

FPGAs for Domain Experts

Lead Guest Editor: Wim Vanderbauwhede

Guest Editors: Sven-Bodo Scholz and Martin Margala





FPGAs for Domain Experts

FPGAs for Domain Experts

Lead Guest Editor: Wim Vanderbauwhede

Guest Editors: Sven-Bodo Scholz and Martin
Margala



Copyright © 2020 Hindawi Limited. All rights reserved.

This is a special issue published in "International Journal of Reconfigurable Computing." All articles are open access articles distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.






Editorial Board

Christophe Bobda, USA
Jose A. Boluda, Spain
João Cardoso, Portugal
René Cumplido, Mexico
Michael Hübner, Germany
John Kalomiros, Greece
Volodymyr Kindratenko, USA
Miriam Leeser, USA
Guy Lemieux, Canada
Martin Margala, USA
Gokhan Memik, USA
Seda Ogrenci-Memik, USA
Marco Platzner, Germany
Ron Sass, USA
Walter Stechele, Germany
Jim Torresen, Norway

Contents

FPGAs for Domain Experts

Wim Vanderbauwhede , Sven-Bodo Scholz , and Martin Margala 

Editorial (2 pages), Article ID 2725809, Volume 2020 (2020)

Automatic Pipelining and Vectorization of Scientific Code for FPGAs

Syed Waqar Nabi  and Wim Vanderbauwhede 




Research Article (12 pages), Article ID 7348013, Volume 2019 (2019)

Dimension Reduction Using Quantum Wavelet Transform on a High-Performance Reconfigurable Computer

Naveed Mahmud  and Esam El-Araby 





Research Article (14 pages), Article ID 1949121, Volume 2019 (2019)

Translating Timing into an Architecture: The Synergy of COTSon and HLS (Domain Expertise—Designing a Computer Architecture via HLS)

Roberto Giorgi , Farnam Khalili , and Marco Procaccini 



Research Article (18 pages), Article ID 2624938, Volume 2019 (2019)

An FPGA-Based Hardware Accelerator for CNNs Using On-Chip Memories Only: Design and Benchmarking with Intel Movidius Neural Compute Stick

Gianmarco Dinelli , Gabriele Meoni , Emilio Rapuano, Gionata Benelli , and Luca Fanucci 

Research Article (13 pages), Article ID 7218758, Volume 2019 (2019)

Implementing and Evaluating an Heterogeneous, Scalable, Tridiagonal Linear System Solver with OpenCL to Target FPGAs, GPUs, and CPUs

Hamish J. Macintosh , Jasmine E. Banks , and Neil A. Kelson

Research Article (13 pages), Article ID 3679839, Volume 2019 (2019)

Editorial

FPGAs for Domain Experts

Wim Vanderbauwhede ¹, **Sven-Bodo Scholz** ², and **Martin Margala** ³

¹University of Glasgow, Glasgow, UK

²Heriot-Watt University, Edinburgh, UK

³University of Massachusetts Lowell, Lowell, MA, USA

Correspondence should be addressed to Wim Vanderbauwhede; wim.vanderbauwhede@glasgow.ac.uk

Received 4 May 2020; Accepted 16 September 2020; Published 27 October 2020

Copyright © 2020 Wim Vanderbauwhede et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Field-Programmable Gate Arrays (FPGAs) have recently gained a lot of attention through demonstrated superior performance over off-the-shelf architectures, not only with respect to energy efficiency but also with respect to wall-clock runtimes. For a long time, FPGAs had been used primarily as prototyping devices or in embedded systems but are now increasingly accepted as first-order computing devices on desktops and servers. This change has been driven by a combination of increasingly larger and resourceful FPGAs and wider availability of mature and stable high-level FPGA programming tools.

The application areas range across many domains from high-finance to advanced machine learning. Despite the availability of many tools for high-level synthesis and increasing ease of access to FPGA-based computing nodes (e.g., via Amazon Web Services), domain experts still are far away from utilising FPGAs to gain processing performance unless preconfigured systems for their particular applications exist in readily available form. Manycore CPUs and GPUs are still generally considered the only viable options for domain experts looking to accelerate their applications.

Against this background, there has been considerable research in recent years on making FPGAs accessible for domain experts. With this special issue, we are bringing together work that aims to break this barrier for a wider applicability of FPGAs.

This special issue combines contributions from researchers and practitioners that share the vision of enabling domain experts to benefit from the performance opportunities of FPGAs.

We hope that you enjoy this special issue, and this paper collection as a whole can introduce readers to the varied and challenging area of FPGA computing, presenting several state-of-the-art solutions from diverse perspectives. All accepted papers provide relevant and interesting research techniques, models, and work directly applied to the area of scientific FPGA programming.

Finally, we would like to thank all the authors for their submissions to this special issue and the also the reviewers for dedicating their time to provide detailed comments and suggestions that helped to improve the quality of this special issue.

The first paper, “Automatic Pipelining and Vectorization of Scientific Code for FPGAs,” focuses on FPGA compilation of legacy scientific code in Fortran. There is a very large body of legacy scientific code still in use today, and much new scientific code is still being written in Fortran-77. Many of these codes would benefit from acceleration on GPUs and FPGAs. Manual translation of such legacy code parallel code for GPUs or FPGAs requires a considerable manual effort. This is a major barrier to wider adoption of FPGAs. The authors of this paper have been developing an automated optimizing compiler to lower this barrier. Their aim is to compile legacy Fortran code automatically to FPGA, without any need for rewriting or insertion of pragma. The compiler applies suitable optimizations based on static code analysis. The paper focuses on two key optimizations, automatic pipelining and vectorization. The compiler identifies portions of the legacy code that can be pipelined and vectorized. The backend generates coarse-grained pipelines and

automatically vectorizes both the memory access and the data path based on a cost model, generating an OpenCL-HDL hybrid solution for FPGA targets on the Amazon cloud. The results show up a performance improvement of up to four times over baseline OpenCL code.

The second paper, “Dimension Reduction Using Quantum Wavelet Transform on a High-Performance Reconfigurable Computer,” introduces a very interesting and exciting new field, the use of FPGAs for the acceleration of quantum computing simulations. Simulation is a crucial step in the development of quantum computers and algorithms, and FPGAs have huge potential to accelerate this type of simulations. The paper proposes to combine dimension reduction techniques with quantum information processing for application in domains that generate large volumes of data such as high-energy physics (HEP). It focuses on using quantum wavelet transform (QWT) [1] to reduce the dimensionality of high spatial resolution data. The quantum wavelet transform takes advantage of quantum superposition to reduce computing time for the processing of exponentially larger amounts of information. The authors present a new emulation architecture to perform QWT and its inverse on high-resolution data, and a prototype of an FPGA-based quantum emulator. Experiments using high-resolution image data on a state-of-the-art multinode high-performance reconfigurable computer show that the proposed concepts represent a feasible approach to reducing the dimensionality of high spatial resolution data generated by applications in HEP.

The third paper, “Translating Timing into an Architecture: The Synergy of COTSon and HLS (Domain Expertise—Designing a Computer Architecture via HLS),” provides an in-depth description of a high-level synthesis workflow around Vivado HLS tools. It comprises tools on both sides of HLS: tools for design space exploration prior to running HLS named COTSon and MYDSE as well as tools for targeting a custom build hardware, the AXIOM board. The article provides a good overview of the tools and the overall workflow through the HLS tool. The abstract description of the workflow is substantiated by an in-depth presentation of example applications including the design of a system for distributed computation across multiple FPGA boards. Finally, some empirical evidence for the predictive capabilities of the tool chain is being presented. Overall, this contribution not only demonstrates the challenges involved when designing complex systems with HLS at the core nicely but also features the presentation of custom-made tooling which can be used by the wider community.

The fourth paper, “An FPGA-Based Hardware Accelerator for CNNs Using On-Chip Memories Only: Design and Benchmarking with Intel Movidius Neural Compute Stick,” presents a full on-chip FPGA hardware accelerator for a separable convolutional neural network designed for a keyword spotting application. This is a quantized neural network realized exclusively using on-chip memories. The design is based on the Intel Movidius Neural Compute Stick and compares against this device, which deploys a custom accelerator, the Intel Movidius Myriad X Vision Processing

Unit (VPU) [2]. The results show that better inference time and energy per inference result can be obtained with comparable accuracy. This is a striking result as the VPU is a dedicated accelerator touting ultralow power and high performance and serves to showcase the potential of quantized CNNs on FPGAs.

The final paper “Implementing and Evaluating an Heterogeneous, Scalable, Tridiagonal Linear System Solver with OpenCL to Target FPGAs, GPUs, and CPUs,” addresses the problem of solving linear systems, a very common problem in scientific computing and HPC. As indicated by the title, the paper focuses in particular on diagonally dominant tridiagonal linear systems using the truncated-SPIKE algorithm [3] and presents a numerically stable optimised FPGA implementation using the open standard OpenCL [4]. The paper compares implementations of the algorithm on CPU, GPU, and FPGA as well as provides comparison against an optimised implementation of the TDMA solver [5]. The FPGA implementation is shown to have better performance per Watt than the CPU and GPU, and the truncated-SPIKE algorithm outperforms the TDMA algorithm on FPGA and CPU. The paper also demonstrates the potential of utilising FPGAs, GPUs, and CPUs concurrently in a heterogeneous computing environment to solve linear systems.

Conflicts of Interest

The editors declare that they have no conflicts of interest.

Acknowledgments

The editors wish to acknowledge the collaborative funding support from the UK EPSRC under grant P/L00058X/1.

Wim Vanderbauwhede
Sven-Bodo Scholz
Martin Margala

References

- [1] A. Fijany and C. P. Williams, “Quantum wavelet transforms: fast algorithms and complete circuits,” in *Proceedings of the NASA International Conference on Quantum Computing and Quantum Communications*, pp. 10–33, Springer, Palm Springs, CA, USA, February 1998.
- [2] S. Rivas-Gomez, A. J. Pena, D. Moloney, E. Laure, and S. Markidis, “Exploring the vision processing unit as co-processor for inference,” in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 589–598, IEEE, Vancouver, Canada, May 2018.
- [3] C. C. K. Mikkelsen and M. Manguoglu, “Analysis of the truncated spike algorithm,” *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 4, pp. 1500–1519, 2009.
- [4] A. Munshi, “The opencl specification,” in *Proceedings of the IEEE Hot Chips 21 Symposium (HCS)*, pp. 1–314, IEEE, Stanford, CA, USA, August 2009.
- [5] D. J. Warne, N. A. Kelson, and R. F. Hayward, “Comparison of high level fpga hardware design for solving tri-diagonal linear systems,” *Procedia Computer Science*, vol. 29, pp. 95–101, 2014.

Research Article

Automatic Pipelining and Vectorization of Scientific Code for FPGAs

Syed Waqar Nabi  and Wim Vanderbauwhede 

School of Computing Science, University of Glasgow, Glasgow, UK

Correspondence should be addressed to Syed Waqar Nabi; syed.nabi@glasgow.ac.uk

Received 4 May 2019; Revised 4 August 2019; Accepted 8 October 2019; Published 18 November 2019

Academic Editor: John Kalomiros

Copyright © 2019 Syed Waqar Nabi and Wim Vanderbauwhede. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

There is a large body of legacy scientific code in use today that could benefit from execution on accelerator devices like GPUs and FPGAs. Manual translation of such legacy code into device-specific parallel code requires significant manual effort and is a major obstacle to wider FPGA adoption. We are developing an automated optimizing compiler TyTra to overcome this obstacle. The TyTra flow aims to compile legacy Fortran code automatically for FPGA-based acceleration, while applying suitable optimizations. We present the flow with a focus on two key optimizations, automatic *pipelining* and *vectorization*. Our compiler frontend extracts patterns from legacy Fortran code that can be pipelined and vectorized. The backend first creates fine and coarse-grained pipelines and then automatically vectorizes both the memory access and the datapath based on a cost model, generating an OpenCL-HDL hybrid working solution for FPGA targets on the Amazon cloud. Our results show up to 4.2× performance improvement over baseline OpenCL code.

1. Introduction

Acceleration devices for high-performance computing (HPC) and scientific computing are becoming increasingly heterogeneous. There is a general consensus that no single type of device—CPU, GPU, or FPGA—will be best suited across the entire range of scientific applications. GPUs are already well-established as a practical alternative to conventional CPUs for accelerating scientific applications. A considerable proportion of supercomputers in the top 500 list contains GPU accelerators. FPGAs are a more recent addition to this canvas, and in spite of significant improvements in their performance and programmability in recent years, they are still far from widespread adoption as mainstream acceleration devices.

A key challenge that applies in a lesser or greater extent to all accelerators is writing parallel, high-performance code customized for performance specifically on that device. The challenge is all the more acute for FPGAs, which are notoriously difficult to program. Improvements in FPGA logic capacity as well as high-level synthesis (HLS) programming

frameworks such as Altera's (Intel's) AOCL, Xilinx's SDAccel, and Maxeler have played an important role in their transition from peripheral, embedded, or prototyping only devices to first-order desktop accelerators. However, FPGAs have failed to make the kind of inroads in HPC that GPUs have made. This is, in part at least, due to the fact that until very recently there were no practical high-level programming platforms for FPGAs, and even with their introduction, it is still a challenging task to write high-performance code. While heterogeneous programming languages like OpenCL provide *code portability*, they are not *performance portable* across devices. For example, [1] report that “even though OpenCL is functionally portable across devices, direct ports of GPU-optimized code do not perform well compared with kernels optimized with FPGA-specific techniques such as sliding windows. However, by exploiting FPGA-specific optimizations, it is possible to achieve up to 3.4x better power efficiency. . . .” Our own previous work has shown that even very simple OpenCL kernel code can lead to very different performance profiles when moving from one FPGA framework to another [2].

It is our contention that such programming and optimization challenges will remain a hurdle to the adoption of acceleration devices—especially FPGAs—in mainstream HPC, and that high-level programming frameworks like OpenCL should themselves be targets for still higher level compilers that can work with sequential, unoptimized legacy code, in which case one could truly have *performance portability*.

We propose an optimizing compiler framework that uses *Type Transformations* (TyTra) to explore FPGA-specific optimizations for a given application and automatically generates the implementation code from legacy Fortran code, leading to the desired code and performance portability. It applies these type transformations on a high-level, functional representation of the kernel (the code to be accelerated on the FPGA), extracted from the Fortran code and then uses a cost model for evaluating the search-space for an optimized solution.

A flowchart of the TyTra framework is shown in Figure 1. The frontend refactor Fortran 77 codes into modern, maintainable, extensible, and accelerator-ready Fortran code (available at <https://github.com/wimvanderbauwhede/RefactorF4Acc>). We can then generate OpenCL code targeted at GPUs (available at <https://github.com/wimvanderbauwhede/AutoParallel-Fortran>) or FPGAs (available at <https://github.com/wimvanderbauwhede/Fortran-to-OpenCL-FGPA>). For FPGA targets specifically, we invoke a more involved optimization pass that translates the OpenCL code to an intermediate representation (IR) (available at <https://github.com/waqarnabi/ocl2tir>) and then generates low-level hardware description language (HDL) code from it. It is this optimizing route in our flow that we will discuss in the paper, with a focus on two key optimizations: pipelining and vectorization.

We briefly discuss the frontend of the TyTra flow in Section 4. We present our view of the requirements of optimizing code on FPGAs in Section 5, leading into the main contribution of this paper in Section 6, the TyTra backend compiler (TyBEC) and the automatic pipelining and vectorization it enables. We present the evaluation of our approach in the next section before concluding the paper. We start by reviewing some related work.

2. Related Work

The viability of FPGAs as *mainstream* acceleration devices for scientific computing is well established in literature. As an example, reference [3] presents a suitability analysis of FPGAs for heterogeneous HPC platforms, using the Berkeley 13 dwarfs as a reference. They found FPGAs to be suitable for 5 of the 13 dwarves, but noted that they are difficult to use for nonspecialized designers and emphasized the importance of more abstraction and less customization. Reference [4] demonstrates the suitability of FPGAs, programmed via OpenCL, for implementing partial differential equation (PDE) based scientific models. The same article, however, showing an OpenCL kernel written with differences in syntax and compiler hints for two different FPGA target devices/vendors, supports our contention that OpenCL code is not performance portable.

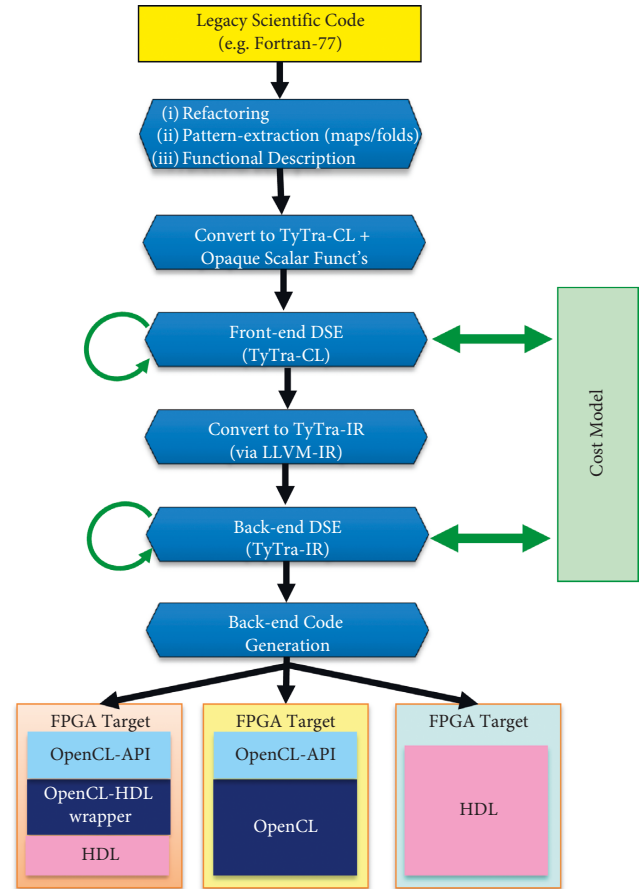


FIGURE 1: The TyTra optimizing compiler framework. The starting point is Fortran 77 scientific code, though there can be other possible entry points as well. There are a number of backend code-generation options, with this paper’s focus on hybrid OpenCL-HDL route, geared towards deployment on the Amazon cloud’s F1 instances.

There are a number of commercial tools available that provide a high-level programming route for accelerating scientific code on FPGAs. Maxeler [5] is a good example, which provides a Java metaprogramming model for describing computation kernels and connecting data streams between them. It has been used for accelerating applications from various domains, e.g., scientific and financial modeling. Altera (now Intel) OpenCL or AOCL [6] is the implementation of the OpenCL heterogeneous parallel programming framework for Intel FPGAs. While it is based on the OpenCL standard, it has vendor-specific optimization extensions. Xilinx similarly has its own OpenCL implementation called SDAccel [7]. Like AOCL, SDAccel is based on the OpenCL standard and also proposes custom optimizations to improve performance.

There have been other studies that are motivated in a way similar to ours, that is, by the need for a higher abstraction design entry than conventional high-level languages. For example, algorithmic skeletons have been proposed to separate algorithm from architecture-specific parallel programming [8]. SparkCL is an attempt to bring increasingly diverse architectures, including FPGAs, into the familiar

Apache Spark framework [9]. Another route for increasing the design abstraction is to use domain-specific languages (DSLs), and there are numerous examples for FPGAs, e.g., *FSMLanguage* for designing FSMs [10] and *CLICK* for networking applications [11].

A work that is quite similar to ours is the *Geometry of Synthesis* project [12]. It proposes design entry in a functional language paradigm, leading to generation of RTL code for FPGAs. It does not have automatic generation and evaluation of architectural design variants though. There has also been work on exploring vectorization for FPGA pipelines for specific applications (for example, see [13]). Automatic pipelining of high-level code is now possible with commercial tools like Xilinx's SDAccel, Intel's AOCL, and Maxeler, and some tools allow vectorization as well, though it has to be manually programmed or hinted via pragmas. Automatic pipelining and vectorization of high-level code based on a cost model, one that can work with legacy Fortran code, is entirely novel as best as we know.

While not the focus of this paper, a key first step of our flow is refactoring legacy Fortran 77 code to make it more *accelerator friendly*. There are a number of similar refactoring tools available for Fortran, though Fortran 77 is supported by very few. ROSE framework (<http://www.rosecompiler.org/index.html>) from LLNL [14] is probably the most well known, which relies on the Open Fortran Parser (OFP) (<http://fortran-parser.sourceforge.net/>). This parser claims to support the Fortran 2008 standard. Furthermore, there is the language-Fortran (<https://hackage.haskell.org/package/language-fortran>) parser which claims to support Fortran 77 to Fortran 2003. A refactoring framework which claims to support Fortran 77 is CamFort [15]; according to its documentation, it supports Fortran 66, 77, and 90 with various legacy extensions. An eclipse-based interactive refactoring tool Photran [16] supports FORTRAN 77-2008.

3. TyTra Frontend

We will briefly discuss this frontend of the TyTra flow here for completeness; more details are in [17], where we assess its correctness, completeness, and capability.

FORTTRAN 77 can be both computationally efficient as well as programmer efficient, allowing the programmer to write code quickly without being too strict about it. However, it becomes very difficult to maintain and port very quickly as a result of this. We aim to make our refactored code modern, maintainable, and extensible.

The requirements in mind were very different compared with today's languages when FORTRAN 77 was designed, especially in terms of avoiding bugs. Some specific features, now unacceptable in modern languages, are: implicit typing, subroutine arguments intended access absent, and absence of a module system, required both for extensibility and maintainability.

The TyTra frontend compiler converts all nonprogram code units into modules. These are then used with an explicit export (only) declaration. There are many more refactorings applied by our compiler, such as rewriting label-based loops as do-loops etc.

The common feature of the vast majority of current accelerators is that they have a separate memory space, usually physically separate from the host memory. Furthermore, the common offload model is to create a "kernel" subroutine (either explicitly or implicitly) which is run on the accelerator device.

It is extremely important to separate the memory-spaces of the host and the kernel when generating code for modern accelerators. Since Fortran 77 uses global variables liberally, our frontend converts them to subroutine arguments across the entire call tree of the program.

Our goal is to convert legacy Fortran 77 code into parallel code so that the computation can be accelerated on FPGAs. We use a three-step process.

First, the above refactorings give us a modern, maintainable, extensible, and accelerator-ready Fortran 95 codebase. This gives an excellent starting point for many of the other existing tools, e.g., the generated code can be conveniently parallelized using OpenMP or OpenACC annotations. We want, however, to provide an end-to-end solution to the user that does not require any annotations.

The next step in our process is identifying data-level parallelism present in the code in the form of *maps* and *folds*. The terms, *map* and *fold*, are from the functional programming domain and refer to ways of performing an operation on all elements of a list. These constructs are broadly equivalent to loop nests with and without dependencies, and as Fortran is loop-based, our analysis is actually an analysis of loops and dependencies. Internally, though our representation uses the functional programming model where *map* and *fold* are higher-order functions (functions operating on other functions), extracted from the bodies of the loops; we thus raise the abstraction level of our representation, making it independent of both the original code and the final code to be generated. We then apply a number of rewrite rules for map and fold based functional programs (broadly speaking equivalent to loop fusion or fission) to optimise the code.

The third step involves the backend of our framework, the focus of the rest of this paper, where we use patterns extracted to ensure that our kernels are guaranteed to be both pipelineable and vectorizable. Then, using our cost model, we generate optimized, synthesizable code for Xilinx FPGAs on the Amazon cloud F1 instances.

4. Transforming Scientific Applications for Performance on FPGAs: A Perspective

The potential to get good performance and energy efficiency on FPGAs is widely recognized, but coupled with the realization that achieving the potential is not trivial. It is our view that, in the context of the domain of scientific computing, the guidelines for creating architectures on FPGAs that give high throughput can be summed up as follows:

- (1) Create a deep and custom pipeline, fine-grained as well as coarse-grained
- (2) Coalesce (vectorize) global memory accesses
- (3) Vectorize the pipelined datapath

- (4) Minimize data stalls on these pipelines
- (5) Maximize throughput by replicating appropriate functional units
- (6) Minimize random access to external memory by optimizing stencil computations
- (7) Use optimized numerics where possible
- (8) Use vendor-optimizations where suitable

The design-space exploration (DSE) in TyTra is informed by the view summarized above and is carried out at two abstractions: the *frontend* and the *backend*. In this work, we highlight the backend, specifically two key optimizations: pipelining and vectorization (points 1–3).

4.1. Pipelining. FPGAs consist of a fine-grained reconfigurable fabric that can be customized for a given application. If the underlying application is amenable to pipeline parallelism, then high throughput on FPGAs can be achieved by creating deep, custom pipelines. Pipelining can be done at three hierarchical levels.

4.1.1. Pipelining inside Instructions. This refers to atomic datapath instructions like floating point operations that require multiple clock cycles to complete. It is important to pipeline them to achieve a high operational frequency on the FPGA and also to maintain throughput. Such pipelined functional units are available both in the academia and via vendor tools. For example, we use *FloPoCo* [18], an open-source tool, to generate pipelined functional units for our solution.

4.1.2. Pipelining across Instructions. To get good performance, pipelining instructions inside computation kernels is crucial. This requires a data-dependence analysis to ensure data hazards are avoided. We refer to such pipelines in this work as *fine-grained* pipelining.

4.1.3. Pipelining across Kernels. While most FPGA HLS tools would automatically pipeline at the fine-grained level, it is important to pipeline at this *coarse-grained* level as well to get viable performance on FPGAs for large scientific problems. Such pipelining obviates the need of using the FPGA external DRAM (i.e., *global memory* in OpenCL terminology) to communicate between kernels.

4.2. Vectorization. To optimize throughput and utilize a target device at its maximum or near-maximum potential, the external memory interface should ideally be operating at or near saturation. This is typically achieved by *coalescing* access to memory [19] and/or using multiple memory banks concurrently [13]. The purpose is to read multiple array indices concurrently as a single, wider data word. This coalescing can also be called *vectorizing* the memory access. In the previous work [2], we adapted a synthetic memory performance benchmark to show the sustained bandwidth to global memory of various devices (benchmark available at

<https://github.com/waqarnabi/mp-stream>), and we showed that vectorization achieved up to 8.5× memory bandwidth increase over the baseline. For any application that is *memory-bound*, this improvement in memory bandwidth will translate to an improved throughput for the overall application. To complement the vectorized memory access, the application’s datapath can also be vectorized, though one must ensure data hazards are avoided.

We will now discuss pipelining and vectorization optimizations in the context of the TyTra backend.

5. TyTra Backend (TyBEC) Compiler

The TyTra backend is designed to be compatible with a variety of frontend entry routes and is composed of a custom intermediate representation (IR) language, a parser, a scheduler, a cost/performance model, and finally an FPGA code generator.

5.1. The TyTra Intermediate Language. The TyTra Intermediate Language (TIR) is the interface provided by the TyTra backend, to which a number of possible frontends can be coupled, one of which was shown in Figure 1. However, for the purpose of this discussion, how one arrives at a TIR description of the problem is not relevant.

The TIR description of a problem lies halfway between the frontend and backend optimizations that together seek to identify the optimal design variant. *Optimal* in this context means that the kernel has been pipelined, ideally to achieve a throughput of one cycle per output, and then vectorized to go beyond this throughput until we either saturate the memory bandwidth (memory wall) or run out of FPGA resources (compute wall). There are some optimizing transformations that take place in the frontend DSE phase. These transformations relate primarily to finding specific computation patterns such as maps and folds and connecting them in a coarse-grained dataflow graph. Once the design variant generated by the frontend has been specified in the TIR, the backend optimizations can be applied, and FPGA implementation code can be generated. This context informs the underlying model of computation and specific requirements for our custom IR.

5.1.1. Model of Computation: Kahn Process Networks. The TIR syntax is quite similar to the LLVM-IR; however, its underlying model of computation is entirely different, as it models a *dataflow* machine. The Kahn process network (KPN) is a suitable abstraction to use, though we apply some additional requirements and constraints on it.

KPNs were first introduced by Gilles Kahn when presenting a simple language for parallel programming [20]. The use of the KPN abstraction for modelling architectures for FPGAs is not a novel concept (e.g., see [21, 22]). The key features of this abstraction, which make it very suitable for use as the underlying model for our IR, are as follows (direct quotes in the list are from [20]):

- (i) Processes (or nodes) communicate via *unbounded* first-in first-out (FIFO) queues.
- (ii) All processes run forever.
- (iii) If any process were to stop for an external reason, the whole system will stop.
- (iv) Communication lines (or edges or channels) are the only way in which processes communicate.
- (v) The time taken by edges to transmit information can be unpredictable, but always finite.
- (vi) At any given time, a process is either computing or waiting for data on one of its input edges.
- (vii) A process can have its own memory or state, so it is a “function from the histories of its input lines to the histories of its output lines.” This means nodes that fold (or reduce) information are possible.
- (viii) Writes to a channel are *nonblocking*, while reads are *blocking*.
- (ix) KPN process is *monotonic*: it “need not have all of its inputs to start computing, since future input concerns only future output.” Monotonicity allows pipeline parallelism.

While these features of the KPN make it a suitable abstraction for our purpose of modelling pipeline parallelism on FPGAs, they cannot be used as is. The key departure required from this abstraction is replacing *unbounded* FIFOs (which are not possible in a real system) with *bounded* FIFOs. Additionally, nonblocking writes are not suitable with finite FIFOs, and we introduce blocking writes. However, the introduction of blocking writes and finite FIFOs can lead to deadlocks unless safe bounds for the sizes of the FIFOs are derived properly [23]. We derive safe FIFO bounds by statically scheduling all the nodes in our dataflow graph discussed in more detail later.

5.1.2. Syntax and Semantics of the TIR. The TIR is strongly and statically typed, and all computations are expressed using Static Single Assignment (SSA). The syntax is based on the LLVM-IR, but the semantics are built on top of a *streaming* paradigm, suitable for inferring pipelined datapaths on an FPGA target.

Static typing is a requirement for synthesizing an FPGA design at compile time. Strong and static typing together provide the basis for our static cost model that underpins the TyTra flow. The FPGA code-generator too requires explicit typing. TIR datatypes are mapped to LLVM datatypes wherever possible and follow the same general scheme for naming datatypes. However, LLVM-IR does not differentiate between signed and unsigned data which are required for TIR. Also, TIR allows custom and nonstandard datatypes, which in fact is one of the reasons for creating our own IR. TIR supports arrays and vectors, again following the syntax of LLVM-IR.

A design is constructed by creating a hierarchy of IR *functions*, which may be considered equivalent to *modules* in an HDL like Verilog. However, these functions are described

at a higher abstraction than the *register transfer* level typical for an HDL. The TyTra backend parses the TIR description and extracts a dataflow architecture from it.

The TIR is neither a subset nor a superset of the LLVM-IR, but an independent language that is inspired by it. There are many features of LLVM-IR that TIR does not support, and at the same time, there are a number of extensions in the TIR that are alien to LLVM-IR. Under the hood, these languages have a fundamentally different view of the machine as discussed earlier, which is eventually the fundamental difference. Other ways in which TIR departs from the LLVM-IR semantics are: all variables are data *streams*, arguments represent ports for connectivity between peer or parent-child functions, there is custom instruction for *splitting* and *merging* nodes, there is a specialized syntax for creating *offset* streams for stencils, we can have custom datatypes with nonstandard widths, and there is an extended syntax to express the creation and consumption of data streams from/to memory. Further discussion of the TIR’s syntax and semantics is outside the scope of this paper, but interested readers can refer to [24]. We also show TIR for two example problems later in Section 6.

5.2. Scheduling and Pipelining. Recall from the previous section that our extension to the KPN requires us to have bounded FIFOs, which are large enough to ensure there is no deadlock in the presence of blocking writes. Hence, although the hardware realization of our nodes is based on asynchronous hand-shaking with back-pressure (based on the AXI-Stream protocol), we do need to determine safe bounds for all inferred buffers. This is why static scheduling of the dataflow graph (DFG) is essential. Moreover, the TyTra flow is predicated on the availability of a performance model that can predict the latency and throughput of each node, which too requires a static scheduler, even if in practice there is no centralized scheduling controller in the synthesized FPGA circuit.

The scheduling algorithm of TyBEC is based on the KPN model of computation. The SSA syntax of the TIR lends itself to a straightforward extraction of the pipelined dataflow from primitive instructions in *leaf* functions. We have adopted an *As Soon As Possible* (ASAP) algorithm for scheduling the instructions. This can be suboptimal in terms of resource usage [25]. More sophisticated algorithms are possible, which exploit reuse of functional units across instructions to save resources. This however would require a fundamental change in the architecture from the current one with a distributed scheduling mechanism, to a centralized one where a controller would orchestrate the reuse of resources. This is a line of investigation we mean to pursue in the future.

Functions can be hierarchical as well, reflected by hierarchical nodes in the DFG. Each hierarchy is captured by the same extended KPN model that we have discussed earlier. The resultant DFG translates to dataflow *pipelines* on the FPGA, which are reflected in the generated HDL. This pipeline parallelism is, as we discussed earlier, an essential component of the FPGA-oriented optimizations we wish to

apply. Both the coarse-grained pipelining of kernels (hierarchical nodes) and fine-grained pipelining of instructions (leaf nodes) are achieved by this hierarchical scheduling process, illustrated in Figure 2. This aspect of our work may be contrasted with tools like SDAccel, where coarse-grained pipelines across kernels have to be explicitly modelled using OpenCL *pipe* semantics.

A finer level of pipelining, where we pipeline multicycle primitive instructions like floating point operations, is achieved by using pipelined functional units generated from the FloPoCo tool [18].

5.3. Vectorization. Vectorization is a well explored optimization for improving performance and has been explored for FPGAs as well. This is similar to, though not exactly the same as, the vectorization optimization in CPUs. The way we use this term, “vectorization” refers to *both* vectorization of the datapath, and the coalescing of memory accesses. In the TyTra backend, once the kernels have been pipelined, the design can be vectorized automatically. Memory-access vectorization results in coalesced transactions, which can help achieve operation at or near memory bandwidth saturation, as we have shown in our earlier work [2]. Vectorized datapath ensures that the overall throughput is not limited by the computation. Together, these automatic vectorization operations enable our backend to push the solution closer to both the memory and computation limits of the device, which is where we ideally want to be. The key novelty in our flow is the complete automation of the process of vectorizing the memory access and the datapath.

5.3.1. Automatic Detection of Vectorizable Loops in Serial Code. Vectorizing a loop by a factor N_V implies concurrent execution of N_V iterations of the loop. Loops can be vectorized only if they have been unrolled, and if there are no *loop-carried dependencies* with reaches smaller than the vectorization width N_V . The TyTra flow is based on extracting *map* and *fold* patterns from serial scientific code, as shown in Figure 1 and then *scalarizing* them before passing them on to the backend. *Scalarization* here implies replacing index-based array accesses with scalar variables, effectively subsuming loops that iterate over arrays. Since the (ostensibly) scalar variables actually refer to data streams, the semantics of the program are preserved, though that does require some meta-information to be carried through by the compiler, e.g., the size of the loops. An example of this transformation in Figure 3 shows two versions of an `update()` function from a scientific model. The code on the left is a conventional loop-based function, converted by our frontend (while we show the loop-based function written in C to emphasize the transformation, the frontend actually uses loop-based Fortran code as the source and emits scalarized C as shown.) to the scalarized version.

