On Predictable Reconfigurable System Design

NILS VOSS, Imperial College London, UK and Maxeler Technologies, UK
BASTIAAN KWAADGRAS and OSKAR MENCER, Maxeler Technologies
WAYNE LUK, Imperial College London
GEORGI GAYDADJIEV, Maxeler IoT-Labs and Imperial College London and Univ. of Groningen

We propose a design methodology to facilitate rigorous development of complex applications targeting reconfigurable hardware. Our methodology relies on analytical estimation of system performance and area utilisation for a given specific application and a particular system instance consisting of a controlflow machine working in conjunction with one or more reconfigurable dataflow accelerators. The targeted application is carefully analyzed, and the parts identified for hardware acceleration are reimplemented as a set of representative software models. Next, with the results of the application analysis, a suitable system architecture is devised and its performance is evaluated to determine bottlenecks, allowing predictable design. The architecture is iteratively refined, until the final version satisfying the specification requirements in terms of performance and required hardware area is obtained. We validate the presented methodology using a widely accepted convolutional neural network (VGG-16) and an important HPC application (BQCD). In both cases, our methodology relieved and alleviated all system bottlenecks before the hardware implementation was started. As a result the architectures were implemented first time right, achieving state-of-the-art performance within 15% of our modelling estimations.

CCS Concepts: • **Hardware → Reconfigurable logic applications**; • **Software and its engineering →** *Software development techniques*; *Software development process management*; • **Hardware → Best practices for EDA**; **Hardware accelerators**; **Modeling and parameter extraction**;

Additional Key Words and Phrases: FPGA, reconfigurable systems, analytical model, performance model, architecture, development methodology

ACM Reference format:
Nils Voss, Bastiaan Kwaadgras, Oskar Mencer, Wayne Luk, and Georgi Gaydadjiev. 2021. On Predictable Reconfigurable System Design. *ACM Trans. Archit. Code Optim.* 18, 2, Article 17 (February 2021), 28 pages.
https://doi.org/10.1145/3436995

1 INTRODUCTION

The steadily increasing computation demands of modern applications, the pressure to reduce energy, and the slowdown of semiconductor technology advancements made specialised accelerators

The support of the United Kingdom EPSRC (grant numbers EP/L016796/1, EP/N031768/1, EP/P010040/1, EP/S030069/1, and EP/L00058X/1), Maxeler, Intel, and Xilinx is gratefully acknowledged.
Authors' addresses: N. Voss, Imperial College London, 180 Queen's Gate, London, UK, Maxeler Technologies, 3 Hammersmith Grove, London W6 0ND, UK; email: nv916@ic.ac.uk; B. Kwaadgras and O. Mencer, Maxeler Technologies, 3 Hammersmith Grove, London W6 0ND, UK; emails: {bkwaadgras, oskar}@maxeler.com; W. Luk, Imperial College London, London SW7 2AZ, UK; email: w.luk@imperial.ac.uk; G. Gaydadjiev, Maxeler IoT-Labs, Imperial College London, Univ. of Groningen, 180 Queen's Gate, London SW7 2AZ, UK; email: g.gaydadjiev@rug.nl.

1544-3566/2021/02-ART17
https://doi.org/10.1145/3436995

widely accepted in the computing industry. This trend leads to the emergence of application-specific accelerators (e.g., GPGPUs, Google Tensor Processing Unit [34], IBM TrueNorth [53]) that are starting to see widespread adoption to provide significant advantages in terms of performance and energy usage compared to more traditional Central Processing Units (CPUs).

One particularly relevant class of accelerators is using Field-Programmable Gate Arrays (FPGAs), which have recently seen wide acceptance in production systems by AWS, Microsoft, IBM, Alibaba, and more [2, 3, 45, 55, 87, 88]. FPGAs provide the opportunity to fully customise the accelerator for a specific application. As a result, they promise better performance and energy efficiency compared to CPU and Graphics Processing Units (GPUs) [4, 15, 25, 26, 46] while offering higher flexibility and lower development costs than Application-specific Integrated Circuits (ASICs). However, it is well known that FPGA development is very specialised and hence challenging, especially when maximum performance by fully utilising FPGA capabilities is sought. This often leads to development process uncertainties, unexpected costs, and subpar results.

To overcome the above challenges, various high-level programming frameworks, paradigms, and languages have been proposed. However, one often-overlooked problem is that the aspects of analysis, architecture, implementation, and debugging should be united in a comprehensive design methodology to program FPGA accelerators rigorously.

This article presents a methodology for the development of complex, Field-Programmable Gate Array (FPGA)-based systems based on systematic arrangement of predominantly well-known techniques organised in a way to complement each other. Our methodology is motivated by more than 15 years of deployment of FPGA-based solutions at multiple Tier 1 institutions in different industries like finance [8, 24, 82, 83], as well as oil and gas [50, 75]. To the best of our knowledge, this article is the first to propose such a holistic design methodology. The aim of our methodology is a first-time-right design process, delivering state-of-the-art performance for highly complex, real applications. This means that the hardware implementation is only done once, and all major design improvements are finished before the hardware implementation is started. Our methodology provides independent design guidelines from algorithm and dataset-specific optimisations that steer toward an optimal architecture maximising the utilisation of all system resources while relieving additional possibilities for improvements.

It is important to stress that we do not aim at the highest performance ever achievable; we rather facilitate the best performance for a given set of known optimisations. New optimisations can be considered, and older ones can become obsolete. Nevertheless, our methodology efficacy does not depend on individual optimisations—it streamlines the design process. As such, no changes to the methodology are needed, as the set of considered optimisations evolves. When new techniques arise, an update of the design might still be desired. In such cases, our methodology will assist the designers to rapidly perform precise cost–benefit analysis for each new candidate. Similarly, our methodology can be used for quick evaluation of system characteristics of platform candidates.

Traditional Central Processing Unit (CPU) development methodologies usually follow an incremental approach, whereby a first implementation is gradually improved by using different profiling and debugging tools. However, there are multiple problems with applying this methodology to FPGA-based designs. (Problem 1) FPGA implementation profiling is very hard, making it difficult to identify what specific part of the design needs improvement. This is especially true if the initial implementation cannot entirely fit onto the FPGA fabric or does not reach the frequency required to highlight certain issues, like bandwidth limitations. (Problem 2) Incremental hardware changes can easily require a complete redesign, which might render nearly all previously undertaken development efforts useless. This is especially the case if the fundamental handling and storage of data are reconsidered to work around bandwidth or memory capacity limitations. (Problem 3) The

process to generate an FPGA bitstream[1] is extremely time consuming and can take days. Combined with the high complexity of FPGA programming, this results in prohibitive development times.

It is, however, possible to avoid these problems. Given that FPGA performance is highly predictable, especially for deeply pipelined designs able to hide Input/Output (I/O) latency, a lot of time and developer effort can be saved by performing large parts of the design process *a priori*, even before a single line of hardware code is written. The methodology proposed here structures this process and, to the best of our knowledge, is the best way to design an architecture for reconfigurable systems with predictable performance.

We envision that wide spread adoption of this approach toward FPGA design will improve designer productivity and help build a rich set of applications and libraries, utilising FPGAs. Such libraries will accelerate FPGA adoption especially in High-Performance Computing (HPC).

The main contributions of this article are as follows:

- A comprehensive methodology for rigorous first-time-right development of complex, FPGA-based systems, delivering the predicted performance;
- A careful evaluation of the proposed methodology using two highly relevant applications: a Convolutional Neural Network (CNN) and Quantum Chromodynamics (QCD);
- An overview of previously published successful applications of the methodology and lessons learned based on more than 15 years of industry usage.

The rest of the article is organised as follows. Section 2 provides brief background on FPGAs and programming abstractions. Related work is described in Section 3. Section 4 introduces the methodology. Section 5 discusses the application analysis and the relevant information needed for the acceleration process. In Section 6 the software model purpose along with details on its design are discussed. Section 7 explains the system-level performance estimations. Section 8 describes tradeoffs at the architectural definition phase. The methodology is evaluated in Section 9 with the example of a Convolutional Neural Network (CNN) and a Quantum Chromodynamics (QCD) application and Section 10 concludes the article.

2 BACKGROUND

2.1 Field Programmable Gate Arrays

FPGAs are semiconductor devices able to implement arbitrary hardware circuits while also being reconfigurable. Initially one of the main use cases for FPGAs was the emulation and verification of Application-Specific Integrated Circuits (ASICs). Since FPGAs can implement the same circuits as ASICs, it is possible to completely test and verify the design, which results in significantly reduced design turn around times. Both tape out and complicated technology mapping are not required, making designing cheaper and faster.

FPGAs quickly expanded to other use cases, most notably signal processing and, recently HPC. The main driver is the massive inherent parallelism of FPGAs, combined with the ability to implement customised architectures for the intended use case. Compared to Graphics Processing Units (GPUs) and CPUs, FPGAs provide higher level of parallelism while operate at an order of magnitude lower clock frequencies, which results in often higher or at least comparable application-level performance with lower energy usage. Compared to ASICs, performance and energy efficiency are often worse for the same technology node; however, ASIC production costs are often prohibitive.

FPGAs typically use a column-based architecture. Each column includes different hardware resources. For most contemporary FPGAs these are multipliers (DSPs), on-chip memories, and logic

[1]The configuration file loaded onto the FPGA.

resources consisting of look-up tables and registers. Dedicated routing resources are used to interconnect the on-chip logic, arithmetic and memory modules.

To map the hardware description of the intended design onto the FPGA fabric, place and route has to be performed. Since place and route is NP-complete [61], it is very time consuming and can take days for practically interesting implementations targeting large FPGAs.

2.2 FPGA Programming Abstractions

The traditional hardware design, also for FPGAs, is based on Hardward-Description Languages (HDLs). These languages operate on the Register-transfer level (RTL). A circuit is described as registers and the combinational logic between these registers. As such, the Register-transfer-level (RTL) provides a higher abstraction level compared to the logic gate or even more detailed transistor level. Consequently the usage of HDL delivered a significant improvement in design productivity compared to the previous description of circuits on an even lower level. However, Hardware-description Languages (HDLs) still provide a significantly lower abstraction level than software languages limiting designer productivity.

Since the design in HDLs can result in fast and efficient hardware implementations, but requires considerable skill and effort combined with low productivity, a wide range of tools to mitigate these disadvantages have been developed. Most of them focus on raising the productivity of FPGA design. A typical approach to boost productivity is by reusing Intellectual Property (IP) blocks. Another possibility is to automatically generate FPGA designs from domain-specific tools such as Matlab Simulink or LabView but this is naturally limited to certain application types. It has also been proposed to increase productivity by using overlay architectures, e.g., Reference [64]. These typically provide a number of customisable templates that can be quickly programmed, providing a compromise in terms of efficiency, performance, and development time.

