University of Copenhagen

Modeling and Implementing High Performance Programs on FPGA

Johannes de Fine Licht (definelicht@nbi.dk)

Supervisors:
Prof. Ken Friis Larsen, University of Copenhagen (kflarsen@diku.dk)
Prof. Torsten Hoefler, ETH Zürich (htor@inf.ethz.ch)

Co-supervisors:
Michaela Blott, Xilinx Research (michaela.blott@xilinx.com)
Sabela Ramos, ETH Zürich (sramos@inf.ethz.ch)

Master Thesis

For the degree of:
Kandidat i datalogi
Master of Science in Computer Science

8th August, 2016

UNIVERSITY OF COPENHAGEN

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

This work investigates the potential of high performance computing (HPC) on field-programmable gate arrays (FPGAs), highlighting concepts and programming techniques to pursue performance using high level synthesis (HLS) tools. We compute the peak single precision floating point performance on the AlphaData 7V3 board using a model of replicated processing elements, then implement a benchmark to verify the predicted performance in hardware, using both the SDAccel framework and a custom reference design provided by Xilinx. The benchmarks reach 302 GOp/s and 548 GOp/s on the two platforms, respectively. The techniques are applied to the field of stencil computations, proposing a temporally pipelined streaming design for the 2D Jacobian stencil that scales with available area on the chip, by using on-chip memory to buffer the incoming wavefront, achieving a sustained performance of 256 GOp/s on a 256×256 grid. Finally the current state of FPGAs is discussed based on the results obtained, and comments are made on the future of reconfigurable computing in HPC.

Acknowledgements

For contributing to this thesis, I would like to thank:

Research Labs at Xilinx Ireland for their generosity and patience, educating a clueless software person in the fine arts of hardware design. In particular: **Michaela Blott**, for taking the initiative, being a great guiding and motivating factor throughout my three month stay in Dublin, and for being the only superior who has ever taken me (and the team) across Europe to a Rammstein concert. **Ken O'Brien** for fruitful collaboration and for quickly getting me set up and fitting me in to the team. **Lisa Liu** for answering countless questions on hardware intricacies and debugging my logic, and **Lisa Halder** for tirelessly working on the reference design necessary to fill the shoes of broken SDAccel.

Xilinx on behalf of the Scalable Parallel Computing Lab (SPCL) at ETH Zürich for providing an AlphaData 7V3 card and all the necessary software licenses to perform in-house development at the university.

My supervisors across Europe. **Torsten Hoefler** in Zurich for helping me come up with an interesting thesis proposal, and offering regular supervision despite his packed schedule and not actually being paid to do so. **Ken Friis Larsen** in Copenhagen for taking on a "risky" master thesis with a student that ran off to Ireland and Switzerland all the time. **Sabela Ramos** for always having time for me, and the rest of SPCL at ETH Zürich for welcoming me in their circle.

The **Irish Centre for High-End Computing** (ICHEC) for providing tools and hardware to build FPGA projects and run SDAccel kernels on their Xavier FPGA node. In particular **Servesh Muralidharan** and **Eoin McHugh** for providing support, and for letting me know when I nearly exceeded available disk space with my Vivado projects, as well as when I *did* exceed available memory and my jobs had to be killed off.

Those who took their time to read through the thesis and provided constructive feedback and suggestions for improvements. In particular **Sabela Ramos**, **Ken Friis Larsen**, **Alexandra Rollings**, **Philip Graae** and the lovely **Philipp Schoppe** for their keen eyes.

Contents

List of Tables

List of Figures

List of Listings

1. Introduction and background

Riding the last nanometers of Moore's law towards the quantum barrier, modern hardware vendors spend their extra transistors on introducing specialized components and replicating existing ones. For high performance computing (HPC), replication is the name of the game, with NVIDIA having gone into the hundreds of processors performing thousands of floating point operations every cycle, and Intel establishing themselves in the business with their Xeon Phi many-core architecture, sporting an order of magnitude more x86 cores than a desktop processor.

A hardware segment that arguably benefits even more from increasing transistor densities is that of reconfigurable computing, with field-programmable gate arrays (FPGAs) being the only prominent example, as they do not have to delegate the additional transistors in the same sense as fixed architectures, but rather can make them available to the end user to assign them where they're most needed. The spread of FPGAs in HPC has been somewhat dampened by the issue of productivity: before the massive surge in popularity induced by the introduction of CUDA, general purpose graphics processing unit (GPU) programming only constituted a sparse group of enthusiasts hacking the graphics rendering pipeline, and while there are now tools aimed at high productivity for FPGAs available, the tipping point of reconfigurable computing reaching the masses has not yet been reached.

This work is written from the point of view of an HPC software programmer gazing into the world of hardware engineering: the field that has so far been the dominant user of reconfigurable computing, and perhaps for a while still will be. The aim is to apply and adapt concepts of HPC to the only broadly available class of reconfigurable architectures, FPGAs. The content presented will inevitably drift into hardware design when pursuing optimal (or even feasible) implementations, as understanding the underlying machinery is crucial to this end, but nonetheless targets the computer scientist reader.

1.1. Thesis structure

The main content of this thesis is divided into five chapters, with every chapter incrementally building on work presented previously:

1. **Background and introduction**: provides information on FPGA architecture, tools available to program them, and derives peak performance for well-known fixed architectures for use with the roofline model.

2. **Programming FPGAs**: treats the engineering aspect of programming FPGAs from the point of view of a software programmer, mapping the applied high level concepts to the resulting hardware.

3. **Performance modeling**: proposes a model for peak performance on FPGA, then measures peak performance of a chip using two different frameworks, as well as memory performance for the board used, comparing them to the predicted numbers.
4. **Stencils**: proposes a scalable high performance design based on the lessons learned, then presents an implementation to achieve this.
5. **Discussion**: evaluates the accuracy and usability of the performance models used, summarizes results and contributions, then discusses productivity on FPGAs and the future of HPC on reconfigurable hardware.

The peak performance and stencil chapters will conclude with individual discussion and future work sections, but are best read in sequence with the other chapters to constitute the road from theory to practice.

1.2. Reconfigurable computing

This section on reconfigurable computing and FPGAs draws on knowledge from literature [1], as well as insights passed on from Xilinx engineers.

Reconfigurable architecture is a general term for microchips that allow at least some post-manufacturing alteration of the on-chip circuitry. Rather than physically adding or removing connections, this is done solely by modifying the data path through the chip's *fabric* (its spatial extent) by configuring on-chip static memory written once at configuration time, making all but the relevant connections redundant. The granularity of components connected can vary widely, from simple logic gates such as AND and NOT, to sophisticated hardened processing and memory units. This granularity varies the trade-off between flexibility and efficiency, as coarser components allow faster and more power efficient implementations of their intended operations by hardening them on the silicon, but restrict the design space of circuits capable of efficiently utilizing the hardware.

FPGAs are a commercially available implementation of a reconfigurable architecture, often described as the middle ground between general purpose fixed architecture hardware, such as CPUs or GPUs, and *application-specific integrated circuits* (ASICs), offering power efficient application-specific circuits without the cost and iteration time of manufacturing ASICs. The peak power consumption of FPGAs is a few tenths of watts, making them suitable for low power environments. Modern FPGAs offer fairly fine-grained components, imposing very few restrictions on the design, but employ some special purpose components to accelerate typical patterns (these will be described in Section 1.4). The high reconfigurability of FPGAs comes at two major trade-offs with respect to fixed architectures, namely clock speed and productivity. Because of the fine component granularity, FPGAs provide dense connectivity between a very high number of components, requiring the clock pulse to travel far on the silicon to pass through the same amount of gates compared to a fixed architecture, increasing the *timing* between stable states (see Section 1.3.1). In terms of productivity, FPGAs live in a completely different paradigm than traditional software design, enabling high performance by utilizing a large amount

of the components on the chip simultaneously, rather than battering fewer, but faster components, as is the norm on fixed architectures. Programming this paradigm poses completely different challenges, and the source to hardware flow is a lot more convoluted and time-consuming, making the iteration time of implementations much longer. The concept of *spatial programming* is crucial to making efficient use of FPGAs, and will be ubiquitous throughout this thesis.

As the implementations presented in this work were carried out using Xilinx tools and hardware, the sections below are based on the state of Xilinx hardware as of writing, but aim at presenting concepts that generalize to other vendors and future hardware iterations whenever relevant.

1.3. FPGA Architecture

The FPGA fabric consists of a large area of interconnected fine-grained components. We consider these components *logic elements*: taking *some* input and producing *some* (or no) output based on their configuration. The most abundant logic elements on contemporary FPGAs are lookup tables (LUTs), as these are Turing complete along with added flip-flops (FFs) to hold state. More specialized components will be described in Section 1.4.

A LUT is a hardware implementation of a boolean truth table: given an N bit input, the table is configured with 2^N bits, enumerating the output for each possible input. A 2^N-way multiplexer then selects the output bit according to the input. An example for XNOR with $N = 2$ is given in Figure 1.1. Chaining sufficiently many LUTs together allows the evaluation of arbitrary logic and arithmetic expressions. A flip-flop is a basic

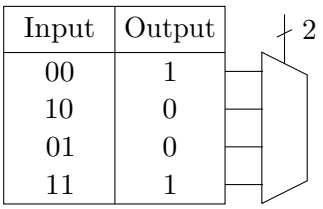


Figure 1.1.: A 2-input LUT implementing XNOR, configured with 4 bits enumerating outputs to each of the 2^2 possible inputs.

storage element capable of storing a single bit of state. In Xilinx hardware these are typically *D flip-flops*, which in addition to storage can act as *shift registers*, that when chained together in a sequence can propagate data through the logic by shifting them forward every clock cycle.

On the Xilinx 7-series [2] architecture used in this work, logic elements are distributed on the board in configurable logic blocks (CLBs), each containing two slices, that each in turn contain four 6-input LUTs and eight FFs. Each 6-input LUT can also be configured as a pair of 5-input LUTs [3] with one FF each. The architecture differentiates between SLICEL and SLICEM logic slices, where the latter has additional capabilities of acting as shift registers or distributed RAM. Shift registers allow SLICEM cells to act as temporary

storage by delaying the propagation of 32 bit values up to 128 clock cycles, whereas distributed RAM effectively assign the logic slices as storage. Buffering on the chip can thus be implemented in different ways (including dedicated BRAM, see Section 1.4.2), offering different properties and consuming different types of resources.

Running a program on this architecture means finding a mapping from the desired algorithm to a data path routed through these components that produced the desired output, the concepts of which we will treat throughout this chapter and the next.

1.3.1. Timing closure

Solving tasks on an FPGA means propagating an input signal through the routed and configured components until a result is produced and output at the other end. This happens in a *pipeline* of stages on the device fabric (see Section 2.3.2), where each electrical signal must propagate to the next step in the pipeline before the following propagating signal can take its place. The time it takes to propagate the signal to the next stage depends on the distance it has to travel through the routed logic, and if it cannot be guaranteed that the signal will reach the next stage within a single clock period, all logic reasoning about the circuit will break down, and any usefulness of the implementation along with it. The process to ensure that this does not happen is called *timing closure*, and achieving it is referred to as *meeting timing*. Timing considerations are typically part of every stage of the source to hardware flow (see Section 1.5), as all mappings and transformations will affect the design's ability to meet timing.

1.4. Hardened components on Virtex 7

This section will describe the two principal hardened components present in the Virtex 7 family of Xilinx FPGAs, namely *digital signal processing* (DSP) slices for computation, and *block RAM* (BRAM) for bulk on-chip memory. While the exact characteristics of these units are specific to Xilinx FPGAs and the Virtex 7 family, they represent the two fundamental aspects most commonly abstracted in coarser components: computation (DSPs) and memory (BRAM).

1.4.1. Digital Signal Processing (DSP) slices

As the name suggests, DSPs are oriented towards signal processing applications, with hardware support for operations acting on particular data widths. The Virtex 7 family specifically hosts the DSP48E1 slice [4], natively supporting operations such as fixed point 25×18 bit multiplication and dual 24 bit addition, pipelined internally in the component. A schematic from the Xilinx user guide is shown in Figure 1.2. While these operations are odd in the context of HPC, DSPs can also be employed to facilitate floating point operations. However, rather than supporting one or more floating point operation natively, DSPs are used in composite Xilinx implementations (see Section 2.3) to handle steps of the floating point operations, reducing the number of logic slices that would otherwise be required for a logic-only implementation. Physically DSP slices are

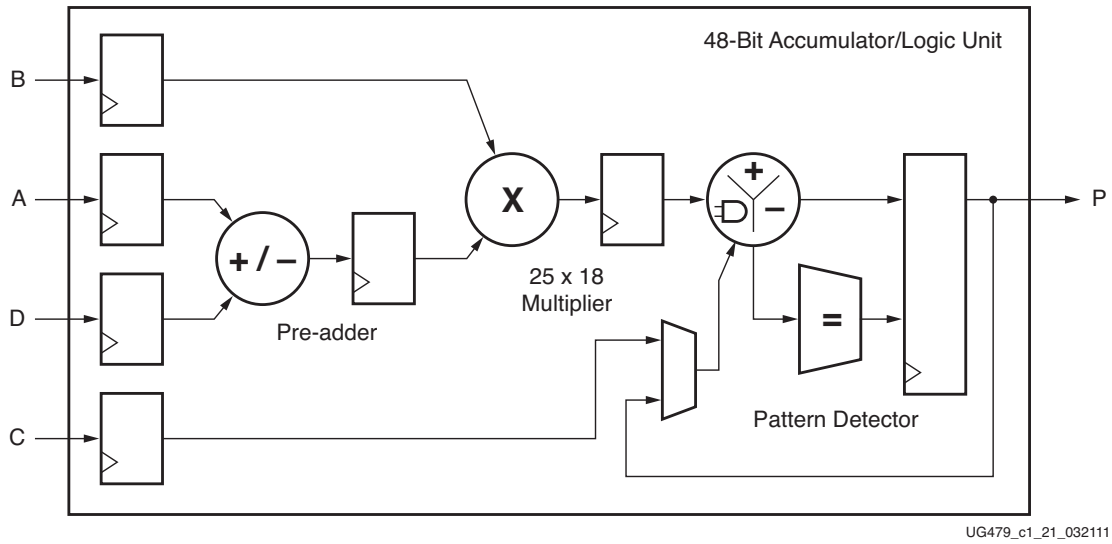


Figure 1.2.: Schematic of the DSP48E1 component for the Virtex 7 family of Xilinx FPGAs [4, Figure 1-1, p. 9].

distributed on vertical lines along the width of the chip as sketched in Figure 1.3. This becomes an important factor when *routing* designs (see Section 1.5.2) for very high area utilization, as the path between logic and DSP slices can get increasingly long, which can introduce timing problems.

1.4.2. Block RAM (BRAM)

BRAM are storage components distributed on-chip. On Virtex 7, BRAM is available as single 36 kbit blocks or as dual 18 kbit blocks, supporting variable I/O widths up to 72 bit [5]. BRAM can be used to store arrays of data elements, or to buffer elements for later use to increase bandwidth to the computational elements. Like DSP slices they are placed in vertical lines along the area of the board, illustrated in Figure 1.3. BRAM can be configured as random access or FIFO memory. By guaranteeing FIFO access the tool can assume a regular access pattern when performing scheduling of pipelined code, as well as providing the programmer with FIFO semantics (see Section 2.4.3).

1.5. Source to hardware flow

Programming an FPGA means mapping the desired semantics to the components inhabiting the target architecture, described by a sequence of configuration bits (called a *bitstream*) that will be written to the FPGA. The flow from source code to hardware capable of performing computations is an intricate process, much less transparent than for software compilation, and constitutes a research field of its own. An overview diagram

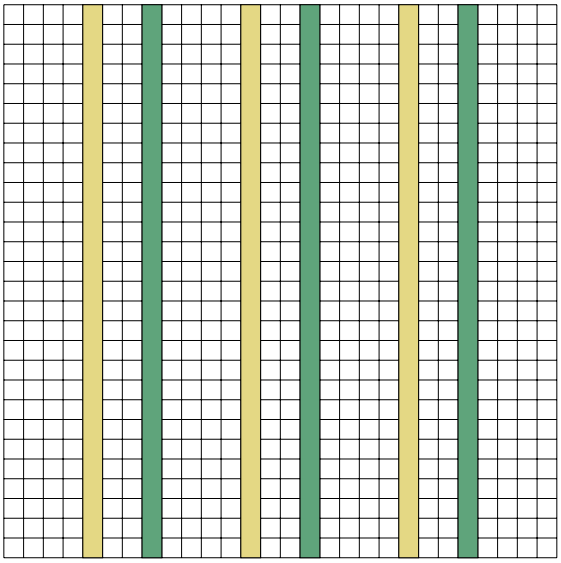


Figure 1.3.: DSP slices and BRAM are distributed in vertical lines along the horizontal dimension of the board, illustrated in yellow and green. This diagram is simplified, and in practice the board is further split into regions with limited connectivity between them.

of the software to hardware flow is included in Figure 1.4, and the major steps and their purpose will be outlined below.

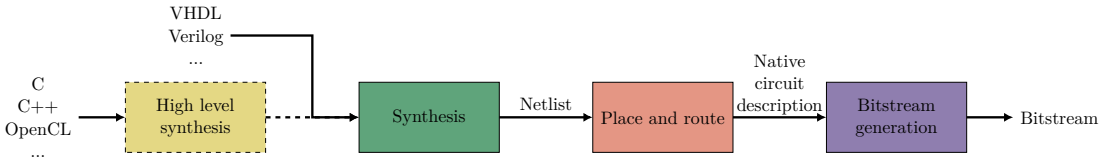


Figure 1.4.: Overview of the steps involved in the source to hardware flow on FPGAs, with the arrows labels indicating the input/output formats. The last three formats are all vendor and implementation specific.

1.5.1. Synthesis

Synthesis is the process of turning the programmer's description of the desired semantics into a description consisting only of components that inhabit the specific target architecture, such as LUTs, FFs, DSPs and BRAM. The source is most commonly written in a *hardware description language* (HDL) such as VHDL or Verilog, describing the circuit at *register transfer level* (RTL), but more recently vendors have started supporting *high level synthesis* (HLS), which takes higher level languages such as C, C++ or OpenCL as input and output HDL, adding an extra step to the compilation flow. The granularity and detail, in which the programmer controls the resulting hardware, is very different

between these two types of input. The Xilinx tools used in this work to synthesize from input source code will be described in more detail in Section 2.1.

1.5.2. Placement and routing

Once the necessary hardware components are known from synthesis, the toolflow has to map these onto the target fabric. This must be done in a way that fits all required instances while minimizing the distance between connected components, in order to reduce the total length of routing required to connect them and meet timing (see Section 1.3.1). Packing components too tightly can however result in *congestion*, where too much routing must run in the same small area, which can subsequently lead to large detours in routing, in turn causing timing problems or fail routing entirely. The quality of placement and routing (often referred to as just *place and route*) therefore depends on a good balance between these two factors. The total number of possible placements of n components is $n!$, and finding the optimal placement is an NP-hard problem. Instead tools rely on heuristics to find good solutions, and improving these algorithms is an active field of research. Because of this heuristic nature, the approach of the HDL or HLS programmer also becomes somewhat heuristic when creating designs, as there is no exact way of determining whether an implementation will place, route and meet timing, apart from obvious boundaries such as outright exceeding the amount of available resources.

1.5.3. Post-routing passes

Because of the amount of heuristics that go into the place and route process, designs require additional passes that can remove routing overlaps, reduce routing distance to improve timing, and reduce resource usage by removing redundancy. While some of these steps are optional in the Xilinx toolflow, optional optimization steps have sometimes proven to be the final nudge needed to meet timing for designs implemented throughout this work, which had otherwise failed due to long routing paths.

1.5.4. Bitstream generation

Once a design has been successfully synthesized, placed and routed, it is transformed to the sequence of bits that must be written to the device SRAM to achieve the implemented circuit. This includes guiding the routing logic to form the paths through the circuit, the truth tables contained in each LUT, configuration of specialized components to activate desired features, and writing constants and default values known at initialization time. Bitstream generation is the final phase of the compilation flow, and the output is the FPGA analogue to binary instructions on a fixed architecture, with the key difference that the bitstream must be written entirely to the device before it is executed. Since binary hardware instructions *are* in fact streamed to the processor during execution, the term *bitstream* is somewhat misleading [1, p. 402].

1.6. Roofline model of computation

The roofline model [6] has, since its introduction in 2008, become a popular tool to express the performance characteristics of hardware. It combines three key concepts in a single plot, namely *peak performance*, *memory bandwidth* and *computational intensity*. A brief introduction to these concepts is given below.

1.6.1. Peak performance

Peak performance is the highest possible throughput of the hardware's computational resources. This is most commonly taken (including in the original roofline paper) as the maximum number of floating point arithmetic operations performed per second, but can be generalized to be the throughput of any useful operation in the target context. Throughout this work performance will be denoted as:

$$F \left[\frac{\text{Op}}{\text{s}}\right] \tag{1.1}$$

where the peak performance F_{peak} is the maximum attainable number for F in a given context. Section 1.7 will compute this number for various fixed architectures, and a model for FPGA will be proposed in Section 3.2.

1.6.2. Memory bandwidth

The memory bandwidth is the maximum rate at which memory can be moved from off-chip memory to the computational resources, and is measured in bytes per second. In practice this usually means moving data from off-chip memory to single cycle accessible memory on fixed architectures. We will denote this as:

$$R \left[\frac{\text{Byte}}{\text{s}}\right] \tag{1.2}$$

This number can be found directly in the datasheet of the memory hardware, but should also be measured by a microbenchmark for comparison. Section 1.6.6 describes how to compute the memory bandwidth for DDR memory.

1.6.3. Computational intensity

Related to the specific application, computational intensity measures the amount of reuse of data loaded from off-chip memory. It is computed as the ratio between the number of useful operations performed and the amount of memory transferred to and from off-chip memory in a given program:

$$I = \frac{\text{Operations performed}}{\text{Bytes transferred}} \left[\frac{\text{Op}}{\text{Byte}}\right] \tag{1.3}$$

This number is by far the hardest to measure accurately on fixed architectures, as it is largely decided by the cache access pattern, which the programmer rarely knows exactly,

especially in multi-layer cache systems that are ubiquitous in modern CPUs. As a first estimate the lower and upper bound for a given application can be computed, assuming no cache and infinite cache, respectively. On FPGA the programmer can handle buffering explicitly rather than leaving it up to general purpose caches, suggesting some merit to this concept applied to FPGAs.

1.6.4. The roofline model

The roofline is defined as the effective throughput of computational elements with peak performance F_{peak} as a function of the number operations per byte transferred to and from off-chip memory I, with a bandwidth to external memory of R:

$$F(I) = \min\{F_{\text{peak}}, R \cdot I\} \tag{1.4}$$

The model owes its name to the shape of the resulting plot, starting as a linear increase with slope R until it becomes constant at the *ridge point* where $F_{\text{peak}} = RI$. Applications with a computational intensity left of the ridge point are said to be *memory bound*, while applications to the right of the ridge point are *compute bound*. This is illustrated in Figure 1.5.

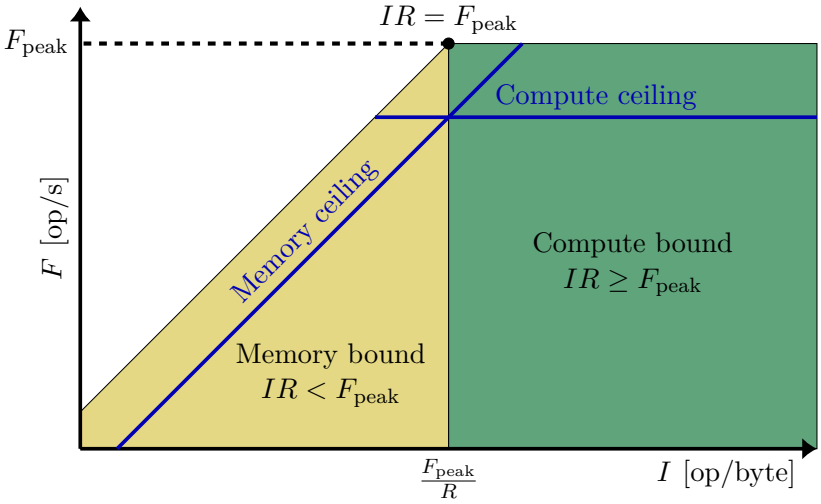


Figure 1.5.: Illustration of a roofline plot. Applications in the memory bound domain are bottlenecked by memory bandwidth, while the compute bound domain is bottlenecked by the throughput of the computational units.

1.6.5. Performance ceilings

Figure 1.5 also illustrates the concept of ceilings. Ceilings are tighter bounds than rooflines, but are allowed to be "breached" by the application. They indicate boundaries imposed by conditions affecting performance. If *fused multiply-add* (FMA) is available,

performing two arithmetic operations in a single instruction, an application that does not utilize them will be upper bounded by a compute ceiling half the height of the peak. If an application accesses off-chip memory by strides larger than the prefetch size (see next section), the memory bound slope will be lowered by a factor of 4 for DDR3 (see Section 1.6.6), and so on.