These transformations convert loops iterating over arrays to *scalar* kernels operating on *streams*. The advantage of this frontend transformation is that we get vectorization opportunities at the backend for free. Loop-unrolling is no longer required, as the frontend has ported the code into the

streaming dataflow domain. More importantly, the frontend transformations guarantee that the kernels exposed to the backend do not have any loop-carried dependencies and thus are vectorizable to arbitrary widths. The TyTra flow is in fact conservative in exposing vectorization opportunities. This is because our flow only vectorizes *mappable* loops that have no loop-carried dependencies, whereas vectorization is possible even if they are there, as long as their reach is larger than the vector width.

5.3.2. Using a Cost Model for Identifying the Optimum Vector Width. The vectorization of kernels results in additional resource usage on the FPGA. By using our cost model, we can estimate the resource usage for different vectorization options. Then, we limit the vector width to the maximum that we can fit within the target FPGA’s resources. Currently, our backend supports vector widths of 1, 2, 4, 8, and 16 (for OpenCL compatibility), but there is no innate reason for limiting our flow to these vectorization factors.

The cost model is discussed in detail in [24], but we present its brief outline here. Figure 4 shows how the cost model is used in our flow. The TIR description of the design variants is fed into the cost model, along with a description of the target. This description is in the form of its available resource, memory bandwidth profile, and the cost of various primitive instructions on that device. The cost model then accumulates the resource requirements for the entire device and also estimates its performance after scheduling all instructions and functions. It then estimates the performance of all variants (in case of the examples in this paper, variants are generated by varying the vectorization factor) and generates OCL-HDL hybrid code for the chosen one.

5.4. Generating OpenCL-HDL Hybrid Implementation for FPGA (F1) Instances on the Amazon Cloud. We considered a number of options for implementing the design generated by the TyTra flow on an FPGA, as shown in Figure 1, finally converging on an OpenCL-HDL hybrid for F1 instances on the Amazon cloud. Commercial vendors like Xilinx, Intel, and Maxeler all provide such a hybrid programming route. The HLS abstraction can be used to handle the *shell* logic conveniently, and kernel datapaths can be expressed at a lower abstraction (register-transfer level or *RTL*), e.g., in Verilog HDL. We avoid the need to generate complex RTL code for shell logic yet maintain much more control over optimizations for the kernel pipeline. Our hybrid approach is more amenable to performance and cost prediction than HLS-only routes. The generated kernel pipeline is more performance portable across FPGA vendor tools and devices, as no vendor-specific pragmas and optimizations are used. We do need to generate vendor-specific shell code, but that follows a standard template with little variation across designs.

Amazon’s EC2 F1 instances on the cloud provide a suitable way of accessing the latest FPGA hardware as well as tools [26]. *FPGA AMI* machine images are available, which come prebuilt with FPGA development and runtime tools based on Xilinx’s SDAccel and Vivado frameworks. These

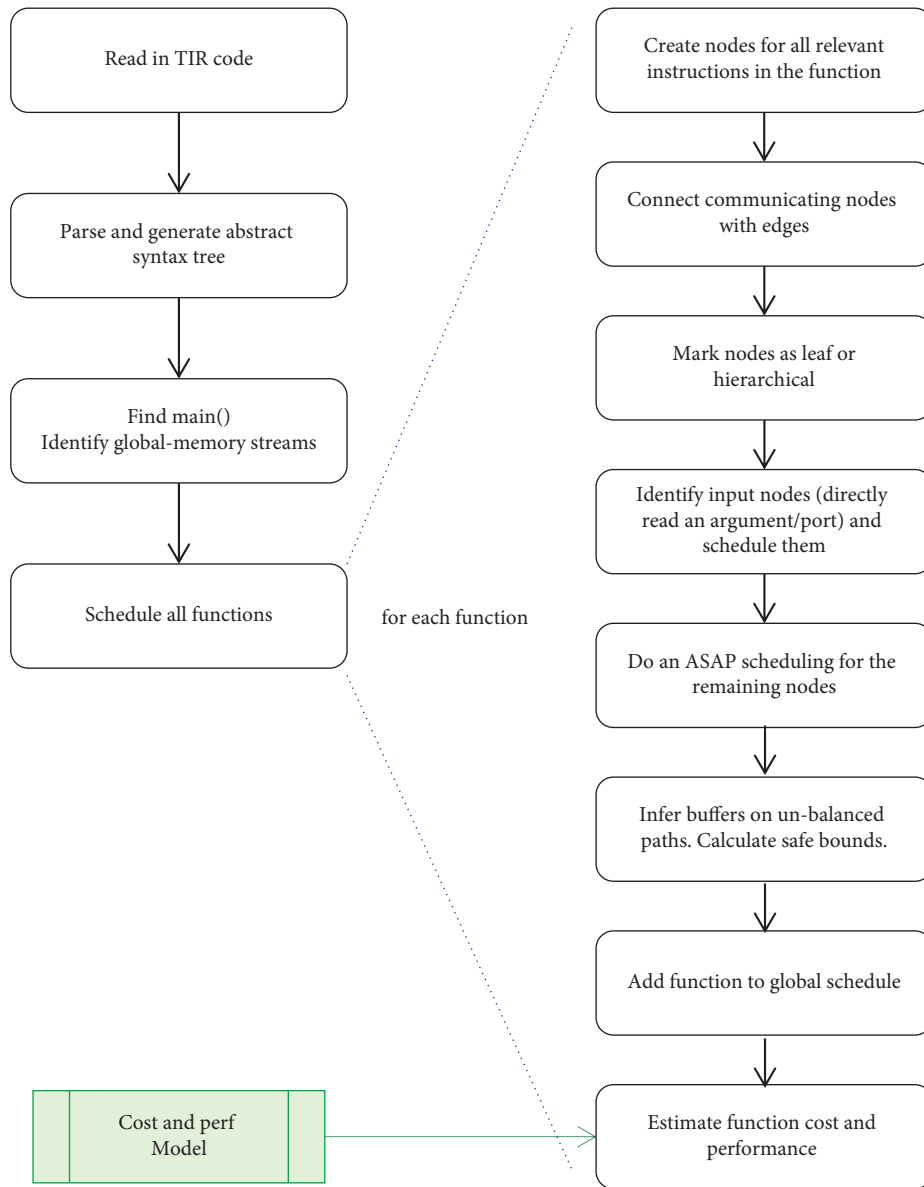


FIGURE 2: The TyTra backend scheduler. It reads in the TIR description of the problem, which has a syntax similar to LLVM-IR’s, using the SSA (single static assignment) format. The output of the scheduler is the dataflow graph of the problem, with buffers inferred if needed (e.g., see Figures 6 and 9), which is then used to estimate performance, as well as generate synthesizable Verilog HDL.

```

/* The original function with loops */
void updates ( host_t *h, host_t *hzero, ...){
  for (int j=0; j<= ROWS-1; j++) {
    for (int k=0; k<=COLS-1; k++) {
      h[j*COLS + k] = hzero[j*COLS + k]
        + eta [j*COLS + k];
      wet[j*COLS + k] = 1;
      if ( h[j*COLS + k] < hmin )
        wet[j*COLS + k] = 0;
      u[j*COLS + k] = un[j*COLS + k];
      v[j*COLS + k] = vn[j*COLS + k];
    }
  }
}

/* The "scalarized" function with streams */
void update_map_24 (float *h_j_k, float hzero_j_k, ...){
  *h_j_k = hzero_j_k+eta_j_k;
  *wet_j_k = 1;
  if ((*h_j_k)<hmin) *wet_j_k = 0;
  *u_j_k = un_j_k;
  *v_j_k = vn_j_k;
}
  
```

FIGURE 3: An illustration showing conversion of a loop-based function (left) to its *scalarized* version (right), where the “scalars” are effectively *streams* of data. Metainformation extracted by the compiler, e.g., the sizes of the streams, is carried through to the backend in order to preserve semantics.

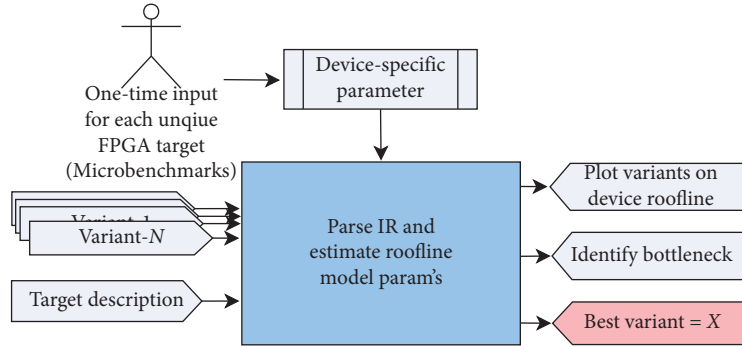


FIGURE 4: The use-case of the cost model that is integrated inside the TyTra flow’s backend. It is used to estimate the resource usage and performance of variants being explored in the design-space.

tools provide various design-entry options. From our vantage point, the utility of these platforms is that we can work with the latest hardware and tool versions, and we can experiment with our hybrid HLS(OpenCL)–HDL(Verilog) flow.

To integrate HDL kernels with the shell logic provided by Xilinx’s SDAccel tool, they need to be compatible with the AXI protocol: AXI4 for DDR read and write controllers; AXI4-Stream for transferring data streams to and from these controllers and also peer communication; and AXI4-Lite for control information exchange with the host [27]. The DDR (AXI4) and control (AXI4-Lite) interface logic is incorporated by using template code provided by SDAccel. This reduces the kernel pipeline compatibility requirement to the AXI4-Stream protocol for interfacing with the memory controllers. Figure 5 demonstrates this setup.

6. Evaluation

We evaluate our approach with two examples, first through a synthetic barebones kernel and then on a scientific code simulating the *Coriolis* force.

6.1. A Synthetic Example. The simple, contrived example creates a coarse-grained pipeline with integer arithmetic operations. This translates to a single cycle throughput, and single path dataflow. The TIR and DFG of this problem are shown in Figure 6. Note that the DFG is generated automatically as part of the backend scheduling and code-generation. HDL code is also generated by the backend. It can be viewed by running the backend on the TIR (the prototype TyTra backend compiler is available at <https://github.com/waqarnabi/tybec>).

This illustrative example highlights the backend automatic pipelining and vectorization optimizations that are the focus of this work. We generated code for all vector widths supported by our flow, in order to demonstrate the effect of vectorization in this paper. In practice, we would follow the following simple algorithm for converging on the vector width:

- (1) Create a design variant with the maximum allowable vector width (currently 16 words)

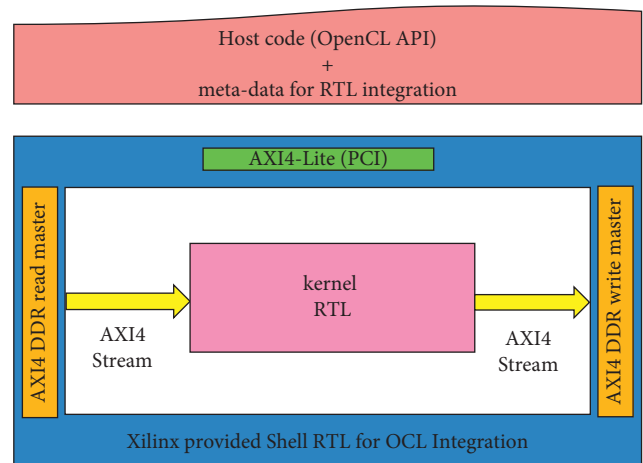


FIGURE 5: The hybrid OpenCL-HDL code-generation setup. TyTra generated kernel pipeline design in Verilog HDL is integrated with the OpenCL-based SDAccel framework, using AXI protocols.

- (2) Estimate resource utilization
- (3) If estimated resources are more than available on target FPGA, step down to the next available vector width
- (4) Repeat 1–3 until design is predicted to fit by the cost model (we aim to use less than 80% of target FPGA resources, as in our experience, beyond this threshold designs typically fail to synthesize)

The results of our experiments are shown in Figure 7. Since this is a small example, the maximum available vectorization factor of 16 was possible within the available resources, and that is the variant selected. Our results show an almost 4.2× speedup over the scalar baseline for the maximum vectorization. The speedup is sub-linear and indicates a memory bottleneck. This bottleneck could be mitigated by using multiple memory banks if available.

The corresponding resource trade-off can be seen in Figure 8. Other than DSP units, all resources show sublinear scaling due to the almost uniform usage of logic in the *shell* of the design across all variants.

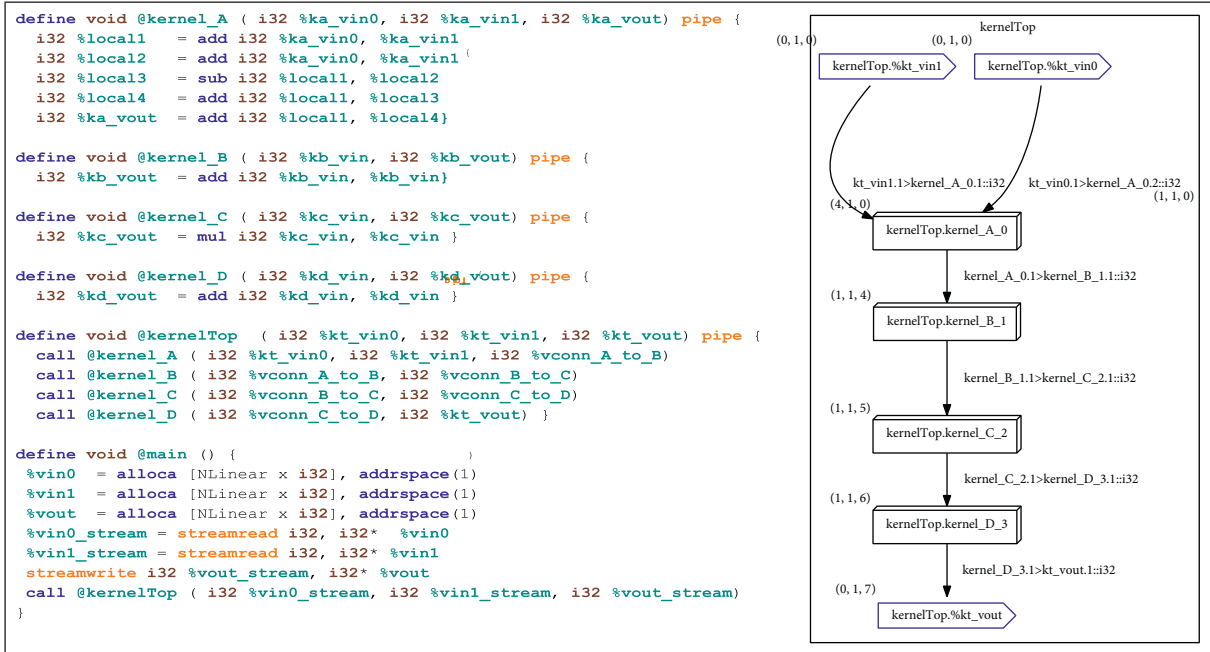


FIGURE 6: TIR code and DFG of the synthetic example. The TIR shows 4 kernels connected in a coarse-grained pipeline in a *top* kernel, which is connected to global memory streams in the *main* function. The DFG is generated by the backend from the TIR, and only the top-level kernel is shown here. The tuple of three integers with each node is the scheduling parameters (latency, firing-interval, and start-delay) inferred from the code and used by the backend for scheduling and RTL code generation.

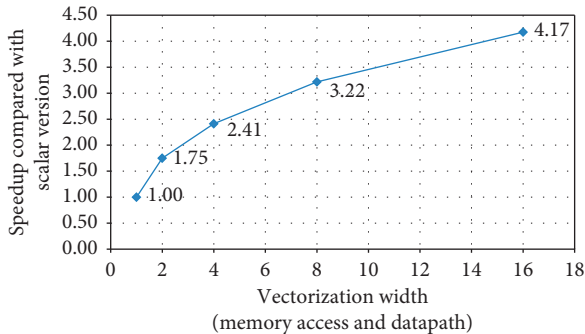


FIGURE 7: Speedup achieved over nonvectorized OpenCL baseline for various vector widths, for the first example. The TyTra solutions are OpenCL-HDL hybrids, and the complete host API, shell, and kernel code for all variants is generated automatically from TIR description.

6.2. *Simulating the Coriolis Force.* This second example is based on Fortran code for modelling the Coriolis force that accompanies a text on ocean modelling [28]. The code predicts the pathway of nonbuoyant fluid parcels in a rotating fluid subject to the Coriolis force. The kernel is computed over a two-dimensional grid for a certain number of time steps. At each time step, the kernel reads the velocities and positions of each grid point and updates them. That is, at each time step, it reads 4 floating point numbers and writes 4 floating point numbers. The Fortran code is shown in Figure 9. The equivalent TyTra-IR code and dataflow graph (top kernel only) as generated by the TyTra backend are shown in Figures 10 and 11, respectively.

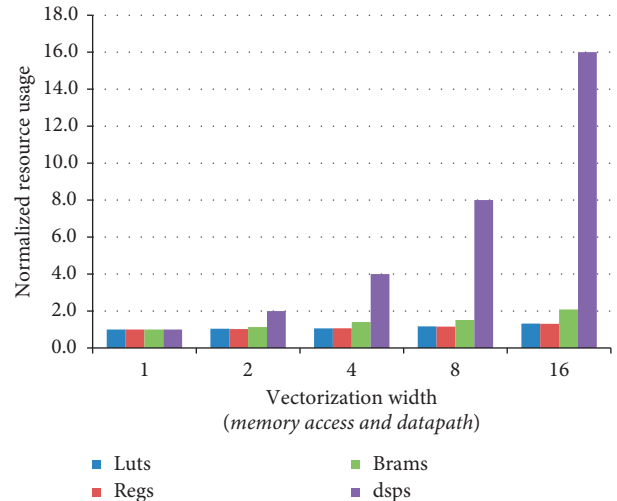


FIGURE 8: Resource usage for various OpenCL-HDL hybrid vectorized versions, normalized against resource usage for non-vectorized OpenCL baseline for the first example.

The TyTra backend generates a baseline RTL (and OpenCL wrappers) that is pipelined and without vectorization. It then generates vectorized versions as well, as long as the predicted cost fits in the target device. In this example as well, like the previous one, all possible vectorization factors up to 16 can be accommodated. The resource cost prediction of one design variant takes in the order of 0.1 seconds. This makes the design-space exploration fairly quick when we consider the vectorization optimization in isolation as there are a limited number of possible variants.

```

!time loop
DO n = 1,ntot
  !space loop
  DO i = 1, maxX*maxY
    ! velocity predictor
    un(i) = (u(i)*(1-beta)+alpha*v(i))/(1+beta)
    vn(i) = (v(i)*(1-beta)-alpha*u(i))/(1+beta)

    ! predictor of new location
    xn(i) = x(i) + dt*un(i)/1000
    yn(i) = y(i) + dt*vn(i)/1000

    ! updates for next time step
    u(i) = un(i)
    v(i) = vn(i)
    x(i) = xn(i)
    y(i) = yn(i)
  END DO
END DO

```

FIGURE 9: Fortran code of the Coriolis example.

```

!--velocity update kernel
define void @coriolis_ker0 ( data_t %u, data_t %v, data_t %un, data_t %vn ) pipe {
  data_t %mul = fmul data_t %u, ONE_MINUS_BETA
  data_t %mul1 = fmul data_t %v, ALPHA
  data_t %add = fadd data_t %mul, %mul1
  data_t %un = fdiv data_t %add, ONE_PLUS_BETA
  data_t %mul4 = fmul data_t %v, ONE_MINUS_BETA
  data_t %mul5 = fmul data_t %u, ALPHA
  data_t %sub6 = fsub data_t %mul4, %mul5
  data_t %vn = fdiv data_t %sub6, ONE_PLUS_BETA
}

!--position update kernels
define void @coriolis_ker1_subker0 ( data_t %x, data_t %un, data_t %xn ) pipe {
  data_t %mul = fmul data_t %un, DT
  data_t %div = fdiv data_t %mul, 1000.0
  data_t %xn = fadd data_t %x, %div
}

define void @coriolis_ker1_subker1 ( data_t %y, data_t %vn, data_t %yn ) pipe {
  data_t %mul1 = fmul data_t %vn, DT
  data_t %div2 = fdiv data_t %mul1, 1000.0
  data_t %yn = fadd data_t %y, %div2
}

!--top kernel connecting the two sub-kernels
define void @kernel_top ( data_t %u, data_t %v, data_t %x, ... ) pipe {
  call @coriolis_ker0 {data_t %u, data_t %v, data_t %un_local, data_t %vn_local}
  call @coriolis_ker1_subker0 { data_t %x, data_t %un_local, data_t %xn }
  call @coriolis_ker1_subker1 { data_t %y, data_t %vn_local, data_t %yn }
  data_t %un = load data_t %un_local
  data_t %vn = load data_t %vn_local
}

!--main: declared device global (DRAM) memory arrays, defines streams to/from it
!-- and "calls" the top kernel
define void @main () {
  %u = alloca [SIZE x data_t], addressspace(1)
  ... --other device memory arrays
  %u_stream = streamof data_t, data_t* %u, !tir.stream.type !streamid, !tir.stream.size !SIZE
  ...--other input streams from device memory
  streamwrite data_t %un_stream, data_t* %un, !tir.stream.type !streamid, !tir.stream.size !SIZE
  ...--other output streams to device memory

  !--call the top level kernel and pass it the streams and the constant
  call @kernel_top { data_t %u_stream, data_t %v_stream, ... }
  ret void
}

```

FIGURE 10: TIR code of the Coriolis example. It shows 3 kernels connected in a coarse-grained pipeline in a *top* kernel, which is connected to global memory streams in the *main* function (not shown).

The actual performance for variants is shown in Figure 12. Note that there is an important difference between this and the first example. The first example had 2 integer inputs and 1 integer output. When the inputs, a total of 64 bits, were vectorized by the maximum factor of 16, it was still within maximum data width allowed by SDAccel (which is 1024 bits). This second example, however, had an input (and output) total width of 128 (32×4) bits, so it can only be vectorized up to a factor of 8, which is reflected in the results. In the future, we plan to incorporate multiple memory

interfaces and banks into our design, which would allow us to exploit this vectorization feature to its full potential.

An interesting observation here is that the performance peaks at $2.7\times$ baseline, at a vectorization factor of 4, with vectorization to a factor of 8 showing virtually no improvement. This example has a wider total input width of 128 (32×4) bits, as opposed to 64 bits for the previous one. Since the DDR memory bus for the target FPGA platform is 512 bits wide, it is saturated at a vectorization factor of 4 already. Using multiple concurrent memory interfaces and banks should allow us to go beyond this saturation limit.

Another observation is that the performance profile is virtually unchanged across different grid sizes and number of time steps. This shows performance gains of vectorization scaling well with the problem size.

The resource usage for all these variants is shown in Figure 13, which shows the expected increase for increasing the vectorization factor. Compared with the first example, this is a larger kernel with resource heavy floating point units, leading to the kernel having a proportionally larger share of the resources versus the shell logic. This is the reason vectorization scales up the resources much more than the first example.

7. Conclusion

FPGAs are fast-becoming mainstream accelerator devices for a variety of HPC applications. Writing optimized programs for FPGAs remains a challenge though, even with the availability of HLS tools. We are developing an optimizing compiler framework called *TyTra*, where we propose to use a combination of transformations and optimizations to automatically generate FPGA implementations from serial, legacy scientific codes. In this paper, we have presented two key optimizations that are part of this framework, pipelining and vectorization, the latter applied to both the external (DDR) memory accesses as well as the kernel datapath. We discussed briefly how we transform legacy serial Fortran code to kernels with *map* patterns, suitable for pipeline parallelism as well as vectorization. We highlighted our custom IR language-based backend that can be used to express the variants in our design space and which schedules computations based on an extended KPN-based machine model, finally emitting an OpenCL-HDL hybrid implementation. Evaluation of our approach on two examples showed performance gains between $2.7\times$ and $4.2\times$. Extending our solution to exploit multiple memory banks concurrently can be reasonably expected to achieve further performance gains.

Exploiting such vectorization opportunities when accelerating HPC code on FPGAs is essential; otherwise, we may be operating far below optimal performance. Our flow, because it is based on a sophisticated frontend analysis and a cost model-based backend code-generation framework, can give these performance gains automatically.

There are a number of complimentary lines of investigation that we are still pursuing. Further optimizations at the frontend and backend of our framework in addition to pipelining and vectorization could further improve

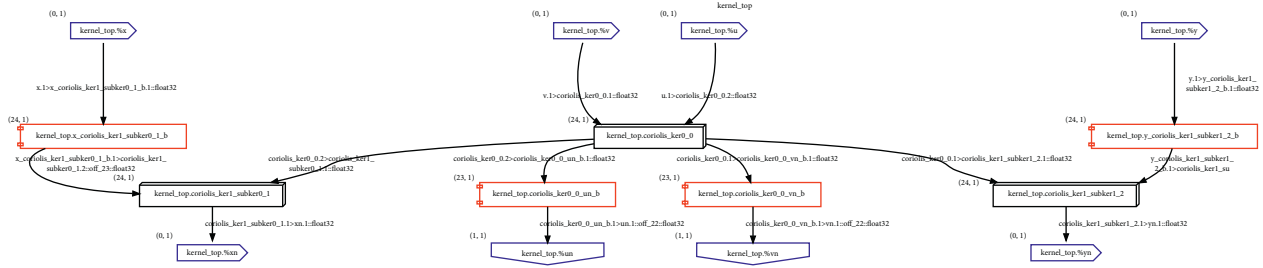


FIGURE 11: The DFG of the Coriolis example generated from the TIR, showing only the top-level kernel. The tuple of integers with each node is the scheduling parameters (latency and firing-interval) used by the backend for scheduling and RTL code generation. The red boxes (the boxes with two small stubs) are inferred buffers for synchronization and deadlock avoidance.

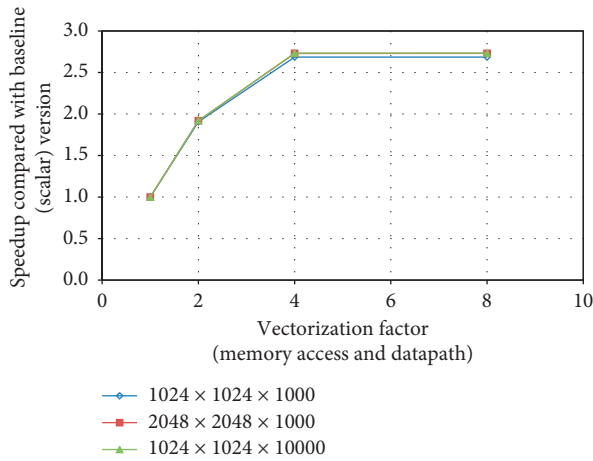


FIGURE 12: Speedup achieved over nonvectorized OpenCL baseline plotted against vectorization factor, for the second example (Coriolis). The TyTra solutions are OpenCL-HDL hybrids, and the complete host API, shell, and kernel code for all variants is generated automatically from TIR description. The speedup is calculated for 3 different grid sizes and time steps (legend shows $dimension1 \times dimension2 \times time\ steps$).

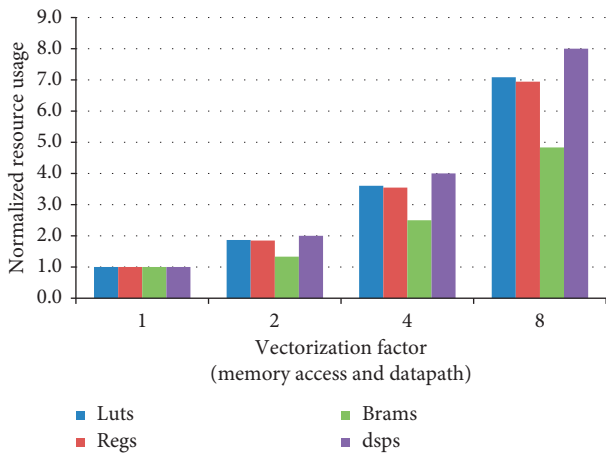


FIGURE 13: Resource usage for various OpenCL-HDL hybrid vectorized versions, normalized against resource usage for non-vectorized OpenCL baseline, for the second example (simulating the Coriolis force).

performance. Until now, we have made models from domain of fluid dynamics the focus for our test cases, and such models are innately amenable to finding mappable loops and hence to *streaming*. We are investigating extending the application domain to deep learning neural networks, which too lends itself to a streaming architecture, but requires closer integration of *folds* in addition to maps, which is an on-going work. We are also in the process of integrating all stages of the flow into a single framework, which we hope will contribute to mainstreaming of FPGAs as HPC accelerators.

Data Availability

The TyTra backend compiler has been deposited in a Github repository at <https://github.com/waqarnabi/tybec>. This is an on-going work, so the authors should be contacted if any issues.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

The authors acknowledge support of the EPSRC for this work carried out as part of the TyTra project (no. EP/L00058X/1).

References

- [1] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with FPGAs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16*, pp. 409–420, Piscataway, NJ, USA, November 2016.
- [2] S. W. Nabi and W. Vanderbauwhede, "MP-STREAM: a memory performance benchmark for design space exploration on heterogeneous HPC devices," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 194–197, Vancouver, British Columbia, Canada, May 2018.
- [3] F. A. Escobar, X. Chang, and C. Valderrama, "Suitability analysis of FPGAs for heterogeneous platforms in HPC," *IEEE*

- Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 600–612, 2016.
- [4] D. Weller, F. Oboril, D. Lukarski, J. Becker, and M. Tahoori, “Energy efficient scientific computing on FPGAs using OpenCL,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA ’17*, pp. 247–256, New York, NY, USA, February 2017.
 - [5] O. Pell and V. Averbukh, “Maximum performance computing with dataflow engines,” *Computing in Science & Engineering*, vol. 14, no. 4, pp. 98–103, 2012.
 - [6] T. Czajkowski, U. Aydonat, D. Denisenko et al., “From OpenCL to high-performance hardware on FPGAs,” in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL)*, pp. 531–534, Oslo, Norway, August 2012.
 - [7] Xilinx, *The Xilinx SDAccel development environment*, 2014 <https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html>.
 - [8] M. Cole, “Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming,” *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.
 - [9] O. Segal, P. Colangelo, N. Nasiri, Z. Qian, and M. Margala, “SparkCL: a unified programming framework for accelerators on heterogeneous clusters,” CoRR, abs/1505.01120, 2015.
 - [10] J. Agron, *Domain-Specific Language for HW/SW Co-design for FPGAs*, pp. 262–284, Springer, Berlin, Heidelberg, Germany, 2009.
 - [11] C. Kulkarni, G. Brebner, and G. Schelle, “Mapping a domain specific language to a platform FPGA,” in *Proceedings of the 41st Annual Design Automation Conference, DAC ’04*, pp. 924–927, New York, NY, USA, June 2004.
 - [12] D. B. Thomas, S. T. Fleming, G. A. Constantinides, and D. R. Ghica, “Transparent linking of compiled software and synthesized hardware,” in *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1084–1089, Grenoble, France, March 2015.
 - [13] M. Weinhardt and W. Luk, “Memory access optimization and ram inference for pipeline vectorization,” in *Proceedings of the International Workshop on Field Programmable Logic and Applications*, pp. 61–70, Glasgow, UK, August 1999.
 - [14] C. Liao, D. J. Quinlan, T. Panas, and B. R. De Supinski, “A rose-based openmp 3.0 research compiler supporting multiple runtime libraries,” in *Proceedings of the International Workshop on OpenMP*, pp. 15–28, Beijing, China, June 2010.
 - [15] D. Orchard and A. Rice, “Upgrading fortran source code using automatic refactoring,” in *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools, WRT ’13*, pp. 29–32, New York, NY, USA, October 2013.
 - [16] J. Overbey, S. Xanthos, R. Johnson, and B. Foote, “Refactorings for fortran and high-performance computing,” in *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, pp. 37–39, St. Louis, MO, USA, May 2005.
 - [17] W. Vanderbauwhede and G. Davidson, “Domain-specific acceleration and auto-parallelization of legacy scientific code in fortran 77 using source-to-source compilation,” *Computers & Fluids*, vol. 173, pp. 1–5, 2018.
 - [18] F. De Dinechin and B. Pasca, “Designing custom arithmetic data paths with FloPoCo,” *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.
 - [19] SDAccel optimization recommendations, 2019, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1207-sdaccel-optimization-guide.pdf.
 - [20] G. Kahn, “The semantics of a simple language for parallel programming,” *Information Processing*, vol. 74, pp. 471–475, 1974.
 - [21] H. Nikolov, T. Stefanov, and E. Deprattere, “Modeling and FPGA implementation of applications using parameterized process networks with non-static parameters,” in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’05)*, pp. 255–263, Napa Valley, CA, USA, April 2005.
 - [22] S. Shukla, N. W. Bergmann, and J. Becker, “QUKU: a FPGA based flexible coarse grain architecture design paradigm using process networks,” in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pp. 1–7, Long Beach, CA, USA, March 2007.
 - [23] T. M. Parks, *Bounded Scheduling of Process Networks*, Technical report, California University Berkeley Department of Electrical Engineering and Computer Sciences, Berkeley, CA, USA, 1995.
 - [24] S. W. Nabi and W. Vanderbauwhede, “FPGA design space exploration for scientific HPC applications using a fast and accurate cost model based on roofline analysis,” *Journal of Parallel and Distributed Computing*, vol. 133, pp. 407–419, 2017.
 - [25] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, “Scheduling algorithms for automated synthesis of pipelined designs on FPGAs for applications described in MATLAB,” in *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, vol. 17, pp. 85–93, San Jose, CA, USA, 2000.
 - [26] Amazon EC2 F1 instances, 2019, <https://aws.amazon.com/ec2/instance-types/f1/>.
 - [27] AMBA, AXI and ACE Protocol Specification, 2011, https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf.
 - [28] J. Kämpf, *Ocean Modelling for Beginners: Using Open-Source Software*, Springer Science & Business Media, Berlin, Germany, 2009.