To improve designer productivity for generic designs, High-level Synthesis (HLS) tools have been developed. These typically attempt to create FPGA designs from conventional programming languages such as C and often require some form of manual intervention in the transformation process. These tools have seen a rapid adoption across industry and academia in recent years. However, while the productivity advancements are undeniable, there is still an ongoing debate on the quality of results compared to HDL designs. Additionally, many design decisions, normally performed by the programmer, are performed automatically or not implemented within the High-level Synthesis (HLS) tool. Our methodology relies on control over the architecture not provided by most HLS tools.

Another commonly used abstraction to simplify programming of hardware accelerators is the employment of dataflow-oriented paradigms. This computational model is an evolution of dataflow and systolic array concepts [20, 43]. Developers describe large dataflow pipelines with massive internal throughput using a high-level language. This results in application-specific, statically interconnected compute structures in the form of dataflow graphs that can be easily optimised and mapped to FPGAs. Dataflow graphs contain nodes representing operations and edges that symbolise the flow of data between these operations. Scheduling of these graphs is fully automated, avoiding error-prone manual scheduling. Comparison to VHDL is reported in Reference [71].

Additionally, it is possible to treat a dataflow graph as a single unit of computation, often called a kernel. These kernels can be fully synchronous themselves and interact with other kernels or components like memory controllers asynchronously through properly sized First In, First Out (FIFO) buffers. As such control signals as well as stall handling can be automatically generated.

In contrast to HLS, this abstraction still leaves important architecture and design decisions to the designer. For this reason, we employ the dataflow abstraction throughout this article to find a balance between ease of programming and fine-grained control. The principles described here

are applicable to RTL and HLS. In the latter case, it needs to be possible to influence the HLS tools enough to implement the architecture developed using the methodology.

3 RELATED WORK

The amount of related work regarding tools, toolchains and or programming languages is significant [1, 6, 10, 22, 52, 62, 63, 74, 76]. This work is fundamentally different in that it looks at how complex FPGA-based applications can be programmed on a methodology level and does not focus on tools.

The topic of hardware software partitioning has been covered extensively in the literature. Reference [32] provides an overview of the research accomplished so far. Reference [70] presents a typical example of this work, which uses a heuristic (genetic algorithms in this case) to find an optimal split between software and hardware components. The previous research has mostly focused on embedded systems, which means that metrics and assumptions are used that do not hold true for the High-Performance Computing (HPC) use case. For example, the authors often assume a shared memory architecture and no large dedicated memory on the hardware device. Additionally, the number of devices in the system and interconnect types are usually limited. The metrics optimised are usually runtime, area usage and energy usage, whereas in the HPC use case the required communication is also of major concern.

Similarly, there is a wide variety of tools to predict performance and area usage. For example, Reference [30] proposed a methodology based on analytic equations focusing on host to accelerator communication and computation time. However, the accuracy of the computational model is limited; they do not include resource usage predictions and require a specific architecture between the host and the accelerator. Overall, the prediction error of the model is up to 40% in some of the use cases.

In Reference [67], a framework for the performance and power prediction of standard cell-based accelerators is presented. The tool uses the intermediate representation generated by a just-in-time compiler to capture the algorithm described in a C program. The tool automatically optimises this graph and analyses data dependencies to capture an accurate model of the accelerator. Additionally, the framework includes memory and cache simulators. It is not possible to influence the optimisations performed manually or describe a target architecture. Instead the C algorithm has to be changed and the framework performs Design Space Exploration (DSE) for a set of parameters like loop unrolling factors and frequencies. The reported error for the tested benchmarks is less than 1% for the execution time, 5% for the power prediction, and 6.5% for area prediction.

In Reference [49] a similar technique is used to estimate the resource usage for HLS tools. The input is C/C++ code with a selected number of HLS pragmas. The application trace and analytical equations are used to predict the area usage with an error of less than 10%. However, only single precision floating-point arithmetic is supported and on-chip port widths and on-chip port widths and communication are not modeled.

The tool described in Reference [92] uses a similar approach but supports more pragmas and more detailed analytical equations and can automatically perform Design Space Exploration (DSE). However, the logic usage and communication overhead are ignored, and it is highly dependent on the specific HLS tool and version used.

The work presented in Reference [93] also considers double data rate synchronous dynamic random access (DDR) communication and uses a Machine Learning (ML) technique for prediction. However, the prediction error is still large and the applicability is limited to a few applications fulfilling specific loop and tiling requirements.

In Reference [17], the normal Roofline model [84] is extended to also support FPGAs. This is achieved by, for example, considering area requirements. The model assumes that FPGA

implementations always make use of processing elements. The area usage and performance for one processing element are measured by performing place and route. Linear interpolation is used to determine the maximum number of processing elements and resulting performance. The authors note that linear scaling is a simplification. Additionally, the communication bandwidth is measured in bytes per second. The result is used to determine the maximum achievable performance on a given platform. The same authors test their model on more use cases and with a different HLS tool in Reference [18].

To summarise, the existing tools either lack accuracy, are dependent on specific optimisations, architectures or tools or do not allow the designer to gain insight of *what* limits the performance.

Fewer papers focus on the development methodology of FPGA applications. Reference [79] and Reference [80] are examples of methodologies based around HLS tools, in this case OpenCL, which are similar to CPU-based methodologies, where an application is incrementally improved. Additionally, in Reference [56] a methodology based on incremental refinement with a brute-force approach to design space exploration is presented. All these approaches might be feasible for small applications. For bigger applications, place and route time as well as the risk that a rewrite of the complete application is needed, based on bottlenecks discovered late in the design process, will prove prohibitive.

Reference [65] presents a top-down methodology for hardware design using VHDL. A specification is implemented in VHDL and then synthesised, allowing for full simulation and verification from the VHDL level downwards. The authors of Reference [44] describe how agile methodologies can be applied to hardware design by splitting work packets into small tasks, and in Reference [72] a methodology around software hardware partitioning based on the manual evaluation of design guidelines is presented. In Reference [35], the authors describe a methodology for the design of electrical control systems that tries to split complex algorithms into smaller modules to ease implementation and improve reusability. The contributions in Reference [57] provide a great overview on the ways in which SystemC can be used to accurately model an algorithm in the context of hardware design. While our methodology reuses parts of these contributions we refocus them on the FPGA design of complex applications for HPC.

Some contributions work toward automating the complete process. In Reference [58], the roofline model is used to automatically explore the design space and implement the pareto-optimal implementation. Similarly, Reference [40] automates the complete design process, including design space exploration using a performance model consisting of analytic equations and machine learning models. In both cases, the authors assume an architecture in which all initial data as well as results are copied into DDR. No direct communication between the FPGA fabric and other components is possible. The authors only use small benchmarks to validate their work and even for those simple examples the speedup over CPUs is often limited.

In contrast, there exists a wide array of work on programming more conventional systems. These range from parallel programming methodologies for single parallel processors [51] to data warehouse systems [7] and especially data-intensive applications [39]. We try to contribute toward similar methodologies and best practices incorporating FPGA-based machines.

4 METHODOLOGY OVERVIEW

The proposed methodology consists of four main parts:

(1) accurate analysis of the targeted application;
(2) design of a representative software model;
(3) modelling of architectural candidates; and
(4) rigorous development of an application-specific architecture based on the results of the above.

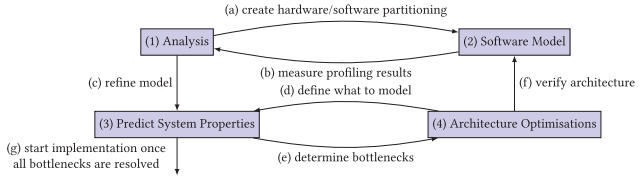

Fig. 1. Design methodology breakdown and its four parts and seven steps.

Figure 1 shows how these four parts interact with each other. The individual steps are explained in the corresponding sections. The methodology involves an iterative process, which requires incremental refinements until the final result cannot be further improved. We start with the analysis of the original application (Part 1). Based on the analysis an initial partitioning in hardware and software components is performed (Step a). The parts selected for hardware implementation are modelled in software (Part 2). Additionally, an initial architecture is designed (Part 4) that informs the representative performance model (Step d, Part 3). The performance model enables prediction of the achievable performance as well as area and bandwidth requirements. It relies on the analysis of the application (Step c), which is further improved by making additional measurements of, for example, data movements and numerical properties. These are hard to obtain from the original application, so the software model (Step b) is used instead. The performance model is used to refine the architecture by identifying bottlenecks, e.g., bandwidth limitations that can be addressed by architectural changes (Step e). However, architectural modifications determine the content of the model, since the performance model is obtained based on a given architecture (Step d). Finally, the software model is used to verify algorithmic and numerical changes of the architecture (Step f).

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

ALGORITHM 1: Iterative refinement using the methodology (Numbers and letters in brackets do not describe specific transition but only refer to the different parts and steps described above).

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

As a result, the analysis (Part 1) combined with the software (Part 2) and the performance model (Part 3) erases the need to profile the FPGA implementation (Problem 1), since bottlenecks are discovered beforehand. Needs to perform redesigns late in the design process (Problem 2) will be avoided, since the architecture (Part 4) is based on the performance model (Part 3), moving this

issue to design time. This, combined with the availability of functional simulators for verification, also removes the need to perform place and route for multiple different design points (Problem 3).

While the analysis of the original application is usually only application dependent, all other steps might become device dependent. For example, the compute and communication capabilities of specific accelerators might be different and, as a result, the developed architecture might need to circumvent different bottlenecks. However, in general, the same performance model and architecture can be reused for different devices as long as they have similar compute and communication capabilities. In other cases it is usually possible to reuse most of the work leading to the system design for the initial target platform. Even though this article covers the case where the target system is already selected, it is also possible to identify a system for optimal performance or energy efficiency by adjusting the performance model and the architecture for all target systems under consideration. Similarly, it is possible to quickly evaluate the performance impact of a new device on the overall system and the changes required to the architecture to achieve this.

The methodology is based on a static dataflow model, to ease prediction. However, it should be possible to generalise the methodology even further, as long as accurate system prediction is possible. Similarly, the toolchain used for programming the target device has to provide enough fine-grained control over the hardware to the programmer to allow for an accurate implementation of the developed architecture. This means, for example, that Sandpiper, which was recently announced by Microsoft, and promises predictable performance as well as explicit control of parallelism and memory access [11], should be fully compatible with the methodology. Even though this article will focus on FPGAs, it is also applicable to ASICs and supports highly heterogeneous systems with multiple, potentially different, accelerators.

Full automation is not intended by the methodology described here; it is overall human-centric. This is intended and not a result of limited effort or time. The main motivation for automation of certain parts of the development methodology would be to save development time and to reduce the knowledge and experience required from the engineers implementing the design. While these are both definitely worthwhile targets we believe they do not fit well into our main goal. We aim at complex applications targeting highly demanding industrial and scientific environments. This requires transformation of the unique application properties within tight performance targets. The design space is highly complex and automated exploration will typically introduce significant performance penalties. In many cases certain performance needs to be reached within tight space, time, power, and cost limits. For example, weather forecasting for the next day has to be ready before the start of the next day to be of practical use. For such systems, the overall costs and all additional performance benefits resulting in slightly better results or reduced running costs justify the additional investment into the application engineering. Additionally, knowledge and experience shortcomings can be partly mitigated by better technical documentation and example designs.