1.6.6. Computing DDR memory bandwidth

All architectures treated here use a form of double access rate (DDR) random access memory (RAM) as their off-chip memory. The peak memory bandwidth in bit/s is computed as:

$$\text{bandwidth} = \text{memory clock} \times \text{transfer rate} \times \text{bus clock multiplier} \times \text{channels} \quad (1.5)$$

The DDR name comes from prefetching of data: whenever a request is serviced, the memory bank running at the *memory clock* frequency returns multiple words at the target address, known as $2n$-prefetching [7]. The first generation of DDR memory had a prefetch of 2 ($n = 1$), which was accommodated by allowing data transfers on both rising and falling edges of the clock, denoted as the *transfer rate* in (1.5). For DDR2 the amount of data prefetched was increased to 4 ($n = 2$), and to 8 for DDR3 ($n = 4$) and onwards, which is instead achieved by separating the memory clock domain from the I/O clock, increasing the I/O clock by a factor n to service all prefetched words from the memory banks, known as the *bus clock multiplier*. Finally, the number of *channels* are the number of I/O pins serviced at the DDR interface.

Hardware vendors typically declare memory performance either by the full memory bandwidth computed from (1.5), or as the frequency at which data arrives through the I/O pins (twice the I/O clock). This number is a physical upper bound on how many bits can travel between memory banks and the interface. The actual rate at which data arrives to the cache depends heavily on the amount of prefetching and scheduling of requests done by the program to overlap latencies between requests and servicing memory requests, and the amount of data read and written that is *useful* depends heavily on the memory access pattern. The latter factor is incorporated into the roofline model as *computational intensity*, whereas the former will constitute the gap between the actual measured performance and the peak at the given computational intensity, given that the computational intensity is computed exactly (this can be expressed as a ceiling in the roofline model).

1.7. Computing peak performance

On fixed architectures the number of operations performed per cycle is computed by assuming that n computational elements on the chip each have a throughput of $\frac{1}{L}$, where L is the latency of reading in a new set of operands. Even fixed architectures have small internal pipelines, so the latency of individual operations can be higher than one cycle, despite having a throughput of 1 operand per cycle. Hardware vendors usually

report peak floating point numbers as FMA operations that effectively perform two computations packed in a single instruction, so the number of *arithmetic* operations per operand A is also taken into account. Since floating point units are typically capable of performing *single instruction multiple data* (SIMD) instructions that operate on vectors of multiple operands, we include the number of elements that can be treated in parallel K, which for fixed architectures is the SIMD width divided by the operand size, resulting in the expression:

$$C = n\frac{KA}{L} \quad \left[\frac{\text{Op}}{\text{cycle}} = 1 \cdot \frac{\text{operand} \cdot \frac{\text{Op}}{\text{operand}}}{\frac{\text{cycle}}{\text{Op}}}\right] \tag{1.6}$$

To obtain peak performance, we multiply the net number of arithmetic operations performed per cycle by f, the *sustained* frequency that the chip is clocked at. For CPUs and GPUs this frequency can vary, as both architectures can clock higher when allowed by thermal levels (Intel's *Turbo Boost* or NVIDIA's *Boost clock*). We use the base clock, as we are interested in sustained performance. The resulting performance is then:

$$F = fC \quad \left[\frac{\text{Op}}{\text{s}} = \frac{\text{cycle}}{\text{s}} \cdot \frac{\text{Op}}{\text{cycle}}\right] \tag{1.7}$$

We additionally consider performance per power (P):

$$E = \frac{F}{P} \quad \left[\frac{\text{Op}}{\text{J}} = \frac{\frac{\text{Op}}{\text{s}}}{\text{W}}\right] \tag{1.8}$$

This number is more delicate from the point of view of an upper bound, as datasheet power consumption numbers are usually provided as maximum values, thus having the nature of a lower bound in (1.8). While it might not be an unreasonable assumption that power usage is high at peak performance, this is also affected by other factors, such as cache performance and non-floating point units activated on the die. This collision of upper and lower bound concepts should be kept in mind when using datasheet values for power efficiency.

1.7.1. Examples for fixed architectures

In the following we apply Equation (1.6) to common fixed architectures. The model will be extended to FPGA in Section 3.2. Since all numbers below are computed assuming FMA ($A = 2\,\frac{\text{Op}}{\text{operand}}$), the upper bound to many applications will in practice be closer to a roofline ceiling at about half of this performance, as the balance between additions and multiplications is rarely 1:1.

Intel CPU

On an Intel Xeon CPU the number n in Equation (1.6) is the number of cores on the chip. K is the maximum number of operands that can be treated by a single SIMD instruction, e.g. $\frac{256\,\text{bit}}{32\,\text{bit/operand}} = 8$ operands for single precision floating point numbers on the 256

bit wide AVX extensions to the x86 instruction set, or 4 for double precision. A is the number of operations performed per operand, which is 2 for CPUs supporting the AVX2 instruction set that has hardware support for FMA instructions. The latency of FMA instructions is 3 cycles [8], but is pipelined for a throughput of 1. Table 1.1 contains Equation (1.6) and Equation (1.7) evaluated for contemporary CPUs, as well as CPUs released around the time of the AlphaData 7V3 card. The roofline for the Xeon E5-2697

						32 bit			**64 bit**		
Product	**Year**	n	f [GHz]	P [W]	R $\left[\frac{\text{GByte}}{\text{s}}\right]$	C $\left[\frac{\text{Op}}{\text{cycle}}\right]$	F $\left[\frac{\text{GOp}}{\text{s}}\right]$	E $\left[\frac{\text{GOp}}{\text{J}}\right]$	C $\left[\frac{\text{Op}}{\text{cycle}}\right]$	F $\left[\frac{\text{GOp}}{\text{s}}\right]$	E $\left[\frac{\text{GOp}}{\text{J}}\right]$
Xeon E7-8890	2016	24	3.4	165	102	384	1306	7.92	192	653	3.96
Xeon E5-2697	2013	12	3.5	130	59.7	192	672	5.17	96	336	2.59
Core i7-6700K	2015	4	4.2	91	34.1	64	269	2.96	32	134	1.47
Core i7-4771	2013	4	3.9	84	25.6	64	250	2.98	32	125	1.49

Table 1.1.: Peak floating point performance for two contemporary Intel CPUs, as well as for two CPUs contemporary at the time of release of the AlphaData 7V3 card.

is plotted in Figure 1.6.

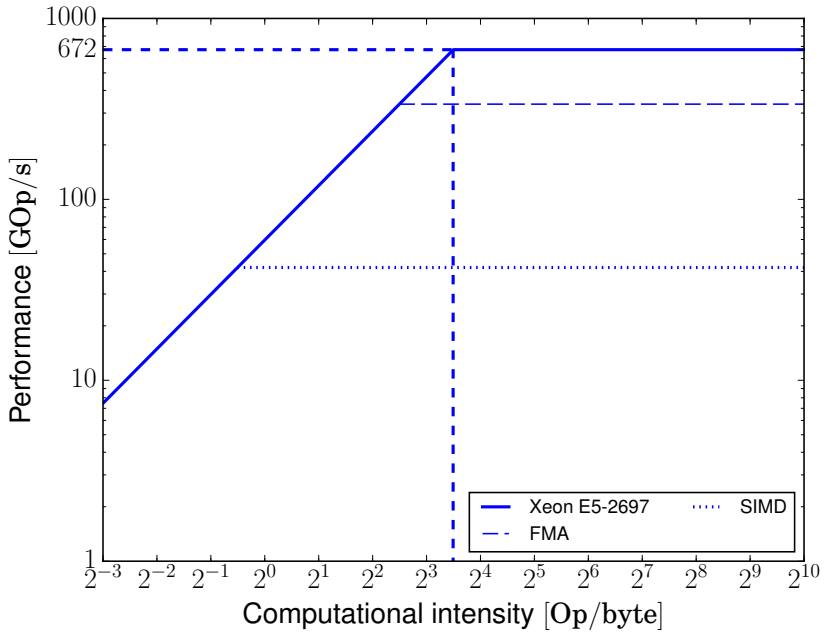


Figure 1.6.: Roofline for the Intel Xeon E5-2697 x86 CPU, including ceilings for FMA and SIMD instructions.

Case: Intel Xeon Phi

For Intel's many-core x86 accelerator cards, Xeon Phi, the numbers in Equation (1.6) are determined similar to above, but with 16 and 8 floats and doubles per instruction

respectively, as the Knight's Corner and Knight's Landing architectures have 512 bit wide vector registers. The numbers are included in Table 1.2. The roofline for the

						32 bit			64 bit		
Product	**Year**	n	f [GHz]	R $\left[\frac{\text{GByte}}{\text{s}}\right]$	P [W]	C $\left[\frac{\text{Op}}{\text{cycle}}\right]$	F $\left[\frac{\text{GOp}}{\text{s}}\right]$	E $\left[\frac{\text{GOp}}{\text{J}}\right]$	C $\left[\frac{\text{Op}}{\text{cycle}}\right]$	F $\left[\frac{\text{GOp}}{\text{s}}\right]$	E $\left[\frac{\text{GOp}}{\text{J}}\right]$
Xeon Phi 7120A	2014	61	1.33	300	352	1952	2596	8.65	976	1298	4.32
Xeon Phi 7290F	2016	72	1.70	260	115.2	2304	3918	15.07	1152	1958	7.53

Table 1.2.: Peak floating point performance for Knight's Corner and Knight's Landing Xeon Phi devices.

Xeon Phi 7120A is plotted in Figure 1.7. The very wide vector units on Xeon Phi result in an even lower SIMD ceiling than for the Xeon procesor.

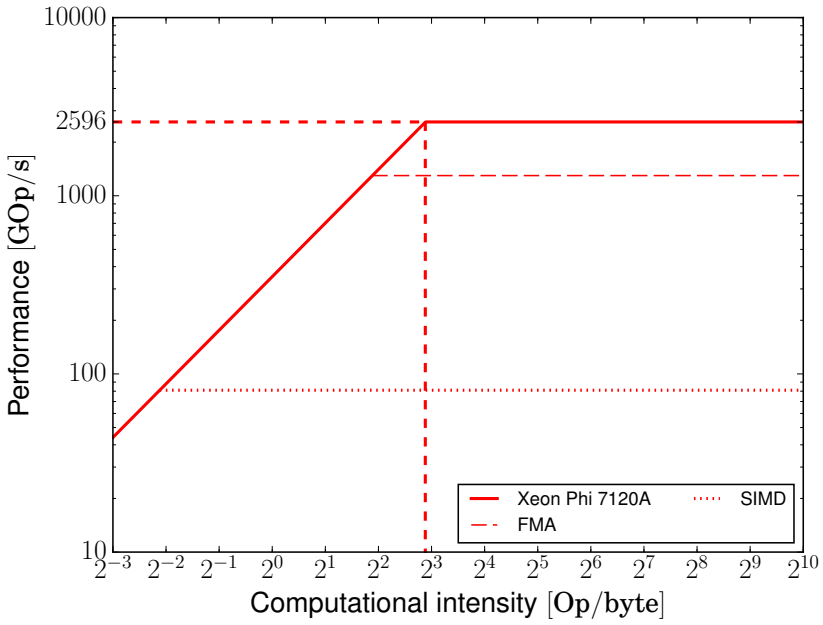


Figure 1.7.: Roofline for the Xeon Phi 7120A x86 accelerator, including ceilings for FMA and SIMD instructions.

Case: NVIDIA GPU

The compute on NVIDIA GPUs is done by a number of *streaming multiprocessors* (SMXes), each with a number of floating point units. Unlike on x86, the number of double precision floating point units is not necessarily half the number of single precision units on one SMX. Table 1.3 lists peak performance numbers computed for both workstation and server cards. The roofline for the Tesla K40 is plotted in Figure 1.8, showing the tremendous floating point performance potential of the GPU.

				32 bit				64 bit			
Product	**Year**	f [GHz]	P [W]	n	C $\left[\frac{\text{Op}}{\text{cycle}}\right]$	F $\left[\frac{\text{GOp}}{\text{s}}\right]$	E $\left[\frac{\text{GOp}}{\text{J}}\right]$	n	C $\left[\frac{\text{Op}}{\text{cycle}}\right]$	F $\left[\frac{\text{GOp}}{\text{s}}\right]$	E $\left[\frac{\text{GOp}}{\text{J}}\right]$
Tesla K20	2013	706	225	13×192	4992	3524	15.7	13×64	1664	1175	5.2
Tesla K40	2013	745	235	15×192	5760	4291	18.3	15×64	1920	1430	6.1
Tesla K80	2016	560	300	$2 \times 13 \times 192$	9984	5591	18.6	$2 \times 13 \times 64$	3328	1864	6.2

Table 1.3.: Peak floating point performance for various NVIDIA Tesla accelerator cards.

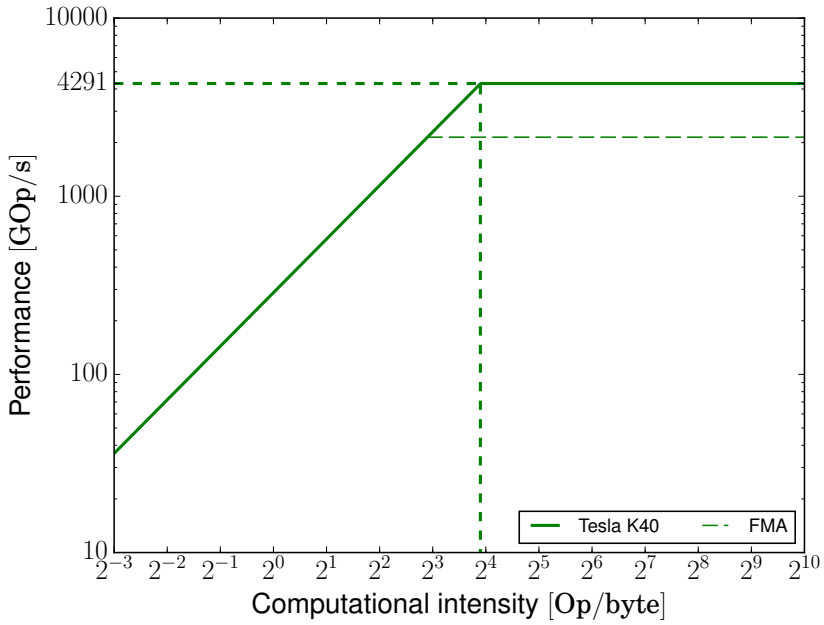


Figure 1.8.: Roofline for the Tesla K40 accelerator.

1.7.2. Discussion

The roofline model is a useful tool for doing quick comparisons of the capabilities of different hardware architectures, and can with the help of ceilings guide optimization efforts, by providing an upper bound of attainable performance for a given algorithm. When more accurate comparisons are required, the variation in tightness of the bound for different architectures becomes hard to factor in, as one architecture might outperform another despite scoring lower on the roofline plot, because the architecture is a better fit. The classic example is irregular memory accesses and branching, which cause some slowdown on a CPU, but completely destroy GPU performance, even if the GPU undisputably wins on the roofline chart, as these factors are not accounted for in the model. Nevertheless, the roofline model is relevant for providing bounds, and Chapter 3 will look into determining the roofline characteristics on FPGA, providing a comparison with the architectures listed here.

Before diving into modeling performance on FPGAs, we will look into the available tools to program them and discuss patterns that produce efficient designs.

2. Programming FPGAs

The exercise of coding for FPGAs is one of *spatial programming*: rather than designing a temporal stream of instructions that will cause an existing circuit to transform the input memory into the desired output, we design the circuit ourselves to produce exactly the output we want for a given input to our target application. When transferred to achieving high performance, the exercise becomes maximizing the amount of FPGA resources concurrently performing useful operations to achieve this on the chip. To this end, the first section will give an overview of the software tools available, before an introduction is given on central concepts to efficient FPGA programs, followed by concepts specific to the high level synthesis tool used, and a proposal for an architecture that fits the FPGA model and will be the base for performant implementations in the following chapters.

2.1. Tools for programming FPGAs

This section describes the Xilinx tools used throughout this work to program FPGA hardware.

2.1.1. Vivado HLS

For all implementations here the core computational kernels were written in Vivado HLS[9], Xilinx' high level synthesis tool. Vivado HLS is capable of synthesizing C or C++ into a hardware description language that will (largely) follow the semantics of the source code. This is represented by the first (optional) step in Figure 1.4, and offers an alternative to writing HDL, intending high productivity FPGA design. The tool is configured with the target FPGA architecture and clock frequency, and takes as input C/C++ source files augmented with HLS-specific pragmas and header files that guide the tool's transformation flow. Vivado HLS has three primary purposes:

1. **Synthesis**: compiles the source file to HDL, reporting transformations done to the user.
2. **C simulation**: runs the source file using a C/C++ testbench as a regular software application to verify correct semantics.
3. **C/RTL co-simulation**: emulates design on hardware by running the synthesized representation along with an instrumented C/C++ testbench to verify correct HDL semantics.

Implementing programs using high level synthesis typically involves many iterations of all three steps above: checking that the program is semantically correct as a C++ program,

verifying that it produces a desirable hardware description, then running the co-simulation to attempt to catch hardware-related bugs in the code. The third step can be very time consuming for large applications, and does not guarantee correct hardware, and thorough testing of all three is primarily done for small prototypes before extrapolating to the full implementation.

Vivado HLS uses the *tool command language* (Tcl) for scripting commands to the command line tool[10]. An example of such a script for running C simulation of a source file then synthesizing to HDL is included in Listing 1.

```
open_project Test
open_solution "virtex7"
set_part "virtex7"
set_top Kernel
create_clock -period 5 -name default
add_files Test.cpp -cflags "-I./"
add_files -tb Testbench.cpp -cflags "-std=c++0x -I./"
csim_design
csynth_design
quit
```

Listing 1: Example of Vivado HLS Tcl script setting up a project targeting the Virtex 7 architecture, adjusting the target clock period, adding source file and testbench, specifying the entry function of the kernel, then running C simulation before synthesizing to HDL.

Synthesizing an HLS program in the version of Vivado HLS used here takes between tenths of seconds for small programs to an hour for many-stage, full utilization programs. Usually synthesizing small programs is sufficient for prototyping and developing the desired architecture, so synthesis for the high resource configuration only needs to be done when building for hardware.

Section 2.4 will give a more thorough walkthrough of the practical aspects of Vivado HLS, describing principles and techniques that can be employed to implement efficient FPGA hardware.

2.1.2. SDAccel

With the increasing popularity of OpenCL[11] for accelerators, FPGA vendors have turned their eyes on supporting OpenCL as a way to approach the mainstream market. Xilinx' effort is called SDAccel, and consists of a tool that compiles directly from source to hardware, as well as a hardware infrastructure on the FPGA fabric that is written as a *partial* bitstream to the chip, hosting a kernel space where user kernels can be placed and executed from, illustrated in Figure 2.1. This is intended to allow programmers to focus on writing the computational kernels, and not bother with the infrastructure specific to each chip. The framework interfaces with DDR memory, allows transfers to and from memory over a PCIe bridge, and exposes a memory interface to the user kernel that can be used to issue requests to the memory controller. SDAccel provides an OpenCL runtime that can be targeted from host code to access DDR memory over the PCIe bridge, and

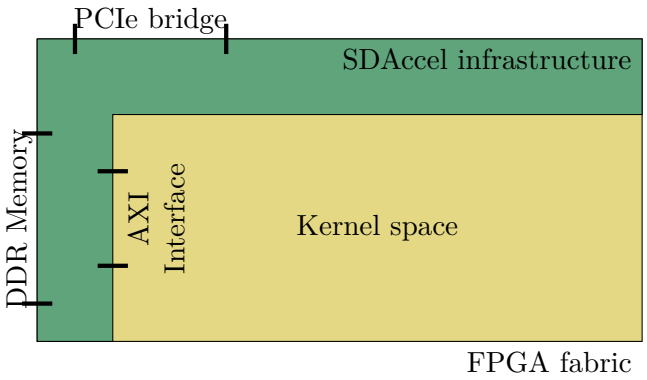


Figure 2.1.: Infrastructure instantiated on the FPGA fabric by SDAccel as a partial configuration. User kernels will be placed and run from the designated user space, interfacing with the SDAccel memory controller to connect to the outside world.

can launch kernels according to the OpenCL model. Like the typical OpenCL flow, but unlike the traditional FPGA flow, the bitstream describing the kernel is written to the device at *runtime*. Kernels generated by the SDAccel build flow are therefore partial bitstreams built to fit the designated kernel userspace, which is a subspace of the FPGA fabric. Unlike other OpenCL platforms SDAccel does not support building kernels from HDL or HLS source at runtime, as this process can take up to several hours (see end of this section).

As illustrated in Figure 2.1, instantiating the SDAccel framework on the chip to provide infrastructure for PCIe and DDR memory requires reserving part of the device fabric, and kernels plugged into the userspace must conform to the SDAccel interface and clock domain to be used. These restrictions are listed below:

- **Clock rate**: the user domain in the SDAccel framework is clocked at 200 MHz, so user kernels must be built with a 5 ns target timing.

- **Memory interface**: the version of the SDAccel toolflow used for writing this work (version 2015.4) provides a single AXI Master interface (see Section 2.4.4) to access address mapped memory, supporting port widths of up to 64 bytes. Along with the clock limit imposed by SDAccel this results in an upper bound of $64\,\text{Byte} \cdot 200\,\text{Hz} = 12.8\,\text{GByte/s}$ on the memory bandwidth between the user kernel and global memory, *regardless of the number of memory modules available* and their I/O clocks. Boards with multiple DDR modules will therefore only be able to utilize a single block from SDAccel kernels.

- **Available logic**: since the infrastructure is instantiated on the FPGA fabric it consumes a number of resources that will not be available to the user program. This can be a significant fraction, and also affects the available number of DSP

slices, which has a great impact on the number of arithmetic operations that can be run concurrently in the user program (this is treated in Section 3.4).

Although SDAccel is marketed as being a target platform for kernels written in OpenCL, experiments performed in the beginning of thesis research suggested that the OpenCL language is a poor match for hardware design, at least with the current state of the tool. While expressing the data path as processing elements and compute units seems fitting, the mapping to pipelines was deemed too opaque, and the examples of efficient implementations encountered seemed to fall back on nested loops in a single worker thread, defeating the purpose of the abstraction. Instead Vivado HLS was chosen as a more suitable abstraction for writing the computational kernels.

Compiling an SDAccel kernel runs all the steps listed in Section 1.5 behind the scenes, packaging the bitstream in SDAccel-specific binaries. This process takes between half an hour and several hours, with the maximum time experienced while developing for this work being in the vicinity of four hours. While it is mostly possible to iron out bugs before running on hardware, some bugs only occurred when running on hardware, being near impossible to debug due to the opaqueness of the SDAccel flow. Kernels however generally compile faster than when building a full Vivado project (see below), as they are more restricted by the SDAccel flow.

2.1.3. Vivado

Vivado is the main workhorse of the Xilinx design suite, and encompasses every stage of the source to hardware flow described in Section 1.5. HLS kernels, HDL components and larger Xilinx intellectual blocks are all instantiated and configured within the application. The tool is meant to give the user detailed control, and provides an extensive user interface that allows tweaking everything from specific FIFO depths to which optimization strategies and constraints should be used during place and route.

In addition to the SDAccel toolflow and infrastructure this work will use a custom hardware design provided by Xilinx implemented as a Vivado project. This design serves as a minimal wrapper around running user kernels, leaving the majority of resources to be used by the kernel (only a few percentages are consumed, depending on the configuration). The design consists of the following major components:

- **PCIe interface**: transfers data over the PCIe bridge in 32-bit words at 125 MHz, and is not meant for high bandwidth use.

- **PCIe input/output FIFOs**: accommodates input and output to/from PCIe, serving as temporary storage for 32-bit words. The input FIFO is populated from the PCIe interface while forwarding to the input data width converter (see below). The output FIFO will be populated by data from the kernel at post-execution after passing through the output data width converter while the data from the output FIFO will be written back over PCIe.

- **Data width converters**: in order to allow narrower and wider pipelines in the user kernel, the 32-bit words from the input/output FIFOs can be converted to/from the required data width before and after passing through the kernel.
- **Kernel input/output FIFOs**: these are the entry and exit point of data streaming to and from the kernel, serving as the main buffers capable of holding the full amount of elements necessary to execute the kernel and results produced by the kernel, respectively.
- **Kernel**: computational kernel provided by the user. Reads from an AXI Stream interface (see Section 2.4.4) and writes back to an AXI Stream interface, only consuming from the input FIFO when a start-flag is set.
- **Performance counter**: in order to accurately measure the number of cycles spent in the kernel, a performance counter is incremented every cycle from when the start flag is set and until the last element is written to the output FIFO.
- **Write path**: connected to the output stream from the kernel, this component detects all elements have been written to the output FIFO (for a perfect pipeline this is the first cycle where no element is written), stopping the increment of the performance counter.

A diagram of the connected components is included in Figure 2.2. The purpose of

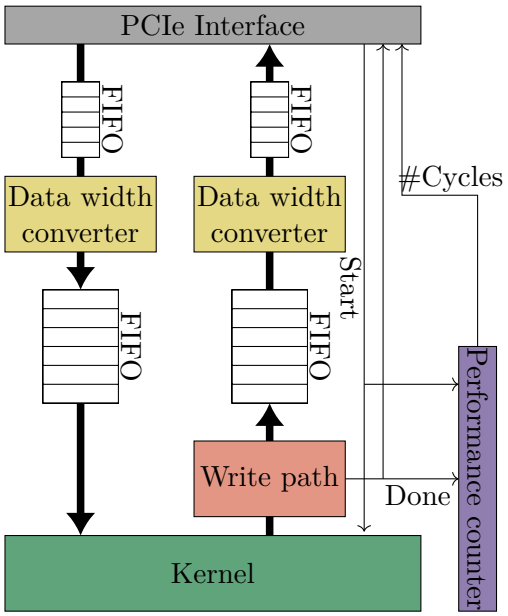


Figure 2.2.: Layout and connectivity of major components in the Xilinx reference design.

this design as an addition to SDAccel is allowing higher resource utilization and clock frequency to push performance further, and to circumvent design restrictions imposed by the SDAccel model.