Research Article

Dimension Reduction Using Quantum Wavelet Transform on a High-Performance Reconfigurable Computer

Naveed Mahmud  and Esam El-Araby 

Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS 66045, USA

Correspondence should be addressed to Naveed Mahmud; naveed_923@ku.edu

Received 4 May 2019; Revised 16 August 2019; Accepted 1 September 2019; Published 11 November 2019

Academic Editor: Wim Vanderbauwhede

Copyright © 2019 Naveed Mahmud and Esam El-Araby. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The high resolution of multidimensional space-time measurements and enormity of data readout counts in applications such as particle tracking in high-energy physics (HEP) is becoming nowadays a major challenge. In this work, we propose combining dimension reduction techniques with quantum information processing for application in domains that generate large volumes of data such as HEP. More specifically, we propose using quantum wavelet transform (QWT) to reduce the dimensionality of high spatial resolution data. The quantum wavelet transform takes advantage of the principles of quantum mechanics to achieve reductions in computation time while processing exponentially larger amount of information. We develop simpler and optimized emulation architectures than what has been previously reported, to perform quantum wavelet transform on high-resolution data. We also implement the inverse quantum wavelet transform (IQWT) to accurately reconstruct the data without any losses. The algorithms are prototyped on an FPGA-based quantum emulator that supports double-precision floating-point computations. Experimental work has been performed using high-resolution image data on a state-of-the-art multinode high-performance reconfigurable computer. The experimental results show that the proposed concepts represent a feasible approach to reducing dimensionality of high spatial resolution data generated by applications such as particle tracking in high-energy physics.

1. Introduction

High-energy physics deal with advanced instruments such as particle accelerators and detectors. These machines use electromagnetic fields to accelerate charged particles to high speeds and create collisions. By studying particle collisions and tracking collision trajectories, physicists can test the predictions of many theories of particle physics such as properties of the Higgs boson [1], discovering new particle families [2] as well as many high-energy physics problems [3]. There are a number of high-energy physics (HEP) research centers [4]. The largest particle accelerator is the Large Hadron Collider (LHC) in Geneva, Switzerland. Large-scale general-purpose particle detectors have been developed at the LHC. The ATLAS [5] and Compact Muon Solenoid (CMS) [6] are two examples which are used for studying the properties of the Higgs boson and investigating new physics. The ATLAS has an inner detector that has been used to observe the decay products of collisions. The pixel

detector [7] is one of the main components of the inner detector, having over 80 million readout channels [8] (pixels), which contribute to half the total readout channels of the entire experiment. Reconstruction of high-energy particles from the pixel detector is considered a critical design and engineering challenge [9], due to its large readout count, high spatial resolution, and 3D space-time measurements. There have been efforts to improve the tracking performance of the ATLAS Inner Detector [9, 10], which involved insertion of additional pixel detector layer (Insertable B-Layer). Another approach that has been considered in the ATLAS FTK (Fast Track Trigger) upgrade [11] is using variable resolution patterns, where the data from the detector is compared to generated pattern banks of particle tracks and non-intersecting data is filtered. In high-dimensional datasets, e.g., the pixel detector readout data, not all the measured data variables are relevant in understanding the underlying regions of interest (RoI). Generally, statistical predictive models are applied to multidimensional datasets for detection and pattern matching, which is a computationally

expensive process. Thus, an effective method is needed to reduce the dimensionality [12] of the data in such high-dimensional spatial sets, for faster detection and matching.

As a feasible solution to this problem, we here propose combining wavelet-based dimension reduction techniques [13–15] with quantum information processing (QIP) [16] for applications in domains that generate high-dimensional data volumes such as high-energy physics (HEP). More specifically, we propose using quantum wavelet transform (QWT) to reduce the dimensionality and high spatial resolution of data in HEP particle tracking. Wavelet-based dimension reduction has been shown to be an effective technique in image pre-processing, reducing computation time, reducing inter-processor overhead, and improving classification accuracy [13–15]. Even so, the large volume of data from domains such as high-energy particle physics, present a challenge for a classical wavelet-based method. The QWT has been demonstrated in previous works to be very useful in quantum image processing and quantum data compression [16–19]. Quantum information processing uses qubits as the basic units of information storage, compared to classical binary forms, and can exploit quantum mechanical properties such as entanglement and superposition [20]. Therefore, applying QIP techniques such as QWT for dimension reduction of HEP data will bring substantial improvements in storage and computation compared to classical signal processing techniques. To the best of our knowledge, this work is the first to investigate QWT-based dimension reduction for HEP applications. We develop simple and effective algorithms for QWT and inverse-QWT (IQWT) that are best suited for dimension reduction and present corresponding emulation hardware architectures for QWT and IQWT.

The objectives and focus of our work are to demonstrate the feasibility of QWT for dimension reduction, through emulation, and to evaluate the performance of the emulation architectures. Our proposed algorithms are prototyped on an FPGA-based quantum emulator that has been developed based on our previous works [21, 22] and has been shown to emulate full quantum algorithms such as quantum Fourier transform (QFT) [23] and Grover’s search algorithm [24]. An FPGA platform was chosen because of its reconfigurability and flexibility in emulating multiple quantum algorithms. The emulator is based on the hardware system of DirectStream [25], which is a state-of-the-art reconfigurable computing platform. This emulation platform can be conveniently used to verify and benchmark future implementations of the proposed system in HEP applications. In the next section, we discuss fundamental concepts of quantum computing, QWT, and the related work done on QWT. In Section 3, we elaborate our proposed methods and emulation architectures. In Section 4, the experimental results and analysis are presented. Section 5 is our conclusion and future directions of this work.

2. Background and Related Work

In this section, we discuss background concepts of quantum computing and the quantum wavelet transform. We also discuss current and related work on QWT and high-energy particle detection.

2.1. Qubits, Superposition, and Entanglement. The qubit is the smallest unit of quantum information that describes a two-level quantum mechanical system. Physical implementations of the qubit can be electron/atomic/nuclear spin, where spin directions of the particle represent the two qubit levels. Other physical representations of the qubit can be photon polarization, superconducting Josephson junction, etc. [26]. The qubit is represented theoretically using the Bloch sphere [20], as shown in Figure 1. The basis states of the qubit, $|0\rangle$ and $|1\rangle$ are denoted by poles of the sphere. The property that distinguishes the qubit from the classical bit is superposition. The qubit can exist in a mixed or superposition state that is any other point on the surface of the sphere other than the poles. The overall state of the qubit can be defined using a linear superposition equation $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers determined from φ and θ as shown in Figure 1 and satisfying $|\alpha|^2 + |\beta|^2 = 1$. Another distinguishing property of qubits is entanglement [20]. Two or more qubits can be entangled together, which means each entangled qubit becomes strongly correlated to the other along all possible combinations of the qubits. Outcome of measurement of one qubit is dependent on the other measurement, but individually they exhibit completely random behavior. In quantum computing, most algorithms assume that the qubits are fully entangled [21]. A system of n entangled qubits can be represented in vector space as $N = 2^n$ complex basis state coefficients.

2.2. Quantum Wavelet Transform. The wavelet transform, similar to other transforms like Fourier transform, decomposes input signals into their components. The principal difference is that Fourier transform decomposes input signals into their sinusoidal orthogonal temporal-only bases, while wavelet transform uses a set of non-sinusoidal functions, usually called mother wavelets, that are both spatially and temporally localized [15]. This results in a very important feature unique to wavelet transform which is the preservation of spatial locality of data. In other words, wavelet transform gives information about both time and frequency of input data. Wavelet transform also has better computation speeds compared to other transforms [14]. Therefore, they are effective and widely used in many image processing applications [16]. The wavelet transform can be effectively implemented in the quantum information processing (QIP) domain as quantum wavelet transform (QWT) [16, 18, 19]. However, the related work on QWT is rare or preliminary. This is because quantum computing and QIP are fields that are gradually developing and have not yet reached full potential. Although many large-scale quantum hardware is being developed [27], their useful applications are still yet to be decided. We discuss the classical wavelet transform first and then apply it in QIP domain, to establish a model for the QWT. The general wavelet transform can be expressed by

$$F(a, b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} f(t) \Psi_{a,b}^* \left(\frac{t-b}{a} \right) dt, \quad (1)$$

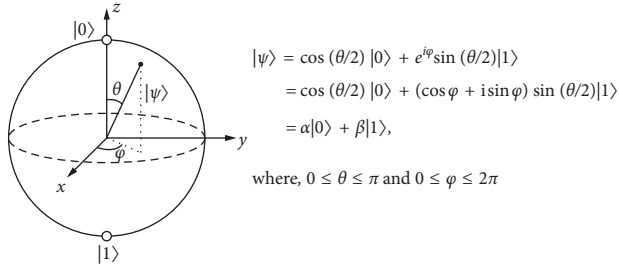


FIGURE 1: Bloch sphere representation of a single qubit.

where Ψ is called the mother wavelet function in complex conjugate form, and a , b are the time dilation and displacement factors, respectively. Wavelet transforms can be classified as *discrete* or *continuous* depending on the use of orthogonal or non-orthogonal wavelets, respectively. For the purposes of this paper, we will discuss the discrete wavelet transform (DWT). The DWT is a decomposition of input signals into a set of wavelet functions that are orthogonal to its translations and scale. The first and simplest DWT was introduced by mathematician Alfred Haar [15] and is thus named the Haar wavelet transform. The Haar mother wavelet function can be constructed using a unit step function, $u(t)$, as shown in (2). The discretized version of the Haar wavelet function is defined as (3), where $t = q \cdot \Delta t$, $b = j \cdot \Delta t$, and $a = K \cdot \Delta t$, Δt is the sampling period, and K is the Haar window size in samples. Applying (3) in (1), the expression for the discrete Haar wavelet transform can be derived to be (4):

$$\Psi\left(\frac{t-b}{a}\right) = u\left(\frac{t-b}{a}\right) - 2u\left(\frac{t-b}{a} - \frac{1}{2}\right) + u\left(\frac{t-b}{a} - 1\right), \quad (2)$$

$$\Psi_D\left(\frac{q-j}{K}\right) = \begin{cases} +1, & 0 \leq (q-j) < \frac{K}{2}, \\ -1, & \frac{K}{2} \leq (q-j) < K, \\ 0, & \text{otherwise.} \end{cases} \quad (3)$$

$$F_D(j, K) = \sum_{q=0}^{N-1} f_D(q \cdot \Delta t) \Psi_D\left(\frac{q-j}{K}\right), \quad (4)$$

where N is the number of data samples. When doing computation in the quantum domain, there are efficient methods of classical-to-quantum encoding [28–30]. Classical signal samples can be encoded as the coefficients of a quantum state, which is in superposition of its constituent basis states [28, 31]. The signal samples are transformed to a normalized sequence of amplitudes as shown in (5), where n is the number of qubits, $N = 2^n$ is the number of basis states of the quantum system, and $|\psi\rangle$ is the input quantum state. By applying the wavelet transform on the input quantum state, we can formulate the equivalent expression for the quantum Haar wavelet transform (QHT) as (6), where $|\psi\rangle_{\text{QHT}}$ is the output quantum state:

$$|\psi\rangle = \sum_{q=0}^{N-1} f(q \cdot \Delta t) |q\rangle, \quad \text{where} \quad \sum_{q=0}^{N-1} |f(q \cdot \Delta t)|^2 = 1, \quad (5)$$

$$|\psi\rangle_{\text{QHT}} = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} \sum_{q=0}^{N-1} f(q \cdot \Delta t) \Psi_D\left(\frac{q-j}{K}\right) |j\rangle. \quad (6)$$

There are many notable works on wavelets and applications of wavelet transforms [32–34]. We focused our survey on works of wavelet transform applied in the field of quantum information processing, i.e., quantum wavelet transforms (QWT). Early work on the QWT was reported in [16], where the authors present gate-level circuits for the quantum Haar wavelet and Daubechies $D^{(4)}$ wavelet. They propose techniques for efficient quantum implementation of permutation matrices, which are required for factorization of the unitary operations of the wavelet transforms. In [35], the authors present quantum algorithms for Haar wavelet transforms and demonstrate applications in analyzing the multiscale structure of a dynamical system by logistic mapping. They show the derivation of the quantum wavelet transform by factoring the classical operators into direct sums, direct products, and dot products, which is the same approach in [16]. The work in [36] also demonstrates similar quantum circuits for QWT based on the well-known pyramid and packet algorithms which are used in classical DWT. The work in [37] presents an analytical study of effects of imperfections in quantum computation of a QWT-based dynamical model. They propose a QWT-based algorithm for the Daubechies wavelets. The works in [38, 39] demonstrate applications of QWT in image watermarking. A more recent, novel watermarking method is proposed in [18], where they demonstrate improvement in invisibility and robustness of the watermarked image. The most recent work on QWT is presented in [19], where the authors provide quantum circuit derivations for the Haar and Daubechies wavelet transforms. The authors propose QHT circuits which contain k levels of permutations, where k is the kernel size.

The previous work on the QWT has mostly presented circuits and software simulations, and no hardware implementations were reported. In comparison, our focus is on efficient hardware implementation of the QWT, and we propose an optimized, low resource-intensive approach for emulation on classical hardware. To the best of our knowledge, our work is the first to (1) propose using QHT for reducing data dimensionality and (2) provide hardware emulation architectures for QHT. Our approach is simpler and optimized for emulation because it uses a single Haar kernel model and a pair of permutation models, where the permutation models are implemented as classical circuits. We propose classical circuits for permutation because (1) quantum permutation circuits implemented using multiple levels of swap operations [16, 19, 35, 36] have large quantum cost, and (2) classical permutation techniques such as index scheduling are space and time efficient for hardware implementation.

Moreover, among the previous work there have been no experimental demonstrations of QWTs on actual quantum hardware or on any quantum emulators. In our work, we

present simplified architectures for implementing multilevel, multidimensional QHT operations on classical hardware and propose application of these methods in dimensionality reduction of particle tracking data in high-energy physics applications. Our proposed algorithms and architectures are easily generalizable, compared to previous works. In addition, our proposed architectures are more effective in utilizing minimal quantum and classical hardware resources which is more suited for dimension reduction. We experimentally evaluate the architectures on a high-throughput and high-accuracy FPGA quantum emulator. To the best of our knowledge, this work is the first to present experimental demonstrations of quantum wavelets used for dimension reduction in large-scale applications, e.g., LHC.

2.3. High-Energy Particle Detectors. The ATLAS Fast Tracker (FTK) is a hardware processor upgrade [9] for the Large Hadron Collider (LHC) which has been developed for faster reconstruction of tracks at 100 kHz. Details of the operating principle, hardware components, and performance can be found in [40]. The reconstruction is done by matching detector data with predefined track patterns that are stored in associative memory on ASICs. The data processing and pattern matching are done using FPGA hardware. The FTK receives data from the ATLAS pixel detector and stores them as clusters to reduce data size. The clusters are arranged into regions for parallel processing. In the processing units (PU), the tracks are stored with full resolution on input FPGAs, while other FPGA processors are responsible for converting the stored data into coarser resolution segments. This is followed by comparison of the coarse-grained segments with pre-stored Monte Carlo track patterns. The coarse granularity of the tracks can cause problems in identification and pattern matching and lead to slower tracking performance of the FTK. In this work, we propose QHT techniques to reduce dimensionality of full resolution data such as FTK particle tracks. We also demonstrate an FPGA-based hardware prototype that can be easily integrated into the current FTK ATLAS architecture.

3. Methodology and Emulation Architectures

In this section, we elaborate our methodology that uses QHT to achieve dimension reduction. We also detail the corresponding emulation hardware architectures that were implemented [41].

3.1. Dimension Reduction. The classical wavelet transform has been shown to achieve dimension reduction efficiently and can be used in various applications that use hyperspectral data, for example, remote sensing, mineralogy, and surveillance. Depending on the type of data and the application in which these data are being used, both 1D wavelet transform (1D-WT) and 2D wavelet transform (2D-WT) techniques can be used for dimension reduction. For example, while the data in remote sensing hyperspectral imagery is in the form of large 3D data cubes, 1D wavelet transform (1D-WT) was previously proposed [13, 14] for efficient dimensionality reduction of such data cubes. In the experimental work in [14], five levels of

wavelet decomposition were used on images of size 217×512 pixels by 192 bands to achieve $\times 32$ reductions in data volume. In current and future large-scale applications, the volume of data can be overwhelming. For example, hyperspectral image cubes are typically hundreds of pixels in width and height [13], with 220–240 frequency bands [14]. The ATLAS pixel detector contains 1700 detection modules corresponding to 8×10^7 pixels [8] and has bandwidth capacity of 48 Gb/s [11]. Hence, it is necessary to investigate and apply newer paradigms of information processing and storage for supporting future applications at full bandwidths. In quantum information processing, exponentially greater amount of information can be held in the state of quantum system compared to a classical binary system. Thus, we propose using quantum information processing techniques such as the QWT for the processing of high volumes of data in large-scale applications. For example, a $64K \times 64K$ image can be reduced to a smaller resolution of 32×32 using a 32-qubit, 12-level QWT decomposition. The pixels are encoded as N basis states of a quantum state, where $N = 2^n$ and n is the number of qubits, i.e., 32.

Our proposed methodology for dimension reduction using quantum wavelet transforms is shown in Figure 2 [41]. In our proposed approach, each pixel of the input image is encoded as a basis coefficient of a quantum state. Input image data first undergoes a multidimensional quantum Haar transform, e.g., one-dimensional QHT (1D-QHT) or two-dimensional QHT (2D-QHT) operation. The operations can have multiple decomposition levels and separate the input image into a number of low frequency and high frequency replications, depending on the number of decomposition levels. The lowest frequency image replication retains the principal components of the input data without significant data loss. More importantly, the mirror images have reduced dimensionality and thus can be used for reducing preprocessing overhead or communication bandwidth congestion. Multidimensional inverse quantum Haar transform (1D-IQHT or 2D-IQHT) is then applied to reconstruct the original data. The 2D operations can be achieved by cascading 1D operations and multiple permutation sets.

The proposed kernel-based algorithms for multilevel 1D-QHT and 2D-QHT are elaborated in Algorithms 1 and 2, respectively. The algorithms perform multilevel decompositions of 1D-QHT or 2D-QHT operations based on a d -dimensional Haar wavelet kernel. The kernel functionality can be represented by a set of operations applied to some input states/pixels and is preceded and followed by perfect shuffle permutation operations [16] on the input and output states/pixels. The permutation operations are performed by means of index calculations and scheduling. Algorithm 1 performs 1D-QHT on a set of input pixels, X , to produce an output pixel set, Y . The input pixels first undergo input permutations, followed by 1D Haar kernel operations on 2 pixels every cycle, and output permutations. Algorithm 2 performs 2D-QHT on a set of input pixels, X , to produce an output pixel set, Y . The input pixels first undergo input permutations, followed by 2D Haar kernel operations on 4 pixels every cycle, and output permutations.

To efficiently extract output state data, quantum-to-classical readout techniques [28] such as quantum Fourier transform (QFT) can be employed. However, this was not

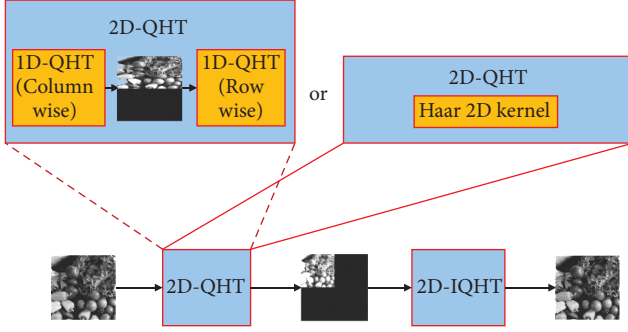


FIGURE 2: Dimension reduction using 2D-QHT and 2D-IQHT.

```

Input:  $X = [x_0, x_1, \dots, x_{N-1}]$ ,  $n_{\text{rows}}$ ,  $n_{\text{cols}}$ ,  $n_{\text{levels}}$ 
Output:  $Y = [y_0, y_1, \dots, y_{N-1}]$ 
 $n_{\text{states}} = n_{\text{rows}} \times n_{\text{cols}} = N$ 
for  $i_{\text{level}} = 1$ ;  $i_{\text{level}} \leq n_{\text{levels}}$ ;  $i_{\text{level}}++$  do
  for  $i_{\text{group}} = 0$ ;  $i_{\text{group}} < (n_{\text{states}}/2)$ ;  $i_{\text{group}}++$  do
    //Initial scheduler setup
     $i_{\text{colGroup}} = \lfloor i_{\text{group}} / (n_{\text{rows}}/2) \rfloor$ 
     $i_{\text{rowGroup}} = i_{\text{group}} - i_{\text{colGroup}} \times (n_{\text{rows}}/2)$ 
     $i_{\text{col}} = i_{\text{colGroup}}$ 
     $i_{\text{row}} = 2 \times i_{\text{rowGroup}}$ 
    //Input Permutations/Scheduler
     $i_{X_{00}} = i_{\text{row}} + (i_{\text{col}} \times n_{\text{rows}})$ 
     $i_{X_{10}} = i_{X_{00}} + 1$ 
     $X_{00} \leftarrow X[i_{X_{00}}]$ 
     $X_{10} \leftarrow X[i_{X_{10}}]$ 
    //1D-QHT kernel
     $Y_{00} = (X_{00} + X_{10}) / \sqrt{2}$ 
     $Y_{10} = (X_{00} - X_{10}) / \sqrt{2}$ 
    //Output Permutations/Scheduler
     $i_{Y_{00}} = (i_{\text{row}} + (i_{\text{col}} \times n_{\text{rows}})) / 2$ 
     $i_{Y_{10}} = i_{Y_{00}} + (n_{\text{rows}}/2)$ 
     $Y[i_{Y_{00}}] \leftarrow Y_{00}$ 
     $Y[i_{Y_{10}}] \leftarrow Y_{10}$ 
  end for
end for

```

ALGORITHM 1: Multilevel 1D quantum Haar transform.

required to be implemented in this work as full emulation of quantum computation was performed on classical hardware and the output of the emulator is in classical representation. For emulation, we develop circuit models based on these algorithms and integrate them into reconfigurable hardware architectures for multilevel, multidimensional (1D and 2D) QHT and IQHT. These models and emulation architectures are elaborated in the next section.

3.2. Quantum Haar Transform Kernel. The Haar wavelet kernel can be generalized by quantum operations using n qubits and a d -dimension kernel as shown in (7), where \otimes is the Kronecker product [42], H is the Hadamard transform [20], and I is an identity matrix. Here, a group of entangled gates is denoted by the gate symbol with the size of the equivalent operation matrix as subscript, for example, H_{2^d} .

```

Input:  $X = [x_0, x_1, \dots, x_{N-1}]$ ,  $n_{\text{rows}}$ ,  $n_{\text{cols}}$ ,  $n_{\text{levels}}$ 
Output:  $Y = [y_0, y_1, \dots, y_{N-1}]$ 
 $n_{\text{states}} = n_{\text{rows}} \times n_{\text{cols}} = N$ 
for  $i_{\text{level}} = 1$ ;  $i_{\text{level}} \leq n_{\text{levels}}$ ;  $i_{\text{level}}++$  do
  for  $i_{\text{group}} = 0$ ;  $i_{\text{group}} < (n_{\text{states}}/4)$ ;  $i_{\text{group}}++$  do
    //Initial scheduler setup
     $i_{\text{colGroup}} = \lfloor i_{\text{group}} / (n_{\text{rows}}/2) \rfloor$ 
     $i_{\text{rowGroup}} = i_{\text{group}} - i_{\text{colGroup}} \times (n_{\text{rows}}/2)$ 
     $i_{\text{col}} = 2 \times i_{\text{colGroup}}$ 
     $i_{\text{row}} = 2 \times i_{\text{rowGroup}}$ 
    //Input Permutations/Scheduler
     $i_{X_{00}} = i_{\text{row}} + (i_{\text{col}} \times n_{\text{rows}})$ 
     $i_{X_{10}} = i_{X_{00}} + 1$ 
     $i_{X_{01}} = i_{X_{00}} + n_{\text{rows}}$ 
     $i_{X_{11}} = i_{X_{01}} + 1$ 
     $X_{00} \leftarrow X[i_{X_{00}}]$ 
     $X_{10} \leftarrow X[i_{X_{10}}]$ 
     $X_{01} \leftarrow X[i_{X_{01}}]$ 
     $X_{11} \leftarrow X[i_{X_{11}}]$ 
    //2D-QHT kernel
     $Y_{00} = (X_{00} + X_{10} + X_{01} + X_{11}) / 2$ 
     $Y_{10} = (X_{00} - X_{10} + X_{01} - X_{11}) / 2$ 
     $Y_{01} = (X_{00} + X_{10} - X_{01} - X_{11}) / 2$ 
     $Y_{11} = (X_{00} - X_{10} - X_{01} + X_{11}) / 2$ 
    //Output Permutations/Scheduler
     $i_{Y_{00}} = (i_{\text{row}} + (i_{\text{col}} \times n_{\text{rows}})) / 2$ 
     $i_{Y_{10}} = i_{Y_{00}} + (n_{\text{rows}}/2)$ 
     $i_{Y_{01}} = i_{Y_{00}} + (n_{\text{states}}/2)$ 
     $i_{Y_{11}} = i_{Y_{01}} + (n_{\text{rows}}/2)$ 
     $Y[i_{Y_{00}}] \leftarrow Y_{00}$ 
     $Y[i_{Y_{10}}] \leftarrow Y_{10}$ 
     $Y[i_{Y_{01}}] \leftarrow Y_{01}$ 
     $Y[i_{Y_{11}}] \leftarrow Y_{11}$ 
  end for
end for

```

ALGORITHM 2: Multilevel 2D quantum Haar transform.

The quantum Haar function can be implemented using d entangled H gates and $n - d$ entangled I gates as shown in (7). For example, the transformation matrix for 2D-QHT with $d = 2$ can be derived as shown in (9):

$$U_{\text{QHT}} = I_{2^{(n-d)}} \otimes H_{2^d}, \quad (7)$$

where

$$H_{2^d} = \frac{H \otimes H \otimes \dots \otimes H}{d}$$

$$I_{2^{(n-d)}} = \frac{I \otimes I \otimes \dots \otimes I}{(n-d)},$$

$$H = H_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad (8)$$

$$I = I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

where $n \implies$ number of qubits and $d \implies$ kernel dimension:

$$U_{\text{QHT}}^{2\text{D}} = I_{2^{(n-2)}} \otimes H_{2^2} = I_{2^{n/4}} \otimes H_4 = I_{N/4} \otimes H_4, \quad (9)$$

where

$$H_4 = H \otimes H = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}. \quad (10)$$

3.3. Permutation Operations. Perfect shuffle permutation on a given vector is described as partitioning the vector in half and shuffling the top and bottom portions of the halves [16]. In our algorithms for QHT and IQHT, we apply similar input and output permutation operations before and after applying the QHT kernel, respectively. The QHT kernel is performed on a set of k points, where $k = 2^d$. An input permutation operation involves dividing the input vector of size N , into k groups, and selecting a state (pixel) from every group(s), to be applied to the kernel operation. For 1D-QHT and 2D-QHT operations, the input permutations, $P_{\text{in}}^{1\text{D}}$ and $P_{\text{in}}^{2\text{D}}$, are shown in (11) and (12), respectively. An output permutation involves arranging the pixels from k groups into a single output state sequence. The output permutation for 1D-QHT and 2D-QHT operations, $P_{\text{out}}^{1\text{D}}$ and $P_{\text{out}}^{2\text{D}}$, are shown in (13) and (14), respectively:

$$P_{\text{in}}^{1\text{D}} : \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{(n_{\text{rows}})} \\ x_{(n_{\text{rows}}+1)} \\ \vdots \\ x_{(N-1)} \end{bmatrix} \mapsto \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{(n_{\text{rows}})} \\ x_{(n_{\text{rows}}+1)} \\ \vdots \\ x_{(N-1)} \end{bmatrix}, \quad (11)$$

$$P_{\text{in}}^{2\text{D}} : \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{(n_{\text{rows}})} \\ x_{(n_{\text{rows}}+1)} \\ x_{(n_{\text{rows}}+2)} \\ x_{(n_{\text{rows}}+3)} \\ \vdots \\ x_{(N-1)} \end{bmatrix} \mapsto \begin{bmatrix} x_0 \\ x_1 \\ x_{(n_{\text{rows}})} \\ x_{(n_{\text{rows}}+1)} \\ x_2 \\ x_3 \\ x_{(n_{\text{rows}}+2)} \\ x_{(n_{\text{rows}}+3)} \\ \vdots \\ \vdots \\ \vdots \\ x_{(N-1)} \end{bmatrix}, \quad (12)$$

$$P_{\text{out}}^{1\text{D}} : \begin{bmatrix} y_0 \\ y_{(n_{\text{rows}}/2)} \\ y_1 \\ y_{((n_{\text{rows}}/2)+1)} \\ \vdots \\ \vdots \\ y_{(N-1)} \end{bmatrix} \mapsto \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ \vdots \\ y_{(N-1)} \end{bmatrix}, \quad (13)$$

$$P_{\text{out}}^{2\text{D}} : \begin{bmatrix} y_0 \\ y_{(n_{\text{rows}}/2)} \\ y_{(N/2)} \\ y_{((n_{\text{rows}}/2)+(N/2))} \\ \vdots \\ \vdots \\ y_{(N-1)} \end{bmatrix} \mapsto \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ \vdots \\ y_{(N-1)} \end{bmatrix}. \quad (14)$$

3.4. Emulation Architectures. While developing the emulation architectures for the proposed system, as an intermediate step, we design circuit models, illustrated in Figures 3 and 4 for 1D- and 2D-QHT/IQHT, respectively. These models are derived from the sequence of operations in Algorithms 1 and 2 and can contain quantum and/or classical circuits. The 1D- and 2D-QHT models in Figures 3(a) and 4(a), respectively, consist of input permutation models (P_{in}), followed by Haar kernel models (U_{QHT}) and then output permutation models (P_{out}). The 1D- and 2D-IQHT models in Figures 3(b) and 4(b), respectively, consist of inverse output permutation models (P_{out}^{-1}), followed by Haar kernel models (U_{QHT}) and then inverse input permutation models (P_{in}^{-1}). The inverse models are equivalent to the direct models, as the permutation operations are reversible. To achieve multilevel decompositions, multiple iterations of the Haar kernel models are applied. The QHT and IQHT operations for 1D and 2D are summarized as unitary transformations in (15) and (16), respectively. The emulation architectures of the 1D-QHT/IQHT and 2D-QHT/IQHT are shown in Figures 5 and 6, respectively. Since the hardware implementations of the 1D and 2D are similar, we focus our following discussions on the implementation of the 2D-QHT emulation architectures:

$$1\text{D} - \text{QHT} : P_{\text{out}}^{1\text{D}} \cdot U_{\text{QHT}}^{1\text{D}} \cdot P_{\text{in}}^{1\text{D}}, \quad (15)$$

$$1\text{D} - \text{IQHT} : (P_{\text{in}}^{1\text{D}})^{-1} \cdot U_{\text{QHT}}^{1\text{D}} \cdot (P_{\text{out}}^{1\text{D}})^{-1},$$

$$2\text{D} - \text{QHT} : P_{\text{out}}^{2\text{D}} \cdot U_{\text{QHT}}^{2\text{D}} \cdot P_{\text{in}}^{2\text{D}}, \quad (16)$$

$$2\text{D} - \text{IQHT} : (P_{\text{in}}^{2\text{D}})^{-1} \cdot U_{\text{QHT}}^{2\text{D}} \cdot (P_{\text{out}}^{2\text{D}})^{-1}.$$

As shown in Figures 4(a) and (16), the first step in the 2D-QHT operation is the input permutation $P_{\text{in}}^{2\text{D}}$, which is described by (12). The permutations can be modeled as quantum circuits with multiple swap gates, but that would incur high resource utilization in the corresponding emulation architecture. For this reason, we use classical models

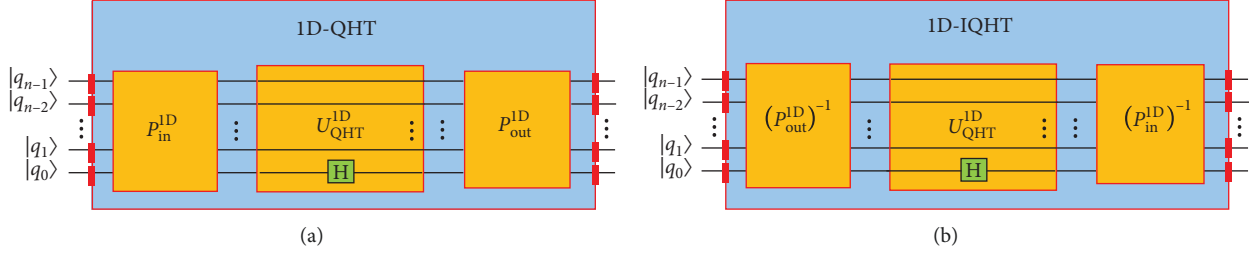


FIGURE 3: (a) 1D-quantum Haar transform circuit. (b) 1D-inverse quantum Haar transform circuit.

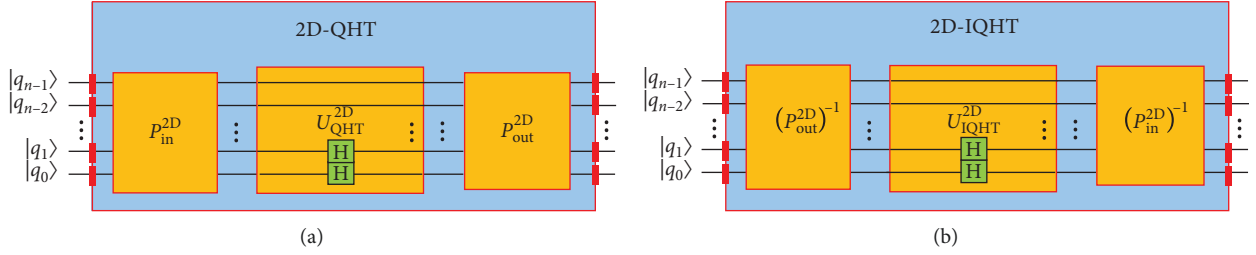


FIGURE 4: (a) 2D-quantum Haar transform circuit. (b) 2D-inverse quantum Haar transform circuit.

that involve simple index scheduling, and the corresponding emulation architecture is shown in Figure 6(a). The input is a vector of quantum state coefficients which are written to a memory array in the index order 0 to $N - 1$. Four coefficient values are then read out each clock cycle, with the scheduler generating the read indices $i_{X_{00}}, i_{X_{01}}, i_{X_{10}},$ and $i_{X_{11}}$ according to the input permutation, see Algorithm 2 and (12). The scheduler maintains a counter, row index i_{row} , and a column index i_{col} to calculate the output indices. Multiplications and divisions by powers of two are replaced by logical shifts for optimizing area and speed. The scheduler also requires a floor operation unit.

As shown in Figures 4(b) and (9), the 2D-Haar transformation, U_{QHT}^{2D} , is modeled using a pair of Hadamard gates. The Hadamard pair operation reduces to kernel operations on a set of four coefficients as we described in Algorithm 2. The emulation architecture for the 2D-Haar kernel is shown in Figure 6(b). The design takes in four input coefficients, applies the kernel operations which involve addition and division, and outputs four coefficients per clock cycle. Conventional operator sharing techniques and logical shifts are applied to optimize for speed and area.