Current automated solutions often fall short of achieving high-performance targets comparable to handwritten code, and at that point the designers have very limited options for improvement. For example, the fully automated development flow presented in Reference [40] implemented single precision Genernal Matrix Multiply (GEMM) on a Maxeler MAX4C Dataflow Engine (DFE) and achieved a speedup of 0.1× over a six-core CPU. In comparison, Maxeler demonstrated more than 20× speedup for double precision General Matrix Multiply (GEMM) using a previous version of the methodology presented here targeting the same FPGA hardware compared to a single core of a roughly 10% slower six-core CPU [33]. The roughly two orders of magnitude performance difference shows the potential of our manual approach when compared to a fully automated solution.

We argue that many problems in the design of complex FPGA applications require human insights, and as such the main goal is to expose the most relevant information to the designers

performing the task and to facilitate them in making the best decisions. As such, we attempt providing relevant insights and are not aiming at reducing the amount of work. Future work might be to automate specific steps in the process while maintaining the amount of insight gained by the engineer and closing the gap between automated and manual processes to change the tradeoffs described above.

5 APPLICATION ANALYSIS

The first step (Part 1), depicted in Figure 1, analyses the original CPU application to identify its compute and data-intensive parts and gain understanding of the expected architectural challenges.

In classical general purpose computing, the view on the performance characteristics is often limited to analysing instructions and their order of execution, as well as the assignment to execution units. This view, however, is not optimal for applications on massively parallel architectures like FPGAs. Instead, it is crucial to understand the required data movements in the system and to develop a strategy which optimises data placement and movement at all levels of the system. Only then is it possible to make maximal use of the available computing resources by providing undelayed access to the working dataset to mitigate limited I/O bandwidths.

The target of the control/dataflow analysis is twofold. First, this step will help to make an initial assessment of which parts of the application should be ported to FPGAs, since in most cases it is not feasible and/or desirable to port the entire application. However, following Gustafson's Law [27] the designer needs to be careful to address a significant part of the execution time, to make sure that the code remaining on the CPU will not dominate the runtime. For example, if the part of the application that contributes 90% of the execution time is ported onto the FPGA the maximum theoretical achievable speedup is 10×, due to the remaining CPU parts of the application.

The second target is to collect all the information needed to model the performance of the accelerated application and consequently make correct design decisions. This includes the amount of data that are needed and when they are needed as well as the operation execution order.

All the widely used methods for static and dynamic code analysis are applicable here. The major outcomes of this effort are application-specific operation counts, data movements and sizes and predictable memory access and communication patterns.

Loopflow Graphs. Counting operations for a given set of functions is typically easier than analysing, representing and understanding data movements. For this reason, we propose the loopflow graph, which is an extension of the control dataflow graph [54] and will help to visualise some of the most important results of the previous analysis. Loopflow graphs focus on the loops in the program and how they interact with each other. Since in general most of the execution time is spent in loops this provides a good level of abstraction. Each loop is represented by a rectangle and the number of nested loops is annotated by a factor. Additionally, the operation count can be annotated inside each rectangle.

The dataflow between loops is shown by directed arrows. Arrow widths hint transferred data amounts. Consequently, the computational and data requirements can be easily visualised together, which helps to identify the portions of the code that should be moved to the FPGA (Step a).

The loopflow graph for the VGG-16 CNN [69] in Figure 2, provides a good example on how this graph can help to perform the code split between CPU and FPGA. One can easily spot that the operations needed to compute the fully connected layers, depicted in the box at the bottom, are two orders of magnitude less than for the convolutional layers, depicted by the remaining boxes. Additionally, only a tiny amount of data need to be transferred between the convolutional and the fully connected part of the network. As such, splitting between the convolutional and the fully connected part of the network becomes the obvious choice.

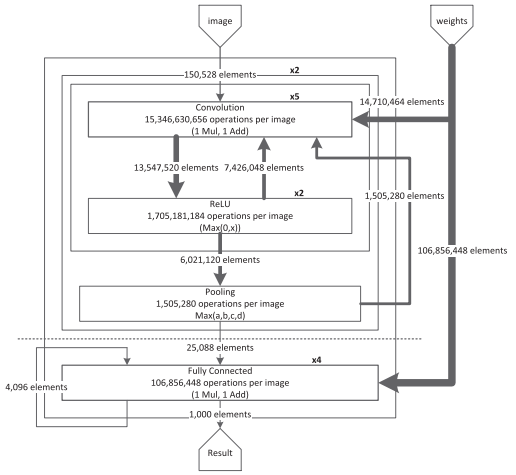

Fig. 2. VGG-16 CNN Loopflow graph. Boxes represent loops and their internal operation types and counts.

6 SOFTWARE MODEL

The software model (Part 2) is a simplified software implementation of the parts of the application that are to be ported onto the FPGA. It is not intended to be optimised for speed, but instead it should be an accurate representation of the intended FPGA implementation. This means that it should be as close as possible and necessary to the planned FPGA architecture and use the same Application Programming Interface (API). For example, it should mirror changes to the algorithm, to verify correct functionality. Usually it is written in C or C++. It serves three different purposes.

(1) Insight into the selected code and easier measurement of profiling results (Step b);
(2) Testbed for verifying numerical and algorithmic changes (Step f); and
(3) Debugging reference for FPGA implementation (Step f).

To achieve the first goal of the software model, the algorithm can be implemented as a simple, not optimised code, helping with the application analysis, and as a result the lessons learned from this implementation are used to refine the performance model and architecture. In the software one would not profile runtimes, but, for example, memory access patterns or the number of loop iterations executed. Similarly, the software model will be used to quickly evaluate different algorithmic and numerical options, which then feeds back into the architecture and performance prediction. As a result, the software model is typically co-developed with both.

It is important to integrate the software model back into the original application. This ensures that the planned Application Programming Interface (API) between the original application and the accelerated modules is sufficient to recreate the original functionality. Additionally, it enables the verification of the software model by comparing the results between the original software and the application using the software model. It has to be noted that due to the fact that arithmetic using floating point numbers is not associative small differences in the result are expected at this stage.

Using the same API for the software model as for the FPGA implementation also creates an easy-to-use debugging tool for the FPGA implementation. The reason to not only verify against the original application is that the software model is supposed to undergo the same numerical and algorithmic changes as the FPGA implementation. As such, it is easy to check if an unexpected result for a given input dataset is due to an error in the FPGA design, side-effects, or fundamental problems with the selected algorithm or its numerical properties. Numerical and algorithmic problems are

significantly easier to resolve in the software model than in an FPGA implementation. Additionally, this provides an FPGA mock-up implementation for earlier integration into the host application.

While the creation of the software model and especially its continuous improvement during the design process represents an overhead in terms of design effort, it avoids the need to make these design iterations in hardware. It is widely accepted that software development, especially if code is not optimised for performance, is easier than hardware development resulting in time and cost savings compared to performing these design iterations in hardware. Additionally, the software model can be developed by software engineers and does not require hardware expert knowledge.

Numerical Analysis and Simulation. One of the most crucial and often also hardest steps in the application porting process is the change from floating-point to custom fixed-point arithmetic. This is due to the more restricted dynamic range of fixed point number representations, which are frozen at compile time. As such the numerical properties of the algorithm should be well understood. This is especially challenging for operations with significant impact on the value range, e.g., all multiplications.

However, to maximise the available hardware resources' utilisation this is necessary. The above comes from the fact that a floating-point addition needs roughly one order of magnitude more logic resources than its fixed-point counterpart. The reason for this is that exponent alignment requires expensive barrel-shifters before and after each operation. Fixed-point implementations will typically facilitate significantly larger number of operations on the FPGA. However, for some I/O bandwidth constrained designs with no option for resolving this bottleneck, floating-point precision might still suffice. The performance model helps to determine this.

The first step toward a fixed-point implementation is understanding of the numerical properties. This can be facilitated by tools collecting data during the software execution. The easiest way to integrate such tools is by instrumenting the software model that should use the same numeric representation as the final design. The selection of a representative input dataset is crucial.

Based on the results of the numerical analysis the software model is adjusted to use the most suitable fixed point sizes. The functional correctness is investigated by a bit-level accurate fixed point simulation. To repeat, the selected minimal data width has the potential of reducing resource usage. Verification is done by comparison to the output of the original software using a representative dataset. The problem of floating-point to fixed point conversion is widely covered and there are various tools addressing this problem [9, 12, 13, 16, 36, 37, 38, 68]. Performing the fixed point conversion at early design stages enables much larger acceleration structures, improves performance model precision and avoids complex debugging of numerical problems in hardware.

7 PREDICTING SYSTEM PROPERTIES

With the code analysis results (Part 1) and the initial software model (Part 2), a representative performance model (Part 3) capturing the planned implementation (Part 4) on the selected platform can be built. To achieve this, the hardware and bandwidth usage, along with execution time for a problem with known size, are predicted. A preliminary speed-up estimation is obtained.

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

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

greatly limit the ability to predict the performance of a given application. Even though fine grain models and simulators exist to simulate these side effects, their accuracy is often still limited [31, 66].

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

If the workload is sufficiently large, which is normally one of the prerequisites for FPGA acceleration, then the execution time is supposed to dominate over the initialisation time so that it can be safely ignored. The precise initialisation time depends on the used platform and framework. For example, the initialisation time of a Maxeler system is usually between 1 and 100 ms. Additionally it is possible to take the time for reconfiguration into account.

In a streaming dataflow design, execution time is the maximum of the time needed for the computations (T_{comp}, Section 7.2), the time to transfer the data between host and FPGA (T_{comm}, Section 7.3) and the time required to transport data between FPGA and board memory (T_{mem}, Section 7.4). The longest latency dominates the overall execution time and is the primary focus.

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

7.1 Predicting Area Usage

To obtain the computational time requirements one first determines the degree of parallelism achievable on a given device. The main limits are the available hardware FPGA fabric resources.

In general, the FPGA hardware resources are used for three different purposes. First, to implement the arithmetic operations, second the scheduling of operations and finally other IP modules, e.g., the Peripheral Component Interconnect Express (PCIe), memory controllers, and so on.

To predict the area usage for arithmetic operations it is first necessary to find out the amount of hardware resources required to implement a simple operation on the target FPGA device. Those figures can be determined either by using automated tools, finding appropriate tool and FPGA vendor documentation or by creating micro-applications and running them through the vendor tools. The overall area usage can be roughly estimated as the area cost of a single operation multiplied by the number of operations to be implemented on the fabric. The scheduling relies on First In, First Outs (FIFOs), usually implemented in on-chip memory or using registers.