Rather than calling a single command to build a hardware kernel from source code as is the case with SDAccel, working with Vivado projects requires manually handling every stage of the software to hardware process, as the full kernel plus infrastructure must be synthesized, placed and routed, before writing the result to the device as a self-contained monolithic bitstream. Synthesizing the HLS code, setting up a Vivado project, synthesizing the HDL of the kernel plus infrastructure, placing and routing, optimizing and generating the bitstream takes between an hour and six hours, depending on the size and complexity of the design. As opposed to compiling kernels with SDAccel, the Vivado flow allows starting, stopping and restarting the process at any stage, making debugging individual stages easier, but trades off this flexibility for the total time from source to bitstream. In addition to the long time to completion, the build process carries an enormous memory and disk footprint, with some designs consuming up to 20 GByte of memory while building, and taking up to 4 GByte of disk space in their post-build state. When running on hardware only the bitstream is necessary, however, and these are only in the order of tenths of megabytes.

Running bitstreams built by Vivado requires significantly more effort than SDAccel. The bitstream produced is a full configuration of the device, overwriting anything already present, including its presence to the PCIe driver of the host system. After writing the bitstream to the device using the Xilinx tool Vivado Lab (part of the Vivado Design Suite [12]) the PCIe device must rediscovered. This is done by rebooting the system. A device driver provided by Xilinx must then be installed with sudo privileges, before running a host side application, also provided by Xilinx, that can write to the PCIe bridge and populate the device input FIFO. The host side application offers commands to start, reset and read out results from the kernel, the latter of which will also display the amount of cycles spent in the kernel, which will be used to verify the expected number of cycles to completion.

2.2. Concepts of FPGA programs

Before taking a closer look at the capabilities of the high level synthesis tools, we introduce concepts central to designing FPGA hardware.

2.3. IP cores

In addition to mapping directly from source to logic, Xilinx offers a number of *intellectual property* cores (IP cores), that are composite software-defined components performing more elaborate tasks. When working in Vivado, these are available as configurable blocks that can be added by the user to their design, but are also automatically inserted by the tool when synthesizing implementations in Vivado HLS. A number of data movement-related IP cores are used in the Xilinx reference design (see Section 2.1.3), but the most central IP cores in the scope of this thesis are the provided *floating point IP cores*. As mentioned in Section 1.4.1, DSP units are optimized for low precision fixed point computations, and do not support floating point computations natively. Instead, Xilinx

offers IP cores that perform floating point operations using a combination of logic and some or no DSPs. While not necessary to perform floating point operations, DSP cores can save logic resources when implementing these operations, which will be important for the peak performance computations in Chapter 3. Table 2.1 contains examples of

Operation	Core	DSPs	LUTs	FFs	LUT/FF Pairs	Max clock
32-bit add/sub	Full	2	205	309	126	462
	No	0	355	562	301	556
32-bit multiply	Max	3	79	123	62	462
	Full	2	92	166	75	462
	Med	1	238	363	217	462
	No	0	571	672	538	500
32-bit division		0	793	1354	742	532
64-bit add/sub	Full	3	657	950	541	508
	No	0	689	1067	574	594
64-bit multiply	Max	11	200	492	156	329
	Full	10	233	503	192	462
	Med	9	275	564	245	462
	No	0	2230	2421	2177	368
64-bit division		0	3273	5982	3135	375

Table 2.1.: Suggested resource consumption of Xilinx floating point IP cores, updated as of Vivado Design Suite release 2016.2 [13]. Although we use the 2016.1 release of Vivado in this thesis there should be little to no variation, as the numbers are primarily tied to the Virtex 7 hardware architecture.

suggested resource consumption for various floating point IP cores. These logic resource consumption values have been measured in isolation [13], but will in practice vary with the specific constraints and surrounding logic. The number of internal pipelines stages in the cores will also change depending on the target timing, which impacts the amount of logic required. Additions and multiplications can be implemented using one of multiple possible IP cores with varying amounts of logic versus DSP resources consumed. These are marked with the official Xilinx labels (“Max”, “Full”, “Med” or “No”) in Table 2.1.

2.3.1. Processing elements

When considering instantiating modules on an FPGA it is useful to think of them in terms of *processing elements*. Processing elements are higher granularity elements that can incorporate not just the computation itself, but all surrounding logic necessary to implement the semantics of the application. While processing elements are not an intrinsic FPGA element, they are a useful abstraction when designing algorithms, and we will use this abstraction when defining FPGA performance in Section 3.2.

2.3.2. Pipelining

In addition to maximizing the number of concurrent computations done on the chip, one other critical condition must be fulfilled to achieve performant FPGA programs: each computational element must work at its highest possible throughput. This is achieved by *pipelining*, and any high performance FPGA implementation must implement perfect pipelining in the computational elements to ensure that resources are kept busy at all times by feeding them new input every cycle. This section will first cover the two principal properties of pipelining, *latency* and *initiation interval*, then go through the levels of pipelining that must be treated in a pipelined program.

Latency and initiation interval

A pipelined program will have a number of stages that data will flow through before a result is produced. Latency is the number of cycles between an element entering the first stage of the pipeline, and *that same element* exiting the last stage of the pipeline. The latency is at the same time a measure for how long it takes for the pipeline to saturate and drain, as maximum performance will not be reached before the pipeline is fully saturated, and will degrade as the pipeline is draining.

The number of cycles between the pipeline being able to accept new elements is called the *initiation interval*. In order to achieve full utilization of the compute units, the initiation interval of the pipeline must be 1: it should consume one input (and produce one output) every cycle. Achieving a perfect pipeline means achieving an initiation interval of 1 in all computational stages of the pipeline and being able to feed these pipelines every cycle.

Levels of pipelining

Pipelining comes at three different levels, and all three must be treated in order to achieve perfectly pipelined computation.

1. **Floating point units**: because the Virtex 7 architecture does not have native support for floating point algorithms, operations must be performed in a number of internal stages. This means that the floating point operation itself has a latency greater than one. In order to produce one result every cycle, the floating point units are therefore themselves internally pipelined, and will exhibit a saturation and draining phase, as well as increase the penalty of potential bubbles in the pipeline. To achieve maximum throughput for floating point units it is crucial that all the floating point cores are kept fully saturated.

2. **Modules**: these can be thought of as higher level functions in software engineering, and are the highest level of modularity in an FPGA implementation. Each module has an input and an output interface, and some internal logic to transform the input data to an output. In order to have a perfectly pipelined implementation, data passed from one module to the next must always be ready when it is needed.

In most cases this implies producing and consuming one data element every cycle, although this can in principle be violated if modules performing the computations have internal reuse of data, thus maintaining full utilization of the computational elements.

3. **Dataflow**: The final and strongest level of pipelining is the one across all modules in the implementation, as it requires all modules to already be internally pipelined. We refer to this as *dataflow*, and can be implemented in the Xilinx toolflow using a dataflow optimization pragma (see Section 2.4.7). Functions will act as separate modules connected by either ping pong buffers (requiring the programmer to guarantee that the producer will never push a new element before the previous element was consumed) or FIFO streams. The programmer typically wants to ensure that the pipeline across all modules in the dataflow is perfect, but applications with internal reuse in individual modules can still achieve full computational efficiency as long as the compute is fed.

2.3.3. Bubbles

Although having an initiation interval of 1 cycle throughout the entire pipeline is what allows all computational units to stay saturated, this assumes that data can be fed to the beginning of the pipeline every cycle. If no data arrives in a cycle a *bubble* occurs, which is a signal that will propagate throughout the entire pipeline without producing a useful result, *wasting operations equivalent to the latency of the pipeline* (as every stage of the pipeline will correspond to a non-operation when the signal passes it). Bubbles seriously degrade overall performance, and having a bubble free program is essential to achieve performance with deep pipelines.

A common source of bubbles can be feeding the pipeline from random access memory. The latency from issuing a request until it is serviced can be substantial, although the low latency of FPGAs means they suffer less from this issue than fixed architectures that clock at an order of magnitude higher. In order to stay saturated, requests must be overlapped to hide the latency before they are serviced. The amount of overlapping needed (and the penalty if the overlap is not sufficient) is reduced if a single burst from memory can feed multiple cycles of the pipeline.

Properties of a program that can help prevent bubbles from random access memory are:

- Semantic support of static scheduling of requests, allowing perfect prefetching (requesting data before it is needed). This is mandatory for a bubble free program.
- Regular access pattern that can fully utilize the data returned by each burst.
- Capable of issuing very large requests to a contiguous area in memory, making it trivial for the memory controller to statically schedule overlapping requests.

All of these properties are fulfilled if the program conforms to a streaming interface. This is a supported interface in the Xilinx toolchain, and will be described for HLS programs in Section 2.4.4.

2.3.4. Feedback loops

On the board studied in this work, floating point arithmetic operations are implemented using Xilinx IP cores (see Section 1.4.1). These modules can involve multiple internal pipelines stages, introducing a latency between receiving operands and outputting the result. While this throughputs one arithmetic operation per cycle when fully saturated, it adds latency to when the result of the operation can be used in a future iteration. For fully pipelinable programs this only adds to the delay of the saturation phase, but if implementing iterative structures, *feedback loops* can become an issue. The simplest example is a circuit implementing the sum of an array of floating point numbers. If we attempted to do this using a single floating point addition core accumulating the result in a single register, we would have the wait for the full latency of the floating point operation before the result of one accumulation was ready, effectively reducing the initiation interval (and thus the throughput) to $1/L$, where L is the latency of the operation.

2.4. High level synthesis

The traditional way of programming FPGAs has been HDLs, mostly leaving the task to hardware engineers. With the development of high level synthesis tools from C and C++, as well as the increasing effort to support OpenCL from both Altera and Xilinx, the world of FPGAs is becoming more accessible to users with software backgrounds. Not unlike programming GPUs, however, the road from writing a functionally correct program to writing a performant program is long, and the tools even less mature. This section will take a closer look at the Vivado HLS tool used to produce all kernels presented throughout this work, highlighting important programming techniques and the hardware they produce. The goal is to provide some general insight in the mapping between the high level C++ program that is fed to the tool and the resulting FPGA hardware.

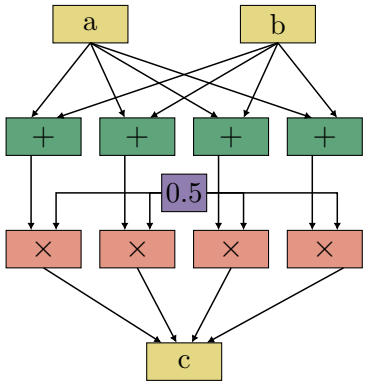
2.4.1. Pipelining

At the heart of the spatial programming paradigm lies pipelining, and building performant HLS programs revolves around expressing the target algorithm in a way that achieves a perfect pipeline (with an initiation interval of 1, see Section 2.3.2). In Vivado HLS pipelines are implemented using the `PIPELINE` pragma, which takes as input the desired initiation interval (the parameter is optional, and if unspecified defaults to 1), and attempts to transform the *scope* in which it is issued to hardware that can throughput one result every cycle when fully saturated. Listing 2 demonstrates the effect of the pipeline and *unroll* (see next section) pragmas by illustrating the resulting hardware from an input code.

2.4.2. Unrolling

The pipeline pragma produces the first spatial dimension: the depth; and unrolling generates the second: the width. Whenever a nested loop appears in a loop that is being

```
void PipelineAndUnroll(const float a[4],
                       const float b[4],
                       float c[4]) {
  #pragma HLS PIPELINE II=1
  Unroll: for (int i = 0; i < 4; ++i) {
    #pragma HLS UNROLL complete
    c[i] = 0.5 * (a[i] + b[i]);
  }
}
```



Listing 2: Source code and illustration of resulting hardware for a pipelined function with an unrolled loop in Vivado HLS. The horizontal dimensions represents unrolled elements with parallel pipelines, while the vertical dimension represents the flow through pipeline stages. The input arrays `a` and `b` are scattered to the processing elements, then gathered after processing before outputting to `c`.

pipelined, the HLS tool will attempt to unroll all iterations of the inner loop in order to execute every stage in parallel. This can only be done if there are no inter-iteration dependencies with a latency longer than the depth of the pipeline. The `UNROLL` pragma used in Listing 2 issues a *complete* unroll (every iteration will be unrolled), but is optional, as pipelining the function will automatically unroll the loop to achieve an initiation interval of 1. Unrolls such as this one control the width of the data path: in the example shown in the listing, the data path is $4 \cdot 4\,\text{Byte} = 16\,\text{Byte}$ wide, as four floats are read in, computed and written out every cycle. This is somewhat analogous to SIMD units on fixed hardware, although there is technically no restriction on performing only a single "instruction" across the entire data path.

2.4.3. Streams

When designing pipelined modules, it is useful to abstract communication between modules as *streams*, only allowing FIFO semantics when reading in and writing out data. Restricting to FIFO behavior requires fewer control paths than random access in hardware, as only a full/empty flag needs to be asserted when data is written/read. An example using high level synthesis is given in Listing 3, using the Xilinx Vivado HLS primitive `hls::stream`, which offers FIFO semantics.

```
void AddOnePointer(float const *input, float *output) {
  for (int i = 0; i < kNumElements; ++i) {
    #pragma HLS PIPELINE II=1
    output[i] = input[i] + 1;
  }
}

void AddOneStream(hls::stream<float> &input, hls::stream<float> &output) {
  for (int i = 0; i < kNumElements; ++i) {
    #pragma HLS PIPELINE II=1
    const float eval = input.read() + 1;
    #pragma HLS RESOURCE variable=eval core=FAddSub_nodsp
    output.write(eval);
  }
}
```

Listing 3: Using stream objects over arrays restricts to FIFO semantics, allowing the tool to produce more efficient hardware.

Vivado HLS allows specifying the depth of stream variables using the `STREAM` pragma, which also makes them useful as buffer primitives. The depth is finite and must be decided at compile-time, as it maps directly to registers or BRAM on the FPGA. Buffers like these can be useful to implement cyclic behavior, such as inner dimensions in a multi dimensional iteration space (this will be useful when implementing stencils). If no pragma is given, the tool is free to implement the stream as a ping pong buffer instead, never allowing more than a single element in flight. An example of using a stream primitive as a buffer is included in Listing 4.

```
const unsigned kPeriod = 32; // Can be defined in external configuration
void FifoBuffer(hls::stream<Burst> &input, hls::stream<Burst> &output) {
  #pragma HLS PIPELINE II=1
  static unsigned i = 0;
  static hls::stream<Burst> buffer;
  #pragma HLS STREAM variable=buffer depth=kPeriod
  #pragma HLS RESOURCE variable=buffer core=FIFO_BRAM
  if (i < kPeriod) {
    buffer.write(input.read());
  } else {
    output.write(buffer.read());
  }
  i = (i + 1) % (2 * kPeriod);
}
```

Listing 4: The HLS stream primitive can be used as a buffer with FIFO semantics by using the `STREAM` pragma to set a FIFO depth. The function shown here periodically loads in a number of elements, then writes them back out to be used at a later stage.

2.4.4. Interfaces

Just like passing larger amounts of variables can be done using either arrays or streams as described above, the entry ports to kernels written in HLS can be implemented as

different types of interfaces. Xilinx offers three different interfaces in their *Advanced eXtensible Interface* (AXI) standard:

- **AXI Lite**: low throughput interface for control flow, used for narrow types that are only read or written once.
- **AXI Master**: a random access interface to addressed memory, using bursts to saturate the bandwidth to e.g. DDR memory. It offers ports for specifying address, burst size and number of bursts to retrieve.
- **AXI Stream**: follows FIFO semantics, eliminating the need to issue requests, as data will simply be streamed at the highest possible rate by statically scheduling very long bursts from memory.

AXI Stream promises the best memory performance, as the sequential semantics allow optimally scheduling the requests to memory to allow maximum bandwidth and prevent bubbles (see Section 2.3.3), but sacrifice random access, requiring the given kernel to follow streaming semantics.

In Vivado HLS interfaces are specified at the top level function (the entry to the kernel) using the `INTERFACE` pragma. An example of this can be found in Listing 8 for AXI Stream interfaces using the `hls::stream` primitive.

2.4.5. Packing bursts

Inside user kernels the data path can be statically scheduled, and the tool will guarantee that data propagates through the pipeline. The pipeline can however only be kept saturated as long as data arrives at the entry point (see Section 2.3.3), which is typically fed from off-chip memory. For a data path with a constant width of K bytes, the pipeline can only stay saturated if K bytes arrive at the entry every cycle, and K bytes are written out at the exit of the pipeline every cycle. The HLS tool must therefore be guided to how much data must be provided from memory per cycle given the relevant bandwidth constraints. Since the user kernel is typically clocked much lower than DDR memory, bursts are also the only way the bandwidth can be saturated.

The version of the Vivado HLS tool used for this work (release 2016.1) does not allow directly specifying the width of the port to global memory when mapping to the AXI Master interface, which is the only interface offered if one wishes to integrate with the SDAccel framework (as of SDAccel 2015.4). Instead the interfaces must be implemented using built-in arbitrary precision types of the desired width, unpacking them as arrays of the desired type, then repacking them before writing them back to the interface. Packing and unpacking are non-operations in FPGA logic, as they are simply reinterpretation style casts from a single, wide type, to an array of narrower types, but must nevertheless be implemented to follow C++ semantics. To this end a burst class was implemented, with the semantics shown in Listing 5.

```
namespace hlsUtil {
template <typename T, unsigned byteWidth>
class Burst {
  static const unsigned elementsPerBurst = byteWidth / sizeof(T);
  ...
public:
  Burst(T const arr[elementsPerBurst]) { Pack(arr); }
  void Pack(T const arr[elementsPerBurst]) { /* Implementation */ }
  void Unpack(T arr[elementsPerBurst]) const { /* Implementation */ }
  void operator<<(T const arr[elementsPerBurst]) { Pack(arr); }
  void operator>>(T arr[elementsPerBurst]) const { Unpack(arr); }
  ...
};
} // End namespace hlsUtil
```

Listing 5: Semantics of C++ burst class, implemented to guarantee wide ports to global memory, allowing multiple elements of the desired type to be read in every cycle. It is also convenient for passing around arrays of elements in the HLS code. The full implementation is included in Appendix A.

With this class, all interfaces are made as reading/writing burst types, regulating the width of the burst along with the width of the data path throughout the program. An example of using bursts to read and write four single precision floating point numbers every single is included in Listing 6, wrapping the function demonstrating pipelining and unrolling, which was presented in Listing 2.

```
typedef hlsUtil::Burst<float, 4*sizeof(float)> Burst;
void Entry(hls::stream<Burst> &aStream,
           hls::stream<Burst> &bStream,
           hls::stream<Burst> &cStream,
           int n) {
  #pragma HLS INTERFACE axis port=aStream
  #pragma HLS INTERFACE axis port=bStream
  #pragma HLS INTERFACE axis port=cStream
  Main: for (int i = 0; i < n; ++i) {
    #pragma HLS PIPELINE II=1
    float aArray[4], bArray[4], cArray[4];
    #pragma HLS ARRAY_PARTITION variable=aArray complete
    #pragma HLS ARRAY_PARTITION variable=bArray complete
    #pragma HLS ARRAY_PARTITION variable=cArray complete
    aStream.read() >> aArray;
    bStream.read() >> bArray;
    PipelineAndUnroll(aArray, bArray, cArray);
    cStream.write(Burst(cArray));
  }
}
```

Listing 6: Example of using bursts to achieve wider port to a C++ kernel written in Vivado HLS, providing four floats to the kernel every cycle, assuming this can be provided by the interface.

2.4.6. Array partitioning

When declaring arrays, the access pattern that must be supported by the contained data can be declared using the `ARRAY_PARTITION` pragma. For treating multiple elements in parallel in an SIMD fashion, every element in an array must be accessed in a single cycle, which can be specified with the `complete` option, shown in Listing 6. Conversely, arrays that are only accessed one element at a time can be implemented as FIFOs and/or in BRAM. The array partitioning pragma is optional, and often the tool can induce the correct partioning from the access pattern in the code, but for codes where the array is accessed from multiple locations it can be desirable to declare it explicitly. There is no direct mapping from the array partitioning to hardware, as the tool can choose implement the array in a variety of ways or optimize it out entirely, so it should merely considered a hint for situations where the most appropriate partioning cannot easily be extracted by the tool.

2.4.7. Dataflow

As described in Section 2.3.2, dataflow is a form of pipelining performed between different modules in a design. This is achieved by issuing the `DATAFLOW` pragma in Vivado HLS, which will attempt to instantiate all functions and loops in the scope in which it is called as *process functions*, which are individual modules connected by ping pong buffers or FIFOs. These connections are explicitly instantiated by the programmer as stream primitives, and must follow the semantics that each stream object is only read from a single process function and written to from a single process function. An example of applying the dataflow optimization in Vivado HLS is included in Listing 7 and Listing 8, demonstrating two different approaches to handling the iteration over elements passed between the dataflow stages. The first, which is required for integrating with SDAccel, is using large loops that communicate using stream primitives, as shown in Listing 7. This approach is problematic for testing the behavior by simulation in C++, because the semantics of the hardware won't follow C++ semantics if there is feedback involved, as it will finish all iterations of one loop before proceeding to the next rather than computing them concurrently. Instead dataflow functions can be implemented as in Listing 8, relying on the function being called once per iteration. This will generate similar hardware to the loop approach, while staying semantically equivalent to the C++ code, as it instead relies on static variables to maintain state. Apart from their C/C++ semantics, static variables have the additional property when used in Vivado HLS that they always map to hardware resources, and thus cannot be optimized away by the compiler. This approach will be used in this work when not restricted by SDAccel.

```
void DataflowLoops(hls::stream<Burst> &in, hls::stream<Burst> &out, int n) {
  #pragma HLS INTERFACE axis port=in
  #pragma HLS INTERFACE axis port=out
  #pragma HLS DATAFLOW
  hls::stream<Burst> pipes[2];
  ComputeFirst: for (int i = 0; i < n; ++i) {
    #pragma HLS PIPELINE II=1
    DoCompute<0>(in, pipes[0]);
  }
  ComputeSecond: for (int i = 0; i < n; ++i) {
    #pragma HLS PIPELINE II=1
    DoCompute<1>(pipes[0], pipes[1]);
  }
  ComputeThird: for (int i = 0; i < n; ++i) {
    #pragma HLS PIPELINE II=1
    DoCompute<2>(pipes[1], out);
  }
}
```

Listing 7: Implementing dataflow by passing data between loops using stream primitives. Each loop will be run concurrently as three different hardware circuits connected by FIFOs (explicitly instantiated as the `pipes` variable).

```
void DataflowStatic(hls::stream<Burst> &in, hls::stream<Burst> &out) {
  #pragma HLS INTERFACE axis port=in
  #pragma HLS INTERFACE axis port=out
  #pragma HLS DATAFLOW
  static hls::stream<Burst> pipes[2];
  DoCompute<0>(in, pipes[0]);
  DoCompute<1>(pipes[0], pipes[1]);
  DoCompute<2>(pipes[1], out);
}
```

Listing 8: Instead of explicitly looping over data, static variables can be used to express the FIFOs carrying data through the functions in the dataflow scope, consuming all the data by repeatedly clocking the hardware generated by the code.

Since the dataflow optimization only allows writing to a given connecting stream from a single dataflow function, feedback in the flow must be implemented with a demultiplexing function that chooses an input but writing to the same output, as shown in Listing 9.

```
void DemuxInput(hls::stream<Burst> &input, hls::stream<Burst> &feedback,
                hls::stream<Burst> &output) {
  #pragma HLS PIPELINE II=1
  if (!input.empty()) {
    output.write(input.read());
  } else if (!feedback.empty()) {
    output.write(feedback.read());
  }
}
```

Listing 9: Since dataflow only allows writing to a given stream from a single dataflow function, a separate demultiplexing function as the one shown here must be used to choose the input.

Implementing writing to the feedback loop is conversely done by multiplexing the incoming data to either the feedback loop or the output. This function has the responsibility of stopping the feedback after the desired amount of iterations. An example is shown in Listing 10. This code also demonstrates the `RESET` pragma, which instructs the tool to wire a reset signal to the specified variable, causing it to return to its default state if a reset flag is sent to the encapsulating kernel.

```
void MuxOutput(hls::stream<Burst> &input, hls::stream<Burst> &feedback,
               hls::stream<Burst> &output, const unsigned feedbackIterations) {
  #pragma HLS PIPELINE II=1
  static unsigned i = 0;
  #pragma HLS RESET variable=i
  if (!input.empty()) {
    Burst read = input.read();
    if (i < feedbackIterations) {
      feedback.write(read);
    } else {
      output.write(read);
    }
    ++i;
  }
}
```

Listing 10: Dataflow element dedicated to writing results to a feedback loop rather than outputting it to e.g. memory. It will start outputting results when the desired number of iterations have been performed.

It should be noted that implementations using feedback of data as shown in Listing 9 and Listing 10 have caused issues for large resource usage, causing the design to hang. This method should therefore be used with caution, as these issues have not yet been resolved.