The final step in the 2D-QHT operation is the output permutation, P_{out}^{2D} , described by (14). The corresponding emulation architecture is shown in Figure 6(c) and works similar to the input permutation scheduler. The input vector of coefficients are written to a memory array, four values per clock cycle, with the scheduler generating the write indices $i_{Y_{00}}, i_{Y_{01}}, i_{Y_{10}},$ and $i_{Y_{11}}$ according to the output permutation described in Algorithm 2. The permuted coefficients are then read out from memory 4 values per clock cycle.

The emulation hardware architectures, i.e., input/output schedulers and 1D/2D Haar kernels, were integrated into a reconfigurable quantum emulator design based on our previous works [21, 22], whose high-level architecture is

shown in Figure 7. The emulator stores input and output quantum states as vectors of the state coefficients and core kernel operations are extracted from the input quantum algorithm. The input state vector goes through the input permutations (input schedulers) before the kernel operation is applied iteratively across each state. To get the correct final quantum state that represents the transformed data, the output permutation (output schedulers) is applied. The architecture uses a fully pipelined dataflow architecture and supports single and double-precision floating-point arithmetic. For example, each quantum state coefficient is complex and is modeled in 32 bit floating-point precision for the real and imaginary components, respectively. The emulator also supports features such as fully-entangled input quantum state preparation from a set of input qubits and output quantum state measurement as a classical bit string. The emulator is generic and can efficiently run a given quantum algorithm that can be reduced to its corresponding unitary transformation.

4. Experimental Work

The experimental work was performed on DS8, a state-of-the-art high-performance reconfigurable computing (HPRC) system developed by DirectStream [25]. On the DS8 platform, developers can build applications on hardware systems ranging from single-node compute instances to multinode structures, see Figure 8. A single C2 compute node of the DS8 system is equipped with a high-end Intel-Altera Arria 10AX115N4F45E3SG FPGA, with on-chip resources such as adaptive logic modules (ALMs), block RAMs (BRAMs), digital signal processors (DSPs), and on-board resources such as two 32 GB SDRAM memory banks and four 8 MB SRAM memory banks, as shown in Figure 8. A user-friendly programming environment, previously known as Carte-C [43], is integrated into the DS hardware

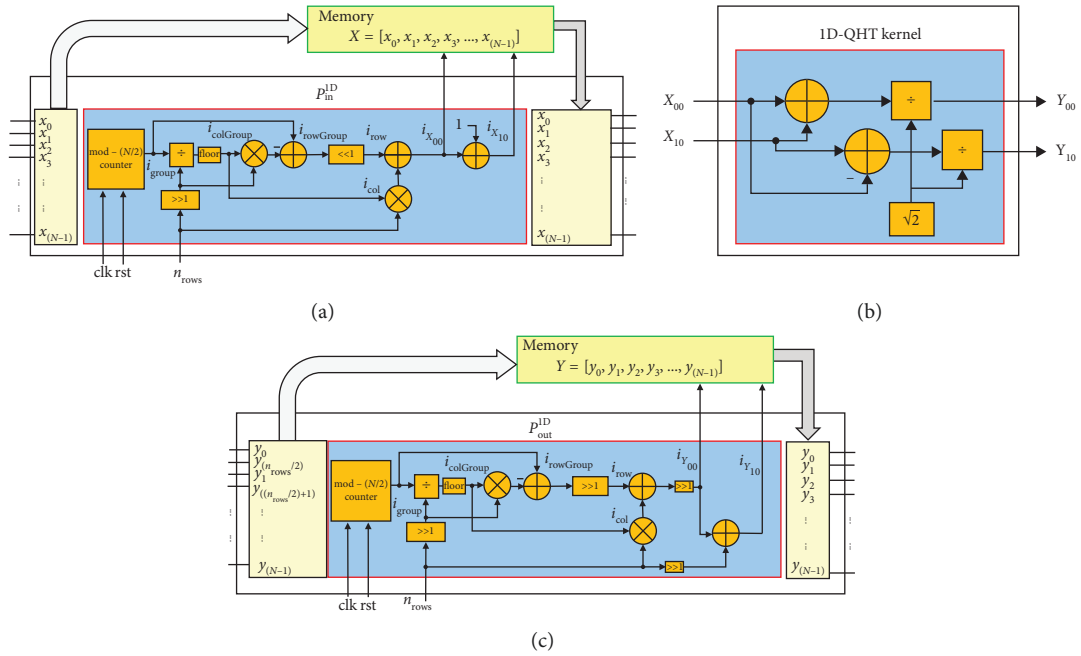


FIGURE 5: Emulation architectures for the (a) 1D input permutation, (b) 1D-Haar kernel, and (c) 1D output permutation.

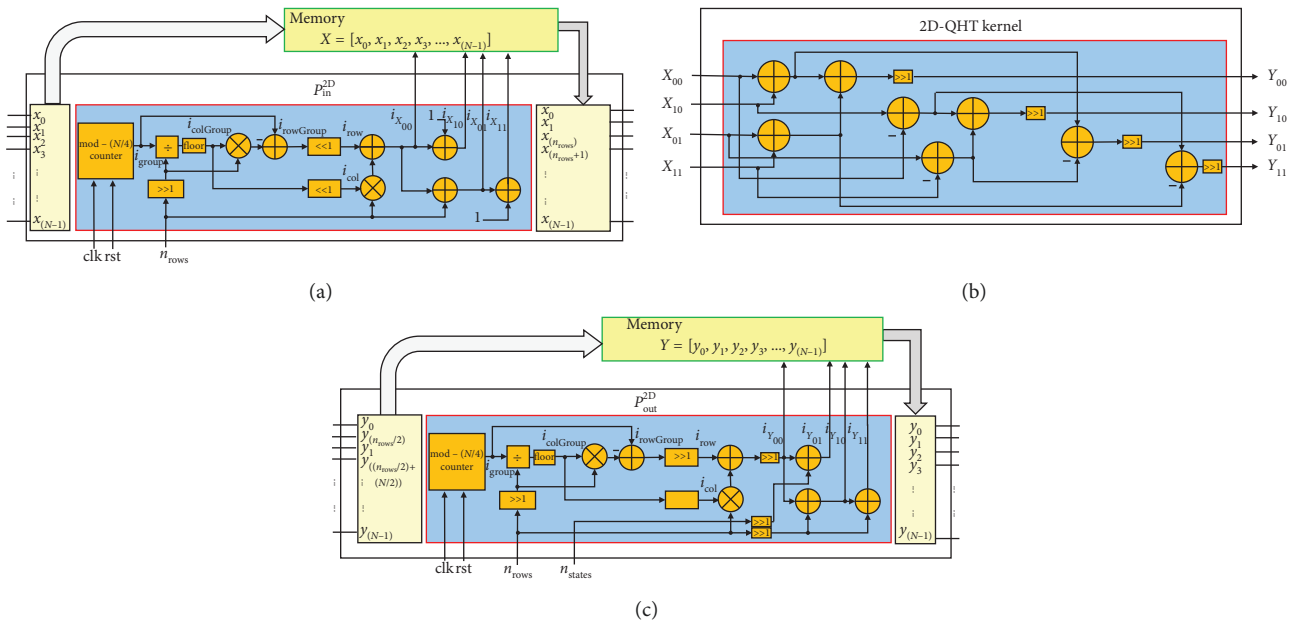


FIGURE 6: Emulation architectures for the (a) 2D input permutation, (b) 2D-Haar kernel, and (c) 2D output permutation.

systems. A high-level language (HLL) facilitates the development of complex, parallel, and reconfigurable codes in an efficient manner. The study in [44] showed that Carte-C has a highly productive environment, short acquisition time, and short learning time as well as a short development time. The DS8 architecture provides a combination of high performance, high scalability, runtime reconfiguration, and ease of use.

The QHT and IQHT architectures were implemented using C++ on the DS8 programming environment. Input

images with a resolution of up to 1024×1024 , and 256 shades of grayscale pixels, were used to test the designs. MATLAB was used to convert the images into greyscale, generate the input vectors for DS8, and reconstruct images from the output vectors. Synthesis and hardware builds were performed using Quartus Prime Version 17.02 on the DS8 environment. Figure 9(a) shows one of the input images converted to greyscale, and Figure 9(b) is the output after a 1D-QHT operation with 1 level of decomposition. Figure 9(c) is the output after a 1D-QHT operation with 2

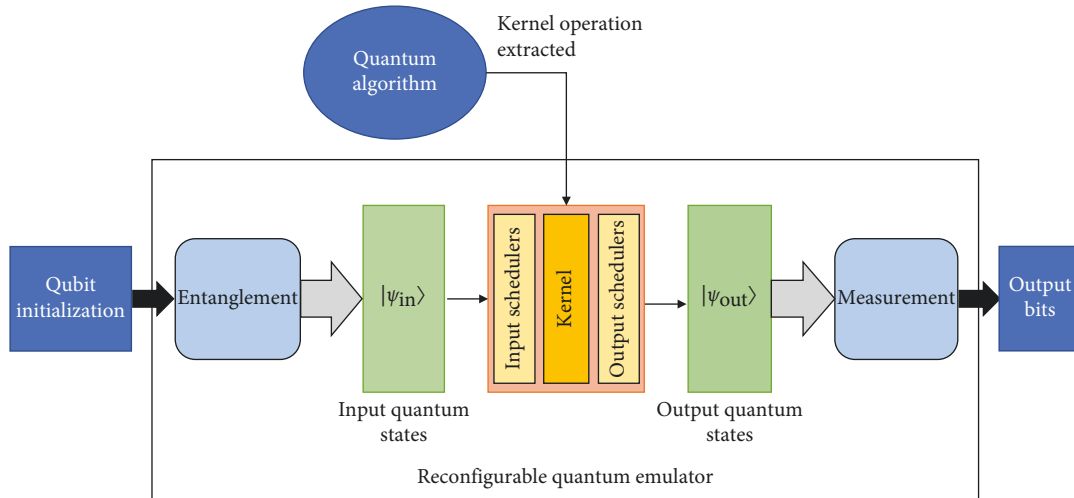


FIGURE 7: Reconfigurable quantum emulator architecture.

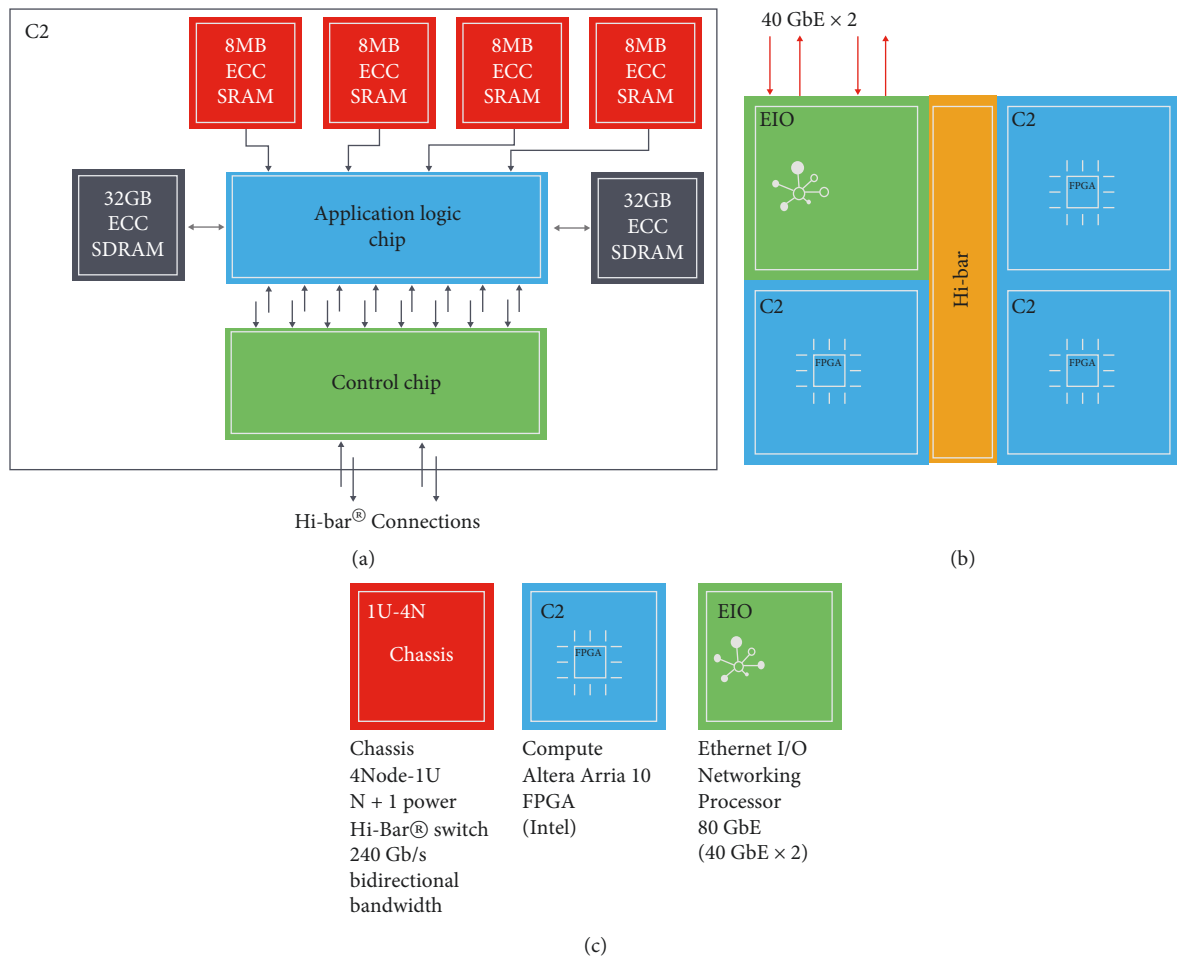


FIGURE 8: DS8 platform architectures. (a) Single compute node. (b) Multinode instance. (c) Node types.

levels of decomposition, and Figure 9(d) shows the reconstructed images after a 1D-IQHT operation was applied. Figures 10(a)–10(d) show the results from repeating the experiment using the 2D-QHT and 2D-IQHT architectures.

Resource utilizations from the hardware implementations are summarized in Tables 1 and 2 for 1D and 2D, respectively. The on-chip resources (ALMs, BRAMs, DSPs) are used up in implementing the static components of the design such as counters, adders, and shift operators and

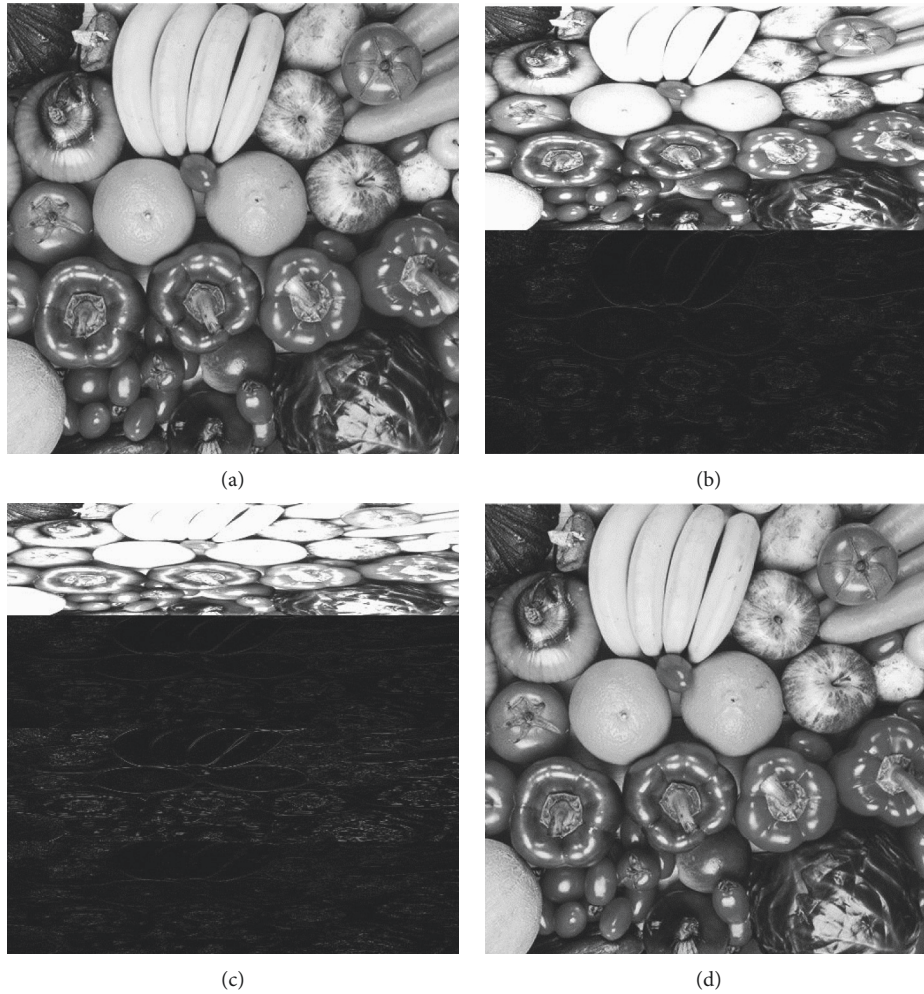


FIGURE 9: Experimental results of multilevel decomposition and reconstruction with 1D-QHT and 1D-IQHT. (a) Original image. (b) 1-level 1D-QHT. (c) 2-level 1D-QHT. (d) Reconstructed image using 1D-IQHT.

hence are constant as the emulated circuit size (number of qubits) increases. The low on-chip resource utilizations indicate that our proposed approach and emulation architecture designs are highly space-efficient. The 1D-QHT architecture consumes lower on-chip resources than 2D-QHT due to its less complex kernel operations. The low resource utilizations also indicate the flexibility of the QHT and IQHT designs for integrating with larger algorithms.

The SDRAM memory requirements for storage of the input and output images as quantum state vectors are also reported in Tables 1 and 2. For the highest resolution image of size 1024×1024 , the pixels occupy 25% of the total on-board SDRAM memory (64 GB) available on a single DS node. The pixels of the input images are encoded as basis coefficients of a quantum state. For example, to store 16×16 or 256 pixels, we need 256 complex coefficients each of which have a real and imaginary component occupying total $2 \times 4 = 8$ bytes in 32 bit floating-point representation. Therefore, for storing both input and output images, $2 \times 256 \times 8 = 4096$ bytes of memory was required. The obtained memory usages for larger QHT circuits are consistent with expected values.

The hardware designs on the FPGA were pipelined to ensure a constant and high operating frequency of 233 MHz. The obtained emulation times for high resolution images are also feasible. For a 1024×1024 image, 20 qubits were sufficient for achieving dimension reduction using 1D-QHT and 2D-QHT. From our experimental results, we observe that the emulation time increases linearly with increase in the number of image pixels (states), as illustrated in Figure 11. This is because a large portion of the emulation time is dedicated to writing in and reading out the input/output state vectors of size N (number of pixels); hence, the emulation time complexity is $O(N)$. This indicates the benefit of using quantum encoding of data, i.e., encoding each image pixel as a basis state coefficient in the quantum state space. Finally, the emulation times for 1D-QHT are higher than 2D-QHT because of the higher number of iterations $N/2$ in the 1D algorithm, compared to $N/4$ iterations in the 2D algorithm, see Algorithms 1 and 2.

In general, on a classical emulation platform, the emulation execution time increases with both the spatial and temporal complexities of the quantum circuit. In other words, the emulation time of a quantum circuit on a classical

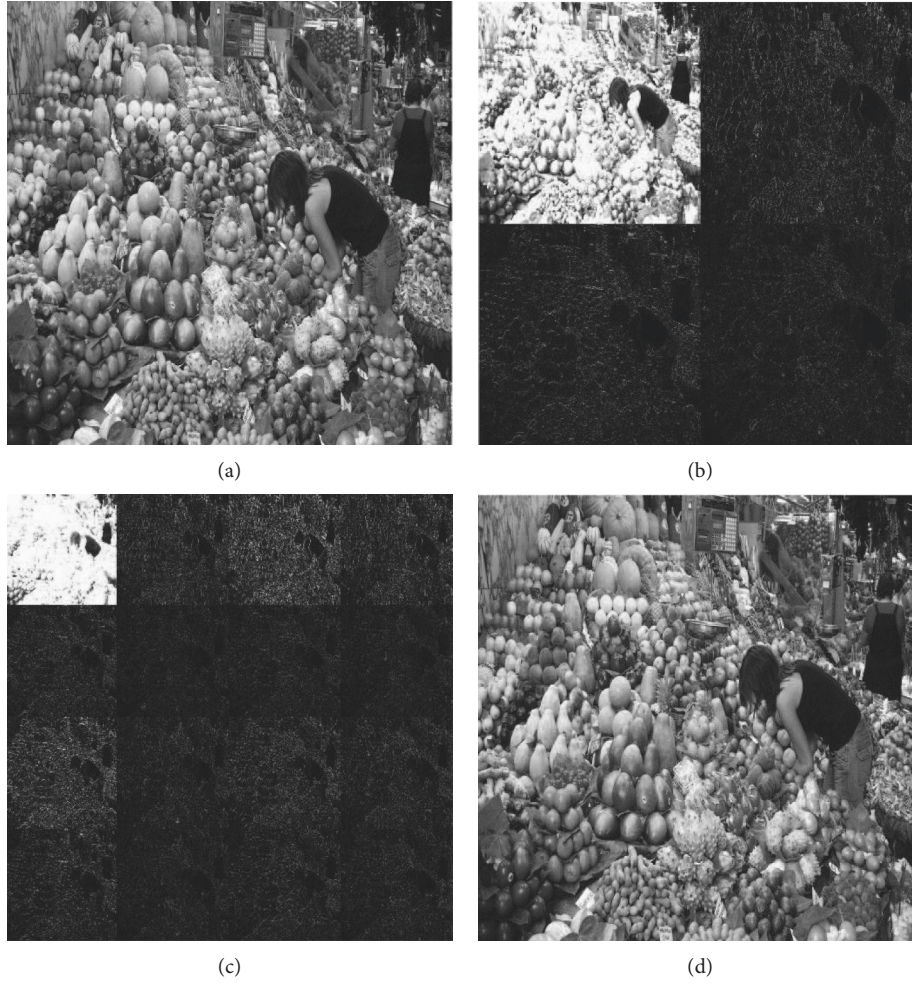


FIGURE 10: Experimental results of multilevel decomposition and reconstruction with 2D-QHT and 2D-IQHT. (a) Original image. (b) 1-level 2D-QHT. (c) 2-level 2D-QHT. (d) Reconstructed image using 2D-IQHT.

TABLE 1: 1D-QHT implementation results on Arria 10AX115N4F45E3SG FPGA.

Number of pixels	Number of qubits	Resource utilization* (%)			SDRAM** (bytes)	Emulation time (sec)
		ALMs	BRAMs	DSPs		
16 × 16	8	11	8	1	4 K	0.00018
32 × 32	10	11	8	1	16 K	0.00071
64 × 64	12	11	8	1	64 K	0.00285
128 × 128	14	11	8	1	256 K	0.01139
256 × 256	16	11	8	1	1 M	0.04557
512 × 512	18	11	8	1	4 M	0.18226
1024 × 1024	20	11	8	1	16 M	0.72905

*Total chip resources: $N_{ALM} = 427,200$; $N_{BRAM} = 2,713$; $N_{DSP} = 1,518$. **Total on-board SDRAM memory: 2 parallel banks of 32 GB each.

TABLE 2: 2D-QHT implementation results on Arria 10AX115N4F45E3SG FPGA.

Number of pixels	Number of qubits	Resource utilization* (%)			SDRAM** (bytes)	Emulation time (sec)
		ALMs	BRAMs	DSPs		
16 × 16	8	14	9	2	4 K	0.00012
32 × 32	10	14	9	2	16 K	0.00047
64 × 64	12	14	9	2	64 K	0.00187
128 × 128	14	14	9	2	256 K	0.00746
256 × 256	16	14	9	2	1 M	0.02982
512 × 512	18	14	9	2	4 M	0.11926
1024 × 1024	20	14	9	2	16 M	0.47704

*Total chip resources: $N_{ALM} = 427,200$; $N_{BRAM} = 2,713$; $N_{DSP} = 1,518$. **Total on-board SDRAM memory: 2 parallel banks of 32 GB each.

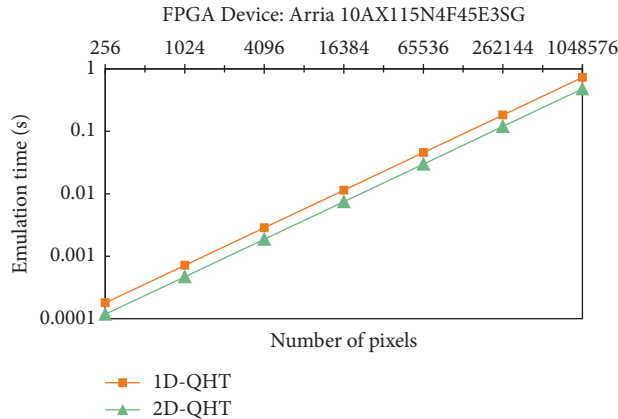


FIGURE 11: Emulation time as a function of data size (number of pixels).

TABLE 3: Comparison of the proposed work against previous works of FPGA emulation.

Reported work	Algorithm	Number of qubits	Precision	Frequency (MHz)	Emulation time (sec)
Fujishima [48]	Shor's factoring	—	—	80	10
Khalid et al. [49]	QFT	3	16 bit fixed pt.	82.1	$61E-9$
	Grover's search	3	16 bit fixed pt.	82.1	$84E-9$
Aminian et al. [50]	QFT	3	16 bit fixed pt.	131.3	$46E-9$
Lee et al. [51]	QFT	5	24 bit fixed pt.	90	$219E-9$
	Grover's search	7	24 bit fixed pt.	85	$96.8E-9$
Silva et al. [52]	QFT	4	32 bit floating pt.	—	$4E-6$
Pilch et al. [53]	Deutsch	2	—	—	—
Mahmud et al. [22]	QFT	5	32 bit floating pt.	233	$4.63E-4^\dagger$
	Grover's search	5	32 bit floating pt.	233	$4.38E-7^\dagger$
Proposed work	QFT	20	32 bit floating pt.	233	18.4
	QHT	20	32 bit floating pt.	233	0.477
	Grover's search	22	32 bit floating pt.	233	$7.5E04$

[†]Results obtained at a later time to publication.

platform is generally a function of both the circuit width (number of qubits) and depth (number of gate levels). Due to optimizations and encoding techniques we used, the emulation time of our proposed emulation architectures is a function of only the quantum circuit width (number of qubits), as shown by our experimental results. On state-of-the-art superconducting NISQ devices [45, 46], the execution time is a function of only the depth (number of gate levels) of the circuit [47]. For our proposed 1D-QHT and 2D-QHT circuits, which are simple quantum circuits of depth 1, we estimate an execution time of 0.01 ms on a typical NISQ device processing a 7×7 qubit array with sampling frequency of 100 kHz [47]. The estimated execution time is constant for a fixed circuit depth and variable number of qubits in the quantum processing unit (QPU) array; i.e., the time complexity is theoretically $O(1)$. In comparison, the time complexity of our emulation is $O(N)$.

Our emulation experiments and implementations help in validating the functionality and feasibility of the proposed QHT-based methodology in achieving dimension reduction of high-resolution images. The emulation provides implications for the proposed system's application in fast, efficient processing of particle tracking data in the large-scale, high-

energy physics domain. The emulation is memory-bound by the resources on a single DS FPGA node. For larger-scale emulation, the on-board memory has to be increased, or multi-node, and/or multichassis architectures of the DS system can be utilized in conjunction with efficient scheduling techniques and high-bandwidth networks [22].

We further quantitatively compare our obtained experimental results with the existing FPGA-based emulation work [48–53] as shown in Table 3. Among the related work on FPGA emulation of quantum circuits, our emulator has the capability of emulating the largest quantum circuits (QFT, QHT, and Grover's search), with highest operating frequency (233 MHz) and high precision (32 bit floating-point). Current FPGA hardware-emulators have many discrepancies (missing resource utilization, operating frequency, and emulation time) in the reporting of their results which makes a comprehensive comparison difficult. In our comparison, we included only hardware emulators, as most parallel-software-simulators are based on large-scale supercomputers such as Summit [47] and Sunway [54], which are extremely costly, power-hungry, and resource-hungry and are not comparable with FPGA-emulators. Also, they provide simulations of random quantum circuits and not full quantum algorithms.

5. Conclusions

Quantum information processing and quantum computing will have significant implications in the future of computing technology. As current quantum technology continues to improve, there is a great need to investigate useful applications in quantum information theory. In this work, we presented a first effort, to the best of our knowledge, to efficiently reduce data dimensionality using quantum processing methods such as quantum wavelet transform. We propose to apply these techniques in physics applications that investigate high-energy particle detection and tracking, where dimension reduction helps to reduce communication bandwidth and speedup preprocessing computations. Our proposed architectures are simpler and optimized for hardware implementation than previously reported works. We demonstrated the minimal resource utilization, high performance/throughout, and high precision of the proposed architectures. We prototyped our designs on a quantum emulator and demonstrated the feasibility of proposed techniques by conducting experiments using high-resolution test image data.

Due to limitations of the current state of quantum technology, e.g., cost, availability, and current scale (size) of quantum processors, it is beyond the scope of this work to actually implement the system and measure performance. Although not yet integrated with the ATLAS FTK project, the proposed approach and emulation hardware architectures are feasible for future implementations, with the maturing of current quantum technology. For future integration into the ATLAS FTK project, data conversion techniques such as quantum-to-classical and classical-to-quantum, which are heavily-researched current topics, must be perfected first, and we plan to conduct investigations of these techniques in our future work. Our future plans also include application of the proposed methods using real HEP data and combining QHT with Grover's search algorithm as a complete solution to HEP FTK problems. We will also investigate 3D-QHT, Daubechies wavelet transforms, and their application for real-time data streaming.

Data Availability

The test data used to support the findings of this study are available from the corresponding author upon request and approval from Directstream.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

We would like to thank Prof. Alice Bean and Prof. Christopher Rogan from the department of Physics and Astronomy at the University of Kansas for their valuable insights and help in this work.

References

- [1] CERN Accelerating science, "The Higgs boson," 2019, <https://home.cern/science/physics/higgs-boson>.
- [2] CERN Accelerating science, "Unified forces," 2019, <https://home.cern/science/physics/unified-forces>.
- [3] V. L. Ginzburg, *The Physics of a Lifetime: Reflections on the Problems and Personalities of 20th Century Physics*, Springer Science & Business Media, Berlin, Germany, 2013.
- [4] I. Kisel, *Track Reconstruction and Pattern Recognition in High-Energy Physics*, 2019, https://www.physik.uni-heidelberg.de/c/image/exp/f/highrr/Kisel_HD_12.04.2016.pdf.
- [5] G. Aad and ATLAS Collaboration, "The ATLAS experiment at the CERN large Hadron collider," *Journal of Instrumentation*, vol. 3, no. 8, 2008.
- [6] C. O'Luanaigh, *New Results Indicate that New Particle is a Higgs Boson*, CERN, Geneva, Switzerland, 2019, <https://home.cern/news/news/physics/new-results-indicate-new-particle-higgs-boson>.
- [7] CERN Accelerating science, "The inner detector," 2019, <https://atlas.cern/discover/detector/inner-detector>.
- [8] F. Hugging, "The ATLAS pixel detector," in *Proceedings of the IEEE Symposium Conference Record Nuclear Science*, vol. 2, pp. 1077–1081, Rome, Italy, October 2004.
- [9] M. Backhaus, "The upgraded pixel detector of the ATLAS experiment for run 2 at the large Hadron collider," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 831, pp. 65–70, 2016.
- [10] H. Pernegger, *The Pixel Detector of the ATLAS Experiment for LHC Run-2*, 2015.
- [11] S. Amerio, A. Andreani, A. Andreazza et al., *ATLAS FTK: Fast Track Trigger*, 2013.
- [12] I. K. Fodor, *A Survey of Dimension Reduction Techniques*, pp. 1–18, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA, 2002.
- [13] S. Kaewpajit, J. Le Moigne, and T. El-Ghazawi, "Automatic reduction of hyperspectral imagery using wavelet spectral analysis," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 41, no. 4, 2003.
- [14] E. El-Araby, T. El-Ghazawi, J. Le Moigne, and K. Gaj, "Wavelet spectral dimension reduction of hyperspectral imagery on a reconfigurable computer," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, pp. 399–402, Brisbane, Australia, December 2004.
- [15] J. Wickmann, "A wavelet approach to dimension reduction and classification of hyperspectral data," Masters Thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, Oslo, Norway, 2007.
- [16] A. Fijany and C. P. Williams, *Quantum Wavelet Transforms: Fast Algorithms and Complete Circuits*, 1998, <http://arxiv.org/abs/9809004v1>.
- [17] J. Stajic, "The future of quantum information processing," *Science*, vol. 339, no. 6124, p. 1163, 2013.
- [18] S. Heidari, M. Naseri, R. Gheibi, M. Baghfalaki, M. R. Pourarian, and A. Farouk, "A new quantum watermarking based on quantum wavelet transforms," *Communications in Theoretical Physics*, vol. 67, no. 6, p. 732, 2017.
- [19] L. Hai-Sheng, P. Fan, H. Xia, S. Song, and X. He, "The multi-level and multi-dimensional quantum wavelet packet transforms," *Scientific Reports*, vol. 8, no. 1, 2018.
- [20] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, Cambridge, UK, 2010.

