output type.

The following functions in the `optimization` member can be used to generate different offset size modes:

- `offsetExact(int offset)`: A strategy that always picks `offset` as the fixed-point offset.
- `offsetLargest()`: A strategy that picks the largest input offset as the fixed-point offset for the result.
- `offsetLargestMsb()`: Max-based typing means that every fixed-point type carries with it a bit width and a maximum value. The fixed point offset is then chosen such that the maximum value is representable in the chosen bit width. `offsetLargestMsb` enables max-based typing, the chosen maximum value is the largest among the maximum representable values of the input types.
- `offsetNoOverflow()`: Max-based typing means that every fixed-point type carries with it a bit width and a maximum value. The fixed point offset is then chosen such that the maximum value is representable in the chosen bit width. `offsetNoOverflow` enables max-based typing, the maximum value is chosen such the largest possible result is representable. This setting is also used when enabling bitgrowth.
- `offsetNoUnderflow()`: Max-based typing means that every fixed-point type carries with it a bit width and a maximum value. The fixed point offset is then chosen such that the maximum value is representable in the chosen bit width. `offsetNoUnderflow` enables max-based typing, the maximum value is chosen to avoid undeflow in the result.

It is possible to further provide hints to the compiler by using:

*dfeFixMax(**int** numBits, **double** max, SignMode signMode)*

Here a maximum value is provided, which is used by the fixed point modes to determine the new result types.

11.4 Casting

Especially in the context of fixed point types casting is a necessity. In order to decide when and how to cast one first has to understand what costs are associated with different casts. In general there are four types of casts:

1. **Fixed to Fixed**: In this case it is free to add fraction bits and of small costs to add integer bits (sing extension). If the number of integer bits is reduced it is also free. If fraction bits are reduced the costs depend on the rounding mode and are between free (`TRUNCATE`) and the costs of an addition (`TONEAREVEN`).
2. **Float to Fix**: Roughly the costs of an floating point addition.
3. **Fix to Float**: Roughly the costs of an floating point addition.
4. **Float to Float**: In the case of casting to bigger types the costs are very small. The costs of casting to a smaller type are similar to a floating point addition.

If the DFE operates on fixed point numbers one has to consider where they are casted to and from float, since CPUs do not support fixed point natively. It is possible to manually pack floating point types on the CPU using structs, unions and a lot of shifts. However, this is not only error prone but also often runtime intensive. In many cases it is easier to perform the cast from float to fixed on the DFE itself. In this case one should also consider parallelism. For example a kernel might read more values in parallel than the PCIe bus is wide. In those cases a dedicated kernel to cast values at the PCIe speed can help to reduce hardware usage.

It is often beneficial to reduce the number of casting stages in fixed point arithmetic if additions are used to maintain precision without additional hardware costs. For example accumulating the output of multipliers at full precision and then only casting the final result often requires a very similar hardware usage as casting to smaller values first and then performing the accumulation.

12 Selected Advanced Kernel Functionalities

In this section we want to quickly discuss very few selected more advanced kernel programming constructs. They can provide further options to simplify code, execution schemes and code maintainability.

12.1 Flushing Modes

Flushing refers to the way in which the execution of a kernel ends. Since the kernel consists of a computational pipeline, it is necessary to stop inputs at the correct time to avoid stalling while still sending out data through the outputs. The process of flushing is automatically handled by the compiler, but the programmer controls the conditions which start the flushing process.

By default each kernel adds a counter keeping track of the cycles executed as well as a scalar input to record how many cycles are supposed to be executed in total. The programmer sets the number of ticks to execute via SLiC and the kernel starts flushing once this amount of ticks has passed.

Alternatively it is possible to define custom flushing modes. The simplest case is to disable flushing using

flush.disabled()

In this the kernel is not flushed at all but runs continuously. Stalling has to be avoided manually, by disabling inputs if the execution is supposed to finish and the remaining words have to be able to finish flowing through the dataflow graph.

Additionally, it is possible to control the start of flushing using MaxJ control signals using

*flush.onTrigger(**DFEVar** trigger)*
*flush.afterTrigger(**DFEVar** trigger)*

In the first case flushing starts once the stream passed in is true and in the second case on the cycle afterwards. Using this function it is possible to also rebuild the standard behaviour as shown in listing 14.