2.4.8. Recursive templates

When programming spatial designs, the exact mapping from the high level language to the underlying hardware is opaque. If a loop containing a function call is pipelined, Vivado HLS can choose to generate hardware for every iteration of the loop, but can also reuse hardware across multiple iterations if allowed to by the latencies of the operations. When the goal is to maximize performance by making pipelines as deep as possible, the latter behavior is undesired. For the designs using the Xilinx reference design presented later using the dataflow optimization, a recursive template approach was used to unroll into functions that map one-to-one to function calls, thereby forcing the tool to generate separate hardware for each pipeline stage. The technique is shown in Listing 11. As was already suggested in Listing 7 and Listing 8, changing the value of a dummy template parameter will generate unique instantiations, and we can use a recursive template to produce `kDepth` separate instances of the function as hardware using tail recursion.

```
static const unsigned kDepth = 3; // Depth defined at compile-time

template <unsigned recurse>
void UnrollCompute(hls::stream<Burst> pipes[kDepth]) {
#pragma HLS INLINE
  UnrollCompute<recurse - 1>(pipes); // Tail recursion
  DoCompute<recurse - 1>(pipes[recurse - 1], pipes[recurse]);
}
template <> void UnrollCompute<0>(hls::stream<Burst> *) {} // Bottom out

void StreamToStream(hls::stream<Burst> &in, hls::stream<Burst> &out) {
  #pragma HLS PIPELINE II=1
  out.write(in.read());
}

void DataflowStaticUnroll(hls::stream<Burst> &in, hls::stream<Burst> &out) {
  #pragma HLS INTERFACE axis port=in
  #pragma HLS INTERFACE axis port=out
  #pragma HLS DATAFLOW
  static hls::stream<Burst> pipes[kDepth + 1];
  StreamToStream(in, pipes[0]);       // Forward to first pipe
  UnrollCompute<kDepth>(pipes);       // Recursive template call
  StreamToStream(pipes[kDepth], out); // Forward from last pipe
}
```

Listing 11: Recursive templates can be used to instantiate unique functions using tail recursion, forcing the tool to generate separate hardware for each stage. This program implements the same functionality as Listing 8, but can be configured to automatically produce separate functions for any (potentially large) value of `kDepth`.

Recursive templates are also useful for implementing hardware for tree reductions. By halving the number of elements treated in the hardware at each stage and bottoming out when two elements are reached, Listing 12 implements a fully pipelined binary reduction, collapsing `width` elements of type `T` using `Functor` as the accumulator.

```
template <typename T, class Functor, unsigned width>
class BinaryReduce { // Wrap in class to allow partial specialization
  static const unsigned halfWidth = width/2; // HLS not fully C++11 compliant,
public:                                      // so no constexpr available.
  static T Apply(const T array[width]) {
    #pragma HLS PIPELINE II=1
    T reduced[halfWidth];
    #pragma HLS ARRAY_PARTITION variable=reduced complete
    for (unsigned i = 0; i < halfWidth; ++i) {
      #pragma HLS UNROLL complete // Not required as pipeline pragma will
                                  // automatically try to unroll.
      reduced[i] = Functor::Apply(array[i], array[i + halfWidth]);
    }
    return BinaryReduce<T, Functor, halfWidth>::Apply(reduced); // Recursion
  }
};
template <typename T, class Functor>
class BinaryReduce<T, Functor, 2> { // Bottom out recursion
public:
  static T Apply(const T array[2]) {
    #pragma HLS PIPELINE II=1
    return Functor::Apply(array[0], array[1]);
  }
};
```

Listing 12: Implementing fully pipelined tree reduction hardware for arbitrary types and functors in hardware using template recursion.

Listing 13 shows an example of a kernel using tree reduction to collapse bursts of width `kWidth` to a single element using an addition functor. The code will generate $\log_2(\text{kWidth})$ stages, each internally pipelined, and produce the hardware shown in Figure 2.3.

```
template <typename T>
class Add { // Addition functor to pass to reduction template
public:
  static T Apply(T const &a, T const &b) { return a + b; }
};

const unsigned kWidth = 8; // Could be defined by external configuration
void SumReduce(hls::stream<Burst<float, kWidth*sizeof(float)>> &in,
               hls::stream<float> &out) {
  #pragma HLS PIPELINE II=1
  float unpacked[kWidth];
  #pragma HLS ARRAY_PARTITION variable=unpacked complete
  in.read() >> unpacked; // Read in 8*4=32 bytes
  float reduced = BinaryReduce<float, Add<float>, kWidth>::Apply(unpacked);
  out.write(reduced);    // Write out 4 bytes
}
```

Listing 13: Kernel using the recursive tree reduction implementation from Listing 12 to collapse 8 floats to their sum using an addition functor. The resulting hardware is shown in Figure 2.3.

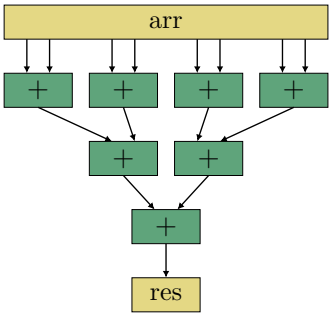


Figure 2.3.: Resulting hardware from the code included in Listing 13, performing a reduction of 8 floats into their sum in a fully pipelined tree structure, where the vertical dimensions indicates the flow of the pipeline.

2.4.9. Specifying resources

For many operations the desired functionality can be implemented in multiple ways using the available resources on the FPGA fabric. The tool will attempt to determine the most appropriate implementation for each case. Sometimes the programmer might wish to assign a particular IP core (see Section 2.3), and to this end the `RESOURCE` pragma is available. Two cases already encountered in this work are choosing floating point cores and storage cores. Since the choice of implementation affects which resources and how many of these resources are consumed, it can affect to total number of operations that can be instantiated on the chip (this will be treated in more detail in Section 3.2). For storage, when smaller amounts of memory are required and only have to be stored for a small amount of cycles, the tool will often choose to implement such buffers by propagating these values through the flip-flops in the logic resources rather than writing them to a dedicated storage element. When attempting to maximize the number of logic used for performing computational tasks, however, it is more desirable to make use of the dedicated BRAM resources, to avoid competing for general logic resources. An example specifying a FIFO to be implemented using BRAM is included in Listing 4, and Listing 3 shows specifying an addition to be implemented without using any DSP resources.

2.5. Streaming pipeline architecture

We will now define the *streaming pipeline* architecture, which will be used for the implementations in this work. It is related to the *systolic array* architecture [14], where a global clock drives data through an array or grid of interconnected processing elements, statically scheduled to solve the target application once the input has propagated through all elements.

A streaming pipeline is a single deep pipeline on the FPGA fabric, but with the following additional properties:

1. The combined flow throughout the streaming pipeline is **statically scheduled**: the rate at which input is consumed and output is produced is constant in the saturated state.

2. The pipeline consists of coarse-grained **processing elements**, which may in turn be internally pipelined, but only need to meet the condition of producing and consuming at a *constant rate* to allow static scheduling.

3. All *computational* units within the processing elements must be **fully pipelined**, so that they work at their maximum throughput in the saturated state.

4. There is only a **single entry** and a **single exit** for data flowing to and from the pipeline, each of which work at a constant rate.

5. Communication between processing elements is unidirectional, and the flow of data through the elements constitute a **directed acyclic graph**.

The rate at which data is consumed and produced by the entry and exit points, or any processing elements in the pipeline, does not have to be the same, as long as the rates are individually constant over some number of cycles ≥ 1. For example, a pipeline that accepts a new input every cycle, but only produces an output for every eight inputs, is still a streaming pipeline, assuming the other conditions are met.

The performance of a streaming pipeline program will be investigated in the next chapter.

3. Performance modeling

In order to gain insight in the performance characteristics of pipelined FPGA programs, we model and measure the performance of replicating identical processing elements to maximize resource utilization on the device. As we will later target the stencil class of algorithms that compute linear combinations, we will treat floating point processing elements with varying combinations of additions and multiplications. Stencils will be described in more detail in Chapter 4. We will however make an argument that the maximum of these numbers will indeed correspond to peak floating point performance (see Section 3.2.3). To verify the theoretical numbers a synthetic benchmark is designed to run on hardware, along with a memory benchmark to measure DDR bandwidth. These will be combined to produce the roofline model for the target chip, in order to compare it to the fixed architectures that were included in Section 1.7.1. While this work focuses on single precision floating point types and target operations used in the stencil domain, the methods apply directly to other operations, as well as other precision floating point types, fixed point and integer computations. Other types and domains will be left as future work (see Section 3.10). Practical benchmarks for fixed architectures will not be performed here, as this is covered extensively in other research.

3.1. Related work

Other authors have modeled or measured floating point performance on FPGAs. Strenski et al. [15] do an analysis of floating point performance on FPGA, using an approach predicting performance numbers similar to this work, dividing predictions into a peak section and a realistic section, but do not run benchmarks on hardware to evaluate the quality of their predictions. Servesh et al. [16] focus on the experimental aspects of performance and power consumption using the roofline model, comparing GPU, Xeon Phi and FPGA accelerators for a single algorithm implemented in OpenCL. da Silva et al. [17] also base their work on the roofline model, predicting performance numbers with a processing element model, but do not run experiments on hardware.

This chapter will include both modeling and prediction of performance and experimental evidence.

3.2. FPGA model of peak performance

Achieving peak performance on an FPGA is centered around maximizing utilization of the available resources on the reconfigurable fabric. In this section we specifically consider the performance achieved when *replicating identical processing elements* in a streaming

pipeline architecture (see Section 2.5), as applications that wish to scale should be able to consume all resources on the FPGA fabric, and these elements must be pipelined in order to utilize their full performance. Performance on an FPGA follows most of the same concepts as in Section 1.7, with the number of operations per cycle determined by Equation (1.6), and the peak performance by Equation (1.7). The difference lies in determining the numbers n and K, that for fixed architectures denoted the number of computational elements and SIMD width, respectively. Their meaning when applied to FPGAs will be analogous, but they will now be correlated, and we will instead refer to them as the *depth* and *width* of the pipeline. They are correlated because expanding either dimension increases the resource utilization of the device fabric, and the highest attainable performance is achieved at some constant equal nK, where n and K can vary. We consider a processing element as the basic component of computation, which we replicate in the n and K dimensions, with the number of arithmetic operations performed by one of these units expressed by the number A in Equation (1.6). The highest attainable performance of a program directly relates how many of these processing elements, characteristic to the application, can be instantiated on the chip, which we will treat in the following.

The highest number of processing elements N that can be instantiated on a chip with available resources $D_{\max}$, $L_{\max}$ and $B_{\max}$ (denoting DSPs, LUTs and BRAM, respectively) is:

$$\begin{aligned} N_{\max}(\text{PE}) = \text{Maximize } & N \\ \text{subject to } & N \cdot \text{PE}.D \leq D_{\max} \\ & N \cdot \text{PE}.L \leq L_{\max} \\ & N \cdot \text{PE}.B \leq B_{\max} \end{aligned} \tag{3.1}$$

where PE.D, PE.L and PE.B are integer counts of DSPs, LUTs and BRAM in a single processing element. The maximum number of operations per cycle is then substituting $nK = N_{\max}(\text{PE})$ in Equation (1.6) and assuming an initialization interval (see Section 2.3.2) of 1, setting $L = 1$:

$$C_{\text{FPGA}}(\text{PE}) = \text{PE}.A \cdot N_{\max}(\text{PE}) \tag{3.2}$$

Here we have additionally defined PE.A as the number of arithmetic (or otherwise useful) operations performed per cycle by a single processing element, analogous to A in Equation (1.6). The peak number of operations per cycle for a pipelined program is then the maximum number of operations attainable among any processing element:

$$C_{\text{FPGA,peak}} = \max_{\text{PE}} \{\text{PE}.A \cdot N_{\max}(\text{PE})\} \tag{3.3}$$

To compute the peak *performance* the frequency f is introduced, denoting the frequency at which the FPGA fabric containing the computational elements is clocked, and is itself an optimization parameter. The frequency is however tightly coupled to the individual board, implementation, computational units and routing algorithm (see Section 1.5.2), and has no useful upper bound, thus offering no direct way of optimization. This leaves

the peak performance as an engineering problem with f and PE as coupled parameters:

$$F_{\text{FPGA,peak}} = \begin{array}{l}\text{Maximize } f \cdot C_{\text{FPGA}}(\text{PE}) \\ \text{subject to design meeting timing.}\end{array} \tag{3.4}$$

In the following we treat the highest attainable performance obtainable with distributions of operations required by typical floating point programs (in particular stencil computations), namely additions and multiplications. This gives a tighter upper bound for the particular configurations and is done easily using Equation (3.1), but we will also argue why this is an indicator of peak floating point performance. In addition, the infrastructures used to implement algorithms in this work restrict the algorithm to use their clock domains, fixing f from the framework. This reduces the problem of peak performance, given our domain, to using Equation (3.1) in Equation (1.7).

3.2.1. Streaming pipeline computational intensity

The streaming pipeline model introduced in Section 2.5 defined an architecture based on a deep pipeline with a number of additional properties. We will now look at the computational intensity for a streaming pipeline implementation of a program on the form of replicated identical processing elements modeled above.

Considering a pipeline of some width K and depth n, the number of operations performed per cycle in the pipeline is simply given by Equation (1.6) with $L_{\text{SPL}} = 1\,\text{cycle/Op}$ from the third property of streaming pipelines.

The fourth property of the streaming pipeline architecture states that the pipeline has a single entry and a single exit point, and that the rate at which data flows through these is constant. The memory bandwidth required to sustain the pipeline is therefore the sum of these two numbers:

$$R_{\text{SPL}} = R_{\text{entry}} + R_{\text{exit}} \tag{3.5}$$

We note that this number is independent of the remaining processing elements in the pipeline, and is therefore constant with the number of processing elements. For a pipeline of a data path with constant width of K of size B_t elements, the required bandwidth becomes:

$$R_{\text{SPL,K}} = 2KB_t f \quad \left[\frac{\text{Byte}}{\text{s}} = \frac{\text{operand}}{\text{cycle}} \cdot \frac{\text{Byte}}{\text{operand}} \cdot \frac{\text{cycle}}{\text{s}}\right] \tag{3.6}$$

where f is the frequency of the pipeline's clock domain. The pipeline needs KB_t bytes to arrive every clock with period $1/f$ to stay saturated, and the same amount of bytes to leave every clock period.

We can write down the computational intensity, using the amount of operations performed by cycle as Equation (1.6), and the number of bytes required per cycle as R_{SPL}/f:

$$I_{\text{SPL}} = \frac{nKA/L_{\text{SPL}}}{(R_{\text{entry}} + R_{\text{exit}})/f} \quad \left[\frac{\text{Op}}{\text{Byte}} = \frac{1 \cdot \text{operand} \cdot \frac{\text{Op}}{\text{operand}} / \frac{\text{cycle}}{\text{Op}}}{\frac{\text{Byte}}{\text{s}} / \frac{\text{cycle}}{\text{s}}}\right] \tag{3.7}$$

For the case of a pipeline with constant width of KB_t bytes we use Equation (3.6):

$$I_{\mathrm{SPL,K}} = \frac{nKA/L_{\mathrm{SPL}}}{2KB_t} = \frac{A/L_{\mathrm{SPL}}}{2B_t}n \tag{3.8}$$

This implies that by increasing the depth of the pipeline (n), we can increase the computational intensity without changing the width of the data path (K). The required memory bandwidth of such a pipeline is given by Equation (3.6) and is independent of the pipeline depth.

If we can construct a program as a streaming pipeline, and put in a form that can arbitrarily scale n for a constant width K, the computational intensity will simply scale with the depth of the pipeline, as the memory bandwidth required is constant.

3.2.2. Time to completion

We can express the time to completion of a pipeline in terms of the concepts introduced in Section 2.3.2. The number of cycles (c) required to execute the pipeline is the number of cycles it takes to push all n elements to be processed into the entry with initiation interval V, plus the latency L before the last element leaves the exit:

$$c(n) = Vn + L \tag{3.9}$$

The time to completion depends on the clock rate f:

$$T(n) = \frac{c(n)}{f} \tag{3.10}$$

If we assume a perfectly pipelined program with $V = 1$ and assume a typical application in HPC where the number of elements that are processed is large, Equation (3.9) reduces to simply:

$$c(n) \approx n \quad \text{for } V = 1 \wedge n \gg L \tag{3.11}$$

While Equation (3.11) might seem enticing, it assumes that a single traversal of the pipeline can solve the problem for an element in its entirety. While this would be the case for arbitrarily large hardware (or very small problems), we will in practice be restricted by the amount of area available on the chip. The number of cycles (and thus the time) to completion will then depend on how we can fold a problem of size $N \leq n$ to minimize the size of n in Equation (3.11). While the number n needs to be determined from the algorithm, this has one important implication: given a perfectly pipelined program, and assuming data arrives every cycle, the time to completion will be *exactly* as given in Equation (3.10), and can be determined using Equation (3.11) to a very good approximation for any large problem.

3.2.3. Domain treated

With the application of stencils in mind, we look at processing elements of n_+ additions and $n_\times$ multiplications, as evaluating stencils consists of evaluating linear combinations of neighboring cells (see Chapter 4). Since divisions (and more advanced functions such as exponential and trigonometry) use strictly more logic resources than the no-DSP implementations of additions and multiplications [13], a processing element with other operations would never surpass one of just additions and multiplications in performance, so we additionally claim that the numbers computed here reflect true peak performance for single precision floating point types. Including a more diverse set of operations is however proposed as future work (see Section 3.10.2). Additions and multiplications can be implemented using different floating point IP cores. Referring to the addition and multiplication IP cores from Table 2.1 with indices i and j respectively, we choose the peak performance of the given configuration as the maximum evaluation of Equation (3.1) among all permutations:

$$N_{\text{peak}} = \max_{i,j} \{N_{\max}(\text{PE}_{i,j})\} \tag{3.12}$$

where $PE_{i,j}$ has the costs:

$$\begin{aligned} PE.D &= n_+ D_i + n_\times D_j \\ PE.L &= n_+ L_i + n_\times L_j \\ PE.B &= 0 \end{aligned} \tag{3.13}$$

with n_+ and $n_\times$ fixed to the given configuration. In addition to the resulting number of processing elements and corresponding performance, the tables below will specify the configuration of floating point IP cores that maximized Equation (3.12).

3.3. Target hardware platform

As the target for performance predictions and measurements in this work, the Alpha-Data ADM-PCIE-7V3 [18] (or AlphaData 7V3 for short) board is used, hosting a Virtex 7 XC7VX690T FPGA, two 8 GB DDR3 DIMMs clocked at 1333 MHz, as well as ethernet and SATA interfaces (the latter two will not be used in this work). The Virtex family is oriented towards high performance applications, sporting the largest amount of logic on the chip in the 7-series of Xilinx FPGAs. Datasheet resource counts are included in Table 3.1.

	LUTs	**FFs**	**DSPs**	**18 Kb BRAM blocks**
Out of the box	433200	866400	3600	2940
SDAccel (estimated)	303240	606480	2520	2058

Table 3.1.: Available resources on the Virtex 7 XC7VX690T chip [19]. The chip has 108300 logic cells, each with four LUTs and eight FFs, resulting the numbers above. When using the SDAccel infrastructure, $\approx 30\%$ of the chip is reserved for the framework, estimated on the second row.

3.4. Expected performance

We will now apply Equation (3.1) to the domain of single precision floating point additions and multiplications for different assumptions: datasheet values for floating point IP cores, the restrictions imposed by the SDAccel framework, and the context of the provided Xilinx reference design.

3.4.1. Datasheet peak performance

Based solely on the datasheet values for available resources, maximum clock speed of operations, and assuming full area utilization is possible, Table 3.2 contains upper bounds for attainable performance on the XC7VX690T chip. The highest performance

$\frac{\text{Add}}{\text{PE}}$/IP	$\frac{\text{Mult}}{\text{PE}}$/IP	#PEs	Ops/cycle	DSPs	LUTs	GOps/s 462 MHz
1/Full	0	1800	1800	1.00	0.85	832
4/Full	1/Med	400	2000	1.00	0.98	924
3/Full	1/Med	507	2028	0.99	1.00	937
2/Full	1/Med	668	2004	0.93	1.00	926
1/No	1/Max	998	1996	0.83	1.00	922
1/No	2/Full	803	2409	0.89	1.00	1113
1/No	3/Full	600	2400	1.00	0.87	1109
1/No	4/Full	450	2250	1.00	0.75	1040
1/No	5/Full	360	2160	1.00	0.68	998
0	1/Med	1820	1820	0.51	1.00	841

Table 3.2.: Upper bound on operations per cycle and per second for the Virtex 7 XC7VX690T chip for floating point units, determined between any permutations of floating point IP cores from Table 2.1, but with identically replicated processing elements, and assuming the FPGA clocked at the highest rate reported in the datasheet for the required IP cores.

is predicted for processing elements with more multiplications than additions, with the highest predicted number for 1 addition per 2 multiplications. The numbers presented in Table 3.2 are unlikely to represent any realistically attainable performance, as meeting timing for 462 MHz is not possible outside of isolated toy scenarios. These numbers thus have the nature of a loose upper bound.

3.4.2. SDAccel peak performance

From the two platforms used for benchmarking, we first treat the SDAccel framework (see Section 2.1.2), intended to allow software developers to quickly implement and test solutions on FPGA hardware, without worrying about implementation details of the infrastructure. The accessibility of SDAccel comes at two major trade-offs: constraints on the kernel and flexibility in the design. The kernel area is constrainted to be clocked at 200 MHz, and $\approx 30\%$ of the chip is reserved for the infrastructure. For the XC7VX690T

board this results in the adjusted resource numbers shown in the second row of Table 3.1. The number of LUTs in Equation (3.1) has been further limited to 80% of the available units to allow routing flexibility. Table 3.3 contains utilization numbers computed assuming these constraints. The highest numbers are achieved for multiplication-heavy

$\frac{\text{Add}}{\text{PE}}$/IP	$\frac{\text{Mult}}{\text{PE}}$/IP	#PEs	Ops/cycle	DSPs	LUTs	GOps/s 200 MHz
1/Full	0	1183	1183	0.66	0.56	237
4/Full	1/Full	252	1260	0.70	0.53	252
3/Full	1/Full	315	1260	0.70	0.51	252
2/Full	1/Full	420	1260	0.70	0.49	252
1/Full	1/Full	630	1260	0.70	0.43	252
1/No	2/Full	450	1350	0.50	0.56	270
1/No	3/Full	384	1536	0.64	0.56	307
1/No	4/Full	315	1575	0.70	0.53	315
1/No	5/Full	252	1512	0.70	0.47	302
1/No	1/Full	1260	1260	0.70	0.27	252

Table 3.3.: Estimated performance number from maximizing resource usage under the SDAccel framework, restricting the board to 200 MHz, 70% of available DSPs and consuming no more than 80% of available LUTs (corresponding to 56% of total LUTs).

processing elements, with the highest number predicted for 1 addition per 4 multiplications. Both Table 3.2 and Table 3.3 show that the floating point IP core chosen for the processing elements yielding the highest overall utilization depend on the distribution of operations, as multiplications generally benefit more from using DSPs, while additions have the highest logic consumption.

3.4.3. Assuming reference design

The Xilinx reference design provides very minimal infrastructure, leaving most resources available for user kernels, and is clocked at 250 MHz, higher than the SDAccel framework. This means higher throughput per floating point unit, but makes timing closure more challenging for the routing tool, potentially constraining resource utilization. Again the number of LUTs used will be limited to 80% in Equation (3.1) to allow routing flexibility, but no restriction is imposed on the number of available DSP units. Table 3.4 contains computed performance numbers with these assumptions. The extra DSPs and higher clock rate give a significant improvement in potential over SDAccel, highlighting the performance restrictions imposed by using the SDAccel flow over a more manual solution such as the Xilinx reference design.

$\frac{\text{Add}}{\text{PE}}$/IP	$\frac{\text{Mult}}{\text{PE}}$/IP	#PEs	Ops/cycle	DSPs	LUTs	GOps/s 250 MHz
1/Full	0	1690	1690	0.94	0.80	423
4/Full	1/Full	360	1800	1.00	0.76	450
3/Full	1/Full	450	1800	1.00	0.73	450
2/Full	1/Full	600	1800	1.00	0.70	450
1/Full	1/Full	900	1800	1.00	0.62	450
1/No	2/Full	642	1926	0.71	0.80	482
1/No	3/Full	549	2196	0.92	0.80	549
1/No	4/Full	450	2250	1.00	0.75	563
1/No	5/Full	360	2160	1.00	0.68	540
0	1/Full	1800	1800	1.00	0.38	340

Table 3.4.: Estimated performance numbers for the Xilinx reference design clocked at 250 MHz with all DSPs available.

3.5. Peak benchmark

To test the peak numbers computed Section 3.4, a synthetic benchmark is constructed that follows the streaming pipeline architecture. We run the design on hardware using both the SDAccel framework and the Xilinx reference design, with some slight variations in the implementation details. The benchmark follows the assumptions of Section 3.2 by replicating identical processing elements connected sequentially in a deep pipeline of a variable data path width. The benchmark suite allows varying the element data type (however going beyond single precision is left for future work, see Section 3.10.1), the number of additions per stage, the number of multiplications per stage, the floating point IP cores used, and the width and depth of the pipeline. The computed numbers of processing elements from Section 3.4 will be used to configure the width and depth of the pipeline.