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

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

(1) in operations and IP-blocks, e.g., double data rate synchronous dynamic random-access (DDR) memory, FPGA to FPGA communication or Peripheral Component Interconnect Express (PCIe);
(2) for on chip buffering or reordering of data; or
(3) for balancing of the pipeline of computational structures.

Predicting the area usage for the first case is very straightforward, since it can be estimated with micro benchmarks. The same can be done for the second case; however, it is also possible to model it using Equation (1), where the number of memory blocks required (n_{mem}) is calculated as the

product of the width (w) required divided by the native width of the on chip memory read and write ports, the depth (d) required divided by the possible depth of the hardware unit and the number of read ports (p) required divided by the number of read ports present in the hardware,

$$n_{mem} = \left\lceil \frac{w_{req}}{w_{hardware}} \right\rceil \times \left\lceil \frac{d_{req}}{d_{hardware}} \right\rceil \times \left\lceil \frac{p_{req}}{p_{hardware}} \right\rceil. \quad (1)$$

The on chip memory resources can often be used in different aspect ratios. It is possible to tile logical memories to fit into these physical on chip memories of different size. The total memory hardware costs is the sum of the costs for each individual tile.

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

7.2 Predicting the Compute Performance

Next the achievable parallelism and the number of data items processable per clock cycle can be predicted while ignoring bandwidth limitations. The time needed to compute a set of data items can be calculated as shown in Equation (2) by dividing the number of items that need processing by the product of the targeted frequency and the number of items processed per cycle,

$$T_{comp} = \frac{n_{total}}{n_{cycle} \times f}. \quad (2)$$

Frequency prediction cannot be precise. However, with sufficiently deep pipelining and if chip resources are used less than 80%, the frequency on the same FPGA can be safely estimated based on previous design experience or experiments using artificial designs to fill up the chip. For example, Maxeler cards with Altera Stratix V FPGAs usually achieve 200 MHz if the chip is filled up to 80%.

7.3 Predicting I/O Bandwidth Usage

The bandwidth between the host and the FPGA depends on the physical interconnect used. For most acceleration platforms, the interconnect is PCIe. PCIe is built by bidirectional links, meaning that the full bandwidth can be used in both read and write direction at the same time. For example, second-generation PCIe can transport up to 4 GB/s over an eight-lane link. As a result, the communication time between host and accelerator can be calculated as the maximum data size transferred in either direction (S_{in} and S_{out}) divided by the bandwidth (BW), as in Equation (3),

$$T_{comm} = \frac{max(S_{in}, S_{out})}{BW}, \quad (3)$$

$$T_{mem} = \frac{S_{write} + S_{read}}{BW \times efficiency}. \quad (4)$$

7.4 Modeling on Board Memory Behaviour

FPGA accelerators on board memory is typically DDR. Bandwidths of this memory are unidirectional, e.g., bandwidths in both directions (read and write) are shared. As such, the time to transfer data between on board memory and the FPGA can be calculated as in Equation (4).

The parameter *efficiency* represents the proportion of the theoretical DDR bandwidth achievable for a given dataset. Only large linear access patterns allow efficient memory access. Figure 3 shows the achievable memory efficiency based on the amount of linearly accessed data for DDR4 memory as used on an FPGA card. Different DDR-based memory technologies behave in a similar manner.

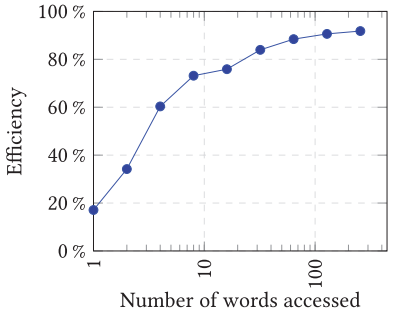

Fig. 3. FPGA card DDR4 memory efficiency.

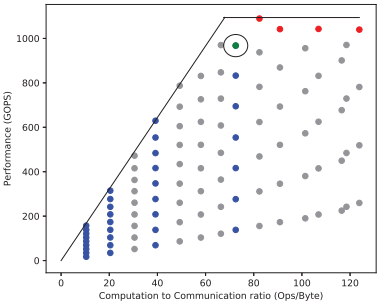

Fig. 4. VGG-16 architecture roofline model.

7.5 Comparison to the Roofline Model

The roofline model [84] is a commonly used tool to estimate system performance, perform design space exploration and identify optimal architectures. It was initially developed for floating point programs executed on multicore architectures and is based around *operational intensity*, which is a measure describing the number of computations per byte accessed from memory. The roofline model is a two-dimensional chart, where both axes are logarithmic. The x-axis in this chart is the computational intensity, which represents the number of operations executed on each byte accessed from off-chip memory. The y-axis represents the performance in floating point operations per second.

The roofline model was expanded to be also applicable to FPGAs in Reference [18] and Reference [17]. Figure 4 shows an example roofline model for a VGG-16 CNN, which will be discussed in more detail in Section 9.1.6. It can be used to quickly provide an overview and a visualisation of the communication to computation ratio of an architecture and the resulting performance impact.

However, for the case of FPGAs this model has clear shortcomings. First and foremost, the model can only focus on one Input/Output (I/O) bandwidth at a time. In reality, systems are not only limited by external memory but also host communication bandwidth. Furthermore, some systems might have heterogeneous memory architectures, e.g., HBM and DDR, or accelerator to accelerator communication. Integrating those into the simplified roofline model is often not achievable.

Additionally factors like on-chip memory capacity are not considered and would need to be addressed separately. Another problem is that for FPGA-based systems the computational roofline inherently includes the frequency the computation is run at. However, the achievable frequency is highly dependent on the routing complexity and the overall resource usage. Finally the roofline model is based on the assumption that the overall system performance is constrained by one computational kernel focused on a specific type of computation. Many applications, especially highly complex applications, benefit heavily from implementing multiple computational kernels on the same accelerator. These computational kernels might even operate on different datatypes. In those cases, it will prove to be problematic to determine consistent computational ceilings.

8 ARCHITECTURAL OPTIMISATIONS

The details of developing an efficient architecture (Part 4) go beyond the scope of this article. However, some general concepts should be explained here, since only the development of a good architecture leads to good results. In general, the target is a balance, where all system resources are fully and equally used, and all bottlenecks of the system as indicated by the performance model (Step e) are removed (Step g). If, for example, all the memory bandwidth is used but only 50% of the

chip resources, then a better architecture should save memory bandwidth. This will increase the degree of parallelism, using more of the chip resources and reducing the overall execution time.

Improving Bandwidth Utilisation of memory-to-accelerator and host-to-accelerator interfaces are very similar. There are three main strategies to achieve this. First, one can buffer more data on chip, reducing the need for streaming it back and forth. Second, customised compression can be used. This could be as simple as changing the data width (e.g., from 32-bit to 12-bit integers). However, more advanced compression algorithms, e.g., run length encoding, can also be a good choice. Third, data sequences with predictable values can be generated on the FPGA, avoiding transmission altogether. In the case of on board memory, it is often also possible to increase memory efficiency, e.g., by reordering data on the chip just before it is written or after it is read to create linear access patterns. Reordering operations to reduce the amount of data that need to be moved is also an option. Additionally, it is possible to optimise the access into the different ranks of the DDR memory, as discussed in Section 7.4, by using an optimised memory layout.

Reducing Area Usage is usually the most important step. To make efficient use of the available hardware resources data types and widths are carefully selected. Floating point arithmetic is typically very inefficient, but even in fixed point, smaller data widths use fewer resources. Otherwise it is possible to replace area expensive operations with cheaper ones. One option to achieve this is by considering alternative implementations of a given algorithm. Even if those implementations are less efficient on a CPU, they might be cheaper to implement in hardware. For example, sorting networks are often used in hardware, while on CPUs other algorithms, e.g., merge sort, are usually preferred. Moreover, one could consider to move calculations to the CPU or precompute some values, if possible. The on chip memory usage can easily be reduced by moving buffers into on board memory. If double buffering[2] is used, then it is sometimes possible to recreate the same functionality and read-write speed with only one memory by developing a custom addressing logic. This custom addressing logic has to ensure that only elements that are already read are overwritten and usually involves switching between multiple complex addressing patterns [77].

Overlapping Host and Accelerator Execution: Real applications cannot be fully ported to FPGAs, typically all control heavy code remains on CPUs. The simplest way of integrating the FPGA functionality into the CPU code is a blocking function call. However, this uses resources inefficiently as only one system component is used at any time. A better way to handle this is to overlap the execution. One way to achieve this is by calling the FPGA in a separate thread. When CPU and FPGA calculations are independent parallelisation is easy. Otherwise, it is recommended to have the CPU and the FPGA work either on different parts of the dataset or on disjoint tasks.

9 EVALUATION

To evaluate our methodology we will apply it to multiple real-world applications targeting real problems in science and engineering. Simple benchmarks would not be sufficient to evaluate system-level performance effectively let alone predicting application performance as is exemplified by the discussions on the widely used LINPACK benchmark [21, 23, 42]. The authors of Reference [14] show that simple synthetic benchmarks provide only very poor indications of application performance and only the combination with significant application analysis will lead to meaningful predictions. Similarly, the authors of Reference [73] use genetic algorithms for predicting application performance based on memory bandwidth. However, this requires *a priori* knowledge about the specific bottleneck of the system under study, which might be possible for von Neumann architectures, but does not map well to FPGAs with their very high degree of freedom in terms of system architecture design. As such, we only select applications that required multiple man-months and in

[2] A technique with two buffer copies to resolve write before read dependencies. One is written, while the other is read.

some cases even man-years of development effort to ensure sufficient complexity to draw meaningful conclusions. This, for example, includes the communication between multiple "computational kernels," which each would normally be a full benchmark in itself and realistic, application-specific data transfers between hosts and accelerators. Only in this way we can validate that predicting the system properties correctly highlights potential system bottlenecks that can be alleviated by developing a suitable architecture. This is a major difference between our approach and previous work, which often focuses on a few oversimplified (predominantly synthetic) examples.

In this section, the proposed methodology is evaluated using two complex use cases. One is the acceleration of the inference of the VGG-16 [69] CNN discussed in Section 9.1 and the second is the acceleration of BQCD presented in Section 9.2. In both cases, we used Maxeler's MaxCompiler version 2017.1 and Vivado 2017.1. We target Maxeler's MAX5C DFE as our FPGA platform. The compute device of the MAX5C DFE is the Xilinx VU9P FPGA, which consists of 1,182,240 LUTs, 2,364,480 FFs, 6,840 DSPs, 4,320 BRAMs, and 960 URAMs. Additionally, the card has three 16-GB DDR4 Dual In-Line Memory Modules (DIMMs), which provide a theoretical peak bandwidth of 15 GB/s each.

To support our claims, we briefly discuss three additional published examples that used our methodology. Finally, we share some experiences gained from applying and refining the methodology to a wide range of industrial and academic use cases for more than 15 years.