Listing 14: A simple example rebuilding the default flushing strategy using teh flushing API.

```
1 DFEVar tickCount = control.count.simpleCounter(48);
2 DFEVar ticksToDo = io.scalarInput("TickToDo", dfeUInt(48));
3 flush.onTrigger(tickCount === ticksToDo);
```

12.2 KernelLib

The `KernelLib` class allows the extraction of certain functionalities in separate classes. Instead of extending the `Kernel` class the `KernelLib` class has to be extended. While all the API that is available in a normal `Kernel` is also available in a `KernelLib` the hardware overhead of creating new kernels is avoided. An example for a simple `KernelLib` is shown in listing 15, which encapsulates an linearly addressed RAM.

Listing 15: A KernelLib which implements a wrapper around a memory.

```
1 public class LinearMemory extends KernelLib {
2
3   private final DFEVar m_readData;
4
5   public LinearMemory(KernelLib owner, DFEVar writeData, DFEVar writeEnable, DFEVar
        enable, int size) {
```

```
6       super(owner);
7
8       Params counterParams = control.count.makeParams(MathUtils.bitsToAddress(size));
9       counterParams = counterParams.withMax(size).withEnable(enable);
10      DFEVar address = control.count.makeCounter(counterParams).getCount();
11
12      Memory<DFEVar> mem = new Memory<DFEVar>(this, size, writeData.getType());
13
14      mem.write(address, writeData, writeEnable);
15      m_readData = mem.read(address);
16    }
17
18    public DFEVar getReadData() {
19      return m_readData;
20    }
21
22  }
```

The `KernelLib` functionality helps to split the DFE design into multiple smaller and more manageable components. These components can then be also tested in isolation.

12.3 Input Registering

For every input, there is a register that stores its value. This behaviour could be changed using `io.pushInputRegistering`. This disables input registering, reduces latency, and reduces the flip-flop usage. By default, this optimization is enabled.

Warning: Reading when input is not enabled returns the previously stored value. If input registering is disabled, the behavior of input changes: reading when input is not enabled, reads undefined data.

```
io.pushInputRegistering(false);
// code that setting applies to
io.popInputRegistering();
```

12.4 Simulation Error Behaviour

The `simulation` member of the `KernelConfiguration` object contains the following helpful functions:

- `setRAMOutOfBoundsBehaviour(SimulationBehaviour behaviour)`
  Changes the behaviour if an out of bounds error for on-chip memory occurs during simulation. Can be changed between ignore, warning or exception. The default error is exception.
- `setRAMAddressCollisionBehaviour(SimulationBehaviour behaviour)`
  Changes the behaviour if an address collision occurs (e.g. write and read to the same address on the same cycle). Can be changed between ignore, warning or exception. The default error is exception.

13 System Description of Current DFEs

This chapter is supposed to provide a more detailed explanation of the individual DFEs. We describe common traits for DFE generations and list the precise properties of each DFE supported by MaxCompiler.

13.1 General Information

Since the interfaces used between different DFE generations are fundamentally the same we provide some general information here first.

13.1.1 PCIe

PCIe is a bidirectional protocol. This means that the bandwidth in each direction is not shared and the bus can send and receive data at full speed at the same time.

The speed of a PCIe link is determined by the link width and the generation. Tab. 8 shows the speed per PCIe lane for different generations. To calculate the PCIe bandwidth of a given DFE these numbers can be multiplied with the width of the bus.

Table 8: Speed per lane for different PCIe generations.

Generation	Speed (GB/s)
1.0	0.25
2.0	0.5
3.0	0.985
4.0	1.97

These numbers only describe the theoretical maximum performance. Realistically, one can usually achieve roughly 80% of these bandwidths.

13.1.2 LMEM

In contrast to PCIe DDR is unidirectional and bandwidth between reading and writing is shared. The theoretical peak bandwidth per DIMM can be calculated as follows:

$$Bandwidth_{DDR} = 8Bytes * 2 * freq \tag{36}$$

Where the frequency is the frequency provided in the manager.

13.2 MAX4

The MAX4 DFE generation is based on the Intel Stratix V FPGA family.