3.5.1. SDAccel implementation

When integrating with the SDAccel framework, the kernel is implemented in Vivado HLS rather than OpenCL, written to conform to the required interface specification (see Section 2.1.2). The outer function serving as the OpenCL kernel entry is included in Listing 14. Since there is no support for streaming interfaces, an internal BRAM buffer is used as a feedback loop to avoid latency from reading from memory, eliminating access to external memory for all but the initialization and termination phase, illustrated in Figure 3.1. The loops populating and clearing the buffer from/to global memory are not pipelined with the main compute function, so the total number of iterations run must be much larger than the number of bursts buffered to hide the initialization and termination latency, as we can only measure the runtime of the full kernel execution. The outer loop over iterations and the inner loop over elements has been flattened to a single loop to overlap the iterations in Listing 14. The width and depth are realized in the `ProcessBurst` function, included in Listing 15, by unrolling a function over the width of

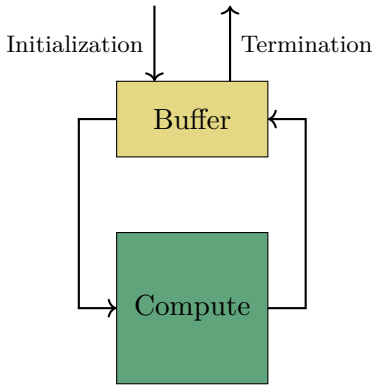


Figure 3.1.: Values are buffered in BRAM on the chip between iterations, requiring no external memory access during execution.

the data path and pipelining the depth of the data path, respectively. In order to

```
void EntryAxiMasterInternal(Burst const input[kBursts], Burst output[kBursts],
                            const unsigned iterations) {
  // Interface specifications here
  // ...
  Burst buffer[kBursts];
PopulateInternal: for (int i = 0; i < kBursts; ++i) {
    #pragma HLS PIPELINE II=1 enable_flush
    buffer[i] = input[i];
  }
  const unsigned totalIterations = iterations * kBursts;
MainInternal: for (int i = 0; i < totalIterations; ++i) {
    #pragma HLS LOOP_TRIPCOUNT min=kBursts
    #pragma HLS PIPELINE II=1 enable_flush
    Burst inputBurst, outputBurst;
    inputBurst = buffer[i % kBursts];
    ProcessBurst(&inputBurst, &outputBurst);
    buffer[i % kBursts] = outputBurst;
  }
ClearInternal: for (int i = 0; i < kBursts; ++i) {
    #pragma HLS PIPELINE II=1 enable_flush
    output[i] = buffer[i];
  }
}
```

Listing 14: Outer kernel function of the SDAccel peak performance benchmark. Data is read into an internal buffer before running the kernel over the data for a large number of iterations. The loop over elements and iterations have been flattened into a single pipelined loop.

loop over a large number of elements to make the time spent in the initialization and terminate phase vanish, the kernel is implemented to loop over elements of a buffer in a cyclic fashion. The buffer must be large enough to avoid data dependencies, and for the experiments performed below it was held at 2^{13} elements with 2^{14} repetitions, resulting in a total of 2^{27} elements being pushed through the pipeline for a single run of the

```
void ProcessBurst(Burst const *input, Burst *output) {
  #pragma HLS PIPELINE II=1 enable_flush
  static Element_t buffers[kDepth+1][kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=buffers complete dim=1
  // Read burst from stream
  Burst inputBurst = *input;
  inputBurst >> buffers[0];
Depth: for (int d = 0; d < kDepth; ++d) {
  Width: for (int i = 0; i < kElementsPerBurst; ++i) {
      Element_t load = buffers[d][i];
      Element_t eval = PerformStage(load);
      buffers[d + 1][i] = eval;
    }
  }
  Burst outputBurst(buffers[kDepth]);
  *output = outputBurst;
}
```

Listing 15: Function defining the width and depth of the pipeline through unrolling and pipelining of loops for the SDAccel peak benchmark. The code for the compute is included in Listing 16.

```
inline Element_t PerformStage(Element_t propagate) {
  #pragma HLS INLINE
  #pragma HLS PIPELINE II=1
  static const Element_t kAddVal =
      kFillVal * ((1 << kMultsPerStage) - 1) / kAddsPerStage;
AddsPerStage:
  for (int j = 0; j < kAddsPerStage; ++j) {
    Element_t eval = propagate + kAddVal;
    #pragma HLS RESOURCE variable=eval core=${PEAK_ADD_CORE}
    propagate = eval;
  }
  static const Element_t kMultVal = 0.5;
MultsPerStage:
  for (int j = 0; j < kMultsPerStage; ++j) {
    Element_t eval = propagate * kMultVal;
    #pragma HLS RESOURCE variable=eval core=${PEAK_MULT_CORE}
    propagate = eval;
  }
  return propagate;
}
```

Listing 16: Compute kernel of a single processing element for the peak performance benchmark in SDAccel. The stage performs a number of pipelined addition and multiplication operations unrolled from the two loops, outputting a number identical to the input (unless the number of additions or multiplications is zero). `Element_t` is the chosen data type. The variables prefixed with `k` are compile time constants. These cannot be declared as `constexpr`, as this is not supported by the HLS compiler, and are handled by a constant propagation optimization phase instead. The floating point core is inserted during configuration.

benchmark. As the number of elements pushed through the pipeline is large compared to the overhead, we can now approximate Equation (3.10) using Equation (3.11) to compute the expected runtime:

$$T_{\text{benchmark}} \approx \frac{2^{27}}{200\,\text{MHz}} = 0.6171\,\text{s} \tag{3.14}$$

As Equation (3.11) is technically a tight lower bound, the actual number of cycles will always be close to, but always slightly higher than, the number used here.

3.5.2. Reference design implementation

The implementation using the Xilinx reference design follows the same general architecture of a deep streaming pipeline as SDAccel with a number of key differences:

- Instead of executing a single large loop, the reference design executes the kernel function every clock cycle as a pipeline (see Section 2.4.7), using static variables to hold state. Every static variable maps to a persistent FPGA register or BRAM and is not optimized out by the compiler.
- The implementation uses the dataflow optimization (see Section 2.4.7) to instantiate each stage of the pipeline as a independent module, passing data between them using FIFOs.
- AXI Stream interfaces are used in place of AXI Master. This is a much better fit to the streaming pipeline architecture, as it gives the desired FIFO semantics in the HLS code.
- No memory controller: rather than reading and writing to global memory, the infrastructure populates an input buffer over a narrow 32-bit PCIe bridge before executing the kernel, the writes it back out over the PCIe bridge (see Section 2.1.3).
- Feedback of data is not necessary, as there is no memory overhead, and the performance can be deduced by the number of instantiated units by verifying against the number of cycles spent.

The HLS code of the infrastructure surrounding and instantiating the processing elements is included in Listing 17. Listing 17 uses the trick described in Section 2.4.8 to replicate the processing element the required number of times, by using a dummy template parameter to direct each function call to a unique instantiation. Listing 17 includes the entry to the kernel where the dataflow optimization is done, connecting the data movement and compute functions as individual modules with FIFOs. As described in Section 2.1.3, the Xilinx reference design is fitted with a cycle counter, providing the exact number of cycles taken to execute the program. Rather than timing the benchmarks, we verify that the reported cycle count is equal to the expected value, as only cycles spent during kernel execution are counted. By verifying that exactly one cycle is spent per element once the pipeline is saturated we justify computing the performance as the number of floating point units instantiated on the chip multiplied by the clock frequency. All benchmarks were

```
template <unsigned recurse>
void UnrollCompute(Stream pipes[kDepth]) {
  #pragma HLS INLINE
  UnrollCompute<recurse - 1>(pipes);
  Compute<recurse - 1>(pipes[recurse - 1], pipes[recurse]);
}
template <>
void UnrollCompute<0>(Stream *) {}

void EntryReferenceDesign(Stream &input, Stream &output) {
  #pragma HLS DATAFLOW
  static Stream pipes[kDepth + 1];
  static Stream feedback("feedback");
  ...
  Forward(input, feedback, pipes[0]);
  UnrollCompute<kDepth>(pipes);
  Forward(pipes[kDepth], feedback, output);
}
```

Listing 17: Core functionality of the kernel entry function in the Xilinx reference design, piping data between the I/O interfaces and the unrolled compute elements using dataflow optimization.

built with a 8 byte data path (2 single precision floating point numbers) and processed 65536 elements, so the number of cycles is computed using Equation (3.9) as:

$$c_{\text{peak}} = \frac{65536}{2} + L \tag{3.15}$$

where L is the latency of the pipeline, which can be extracted from the synthesis report from the HLS tool.

3.6. Performance results

Both the implementations of the peak performance benchmark will now be built and run, comparing the results to the predicted peak numbers.

3.6.1. SDAccel

The kernel from Listing 14 is configured using the IP cores and number of processing elements predicted in Table 3.3 and built using SDAccel, lowering the number of processing elements if the build fails due to failing to meet timing closure, repeating this process until the build succeeds. The highest succeeding build is included as the *Built* column, and is presented as the expected performance of the successfully built number of processing elements, and is shown next to the actual measured performance when run on hardware. The numbers are plotted in Figure 3.2. All configurations are seen to build at or very close to their peak resource utilization, and run in a time very close to what was computed in Equation (3.14), with only a slightly higher runtime due to reading/writing memory and saturating the pipeline. Configurations 3/1 and 2/1 show unexpected behavior, as they both take *exactly twice* the expected execution time, despite not differing in

Adds	Mults	Peak $\left[\frac{\text{GOp}}{\text{s}}\right]$	Built $\left[\frac{\text{GOp}}{\text{s}}\right]$	Time [s]	Performance $\left[\frac{\text{GOp}}{\text{s}}\right]$	Frac.
1	0	236	230	0.6714	230	0.98
4	1	252	216	0.6714	216	0.86
3	1	252	230	1.3425	115	0.91
2	1	252	240	1.3425	120	0.95
1	1	252	243	0.6714	243	0.97
1	2	270	270	0.6714	270	1.00
1	3	307	294	0.6714	294	0.96
1	4	315	288	0.6714	288	0.91
1	5	302	302	0.6714	302	1.00
0	1	252	252	0.6714	252	1.00

Table 3.5.: Peak performance, predicted performance for highest built configuration, and measured performance of highest built configuration. All benchmarks process the same amount of elements. Statistical uncertainties have been left out as they are neglicible. Configurations 3/1 and 2/1 exhibit unexpected behavior, as they run at exactly double the expected time.

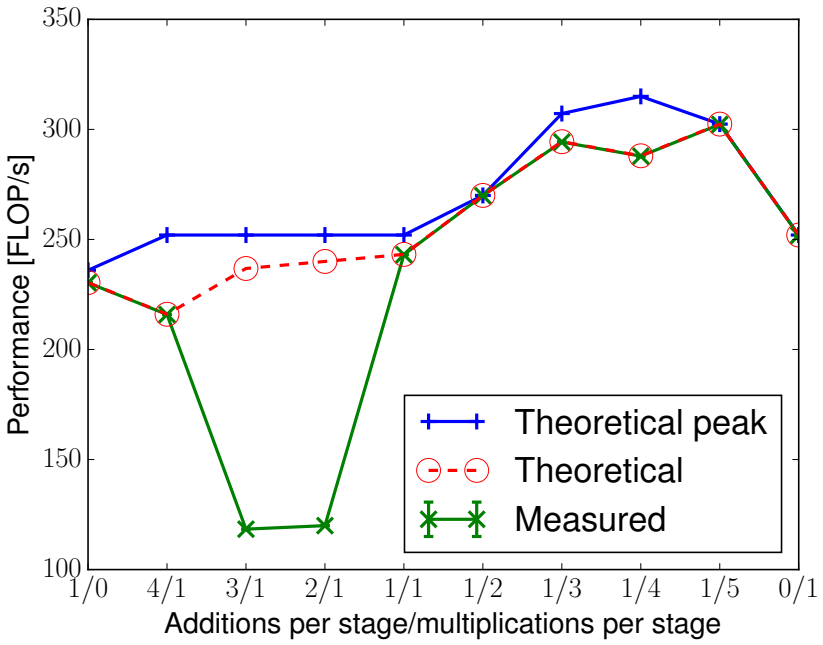


Figure 3.2.: Plotted data from Table 3.5, showing peak performance from Table 3.3 versus highest built configuration, with both predicted and measured performance for the latter. Configurations 3/1 and 2/1 run at exactly double the expected time. Statistical uncertainties are left out as they are too low to be visible in the plot.

architecture from the other configurations, both producing correct results, and showing no difference in their synthesis reports, suggesting a bug in the SDAccel toolflow or runtime. The highest performance is achieved for configuration 1/4 additions to multiplications, reaching 302 $\frac{\text{GOp}}{\text{s}}$ with full resource utilization. This is to the author's knowledge the highest performance number measured with SDAccel on the AlphaData 7V3 card.

3.6.2. Xilinx reference design

The process above is repeated for the Xilinx reference design, starting from full resource utilization and lowering the number of processing elements if necessary until the design builds successfully. These numbers are included in Table 3.6 and are plotted in Figure 3.3. The two rightmost columns include the number of cycles predicted from Equation (3.15) using the latency reported from Vivado HLS, along with the "true" number of cycles reported by the cycle counter in the reference design. Many results run close to or at

Adds	**Mults**	**Max #PE**	$F\left[\frac{\text{Op}}{\text{s}}\right]$	**Built #PE**	$F\left[\frac{\text{Op}}{\text{s}}\right]$	Frac.	**Cycles est.**	**Cycles**
1	0	1690	423	1370	343	0.81	40306	40316
4	1	360	450	248	310	0.69	38227	38237
3	1	450	450	322	322	0.72	38406	38416
2	1	600	450	504	378	0.84	39323	39333
1	1	900	450	740	370	0.82	39061	39071
1	2	642	482	642	482	1.00	39191	39201
1	3	549	549	548	548	1.00	39895	39905
1	4	450	563	434	543	0.96	39715	39725
1	5	360	540	328	492	0.91	39003	39013
0	1	1800	450	1800	450	1.00	39971	39981

Table 3.6.: Performance at full utilization, along with performance for the highest successfully built configuration, for the benchmark application built with the Xilinx reference design clocked at 250 MHz. The last two columns show the predicted and actual number of cycles spent in the kernel.

the predicted peak performance. The highest performance is achieved for configuration 1/3, reaching 548 GOp/s. All benchmarks run at the number of cycles predicted, plus an extra 10 caused by an unknown additional overhead constant across all measurements. This means that only a single cycle is spent per work item in addition to the saturation overhead, implying that the sustained performance is exactly as predicted. Configurations heavy on multiplications are generally seen to come closer to the predicted performance, perhaps due to having a more balanced ratio of DSPs to LUTs relative to the available resources on the chip, as this is also reflected on a local level, which can affect timing closure. The configuration required for the stencil implementation (see Section 4.3) only reaches 322 GOp/s, which is the second lowest number among all the configurations tested.

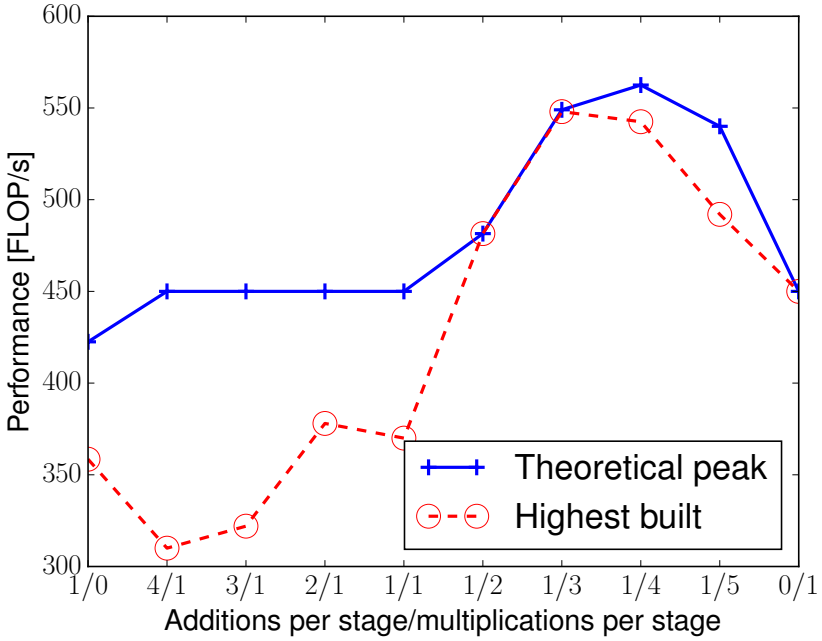


Figure 3.3.: Plotted data from Table 3.6, showing performance at full resource utilization and the performance of the highest successfully built configuration for the Xilinx reference design clocked at 250 MHz.

3.7. Memory benchmark

To measure the other component of the roofline model, a memory benchmark was set up in SDAccel, testing raw memory throughput when reading and writing to/from an AXI master interface. Memory bandwidth for the Xilinx reference design could not be measured, as it does not provide a memory controller. The benchmark is split in three flavors: a read only benchmark, a write only benchmark, and a read and write benchmark. The AlphaData 7V3 board comes with two DDR3 DIMMs declared at 1333 MT/s, implying a memory clock of 167 MHz for a base clock multiplier of 4 (DDR3) with a transfer rate of 2 (DDR). Using Equation (1.5) this corresponds to a per-DIMM memory bandwidth of:

$$\frac{1}{2}R_{7\mathrm{V3}} = 167\,\mathrm{Mcycle/s} \cdot 2\,\mathrm{bit/cycle} \cdot 4 \cdot 64 = 85.5\,\mathrm{Gbit/s} = 10.7\,\mathrm{GByte/s} \tag{3.16}$$

For both DDR modules the peak bandwidth is then $R_{7\mathrm{V3}} = 21.4\,\mathrm{GByte/s}$, but because the version of SDAccel used for this work only supports a single DIMM, the practical ceiling will be Equation (3.16).

The code for the read-and-write benchmark is included in Listing 18, and simply consists of a pipelined loop reading bursts of 512 bits from an input interface and writing it back out to an output interface. Although the input and output are separate ports to the kernel, they map to the same memory DIMM, because of the restriction

in SDAccel described above. Since SDAccel kernels are clocked at 200 MHz and the

```
void EntryBoth(Burst const *input, Burst *output) {
  #pragma HLS INTERFACE m_axi port=input offset=slave bundle=gMemIn
  #pragma HLS INTERFACE m_axi port=output offset=slave bundle=gMemOut
  #pragma HLS INTERFACE s_axilite port=input
  #pragma HLS INTERFACE s_axilite port=output
  #pragma HLS INTERFACE s_axilite port=return
Main:
  for (unsigned long i = 0; i < kBursts; ++i) {
    #pragma HLS PIPELINE II=1
    output[i] = input[i];
  }
}
```

Listing 18: Memory benchmark for read/write written in HLS consisting of a pipelined loop reading and writing 512-bit values from and to two AXI master ports, both wired to the same memory controller.

kernel reads and writes 512-bit values, the kernel attempts to read at a bandwidth of 200 cycle/s · 64 Byte/cycle = 12.8 GByte/s, which is sufficient to keep the memory controller busy. The variable `kBursts` in Listing 18 is known at compile time, so the HLS tool can statically schedule the full series of bursts to the memory controller, and the number of bursts is reported by the tool during synthesis. For the benchmarks carried out here, 2 GByte is read/written, issuing ≈ 33 million bursts of 512 bits. Results for each of the three benchmarks are included in Table 3.7. All three numbers are significantly lower

Benchmark	**Measured** [GByte/s]	**Fraction of peak** (10.7 GByte/s)
Read	3.54	0.33
Write	4.09	0.38
Both	5.44	0.51

Table 3.7.: Memory benchmarks for a single DDR3 DIMM on the AlphaData 7V3 cards along with their fraction of the theoretical peak bandwidth. Errors are in the order of 10^{-4} GByte/s and have been left out.

than the theoretical bandwidth, and seem to suggest poor scheduling to the memory controller, as the access pattern by the benchmark is completely sequential. This could possibly be improved by moving to the AXI Stream interface, but this is not yet supported by the SDAccel tool (as of release 2015.4).

3.8. Resulting FPGA roofline

Using the peak numbers for performance and memory bandwidth computed in this section, we now generate a roofline plot of the AlphaData 7V3 card, shown in Figure 3.4, and compare it to the fixed architectures presented in Section 1.7, shown in Figure 3.5. The measured numbers for FPGA are included for reference. For both the predicted and

measured performance the highest number obtained is used, which are the 1 addition/3 multiplication and 1 addition/4 multiplication configurations, respectively.

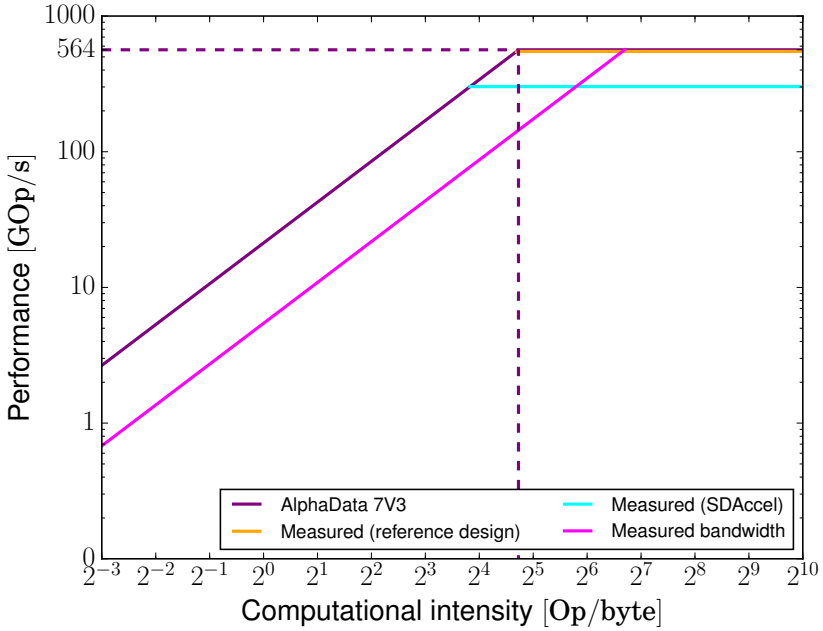


Figure 3.4.: Roofline for the AlphaData 7V3 card, including the highest measured memory bandwidth and peak performance under both frameworks.

3.9. Discussion

Using a streaming pipeline design, the benchmarks implemented here achieve numbers that lie close to the predicted peak performance, and in some cases even reach the theoretical maximum for the given configuration. The highest measured performance is 302 GOp/s for SDAccel and 548 GOp/s for the Xilinx reference design, for 1/5 and 1/3 configurations of additions to multiplications respectively, both using the "No" and "Full" DSP IP cores, spending all DSPs on multiplications, while implementing additions using logic only. It is clear that the restrictions imposed by SDAccel are a significant hindrance to performance, which raises doubts to its viability as a high performance computing framework. It should be mentioned, however, that including a memory controller in the Xilinx reference design would consume in the order of 20000 LUTs (according to Xilinx engineers), but leaving DSP resources untouched. This would amount to a decrease in performance of between none (if only limited by DSPs) to $20000/433200 \approx 5\%$ (if only limited by LUTs).

The memory bandwidth on the AlphaData 7V3 card was measured to be only 0.51 of the datasheet value with a completely regular access pattern. The low memory bandwidth already put the required computational intensity to achieve peak performance far to the

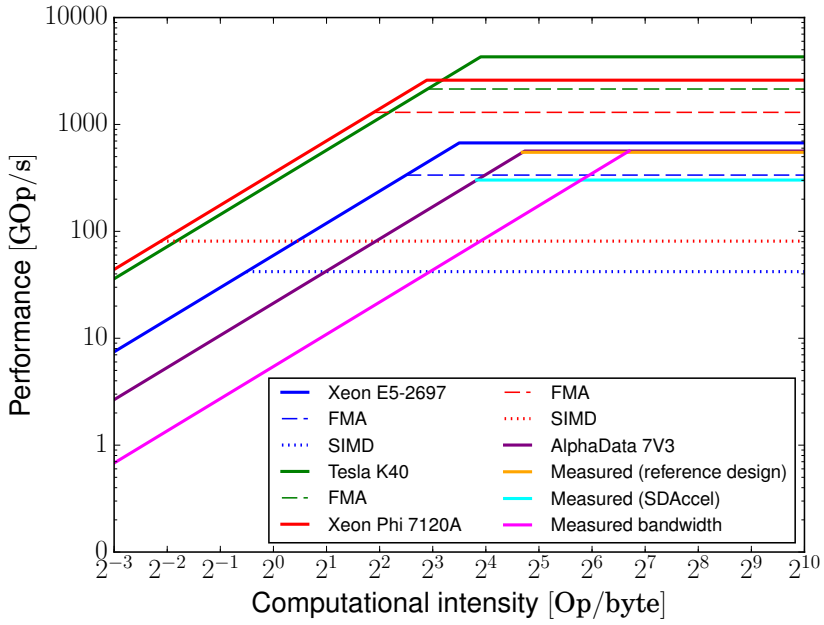


Figure 3.5.: Combined rooflines for examples from all hardware architectures discussed in this work.

right in Figure 3.4, and the measured result pushes the around 27 operations per byte for the datasheet memory bandwidth to more than 100 for the bandwidth measured off the single DIMM in SDAccel. Although the measured bandwidth would be doubled once the tool supports two DIMMs, it is clear that FPGA applications will need to be implemented in a way that allows scaling the design with constant memory bandwidth, such as the streaming pipeline architecture, in order to perform on contemporary hardware. The next chapter will look at doing this for the application of stencils.