- [21] N. Mahmud and E. El-Araby, "A scalable high-precision and high-throughput architecture for emulation of quantum algorithms," in *Proceedings of the 31st IEEE International System-on-Chip Conference (SOCC 2018)*, Washington, DC, USA, September 2018.
- [22] N. Mahmud and E. El-Araby, "Towards higher scalability of quantum hardware emulation using efficient resource scheduling," in *Proceedings of the 3rd IEEE International Conference on Rebooting Computing (ICRC 2018)*, Washington, DC, USA, November 2018.
- [23] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings of the 35th IEEE Annual Symposium on Foundations of Computer Science (SFCS '94)*, pp. 124–134, Santa Fe, NM, USA, November 1994.
- [24] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of computing (STOC '96)*, pp. 212–219, Philadelphia, PA, USA, May 1996.
- [25] DirectStream LLC, 2019, <https://directstream.com>.
- [26] Quantum Computing Report, *Qubit Technology*, 2019, <https://quantumcomputingreport.com/scorecards/qubit-technology/>.
- [27] L. Gomes, "Quantum computing: both here and not here," *IEEE Spectrum*, April 2019, <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=83-22045>.
- [28] C. P. Williams, *Explorations in Quantum Computing*, Springer Science & Business Media, Berlin, Germany, 2010.
- [29] L. Grover and T. Rudolph, *Creating Superpositions that Correspond to Efficiently Integrable Probability Distributions*, 2002, <http://arxiv.org/abs/0208112>.
- [30] P. Kaye and M. Mosca, *Quantum Networks for Generating Arbitrary Quantum States*, pp. 1–3, 2004, <http://arxiv.org/abs/0407102>.
- [31] X. Yao, H. Wang, Z. Liao et al., "Quantum image processing and its application to edge detection: theory and experiment," *Physical Review X*, vol. 7, no. 031041, 2017.
- [32] S. G. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 7, pp. 674–693, 1989.
- [33] B. Toufik and N. Mokhtar, "The wavelet transform for image processing applications," in *Advances in Wavelet Theory and Their Applications in Engineering, Physics and Technology*, pp. 395–422, InTech, Rijeka, Croatia, 2012.
- [34] P. S. Addison, *The Illustrated Wavelet Transform Handbook: Introductory Theory and Applications in Science, Engineering, Medicine and Finance*, CRC Press, Boca Raton, FL, USA, 2017.
- [35] D. Gosal and W. Lawton, *Quantum Haar Wavelet Transforms and Their Applications*, 2001.
- [36] H. Ohnishi, H. Matsueda, and L. Zheng, "Quantum wavelet transform and matrix factorization," in *Proceedings of the IEEE International Quantum Electronics Conference*, pp. 1327–1328, Antwerp, Belgium, 2005.
- [37] M. Terraneo and D. L. Shepelyansky, "Imperfection effects for multiple applications of the quantum wavelet transform," *Physical Review Letters*, vol. 90, no. 25, 2003.
- [38] X.-H. Song, S. Wang, S. Liu, A. A. Abd El-Latif, and X.-M. Niu, "A dynamic watermarking scheme for quantum images using quantum wavelet transform," *Quantum Information Processing*, vol. 12, no. 12, pp. 3689–3706, 2013.
- [39] Y.-G. Yang, P. Xu, J. Tian, and H. Zhang, "Analysis and improvement of the dynamic watermarking scheme for quantum images using quantum wavelet transform," *Quantum Information Processing*, vol. 13, no. 9, pp. 1931–1936, 2014.
- [40] N. Ilic, "The ATLAS fast tracker and tracking at the high-luminosity LHC," *Journal of Instrumentation*, vol. 12, no. 2, 2017.
- [41] N. Mahmud, E. El-Araby, and D. Caliga, "Scaling reconfigurable emulation of quantum algorithms at high precision and high throughput," *Quantum Engineering*, vol. 1, no. 2, 2019.
- [42] R. K. Brylinski and G. Chen, *Mathematics of Quantum Computation*, CRC Press, Boca Raton, FL, USA, 2002.
- [43] E. El-Araby, T. El-Ghazawi, J. Le Moigne, and R. Irish, "Reconfigurable processing for satellite on-board automatic cloud cover assessment," *Journal of Real-Time Image Processing*, vol. 4, no. 3, pp. 245–259, 2009.
- [44] E. El-Araby, S. G. Merchant, and T. El-Ghazawi, "Assessing productivity of high-level design methodologies for high-performance reconfigurable computers," in *High-Performance Computing Using FPGAs*, W. Vanderbauwhede and K. Benkrid, Eds., pp. 719–745, Springer, New York, NY, USA, 2013.
- [45] A Preview of Bristlecone, Googles New Quantum Processor, Google AI Blog, March 2018.
- [46] IBM Announces Advances to IBM Q Systems & Ecosystem, IBM Press Release, November 2017.
- [47] B. Villalonga, D. Lyakh, S. Boixo et al., "Establishing the quantum supremacy frontier with a 281 Pflop/s simulation," 2019, <http://arxiv.org/abs/1905.00444v1>.
- [48] M. Fujishima, "FPGA-based high-speed emulator of quantum computing," in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT 2003)*, Tokyo, Japan, December 2003.
- [49] A. U. Khalid, Z. Zilic, and K. Radecka, "FPGA emulation of quantum circuits," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 04)*, pp. 310–315, San Jose, CA, USA, October 2004.
- [50] M. Aminian, M. Saeedi, M. S. Zamani, and M. Sedighi, "FPGA-based circuit model emulation of quantum algorithms," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI '08)*, pp. 399–404, Montpellier, France, April 2008.
- [51] Y. H. Lee, M. Khalil-Hani, and M. N. Marsono, "An FPGA-based quantum computing emulation framework based on serial-parallel architecture," *International Journal of Reconfigurable Computing*, vol. 2016, Article ID 5718124, 18 pages, 2016.
- [52] A. Silva and O. G. Zabaleta, "FPGA quantum computing emulator using high level design tools," in *Proceedings of the Eight Argentine Symposium and Conference on Embedded Systems (CASE'17)*, pp. 1–6, Buenos Aires, Argentina, August 2017.
- [53] J. Pilch and J. Długopolski, "An FPGA-based real quantum computer emulator," *Journal of Computational Electronics*, pp. 1–14, 2018.
- [54] R. Li, B. Wu, M. Ying, X. Sun, and G. Yang, "Quantum supremacy circuit simulation on Sunway TaihuLight," 2018, <http://arxiv.org/abs/1804.04797>.

Research Article

Translating Timing into an Architecture: The Synergy of COTSon and HLS (Domain Expertise—Designing a Computer Architecture via HLS)

Roberto Giorgi ¹, Farnam Khalili ^{1,2}, and Marco Procaccini ¹

¹Department of Information Engineering and Mathematics, University of Siena, Siena, Italy

²Department of Information Engineering, University of Florence, Florence, Italy

Correspondence should be addressed to Roberto Giorgi; giorgi@dii.unisi.it

Received 6 May 2019; Accepted 20 September 2019; Published 3 November 2019

Academic Editor: Wim Vanderbauwhede

Copyright © 2019 Roberto Giorgi et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Translating a system requirement into a low-level representation (e.g., register transfer level or RTL) is the typical goal of the design of FPGA-based systems. However, the Design Space Exploration (DSE) needed to identify the final architecture may be time consuming, even when using high-level synthesis (HLS) tools. In this article, we illustrate our hybrid methodology, which uses a frontend for HLS so that the DSE is performed more rapidly by using a higher level abstraction, but without losing accuracy, thanks to the HP-Labs COTSon simulation infrastructure in combination with our DSE tools (MYDSE tools). In particular, this proposed methodology proved useful to achieve an appropriate design of a whole system in a shorter time than trying to design everything directly in HLS. Our motivating problem was to deploy a novel execution model called data-flow threads (DF-Threads) running on yet-to-be-designed hardware. For that goal, directly using the HLS was too premature in the design cycle. Therefore, a key point of our methodology consists in defining the first prototype in our simulation framework and gradually migrating the design into the Xilinx HLS after validating the key performance metrics of our novel system in the simulator. To explain this workflow, we first use a simple driving example consisting in the modelling of a two-way associative cache. Then, we explain how we generalized this methodology and describe the types of results that we were able to analyze in the AXIOM project, which helped us reduce the development time from months/weeks to days/hours.

1. Introduction

In recent decades, applications are becoming more and more sophisticated and that trend may continue in the future [1–3]. To cope with the consequent system design complexity and offer better performance, the design community has moved towards design tools that are more powerful. Today, many designs rely on FPGAs [4, 5] in order to achieve higher throughput and better energy efficiency, since they offer spatial parallelism on the portion of application characterized by data-flow concurrent execution. FPGAs are becoming more capable to integrate quite large designs and can implement digital algorithms or other architectures such as soft processors or specific accelerators [5]. For the efficient use of FPGAs, it is essential to have an appropriate toolchain. The toolchain provides an

environment in which the user can define, optimize, and modify the components of the design, by taking into account the power, performance, and cost requirements of a particular system and eventually synthesize and configure the FPGA.

The conventional method to implement an application code on FPGAs is to write the code in Hardware Description Language (HDL) (e.g., VHDL or Verilog). Although working with HDL languages still is the most reliable and detailed way of designing the underlying hardware for accelerators, their use requires advanced expertise in hardware design as well as remarkable time. The Design Space Exploration (DSE) and debugging time of FPGAs and the bitstream generation may reach many hours or days even with powerful workstations. As such, moving an already-validated architecture to the FPGA's tool flow may save

significant time and effort and, as a result, facilitates the design development.

This situation is exacerbated by the interaction with the Operating Systems and by the presence of multicore. Therefore, the use of full-system simulators in combination with HLS tools permits a more structured design flow. In such a case, a simulator can preliminarily validate an architecture and the HLS-to-RTL time is repeated less times.

There are parameters which make simulators preferable to reach a certain level of performance, scalability, and accuracy as well as reproducibility and observability. Based on the experience of previous projects such as TERAFLUX [6, 7], ERA [8, 9], AXIOM [4, 10–13], and SARC [14, 15], we choose to rely on the HP-Labs COTSon simulation infrastructure [16]. The key feature of COTSon that is useful in HLS design is its “functional-directed” approach, which separates the functional simulation from the timing one. We can define custom timing models for any component of an architecture (e.g., FPGA, CPU, and caches) and validate them through the functional execution; however, the actual architecture has to be specified by a separate “timing model” (see Section 2 for more details). The latter is what can be migrated in a straightforward way to HLS. Moreover, COTSon is a full-system simulator; hence, it permits to study the OS impact on the execution and choose the best OS configuration based on the application requirements [17]. The OS modelling is sometimes not available in other tools (reviewed in Section 2).

In this article, we illustrate the importance of the simulation in synergistic combination with the Xilinx HLS tool [18], in order to permit a faster design environment, while providing a full-system Design Space Exploration (DSE).

Additionally, thanks to our DSE toolset (MYDSE) [17, 19], we facilitate the extraction of not only important metrics such as the execution time but also more detailed ones such as cache miss rates and bus traffics, which help investigate the appropriate system design. In order to illustrate our methodology, we start from a driving example related to design a simple two-way associative cache system. The methodology is then generalized by considering the case of the AXIOM project, in which this methodology was actually used to design and implement a novel data-flow architecture [20–22] through the development of our custom AXIOM board [11].

The contributions of this work are as follows:

- (i) Presenting our methodology for designing FPGA-based architectures, which consists in the direct mapping of COTSon “timing models” into HLS, where such models are pre-verified via our MYDSE tools: the DSE is performed before using the HLS tools, thus saving much design time
- (ii) Illustrating a simple driving example based on the modelling and synthesis of a simple two-way set-associative cache in order to grasp the details of our methodology
- (iii) Presenting the bigger picture of using our proposed methodology to design a whole software/hardware platform (called AXIOM)

The rest of the article is organized as follows: in Section 2, we analyze related work; in Section 3, we illustrate our methodology and tools; in Section 4, we provide a simple driving case study; in Section 5, we show the possibilities of our tools in the more general context of the AXIOM project.

2. Related Work

Our design and evaluation methodology aims at integrating simulation tools and HLS tools to ease the hardware acceleration of applications, via custom programmable logic. HLS tools improve design productivity as they may provide a high level of abstraction for developing high-performance computing systems. Most typically, these tools allow users to generate a RTL representation of a specific algorithm usually written in C/C++ or SystemC. Several options and features are included in these tools in order to provide an environment with a set of directives and optimizations that help the designer meet the overall requirements. In our case, we realized that more design productivity could be achieved by identifying in the early stages a candidate architecture through the use of a simulator: however, *the use of a generic simulator may not help identify the architecture, since often the simulation model is too distant from the actual architecture or is too much intertwined with the modelling tool* [23–26]. On the other hand, *the COTSon simulator uses a different approach, called “functional-directed” simulation, in which the functional and timing models are neatly separated and the first one drives the latter* (It is important to note that the “timing model” implicitly defines an architecture, which is functionally equivalent to the “functional model,” but it is a totally separated code with different simulation speeds [16].). The similarity of our “timing model” specification to an actual architecture is an important feature and it is the basis for our mapping to a HLS specification.

In our research, we used Xilinx Vivado HLS, but other important HLS frameworks are available and are briefly illustrated in the following; their main features are summarized in Table 1. LegUp [27] supports C/C++, Pthreads, and OpenMP as programming models for HLS [35] by leveraging the LLVM compiler framework [36], and permits parallel software threads to run onto parallel hardware units. LegUp can generate customized heterogeneous architectures based on the MIPS soft processor. Bambu [28] is a modular open-source HLS tool, which aims at the design of complex heterogeneous platforms with a focus on several trade-offs (such as latency versus resource utilization) as well as partitioning on either hardware or software. GAUT [29] is devoted to real-time digital signal processing (DSP) applications. It uses SystemC for automatic generation of test-benches for more convenient prototyping and design space exploration. DWARV [30] supports a wide range of applications such as DSP, multimedia, and encryption. The compiler used in DWARV is the CoSy commercial infrastructure [37], which provides a robust and modular foundation extensible to new optimization directives. Stratus HLS of Cadence [31] is a powerful commercial tool accepting C/C++ and SystemC and targeting a variety of platforms, including FPGAs, ASICs, and SoCs. Thanks to

TABLE 1: Key features of discussed HLS tools.

Tool	Owner	License	Input	Output	Domain	Testbench	SW/ HW	Simulation	Floating point	Fixed point
LegUp [27]	LegUp computing	Commercial	C, C++	Verilog	All	Yes	Yes	HW	Yes	No
Bambu [28]	Politecnico di Milano	Academic	C	VHDL, Verilog	All	Yes	Yes	SW, HW	Yes	No
GAUT [29]	U. Bretagne sud	Academic	C, C++	VHDL, SystemC	DSP	Yes	No	HW	No	Yes
DWARV [30]	TU delft	Academic	C	VHDL	All	Yes	Yes	HW	Yes	Yes
Stratus HLS [31]	Cadence	Commercial	C, C++, SystemC	C, C++, SystemC	All	Yes	Yes	SW, HW	Yes	Yes
Intel HLS compiler [32]	Intel	Commercial	C, C++	Verilog	All	No	No	SW, HW	Yes	Yes
Vivado HLS [18]	Xilinx	Commercial	C, C++, OpenCL, SystemC	VHDL, Verilog, SystemC	All	Yes	No	SW, HW	Yes	Yes
SDSoC [33]	Xilinx	Commercial	C, C++	VHDL, Verilog	All	No	Yes	SW, HW	Yes	Yes
SDAccel [34]	Xilinx	Commercial	C, C++, OpenCL	VHDL, Verilog, SystemVerilog	All	Yes	Yes	SW, HW	Yes	Yes

For the nonobvious columns, *Testbench* means the capability of automatic testbench generation. *SW/HW* means the support for the software/hardware co-design environment. *Floating Point* and *Fixed Point* are the supported data types for the arithmetic operations.

low power optimization directives, the user can achieve a consistent power reduction. It gives support for both control flow and data-flow designs, and actively applies constraints to trade-off speed, area, and power consumption. The Intel HLS compiler [32] accepts ANSI C/C++ and generates RTL for Intel FPGAs, which is integrated into the Intel Quartus Prime design software. Xilinx Vivado HLS tool targets Xilinx FPGAs [18], which offers a subset of optimization techniques, including loop unrolling, pipelining, data flow, data packing, function inline, and bit-width reduction for improving the performance and resource utilization.

Xilinx SDSoC is a comprehensive automated development environment for accelerating embedded applications [33]. The tool can generate both RTL level and the software running on SoC cores for the “bare-metal” libraries, Linux, and FreeRTOS. Xilinx SDAccel [34] aims at accelerating functionalities in data centers through FPGA resources. We summarize the key features of the aforementioned HLS tools in Table 1.

Although some of the HLS tools provide a general software/hardware simulation framework, the possibility of easily evaluating a complex architecture-oriented design (e.g., computer organization: level and size of caches, number of cores/nodes, and memory hierarchy) is still missing. Moreover, before reaching a bug-free physical design, which meets all the design specifications, the debug and development of such designs by using the aforementioned HLS tools may require a significant time and effort despite all benefits that HLS tools provide to the design community. Consequently, powerful design frameworks that simplify the verification of the design and provide an easy design space exploration are welcome. In this respect, many design frameworks have emerged to implement efficient hardware in less time and effort. Authors in [38] propose a framework relying on Vivado HLS to efficiently map processing specifications expressed in PolyMageDSL to

FPGA. Their framework supports optimizations for the memory throughput and parallelization. ReHLS [39] is a framework with automated source-to-source resource-aware transformation leveraging Vivado HLS tool. Their framework improves the resource utilization and throughput by identifying the program inherent regularities that are invisible to the HLS tool. FROST [40] is a framework that generates an optimized design for the HLS tool. This framework is mainly appropriate for applications based on streaming data-flow architectures such as image-processing kernels.

However, whereas these tools focus on optimizing the whole application performance, we are proposing instead an architecture-oriented approach, where the designer can manipulate and explore the architecture itself, before passing it to the HLS toolchain. By using our proposed framework (see Section 4 for more details), we can validate the design in terms of the functional and timing models, and then define a specific architecture, while constantly monitoring the selected key performance metrics. The architecture model is specified in C/C++ and, thanks to the decoupling from the simulation details and functional model, it can be easily migrated into the HLS description. This is illustrated in Sections 4 and 5. In particular, we leverage the Vivado HLS tool and on top of it, we build our design space exploration tools relying on COTSon simulator, which is one of the key components of our framework. In the following, we highlight relevant features and compare several simulators (Table 2), and we contrast them with our chosen simulator (i.e., COTSon).

SlackSim [23] is a parallel simulator to model single-core processors. SimpleScalar [24] is a sequential simulator, which supports single-core architectures at the user level. GEMS [25] is a virtual machine-based full-system multicore simulator built on top of Intel’s Simics virtual machine. GEMS relies on timing-first simulation approach, where its

TABLE 2: Interesting features of simulators for high-performance computing architectures. For the nonobvious columns, *Parallel/Sequential* means the simulator core can be executed either in parallel or sequential by the host processor. *Full System* means taking into account all events, including the OS.

Simulator	Parallel/sequential	Single-core/multicore	Full system	Simulation methodology
COTSon [16]	Parallel	Multicore	Yes	Decoupled-functional first
GEMS [25]	Sequential	Multicore	Yes	Decoupled- timing first
Graphite [41]	Parallel	Multicore	No	Not decoupled- trace-driven
SimpleScalar [24]	Sequential	Single-core	No	Not decoupled- execution driven
MPTLsim [26]	Sequential	Multicore	No	Not decoupled- timing first
SlackSim [23]	Parallel	Single-core	No	Not decoupled- timing first

timing model drives one single instruction at a time. Even though GEMS provides a complete simulation environment, we found that COTSon simulator provides better performance as we increase the number of modelled cores and nodes. MPTLsim [26] is a full-system x86-64 multicore cycle-accurate simulator. In terms of simulation rate, MPTLsim is significantly faster than GEMS. MPTLsim takes advantage of a real-time hypervisor scheduling technique [42] to build hardware abstractions and fast-forward execution. However, during the execution of hypervisor, the simulator components, such as memory, instructions, or I/O, are opaque to the user (no statistics is available). On the contrary, e.g., COTSon provides an easily configurable and extensible environment to the users [43] with full detailed statistics. Graphite [41] is an open-source distributed parallel simulator leveraged in the PIN package [44], with the trace-driven functionalities. COTSon permits full-system simulation from multicore to multinode and the capability of network simulation, which makes COTSon a complete simulation environment. Both COTSon and Graphite permit large core numbers (e.g., 1000 cores) with reasonable speed, but COTSon provides also the modelling of peripherals such as disk and Ethernet card. Compared to COTSon, the above simulators do not express a timing model in a way that can be easily ported to HLS: COTSon is based on the “functional-directed” simulation [16], which means that the functional part drives the timing part and the two parts are completely separated, both in the coding and during the simulation. The functional model is very fast but does not include any architectural detail, while the timing model is an architecturally complete description of the system (and, as such, includes also the actual functional behaviour, of course). In this way, once the timing model is defined and the desired level of the key performance metric (e.g., power or performance) has been reached, the design can be easily transported to an HLS description, as illustrated in the next sections.

3. Methodology

In this section, we present our methodology (Figure 1) for developing hardware components for a reconfigurable platform, as developed in the context of the AXIOM project.

First, we define the functional and the timing model of a desired architectural component (e.g., a cache system, as described in Section 4). Such models are described by using C/C++ (two orange blocks in the top left part of Figure 1).

These models are then embedded in the COTSon simulator, which is managed in turn by the MYDSE tools in order to perform the design space exploration [16, 17, 19]. The latter is a collection of different tools, which provide a fast and convenient environment to simulate, debug, optimize, and analyze the functional and timing models of a specific architecture and to select the candidate design to be migrated to the HLS (top part of Figure 1).

Afterwards, we manually migrate a validated architecture specification from COTSon to the Vivado HLS tool (bottom part of Figure 1), where the user can apply the specific directives defined in the timing model of COTSon into the Vivado HLS. This is possible because of the close syntax of the architecture specification in COTSon and Vivado HLS.

Our framework has the purpose of reducing the total DSE time to define an architecture (as input to Vivado HLS itself). We do not aim to define a precise RTL, but simply to select an architecture suitable as input to Vivado HLS (see Figure 2).

Finally, we pass the generated bitstream by Vivado to the XGENIMAGE, which is a tool that assembles all needed software, including drivers, applications, libraries, and packages, in order to generate the operating system full image to be booted on the AXIOM board. In Figure 1, we highlight *in green* the existing (untouched) tools and *in blue* the research tools that we developed from scratch or that we modified (like COTSon). In our case, part of the process involves the design of the FPGA board (the AXIOM board). An important capability of the board is also to provide fast and inexpensive clusterization. The simulator allowed us to model exactly this situation, in which the threads are distributed across several boards, through a specific execution model (called DF-Threads). To that extent, the AXIOM board [11] has been designed to include a soft-IP for the routing of data (via RDMA custom messages) and the FPGA transceivers are directly connected to USB-C receptacles, so that four channels at about 18 Gbps are available for simple and inexpensive connection of up to 255 boards, without the need of an external switch [12].

3.1. DSE Toolset and COTSon Simulator. Based on our experience of the AXIOM project [4, 12, 13], the main motivation behind the choice of the COTSon simulation framework resides in the “functional-directed” approach [16]. COTSon also permits to model a complete system like a

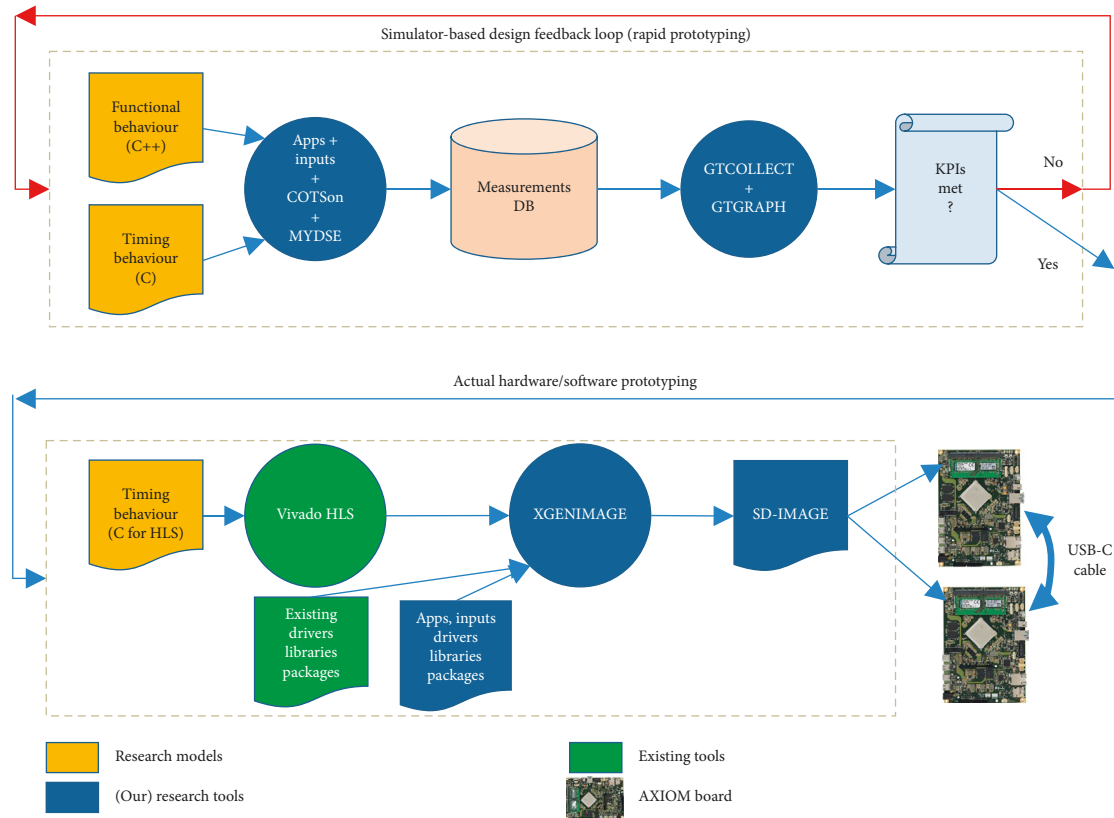


FIGURE 1: Design and test methodology of the AXIOM involved a mix of simulation (via the COTSon simulator and other custom tools) and FPGA prototyping (via our custom AXIOM board and hardware synthesis tools (like Vivado HLS)) [11].

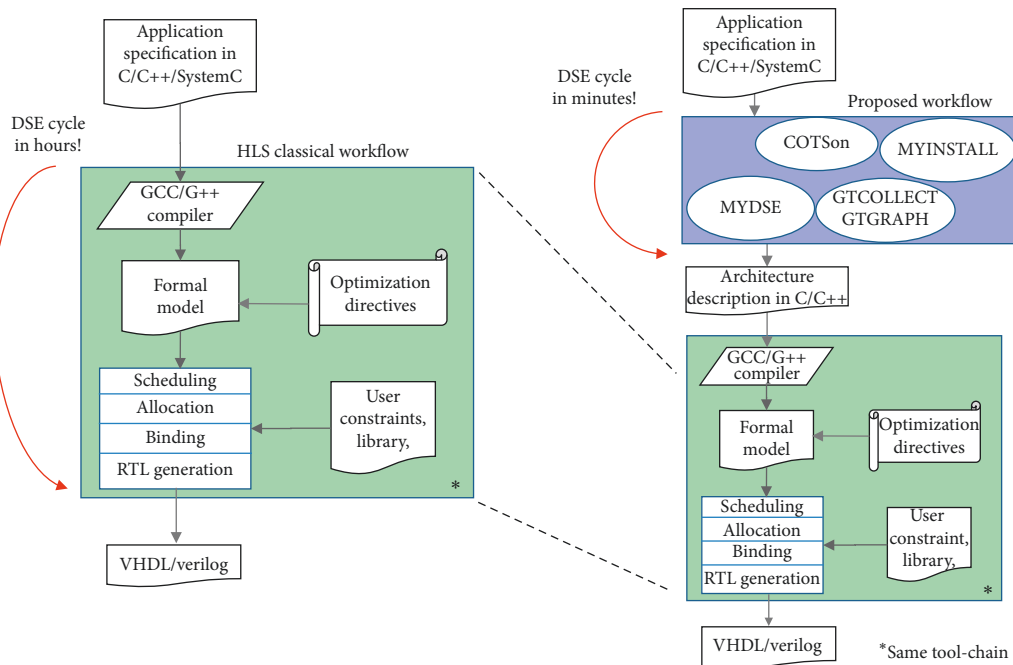


FIGURE 2: Differences between classical and proposed architecture modelling frameworks. Workflows to generate VHDL/Verilog hardware description language from the application specification written in C/C++. On the left, a typical workflow of existing HLS tools. On the right, we leverage the HLS tool, and on top of it, we build our framework to simulate and validate the design specification.

cyber-physical system (CPS), i.e., including the possibility of running a real software performing Input/Output (I/O) and an off-the-shelf Linux distribution (or other operating systems). Since the performance of a CPS is affected also by the Operating System (OS) and libraries [17], it is important to model not only the memory hierarchy and cores but also all the devices of the system: this is possible in the COTSon framework.

In Section 5.1, we show that the OS influence can be detected earlier in the DSE by using our methodology. Moreover, COTSon permits building a complete distributed system with multi-cores and multiple nodes, where we can observe and analyze any aspect of the application and, e.g., the OS activity. In order to guarantee a proper scientific methodology for studying the experimental results that are coming from the framework, we designed a DSE toolset (called “MYDSE”) [19], through which it is possible to easily set up a distributed simulation, as well as automatically extract, calculate the appropriate averages, and examine the key metrics. MYDSE addresses the designer’s needs mostly on the first part of the workflow represented on the top of Figure 1.

Moreover, MYDSE represents a higher abstraction level in the design (Figure 3), in which existing architectural blocks (e.g., caches) can be combined and parameterized for a preliminary design exploration. The MYDSE phase permits us to answer questions such as the following: “How large should be the cache in the target platform?” “How many cores I need in my design?” “What would be the overhead of distributing the computation across several FPGAs?”

3.2. COTSon Framework. In this subsection, we briefly summarize the features of the essential component of our toolchain—the COTSon—for the sake of a more self-contained illustration of our framework. More details can be found in [16, 17, 19, 43].

The COTSon framework has been initially developed by HP-Labs and its simulation core is based on the AMD SimNow virtualization tool, which is an x86_64 virtual machine provided by AMD to test and develop their processors and platforms [16]. COTSon relies on the so-called functional-directed simulation approach, where the functional execution (top part of Figure 4) runs in the SimNow Virtual Machine (VM) and the detailed timing (bottom part of Figure 4) is totally decoupled and reconstructed dynamically based on the events coming from the functional execution.

COTSon can also model a distributed machine composed of several nodes: each SimNow VM models a complete multicore node with all its peripherals, and an additional component (called “Mediator”), which models a network switch. The virtual machines can run in parallel, thus speeding up a simulation consisting of several nodes. Moreover, we can use different available simulation acceleration techniques, such as dynamic sampling or SMARTS [46], and perform other accounting activities, such as tracing, profiling, and (raw) statistic collection. The instruction stream coming out from each SimNow functional

core is interleaved for a correct time ordering. The COTSon control interface extracts the instruction stream, passing it to the timing simulation (Figure 4).

In the “Timing Simulation” portion of the COTSon (see the bottom part of Figure 4), we can model any architectural components (i.e., CPU, L1 cache, network switch, accelerator, etc.) with a few lines of C++ codes. The architecture of the modelled system is customizable by setting all the relevant information in a configuration file (written in the Lua scripting language) [47] as illustrated in the bottom part of Figure 3). Other aspects of the simulation can be customizable as well in the configuration file: e.g., the sampling method, how to log statistics, and which kind of Operating System (OS) image to use.

3.3. DSE Toolset. In this subsection, we describe the tools that we have designed for the DSE. A detailed overview of these tools has been introduced in a previous work [19]. Here, we recall the main features.

Design Space Exploration (DSE) and its automation are a significant part of modern performance evaluation and estimation methodologies to find an optimal solution among the many design options, while respecting several constraints on the system (e.g., a certain level of performance and energy efficiency).

In order to facilitate and speed up the DSE, we developed a set of tools (called “MYDSE”), through which it is possible to easily configure the relevant aspects of our simulation framework and automate the routine work. Thanks to MYINSTALL, a tool included in the MYDSE, the installation and validation phase of the overall environment (which was previously taking a lot of human effort and many hours of work) now takes less than 10 minutes, minimizing the human interaction and giving us the possibility of setting up several host machines in a fast and easy way. At the end of the installation phase, a set of regression tests is performed to verify if the software is correctly patched, compiled, and installed (see Figure 5–left). This permits a fast deployment of different machines with possibly different characteristics and, at the same time, has a monitoring of the actual resources that are available for an optimal utilization of them.

Another critical aspect of the simulation is the automatic management of experiments, mostly in the case when a large number of design points need to be explored: this is managed by the MYDSE tool. Using a small configuration file, we can define the Design Space of an experiment by using a simple scripting syntax (In our case, we refer to “bash.” “bash” is a popular scripting language for Linux.): `<key> = <value>`. For example, it is possible to define not only a modelled architecture (e.g., number and types of cores, cache parameters, and multiple levels of caches), the Operating System image, and other parameters of the COTSon simulator, but also other higher level parameters related to the applications, their inputs, and the standard libraries to be used. The MYDSE configuration file also permits listing a set of values for each parameter so that the design points are automatically generated. Once the design points are generated, the tools manage the execution of the

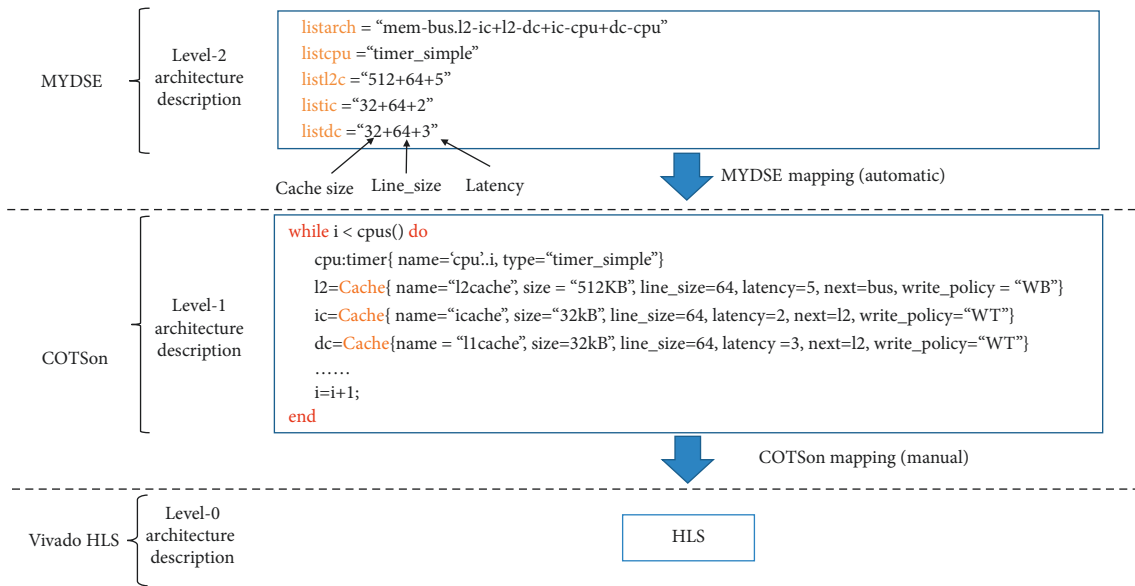


FIGURE 3: The relation between the higher level MYDSE description, the COTSon configuration file, and the final HLS translation: At the higher level, we specify the parameters in a compact way (level 2 architecture description), and we can indicate several instances of such parameters so that MYDSE can generate the design space points to be explored. In the COTSon configuration (level 1), the MYDSE points will be automatically mapped to the parameter of the corresponding architectural element (bottom part of the figure: the “next” field specifies the position in the architecture tree, WB means write-back, and WT is the write-through policy). Finally, the architecture description is mapped manually from COTSon description to HLS (level 0).