9.1 Convolutional Neural Network

The details of the analysis (Part 1) and software model (Part 2) for the CNN are not described here, since their results are already encapsulated in Figure 2. The loopflow graph also motivates the split of the application (Step a). In this case, the convolutional layers are ported onto the FPGA, while the fully connected layers stay on the CPU.

9.1.1 Architecture. In this section, we first define an initial architecture (Part 4) to determine what to model (Step d) in the performance model (Part 3). There are two fundamentally different options to implement a CNN on FPGAs. The first option is the implementation of a fully streaming architecture, in which hardware is dedicated to each layer separately and all layers are computed in parallel. However, this architecture often requires a prohibitive amount of on chip memory, since on board memory does not provide the required bandwidth. The second option is the usage of a generic hardware unit, often called a Processing Element (PE), which is able to perform the necessary computations for all layers. In this case, only parallelism within a layer is exploited.

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

As a result, it is not feasible to implement the fully streaming architecture (Step e). Instead the PE-based architecture shown in Figure 5 is selected. Each PE performs the convolution operations, accumulation over input channels and the Rectifier Linear Unit (ReLU) activation functions [89]. To save on board memory bandwidth, all PEs operate on the same input channel producing different output channels, but the units performing pooling and writing data to on board memory are shared.

9.1.2 Modelling Host to Accelerator Communication. First the I/O communication time will be predicted, using Equation (3). For this the amount of data needed to be transferred to and from the FPGA would need to be determined. This consists of the weight and input image data.

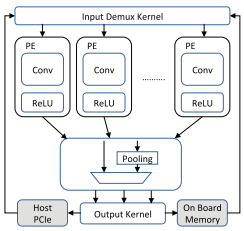

Fig. 5. Convolution architecture.

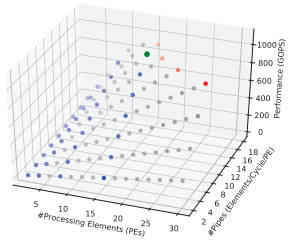

Fig. 6. Design Space for VGG-16 architecture.

Weights are constant across iterations and are transmitted once. As such, they can be neglected, since the one-time transmission is of no consideration as soon as thousands of images are processed.

In cases when it is not possible to compute all output channels of the first layer in parallel, it is necessary to retransmit the input or buffer it in memory. Equation (5) shows the case when data are retransmitted. n_{out} is the number of output channels of the first layer, n_{PE} the number of PEs and n the number of images transferred. Equation (5) becomes Equation (6) to estimate the number of processed items within a given time. To obtain the number of input items per second, T_{comm} is set to 1 s,

$$T_{comm} = \frac{n \times max(S_{in} \times \frac{n_{out,L1}}{n_{PE}}, S_{out})}{BW}, \quad (5) \qquad n = min\left(\frac{BW \times T_{comm}}{S_{in} \times \frac{n_{out,L1}}{n_{PE}}}, \frac{BW \times T_{comm}}{S_{out}}\right). \quad (6)$$

9.1.3 Predicting Accelerator Computational Latency. Similarly, the accelerator runtime can be estimated as shown in Equation (7), where n again denotes the number of input images, OPS_{needed} the number of operations performed per input image and $OPS_{available}$ the number of operations that can be performed on the FPGA for the developed architecture within a given amount of time. The maximal number of PEs, as limited by the on chip compute capabilities, is given by Equation (8),

$$T_{comp} = \frac{n \times OPS_{needed}}{OPS_{available}}, \quad (7) \qquad n = \frac{OPS_{available} \times T_{comp}}{OPS_{need}}. \quad (8)$$

Convolutional layers mainly consist of Muliply Accumulate (MAC) operations; hence, the total number of Multiply Accumulate (MAC) operations and the FPGA capacity are needed to estimate T_{comp}. This assumes the implementation of the Rectifier Linear Unit (ReLU) and pooling functions inside the PEs as proposed in this architecture.

9.1.4 Estimating on Chip Memory Usage. Depending on the on chip memory capacity, it might be feasible to buffer on chip all input and output channels for all layers. However, on smaller chips, or for layers with either more or larger output channels, this is not feasible. Similarly, networks like ResNet-101 or other residual networks reuse data from earlier layers [28] and therefore require even more buffer space. In those cases it is necessary to store the results produced by the computation of each layer in on-board memory. Only this case will be modelled here. Each PE works on a separate output channel; hence, all the data produced by PEs should be written back to memory.

Based on Equation (1), Equation (9) describes the amount of required on chip memory, where n_{PE} is the number of processing elements, p the number of output pixels created per PE per cycle and w the datapath width. Furthermore, $elem_{out,layer}$ presents the number of elements in a given

output channel. MEM_{width} and MEM_{depth} are the memory port width and memory depth,

$$n_{mem} = \left\lceil \frac{n_{PE} \times p \times w}{w_{hardware}} \right\rceil \times \max_{layer} \left(\left\lceil \frac{2 \times elem_{out,layer}}{p \times d_{hardware}} \right\rceil \right). \tag{9}$$

The first term in Equation (9) represents the requirement to write all results produced by the PEs into memory on every cycle, while the second term represents the need to store the complete output channel, using double buffering. Only one read port is needed, removing the last term of Equation (1).

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

9.1.5 Estimating on Board Memory Bandwidth Utilisation. The amount of data that need to be read to load the weights is shown in Equation (10), where n_{in} and n_{out} represent the number of inputs and outputs per layer and $S_{weights}$ represents the size per weight filter,

$$S_{weights,total} = \sum_{l=0}^{layer} n_{in,l} \times n_{out,l} \times S_{weights,l}. \tag{10}$$

The amount of memory that need to be accessed for the data per layer can be estimated by Equation (11), where n_{in} represents the number of input channels to the layer and n_{out} reflects the number of output channels. S_X represents the size of one input channel, S_Z the size of one output channel, and n_{PE} the number of PEs and therefore output channels processed in parallel. The boundary can be estimated as shown in Equation (12), where the efficiency will be very high, since all memory accesses are linear and most of them will use the maximum number of bursts,

$$S_{layer} = S_X \times n_{in} \times \left\lceil \frac{n_{out}}{n_{PE}} \right\rceil + S_Z \times n_{out}, \tag{11}$$

$$n = \frac{BW_{mem} \times efficiency \times T_{mem}}{\sum_{layer} S_{layer} + S_{weights,total}}. \tag{12}$$

9.1.6 Design Space Exploration. Using these equations from the performance model (Part 3) it is straightforward to perform a design space exploration. The targeted FPGA consists of three separate dies, called Super Logic Regions (SLRs), which are mounted on the same silicon interposer. The communication between those dies is a significant bottleneck.

However, to get around this problem, all three Super Logic Regions (SLRs) are treated as individual FPGAs. Each SLR is connected to one DDR DIMM and the PCIe bandwidth is shared.

The CNN input image sizes are fixed to 224×224, and as a result 150,528 pixels are sent to the FPGA and 100,352 elements are received. Assuming that 16 PEs are used the number of images that can be processed within 1 s can be calculated using Equation (6) as shown in Equation (13),

$$n = min\left(\frac{4GB/s \times 1s}{401{,}408B \times \frac{512}{16}}, \frac{4GB/s \times 1}{602{,}112B} \right) = 334. \tag{13}$$

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

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

Table 1. CNN Performance Comparison

	[90]	[48]	[81]	[5]	[91]	[47]	*This Work*
Precision	16 bits fixed	16 bits fixed	16 bits fixed	16 bits fixed	16 bits fixed	16 bits fixed	18-27 bits fixed
Freq (MHz)	150	150	231.85	303	385	200	240
Logic cell (K)	300	161	313	246	-	600	788
SRAM (Kb)	$1,248 \times 18$	$1,900 \times 20$	$1,668 \times 20$	$2,487 \times 20$	$1,450 \times 20$	$1,824 \times 18$	$3,128 \times 18$
Multipliers	2,833	1,518	1,500	1,476	2,756	2,520	6,057
TeraOp/s	0.354	0.645	1.17	1.3	1.7	2.9	2.45

For the whole network 1,705,181,184 convolutions need to be computed for each image. The proposed architecture can perform, as just calculated, $16 \times 14 = 224$ convolutions per second. Using Equation (8), we can calculate the compute boundary based on the frequency as shown in Equation (15). For a frequency of 250 MHz, this would mean that 32.8 images can be processed per cycle. It was decided that the data between layers should be buffered on board. As a result, the DDR bound performance can be estimated using Equation (16). Per image, 355 MB of pixel data and 33 MB of weights need to be transported. The memory efficiency used for this estimation is 0.85,

$$n = \frac{224 \frac{OPS}{cycle} \times f \times 1s}{1{,}705{,}181{,}184 OPS}, \tag{15}$$

$$n = \frac{14GB/s \times 0.85 \times 1s}{388MB} = 33. \tag{16}$$

As a result, the overall expected performance of all three SLRs is 99 images per second. Figure 6 shows other possible design points, where the equations are evaluated for the different degrees of parallelism and the area usage is predicted to discard all invalid design points. Blue dots in the figure are considered in the DSE (chosen design point in green). Red and grey dots were not considered because of resources and complexity constraints. It is possible to also represent the results of the design space exploration in a roofline model. This is shown in Figure 4. In this case, only the memory bandwidth with full efficiency and the optimal computational performance for multiply-adds at 250 MHz frequency are considered to determine the ceilings. In comparison to Figure 6, the roofline model includes memory bandwidth but does still not include on-chip memory requirements, which would increase with additional PEs, or PCIe bandwidth. As such, the more detailed performance model is required to determine invalid design points due to prohibitive on-chip memory usage.

9.1.7 Experimental Results and Discussion. Following the results of the design space exploration, the actual implementation was performed (Step g). Table 1 shows hardware utilisation as well as performance of our implementation in comparison to other state-of-the-art designs. Benchmarking of the CNN shows that the implemented design delivers 84.5 images per second at 240 MHz. This means that the error of the performance model is less than 15%. If a frequency of 240 MHz is used in the model, then the error decreases to less than 10%. This remaining difference can be explained by an overestimation of the on board memory efficiency.

Comparing the performance of our implementation with other state-of-the-art results, one can see that we are faster than in Reference [91] but slower than in Reference [47]. The reason for this can be largely explained by the difference in datatypes. Due to the fact that the multipliers of the VU9P FPGA have a native width of 18 by 27 bit and are the most limiting resource for the design, it was decided to use the full width for improved accuracy. On the Intel platform, used by most of the related work, it is possible to tile the multipliers into two 16-by-16 bit multipliers, which explains the slightly higher performance. Additionally, Reference [47] uses the Winograd efficient filtering algorithm [86], reducing the number of multiplications necessary for each convolution.