When compared to the same generation of fixed architecture hardware, the FPGA does not seem competitive for single precision performance when looking at Figure 3.5. One aspect where the FPGA could hold its own is power efficiency, but this is left as future work (see Section 3.10.4). Perhaps the most encouraging result of the benchmarks performed here is the accuracy of benchmark results to the predicted numbers: measured performance can converge to peak performance, and we can show extremely precise measurements by verifying against the exact number of cycles spent in the benchmark. The question remains whether this can be transferred to a real application, which will be treated in the following chapter.

3.10. Future work

This section will go over paths to pursue on the subject of peak performance on FPGA, some of which have already been hinted at above.

3.10.1. Floating point precision

Throughout this chapter we have only looked at 32 bit floating point numbers. The Xilinx toolflow also offers IP core support for 16 and 64 bit floating numbers. Repeating the predictions and measurements for these precisions would give a more complete picture of the state of floating point precisions on FPGA.

3.10.2. Operation diversity

Because we focused primarily on achieving peak floating point performance, additions and multiplications were chosen as the constituents of the replicated processing elements. The method used to predict and measure performance can be applied directly to a more diverse set of operations, making it a useful tool for computing the peak performance of any input application with a given distribution of operations. Incorporating division and advanced math functions in the benchmarks would give a broader idea of what performance to expect from a wide range of applications.

3.10.3. Fixed point

As mentioned in Section 1.4.1, the DSP units on the Virtex 7 architecture are optimized for signal processing applications, and in particular fixed point types, that achieve computation of reals using just integer logic. Moving to fixed point requires a lot more attention from the programmer, as all numbers entering computation must be in a similar order of magnitude to prevent fatal overflow. If this suits the target domain, however, and the fixed point precision can be limited to 18 bit, every DSP unit can be pipelined with two additions and one multiplication per cycle, offering $3 \cdot 3600\,\mathrm{Op/cycle} = 10\,800\,\mathrm{Op/cycle}$ from DSPs alone on the XC7VX690T, corresponding to 2700 GOp/s at 250 MHz. It would thus be interesting carrying out the procedure done for single precision floating point numbers in thie chapter for DSP-native fixed point types on the Virtex 7 architecture.

3.10.4. Power measurements

FPGAs are often praised for their power efficiency over fixed architetures, running at an order of magnitude less power than GPU and many-core accelerators. On the road to exascale computing, power efficiency has become a crucial factor when choosing hardware, which so far seems to be dominated by GPUs. It would therefore be relevant to include performance per power comparisons for peak performance configurations on FPGA and compare them to major fixed architectures. In the following chapter we will make a rough approximation from power numbers measured in other works, but a proper study would require accurate power measurements during operation of the given implementation.

3.10.5. Memory

Breaking free from the constraints imposed by SDAccel proved to be a significant improvement on peak performance, and the same exercise should be done for memory

bandwidth. The Xilinx reference design did not offer a memory controller when these experiments were run, but once such a design is available offering an AXI Stream interface, the memory performance might do a better job at approaching the theoretical DDR numbers than the address mapped memory supported by SDAccel.

3.10.6. Other chips

The measurements here assumed Virtex 7 and were done for a single chip, but the methods applied should generalize to any FPGA. Altera claim 9.2 TOp/s for their largest device in the unreleased 14 nm Stratix 10 series. If this is indeed achievable it would be an order of magnitude higher than was measured here, and would challenge top-end contemporary GPU performance (the Tesla K80 is reported as having a peak of 8.73 TOp/s). Xilinx themselves have released the 16 nm Virtex Ultrascale series as the followup to the 7-series, offering a 50% increase in logic and DSP slices, and the benchmarks performed here suggest that this translate directly into an equivalent factor of increase in performance, but this would have to be seen.

4. Stencils

Stencil codes are commonly found in HPC, as they are a simple, but powerful method capable of numerically solving partial differential equations [20] for a wide range of applications [21, 22]. Their regular structure mean that much can be done in terms of optimization to achieve the highest possible hardware utilization on various architectures.

Stencil algorithms work on a regular k-dimensional grid, iteratively applying updates across the grid cells over a number of sequential timesteps. Updates to a grid point are performed using a stencil *kernel*. The kernel specifies a neighborhood around the cell coordinate from which the updated value will be computed. This can be represented as a *dependency set* S of relative tuples of k-dimensional indices $s = (x_0, ..., x_k)$:

$$S = \{s_0, ..., s_l\} \tag{4.1}$$

Each timestep is computed exclusively using values from previous timesteps, meaning there are no intra-timestep dependencies. This locality of dependencies means that grid updates can easily be partitioned in space, computing each partition independently, only requiring overlaps at the border between partitions to be resolved between timesteps, making stencil codes well-suited for heterogeneous and distributed systems. Because the computation performed to evaluate a cell update based on its neighbors are simple as linear combinations, the ratio between computation and data movement (computational intensity) is often low. This makes stencil algorithms very sensitive to cache access patterns, and make them good candidates for architectures that can provide high memory bandwidth, such as GPGPU. This chapter will look into an approach to attain high performance of stencil code on FPGA, where memory bandwidth is not available in abundance.

4.1. Related work

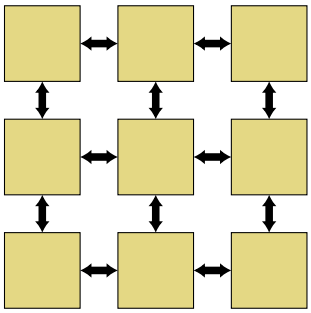
Optimizing stencil codes on fixed architecture is an active and ongoing field of research. Revolving around the *polyhedral model*, many frameworks have been created to generate performant architecture specific code from formal representations of the iteration spaces and dependencies of nested loops [23, 24, 25]. Stencil codes on FPGA have received less attention. Kobori et al. [26] demonstrate a cellular automaton implementation for integer data, dividing the approach into three cases depending on the grid size, where the largest uses invalidation of results and repeated passes to achieve a regular structure of computation. Anshuman et al. [27] evaluate two OpenDwarf benchmarks implemented in OpenCL on various architectures, but dismiss stencils on FPGAs as not being competitive with GPU. The algorithm implemented in this work follows the design presented by

Sano et al. [28], who demonstrate 260 GOp/s for a 2D Jacobian on *nine* older Stratix III FPGAs connected in sequence as systolic array nodes.

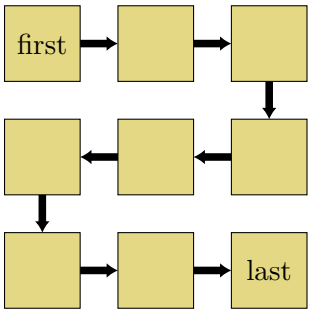
4.2. Proposed architecture

To bring a program to the constant memory bandwidth form described in Section 3.2.1, we restrict the global memory access pattern of the program to a single input stream and a single output stream, reading and writing KB_t bytes every cycle. For a stencil program this means that data will flow from the first coordinate to the last coordinate in a lowest-to-highest index fashion, wrapping around when the end is reached. A performant program on this form must be able to scale the depth parameter n in order to saturate the floating point resources on the FPGA (increasing the computational intensity, as the bandwidth is constant, according to Equation (3.8)). This requires every element necessary to evaluate the stencil, that is not currently on the wavefront, to be either buffered internally from when it is read and until the last coordinate depending on it is evaluated, or propagated throughout the pipeline according to the access pattern.

4.2.1. Systolic array



(a) Cellular automaton architecture, where cells are represented as a physical grid of identical processing elements on the chip, communicating with neighboring elements during execution.



(b) Systolic array architecture, where identical processing elements are connected in sequence, feeding results downstream as they become available.

Figure 4.1.: Cellular automaton versus systolic array architecture. All processing elements represented by squares are identical replicated circuits.

We wish to achieve indefinitely replicable processing elements without scaling the required memory bandwidth along with the number of processing elements, in order to approach peak performance with constant memory bandwidth (see Section 3.2.1).

One option is cellular automata, intuitively expressing cells or clusters of cells as identical processing elements, distributing processing elements like a physical grid on the chip. Neighboring cells will then communicate with each other through local connections (Figure 4.1a). Once the initial data has been written to the automata's local memories,

there is no dependence on global memory, as data will be buffered locally throughout the entire computation. This design has the severe issue of being bound in either direction by the problem size: too small grids will not make full use of the computational resources on the fabric, and too large grids will not fit the on-board memory. It is therefore not a fitting design to achieve peak performance on an arbitrary chip.

Instead we will focus on systolic arrays. Like cellular automata the architecture consists of a number of identical processing elements, but orders them from upstream to downstream (Figure 4.1b). Each element receives data from one or more upstream elements, and sends data to one or more downstream elements. Data is fed to the first element and propagates downstream until it has passed through all processing elements, and is then written back out. While systolic arrays are often used in networks where processing elements represent hardware nodes, elements will for this architecture be represented by replicated components across the FPGA fabric, connected by on-chip interfaces (either ping-pong buffers or FIFOs) in a streaming pipeline architecture (see Section 2.5). This can potentially be extrapolated to multiple FPGAs, connecting the last element in one FPGA with the first element of the next via e.g. ethernet, but this work will focus on a single device.

4.2.2. Temporal pipelining

Processing elements in systolic arrays receive data from upstream, solve part of the problem, then pass it downstream to the next processing element. As we only have one streaming entry to the design, a constant memory bandwidth will not be able to saturate a spatial split of the iteration space. Instead we approach stencils by pipelining the outer time dimension, letting processing element treat the full spatial dimension for subsequent timesteps. The width of the pipeline will extend in the spatial dimension, controlling the number of elements travelling the data path for a single timestep according the memory bandwidth, while the depth of the pipeline will correspond to the number of timesteps treated in parallel. The number of timesteps to be processed is assumed to be much larger than the maximum depth of the pipeline, so the depth of the pipeline will decide a *folding* factor that will decide the final number of steps n in Equation (3.11) (not to be confused with the pipeline depth, which we also denote as n). For N processing elements processing T timesteps, the total number of steps is reduced to $n = T/N$, where processing element $0 \leq i < N$ processes timesteps $Nt + i$ where t is the folded iteration variable $0 \leq t < n$.

4.2.3. Folding and feedback

To achieve time folding the pipeline must process the same spatial elements multiple times, requiring a feedback design where elements written out from the last element in the pipeline are written back to memory, before being read in again to compute the next folded timestep. In order to maintain a fully saturated pipeline, subsequent folds must overlap, requiring elements to fall out of the end of the pipeline before they are required at the entry. For smaller grids, the feedback can happen exclusively in on-chip memory,

and the latency of the pipeline will offset the amount of memory required. For large grids off-chip memory can be used as the feedback storage between folds.

4.2.4. Logic and BRAM requirements

Because the spatial dimension will only pass through the systolic array once per fold, each processing element is responsible for buffering data necessary to evaluate future iterations of that same timestep. The amount of on-chip storage space required per processing element depends on the grid dimensions, the shape of the stencil and to a lesser extent the latency of evaluating the stencil. The maximum number of processing elements that can be instantiated on the chip due to BRAM limitations only is:

$$N_{\max,\text{BRAM}} = \frac{B_{\max}}{\text{PE}.B} \tag{4.2}$$

The optimization expression in Equation (3.1) determines the maximum number of processing elements that can be instantiated taking both compute and memory resources into account, but since PE.B varies with the size of the spatial domain, it is useful to treat logic and memory constraints separately:

$$N_{\max} = \min\{N_{\max,\text{FP}}, N_{\max,\text{BRAM}}\} \tag{4.3}$$

where $N_{\max,\text{PE}}$ is the maximum number of processing elements that can be instantiated due to logic and DSP limitations only, as computed in Section 3.4. We can then compute the tipping point $N_{\max,\text{FP}} = N_{\max,\text{BRAM}}$ where the performance is limited by the BRAM resources on the board in terms of the grid parameters, effectively giving us the maximum grid dimensions where peak performance can theoretically be reached. This will be done for the specific case treated below.

4.3. 2D Jacobian stencil

This work will focus on the case of a 2D Jacobian stencil. This stencil averages the immediate neighborhood of the evaluated cell:

$$h(x, y, t+1) = \frac{1}{4}(h(x-1, y, t) + h(x, y-1, t) + h(x+1, y) + h(x, y+1)) \tag{4.4}$$

where $h(x, y, t)$ is the value of grid coordinate (x, y) at time t. The dependency set is:

$$S = \{(-1, 0), (0, -1), (1, 0), (0, 1)\} \tag{4.5}$$

and the required on-chip memory per processing element is:

$$M_{\text{Jacobi2D}} = 2d_x \tag{4.6}$$

Figure 4.2 illustrates the systolic array architecture for this stencil, feeding data from the input stream to PE_0, passing the computed result for time t downstream, computing

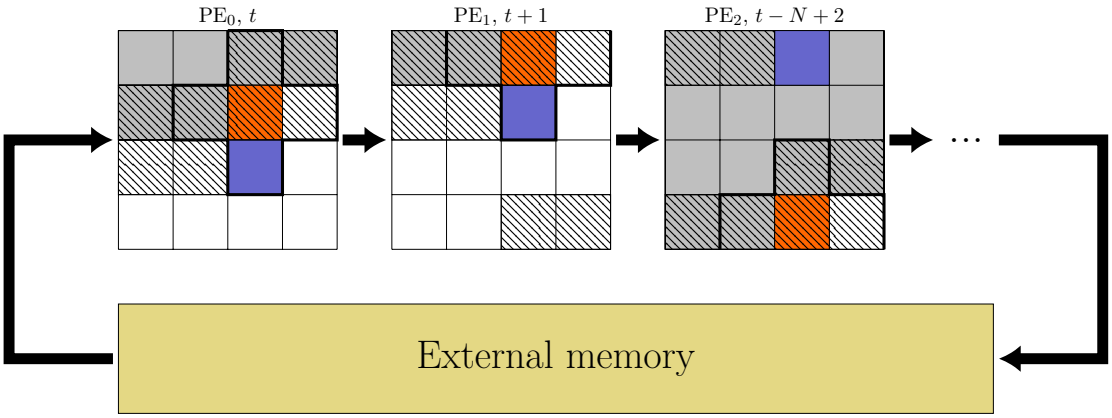


Figure 4.2.: Data flowing between processing elements in the systolic array architecture. A processing element at position i processes every Nth timestep with offset i, passing along evaluated values are they become available throughout the pipeline. This results in a total of N timesteps being evaluated in parallel. Orange indicates the element currently being evaluated, blue indicates the current wavefront of the dataflow, gray indicates elements already evaluated for the current timestep, and shaded squares indicate elements currently held in buffers.

$t+1$ in PE$_1$, repeating until the result falls out at the end of the pipeline and is written back to the output stream. The input and output streams will then wrap around for n iterations until all folds have been computed, by using either on-chip or off-chip memory as feedback buffer. In the saturation phase the wavefront populates the buffers until it reaches the first element where all necessary values are available, which in the case of the 2D Jacobian stencil is on the second row, as there is only a dependency one row forward. For following spatial iterations, processing the last row of t overlaps with reading in values into the buffers for the first row of $t+1$, as all elements for the last row are already available by then, as seen in the second and third processing element in Figure 4.1b.

4.3.1. Determining resource bottleneck

A processing element is responsible for evaluating the stencil for every element in the data path every cycle, consisting of 3 additions and 1 multiplication per element ($3+1=4$ operations for a data path of 1 element, $12+4=16$ for a data path of 4 elements etc.). Assuming that the design is limited by LUT and DSP resources, the peak performance of this stencil will be taken as the numbers presented in Table 3.3 and Table 3.4. In practice the number of processing elements will be constrained by either the amount of computational resources or the amount of on-chip memory. An 18 kbit BRAM block can read and write a 32-bit floating point number every cycle. If we can exploit reuse of the horizontal elements the minimum number of BRAM blocks required per processing element is 2, one for each line buffered Equation (4.6). For 32-bit entries, each BRAM

block can hold a maximum of 18 kbit/32 bit = 562 elements, which combined with Equation (4.6) gives us the amount of BRAM blocks required per processing element. The resources required per processing elements are then:

$$\begin{aligned} \text{PE}.D &= 3D_i + D_j \\ \text{PE}.L &= 3L_i + L_j \\ \text{PE}.B &= 2\frac{d_x}{562} \end{aligned} \tag{4.7}$$

where i and j and indices to the chosen floating point IP core and the BRAM number is computed using integer division. Since the chosen floating point IP core is not affected by the addition of BRAM, the maximum number of floating point units that can be instantiated from Equation (3.12) is unaffected, which we'll denote $N_{\text{max,FP}}$. The maximum number of processing elements in terms of BRAM only is:

$$N_{\text{max,BRAM}} = \frac{B_{\text{max}}}{2(d_x/562)} \tag{4.8}$$

We can now compute the tipping point $N_{\text{max,FP}} = N_{\text{max,BRAM}}$ of Equation (4.3) where the performance becomes limited by the BRAM resources on the board, in terms of the grid x-dimension d_x:

$$\frac{B_{\text{max}}}{2(d_x/562)} = N_{\text{max,FP}} \Rightarrow d_x = \frac{562}{2}\frac{B_{\text{max}}}{N_{\text{max,FP}}}$$

Plugging in the peak floating point performance for the Xilinx reference design from Table 3.6 and available BRAM resources on the XC7VX690T board we obtain:

$$d_x = \frac{562}{2} \cdot \frac{2940}{450} = 1835 \tag{4.9}$$

Meaning we can buffer grids of up to 1835 elements in the smallest dimension on the board and still theoretically obtain peak performance. This argument extends to k dimensions, where d_x is simply replaced with the product of the $k-1$ smallest dimensions in Equation (4.9).

4.4. Stencil implementation

This section will describe a systolic array stencil implementation on FPGA, representing the kernel module in the Xilinx reference design in Figure 2.2. The main kernel design is synthesized with Vivado HLS from C++ code augmented with HLS pragmas, and is integrated with the infrastructure of the reference design using the main Vivado tool, in a fully automated flow that generates the appropriate hardware from a given configuration.

4.4.1. Choice of framework

The stencil design was originally intended to be implemented in SDAccel to avoid dealing with performance-unrelated components on the FPGA. While the peak performance benchmark ran as expected, SDAccel (release 2015.4) would hang up when introducing a feedback loop in internal BRAM for a dataflow program, possibly related to issues with memory latency. Instead Xilinx provided the reference design described in Section 2.1.3 to test stencil codes, which also offers a higher maximum resource count for the kernel. This framework does not provide a memory controller, so the designs tested here rely on emulating external memory using on-chip memory. While this limits the grid dimensionality that can be treated, all concepts related to the performance are the same, assuming a design including a memory controller would expose a streaming interface to memory that could feed the pipeline every cycle, which is not unrealistic given the perfectly regular access pattern and low bandwidth requirements.

4.4.2. Dataflow and modularity

The processing element is designed as a single C++ function, using the same dataflow infrastructure as the peak benchmark kernel described in Section 3.5.2, unrolling the compute function as dataflow functions with the recursive template included in Listing 17. The unrolled functions are compiled separately and connected by FIFOs on the fabric, which lets the toolflow treat them as separate smaller pipelines rather than having to do static scheduling for the full depth. The C++ functions are implemented expecting to be executed *every clock cycle*, rather than being based on loops. Each stage will execute whenever data is received from the input stream, producing output triggering the following steps. The amount of compute elements to unroll is adjusted by the configuration-time parameter `kDepth`, deciding when the template recursion bottoms out.

4.4.3. Buffering dependencies

A given element must be buffered from the time it is first read in via the wavefront until the last time it is used by a computation. This is illustrated by the gray area in Figure 4.3, where the arrows indicate the flow of the data from when it arrives at the wavefront to the cells where the values are needed. After used as south values, data is first propagated to the center row buffer. When it is read out from the center buffer it is passed along iterations in registers to be used as eastern and western value, as well as being written to the north buffer, where it will read out for the following row. Dependencies have very different characteristics in terms of buffering in hardware depending on their relative position to the cell being evaluated:

- Dependencies backwards in the inner dimension can be handled by shifting values through registers.
- Dependencies forward in the inner dimension will shift the iteration space by the distance to the center cell, and are otherwise handled using shift registers.

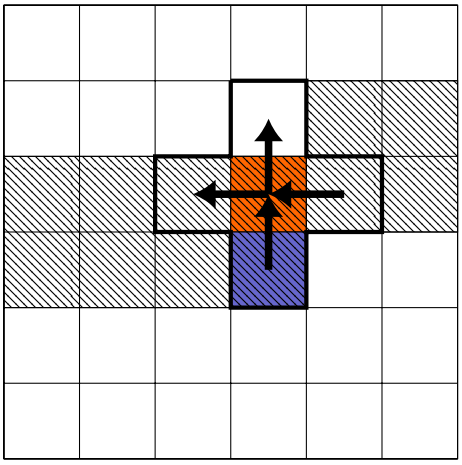


Figure 4.3.: Elements currently held in buffer are shaded in gray, as they are still necessary to evaluate one or more future iterations of the stencil. The cell currently being evaluated is marked in orange, and the current position of the incoming wavefront is marked in blue. Arrows indicate flow of data through buffers.

- Dependencies backward in an outer dimension increase the amount of rows (or planes/hyperplanes in higher dimensions) that must be buffered.
- Dependencies forward in an outer dimension increase the number of rows that must be buffered and shifts the iteration space by the vertical distance to the center cell times the number of elements per row.

Shifting the iteration space means displacing the iteration over cells being evaluated from the wavefront of incoming data. The hardware circuit designed to buffer these dependencies will be presented in the following section.

4.4.4. The processing element

Figure 4.4 includes a diagram of the proposed circuit for a single processing element, simplified to not include control logic for saturating and draining the pipeline. Handling the saturation and draining phase adds a significant amount of logic to the circuit, as many cases must be handled, potentially causing congestion in the processing element due to multiplexers that must route the relatively wide data path. The way in which this logic is implemented is seen to cause large variations in the amount of LUTs consumed despite having the same semantics, highlighting the opaqueness of the HLS to hardware flow.

4.4.5. Deviation from proposed design

The processing element used in the design that was built for hardware and used for the benchmarks below differs somewhat from the circuit presented in Figure 4.4, and uses

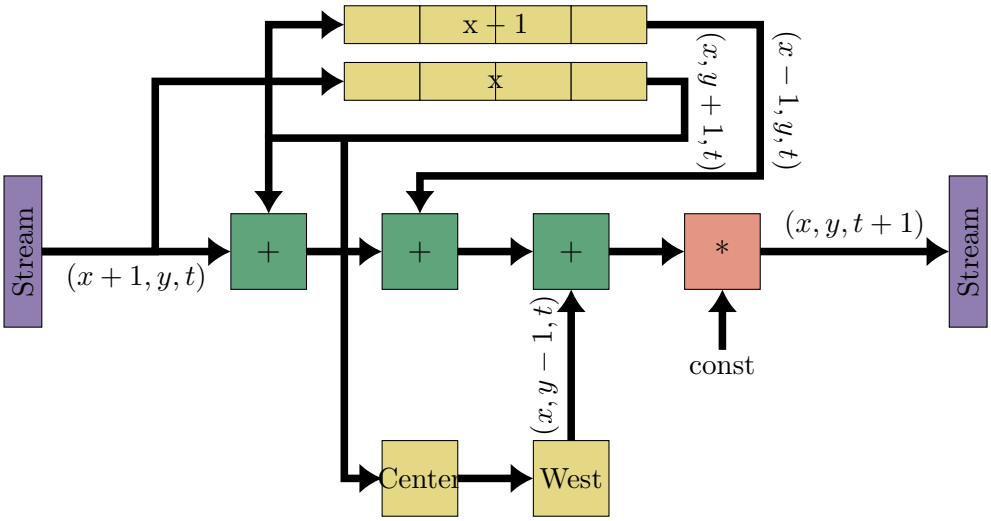


Figure 4.4.: Simplified diagram of the circuit of a single processing element. Connections are annotated with the index in (x, y, t)-space they provide an element from. The wavefront coming from the input stream is one row ahead of the row being evaluated, storing the row being evaluated and the previous row in the two buffers.

four buffers rather than two, storing western and eastern values separately to the two row buffers. This consumes twice the amount of BRAM (halving the maximum grid size in Equation (4.9)), but was used because an implementation of the more memory efficient design was not finished in time for writing this thesis.

Appendix B contains both the implementation used for the benchmarks run here (functions suffixed with `Four`, due to using four buffers rather than the minimum two required), and a proposal for a new design, splitting the buffering and computational elements to increase readability and modularity. This design uses the FIFOs between the buffering module and the computational module as the buffer, shipping data to the compute elements when they're ready, thereby decoupling the computational iteration space from the wavefront. The newer design produces correct results, but was found to fail timing at lower resource utilization than the old, so the older one was kept for the sake of maximizing performance.

Reducing the required BRAM blocks to the number in Equation (4.8) is left for future work.