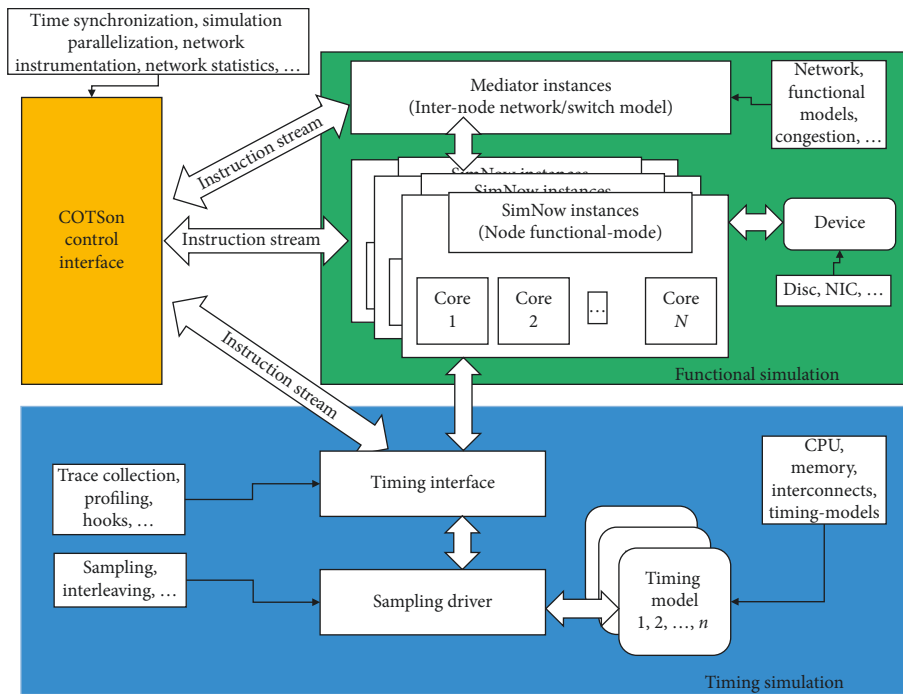


FIGURE 4: The COTSon simulation framework architecture [19, 45].

experiments by scheduling and distributing the simulations on, e.g., a cluster of simulation hosts, by collecting the results of each simulation and inserting them in a database, where off-line data mining can be performed afterwards. Moreover, the tools constantly monitor the simulations: if one of them is failing, then it is automatically retried (thresholds are applied to limit the re-trials).

A large number of output statistics are produced during the simulations; thus, a database is necessary to store such data. Statistical processing can also be selected to give a quantification of the goodness of the collected numbers (e.g., the coefficient of variation and the presence of outliers). Other tools in Figure 5 (GENIMAGE, ADD-IMAGE, GTCOLLECT, and GTGRAPH) are described in more detail in [19].

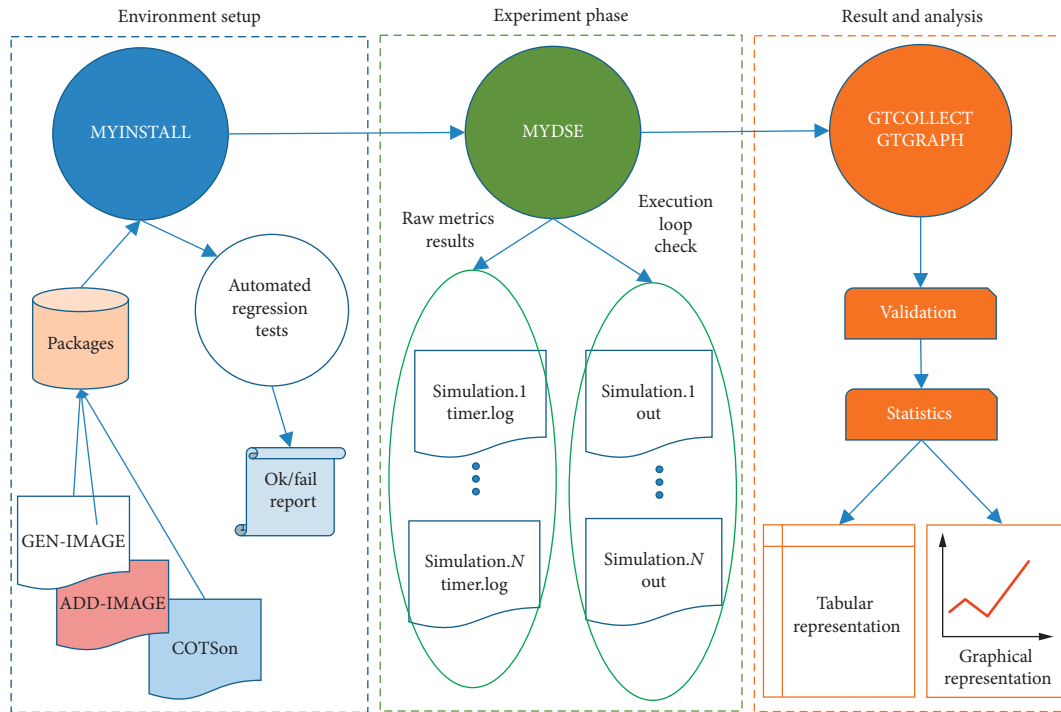


FIGURE 5: Tool flow of the Design Space Exploration tool. The MYINSTALL tool prepares the whole environment and performs automated regression tests in the end. The MYDSE tool takes care of the experiment loop and the reordering of several output files generated by each simulation. Finally, the GTCOLLECTS and GTGRAPH tools collect the results, perform validation and statistical operations on the results, and plot the data in a tabular or graphical format [19].

3.4. *Mapping an Architecture to HLS.* High-level synthesis (HLS) aims at enhancing design productivity via facilitating the translation from the algorithmic level to RTL (register transfer level) [48, 49]. In the current state of the art, given an application written in a language like C/C++ or SystemC, an HLS tool particularly performs a set of successive tasks to generate the corresponding register transfer level (RTL, e.g., VHDL or Verilog) description suitable for a reconfigurable platform, such as an FPGA [49] (Figure 2-left). This workflow typically involves the following steps:

- (i) Compiling the C/C++/SystemC code to formal models, which are intermediate representations based on control flow graph and data-flow graph.
- (ii) Scheduling each operation in the generated graph to the appropriate clock cycles. Operations without data dependencies could be performed in parallel, if there are enough hardware resources during the desired cycle.
- (iii) Allocating available resources (LUTs, BRAMS, FFs, DSPs, and so on) in regard to the design constraints. For instance to enhance the parallelism, different resources could be statistically allocated at the same cycle without any resource contention.
- (iv) Binding each operation to the corresponding functional units, and binding the variables and

constants to the available storage units as well as data paths to data buses.

- (v) Generating the RTL (i.e., VHDL or Verilog).

All these operations continue to be performed in our proposed framework, but the designer would like to avoid excessive iterations through them, since they may require many hours of computing processing or even more, depending on the complexity of the design, even on powerful workstations and with not-so-big designs. However, COTSon and MYDSE tools (illustrated above) act like a “front-end” to the HLS tool, as outlined in Figure 2. We use HLS also for defining a specific architecture to accelerate the application. Our tools allow the designer to explore possible options for the architecture, without going to the synthesis step: only when the simulation phase has successfully selected an architecture (output of the blue block in Figure 2), the model will be manually translated by the programmer as an input to the HLS tools. Doing this step automatically is out of the scope of this work.

A comparison of the total time of the DSE loops between our framework (Figure 2-right) and HLS (Figure 2-left) is reported here for different benchmarks (Table 3). For example, a blocked matrix multiplication benchmark (matrix size 864 and block size 8) and a Fibonacci benchmark (order of up to 35) are executed based on our DF-Threads execution model (data-flow model). As a result, thanks to our framework, we were able to reduce the required time for validating and developing the architecture compared with solely HLS

TABLE 3: Comparison of different total DSE time of the classical design workflow for FPGAs (Figure 2-left) and our proposed methodology (Figure 2-right).

Application	HLS + Synthesis (hours) (Figure 2-left)	Our framework (seconds) (Figure 2-right)
2-way cache	3 : 50	5
Blocked matrix multiplication (DF-Threads, matrix size = 864, block size = 8, integer)	4 : 25	8
Fibonacci (DF-Threads, $N = 35$)	1 : 40	8

workflow, through which applying any changes in the source codes may require many hours for the synthesis process.

4. Case Study

In this section, first, we explain our workflow by using a simple and well-known driving example, i.e., the design of a two-way set-associative cache in a reconfigurable hardware platform through our methodology. Afterwards, we illustrate the more powerful capabilities of our framework for a more complex example, which is the design of the AXIOM hardware/software platform. In both cases, first, we design the architecture in the COTSon simulator and then we test its correct functioning and achieve the desired design goals. Finally, we migrate the timing description of the desired architecture into the Xilinx HLS tools.

4.1. From COTSon to Vivado HLS—A Simple Example. In COTSon, the architecture is defined by detailing its “timing model.” A timing model is a formal specification that defines the custom behaviour of a specific architectural or micro-architectural component; in other terms, the timing model defines the architecture itself [16, 19]. The timing model in the COTSon simulator is specified by using C/C++. The designer defines the *storage* by using C/C++ variables (more often structured variables). The timing model *behaviour* is specified by *explicating* into C/C++ statements the steps performed by the control part and *associating them with the estimated latency*, which can be defined through our DSE configuration files (see Figure 3) easily. After defining the model, we can simulate and measure the performance of it. This is illustrated in Figure 6 and discussed in the following paragraphs.

Let us assume here that we wish to design a simple two-way set-associative cache: we show how it is possible to define the timing model of a simple implementation of it in COTSon and then how we can map it in HLS. We start from a conceptual description of such cache, as shown in Figure 6. In particular, for each way of the cache, we need to store the “line” of the cache, i.e., the following information:

- (1) *Valid bit or V-bit (1 bit)*: used to check the validity of the indexed data
- (2) *Modify bit or M-bit (1 bit)*: used to track if data has been modified.
- (3) *LRU bits or U-bits (e.g., 1 bit in this case)*: used to identify the Least Recently Used data between the two cache ways.
- (4) *Tag (e.g., 25 bits)*: used to validate the selected data of the cache.

- (5) *Data (e.g., 512 bits, 64 bytes, or 16 words)*: contains the (useful) data.

The data structure to store this information in COTSon is given by the “Line” structure, which is shown in Figure 7 (left side).

When we want to read or write data, which are stored in a byte address (X in Figure 6), we check if the data are already presented into the cache. The cache controller implements the algorithm to find the data in the cache. Although not visible in the left part of Figure 6, there is a control part also for identifying the LRU block. We can implement this control in COTSon by using the two functions (shown in the right part): one named “find” (Figure 7), which is a simple linear search, and the other named “find_lru” (Figure 8).

From the timing model of the implemented cache in COTSon, we migrate the design into the Xilinx HLS tools. One minor restriction in Vivado HLS is to use fixed size arrays instead of dynamic data structures because of the direct transformation of the structures to the available hardware resources.

The advantage of using our hybrid methodology is that the DSE (see Figure 2 and Table 3) of the architecture of this small cache takes a few seconds in the COTSon, while it takes approximately four hours on a powerful workstation to synthesize and perform the DSE with the HLS version of the same architecture (right side of Figures 7 and 8).

In the next section, we will illustrate how, thanks to our methodology, we were able to reduce significantly the DSE cycles and development time of a relatively large project like the AXIOM project, and produce reliable specification to be implemented on the AXIOM board.

4.2. COTSon Configuration and Timing. One more feature of our environment, based on COTSon + MYDSE, is the capability of easily integrating the modelled components (i.e., the simple two-way set-associative cache of the previous subsection). As we can see in Figure 9, we can build the overall architecture by specifying how to integrate the component in a higher level configuration file (“Level 2” in Figure 9, the “MYDSE” configuration file).

In particular, we define the following simple syntax: the character “-” is the link between two architectural COTSon blocks and the “+” character separates different links between such blocks. The architectural blocks are implicitly defined, since they appear in the link specification. The “.” character serves to replicate a set of architectural blocks, which follow the “.” for m times, where, by default, “ m ” is the number of cores. This is shown in the “listarch” variable of

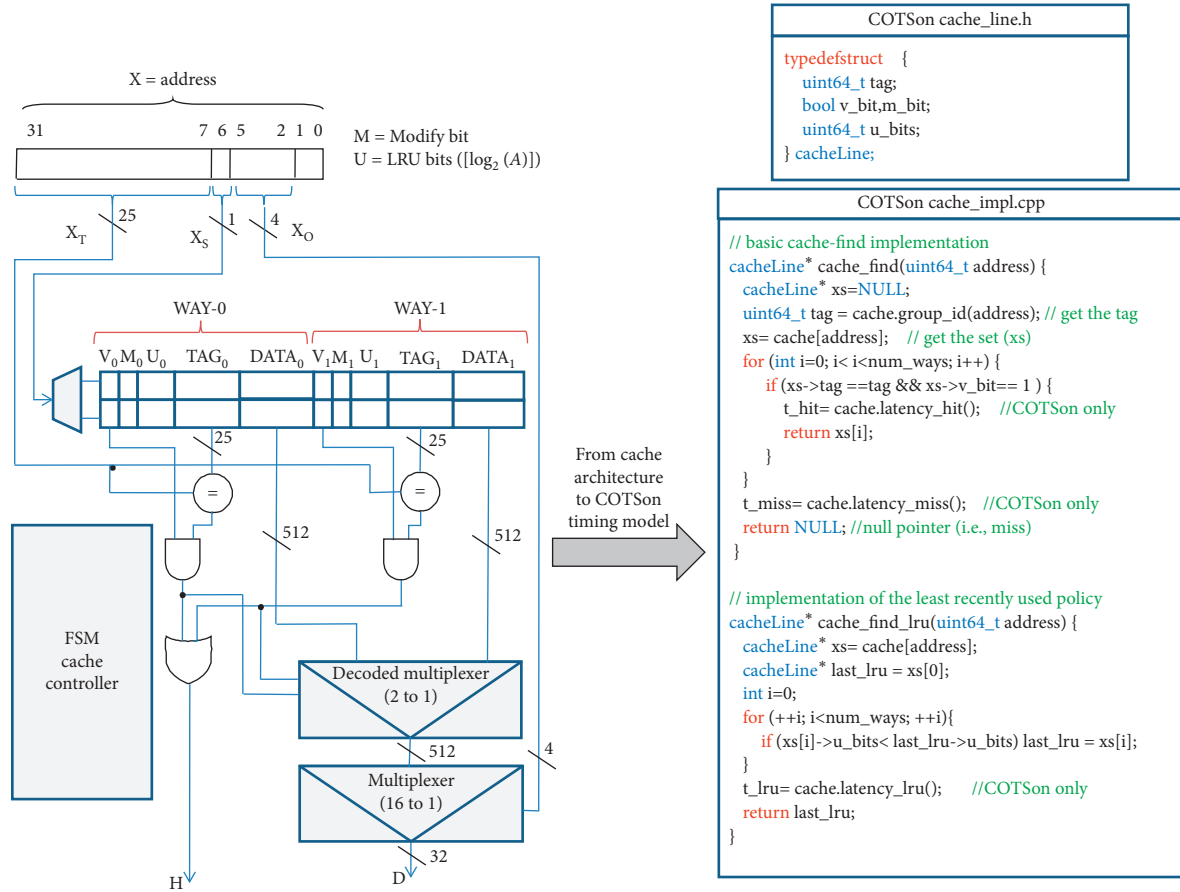


FIGURE 6: Example of the logic scheme of a two-way set-associative cache. Given the byte address X on 32 bits, in this example, the cache indexes four 64-byte blocks (2 words in 2 sets). This implies that the last 6 bits are needed to select a byte inside the block: the first 25 bits of the address (X_T) are used for tag comparison and the remaining 1 bit (X_S) is used for cache set indexing. The cache hit (signal H) is set if the tag of the X is present in the cache at the specified index and if the valid bit is equal to one.

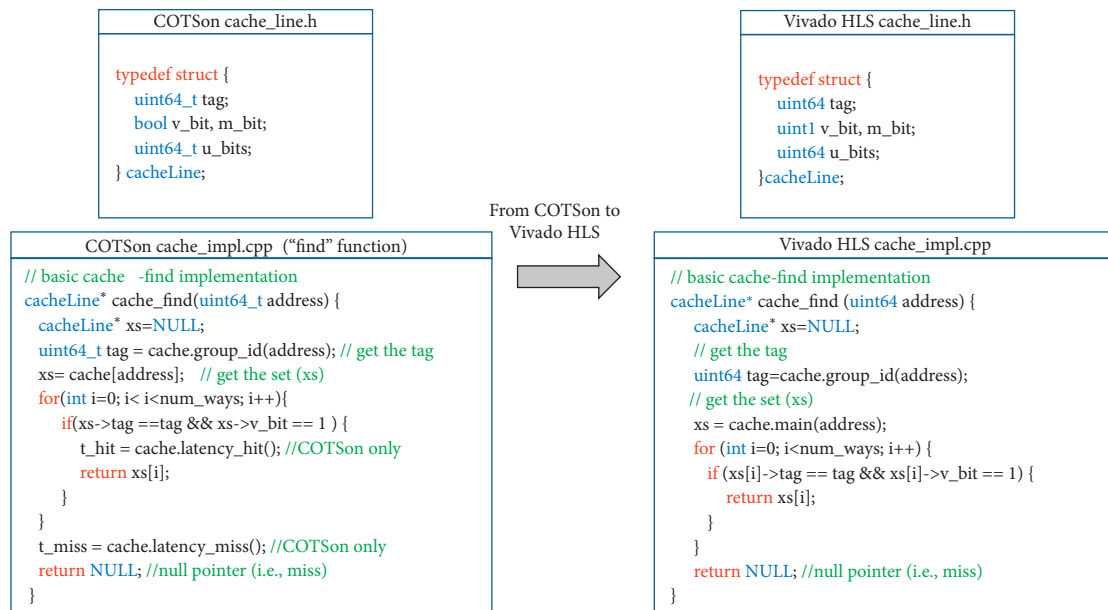


FIGURE 7: Example of the timing model of the cache “find” function, which is translated from the COTSon to the Vivado HLS. The implementation of this function for both the COTSon (left) and Vivado HLS (right) environments is shown in the bottom part of the figure.

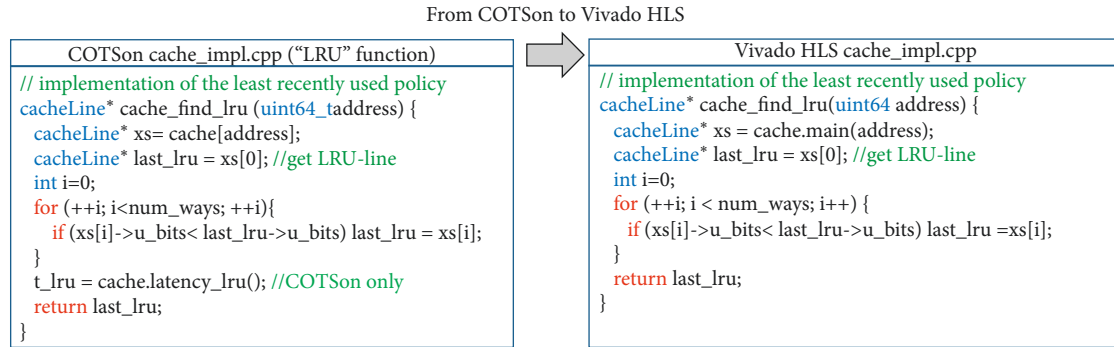


FIGURE 8: Example of translation of the timing model of the LRU (least recently used) function from the COTSon (left side) to the Vivado HLS (right side).

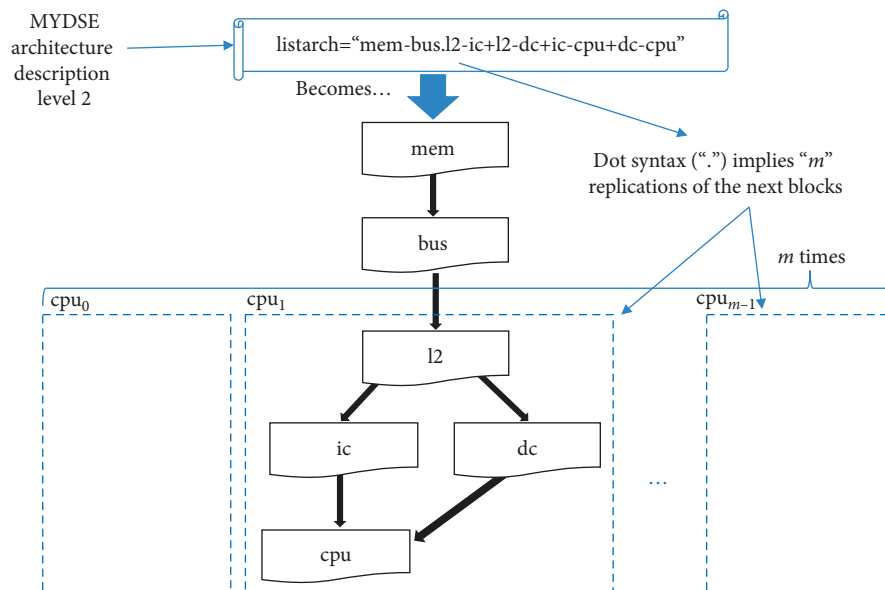


FIGURE 9: Level 2 architecture description of the cache model in COTSon by using the MYDSE toolset. In this design, the CPU is directly connected to both an Instruction Cache ("ic") and a Data Cache ("dc"). The "ic" and "dc" caches are then connected to another level of caching, the L2 cache ("l2"), which is connected to the main memory ("mem") through the "bus."

Figure 9: i.e., the part "l2-ic + l2-dc + ic-cpu + dc-cpu" will be instantiated " m " times.

As depicted in Figure 3, at the higher level, we specify the parameters in an even compact way, and we can indicate several instances of such parameters so that MYDSE can generate the design space points to be explored. In the COTSon configuration, the MYDSE points will be assigned to the parameter of the corresponding architectural element. Moreover, we can specify the latencies of an architectural block, which are used by its timing model for the execution time estimation.

5. Generalization to the AXIOM Project and Evaluation

The aim of the AXIOM project was to define a software/hardware architecture configuration, to build scalable embedded systems, which could allow a distributed computation across several boards by using a transparent scalable method such as the DF-Threads [20–22].

In order to achieve this goal, we rely on RDMA capabilities and a full operating system to interact with the OS scheduler, memory management, and other system resources. Following our methodology, we included the effects of all these features, thanks to the COTSon + MYDSE full-system simulation framework. We will present in the next subsection the results that we were able to obtain through this preliminary DSE phase reasonably quickly.

After the desired software and hardware architecture was selected in the simulation framework, we started the migration to the physical hardware: we had clear evidence that we needed at least the following features:

- (i) Possibility to exchange rapidly data frames via RDMA across several boards: this could be implemented in hardware, thanks to the FPGA high-speed transceivers;
- (ii) Possibility to accelerate portions of the application on the programmable logic (PL), not only on one board but also on multiple FPGA boards: this could

be implemented by providing appropriate network interface IPs in the FPGA.

In this way, we preselected the basic features of the AXIOM board (Figure 10-left) through the COTSon framework and the MYDSE toolset. Then, once the DSE was completed, we migrated the final architecture specification with the Vivado HLS tool into the AXIOM distributed environment (Figure 10-right).

The DF-Threads execution model is a promising approach for achieving the full parallelism offered by multi-core and multi-node systems, by introducing a new execution model, which internally represents an application as a direct graph named data-flow graph. Each node of the graph is an execution block of the application and a block can execute only when its inputs are available [20].

5.1. Designing the AXIOM Software/Hardware Platform.

During the AXIOM project, we analyzed two main real-world applications: Smart Video Surveillance (SVS) and Smart Home Living (SHL) [50]. These applications are very computationally demanding, since they require analyzing a huge number of scenes coming from multiple cameras located, e.g., at airports, home, hotels, or shopping malls.

In these scenarios, we figured out that one of the computationally intensive portions of those applications relies on the execution of the matrix multiplication kernel. For these reasons, the experiment results presented in this section are based on the execution of the block matrix multiplication benchmark (BMM) using the DF-Threads execution model. The BMM algorithm is based on the classical three nested loops, where a matrix is partitioned into multiple submatrices, or blocks, according to the block size.

As we generalized the methodology described in the previous section to the AXIOM project [10–12], we were able to experiment on the simulator our DF-Threads execution model [15, 20, 21] before spending time-consuming development on the reconfigurable hardware. We show here some evaluations that are possible within the MYDSE and COTSon framework once applied to the test case of the DF-Threads modelling. In such a test case, we aim to understand the impact of architectural and operating system choices on the execution time of our novel data-flow execution model [21].

Thanks to the MYDSE, we were also able to easily explore different architecture parameters, e.g., for the L2 cache size (from 2 to 1024 kB) and for the number of nodes/boards (ranging from one to four). Thus, in the case of deploying a soft processor and its peripherals on the FPGA, the designer can choose safely a well-optimized configuration for, e.g., the L2 cache size.

Moreover, we choose different operating system (OS) distributions to analyze the overhead produced by the OS in a target architecture: four different Ubuntu Linux distributions have been used: Karmic (or Ubuntu 9.10–label “karmic64”), Maverick (or Ubuntu 10.10–label “tfxv4”),

Trusty (or Ubuntu 14.04–label “trusty-axmv3”), and Xenial (or Ubuntu 16.04–label “xenv0”). The different architecture configurations used in the experimental campaign are summarized in Table 4.

The simulation framework permits exploring the execution of our benchmark easily, while we vary, e.g., the number of nodes (1, 2, 4), the OS. The input size of the shown example is fixed (matrix size = 512 elements).

As can be seen in Figure 11, there is a large variation of the kernel cycles between “xenv0” (Linux Ubuntu 16.04) and the other three Linux distributions. This indicated us to focus attention on the precise configuration of many daemons that run in the background and that may affect the activity of the system. While doing tests directly on the FPGA, it would not have been easy to understand that most of the time taken by the execution was actually absorbed by the OS activity: a designer could have taken it for granted or he/she could have not even had the possibility of changing the OS distribution for testing the differences, since the whole FPGA workflow is typically oriented to a fixed decision for the OS (e.g., Xilinx Petalinux). The situation is even worse for cache parameters or for the number of cores, since the designer might be forced to choose a specific configuration.

However, the important information for us was to confirm the scaling of the DF-Threads model, while we increase the number of nodes/boards. We can observe that the number of cycles is decreasing almost linearly—except the case of “xenv0,” which is decreasing sublinearly—when we use two and four nodes compared to the case of a single board/node (Figure 12). Moreover, we were able to understand the size of the cache that we should use in the physical system in order to properly accommodate the working set of our applications.

We further explored the reasons why we can obtain a good scaling in the execution time with more nodes by analyzing the behaviour of L2 cache miss rate (Figures 13 and 14).

Again, this type of measurement was conveniently done in the simulator, while it is more difficult to perform on the Xilinx Vivado HLS model, especially when it comes to design a soft processor and choose the best configuration (e.g., size of L2 cache, and OS). In particular, we varied again the number of nodes (1, 2, and 4), the OS distribution as before, and the cache size for L2 with larger values (64 KiB, 256 KiB, and 1024 KiB, to allow a wider range of exploration of the L2 cache). As we can see from Figure 14, the L2 cache miss rate is decreasing for all OS distributions while we vary the number of nodes, thus confirming that this is one of the main factors of the improvement of the execution time. Moreover, we can analyze which OS distribution leads to best performance. For example, the “xenv0” produces a huge amount of kernel activity during the computation (Figure 11). However, the combined effect of the kernel activity (Figure 11) and the average data latency (Figure 15)—considering L1, L2, and L3 caches—may affect the total execution time (Figure 12) quite heavily. Thanks to this preliminary DSE, we found that the OS distribution with

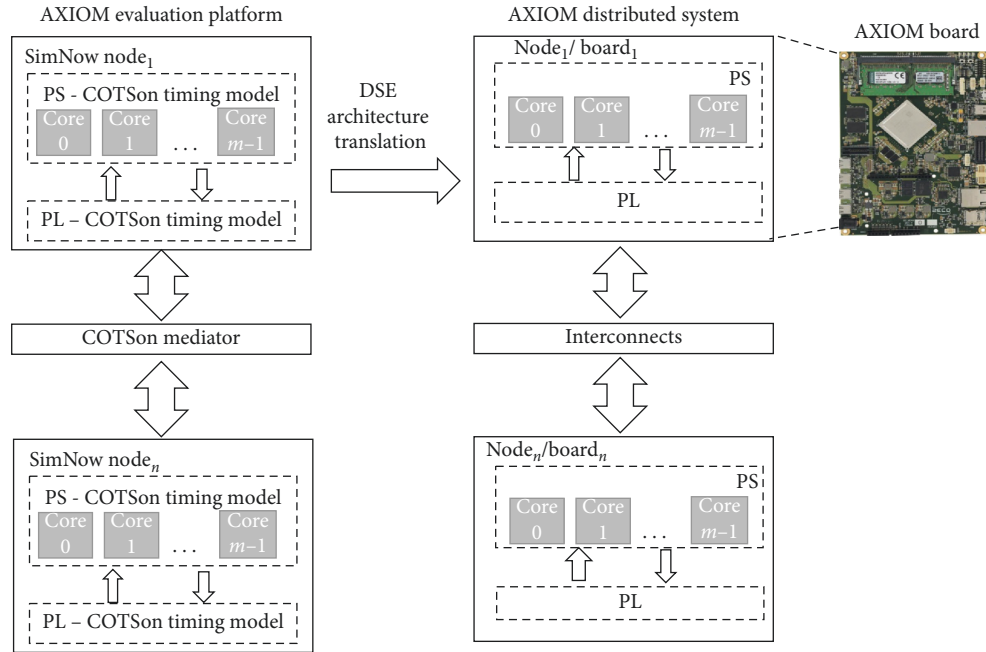


FIGURE 10: From COTSon Distributed System definition to AXIOM Distributed System by using the DSE tools. The processing system (PS), the programmable logic (PL), and the interconnects of the AXIOM board are simulated and evaluated into the COTSon framework with the definition of the respective timing models.

TABLE 4: COTSon architectural parameter.

Parameter	Description
SoC	1 core connected by shared-bus, IO-bus, MC, high-speed transceivers
Core	3 GHz, in-order super-scalar
Branch predictor	Two levels (history length = 14 bits, pattern history Table = 16 KiB, 8-cycle miss-prediction penalty)
L1 cache	Private I-cache 32 KiB, private D-cache 32 KiB, 2 ways, 3-cycle latency
L2 cache	Private 2, 8, 32, 64, 256, 1024 KiB, 4 ways, 5-cycle latency
L3 cache	Shared 4 MiB, 4 ways, 20-cycle latency
Coherence protocol	MOESI
Main memory	1 GiB, 100-cycle latency
I-L1-TLB, d-L1-TLB	64 entries, direct access, 1-cycle latency
L2-TLB	512 entries, direct access, 1-cycle latency
Write/read queues	200 Bytes each, 1-cycle latency

the best trade-off between memory accesses and kernel utilization is the “trusty-axmv3.”

Figure 16 shows our evaluation setup of two AXIOM boards interconnected via USB-C cables, without the need of an external switch. By using synergistically our framework and Vivado toolchain, we synthesized the DF-thread execution model on programmable logic (PL). Table 5 reports resource utilization of the key components of the implemented design on PL in order to perform BMM benchmark across two AXIOM boards.

5.2. Validating the AXIOM Board against the COTSon Simulator. An important step in the design is to make sure

that the design in the physical board is matching the system that was modelled in the COTSon simulator. As an example, we show in Figure 17 the execution time in the case of the BMM and RADIX-SORT benchmarks, when running on the simulator and on the AXIOM board, while we vary the input data size. The timings are matching closely, thus confirming the validity of our approach. We scaled the inputs in such a way that the number of operations doubles from the left to right (input size). On the left (Figure 17), we have the BMM benchmark, where the input size represents the size of the square matrices, which are used in the multiplication. On the right (Figure 17), we have the Radix-Sort benchmark, where the input size represents the size of the list to be sorted.

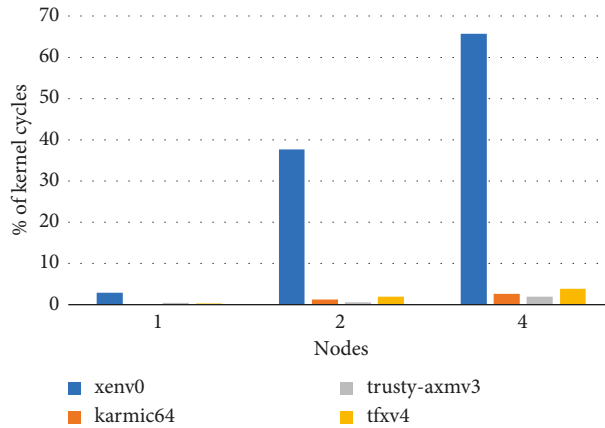


FIGURE 11: Percentage of kernel cycles (over total number of cycles) in the COTSon framework when using the matrix multiplication benchmark with 512 as the matrix size and 32 KiB as the cache size. We varied the number of nodes of the distributed system and the Linux distribution (“xenv0” = Ubuntu 16.04, “karmic64” = Ubuntu 9.10, “trusty-axmv3” = Ubuntu 14.04, and “tfxv4” = Ubuntu 10.10). This DSE test permitted to detect a much larger kernel activity of the “xenv0” distribution compared to the other three Linux distributions in both single-node and multiple-node configurations.

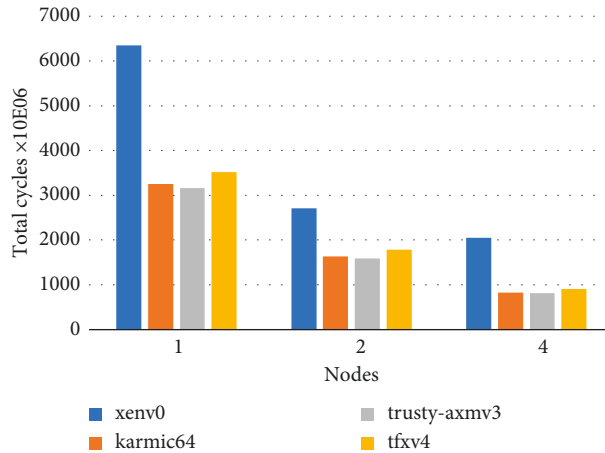


FIGURE 12: Total number of cycles in the COTSon framework when using the matrix multiplication benchmark with 512 as the matrix size and 32 KiB as the cache size. We varied the number of nodes of the distributed system and the Linux distribution (“xenv0” = Ubuntu 16.04, “karmic64” = Ubuntu 9.10, “trusty-axmv3” = Ubuntu 14.04, and “tfxv4” = Ubuntu 10.10). The DSE allows us to determine that the four Linux distributions permit to obtain a good scalability when we increase the number of nodes. However, the “xenv0” confirms the worst performance in terms of executed cycles due to the huge number of kernel cycles shown in Figure 11.