13.2.1 Device Overview

Tab. 9 shows an overview over all devices of the MAX4 DFE generation.

Table 9: Overview of MAX4 Generation DFEs

	MAX4C	MAX4U	MAX4N
FPGA	5SGSD8	5SGXA5	5SGXAB
ALMs	262,400	185,000	359,200
DSPs	1,963	256	352
M20Ks	2,567	2,304	2,640
On Chip Memory Capacity	6.3 MByte	5.6 MByte	6.4 MByte
DDR DIMMs	6	3	3
DDR Capacity per DIMM	8 GB	4 GB	8 GB
Supported DDR Frequencies (MHz)	400, 533, 666, 733, 800	400, 666	400, 533, 666, 733, 800
PCIe	PCIe Gen 2 x8	PCIe Gen 2 x8	PCIe Gen 2 x8
Networking	-	-	up to 3 x 40 GBit/s

13.2.2 ALMs

Figure 34 shows the block diagram of an ALM used on Altera Stratix V FPGAs. It can be observed that there are many different options, in which an ALM can be configured. Basically, an ALM has eight inputs and four different, registered outputs. The lookup tables (LUT) can be used as two six input LUTs, if four inputs are shared. Otherwise they can also be used as two four input LUTs or in many other, different combinations. Additionally a ALM also contains two one bit adders and a selection of multiplexers (MUX), which are used to connect everything [3].

13.2.3 On Chip Memory

The Stratix V FPGA has two different types of on chip memory. The first type is called MLAB (memory logic array block), and is build from ALMs. Ten ALMs are combined to build either a 10 bit wide and 64 deep or a 20 bit wide and 32 bit deep two port memory.

The second type of memory is called M20K and has, as the name suggests, a capacity of 20 Kb. This type of memory has also two ports, which can be either set up as a write or a read port. As a result, in order to generate a memory with one write but two read ports the memory has to be duplicated. Table 10 shows the different options to configure an M20K with one read and one write port. It is important to note, that not all bits can be addressed, once the memory width drops below 5 bits and also that an optimal width is crucial to fully utilise this type of memory [3].

Table 10: Different configurations for the M20K memory blocks of the Stratix V FPGA [3]