4.4.6. Control flow

As suggested in Section 4.4.4, some control flow is necessary to handle the saturation and draining phase of the pipeline. Listing 19 includes the general structure of these conditions in the code, showing the flow between the buffer and compute elements. The code follows the four buffer (rather than two) design for the performance reasons described above. In addition to the natural saturation of the pipeline by incoming data, the algorithm

```
if (!isDraining) {
  streamIn.read() >> inputValues;
}
if (onFirstRow) { // North row
  FillBoundary(northValues);
} else if (isSaturated) {
  northBuffer.read() >> northValues;
}
if (isSaturated) { // Center row
  centerBuffer.read() >> ...;
  westBuffer.read() >> westValues;
  eastBuffer.read() >> eastValues;
}
if (onLastRow) { // South row
  FillBoundary(southValues);
} else {
  inputBurst >> southValues;
}
// ...do compute and forward values...
if (!isDraining) {
  if (!onLastRow) {
    northBuffer.write(...);
  } // Overlap to next timestep
  centerBuffer.write(...);
  westBuffer.write(...);
}
if (i > 0 && i < kTotalBursts * kTimestepsPerStage + 1) { // Shift by one
  eastBuffer.write(...);
}
if (isSaturated) {
  streamOut.write(...);
}
```

Listing 19: Skeleton of the control flow handling the saturation and draining phase, as well as overlapping between consecutive timesteps.

requires buffering the first incoming row of data before starting the computations, as three rows are required to compute the 2D Jacobian stencil, but the boundaries are computed as constant values. This additional saturation phase is only done at the first folded timestep, as subsequent folds will overlap to hide this delay. The added latency is equivalent to the number of cycles required to read in a single row, multiplied by the folding factor N.

4.4.7. Width of data path

In order to allow peak performance at constant memory bandwidth the width of the pipeline data path only needs to be smaller than or equal to the bandwidth. This allows some flexibility in the granularity of pipelines stages in order to optimize resource consumption and meet timing. We will build the stencil code using various widths and depths where $nK = \text{const}$ to see the effect of this. Section 4.6 will look at the scaling of resources with varying configurations.

4.5. Performance results

Based on the results obtained for the peak performance benchmark in Section 3.6, we follow the same procedure for obtaining the maximum performance of the stencil code. Starting at the highest number of replications achieved for the synthetic benchmark, the number of PEs is gradually lowered until the build is successful. The process is also attempted for various widths of the pipeline, altering the ratio between n and K. We additionally sweep over lower configurations in order to do investigate scaling of resource consumption with replication, treated in Section 4.6. The build results are included in Figure 4.5, indicating successful and failed builds with circles and crosses, respectively. Because we perform 4 operations per cycle (3 additions and 1 multiplication) per processing element and the kernel is clocked at 250 MHz, the reported number of processing elements can be translated directly into performance in GOp/s. The highest performance result will therefore be the rightmost green circle in the plot. The depth of a given configuration can be computed as $\frac{\#\text{PEs}}{\text{pipeline width}}$, where the pipeline width is measured in number of operands. Correctness in terms of domain and performance was verified after determining the highest performing build, but for that build only. The highest build meeting timing is for 256 processing elements with a data width of 8 bytes for a 256×256 grid, corresponding to a performance of:

$$F_{\text{Jacobi2D}} = 250\,\text{cycle/s} \cdot 256 \cdot 4\,\text{Op/cycle} = 256\,\text{GOp/s} \tag{4.10}$$

This corresponds to $\frac{256\,\text{GOp/s}}{322\,\text{GOp/s}} = 80\%$ of the highest measured performance for the given combination of floating point operations (see Section 3.6.2), and to $\frac{256\,\text{GOp/s}}{450\,\text{GOp/s}} = 57\%$ of the theoretical peak.

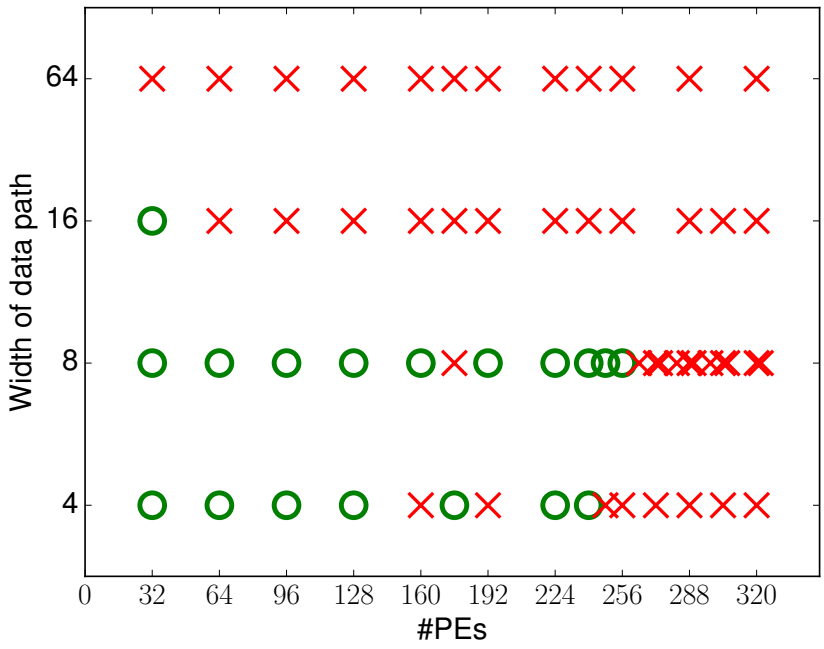


Figure 4.5.: Builds for various configurations of data width, number of processing elements and grid sizes. Because of the clock frequency of 250 MHz and the 4 operations performed per stencil evaluation, the number of processing elements can be directly translated into performance in GOp/s. Green circles indicate a successful build, while red crosses are builds that failed placement or timing.

4.5.1. Verifying performance

To verify that the design works at the expected performance for a sustained workload, we look at the number of cycles spent in the computational kernel when running it for a large number of iterations. The expected number of cycles for a 2D stencil built with the architecture proposed here is:

$$\begin{aligned} c_{\text{stencil}} &= \text{domain saturation} + \text{hardware saturation} + \text{number of iterations} \Rightarrow \\ c_{\text{stencil}} &= \text{rows to saturate} \cdot \text{pipeline depth} \cdot \frac{\text{cols}}{\text{pipeline width}} + \text{latency} \\ &\quad + \frac{\text{rows} \cdot \text{cols}}{\text{pipeline width}} \cdot \frac{\text{timesteps}}{\text{pipeline depth}} \end{aligned} \tag{4.11}$$

In the case of our 2D Jacobian result, we need to saturate the pipeline with a single row before we can start evaluating values, as we use fixed boundaries. The highest performance build achieved has 256 processing elements and a width of 8 bytes (corresponding to 2 single precision floating point numbers). The depth of the pipeline is the number of processing elements instantiated divided by the width ($256/2 = 128$). The hardware latency of the pipeline is reported by the synthesis tool as being 3460 cycles. The benchmark was run for 16384 timesteps. The resulting expected number of cycles is:

$$c_{\text{expected}} = 1 \cdot 128 \cdot \frac{256}{2} + 3460 + \frac{256 \cdot 256}{2} \cdot \frac{16384}{128} = 4214148 \tag{4.12}$$

In comparison, the cycle counter of the reference design reports 4214158 cycles, which is 10 cycles more than the expected number, exactly as seen for the peak performance benchmarks in Table 3.6. We can thus conclude that the program runs at the expected performance for a sustained workload, as the circuit only spends one cycle per work item in addition to the saturation overhead accounted for above.

4.5.2. Failing wide data paths

The stencil designs generally did not meet timing for data paths wider than 8 bytes. This is likely due to routing congestion, as too many BRAM blocks and DSP slices have to route to the same logic where the control flow is located. The failing paths are both from BRAM blocks to logic and between logic and DSP slices. Which designs passed timing is included as circles in Figure 4.5.

4.6. Resource scaling

In order to gain insight in the scalability of systolic array designs, we investigate the increase in resource consumption with increasing number of processing elements. For the design to scale to arbitrarily large FPGAs, the resource consumption when replicating processing elements must be linear. Scaling with number of processing elements is plotted in Figure 4.6 for LUTs and FFs, where every vertical slice is a constant number of processing elements, and thus a constant number of DSPs consumed. The y-axes

are given as fraction of the maximum available resources on the XC7VX690T board, with the exception of the power plot, which is a fraction of the highest measured power consumption between all configurations. The power numbers reported here are as reported by the Vivado tool, and are merely an estimate. They have only been included as a relative comparison between the different configurations, as the absolute numbers have little meaning without taking the whole board (as they are for the FPGA chip only) into account and performing measurements.

These plots reveal two key observations. The first is that scaling of logic resources is linear in the number of processing elements, with the exception of power, which is sublinear. The second observation is that, for a vertical slice, the number of logic resources consumed decreases for wider data paths. Despite the algorithm in principle offering peak performance at any memory bandwidth, the amount of control logic saved by gathering more computations in a single processing element means fewer resources consumed, which in turn could allow for more processing elements on the board. With the current design, however, only data paths up to 8 byte wide meet timing. If the fundamental issue of wider data paths not meeting timing could be solved, the resources saved could potentially lead to higher number of floating point units on the chip.

4.7. Discussion

The high memory bandwidth of GPUs has made them a popular target for stencil code generators, and much research has gone into developing optimal *tiling* strategies of the iteration space to optimize the memory access pattern and reducing redundant computations [29, 30, 31], increasing the effective computational intensity. To the author's knowledge, the highest measured performance by the time of writing this work for the 2D Jacobian stencil on a single accelerator is by Rawat et al. [32], demonstrating 396 GOp/s on a Tesla K20 GPU. The implementation presented here reaches $256/396 = 65\%$ of this performance. As the stencil computation can not make use of FMA (since the multiplication is done *after* the additions), the GPU number corresponds to $396\,\frac{\text{GOp}}{\text{s}}/1762\,\frac{\text{GOp}}{\text{s}} = 22.5\%$ of the FMA ceiling. As noted above, the stencil performance of 256 GOp/s achieved on the AlphaData 7V3 card corresponds to 57% of the theoretical peak performance given the distribution of additions and multiplications. We make three important observations from this:

- When algorithms can be fit in to the streaming pipeline model, FPGA performance can achieve high fractions of board utilization for a non-synthetic application. By tweaking the processing elements there is potential for increasing the number instantiated significantly, further approaching peak utilization. This is interesting in particular when moving to larger FPGAs (see Section 4.8.5), as the algorithms on the form of the systolic array used for stencils proposed here scale directly with the peak performance, because the computational intensity scales with the number of processing elements instantiated (we can consider it infinite in practice for this architecture). The stencil results presented here have been limited by the place and route process not meeting timing due to congestion or high resource

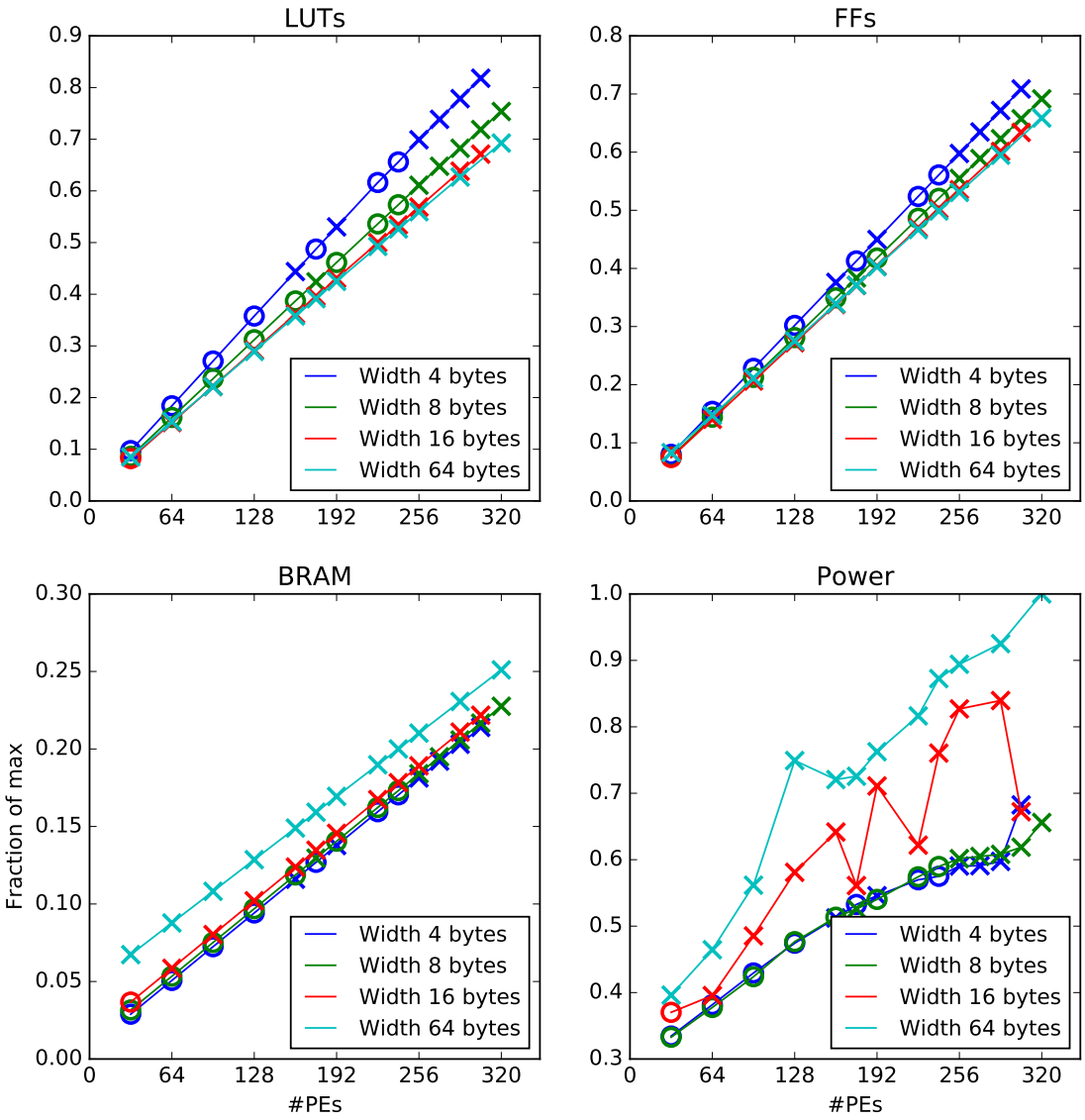


Figure 4.6.: Scaling of various resources for increasing number of processing elements for the stencil design. Only builds configured with a 64×64 grid are included to allow comparing BRAM, so not all builds from Figure 4.5 are included. Circles indicate successful builds, while crosses indicate builds that failed to meet timing.

utilization, rather than by conceptual barriers, and Section 4.6 showed that the resource scaling for increasing number of processing elements in linear. It however remains to be shown that the timing issue is due to high fractional utilization, and not the absolute utilization number, in order to scale to bigger chips. This can not be asserted here, as we only have data for a single chip.

- Contemporary FPGAs are not optimized for floating point computations. Although there is potential in moving to fixed point types supported natively by the DSP units (see Section 4.8.3), there is vast potential in FPGAs engineered specifically for HPC. This would involve supporting the required data types by implementing fewer but wider routing paths to accommodate large data types, and in particular offering native support in the hardened components (perhaps even small vector units). The Altera Stratix 10 generation promises an HPC-oriented device, but is not yet commercially available.

- The performance offered on the FPGA relative to the GPU is interesting in terms of power efficiency (see below).

The next section will elaborate on the third point. A final thing to note is that Rawat et al. use an implementation based on streaming, suggesting that this is a promising pattern at *least* in the field of stencils.

4.7.1. Power efficiency

When FPGAs come up in an HPC context, power efficiency is usually a key point. To properly address this, accurate power measurements must be made of all compared architectures. Not unlike peak performance numbers, datasheet power consumption does often not reflect reality well [16]. As mentioned in Section 1.7, datasheet power consumption are upper bounds, thus clashing with peak performance when used in the denominator of computing power efficiency. Still we below compute a ballpark number for the stencil kernel using measured peak power consumption numbers from Servesh et al. [16] for floating point performance microbenchmarks, who arrive at 225 W and 25 W for the Tesla K20 and the AlphaData 7V3 card, respectively. These numbers include power consumption of off-chip memory on the accelerators. Equation (1.8) then evalutates to:

$$\begin{aligned} E_{\text{stencil,K20}} &= \frac{396\,\text{Op/s}}{225\,\text{W}} = 1.76\,\text{Op/J} \\ E_{\text{stencil,7V3}} &= \frac{256\,\text{Op/s}}{25\,\text{W}} = 10.24\,\text{Op/J} \end{aligned} \tag{4.13}$$

corresponding to a factor of $10.24/1.76 = 5.82$ increase in power efficiency for a 35% drop in performance on a single accelerator card. While these numbers are rough estimates and should be taken as such, the FPGA implementation shows promise of delivering high performance per power, in particular if there is still significant untapped potential in the HLS implementation. To make any definite conclusions, however, accurate performance and power measurements must be made of each accelerator (see Section 4.8.7).

4.8. Future work

We will now address some of the many paths that can be taken to improve upon the work of implementing efficient stencil code on FPGA presented here.

4.8.1. Finish proposed implementation

As noted in Section 4.4.5, the proposed design was not finished in time for writing this thesis, so the benchmarked design uses twice the amount of BRAM resources necessary. The next step is to finish implementing the proposed design to halve the on-chip memory cost.

4.8.2. Generalizing the model

In this work we treated a specific 2D stencil. Generalizing the method in a more formal way than what was done in Section 4.4.3 to at least 3-dimensional stencils for arbitrary dependency sets would create an interesting framework in which to pursue stencil computations on FPGA. This would also help in assessing the feasibility of larger stencils given the available BRAM resources on the FPGA.

4.8.3. Fixed point types

As suggested for the peak performance benchmark in Section 3.10.3, moving to fixed point types supported natively by the DSP units and requiring significantly less logic due to their integer nature has a significant potential for increase in performance. This would also cut the amount of DSPs required for each stencil evaluation by up to two thirds, which could help the timing issues by reducing congestion.

4.8.4. Off-chip memory

Because we were restricted to a narrow PCIe bridge when developing the stencil algorithm presented here, we used a large BRAM buffer on chip in place of streaming data from off-chip memory. While this successfully emulated the required streaming semantics, it needs to be investigated whether the memory controller can sustain feeding the pipeline without introducing bubbles (see Section 2.3.3). A future version of the implementation should use two AXI Stream interfaces to stream to and from memory once this is introduced to the Xilinx reference design or a similar framework.

4.8.5. Larger FPGA

As for the peak performance benchmark, building the stencil kernel for a larger FPGA and comparing resource utilization would be an important sign of scalability of the algorithm. SDAccel supports the Virtex Ultrascale KU115 FPGA with 5520 DSP slices, and the Xilinx reference design used here is being built to support this as well. AlphaData has released a board using this chip, the ADM-PCIE-KU3, so running the stencil code on this board would be an obvious next step.

4.8.6. Scaling with grid size

Because data is streamed from a top left to bottom right fashion and must be buffered until the last time it is used, the method described here requires storing in order of n elements, where n is the smallest dimension in a 2D grid. For 3D this would increase to $n \times m$, where n and m are the two smallest dimensions in a 3D grid. For real life applications these numbers can become significantly higher than e.g. the maximum computed in Equation 4.9, so a general algorithm would have to deal with these memory considerations. One solution could be scaling to multiple FPGAs, exchanging borders via FIFOs over an interconnect, as maximum performance can always be reached on each FPGA by unrolling the time dimension, regardless of the per-FPGA grid size. Other solutions could involve increasing the reliance on higher memory bandwidths, but any solution where the amount of bandwidth scales with the amount of processing elements loses the benefits of the infinite computational intensity offered by the streaming pipeline architecture. The deep pipelining architecture presented here will suffer even more than fixed architectures from waiting for memory, as this will introduce bubbles to the pipeline, effectively losing C in Equation (1.6) operations for every cycle waiting for memory. Fixed architectures use tiling to minimize the amount of memory transferred from global memory, and looking into a tiling approach that preserves a perfect pipeline despite loading elements at the borders between tiles multiple times and breaking up the iteration space is among the highest priority of future work. Deest et al. [33] have presented a work in progress looking into tiling on FPGA, but their preliminary results have not exceeded 40 GOp/s.

4.8.7. Accurate power measurements

To accurately assess the power efficiency of stencil computations on FPGA, power measurements should be done and compared to other architectures running respective state of the art implementations. The preliminary estimate done in Section 4.7.1 suggests a factor of 5.6, which is promising enough to warrant further investigation, especially considering the potential for further optimizations of the FPGA stencil code.

5. Discussion

This chapter discusses the lessons learned throughout the thesis, touches upon the issue of productivity when programming FPGAs, and comments on the future of HPC on FPGAs. The work is concluded with a list of the contributions made and a summary of proposed future work.

5.1. Performance modeling

Here we discuss the usefulness of the two models used for this work, namely the peak performance model based on maximizing resource utilization presented in Section 3.2, and the well known roofline model used to compare the FPGA to other architectures in Section 3.8.

5.1.1. Peak performance model

The model of peak performance by replicating identical processing elements presented in Section 3.2 proved effective in predicting performance for synthetic benchmarks run on hardware (see Section 3.6). Many synthetic benchmarks achieved 100% of the expected performance for their configuration, and the best result reached 98% of the highest performance predicted. The highest performing stencil build corresponded to 57% of peak performance (see Section 4.5), and 80% of the highest measured performance for the given configuration using the synthetic benchmark. Because of the opaque coupling between frequency and resource utilization, the model is however restricted to predict the the number of operations C performed per cycle using Equation (3.2) for a given set of available resources, and leaves the optimization of the coupled frequency and utilization parameters, Equation (3.4), as an engineering challenge. This did not affect the benchmarks done here, as f was fixed by the frameworks used. As expected for a peak model, it also revealed some shortcomings when it comes to realistic applications, such as the stencil program:

- Predicting the amount of resources consumed by a processing element can only be done to some approximation, typically as a lower bound computed from the computational units and storage units following the characteristics of the problem. In real applications a non-neglicible part of logic will be consumed for control and routing, and this number is hard to predict before having a ready implementation.
- Varying the width of the pipeline K can impact the resource utilization of the processing element, thus affecting the peak performance achievable with the same amount of computational units instantiated on the chip. This was demonstrated in

Section 4.6, where it was seen that wider pipelines significantly reduced the total amount of control logic needed, ultimately affecting the number of computational units that could be instantiated.

- For a more complex design such as the stencil, meeting timing is not just a constraint when approaching maximum utilization, but also when dealing with congestion inside the processing elements. Even though they use fewer overall resources, almost all configurations with a data path wider than 8 bytes fail timing (see Section 4.5).

Despite these limitations, predicting peak performance on FPGA is a satisfying exercise, as it is accurately confirmed by the benchmarks, and can be deduced directly from the pipeline achieved, assuming there are no bubbles introduced by memory hiccups.

5.1.2. Roofline model

Using the peak numbers predicted and measured, we constructed a roofline model for the AlphaData 7V3 card in Section 3.8 and plotted it alongside the fixed architectures from Section 1.7.1. From the plot we see that the theoretical peak performance is not competitive with GPU and many-core, and the memory performance in particular s far behind the competitors (even without the further constraints of SDAccel), requiring a massive computational intensity to reach the ridge point. Current generations of FPGA hardware are not optimized for HPC, as the hardened DSP units are optimized for low precision fixed point computations (see Section 1.4.1). Likewise, memory bandwidth has not been central in designing the chip used, as it only offers enough pinouts for two DDR3 modules, and leaves the rest up to the third party board vendor. The merit of the FPGA comes from its predictability: once a design is successfully built we can predict its performance down to the exact number of cycles, and we showed that synthetic benchmarks can reach the predicted peak with a fairly straightforward implementation. Additionally, because we have such fine-grained control over how data is buffered, it is actually possible to push the computational intensity to the high levels needed to achieve performance, as was shown for the stencil implementation, although it did not quite reach the performance levels of the synthetic benchmark. Because of this control, the roofline arguably becomes a more accurate tool to describe FPGA performance, as it is more likely that we can extract the parameters exactly.

5.2. FPGA programming productivity

The subject of productivity has been left largely untouched throughout the thesis, but is a central argument when discussing the viability of a hardware architecture. As briefly noted in Section 2.1, running the source to hardware flow is very time consuming for large designs with both SDAccel and Vivado, and since aiming for peak performance always involves high resource utilization, most benchmarks presented here took four to five hours to build for each individual configuration. The massive memory footprints of up to 20 GB also mean that few personal computers would be able run the software

without heavy swapping to disk, let alone run multiple concurrent compilations to save time. What makes FPGA design more realistic are the tools that can test correctness before or during the full build, such as reports generated by the high level synthesis and simulation tools.

While high level synthesis makes FPGAs more accessible to users coming from software design, there is no way around diving into hardware design when implementing efficient codes. There seems to be a gap to be filled by a suitable abstraction to the FPGA architecture, and OpenCL in its current form does not look like a satisfactory choice (briefly discussed in Section 2.1.2). When using Vivado HLS, as is often the case when trying to fit into the C/C++ language, the power and flexibility of the language become the programmer's biggest enemy, as their semantics do little to restrict bad implementation patterns. However, once the correct patterns have been established (an attempt was made in Section 2.4), HLS offers fast prototyping and decent diagnostics while designing the hardware, and can be fairly easily integrated onto the FPGA using the SDAccel framework. Still, the total road from algorithm to hardware is long, and devising the fairly simple stencil code included here took of the order of months and has much work ahead of it yet, failing FPGAs on the scale of speedup per programming hour.

5.3. Reconfigurable computing in HPC

As was demonstrated and mentioned multiple times already, both FPGA floating point performance and memory bandwidth are hampered by the chip design not being oriented towards HPC. Addressing these two issues could potentially allow for Moore's law to benefit the end user's domain directly, rather than through further replication of existing general purpose units. Adding in the potential for energy efficiency, assuming that more HPC-oriented devices could keep the power consumption at the low rates of current chips, FPGA vendors could set themselves on a path to become a central player on the road to exascale computing.