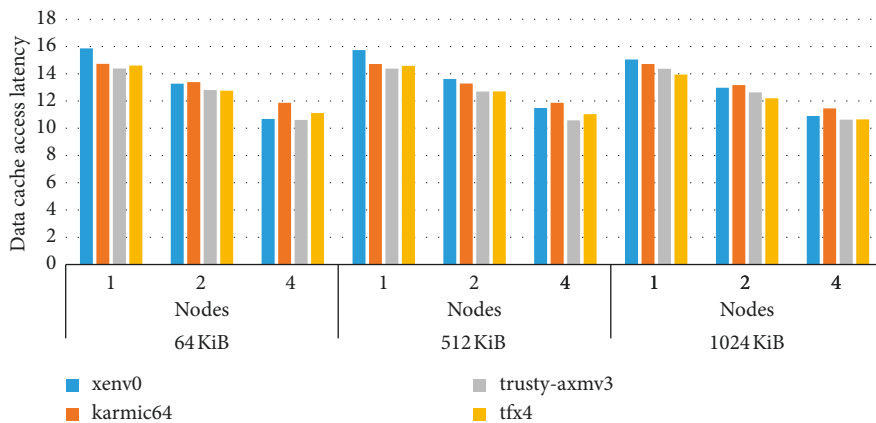


FIGURE 13: Evaluation of the data access latency in the COTSon framework when using the matrix multiplication benchmark by varying cache size, number of nodes of the distributed system, and different Linux distribution (“xenv0” = Ubuntu 16.04, “karmic64” = Ubuntu 9.10, “trusty-axmv3” = Ubuntu 14.04, and “tfxv4” = Ubuntu 10.10). The DSE shows that the data-cache access latency is almost similar in each Linux distribution, but it is lowering when we increase the number of nodes. Thus, multiple-node configuration can be more convenient in the DF-Threads execution model.

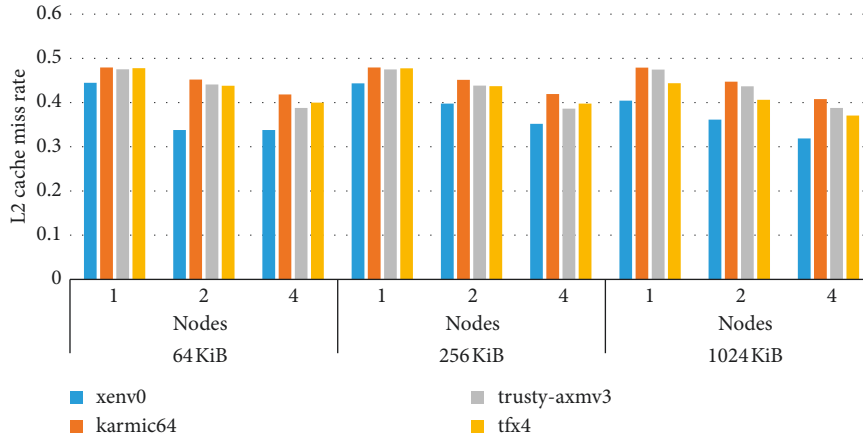


FIGURE 14: Evaluation of the L2-cache miss rate in the COTSon framework when using the matrix multiplication benchmark by varying cache sizes, number of nodes of the distributed system, and different Linux distribution (“xenv0” = Ubuntu 16.04, “karmic64” = Ubuntu 9.10, “trusty-axmv3” = Ubuntu 14.04, and “tfxv4” = Ubuntu 10.10). The Linux distribution “xenv0” shows the lowest L2 miss rate compared to the other Linux distributions for all the presented configurations.

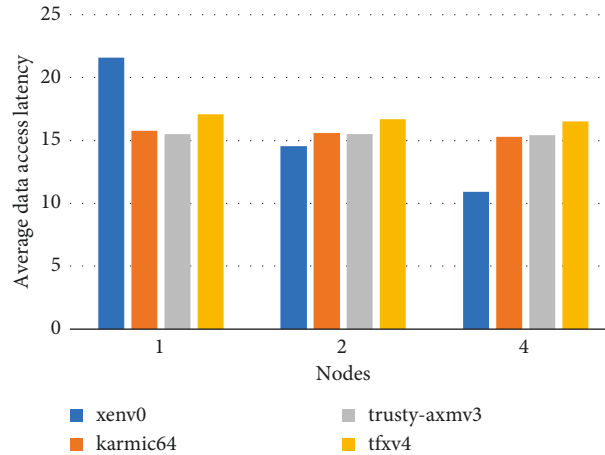


FIGURE 15: Average data latency in the COTSon framework when using the matrix multiplication benchmark with 512 as the matrix size and 32 KiB as the cache size. We varied the number of nodes of the distributed system and the Linux distribution (“xenv0” = Ubuntu 16.04, “karmic64” = Ubuntu 9.10, “trusty-axmv3” = Ubuntu 14.04, and “tfxv4” = Ubuntu 10.10). The data access latency of “xenv0” is improved when we have more nodes. This improvement has less impact on total cycles (Figure 12) than the impact of kernel activity (Figure 11).

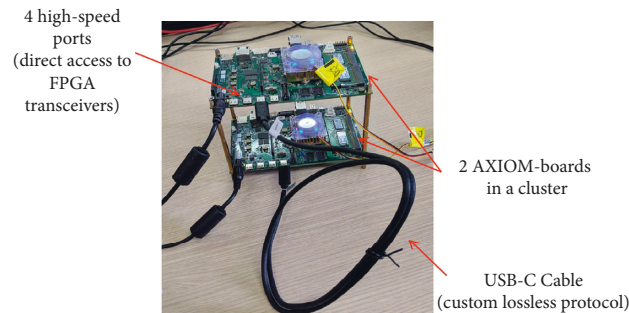


FIGURE 16: Two AXIOM boards interconnected up to 18 Gbps via inexpensive USB-C cables. The AXIOM board is based on a Xilinx Zynq Ultrascale + ZU9EG platform, four high-speed ports (up to 18 Gbps), an Arduino socket, and DDR4 extensible up to 32 GiB. As can be seen from the picture, we do not need any external switch but just two simple USB-C cables to connect the two systems.

TABLE 5: Resource utilization of the key components of the implemented programmable logic on ZU9EG FPGA (AXIOM board).

Component	LUT (%)	LUTRAM (%)	FF (%)	BRAM (%)	GT (%)
DF-Threads	10.03	1.8	6.01	5.43	—
NIC [4]	41.36	8.96	19.29	15.19	50

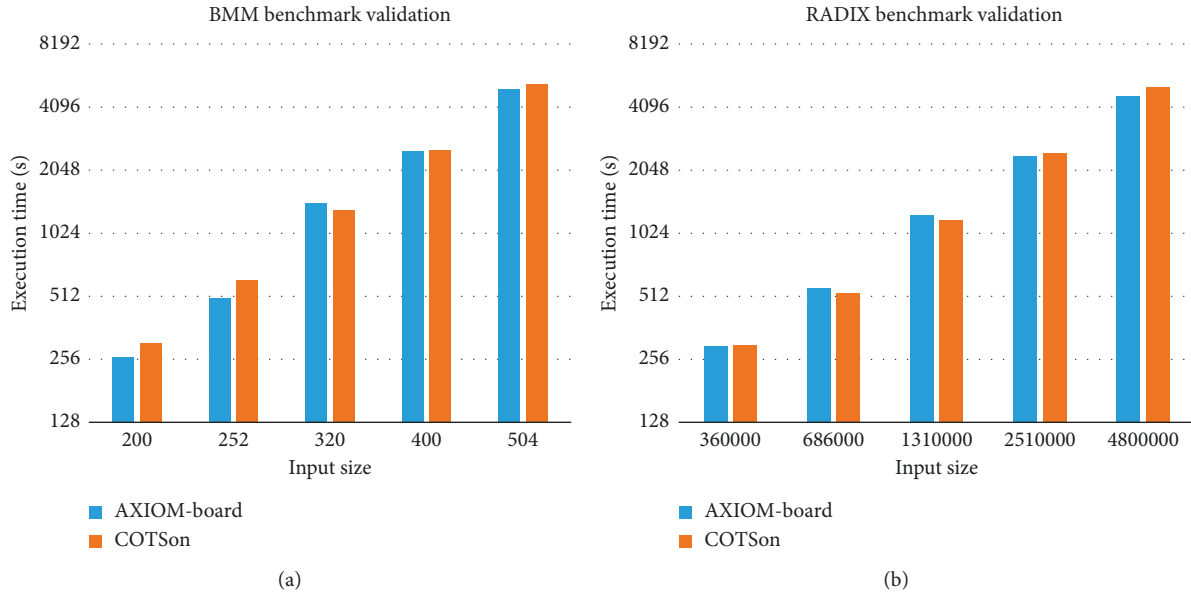


FIGURE 17: Validation of the execution time of the simulator against the AXIOM board. We used the blocked matrix multiplication (BMM) and Radix-Sort benchmarks with different sizes (weak scaling). The results on the actual board match closely the simulations.

6. Conclusions

In this article, we presented our workflow in developing an architecture that could be controlled by the designer in order to match the desired key performance metrics. We found that it is very convenient to use synergistically the Xilinx HLS tools and the COTSon + MYDSE framework in order to select a candidate architecture, instead of developing everything just with the HLS tools.

We illustrated the main features of the COTSon simulator and the “MYDSE” toolset, and we motivated their purpose in our simulation methodology. Thanks to the “functional-directed” approach of the COTSon simulator, we can define the architecture of any architectural components (i.e., a cache) for an early DSE and migrate to HLS only the selected architecture. Our DSE toolset facilitates the modelling of architectural components in the earlier stages of the design.

We have modified the classical HLS tool flow, by inserting a modelling phase with an appropriate simulation framework, which can facilitate the architecture definition and reduce significantly the developing time.

We described the simple example of defining a two-way set-associative cache through the timing model of COTSon. Afterwards, we illustrated the code migration from COTSon to Xilinx HLS tool, showing that the timing description made in the COTSon simulator is conveniently close to the final HLS description of our architecture. However, synthesizing of the HLS description of the cache design in Vivado HLS takes about four hours on a powerful workstation, while we were able to simulate it in COTSon in a few seconds.

By using the workflow presented in this article, we were able to successfully prototype a preliminary design of our data-flow programming model (called the DF-Threads) for a

reconfigurable hardware platform leading to the AXIOM software/hardware platform, a real system that includes the AXIOM board and a full software stack of more than one million lines of codes made available as open source (<https://git.axiom-project.eu/>).

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was partly funded by the European Commission through projects AXIOM H2020 (id. 645496), TERAFLUX (id. 249013), and HiPEAC (id. 779656).

References





- [1] S. Mittal and J. S. Vetter, “A survey of CPU-GPU heterogeneous computing techniques,” *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–35, 2015.
- [2] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini, “An integrated open framework for heterogeneous MPSoC design space exploration,” in *Proceedings of the Design Automation and Test in Europe Conference*, pp. 1145–1150, Munich, Germany, March 2006.
- [3] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, “Heterogeneous chip multiprocessors,” *Computer*, vol. 38, no. 11, pp. 32–38, 2005.

- [4] D. Theodoropoulos, S. Mazumdar, E. Ayguade et al., “The AXIOM platform for next-generation cyber physical systems,” *Microprocessors and Microsystems*, vol. 52, pp. 540–555, 2017.
- [5] R. Dimond, S. Racaniere, and O. Pell, “Accelerating large-scale HPC applications using FPGAs,” in *Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic*, pp. 191–192, Tuebingen, Germany, July 2011.
- [6] A. Portero, Z. Yu, and R. Giorgi, “TERAFLUX: exploiting tera-device computing challenges,” *Procedia Computer Science*, vol. 7, pp. 146–147, 2011.
- [7] R. Giorgi, R. M. Badia, F. Bodin et al., “TERAFLUX: harnessing dataflow in next generation teradevices,” *Microprocessors and Microsystems*, vol. 38, no. 8, pp. 976–990, 2014.
- [8] S. Wong, A. Brandon, F. Anjam et al., “Early results from ERA—embedded reconfigurable architectures,” in *Proceedings of the 2011 9th IEEE International Conference on Industrial Informatics*, pp. 816–822, Lisbon, Portugal, July 2011.
- [9] S. Wong, L. Carro, M. Rutzig et al., “ERA—embedded reconfigurable architectures,” in *Reconfigurable Computing*, pp. 239–259, Springer, Berlin, Germany, 2011.
- [10] R. Giorgi, “AXIOM: A 64-bit reconfigurable hardware/software platform for scalable embedded computing,” in *Proceedings of the 2017 6th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–4, Bar, Montenegro, June 2017.
- [11] R. Giorgi, M. Procaccini, and F. Khalili, “AXIOM: a scalable, efficient and reconfigurable embedded platform,” in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Florence, Italy, September 2019.
- [12] D. Theodoropoulos, D. Pnevmatikatos, C. Alvarez et al., “The AXIOM project (agile, extensible, fast I/O module),” in *Proceedings of the 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 262–269, Samos, Greece, July 2015.
- [13] R. Giorgi, F. Khalili, and M. Procaccini, “Energy efficiency exploration on the ZYNQ ultrascale+,” in *Proceedings of the 30th International Conference on Microelectronics (ICM)*, Sousse, Tunisia, December 2018.
- [14] SARC, <http://www.sarc-ip.org>.
- [15] R. Giorgi, Z. Popovic, and N. Puzovic, “Implementing fine/medium grained TLP support in a many-core architecture,” in *Proceedings of the International Workshop on Embedded Computer Systems*, pp. 78–87, Ancona, Italy, March 2009.
- [16] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, “COTSon: infrastructure for full system simulation,” *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 52–61, 2009.
- [17] R. Giorgi, M. Procaccini, and F. Khalili, “Analyzing the impact of operating system activity of different linux distributions in a distributed environment,” in *Proceedings of the 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pp. 422–429, Pavia, Italy, February 2019.
- [18] Xilinx, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf.
- [19] R. Giorgi, M. Procaccini, and F. Khalili, “A design space exploration tool set for future 1k-core high-performance computers,” in *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools on-RAPIDO’19*, Valencia, Spain, January 2019.
- [20] R. Giorgi and P. Faraboschi, “An introduction to DF-Threads and their execution model,” in *Proceedings of the 2014 International Symposium on Computer Architecture and High Performance Computing Workshop*, pp. 60–65, Florianópolis, Brazil, October 2014.
- [21] R. Giorgi, “Exploring dataflow-based thread level parallelism in cyber-physical systems,” in *Proceedings of the ACM International Conference on Computing Frontiers—CF’16*, pp. 295–300, Como, Italy, May 2016.
- [22] R. Giorgi, “Scalable embedded computing through reconfigurable hardware: comparing DF-Threads, cilk, openmpi and jump,” *Microprocessors and Microsystems*, vol. 63, pp. 66–74, 2018.
- [23] J. Chen, M. Annavaram, and M. Dubois, “SlackSim,” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 20–29, 2009.
- [24] T. Austin, E. Larson, and D. Ernst, “SimpleScalar: an infrastructure for computer system modeling,” *Computer*, vol. 35, no. 2, pp. 59–67, 2002.
- [25] M. M. K. Martin, D. J. Sorin, B. M. Beckmann et al., “Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.
- [26] H. Zeng, M. Yourst, K. Ghose, and D. Ponomarev, “MPTLsim: a simulator for X86 multicore processors,” in *Proceedings of the 46th ACM/IEEE Design Automation Conference*, pp. 226–231, San Francisco, CA, USA, July 2009.
- [27] A. Canis, J. Choi, M. Aldham et al., “LegUp: high-level synthesis for FPGA-based processor/accelerator systems,” in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 33–36, Monterey, CA, USA, February 2011.
- [28] C. Pilato and F. Ferrandi, “Bambu: a modular framework for the high level synthesis of memory-intensive applications,” in *Proceedings of the 2013 23rd International Conference on Field Programmable Logic and Applications*, pp. 1–4, Porto, Portugal, September 2013.
- [29] P. Coussy, C. Chavet, P. Bomel et al., “GAUT: a high-level synthesis tool for DSP applications,” in *High-Level Synthesis*, pp. 147–169, Springer, Berlin, Germany, 2008.
- [30] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis, “DWARV: delftworkbench automated reconfigurable VHDL generator,” in *Proceedings of the 2007 International Conference on Field Programmable Logic and Applications*, pp. 697–701, Amsterdam, The Netherlands, August 2007.
- [31] Cadence, https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.
- [32] Intel, <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls/pdf>.
- [33] Xilinx, <https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html>.
- [34] Xilinx, <https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html>.
- [35] J. Choi, S. Brown, and J. Anderson, “From software threads to parallel hardware in high-level synthesis for FPGAs,” in *Proceedings of the 2013 International Conference on Field-Programmable Technology (FPT)*, pp. 270–277, Kyoto, Japan, December 2013.
- [36] C. Lattner and V. Adve, “LLVM: a compilation framework for lifelong program analysis & transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, p. 75, Palo Alto, CA, USA, March 2004.
- [37] ACE CoSy, <http://www.ace.nl>.

- [38] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, "A DSL compiler for accelerating image processing pipelines on FPGAs," in *Proceedings of the 2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 327–338, Haifa, Israel, March 2016.
- [39] A. Lotfi and R. K. Gupta, "ReHLS: resource-aware program transformation workflow for high-level synthesis," in *Proceedings of the 2017 IEEE International Conference on Computer Design (ICCD)*, pp. 533–536, Orlando, FL, USA, November 2017.
- [40] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio, "A unified backend for targeting FPGAs from DSLs," in *Proceedings of the 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 1–8, Cornell Tech, NY, USA, July 2018.
- [41] J. E. Miller, H. Kasture, G. Kurian et al., "Graphite: a distributed parallel simulator for multicores," in *Proceedings of the Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–12, San Antonio, TX, USA, February 2010.
- [42] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: towards real-time hypervisor scheduling in Xen," in *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pp. 39–48, Taipei, Taiwan, October 2011.
- [43] A. Portero, A. Scionti, A. Yu et al., "Simulating the future kilo-x86-64 core processors and their infrastructure," in *Proceedings of the 45th Annual Simulation Symposium*, pp. 1–9, Orlando, FL, USA, March 2012.
- [44] C.-K. Luk, R. Cohn, R. Muth et al., "Pin," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.
- [45] R. Giorgi, "Exploring future many-core architectures: the TERAFLUX evaluation framework," in *Advances in Computers*, vol. 104, pp. 33–72, Elsevier, Amsterdam, Netherlands, 2017.
- [46] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*, pp. 84–97, San Diego, CA, USA, June 2003.
- [47] R. Ierusalimschy, W. Celes, and L. H. de Figueiredo, "The evolution of lua," 2005.
- [48] S. Windh, X. Ma, R. J. Halstead et al., "High-level language tools for reconfigurable computing," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 390–408, 2015.
- [49] D. D. Gajski, N. D. Dutt, A. C. H. Wu, and S. Y. L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Springer Science & Business Media, Berlin, Germany, 2012.
- [50] R. Giorgi, N. Bettin, P. Gai, X. Martorell, and A. Rizzo, "AXIOM: a flexible platform for the smart home," in *Components and Services For IoT Platforms*, pp. 57–74, Springer, Berlin, Germany, 2017.

Research Article

An FPGA-Based Hardware Accelerator for CNNs Using On-Chip Memories Only: Design and Benchmarking with Intel Movidius Neural Compute Stick

Gianmarco Dinelli ¹, Gabriele Meoni ¹, Emilio Rapuano,¹ Gionata Benelli ²,
and Luca Fanucci ¹

¹Department of Information Engineering, University of Pisa, Pisa 56122, Italy

²IngeniArs, Pisa 56121, Italy

Correspondence should be addressed to Gianmarco Dinelli; gianmarco.dinelli@ing.unipi.it

Received 2 May 2019; Revised 3 September 2019; Accepted 3 October 2019; Published 22 October 2019

Academic Editor: Martin Margala

Copyright © 2019 Gianmarco Dinelli et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

During the last years, convolutional neural networks have been used for different applications, thanks to their potentiality to carry out tasks by using a reduced number of parameters when compared with other deep learning approaches. However, power consumption and memory footprint constraints, typical of on the edge and portable applications, usually collide with accuracy and latency requirements. For such reasons, commercial hardware accelerators have become popular, thanks to their architecture designed for the inference of general convolutional neural network models. Nevertheless, field-programmable gate arrays represent an interesting perspective since they offer the possibility to implement a hardware architecture tailored to a specific convolutional neural network model, with promising results in terms of latency and power consumption. In this article, we propose a full on-chip field-programmable gate array hardware accelerator for a separable convolutional neural network, which was designed for a keyword spotting application. We started from the model implemented in a previous work for the Intel Movidius Neural Compute Stick. For our goals, we appropriately quantized such a model through a bit-true simulation, and we realized a dedicated architecture exclusively using on-chip memories. A benchmark comparing the results on different field-programmable gate array families by Xilinx and Intel with the implementation on the Neural Compute Stick was realized. The analysis shows that better inference time and energy per inference results can be obtained with comparable accuracy at expenses of a higher design effort and development time through the FPGA solution.

1. Introduction

During the last years, convolutional neural networks (CNNs) found application in many different fields like object detection [1, 2], object recognition [3, 4], and KeyWord Spotting (KWS) [5, 6]. Although they proved excellent results on cloud, their applicability for portable systems is challenging because of the additional constraints in terms of memory footprint and power consumption, which generally conflict with latency and accuracy requirements. In particular, in general purpose solutions based on the use of a microcontroller, the limited available memory limits the complexity of the network, with possible impact on the accuracy of the system [7]. In the same way, microcontroller-

based systems feature the worst trade-off between power consumption and timing performances [8].

For this reason, commercial hardware accelerators for CNNs such as Neural Compute Stick (NCS) [9], Neural Compute Stick 2 (NCS2) [9], and Google Coral [10] were produced. Such products feature optimized hardware architectures that allow to realize inferences of CNN models with low latency and reduced power consumption. Standard communication protocols, such as Universal Serial Bus (USB) 3.0., are generally exploited for communication purposes.

Nevertheless, since they were designed for the implementation of generic CNNs, their architectures are extremely flexible at the expense of the optimization of the single model.

For such a reason, hardware accelerators customized for a specific application might offer an interesting alternative for accelerating CNNs. In particular, field-programmable gate arrays (FPGAs) represent an interesting trade-off between cost, flexibility, and performances [11], especially for applications whose architectures have been changing too rapidly to rely on application-specific integrated circuits (ASICs) and whose production volumes might be not sufficient. FPGAs offer high flexibility at the same time, which permits the implementation of different models with a high degree of parallelism [8] and the possibility of customizing the architecture for a specific application.

The aim of this paper is to investigate the use of custom FPGA-based hardware accelerators to realize a CNN-based KWS system, analysing their performances in terms of power consumption, number of hardware resources, accuracy, and timing. A KWS system represents an example of application whose porting on the edge requires much effort, owing to the hard design trade-offs.

The study involves the use of different FPGA families by Xilinx and Intel, analysing design portability on devices with different sizes and performances. This allowed to realize a benchmark that compares the obtained results with the ones presented in our previous work for the full-SCNN (separable convolutional neural network) model [12], which implements the same architecture exploiting a NCS (version 1, mounting Myriad 2 Vision Processing Unit (VPU)).

To realize the architecture implemented on-board FPGA, a bit-true simulation was performed to appropriately quantize the model, reducing the number of resources used, saving power, and increasing throughput when compared with a floating-point approach.

The remainder of the paper is structured as follows: the *Keras* model used to describe the KWS system is presented in Section 2. Section 3 presents the approach used to quantize and compress the model to optimize its implementation on-board FPGAs. In Section 4, the results of the quantization analysis are provided and discussed. The preferred FPGA-based accelerator architecture is then described in Section 5, focusing on the analysis of design trade-offs. Results of the implementation on the different FPGA families are presented in Section 6. In Section 7, results in terms of maximum achievable clock frequency, hardware resources, and power consumption are presented and compared with the NCS solution. In Section 8, the usability of FPGA devices to accelerate the inference of CNNs is discussed with respect to the presented solution and similar applications. Finally, in Section 9, conclusions are given.

2. Architecture of the KWS System

KWS systems are a common component in speech-enabled devices: they continuously listen to the surrounding environment with the task to recognize a small set of simple commands in order to activate or deactivate specific functionalities. Commercial examples of KWS systems include “OK Google” and “Hey Siri.” The proposed KWS system is designed to operate inside a domotic installation for improving the quality of life of people with disabilities. In

particular, it is able to recognize 10 different commands: “yes,” “no,” “up,” “down,” “left,” “right,” “on,” “off,” “stop,” and “go.” Moreover, it identifies two additional classes: “silence,” when no word is pronounced, and “unknown,” when the pronounced word does not belong to any class.

The KWS system was pretrained in the Python framework called *Keras* [13], using *Google Speech Command dataset*.

The proposed architecture is based on the SCNN described in [12], whose architecture is shown in Figure 1.

The input of the network is a 63×13 mel frequency spectral coefficient (MFSC) matrix [14]. The bin (n, k) of the matrix contains information over the spectral content at frequency f , as shown in equation (1):

$$f = k \cdot \frac{f_{\text{sample}}}{N + 1}, \quad \text{for } k = 1, \dots, K, \quad (1)$$

where $f_{\text{sample}} = 16$ kHz is the sample rate and $N = 512$ (32 ms) is the number of bins used to calculate the fast fourier transform (FFT), measured at the instant n/f_{sample} , with $n \in [0, N - 1]$. Every N -sample window is weighted through a Hann window and overlapped with the previous $N/2$ samples for the calculation of the FFT.

The input layer provides the 63×13 MFSC input matrix. Then, three separable convolutional (SC) layers follow, and their generic structure is shown in Figure 2.

SC layers improve standard convolutional layers by reducing the number of parameters used to process the inputs [12]. For this reason, SCNNs are particularly interesting for the realization of FPGA-based hardware accelerators because they reduce memory and computation requirements in comparison with the classic CNN approach.

A standard convolutional layer contains $c_{\text{out}}(w_f x h_f)$ filters that are convolved over.

$c_{\text{in}}(w_{\text{cin}} x h_{\text{cin}})$ input channels, producing $c_{\text{out}}(w_{\text{cin}} - w_f + 1) x (h_{\text{cin}} - h_f + 1)$ output channels. On the contrary, a separable convolution is realized through two distinct convolutions performed by means of filters, whose dimensions are, respectively, $(f_w x 1)$ and $(1 x f_h)$. Figure 3 better illustrates the difference between these two approaches.

Considering the structure of the MFCS input matrix, each SC layer performs two separated convolutions, realizing a “time” convolution followed by a “frequency” convolution.

A batch normalization (BN) layer, which has the role to accelerate deep network training by reducing internal covariance shift [15], follows the frequency convolution. Finally, the rectified linear unit (ReLU) is the activation function of each SC layer. ReLU is defined in equation (2) as

$$f_{\text{ReLU}}(x) = \begin{cases} 0, & \text{when } x \leq 0, \\ k \cdot x, & \text{when } x > 0 \text{ with } k \in \mathbb{R}. \end{cases} \quad (2)$$

A classic convolutional layer follows the three SC layers.

Table 1 summarizes the dimension of time/frequency filters, number of input channels (C_{in}), output channels (C_{out}), and input/output matrix dimensions for each convolutional layer of the network. Time_0 and freq_0 are,

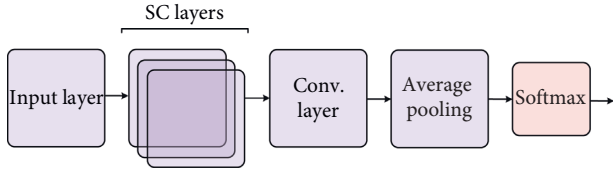


FIGURE 1: SCNN architecture.

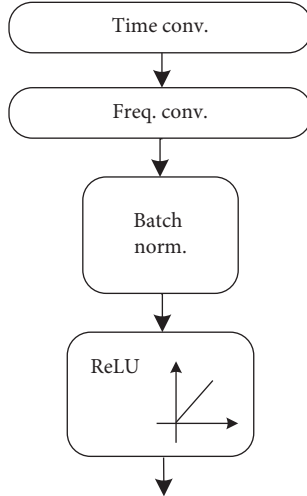


FIGURE 2: SC hidden layer architecture.

respectively, the temporal and frequency convolutional layer of the hidden layer 0, and similarly time₁/freq₁ for the hidden layer 1 and time₂/freq₂ for the hidden layer 2. Final_conv refers to the last convolutional layer of the network.

The average pooling layer computes the average value of each output channel of the final_conv layer, condensing them in 12 values, one for each class of the KWS system. Finally, a Softmax (or normalized exponential function) layer activation function follows. It takes a vector Z_j as input and produces an output vector in which each element $f_{\text{softmax}}(Z_j)$ is normalized in the interval $[0, 1]$ and can be interpreted as the probability that input belongs to the class j . The standard Softmax function is described by equation (3):

$$f_{\text{softmax}}(Z_j) = \frac{e^{z_j}}{\sum_{i=1}^K e^{z_i}}, \quad \text{for } j = 1, \dots, K. \quad (3)$$

In this network, the Softmax input vector is composed of 12 elements, one for each of the class of the KWS system.

The proposed SCNN model was implemented on the Intel Movidius NCS, showing an accuracy of 87.77%. The number of parameters necessary for its implementation is 15000, including bias, weights, and batch normalization parameters.

3. Keras Model Optimization toward the Hardware Implementation

In the next sections, methods to map the Keras–Python model of the KWS system on an FPGA are analysed. In

fact, this model is implemented in a high-level language and its parameters are based on the floating-point representation.

The main issue about the implementation of a CNN-based model on an FPGA regards the limitation in terms of available hardware resources (combinatorial elements, sequential elements, Digital Signal Processors (DSPs), ram blocks, etc.) of such devices [11, 16, 17]. CNN algorithms are based on Multiply-and-ACcumulate (MAC) operations that require a large amount of combinatorial logic elements or DSPs. Furthermore, CNNs are characterised by a great number of parameters that shall be stored into off-chip memories if exceeding the available on-chip memory. The use of off-chip memory could be inevitable, complicating the design and increasing the inference time. For these reasons, the architecture of the hardware accelerator was carefully designed considering the trade-off between inference time and available resources.

3.1. Model Quantization. Before realizing the FPGA implementation, a quantization of the SCNN model was performed. In literature, there are many examples of quantization applied to CNNs [18–21]. The main advantage offered by a fixed-point representation is the possibility to shrink the model dimension and complexity with a negligible loss in accuracy [22]. In addition, fixed-point arithmetic requires simpler calculation than floating-point arithmetic, with advantages in terms of complexity and power consumption [23].

The quantization of the original floating-point model was performed through a bit-true simulation. The aim of the simulation is to determine the number of bits necessary to represent numbers in every internal node of the network by limiting the loss in accuracy.

The fixed-point representation of the model weights (or filter elements) w_q was calculated by using the approach described by the following equation:

$$\begin{cases} w_q = \text{round}\left(\frac{w}{\text{lsb}_w}\right) \cdot \text{lsb}_w, \\ \text{lsb}_w = \frac{|w|_{\max}}{2^{b_w-1}}, \end{cases} \quad (4)$$

where w is the floating-point representation of the weight and lsb_w is the value of the least significant bit (lsb). The latter is calculated by dividing $|w|_{\max}$, which represents the absolute value of the maximum weight over each layer, by 2^{b_w-1} , where b_w is the number of bits used to represent weights, as required by the 2's complement format. In particular, since the range of weights amplitude is roughly the same for every layer, the same value of $|w|_{\max}$ was used for each layer. Such choices are due to the necessity to reduce the conspicuous degrees of freedom in the simulation. Furthermore, in order to reduce the number of operations to implement in hardware, the effects of the BN are included in weight and bias values (BN simply consists in algebraic operations). In formulas, each weight $w(i)$ and each bias $b(i)$ belonging to a frequency convolutional layer

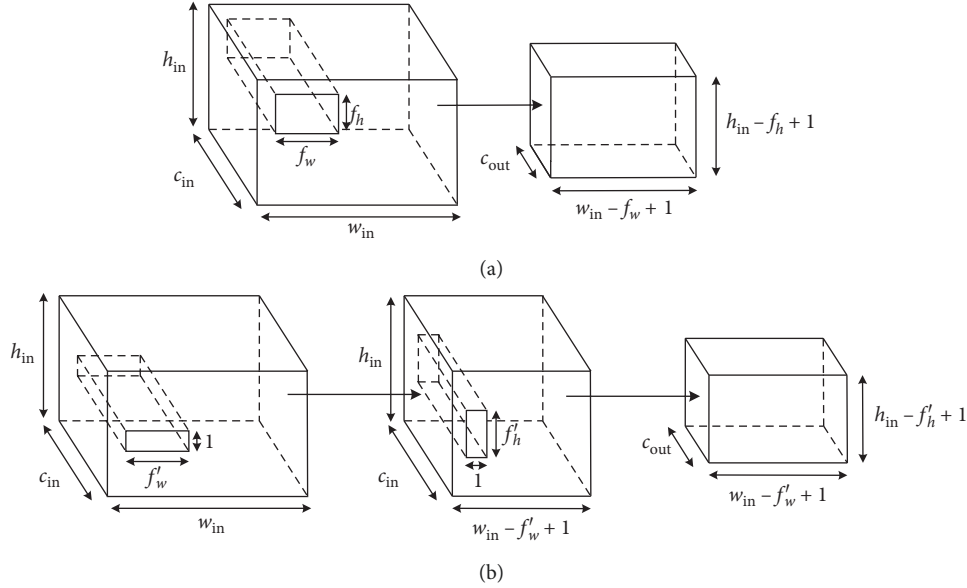


FIGURE 3: Convolutional layers: (a) classic CNN network and (b) SCNN network.

TABLE 1: Convolutional parameters for the network.

	Layer	Input matrix	Filter	C_{in}	C_{out}	Output matrix
Hidden layer 0	Time_0	63×13	5×1	1	1	59×13
	Freq_0	59×13	1×3	1	8	59×11
Hidden layer 1	Time_1	59×11	5×1	8	8	55×11
	Freq_1	55×11	1×3	8	16	55×9
Hidden layer 2	Time_2	55×9	11×1	16	16	45×9
	Freq_2	45×9	1×3	16	192	45×7
	Final_conv	45×7	1×1	192	12	45×7

or final_conv was modified as described by equations (5) and (6):

$$w(i)' = \frac{w(i) \cdot \gamma}{\sigma}, \quad (5)$$

$$b(i)' = \beta + \frac{(b(i) - \mu) \cdot \gamma}{\sigma}, \quad (6)$$

where γ and β are the scaling factors and bias of the BN, respectively, σ the standard deviation, and μ the average of the weights of a given input channel.