Depth	Width (bits)
512	40
1024	20
2048	10
4096	5
8192	2
16384	1

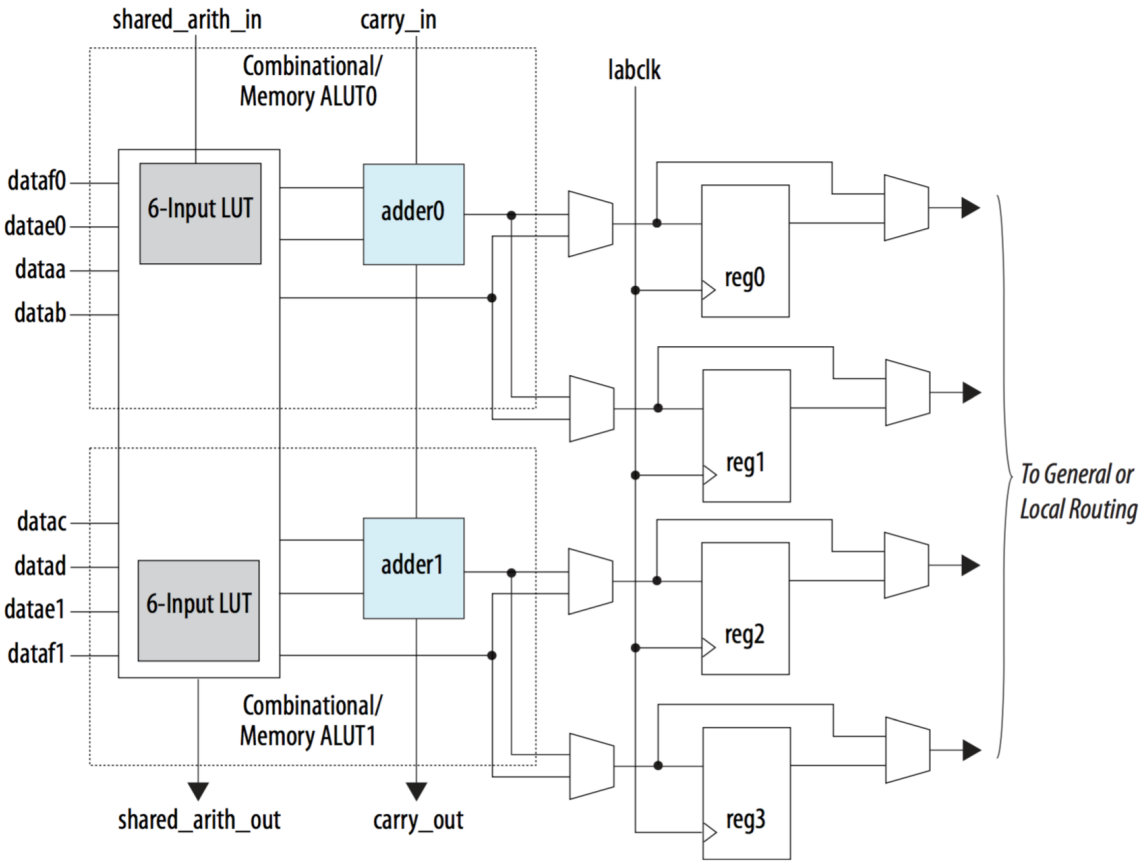

Figure 34: Block diagram of an Stratix V ALM. Source: [3]

13.2.4 DSPs

DSPs are used to implement multiplications more efficiently and at high frequencies. One DSP block can be used as shown in table 11. It is important to note, that in many of these cases it is also possible to use the DSP to add the result of several multiplications (post-add). DSPs also support floating point arithmetic formats, a pre-addition before the multiplication, an accumulation of the results and the addition or subtraction of multiplication results from other DSPs. [3].

Table 11: Different configurations for the DSP blocks of the Stratix V FPGA [3]

Width (bits)	Count per DSP
9x9	3
16x16	2
18x18	2^{14}
27x27	1
36 x18	1

13.3 MAX5

The MAX5 DFE generation is based on the Xilinx Ultrascale+ FPGA family.

13.3.1 Device Overview

Tab. 12 shows an overview over all devices of the MAX5 DFE generation.

13.3.2 CLBs

The logic resources of the Ultrascale+ architecture are organised in slices. Each slice consists of eight six input LUTs, 16 FFs and one carry chain The LUTs can also act as two five input LUTs instead, if all the inputs are the same.

MUX combinations can be creates as shown in tab. 13.

Additionally, some CLBs can be used to build 256 bit shift registers or 512 bit RAMs. In these cases the complete CLB is used. The RAMs can be configured as shown in tab. 14.

13.3.3 BRAMs

BRAMs always exist in blocks in the Ultrascale+ architecture. Two BRAM18s form on BRAM36 block, where the BRAM18s can either be used independently or combined as a BRAM36. Each BRAM has two write and two read ports, which can be bundled up to double the port width while halving the depth of the memory. Tab. 15 shows the possible configurations for a BRAM18 block.

13.3.4 URAMs

URAMs are a single dual port memory which is 4096 elements deep and 72 bits wide. It is not possible to tile the memory in any other way. Additionally, URAMs can not be used to build dual clock FIFOs. As a result they will not be used by MaxCompiler for Manager FIFOs, which cross clock domains (different stream clocks / user logic to PCIe or DDR).

13.3.5 DSPs

The core of the Xilinx Ultrascale+ DSP consists of a 27 by 18 bit multiplier. It is not possible to tile this multiplier in any other way. Additionally, there exists a 27 bit pre adder and a 48 bit accumulator after the multiplier, which can also be used to add numbers to the result of other DSPs. Fig 35 shows an image showing the DSP architecture.