It should be stressed that the methodology does not promise the best performance achievable in general but only the state-of-the-art performance given a set of optimisations. For the architecture presented in this article, the Winograd optimisation was not considered. We still included one example of a design using Winograd in our comparison to show how our methodology leads to a design that outperforms related work using the same set optimisations but is itself outperformed once a wider optimisation space is considered. Especially in the space of Machine Learning (ML), the huge research interest leads to rapid improvements and many novel optimisation approaches. As a result, every design developed using our methodology only delivers a snapshot of the achievable performance considering the current known optimisations and in fields with such rapid progress regular reevaluation of the architecture might be required. Due to quick performance estimation enabled by the proposed methodology it is possible to judge if an update to a design yields sufficient advantages to justify the required effort. For the same reason, we do not compare against more recent related work, since the rapid progress in ML acceleration would only show the limitations of this specific implementation but not contribute to the evaluation of the overall methodology.

9.2 Berlin Quantum Chromodynamics

Our methodology was applied to a widely used QCD application. QCD is the physical theory of strong interactions between subatomic particles, and Lattice QCD (LQCD) is an approach to computationally simulate such particles, based on discretising space and time into a 4D lattice [85]. Berlin QCD (BQCD) [59] is a popular implementation of LQCD. It natively supports timing its compute steps, and a generated output file includes a detailed timing report that made utilities like gprof unnecessary (Part 1). Since the majority of the BQCD compute time was spent in its Conjugate Gradient (CG) solver, we focused on porting this (Step a). Overall BQCD contains roughly 200,000 lines of source code illustrating the complexity of the application. The results presented below show 20 times improvement over a BlueGene/Q on a compute-per-volume basis.

A Conjugate Gradient (CG) algorithm [29] iteratively solves an $M\mathbf{x} = \mathbf{b}$ equation for the unknown vector $\mathbf{x}$, where M and $\mathbf{b}$ are known. The algorithm relies on repeated multiplication of M by conjugate vectors. If M is sparse, rather than performing explicit matrix-vector multiplication, then it is more efficient to infer the effect of M when applied to a vector $\mathbf{p}$ by referencing only selected elements of M.

The CG variant that was chosen for the BQCD bitstream is a so-called s-step CG [19]. A small modification to the cited algorithm has been applied that reduces on-chip memory usage. It makes use of the fact that, in the present case, the matrix M can be written as a product of another matrix W and its conjugate transpose, $M = W^\dagger W$, such that we can replace an inner product of the form $\langle \mathbf{r}, M\mathbf{r} \rangle$ by $\langle W\mathbf{r}, W\mathbf{r} \rangle$, where $W\mathbf{r}$ happens to be an intermediate result that arises during the computation of $M\mathbf{r}$. The resulting algorithm is shown in Algorithm 2.

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

9.2.1 Architecture. During performance modelling (Part 3), it became clear that the QCD application would be bound by on-board memory bandwidth (Step e) and, hence, most design decisions

ALGORITHM 2: Conjugate Gradient algorithm used by BQCD bitstream

```
inputs: vectors b, x_0; matrices D̸_e, D̸_o, C_e, C_o; scalars μ, n, ϵ
goal: solve equation Mx = b for x, where M = (μA_oA_e + I)^† (μA_oA_e + I) and A_{e,o} = C_{e,o} D̸_{e,o}.
begin
  initialize vector r_0 = b − Mx_0, u_0 = Mr_0, p_0 = q_0 = 0
  initialize scalar ρ_k = ⟨r_0, r_0⟩, α_0 = ρ_k / ⟨r_0, u_0⟩, β_0 = 0
  loop for k ≥ 0
    update p_{k+1} = r_k + β_k p_k, q_{k+1} = u_k + β_k q_k, x_{k+1} = x_k + α_k p_{k+1}, r_{k+1} = r_k − α_k q_{k+1}
    calculate v = A_e r_{k+1}, s = μA_o v + r_{k+1}, h = A_o^† s
    update u_{k+1} = μA_e^† h + s, ρ_{k+1} = ⟨r_{k+1}, r_{k+1}⟩, β_{k+1} = ρ_{k+1}/ρ_k, α_{k+1} = ρ_{k+1} / (⟨s, s⟩ − ρ_{k+1}β_{k+1}/α_k)
    if ρ_{k+1} < ϵ or k ≥ n then break
  end
  return x_{k+1}
end
```

were taken to minimise on board memory I/O (Part 4). The resulting architecture is displayed in Figure 7. Spinors, gauges, and clover matrices are read from on board memory and streamed into the CG algorithm. Each CGKernel performs a $\not{D}$ and clover operation and some perform additional $a\mathbf{x} + \mathbf{y}$ operations required by the CG algorithm. Some spinor vector results are output by CGKernel0, whereas the result of the matrix multiplication $M\mathbf{p}$ is output by CGKernel3. These results are streamed back to on board memory to be used by the next CG iteration. CGKernels 0 and 1 accumulate the square norm of certain spinor vectors, which are fed to the CGControlKernel at the end of a CG iteration, which uses these numbers to compute the scalars required for the next CG iteration. Note that to save chip space, the CGKernels process a lattice site only every 4 cycles. Since a CGKernel needs to reference all neighbours in 4D, its latency is effectively $2L_XL_YL_Z \times 4$, where L_i denotes the number of sites in direction i, and 4 the cycles needed per site.

9.2.2 Performance Modelling. The performance model (Part 3) is created (Step d) based on this Architecture (Part 4).

Compute-bound time to solution. The number of items processed by the kernels shown in Figure 7 can be seen in Equation (17),

$$n_{items} = (L_X + 2H_X)(L_Y + 2H_Y)(L_Z + 2H_Y)(L_T + 2H_T), \tag{17}$$

where H_i denotes the number of halo sites in direction i ($i \in \{X, Y, Z, T\}$). In our example, $H_X = H_Y = H_Z = 0$ for all kernels, and the maximum value taken by H_T is 4 (this occurs both in the gauge initialisation kernel as well as CGKernel0), such that $n_{items} = L_XL_YL_Z(L_T + 8)$. The number of items processed per cycle is 1/4, as 4 cycles are required to process a single item. In addition, we consider the pipeline latency, because a new CG iteration cannot start until the previous one has finished. The number of extra cycles incurred this way is equivalent to $3L_XL_YL_Z$ elements[3], which gets added to n_{items}. Using Equation (2), the compute-bound time for single CG iteration becomes Equation (18),

$$T_{comp} = \frac{4\,(n_{items} + 3L_XL_YL_Z)}{f} = \frac{4L_XL_YL_Z(L_T + 11)}{f}. \tag{18}$$

The compute-bound time can be plotted as a function of inverse frequency (i.e. clock period), for a $16 \times 16 \times 16 \times 32$ problem size and a CG run that took 53 iterations to converge, as shown Figure 8.

Memory-bound time to solution. T_{mem} can be gained from the total data size read from, and written to, on board memory. For each ordinary lattice point 8 gauge matrices are needed that each consist of 12 real numbers, 2 clover matrices of 72 numbers, and 5 spinors of 24 numbers. For

[3]The factor 3 (rather than 4) arises from the fact that the next CG iteration can start even if CG kernel 3 has not finished emptying its pipeline but has finished sending out the results of the previous iteration.

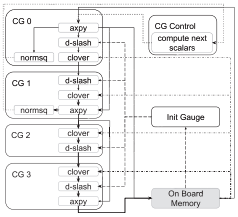

Fig. 7. BQCD design architecture.

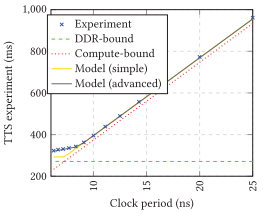

Fig. 8. BQCD CG time-to-solution.

halos only 3 spinors per lattice point are needed. The number of halos to read differs per data type: The gauges and 3 spinor streams are read with 4 halos, whereas one clover stream requires 3 and the other 2 halos. Hence the total number of values read from on board memory is computed in Equation (19),

$$\begin{aligned} N_{num,r} &= L_X L_Y L_Z \left(L_T \left(5 \times 24 + 8 \times 12 + 2 \times 72\right) + 8 \left(3 \times 24 + 8 \times 12\right) + 6 \times 72 + 4 \times 72\right) \\ &= L_X L_Y L_Z (2064 + 360 L_T). \end{aligned} \tag{19}$$

At ordinary lattice points, 5 spinors of 24 numbers each are written to on board memory, and no data are written during halos as shown in Equation (20). Since for this example we stream all data in 24-bit fixed point format, Equation (4) can thus be evaluated as shown in Equation (21) by combining Equation (19) and Equation (20),

$$N_{num,w} = L_X L_Y L_Z L_T \times 5 \times 24 = 120 L_X L_Y L_Z L_T, \tag{20}$$

$$T_{mem} = \frac{L_X L_Y L_Z \left(2064 + 480 L_T\right) \times 3\text{B}}{BW \times efficiency}. \tag{21}$$

In Figure 8, the on board memory bound time is shown as a horizontal line (it is independent of kernel clock frequency), for a problem size of $16 \times 16 \times 16 \times 32$, a bandwidth of 49 GB/s and efficiency of 80% and, as before, 53 CG iterations give $T_{mem} = 270.98$ ms.

Modelled time-to-solution. The previous two ingredients (compute-bound and on board memory bound time to solution) already provide good estimate of the expected time-to-solution. Two additional factors are the PCIe I/O time and the initialisation overhead. Both are additive (data are written and read strictly before and after CG is ran) to the total time-to-solution of all CG iterations needed to converge to a solution. Due to implementation details, only a pair of spinor vectors need to be streamed before and after the FPGA run, whose size is computed by multiplying the number of values by 2 (to account for input and output) as shown in Equation (22). For a problem size of $16 \times 16 \times 16 \times 32$, the estimated PCIe time is 11.7 ms. The estimated initialisation overhead is 10ms and the resulting theoretical time-to-solution as shown in Equation (23) is plotted in Figure 8 as a yellow line,

$$N_{num,pcie} = L_X L_Y L_Z L_T \times 2 \times 24 = 48 L_X L_Y L_Z L_T, \tag{22}$$

$$\max(T_{comp}, T_{mem}) + T_{PCIe} + T_{overhead}. \tag{23}$$

9.2.3 Experimental Results and Discussion. Figure 8 depicts our experimental results. Generally the estimated results matched experimental measurements very closely. However, at higher operating frequencies, a discrepancy in time-to-solution of around 10% is observed. This is due to the assumption that data accesses to on board memory can be averaged over the run. In reality, on board memory I/O varies during a CG iteration, being lower for halo sites than when computing

ordinary sites. This means that there will be a transitional frequency range where part of the execution is on board memory bound and the other part is compute bound. By treating the halo and core compute separately, whilst assuming the "flushing" cycles after each CG iteration to be compute bound, a more accurate model is created (the grey line in Figure 8). It should be noted, that more simplified models, e.g., the roofline model, will not be able to accurately predict this behaviour.

9.3 Additional Previously Reported Examples

The methodology at hand was used and refined within Maxeler for more than a decade in both industrial and academic contexts. The industrial results, however, are mostly confidential. Instead we highlight published work, that benefited from our methodology to stress its wide applicability.