Although they have become so in practice, reconfigurable architectures are not principally synonymous with FPGAs. Reconfigurability can be employed at different levels of granularity, and developing products working at a less general purpose level, but more optimized towards common HPC patterns, could be the step needed for reconfigurable architectures to become a dangerous competitor to fixed ones, although this was already proposed in 1994 [34], and has not seemed to pick up since. The trade-off is at tricky one, as specializing in any direction will be a disadvantage for others, therefore opening new markets but closing others.

Perhaps more than anything, the issues of productivity mentioned above need to be addressed, giving programmers a powerful and easy to use abstraction to go from domain to hardware, encouraging laymen to dive into the world of reconfigurable computing.

5.4. Contributions and conclusion

The main contributions of this work are:

- Descriptions of useful HLS patterns and how they map to hardware.
- A model of peak performance for replicable processing elements on an FPGA based on the available hardware resources, which we applied to the AlphaData 7V3 board and compared to common fixed architectures.
- Benchmarks to demonstrate the predicted performance on hardware. In the SDAccel framework we reached 302 GOp/s, which to the author's knowledge is the highest number obtained with SDAccel on the Virtex 7 architecture. Based on a custom platform provided by Xilinx we achieved 548 GOp/s, and we showed how to verify the results by accounting for the exact number of cycles spent in the kernel.
- Applying the developed methods to the domain of stencils, we proposed a temporally pipelined streaming design that scales with the amount of area on the chip for a constant memory bandwidth. A variation of the design was implemented in hardware, and we obtained a performance of 256 GOp/s for the 2D Jacobian stencil. The exact number of cycles spent in the kernel was accounted for in order to verify the performance claimed.

Finally, we discussed the current state of FPGAs based on the results measured, tools used to program them and the productivity that they offer, as well as the role of reconfigurable computing the future of HPC. We concluded that there is potential for FPGA on the road to exascale computing, but that steps need to be taken to better accommodate the needs of HPC, in particular in terms of productivity and the potential of floating point performance.

5.5. Future work

Section 2.4 enumerated a number of programming techniques to guide the implementation of efficient FPGA hardware using the Vivado HLS tool. To promote investigations into the viability of FPGAs in HPC, a thorough guide on general performance concepts and specifics on to how to wrangle the available tools would be a valuable contribution.

Section 3.10 and Section 4.8 proposed future work for the peak benchmarks and stencil implementation.

Peak performance predictions and benchmarks were only done for single precision floating point units, and expanding this to include other floating point precisions, as well as fixed point and integer computations, would give a more complete picture of the performance potential of FPGAs. Additionally, the hardware used for the experiments has already been succeeded by a new generation, and upgrading to a more modern chip would give a better picture of the contemporary state of the art.

The 2D Jacobi systolic array design proposed was only partially achieved, leaving the work on reducing the required buffer space to the minimum required of two lines.

Even with this improvement, the issue of scaling to very large grid dimensions must be addressed, and the method should be generalized to arbitrary stencils in order to make the proposed architecture useful in real scenarios. The measured performance did not reach the level of a state of the art GPU implementation, but was close enough, in particular on power efficiency, that improvements on the architecture could make it competitive. Measuring power consumption and comparing to other hardware architectures is also left for future work.

Bibliography

[1] Hauck, Scott and DeHon, Andre, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[2] (2015, May) DS180 - 7 series FPGAs overview. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[3] (2014, November) UG474 - 7 series FPGAs configurable logic block. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[4] (2014, November) UG479 - 7 series FPGAs DSP48E1 slice user guide. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[5] (2014, November) UG473 - 7 series FPGAs memory resources. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[6] Williams, Samuel, Waterman, Andrew, and Patterson, David, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[7] Micron. (2001) TN-46-05: General DDR SDRAM functionality. [Online]. Available: https://www.micron.com/resource-details/dc375728-e2a1-4911-a1b2-b4200602f1b8

[8] Agner Fog, "Instruction tables - lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," January 2016. [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf

[9] Vivado high-level synthesis. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[10] (2016, June) UG835 - Vivado design suite tcl command reference guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug835-vivado-tcl-commands.pdf

[11] The open standard for parallel programming of heterogeneous systems. [Online]. Available: https://www.khronos.org/opencl/

[12] Vivado design suite. [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html

[13] (2016) Performance and resource utilization for floating-point v7.1. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html#virtex7

[14] R. P. Brent and H. T. Kung, "Systolic VLSI arrays for polynomial GCD computation," *IEEE Transactions on Computers*, vol. C-33, no. 8, pp. 731–736, Aug 1984.

[15] Strenski, Dave, Simkins, Jim, Walke, Richard, and Wittig, Ralph, "Evaluating FPGAs for floating-point performance," in *Proceedings of the 2008 Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications*. IEEE, 2008, pp. 1–6.

[16] Muralidharan, Servesh, O'Brien, Kenneth, and Lalanne, Christian, "A semi-automated tool flow for roofline analysis of OpenCL kernels on accelerators," in *First International Workshop on Heterogeneous High-performance Reconfigurable Computing*, ser. H2RC'15, 2015.

[17] da Silva, Bruno, Braeken, An, D'Hollander, Erik H., and Touhafi, Abdellah, "Performance modeling for fpgas: extending the roofline model with high-level synthesis tools," *International Journal of Reconfigurable Computing*, vol. 2013, p. 7, 2013.

[18] (2013, November) AlphaData ADM-PCIE-7V3 datasheet. [Online]. Available: http://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf

[19] Virtex-7 FPGAs. [Online]. Available: http://www.xilinx.com/support/documentation/selection-guides/virtex7-product-table.pdf

[20] Smith, Gordon D., *Numerical solution of partial differential equations: finite difference methods*. Oxford University Press, 1985.

[21] Doms, G. and Schättler, U., "The nonhydrostatic limited-area model LM (lokal-modell) of DWD," *Part I: Scientific documentation, Deutscher Wetterdienst (DWD)*, 1999.

[22] Taflove, Allen, Hagness, Susan C. *et al.*, "Computational electrodynamics: The finite-difference time-domain method," *Norwood, 2nd Edition, MA: Artech House, 1995*, 1995.

[23] Cédric Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings of the PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, France, September 2004, pp. 7–16.

[24] Bondhugula, Uday, Hartono, A., Ramanujam, J., and Sadayappan, P., "PLuTo: A practical and fully automatic polyhedral program optimization system," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*, 2008.

[25] Gysi, Tobias, Grosser, Tobias, and Hoefler, Torsten, "MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 177–186. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751223

[26] Kobori, Tomoyoshi, Maruyama, Tsutomu, and Hoshino, Tsutomu, "A cellular automata system with FPGA," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2001. FCCM'01.* IEEE, 2001, pp. 120–129.

[27] Anshuman, Verma, Helal, Ahmed E., Krommydas, Konstantinos, and Feng, Wu-chun, "Accelerating workloads on FPGAs via OpenCL: A case study with OpenDwarfs," *Computer Science Technical Reports*, 2016.

[28] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 695–705, March 2014.

[29] Bandishti, Vinayaka, Pananilath, Irshad, and Bondhugula, Uday, "Tiling stencil computations to maximize parallelism," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 40:1–40:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389051

[30] Grosser, Tobias, Cohen, Albert, Holewinski, Justin, Sadayappan, P., and Verdoolaege, Sven, "Hybrid hexagonal/classical tiling for GPUs," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 66:66–66:75. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544160

[31] Grosser, Tobias, Cohen, Albert, Kelly, Paul H. J., Ramanujam, J., Sadayappan, P., and Verdoolaege, Sven, "Split tiling for GPUs: Automatic parallelization using trapezoidal tiles," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6. New York, NY, USA: ACM, 2013, pp. 24–31. [Online]. Available: http://doi.acm.org/10.1145/2458523.2458526

[32] Rawat, Prashant Singh, Hong, Changwan, Ravishankar, Mahesh, Grover, Vinod, Pouchet, Louis-Noël, and Sadayappan, P., "Effective resource management for enhancing performance of 2D and 3D stencils on GPUs," in *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, ser. GPGPU '16. New York, NY, USA: ACM, 2016, pp. 92–102. [Online]. Available: http://doi.acm.org/10.1145/2884045.2884047

[33] Deest, Gaël, Estibals, Nicolas, Yuki, Tomofumi, Derrien, Steven, and Rajopadhye, Sanjay, "Towards scalable and efficient FPGA stencil accelerators," in *Proceedings of*

the 6th International Workshop on Polyhedral Compilation Techniques (IMPACT'16), held with HIPEAC'16, 2016.

[34] Hartenstein, Reiner W., Kress, Rainer, and Reinig, Helmut, "A new FPGA architecture for word-oriented datapaths," in *Proceedings of Field-Programmable Logic Architectures, Synthesis and Applications: 4th International Workshop on Field-Programmable Logic and Applications, FPL'94 Prague, Czech Republic, September 7–9, 1994*, Hartenstein, Reiner W. and Servít, Michal Z., Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 144–155. [Online]. Available: http://dx.doi.org/10.1007/3-540-58419-6_85

Appendices

A. Burst class implementation

Vivado HLS implementation of a class to facilitate reading and writing wide bursts to and from memory interfaces.

```
/// \author Johannes de Fine Licht (jannesd@xilinx.com)
/// \date April 2016
#pragma once
#include "ap_int.h"

namespace hlsUtil {

namespace {

template <unsigned byteWidth>
struct UnsignedIntType {};

template <>
struct UnsignedIntType<sizeof(unsigned char)> {
  typedef unsigned char T;
};

template <>
struct UnsignedIntType<sizeof(unsigned short)> {
  typedef unsigned short T;
};

template <>
struct UnsignedIntType<sizeof(unsigned int)> {
  typedef unsigned int T;
};

template <>
struct UnsignedIntType<sizeof(unsigned long)> {
  typedef unsigned long T;
};

} // End anonymous namespace

template <typename T, unsigned byteWidth>
class Burst {

  typedef typename UnsignedIntType<sizeof(T)>::T Pack_t;
  static const int kElementsPerBurst = byteWidth / sizeof(T);
  static const int kBits = 8 * sizeof(T);

public:
  Burst() {}
  Burst(T const arr[kElementsPerBurst]) {
    #pragma HLS INLINE
    Pack(arr);
  }
  void Pack(T const arr[kElementsPerBurst]) {
    #pragma HLS INLINE
    #pragma HLS PIPELINE II=1 enable_flush
```

```
Burst_Pack:
    for (int i = 0; i < kElementsPerBurst; ++i) {
      #pragma HLS UNROLL
      T element = arr[i];
      Pack_t temp = *reinterpret_cast<Pack_t const *>(&element);
      data_.range((i + 1) * kBits - 1, i * kBits) = temp;
    }
  }
  void Unpack(T arr[kElementsPerBurst]) const {
    #pragma HLS INLINE
    #pragma HLS PIPELINE II=1 enable_flush
Burst_Unpack:
    for (int i = 0; i < kElementsPerBurst; ++i) {
      #pragma HLS UNROLL
      Pack_t temp = data_.range((i + 1) * kBits - 1, i * kBits);
      arr[i] = *reinterpret_cast<T const *>(&temp);
    }
  }
  void operator<<(T const arr[kElementsPerBurst]) {
    #pragma HLS INLINE
    Pack(arr);
  }
  void operator>>(T arr[kElementsPerBurst]) const {
    #pragma HLS INLINE
    Unpack(arr);
  }

private:
  ap_uint<8 * byteWidth> data_;
};

} // End namespace hlsUtil
```

B. Stencil kernel implementation

Implementation of stencil kernel in Vivado HLS. Functions suffixed with `Four` were used for the benchmarks in the thesis, while the non-suffixed functions are a newer design in progress decoupling buffering the computation. All variables prefixed with `k` are compile-time constants configured by CMake in a configuration header file.

```
#include "Stencil.h"
#include "Dataflow.h"
#include <hls_stream.h>
#include <cstring>

void ReadFeedbackSingle(hls::stream<Burst> &in, hls::stream<Burst> &feedback,
                        hls::stream<Burst> &out) {
  #pragma PIPELINE II=1 enable_flush
  if (!in.empty()) {
    out.write(in.read());
  } else if (!feedback.empty()) {
    out.write(feedback.read());
  }
}

void WriteFeedbackSingle(hls::stream<Burst> &in, hls::stream<Burst> &feedback,
                         hls::stream<Burst> &out) {
  #pragma PIPELINE II=1 enable_flush
  static unsigned i = 0;
  #pragma HLS RESET variable=i
  if (!in.empty()) {
    Burst element = in.read();
    if (i < kTotalBursts * (kTimestepsPerStage - 1)) {
      feedback.write(element);
    } else {
      out.write(element);
    }
    ++i;
    // i = (i + 1) % (kTotalBursts * kTimestepsPerStage);
  }
}

void FillBoundary(Element_t array[kElementsPerBurst]) {
  #pragma HLS INLINE
FillBoundary:
  for (unsigned i = 0; i < kElementsPerBurst; ++i) {
    #pragma HLS UNROLL
    array[i] = kBoundary;
  }
}

Burst ScalarBurst(const Element_t value) {
  #pragma HLS INLINE
  Element_t arr[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=arr complete
BoundaryBurst:
  for (unsigned i = 0; i < kElementsPerBurst; ++i) {
```

```
    #pragma HLS UNROLL
    arr[i] = value;
  }
  return Burst(arr);
}

void Move(Element_t const src[kElementsPerBurst],
          Element_t tgt[kElementsPerBurst]) {
  #pragma HLS INLINE
Move:
  for (unsigned i = 0; i < kElementsPerBurst; ++i) {
    #pragma HLS UNROLL
    tgt[i] = src[i];
  }
}

template <unsigned stage>
void GatherData(Stream &in, Stream &northOut, Stream &westOut, Stream &eastOut,
                Stream &southOut) {
  #pragma PIPELINE II=1 enable_flush

  // Iteration counter
  static int i = -kBurstsPerLine;
  #pragma HLS RESET variable=i

  // Compute various indices
  const unsigned time = i / kTotalBursts;
  const unsigned iGrid = i % kTotalBursts;
  const unsigned iRow = iGrid / kBurstsPerLine;
  const unsigned iBurst = iGrid % kBurstsPerLine;

  const bool streaming = i >= 0 && i < kTotalBursts * kTimestepsPerStage;
  // Shifted north by one, because the last row is never used as a north value,
  // and to saturate the pipeline to begin with.
  const bool streamingNorth =
      i < kTimestepsPerStage * kTotalBursts - kBurstsPerLine;
  // Shifted south by one, because the first row is never used as a south value,
  // and to drain the pipeline in the end.
  const bool streamingSouth =
      i >= kBurstsPerLine &&
      i < kTimestepsPerStage * kTotalBursts + kBurstsPerLine;
  // Shifted east by one because we need to get the eastern border from the next
  // iteration
  const bool streamingCenter =
      i >= 1 && i < kTimestepsPerStage * kTotalBursts + 1;

  Burst inputBurst;
  Element_t inputArr[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=inputArr complete

  if (streaming) {
    if (in.empty()) {
      return;
    }
    inputBurst = in.read();
    inputBurst >> inputArr;
  }

  if (streamingNorth) {
    if (iRow == kNX - 1 || !streaming) {
      northOut.write(ScalarBurst(kBoundary));
    } else {
      northOut.write(inputBurst);
```

```
    }
  }

  if (streamingSouth) {
    if (iRow == 0 || !streaming) {
      southOut.write(ScalarBurst(kBoundary));
    } else {
      southOut.write(inputBurst);
    }
  }

  // Forward elements from previous iteration
  static Element_t forwardWest[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=forwardWest complete
  static Element_t forwardEast[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=forwardEast complete
  static Element_t eastBorder = kBoundary;

  // Append edge to burst forwarded from previous iteration
  forwardEast[kElementsPerBurst - 1] = iBurst > 0 ? inputArr[0] : kBoundary;

  if (streamingCenter) {
    westOut.write(Burst(forwardWest));
    eastOut.write(Burst(forwardEast));
  }

  // Gather west and east values
ShiftData:
  for (unsigned j = 0; j < kElementsPerBurst; ++j) {
    #pragma HLS UNROLL
    if (j > 0) {
      forwardWest[j] = inputArr[j - 1];
    }
    if (j < kElementsPerBurst - 1) {
      forwardEast[j] = inputArr[j + 1];
    }
  }
  if (iBurst == 0) {
    forwardWest[0] = kBoundary;
  } else {
    forwardWest[0] = eastBorder;
  }
  eastBorder = inputArr[kElementsPerBurst - 1];

  ++i;
}

template <unsigned stage>
void Compute(Stream &northIn, Stream &westIn, Stream &eastIn, Stream &southIn,
             Stream &out) {
  #pragma HLS PIPELINE II=1 enable_flush

  Element_t north[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=north complete
  Element_t west[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=west complete
  Element_t east[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=east complete
  Element_t south[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=south complete

  if (northIn.empty() || westIn.empty() || eastIn.empty() || southIn.empty()) {
    return;
```

```
  }

  northIn.read() >> north;
  westIn.read() >> west;
  eastIn.read() >> east;
  southIn.read() >> south;

  Element_t result[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=south complete

PipelineWidth:
  for (unsigned j = 0; j < kElementsPerBurst; ++j) {
    #pragma HLS UNROLL
    result[j] = static_cast<Element_t>(0.25) *
                (north[j] + west[j] + east[j] + south[j]);
  }

  out.write(Burst(result));
}

template <unsigned recurse>
void UnrollCompute(Stream pipes[kDepth]) {
  #pragma HLS INLINE

  UnrollCompute<recurse - 1>(pipes);

  static const unsigned kNorthDepth = 2 * kBurstsPerLine + 1;
  static const unsigned kCenterDepth = kBurstsPerLine;
  static Stream north("north");
  #pragma HLS STREAM variable=north depth=kNorthDepth
  static Stream west("west");
  #pragma HLS STREAM variable=west depth=kCenterDepth
  static Stream east("east");
  #pragma HLS STREAM variable=east depth=kCenterDepth
  static Stream south("south");

  GatherData<recurse - 1>(pipes[recurse - 1], north, west, east, south);

  Compute<recurse - 1>(north, west, east, south, pipes[recurse]);
}

template <>
void UnrollCompute<0>(Stream *) {}

void EntryReferenceDesign(Stream &input, Stream &output) {

  #pragma HLS INTERFACE axis port=input
  #pragma HLS INTERFACE axis port=output

  #pragma HLS DATAFLOW

  static Stream pipes[kDepth + 1];
  #pragma HLS STREAM variable=pipes depth=kPipeDepth
  static Stream feedback("feedback");
  #pragma HLS STREAM variable=feedback depth=kNX*kBurstsPerLine
  #pragma HLS RESOURCE variable=feedback core=FIFO_BRAM

  ReadFeedbackSingle(input, feedback, pipes[0]);

  UnrollCompute<kDepth>(pipes);

  WriteFeedbackSingle(pipes[kDepth], feedback, output);
```

```
}

template <unsigned stage>
void ComputeFour(Stream &in, Stream &out) {
  #pragma PIPELINE II=1 enable_flush

  static Buffer northBuffer("northBuffer");
  #pragma HLS STREAM variable=northBuffer depth=kBurstsPerLine
  #pragma HLS RESOURCE variable=northBuffer core=FIFO_BRAM

  static Buffer westBuffer("westBuffer");
  #pragma HLS STREAM variable=westBuffer depth=kBurstsPerLine
  #pragma HLS RESOURCE variable=westBuffer core=FIFO_BRAM

  static Buffer centerBuffer("centerBuffer");
  #pragma HLS STREAM variable=centerBuffer depth=kBurstsPerLine
  #pragma HLS RESOURCE variable=centerBuffer core=FIFO_BRAM

  static Buffer eastBuffer("eastBuffer");
  #pragma HLS STREAM variable=eastBuffer depth=kBurstsPerLine
  #pragma HLS RESOURCE variable=eastBuffer core=FIFO_BRAM

  // One extra line to flush the pipeline
  static const unsigned kTotalIterations =
      (kNX + 1) * kBurstsPerLine * kTimestepsPerStage;

  // Iteration counter
  static unsigned i = 0;
  #pragma HLS RESET variable=i
  // #pragma AP DEPENDENCE variable=i inter false

  const unsigned time = i / kTotalBursts;
  const unsigned iLocal = i % kTotalBursts;
  const unsigned inputRow = iLocal / kBurstsPerLine;
  const unsigned workingRow = (inputRow + kNX - 1) % kNX;
  const unsigned lineBurst = iLocal % kBurstsPerLine;

  // Input burst fanout
  Element_t northInputArr[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=northInputArr complete
  Element_t westInputArr[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=westInputArr complete
  Element_t eastInputArr[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=eastInputArr complete
  Element_t southInputArr[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=southInputArr complete

  // Handy conditions
  const bool isSaturated = time > 0 || inputRow > 0;
  const bool onFirstRow = workingRow == 0;
  const bool onLastRow = workingRow == kNX - 1;
  const bool isDraining = time >= kTimestepsPerStage;

  // Read from wavefront
  Burst inputBurst;
  Element_t inputArr[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=inputArr complete
  if (!isDraining) {
    if (in.empty()) {
      return;
    }
    inputBurst = in.read();
    inputBurst >> inputArr;
```

```
  } else if (!onLastRow) {
    return;
  }

  Burst centerBurst; // Need this later for propagating to north buffer

  // North
  if (onFirstRow) {
    // On first row: no northern value
    FillBoundary(northInputArr);
  } else if (isSaturated) {
    northBuffer.read() >> northInputArr;
  }


  // Read buffers
  if (isSaturated) {

    // Center
    centerBurst = centerBuffer.read();

    // West
    westBuffer.read() >> westInputArr;

    // East
    eastBuffer.read() >> eastInputArr;

  }

  // South
  if (onLastRow) {
    // On last row: no southern value
    FillBoundary(southInputArr);
  } else {
    // The wavefront provides the south and final value
    inputBurst >> southInputArr;
  }

  // Output burst fanin
  Element_t outputArr[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=outputArr complete
  Element_t westOutputArr[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=westOutputArr complete
  Element_t eastOutputArr[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=eastOutputArr complete

  // Inter-iteration dependencies
  static Element_t forwardWest(kBoundary);
  static Element_t forwardEast[kElementsPerBurst];
  #pragma HLS ARRAY_PARTITION variable=forwardEast complete

  // Append edge to burst forwarded from previous iteration
  forwardEast[kElementsPerBurst - 1] = lineBurst > 0 ? inputArr[0] : kBoundary;

  // Append edge forwarded from previous iteration to burst
  westOutputArr[0] = forwardWest;

PipelineWidth:
  for (unsigned j = 0; j < kElementsPerBurst; ++j) {
    #pragma HLS UNROLL

    const unsigned col = lineBurst * kElementsPerBurst + j;
```

```
    // Fan out values
    const Element_t north = northInputArr[j];
    const Element_t west  = westInputArr[j];
    const Element_t east  = eastInputArr[j];
    const Element_t south = southInputArr[j];

    // Evaluate stencil
    const Element_t eval = EvaluateStencil(north, west, east, south);

    // Buffer east values from previous iteration
    eastOutputArr[j] = forwardEast[j];

    if (j > 0) {
      // Buffer west values
      westOutputArr[j] = inputArr[j - 1];
      // Forward east values to be buffered by next iteration
      forwardEast[j - 1] = inputArr[j];
    }

    // Fan in evaluated stencil
    outputArr[j] = eval;
  }

  // Forward eastern edge to be the western edge of next iteration
  if (lineBurst == kBurstsPerLine - 1) {
    forwardWest = kBoundary;
  } else {
    forwardWest = inputArr[kElementsPerBurst - 1];
  }

  // Write back buffers
  if (!isDraining) {
    if (!onLastRow) {
      // Overlap to next timestep, values can be discarded
      northBuffer.write(centerBurst);
    }
    centerBuffer.write(inputBurst);
    Burst westOutputBurst(westOutputArr);
    westBuffer.write(westOutputBurst);
  }
  if (i > 0 && i < kTotalBursts * kTimestepsPerStage + 1) {
    // Shift forward by one, as eastern values are buffered by the following
    // pipeline step
    Burst eastOutputBurst(eastOutputArr);
    eastBuffer.write(eastOutputBurst);
  }
  if (isSaturated) {
    // First row of first timestep does not have meaningful buffered values yet,
    // because there has been no overlap from a previous iteration
    Burst outputBurst(outputArr);
    out.write(outputBurst);
  }

  ++i;
  // Process one extra row
  // i = (i + 1) % kTotalIterations;

}

template <unsigned recurse>
void UnrollComputeFour(Stream pipes[kDepth]) {
  #pragma HLS INLINE
  UnrollComputeFour<recurse - 1>(pipes);
```

```
  ComputeFour<recurse - 1>(pipes[recurse - 1], pipes[recurse]);
}

template <>
void UnrollComputeFour<0>(Stream *) {}

void EntryReferenceDesignFour(Stream &input, Stream &output) {

  #pragma HLS INTERFACE axis port=input
  #pragma HLS INTERFACE axis port=output

  #pragma HLS DATAFLOW

  static Stream pipes[kDepth + 1];
  #pragma HLS STREAM variable=pipes depth=kPipeDepth
  static Stream feedback("feedback");
  #pragma HLS STREAM variable=feedback depth=kNX*kBurstsPerLine
  #pragma HLS RESOURCE variable=feedback core=FIFO_BRAM

  ReadFeedbackSingle(input, feedback, pipes[0]);

  UnrollComputeFour<kDepth>(pipes);

  WriteFeedbackSingle(pipes[kDepth], feedback, output);

}
```