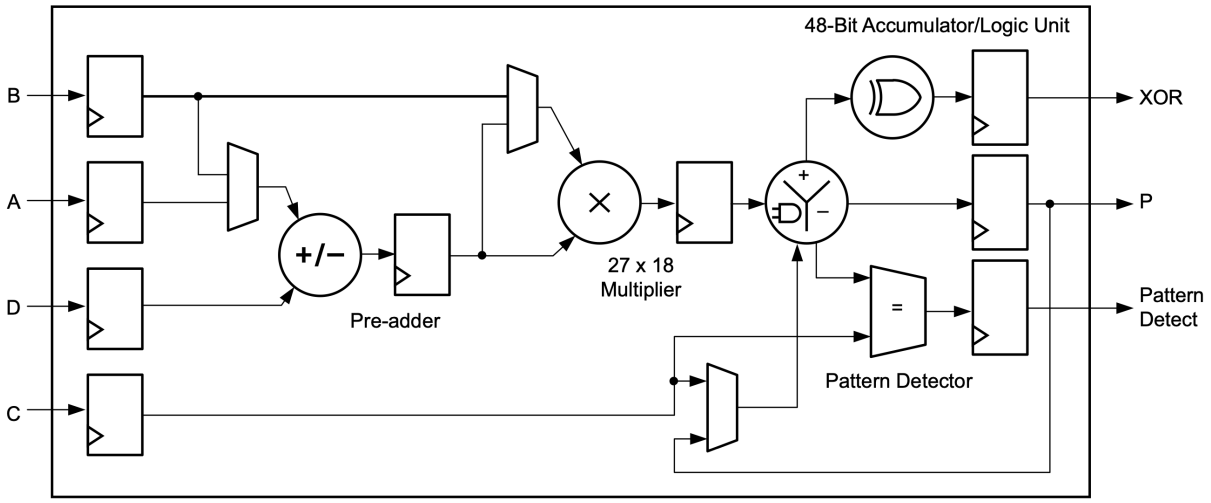

Figure 35: Block diagram of an Xilinx Ultrascale+ DSP48E2. Source: [40]

13.3.6 F1

On the AWS EC2 F1 instance Amazon blocks of a part of the FPGA to instantiate I/O and board management functionality. While some of these functions are needed for the manager the shell can be seen as reducing the amount of resources available on the FPGA in general. The shell occupies roughly 20% of the available resources. More precisely it blocks a third of both SLR0 and SLR1. Fig. 36 shows an chip image of the AWS shell.

13.3.7 Resource Usage for Operations

Tab. 16 to 27 show the resource usage for selected operations and data types using the default settings.

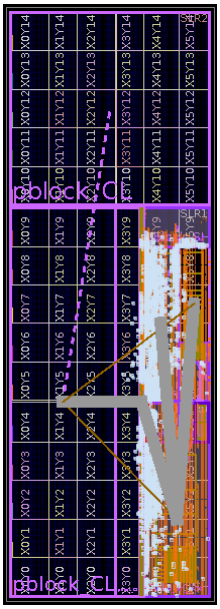

Figure 36: Chip image of the AWS F1 instance shell. SLR0 is at the bottom.

Table 12: Overview of MAX5 Generation DFEs

	MAX5C	Alveo U200	Alveo U250	AWS EC2 F1	miniMAX5
FPGA	VU9P	VU9P	VU13P	VU9P	KU5P
SLRs	3	3	4	3	1
LUTs (k)	1,182	1,182	1,728	900	217
FFs (k)	2,364	2,364	3,456	1,800	434
DSPs	6,840	6,840	12,288	5,832	1,824
BRAM18s	4,320	4,320	5,376	3,360	960
URAMs	960	960	1,280	800	64
On Chip Memory Capacity	43.2 MByte	43.2 MByte	56.8 MByte	37.2 MByte	4.4 MByte
DDR DIMMs	3	4	4	4	1
DDR Capacity per DIMM	16 GB	16 GB	16 GB	16 GB	16 GB
Supported DDR Frequencies (MHz)	933, 1066, 1200	1200	1200	1200	933, 1066, 1200
DIMM to SLR Mapping	DIMM_0 ->SLR0 DIMM_1 ->SLR1 DIMM_2 ->SLR2	DIMM_0 ->SLR0 DIMM_1 ->SLR1 DIMM_2 ->SLR1 DIMM_3 ->SLR2	DIMM_0 ->SLR0 DIMM_1 ->SLR1 DIMM_2 ->SLR2 DIMM_3 ->SLR3	DIMM_0 ->SLR0 DIMM_1 ->SLR1 DIMM_2 ->SLR1 DIMM_3 ->SLR2	DIMM_0 ->SLR0
PCIe Placement	SLR1	SLR0	SLR0	SLR1	SLR0
PCIe	PCIe Gen 2 x8	PCIe Gen 2 x8	PCIe Gen 2 x8	PCIe Gen2 x8	PCIe Gen2 x4
Networking	1 x 100 GBit/s	2 x 100GBit/s	2 x 100GBit/s	-	2 x 10 GBit/s
Networking Placement	SLR2	SLR2	SLR2	-	SLR0