Curran's Approximation Algorithm: The proposed methodology was used in the development of the hardware accelerated version of Curran's approximation algorithm as presented in [60]. Using the performance model (Part 3) the authors were able to quickly identify a feasible and promising architecture (Part 4). However, they diverged from the methodology slightly by testing different numerical options in hardware. The reason for this was twofold. First, the performance model showed that the chosen architecture did not change based on the final datatype used (Step e). Second, the required changes in the hardware design were very limited and enabled rapid evaluation on large datasets compared to a CPU simulation. This study shows how the methodology can be adjusted for specific use cases while still fulfilling its major objectives and accelerating the development. Speed-ups between 4.0× and 9.2× for different data sets using multi-threaded CPU implementations as a baseline were achieved. Predicted and actual runtimes fell within 1%.

Smith-Waterman with Traceback for High Throughput Next Generation Sequencing: The accelerated DNA sequencing aligner developed in Reference [41] used the methodology. The analysis (Part 1) and performance prediction (Part 3) of the methodology identified early that communication overheads could negate any alignment acceleration of the entire application. As a result the architecture was developed with the target of minimising the number of calls to the accelerator and reduce the amount of data transferred. This resulted in a 37% application-level speedup with a single accelerator. By highlighting this issue early on (Step e), the methodology avoided the danger of a complete redesign, which would otherwise have been required, even though the performance model was less accurate and the predicted runtime was off by close to 25%.

Real Time Radiotherapy Simulation: In Reference [78] an application to simulate the dose distribution in human tissue in the context of radiotherapy using a Monte Carlo simulation is presented. The authors describe the architecture (Part 4) and performance model (Part 3) in great detail. The major challenge of this application is, that the amount of computation and the data access pattern depend on random number generators. As a result it was necessary to analyse the application on a large dataset to determine the statistic properties of the execution (Part 1, Step b). The resulting performance model managed to predict the execution time highly accurate in most cases, however introduced some larger errors in some cases, which are explained in depth in the cited article. Additionally, the area prediction was correct for DSPs, had small errors for LUTs (between 1.5% and 15%) and larger errors for FFs and memories (up to 36%). All in all, the methodology enabled the development of a feasible architecture, which circumvented all bottlenecks (Step e) and the first hardware implementation managed to outperform previously published CPU and GPU solutions.

9.4 Discussion

In this section we were able to illustrate that the methodology was successfully applied to multiple different use cases. Additionally, there are many more cases, that we cannot publicly share. We have not yet encountered an application, in which the methodology failed or was not helpful.

One lesson we learned is that complete accuracy of the area prediction is often not important. It is possible to quickly make a worst-case calculation, which includes large safety margins, to discover the resource that is most scarce. One can then model the resource usage for this specific resource in greater detail with higher accuracy. One example of this is the CNN acceleration as described in Section 9.1 where we fully focus on multipliers, on-chip memory capacity and bandwidths. In the radiotherapy simulation use case described above, we focused more on accurate prediction of all resources, but even there the error for logic and on-chip memory is larger than for the more critical memory bandwidth. Additionally, the obtainable accuracy for all hardware resource is different. For example, the usage of DSPs can be usually calculated without any error with low effort, but the usage of memory resources is harder to accurately model due to automated scheduling and tool decisions. However, to aid timing closure designs should use less than 80% of each resource. This means that for all resources not introducing bottlenecks bull park calculations and worst-case assumptions are sufficient and only for scarce resources more attention to detail is required.

In all examples provided in this article the methodology helped to quickly discover bottlenecks and design an architecture that proved to achieve the design goals. This enabled us to only implement the hardware design once and avoid time consuming and risky design iterations on the hardware design. Measuring the advantages of one development process compared to another in great detail is always a difficult undertaking, since developer experience has a great impact. Additionally, the same developer cannot implement the same application using different methodologies, since lessons learned in earlier designs will automatically influence design iterations in subsequent implementations. As such, we cannot measure the time savings using the proposed methodology but need to argue based on our experience. Some of the projects discussed above required multiple man-years of development. Discovering major issues late can result in a complete redesign causing unmanageable financial and operational risks. If one manages to directly implement a hardware design that is not bottlenecked without using the methodology, then there is a time saving that we can estimate to roughly 10%. But this also means that if any design iteration in hardware that changes more than 10% of the design is required, then the methodology will have reduced the overall workload. In all use cases presented here, the final results were obtained after tens of iterations between the performance model and the architecture. While early iterations were usually quite impacting, later ones only optimised architectural details. As such, the ability to perform design iterations in days instead of years greatly reduced the risks associated with hardware design.

10 CONCLUSIONS AND FURTHER WORK

We presented a methodology to rapidly develop predictable high-performance applications using FPGAs. Application-centric modelling is used to accurately predict system performance and identify the best system architecture before the hardware coding is started. As such, complicated and time consuming design iterations on the actual hardware design are omitted. Accurate predictions of system properties early in the development process allow various business and design decisions.

Our proposal involves four phases, analysis, software modelling, predicting system properties and architectural development, iteratively co-developed to gain the best results. Two realistic case studies demonstrated that following our methodology yields state-of-the-art performance. Predicted speedups stayed within 15% of real measurements and as intended the first implementations met specification targets. Additionally, we highlighted published work based on the methodology targeting different use cases. Future work includes portability enhancement between different accelerator generations and accuracy improvements of hardware estimations for operation scheduling.

REFERENCES

[1] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. 2014. Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*. ACM.

[2] Alibaba. [n.d.]. *AliBaba f2 Instance*. Retrieved from https://www.alibabacloud.com/help/doc-detail/25378.htm#concept-sx4-lxv-tdb-f2.

[3] Amazon. [n.d.]. *Amazon F1 Instance*. Retrieved from https://aws.amazon.com/ec2/instance-types/f1/.

[4] J. Arram, K. H. Tsoi, Wayne Luk, and P. Jiang. 2013. Hardware acceleration of genetic sequence alignment. In *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, Berlin, 13–24.

[5] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL™ deep learning accelerator on arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 55–64.

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the Design Automation Conference*.

[7] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*.

[8] Tobias Becker, Oskar Mencer, Stephen Weston, and Georgi Gaydadjiev. 2015. *Maxeler Data-Flow in Computational Finance*. Springer International Publishing, Cham, 243–266.

[9] David Boland and George A. Constantinides. 2013. Word-length optimization beyond straight line code. In *Proceedings of the 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'13)*. 105–114.

[10] Jacob A. Bower, James Huggett, Oliver Pell, and Michael J. Flynn. 2008. A Java-based system for FPGA programming. In *Proceedings of the FPGA World Conference*.

[11] Doug Burger. 2020. Keynote: Will programmable hardware reach scale. In *Proceedings of the International Conference on Field Program. Logic and App.*

[12] M. A. Cantin, Y. Blaguiere, Y. Sarvaria, P. Lavoie, and E. Granger. 2000. Analysis of quantization effects in a digital hardware implementation of a fuzzy ART neural network algorithm. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'00)*, Vol. 3. 141–144.

[13] M. A. Cantin, Y. Savaria, and P. Lavoie. 2002. A comparison of automatic word length optimization procedures. In *Proceedings of the 2002 IEEE International Symposium on Circuits and Systems. Proceedings*, Vol. 2.

[14] L. C. Carrington, M. Laurenzano, A. Snavely, R. L. Campbell, and L. P. Davis. 2005. How well can simple metrics represent the performance of HPC applications? In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC'05)*. 48–48.

[15] Gary Chun Tak Chow, Anson Hong Tak Tse, Qiwei Jin, Wayne Luk, Philip H. W. Leong, and David B. Thomas. 2012. A mixed precision Monte Carlo methodology for reconfigurable accelerator systems. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 57–66.

[16] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens. 1999. A methodology and design environment for DSP ASIC fixed point refinement. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 1999*. 271–276.

[17] Bruno da Silva, An Braeken, Erik D'Hollander, and Abdellah Touhafi. 2014. Performance and resource modeling for FPGAs using high-level synthesis tools. In *Advances in Parallel Computing*, Vol. 25. IOS Press, 523–531.

[18] Bruno da Silva, An Braeken, Erik H. D'Hollander, and Abdellah Touhafi. 2013. Performance modeling for FPGAs: Extending the roofline model with high-level synthesis tools. *Int. J. Reconfig. Comput.* 2013, Article 7 (Jan. 2013), 1 page.

[19] James W. Demmel, Michael T. Heath, and Henk A. van der Vorst. 1993. Parallel numerical linear algebra. *Acta Numer.* 2 (1993), 111–197.

[20] Jack B. Dennis. 1980. Data flow supercomputers. *Computer* 13, 11 (Nov. 1980), 48–56.

[21] Jack Dongarra and Whitney Heins. 2013. Professor Jack Dongarra Announces New Supercomputer Benchmark. Retrieved from https://news.utk.edu/2013/07/10/professor-jack-dongarra-announces-supercomputer-benchmark/.

[22] E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G. B. Newby. 2007. Comparative analysis of high level programming for reconfigurable computers: Methodology and empirical study. In *Proceedings of the 2007 3rd Southern Conference on Programmable Logic*. 99–106.

[23] Ad Emmen. 2020. Benchmarking Fugaku and Summit: A Revealing Process. Retrieved from http://primeurmagazine.com/flash/LV-PL-06-20-7.html.

[24] Forbes. [n.d.]. *Supercomputer Manages Fixed Income Risk at JPMorgan*. Retrieved from https://www.forbes.com/sites/tomgroenfeldt/2012/03/20/supercomputer-manages-fixed-income-risk-at-jpmorgan/#376c5e5f1001.

[25] L. Gan, H. Fu, W. Luk, C. Yang, W. Xue, X. Huang, Y. Zhang, and G. Yang. 2013. Accelerating solvers for global atmospheric equations through mixed-precision data flow engine. In *Proceedings of the 2013 23rd International Conference on Field programmable Logic and Applications*. 1–6.

[26] C. Guo, H. Fu, and W. Luk. 2012. A fully-pipelined expectation-maximization engine for Gaussian Mixture Models. In *Proceedings of the 2012 International Conference on Field-Programmable Technology*. 182–189.

[27] John L. Gustafson. 1988. Reevaluating Amdahl's law. *Commun. ACM* 31, 5 (May 1988), 532–533.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. abs/1512.0338. Retrieved from http://arxiv.org/abs/1512.03385.

[29] M. R. Hestenes and E. Stiefel. 1952. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Standards* 49, 6 (Dec. 1952), 409–436.

[30] Brian Holland, Karthik Nagarajan, Chris Conger, Adam Jacobs, and Alan D. George. 2007. RAT: A methodology for predicting performance in application design migration to FPGAs. In *Proceedings of the 1st International Workshop on High-performance Reconfigurable Computing Technology and Applications: Held in Conjunction with SC07 (HPRCTA'07)*. ACM, New York, NY, 1–10.

[31] Sunpyo Hong and Hyesoon Kim. [n.d.]. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, New York, NY, 280–289.