Table 13: MUX combinations creatable in Ultrascale+ CLBs

MUX type	LUTs required
4 inputs : 1 outputs	1
8 inputs : 1 outputs	2
16 inputs : 1 outputs	4
32 inputs : 1 outputs	8

Table 14: RAM configurations of Ultrascale+ CLBs

Port count	Depth	Width
1	32	1-16
2	32	1-8
4	32	1-4
1	64	1-8
2	64	1-4
4	64	1-2
1	128	1-4
2	128	1-2
4	128	1
1	256	1-2
2	256	1
1	512	1

Table 15: RAM configurations of Ultrascale+ BRAM18

Port count	Depth	Width
1	512	36
2	1,024	18
2	2,048	9
2	4,096	4
2	8,192	2
2	16,384	1

Table 16: Area Usage of Operations on MAX5 using 18 bit fixed point

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	18	19	0	0
Mul	1	1	0	1
Div	387	821	0	0
KernelMath.sin	78	255	1	4
KernelMath.cos	77	276	1	4
KernelMath.exp	237	489	1	4
KernelMath.pow2	229	456	1	4
KernelMath.log	102	247	1	4
KernelMath.sqrt	130	167	0	0

Table 17: Area Usage of Operations on MAX5 using 27 bit fixed point

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	27	28	0	0
Mul	5	45	0	2
Div	826	1716	0	0
KernelMath.sin	130	436	2	8
KernelMath.cos	130	466	2	8
KernelMath.exp	412	949	1	10
KernelMath.pow2	390	825	1	5
KernelMath.log	140	366	1	5
KernelMath.sqrt	268	320	0	0

Table 18: Area Usage of Operations on MAX5 using 32 bit fixed point

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	32	33	0	0
Mul	4	99	0	4
Div	1143	2357	0	0
KernelMath.sin	158	537	2	10
KernelMath.cos	157	572	2	10
KernelMath.exp	526	1368	1	14
KernelMath.pow2	512	1109	1	5
KernelMath.log	139	420	1	7
KernelMath.sqrt	317	376	0	0

Table 19: Area Usage of Operations on MAX5 using 36 bit fixed point

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	36	37	0	0
Mul	5	143	0	5
Div	1430	2939	0	0
KernelMath.sin	180	770	2	17
KernelMath.cos	179	809	2	17
KernelMath.exp	551	1618	2	19
KernelMath.pow2	532	1313	2	9
KernelMath.log	180	514	1	8
KernelMath.sqrt	397	464	0	0

Table 20: Area Usage of Operations on MAX5 using 48 bit fixed point

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	48	49	0	0
Mul	53	284	0	9
Div	2555	5069	0	0
KernelMath.sin	466	1293	11	27
KernelMath.cos	458	1344	11	27
KernelMath.exp	1172	1792	3	17
KernelMath.pow2	956	1466	3	9
KernelMath.log	420	776	2	12
KernelMath.sqrt	712	818	0	0

Table 21: Area Usage of Operations on MAX5 using 54 bit fixed point

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	54	55	0	0
Mul	10	333	0	10
Div	3196	6339	0	0
KernelMath.sin	282	1593	12	31
KernelMath.cos	285	1650	12	31
KernelMath.exp	640	1779	3	19
KernelMath.pow2	626	1484	3	9
KernelMath.log	186	999	3	22
KernelMath.sqrt	1120	1235	0	0

Table 22: Area Usage of Operations on MAX5 using dfeFloat(8,8)

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	154	220	0	0
Mul	56	89	0	1
Div	152	244	0	0
KernelMath.exp	201	427	0	3
KernelMath.pow2	183	399	0	1
KernelMath.sqrt	89	161	0	0

Table 23: Area Usage of Operations on MAX5 using dfeFloat(8,24)

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	177	308	0	2
Mul	77	165	0	2
Div	742	1353	0	0
KernelMath.sin	500	917	2	5
KernelMath.cos	497	917	2	5
KernelMath.exp	444	982	1	9
KernelMath.pow2	413	870	1	4
KernelMath.log	319	716	2	5
KernelMath.sqrt	412	785	0	0

Table 24: Area Usage of Operations on MAX5 using dfeFloat(11,32)

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	427	722	0	0
Mul	95	244	0	4
Div	1247	2302	0	0
KernelMath.sin	771	1298	2	10
KernelMath.cos	771	1298	2	10
KernelMath.exp	555	1472	2	16
KernelMath.pow2	519	1233	2	7
KernelMath.log	430	1063	2	10
KernelMath.sqrt	673	1298	0	0

Table 25: Area Usage of Operations on MAX5 using dfeFloat(11,39)

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	483	837	0	0
Mul	105	340	0	5
Div	1780	3335	0	0
KernelMath.sin	976	1730	3	17
KernelMath.cos	976	1730	3	17
KernelMath.exp	645	1799	3	19
KernelMath.pow2	602	1510	3	9
KernelMath.log	525	1436	3	17
KernelMath.sqrt	951	1845	0	0

Table 26: Area Usage of Operations on MAX5 using dfeFloat(11,47)

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	553	969	0	0
Mul	115	458	0	9
Div	2507	4743	0	0
KernelMath.sin	1155	2296	11	27
KernelMath.cos	1150	2296	11	27
KernelMath.exp	887	2810	3	33
KernelMath.pow2	8378	2330	3	17
KernelMath.log	601	1924	11	27
KernelMath.sqrt	1328	2589	0	0

Table 27: Area Usage of Operations on MAX5 using dfeFloat(11,53)

Operation	LUTs	FFs	BRAMs	DSPs
Add / Sub	582	949	0	3
Mul	132	534	0	7
Div	3135	5979	0	0
KernelMath.sin	1261	2714	12	28
KernelMath.cos	1279	2714	12	28
KernelMath.exp	1290	2873	3	38
KernelMath.pow2	989	2797	3	26
KernelMath.log	678	2272	12	28
KernelMath.sqrt	1653	3231	0	0

14 Bibliography

[1] Altera. *Floating-Point IP Cores User Guide. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/ug/ug_altfp_mfug.pdf*.

[2] Altera. *Integer Arithmetic IP Cores User Guide. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/ug/ug_altmult_add.pdf*.

[3] Altera. Stratix V Device Handbook. Technical report, Altera, 2016.

[4] David Boland and George A. Constantinides. Word-length optimization beyond straight line code. In *The 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, Monterey, CA, USA, February 11-13, 2013*, pages 105–114, 2013.

[5] M. A. Cantin, Y. Blaguiere, Y. Sarvaria, P. Lavoie, and E. Granger. Analysis of quantization effects in a digital hardware implementation of a fuzzy ART neural network algorithm. In *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No.00CH36353)*, volume 3, pages 141–144 vol.3, 2000.

[6] M. A. Cantin, Y. Savaria, and P. Lavoie. A comparison of automatic word length optimization procedures. In *2002 IEEE International Symposium on Circuits and Systems. Proceedings (Cat. No.02CH37353)*, volume 2, pages II–612–II–615 vol.2, 2002.

[7] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens. A methodology and design environment for DSP ASIC fixed point refinement. In *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*, pages 271–276, 1999.

[8] Michael Curran. Valuing Asian and portfolio options by conditioning on the geometric mean price. *Manage. Sci.*, 40(12):1705–1711, December 1994.

[9] James W. Demmel, Michael T. Heath, and Henk A. van der Vorst. Parallel numerical linear algebra. *Acta Numerica*, 2:111–197, 1993.

[10] Jack B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, Nov. 1980.

[11] Forbes. *Supercomputer Manages Fixed Income Risk At JPMorgan. [Online]. Available: https://www.forbes.com/sites/tomgroenfeldt/2012/03/20/supercomputer-manages-fixed-income-risk-at-jpmorgan/#376c5e5f1001*.

[12] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[13] John L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, May 1988.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385, 2015.