[32] Neng Hou, Xiaohu Yan, and Fazhi He. 2019. A survey on partitioning models, solution algorithms and algorithm parallelization for hardware/software co-design. *Des. Autom. Embed. Syst.* 23, 1 (01 Jun. 2019), 57–77.

[33] Chris Jones. 2015. Maxeler Dense Matrix Multiply. Retrieved from https://github.com/maxeler/Dense-Matrix-Multiplication.

[34] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, 1–12.

[35] Y. Kebbati and H. K. Souffi. 2007. Optimized design methodology for an integration of electrical control systems. In *Proceedings of the 2007 Canadian Conference on Electrical and Computer Engineering*. 1657–1660.

[36] H. Keding, M. Willems, M. Coors, and H. Meyr. 1998. FRIDGE: A fixed-point design and simulation environment. In *Proceedings of the Design, Automation and Test in Europe Conference*. 429–435.

[37] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. 1995. Fixed-point optimization utility for C and C++ based digital signal processing programs. In *VLSI Signal Processing, VIII*. 197–206.

[38] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. 1998. Fixed-point optimization utility for C and C++ based digital signal processing programs. *IEEE Trans. Circ. Syst. II: Analog Dig. Sign. Process.* 45, 11 (Nov. 1998), 1455–1464.

[39] Martin Kleppmann. 2017. *Designing Data-intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc.

[40] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. 2016. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 115–127.

[41] K. Koliogeorgi, N. Voss, S. Fytraki, S. Xydis, G. Gaydadjiev, and D. Soudris. 2019. Dataflow acceleration of smith-waterman with traceback for high throughput next generation sequencing. In *Proceedings of the 2019 29th International Conference on Field Programmable Logic and Applications (FPL'19)*.

[42] W. Kramer. 2012. Top500 versus sustained performance - the top problems with the TOP500 list - and what to do about them. In *Proceedings of the 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. 223–230.

[43] H. T. Kung. 1982. Why systolic architectures? *Computer* 15, 1 (Jan. 1982), 37–46.

[44] G. L. B. Lima, G. A. L. Ferreira, O. Saotome, A. M. d. Cunha, and L. A. V. Dias. 2015. Hardware development: Agile and co-design. In *Proceedings of the 2015 12th International Conference on Information Technology—New Generations*. 784–787.

[45] Yonghua Lin and Ling Shao. 2015. Super vessel: The open cloud service for OpenPOWER. *White Paper, IBM Corporation* (2015).

[46] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and O. Mencer. 2011. Beyond traditional microprocessors for geoscience high-performance computing applications. *IEEE Micro* 31, 2 (Mar. 2011), 41–49.

[47] L. Lu, Y. Liang, Q. Xiao, and S. Yan. 2017. Evaluating fast algorithms for convolutional neural networks on FPGAs. In *Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*. 101–108.

[48] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM.

[49] Mariem Makni, Mouna Baklouti, Smail Niar, and Mohamed Abid. 2017. Hardware resource estimation for heterogeneous FPGA-based SoCs. In *Proceedings of the Symposium on Applied Computing (SAC'17)*. ACM, New York, NY, 1481–1487.

[50] Paolo Marchetti, Diego Oriato, Oliver Pell, A. M. Cristini, and D. Theis. 2010. Fast 3D ZO CRS stack—An FPGA implementation of an optimization based on the simultaneous estimate of eight parameters. In *Proceedings of the 72nd EAGE Conference and Exhibition incorporating SPE EUROPEC'10*.

[51] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation* (1st ed.). Morgan Kaufmann, San Francisco, CA.

[52] O. Mencer, M. Platzner, M. Morf, and M. J. Flynn. 2001. Object-oriented domain specific compilers for programming FPGAs. *IEEE Trans. VLSI Syst.* 9, 1 (Feb. 2001), 205–210.

[53] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and Dharmendra S. Modha. 2014. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* 345, 6197 (2014), 668–673.

[54] Giovanni De Micheli. 1994. *Synthesis and Optimization of Digital Circuits* (1st ed.). McGraw–Hill Higher Education.

[55] Microsoft. [n.d.]. *Inside the Microsoft FPGA-based Configurable Cloud*. Retrieved from https://azure.microsoft.com/en-gb/resources/videos/build-2017-inside-the-microsoft-fpga-based-configurable-cloud/.

[56] E. Monmasson and M. N. Cirstea. 2007. FPGA design methodology for industrial control systems—A review. *IEEE Trans. Industr. Electr.* 54, 4 (Aug. 2007), 1824–1842.

[57] Wolfgang Müller, Wolfgang Rosenstiel, and Jürgen Ruf. 2007. *SystemC: Methodologies and Applications*. Springer Science & Business.

[58] Syed Waqar Nabi and Wim Vanderbauwhede. 2019. FPGA design space exploration for scientific HPC applications using a fast and accurate cost model based on roofline analysis. *J. Parallel Distrib. Comput.* 133 (2019), 407–419.

[59] Yoshifumi Nakamura and Hinnerk Stüben. 2014. BQCD: Berlin quantum chromodynamics program. arXiv1011.0199. Retrieved from https://arxiv.org/abs/1011.0199.

[60] A. M. Nestorov, E. Reggiani, H. Palikareva, P. Burovskiy, T. Becker, and M. D. Santambrogio. 2017. A scalable dataflow implementation of Curran's approximation algorithm. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'17)*. 150–157.

[61] Rajat K. Pal. 2000. *Multi-layer Channel Routing Complexity and Algorithms*. Alpha Science Int'l Ltd.

[62] Oliver Pell, Oskar Mencer, Kuen Hung Tsoi, and Wayne Luk. 2013. Maximum performance computing with dataflow engines. In *High-Performance Computing Using FPGAs*. Springer, New York, NY, 747–774.

[63] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating configurable hardware from parallel patterns. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. 651–665.

[64] R. Rashid, J. G. Steffan, and V. Betz. 2014. Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS. In *Proceedings of the 2014 International Conference on Field-Programmable Technology (FPT'14)*. 20–27.

[65] T. Riesgo, Y. Torroja, and E. de la Torre. 1999. Design methodologies based on hardware description languages. *IEEE Trans. Industr. Electr.* 46, 1 (1999), 3–12.

[66] B. Rountree, D. K. Lowenthal, M. Schulz, and B. R. de Supinski. 2011. Practical performance prediction under dynamic voltage frequency scaling. In *Proceedings of the 2011 International Green Computing Conference and Workshops*. 1–8.

[67] Y. S. Shao, B. Reagen, G. Y. Wei, and D. Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture*.

[68] Changchun Shi and R. W. Brodersen. 2004. Automated fixed-point data-type optimization tool for signal processing and communication systems. In *Proceedings of the 41st Design Automation Conference*. 478–483.

[69] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv 1409.1556. Retrieved from https://arxiv.org/abs/1409.1556.

[70] V. Srinivasan, S. Radhakrishnan, and R. Vemuri. 1998. Hardware software partitioning with integrated hardware design space exploration. In *Proceedings of the Design, Automation and Test in Europe Conference*. 28–35.

[71] S. Summers, A. Rose, and P. Sanders. 2017. Using MaxCompiler for the high level synthesis of trigger algorithms. *J. Instrum.* 12, 02 (Feb. 2017), C02015–C02015.

[72] D. E. Thomas, J. K. Adams, and H. Schmit. 1993. A model and methodology for hardware-software codesign. *IEEE Des. Test Comput.* 10, 3 (1993), 6–15.

[73] M. M. Tikir, L. Carrington, E. Strohmaier, and A. Snavely. 2007. A genetic algorithms approach to modeling the performance of memory-bound computations. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC'07)*. 1–12.

[74] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung. 2005. Reconfigurable computing: Architectures and design methods. *IEE Proc. Comput. Dig. Techn.* 152, 2 (Mar. 2005), 193–207.

[75] Carlo Tomas, Luca Cazzola, Diego Oriato, Oliver Pell, Daniela Theis, Guido Satta, and Ernesto Bonomi. 2012. Acceleration of the anisotropic PSPI imaging algorithm with dataflow engines. In *SEG Technical Program Expanded Abstracts 2012*. 1–5.

[76] Nils Voss, Tobias Becker, Oskar Mencer, and Georgi Gaydadjiev. 2017. *Rapid Development of Gzip with MaxJ*. Springer International Publishing, Cham, 60–71.

[77] Nils Voss, Stephen Girdlestone, Tobias Becker, Oskar Mencer, Wayne Luk, and Georgi Gaydadjiev. 2019. Low area overhead custom buffering for FFT. In *Proceedings of the 2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig'19)*. IEEE, 1–8.

[78] Nils Voss, Peter Ziegenhein, Lukas Vermond, Joost Hoozemans, Oskar Mencer, Uwe Oelfke, Wayne Luk, and Georgi Gaydadjiev. 2020. Towards real time radiotherapy simulation. *J. Sign. Process. Syst.* 92, 9 (01 Sep. 2020), 949–963.

[79] Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Kunio Uchiyama. 2018. *FPGA Accelerator Design Using OpenCL*. Springer International Publishing, 29–43.

[80] Z. Wang, B. He, W. Zhang, and S. Jiang. 2016. A performance analysis framework for optimizing OpenCL applications on FPGAs. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 114–125.

[81] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 29.

[82] S. Weston, J. Marin, J. Spooner, O. Pell, and O. Mencer. 2010. Accelerating the computation of portfolios of tranched credit derivatives. In *Proceedings of the 2010 IEEE Workshop on High Performance Computational Finance*. 1–8.

[83] Stephen Weston, James Spooner, Sébastien Racanière, and Oskar Mencer. 2011. Rapid computation of value and risk for derivatives portfolios. *Concurr. Comput.: Pract. Ex.* 24, 8 (2011), 880–894.

[84] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52 (Apr. 2009).

[85] Kenneth G. Wilson. 1974. Confinement of quarks. *Phys. Rev. D* 10, 8 (Oct. 1974), 2445–2459.

[86] S. Winograd. 1980. *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics.

[87] Xilinx. [n.d.]. *Baidu Deploys Xilinx FPGAs in New Public Cloud Acceleration Services*. Retrieved from https://www.xilinx.com/news/press/2017/baidu-deploys-xilinx-fpgas-in-new-public-cloud-acceleration-services.html.

[88] ZDNet. [n.d.]. *Intel FPGAs Picked up by Dell EMC and Fujitsu*. Retrieved from https://www.zdnet.com/article/intel-fpgas-picked-up-by-dell-emc-and-fujitsu/.

[89] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. E. Hinton. 2013. On rectified linear units for speech processing. In *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 3517–3521.

[90] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'16)*. 1–8.

[91] Jialiang Zhang and Jing Li. 2017. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 25–34.

[92] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He. 2017. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*. 430–437.

[93] G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar. 2017. Design space exploration of FPGA-based accelerators with multi-level parallelism. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'17)*.

Received June 2020; revised October 2020; accepted November 2020