[15] M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, December 1952.

[16] Intel. *Intel Advisor Profiler. [Online]. Available: https://software.intel.com/en-us/advisor*.

[17] H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: a fixed-point design and simulation environment. In *Proceedings Design, Automation and Test in Europe*, pages 429–435, Feb 1998.

[18] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. Fixed-point optimization utility for C and C++ based digital signal processing programs. In *VLSI Signal Processing, VIII*, pages 197–206, Oct 1995.

[19] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. Fixed-point optimization utility for C and C++ based digital signal processing programs. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 45(11):1455–1464, Nov 1998.

[20] H. T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, January 1982.

[21] Paolo Marchetti, Diego Oriato, Oliver Pell, A M. Cristini, and D Theis. Fast 3D ZO CRS Stack – An FPGA Implementation of an Optimization Based on the Simultaneous Estimate of Eight Parameters. In *72nd EAGE Conference and Exhibition incorporating SPE EUROPEC 2010*, 06 2010.

[22] Micron. *Data Sheet: 2GB, 4GB, 8GB (x72, ECC, DR) 204-Pin DDR3L SODIMM. [Online]. Available: https://www.micron.com/parts/modules/ddr3-sdram/mt18ksf1g72hz-1g6*.

[23] Yoshifumi Nakamura and Hinnerk Stüben. BQCD - Berlin quantum chromodynamics program. abs/1011.0199, 2014.

[24] A. M. Nestorov, E. Reggiani, H. Palikareva, P. Burovskiy, T. Becker, and M. D. Santambrogio. A scalable dataflow implementation of curran's approximation algorithm. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 150–157, May 2017.

[25] Partnership for Advanced Computing in Europe. *PCP-PRACE Research Infrastructure. [Online]. Available: http://www.prace-ri.eu/pcp/*.

[26] O. Pell, J. Bower, R. Dimond, O. Mencer, and M. J. Flynn. Finite-Difference Wave Propagation Modeling on Special-Purpose Dataflow Machines. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):906–915, May 2013.

[27] Changchun Shi and R. W. Brodersen. Automated fixed-point data-type optimization tool for signal processing and communication systems. In *Proceedings. 41st Design Automation Conference, 2004.*, pages 478–483, July 2004.

[28] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2014.

[29] Carlo Tomas, Luca Cazzola, Diego Oriato, Oliver Pell, Daniela Theis, Guido Satta, and Ernesto Bonomi. Acceleration of the Anisotropic PSPI Imaging Algorithm with Dataflow Engines. In *SEG Technical Program Expanded Abstracts 2012*, pages 1–5, 2012.

[30] S. Weston, J. Marin, J. Spooner, O. Pell, and O. Mencer. Accelerating the computation of portfolios of tranched credit derivatives. In *2010 IEEE Workshop on High Performance Computational Finance*, pages 1–8, Nov 2010.

[31] Stephen Weston, James Spooner, Sébastien Racanière, and Oskar Mencer. Rapid Computation of Value and Risk for Derivatives Portfolios. *Concurrency Comput.: Pract. Ex.*, 2011.

[32] Kenneth G. Wilson. Confinement of quarks. *Physical Review D*, 10:2445–2459, Oct 1974.

[33] Xilinx. *Large FPGA Methodology Guide. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/ug872_largefpga.pdf*.

[34] Xilinx. *Performance and Resource Utilization for Adder/Subtracter v12.0. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/c-addsub.html*.

[35] Xilinx. *Performance and Resource Utilization for Divider Generator v5.1. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/div-gen.html*.

[36] Xilinx. *Performance and Resource Utilization for Floating-point v7.1. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html*.

[37] Xilinx. *Performance and Resource Utilization for Multiply Adder v3.0. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ru/xbip-multadd.html*.

[38] Xilinx. *UltraScale Architecture and Product Data Sheet: Overview*.

[39] Xilinx. *UltraScale Architecture Configurable Logic Block. User Guide. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf*.

[40] Xilinx. *UltraScale Architecture DSP Slice. User Guide. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf*.

[41] Xilinx. *UltraScale Architecture Memory Resources. User Guide. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf*.

[42] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. E. Hinton. On rectified linear units for speech processing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3517–3521, May 2013.