At the end of each layer, the acceptable number of truncated bits b_{tr_i} and a saturation (truncation of the most significant bits) of b_{sat_i} bits were also studied through the bit-true simulation to reduce the complexity of the hardware. In terms of formulas, truncation consists in changing the value of the lsb, as described by the following equation:

$$lsb_w' = lsb_w \cdot 2^{b_{tr_i}}. \quad (7)$$

Instead, saturating b_{sat_i} bit means discarding the b_{sat_i} most significant bits. Such operation does not affect lsb_w . For this aim, the worst case (greatest value in absolute meaning) of each layer output was considered so as to eliminate unused bits that were previously added for avoiding overflow of arithmetic operations. To sum up, the accuracy of the

model for different sets of $b_w, b_{tr1}, b_{sat1}, b_{tr2}, b_{sat2}, \dots, b_{tr7}, b_{sat7}$ was evaluated.

A possible optimization of the model consists in quantizing separately the weights of the last convolutional layer, by using $b_{w_{last}}$ bits. Indeed, the coefficients of final_conv may be divided by the divisor of the average pooling, saving hardware operations. This optimization significantly changes the range of weights for the last layer and a different quantization should be applied to it. For this reason, a second model to evaluate the overall accuracy takes into consideration different sets $(b_w, b_{w_{last}}, b_{tr1}, b_{sat1}, b_{tr2}, b_{sat2}, \dots, b_{tr7}, b_{sat7})$.

3.2. Pruning. Another technique to reduce the complexity of the hardware accelerator is *pruning*. It consists in dropping the least important connections of the network [24, 25] by identifying the weights or biases with a magnitude smaller than a given threshold. In this network, the biases of the temporal convolutional layers have magnitudes in the order of 10^{-9} – 10^{-7} . Considering their small values with respect to the other network parameters, they were pruned to reduce the model size. Indeed, it is possible to eliminate temporal bias terms without significantly affecting accuracy and reducing the number of sums to be computed.

4. Results of the Quantization Analysis

In this section, the results obtained from the quantization process are presented and discussed.

The SCNN model of this network has many degrees of freedom. For this reason, the first simulation step is finalised to identify a starting point for a more complex analysis, and it only focuses on the quantization of input layer words and weights.

Figure 4 shows simulation results, in terms of accuracy and mean square error (MSE) in relation to the floating-point model, when only the number of bits for input words representation is quantized. A number of 4 or 5 bits optimize accuracy and minimize the MSE. This first analysis gives intuitions about a possible optimization for the input layer: in particular, the number of bits of every input can be forced to be multiple of 4 bits, so that several inputs might be contained in buses such as the Advanced eXtensible Interface 4 (AXI4) bus [26], whose size is usually a multiple of 8 bits.

Figure 5 shows simulation results, in terms of accuracy and MSE, when only the number of bits for the representation of weights has been quantized. In this case, accuracy rapidly grows between 8 and 12 bits, reaching even higher values than the original ones in correspondence of 11 and 12 bits. Finally, accuracy saturates for 16 or more bits. This parameter is crucial because it influences the number of bits necessary to represent the result of MAC operations and, consequentially, the complexity of the entire network.

This first analysis was the starting point for a more detailed design exploration, involving the number of bits for the representation of the output of each layer.

Table 2 reports the best results obtained in terms of accuracy. Only the number of bits of SC layers and final_conv outputs are presented in the table, whereas data regarding temporal convolutional sublayers are omitted. The parameters listed in the table are as follows:

- (i) b_{in} : number of bits for the representation of input words.
- (ii) b_{filter} : number of bits for the representation of filters.
- (iii) bit_{out_0} : number of bits for the representation of the outputs of the first hidden layer.
- (iv) bit_{out_1} : number of bits for the representation of the outputs of the second hidden layer.
- (v) bit_{out_2} : number of bits for the representation of the outputs of the third hidden layer.
- (vi) bit_{out_fc} : number of bits for the representation of the outputs of the last convolutional layer.

Collected data show that it is possible to increase model accuracy through quantization. In fact, the best accuracy obtained for the floating-point model is 87.77%, whereas for the fixed-point representation, the highest accuracy is 90.23%.

The second part of the simulation considers a different quantization for the final_conv layer due to the inclusion of the average pooling effects, as explained in Section 3.1. The results of this simulation are summarized in Table 3.

These models show smaller hardware requirements than the single-quantization versions presented in Table 2. Sets of data are the same as those in Table 2, excepting for b_{last} that represents the number of bits for the representation of final_conv layer weights, whereas b_{filter} refers only to SC layer weights. This second model allows to shrink weights representation for all convolutional layers, significantly reducing the impact of MAC operations on hardware resource requirements. Furthermore, several quantized models have an accuracy score higher than the original one (87.77%).

The model chosen for the FPGA implementation considers both accuracy and the possibility to shrink parameter representations. Model number (7) from Table 3 was selected: it has a higher accuracy than the Keras-Python model (88.09 versus 87.77), and it minimizes the number of bits necessary for the representation of layer outputs and weights. Input layer results compatible with AXI4 because Input words are represented on 4 bits. The number of bits for the representation of temporal convolutional outputs of model (7) is 10 for time_0, 8 for time_1, and 10 for time_2.

5. FPGA Hardware Architecture

This section describes the architecture of the hardware accelerator that was implemented on different FPGA families. Thanks to the reduced number of parameters of the SCNN investigated in our previous work [12], it was possible to realize a full on-chip design with high advantages in terms of latency and energy per inference, avoiding accesses to off-chip memories [11, 21].

Figure 6 shows the block diagram of the accelerator. The number of bits of the words read from and written into the Input memory and RAMs is related to our preferred model, described in the previous section.

An input memory is used as an interface between the hardware accelerator and the system that records and elaborates the audio samples. The input memory stores 4-bit input data. The time/frequency layers and final_conv layer perform convolutional operations and store the results into a RAM memory, used as a buffer. Once the previous layer completes an entire convolution, the next one starts reading out its input matrix from the memory.

Each of the seven convolutional layers has its own MAC module to perform multiply-accumulate operations. Figure 7 shows the structure of the MAC module, designed to compute one element of the output matrix per clock cycle.

It reads n_{elem} elements from the RAM memory, where the value of n_{elem} is shown in the following equation:

$$n_{elem} = C_{in} \cdot f_{elem}, \quad (8)$$

where C_{in} is the number of input channels and f_{elem} is the number of elements composing a channel filter. The adder-tree structure is used for accumulation, and it was chosen to reduce the overall latency of the circuit. Considering this configuration of the MAC module, the total number of clock cycles needed to complete an entire convolution for each convolutional layer is N_{clk} , as shown in the following equation:

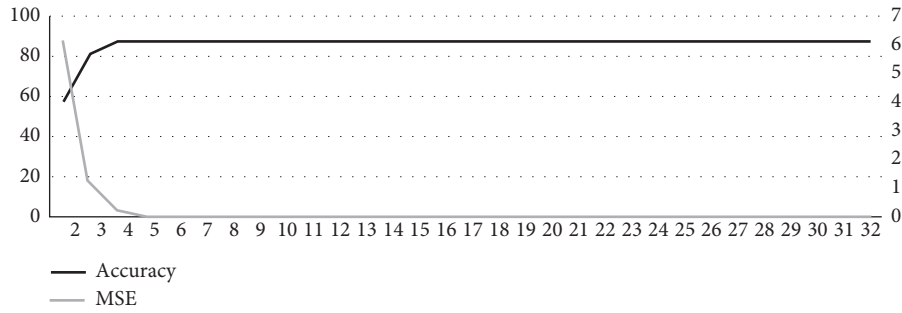


FIGURE 4: Accuracy and MSE to the change of the number of bits for input layer words.

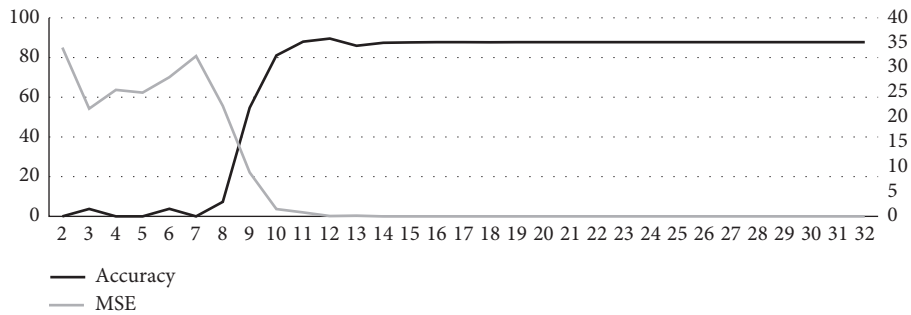


FIGURE 5: Accuracy and MSE to the change of the number of bits of filter elements.

TABLE 2: Results of the first quantization analysis.

b_in	b_filter	bit_out_0	bit_out_1	bit_out_2	bit_out_fc	Accuracy (%)
5	12	10	8	10	10	90.23
5	12	8	8	10	10	90.14
5	12	8	8	10	8	89.74
5	11	8	8	10	10	88.91
4	12	8	8	10	12	88.87
4	12	8	8	10	10	88.78
5	11	8	8	10	8	88.60
4	12	8	8	10	8	88.46
5	11	8	8	10	8	88.40
4	11	8	8	10	12	87.84
4	11	8	8	10	10	87.61

TABLE 3: Results of the second quantization analysis.

No.	b_in	b_filter	b_last	bit_out_0	bit_out_1	bit_out_2	bit_out_fc	Accuracy (%)
1	5	8	6	8	8	10	12	89.88
2	5	8	6	8	8	10	10	89.74
3	4	12	6	8	8	10	12	88.87
4	4	12	6	8	8	10	10	88.75
5	4	12	6	8	8	10	12	88.21
6	4	12	6	8	8	10	10	88.09
7	4	8	6	8	8	10	10	88.09
8	4	8	6	8	8	10	10	87.61
9	4	11	6	8	8	10	12	87.55
10	4	8	6	8	8	10	10	87.43
11	5	8	6	8	8	8	10	87.39
12	4	12	6	8	8	10	8	87.36
13	4	11	6	8	8	10	10	87.29
14	5	8	6	8	8	8	8	86.93

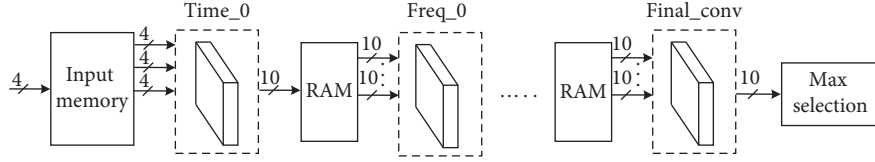


FIGURE 6: SCNN architecture for FPGA implementation.

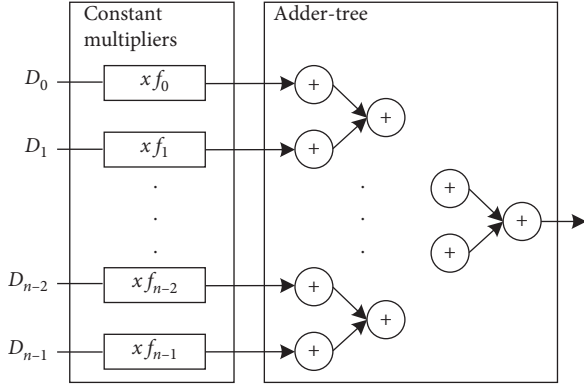


FIGURE 7: MAC module architecture.

$$N_{\text{clk}} = C_{\text{out}} \cdot W_{\text{out}} \cdot H_{\text{out}}, \quad (9)$$

where C_{out} is the number of output channels and W_{out} and H_{out} are the dimensions of the output matrix. Table 4 shows N_{clk} of each convolutional layer of the network considering the values of C_{out} , W_{out} , and H_{out} listed in Table 1. Finally, 819 clock cycles shall be added in order to store the 63×13 input matrix in the Input memory. A total of 90278 clock cycles are required to complete an inference.

A major parallelization of MAC operations would offer the opportunity to speed-up accelerator performances, reducing the inference time. On the other hand, it is not generally possible to perform an arbitrary number of operations per clock cycle because of the limited number of FPGA resources (combinatorial logic, DSPs, etc.). Furthermore, if the level of parallelism is too high, routing can become the bottleneck of the implementation.

It is possible to boost MAC module operations, increasing the number of output elements n computed per clock cycle. In particular, for $n > 1$, N_{clk} is reduced of a factor $1/n$, as described by the following equation:

$$N_{\text{clk}} = \frac{C_{\text{out}} \cdot W_{\text{out}} \cdot H_{\text{out}}}{n}. \quad (10)$$

Whilst this strategy leads to better timing optimization, it increases the design effort necessary to find the best combination that can fit on a specific FPGA device. Indeed, the appropriate value of n for each layer should be tuned depending on the size of the target FPGA, in order to guarantee design implementability. Furthermore, parallelizing each layer guarantees negligible advantages in terms of inference time when the number of operations necessary to carry out an entire convolution is strongly different for every layer. Considering the limitation of FPGA hardware

TABLE 4: N_{clk} values for the various layers.

Layer	N_{clk}
Input memory	819
Time_0	767
Freq_0	5192
Time_1	4840
Freq_1	7920
Time_2	6480
Freq_2	60480
Final_conv	3780
Total	90278

resources, it results appropriate to parallelize MAC operations only for the layers with the highest values of N_{clk} . In this specific case, freq_2 layer contributes to 60460 over 90278 total number of clock cycles due to the very high number of output channels (192). For this reason, the MAC module of the freq_2 layer was customized so that it calculates 4 values of the output matrix per clock cycle. According to equation (10), this allows to drastically reduce freq_2 N_{clk} from 60480 to 15120 and consequently the total inference time from 90278 to 44918 clock cycles, halving the inference time. If a similar parallelization was realized for the other convolutional layers of the network, it would increase hardware resources without a significant improvement of timing performances because of their limited effect on the overall inference time.

As previously specified, batch normalization operations were absorbed in the frequency convolutional layer of each SC layer. The average pooling layer was included in final_conv that provides 12 outputs, corresponding to the sum of all the elements belonging the output matrix of each output channel. Finally, Softmax layer can be omitted. Indeed, to provide a direct decision on the pronounced word, it is sufficient to select the maximum value among the twelve outputs of final_conv.

This architecture was chosen because its simplicity heightens the possibility to fit the hardware accelerator in a target FPGA, reducing design time and increasing design portability among devices with different sizes.

6. Hardware Implementation Results

This section describes the performances of the hardware accelerator on different FPGA families. The presented architecture was implemented on several Xilinx and Intel devices to analyse its design portability on FPGAs with different sizes and performances. Results are presented in terms of hardware resource occupation, maximum achievable clock frequency, inference time, and power consumption. Finally, an analysis of how MAC module

parallelization influences design portability on smaller FPGAs is provided.

The devices included in the analysis are as follows:

- (i) Zynq UltraScale+ (US+), xczu9eg-ffvb1156-2-e [27]
- (ii) Virtex UltraScale+, xcvu3p-ffvc1517-2-e [28]
- (iii) Virtex UltraScale (US), xcvu065-ffvc1517-2-e [29]
- (iv) Zynq-7000, xc7z045ffg900-1 [30]
- (v) Virtex-7, xc7vx330tffg1157-2 [31]
- (vi) Kintex-7 low voltage (lv), xc7k160tfg484-2L [31]
- (vii) Artix-7 low voltage (lv), xc7a200tfg484-2 [31]
- (viii) Arria 10 GX, 10AX027H3F35E2SG [32]
- (ix) Stratix V GS, 5SGSMD4E1H29C1 [33]
- (x) Stratix V GX, 5SEE9F45C2 [33]
- (xi) Stratix V E, 5SEE9H40C2 [33]
- (xii) Cyclone V, 5CEFA9U19C8 [34]

All the implementations were realized by using *Vivado design suite* for Xilinx devices and *Quartus Prime Software* for Intel devices.

Table 5 shows the hardware resources needed for the implementation of the accelerator on Xilinx FPGAs. Results are presented in terms of combinatorial elements, sequential elements, BRAMs, and LUTRAMs (LRAMs). The percentage of used resources out of the total is also indicated. Table 6 shows hardware resources needed for the implementation of the accelerator on Intel FPGAs. In this case, results are presented in terms of combinatorial elements, sequential elements, BRAMs, and DSPs.

All the implementations refer to the version of the accelerator in which the MAC module of the freq_2 layer was parallelized to compute 4 elements of its output matrix per clock cycle. The structure and the number of combinatorial/sequential elements and memory dimensions and typologies are specific for each device. Please refer to FPGA datasheets for more information about the architecture of Xilinx devices [27–31] and Intel devices [32–34].

Figures 8 and 9 show the maximum achievable clock frequency and the inference time for Xilinx and Intel FPGAs, respectively. The minimum inference time for each layer can be calculated taking into consideration MAC module optimizations for the freq_2 layer and N_{clk} values listed in Table 4. The best result is obtained for the Zynq UltraScale+ with a maximum clock frequency of 116.2 MHz and a corresponding inference time of less than 0.4 ms.

A power analysis was performed for both Xilinx and Intel FPGAs. To obtain a more accurate estimation of the power consumption for Xilinx devices, a post-implementation timing simulation was carried out by using *Questa® Advanced Simulator* to extract information about the switching activity of the internal nodes of the circuit. Since Intel devices do not support postlayout simulation, only a RTL-level estimation of the switching activity has been included in the power consumption analysis as suggested by Intel guidelines [35]. Results are shown in Table 7.

In general, Xilinx devices show a lower power consumption than Intel devices for both static and dynamic power. The only exception is the Arria 10, featuring a power consumption of 1 W and resulting the second best device after the Kintex-7 lv.

6.1. Design Portability. An analysis of the hardware accelerator portability has been carried out in order to investigate how the proposed design fits in smaller FPGAs. In particular, the freq_2 layer has been customized to compute 1, 2, 4, and 8 elements (n_{out}) of a given output channel per clock cycle. Results are presented in terms of hardware resources, maximum clock frequency, and inference time.

Two FPGAs with different sizes belonging to the same family were selected:

- (i) xc7z045ffg900-2 (xc7z045) and xc7z030fbg484-2 (xc7z030) for the Zynq-7000 family [30]
- (ii) xczu9eg-ffvb1156-2-e (xczu9eg) and xczu3eg-sfva625-2L-e (xczu3eg) for the Zynq UltraScale+ family [27]

Tables 8 and 9 show the results in terms of hardware resource occupation for the Zynq-7000 FPGAs and for the Zynq-US+ FPGAs, respectively.

The xc7z030 and the xczu3eg have a limited number of hardware resources and only the version of the accelerator with n_{out} equal to 1, 2, and 4 can be implemented in these devices; the version with n_{elem} equals to 8 fits only in the xc7z045 and in the xczu9eg. Owing to the limited number of LUTs available on-board xc7z030 and xczu3eg DSPs are included to perform MAC operations. In addition, xczu3eg implementations exploit all the available BRAMs on-board, and LRAMs have to be included. Versions of the hardware accelerator with a lower level of MAC parallelization have worst performance in terms of inference time but can fit in smaller devices because their requirements in terms of combinatorial elements are more relaxed. Unfortunately, the number of RAMs required does not change because intralayer RAM dimensions and the number of parameters of the network do not, and it can represent a bottleneck for the implementation of the not-customized version of the accelerator on smaller FPGAs.

Figures 10 and 11 show the performance in terms of clock frequency and inference time for Zynq-7000 and Zynq-US+ FPGAs, respectively. For the xc7z045 and the xczu9eg, the maximum achievable clock frequency does not show large variation increasing the level of MAC parallelism. For the xc7z030 and the xczu3eg, maximum achievable clock frequency tends to decrease because the limited size of these devices leads to a less optimized routing and consequently to worse timing performances. For this reason, inference times for xc7z030 with n_{mac} equals to 4 and n_{mac} equals to 2 are almost the same.

Similarly, when n_{out} is equal to 4, timing performance of the xc7z030 solution features an implementation loss of the 31% with respect to the solution on-board the xc7z045, and the xczu3eg solution features an implementation loss of 47% with respect to the one on-board the xczu9eg.

TABLE 5: Hardware accelerator implementation on Xilinx FPGAs.

FPGA family	Comb. elem.	Comb. elem. (%)	Seq. elem.	Seq. elem. (%)	BRAM	BRAM (%)	LRAM	LRAM (%)
Zynq US+	81345	30	860	<1	228	25	2560	2
Virtex US+	81367	21	864	<1	228	32	2560	1
Virtex US	81427	23	952	<1	228	18	2560	3
Zynq-7000	76283	35	632	<1	244	45	0	0
Virtex-7	76163	37	632	<1	244	33	0	0
Kintex-7 lv	81737	81	633	<1	244	75	0	0
Artix-7 lv	87406	86	1081	<1	228	70	0	0

TABLE 6: Hardware accelerator implementation on Intel FPGAs.

FPGA family	Comb. elem.	Comb. elem. (%)	Seq. elem.	Seq. elem. (%)	BRAM	BRAM (%)	DSP	DSP (%)
Arria 10 GX	23722	23	296	<1	344	46	323	39
Stratix V GS	25532	18	2851	<1	344	36	323	31
Stratix V GX	23370	7	1860	<1	344	13	323	92
Stratix V E	23099	7	1843	<1	344	13	323	92
Cyclone V	24111	21	2911	<1	392	32	323	94

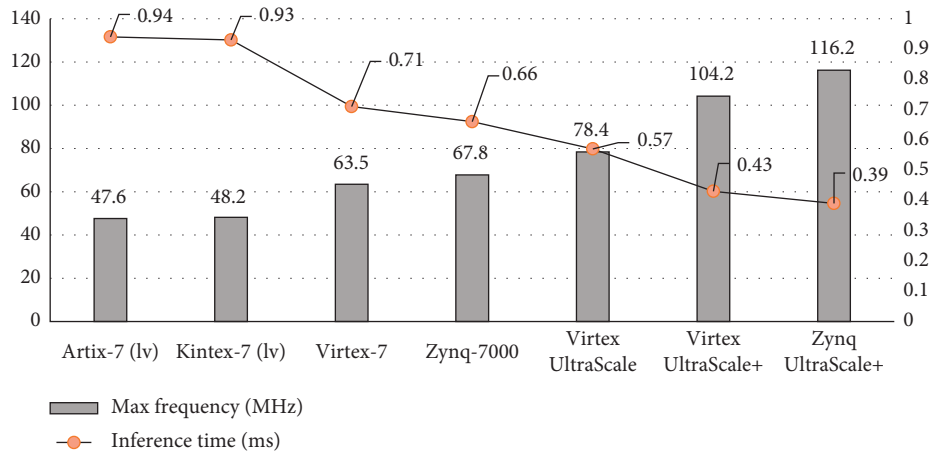


FIGURE 8: Maximum clock frequency and inference time for different Xilinx FPGA families.

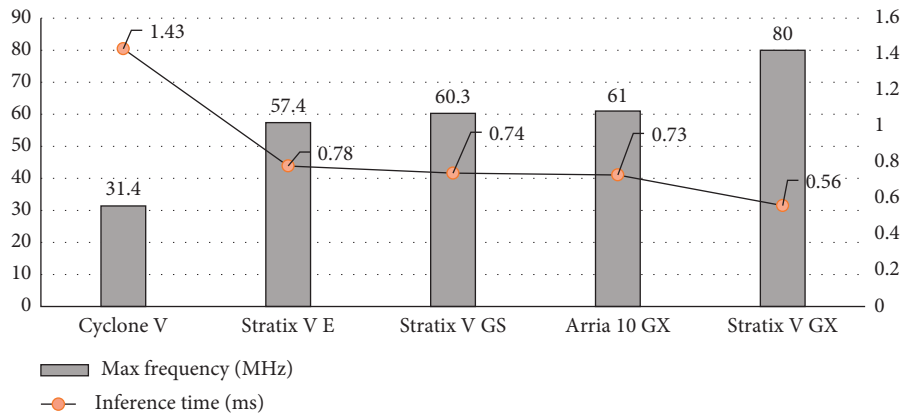


FIGURE 9: Maximum clock frequency and inference time for different Intel FPGA families.

7. Comparison with Intel Movidius Neural Compute Stick

In this section, the FPGA-based accelerator is compared with a commercial hardware accelerator for machine

learning on the edge: the Intel Movidius Neural Compute Stick.

The same model of SCNN keyword spotting was implemented on the NCS in our previous work [12], and a direct comparison between the performances of the two

TABLE 7: Power consumption for Xilinx and Intel FPGAs.

Device	Static power (W)	Dynamic power (W)	Total power (W)
Artix 7	0.151	0.892	1.043
Kintex-7 lv	0.110	0.859	0.969
Zynq-7000	0.215	1.172	1.387
Virtex 7	0.204	1.147	1.351
Virtex-US	0.626	1.235	1.861
Virtex-US+	0.839	1.302	2.141
Zynq-US+	0.627	1.532	2.259
Cyclone V	0.570	1.731	2.301
Stratix V E	1.607	2.150	3.757
Stratix V GS	0.857	3.153	4.010
Arria 10	0.272	0.730	1.002
Stratix V GX	1.244	2.141	3.385

TABLE 8: Design portability analysis for Zynq-7000 family.

Zynq-7000 family	Num mac.	Comb. elem. (%)	Seq. elem. (%)	BRAM (%)	DSP (%)
xc7z030	1	85	<1	92	0
	2	88	<1	92	0
	4	55	<1	92	100
	8	—	—	—	—
xc7z045	1	33	<1	45	0
	2	35	<1	45	0
	4	38	<1	45	0
	8	44	<1	45	0

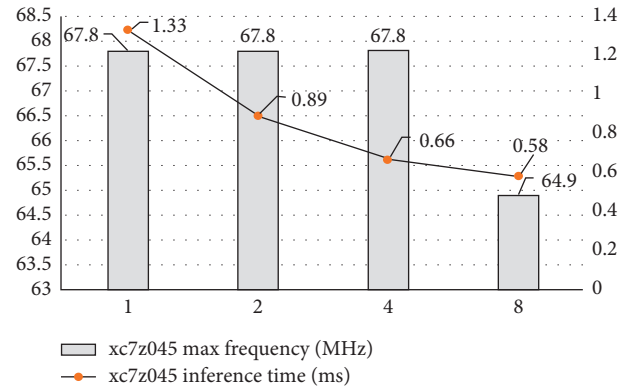
TABLE 9: Design portability analysis for Zynq-US+ family.

Zynq-US+ family	Num mac.	Comb. elem. (%)	Seq. elem. (%)	BRAM (%)	LRAM (%)	DSP (%)
xczu3eg	1	54	<1	100	16	92
	2	64	<1	100	16	100
	4	70	<1	100	16	100
	8	—	—	—	—	—
xczu9eg	1	25	<1	27	0	0
	2	26	<1	27	0	0
	4	30	<1	27	0	0
	8	36	<1	27	0	0

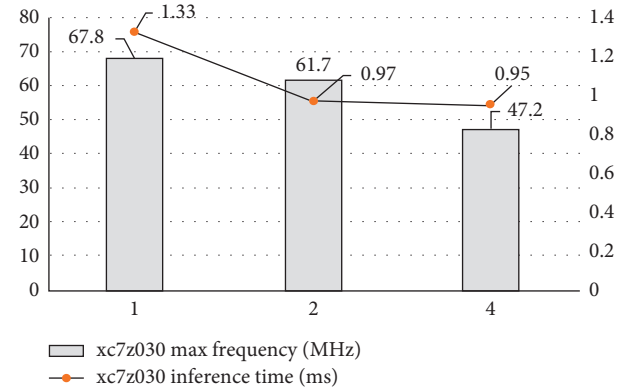
solutions, in terms of inference time, power consumption, and energy per inference, is now presented.

The NCS is a commercial deep learning hardware accelerator hosting the Myriad 2 VPU by Intel Movidius [9]. The VPU includes the following:

- (i) 4Gb of LPDDR3 DRAM
- (ii) 12 very long instruction word (VLIW) streaming hybrid architecture vector engine (SHAVE) processors optimized for machine vision used to run parts of a neural network in parallel
- (iii) 2MB on-chip memory shared between SHAVE processors and fixed-function accelerators
- (iv) 2 Leon microprocessors that coordinate the reception of the network graph file and of inputs via USB connection



(a)



(b)

FIGURE 10: Max frequency and inference time for the Zynq-7000 family.

The Myriad 2 VPU supports fully connected, convolutional (with arbitrary sized kernel), and depthwise convolutional layers.

The NCS implements the floating-point version of the SCNN model with a maximum accuracy of 87.77%. Quantization allows to increase this value to 90.23%, even if our preferred implementation has an accuracy of 88.09%.

The inference time for the SCNN implemented on the NCS is approximately 10 ms. The FPGA-based accelerator has a lower inference time for all the FPGA implementations presented, swinging from 1.45 ms for the Cyclone V to 0.39 ms for the Zynq-US+. Finally, the NCS power consumption is 0.81 W. Such a result is provided by considering the hardware setup of our previous work [12], featuring a Raspberry PI 3B [36] connected to the NCS. Power consumption can be estimated by subtracting the Raspberry PI 3B power consumption in the absence of the NCS (1.3 W) to the total power consumption of the system during an inference (2.11 W).

As shown in Table 7, power consumption for the design implemented on-board all FPGAs is higher than that for the NCS one. Nevertheless, for all the implementations, the energy dissipated during an inference (E_{inf}) is lower than the one of NCS. In fact, it is possible to calculate E_{inf} as shown in the following equation:

$$E_{inf} = P \cdot t_{inf}, \quad (11)$$

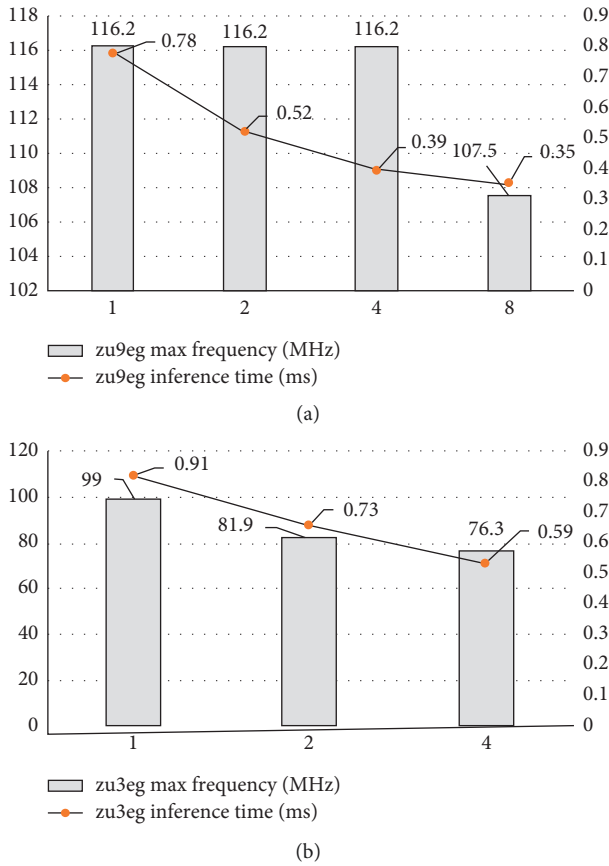


FIGURE 11: Max frequency and inference time for the Zynq-US+ family.

where P is the average power consumption during an inference and t_{inf} is the inference time.

Indeed, even if Xilinx and Intel devices show a higher power consumption, the significantly lower t_{inf} leads to a reduced E_{inf} .

Table 10 shows a comparison among FPGAs and NCS in terms of inference time, power, and energy, by using model (7) of Table 3. The power analysis was performed by considering the maximum achievable clock frequency (f_{clk}) of each FPGA in order to minimize the inference time.

Results show that FPGAs offer great design flexibility, allowing to tune inference time and power consumption through the choice of the different platforms. FPGAs are promising devices for the implementation of CNN-based hardware accelerators for portable applications and in particular for those requiring low latency and high accuracy. Indeed, inference time results to be diminished approximately of a factor between 7 and 25 and energy per inference is reduced, respectively, of a factor between 2.5 and 9 in the investigated cases.

Finally, Figure 12 provides a graphical representation of the power consumption/inference time results shown in Table 10. It is evident from results that all FPGA solutions feature a reduced inference time with respect to the NCS

TABLE 10: Performance comparison between Xilinx FPGAs, Intel FPGAs, and NCS.

Device	f_{clk} (MHz)	Inference time (ms)	Total power (W)	Energy (mJ)
Xilinx FPGA families				
Artix 7	47.6	0.94	1.043	0.98
Kintex-7 lv	48.2	0.93	0.969	0.90
Zynq-7000	67.8	0.65	1.387	0.90
Virtex 7	63.5	0.71	1.351	0.96
Virtex-US	78.4	0.57	1.861	1.01
Virtex-US+	104.2	0.43	2.141	0.92
Zynq-US+	116.4	0.39	2.259	0.88
Intel FPGA families				
Cyclone V	31.4	1.43	2.301	3.29
Stratix V E	57.4	0.78	3.757	2.9
Stratix V GS	60.3	0.74	4.010	2.96
Arria 10	61	0.73	1.002	0.73
Stratix V GX	80	0.56	3.385	1.9
Intel movidius neural compute stick				
NCS	600	10	0.810	8.1

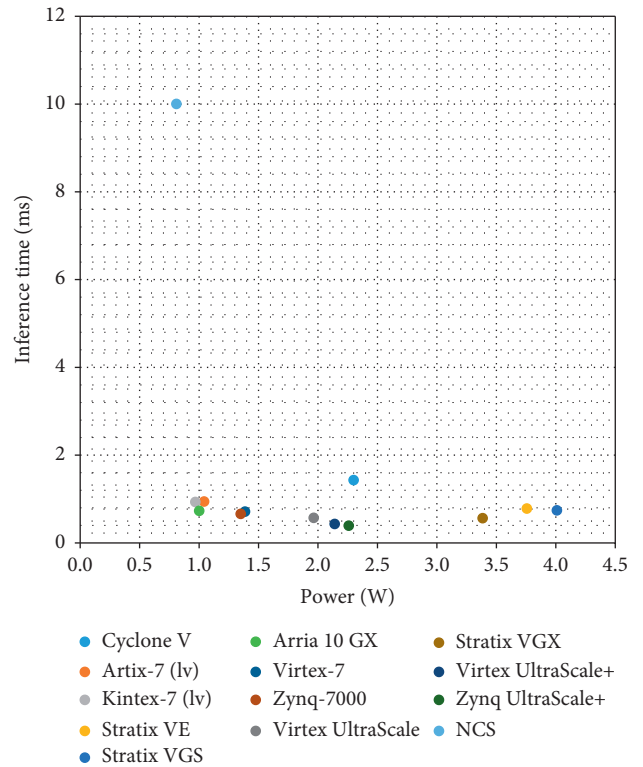


FIGURE 12: Inference time/power consumption trade-off analysis.

implementation at expense of a higher power consumption, even if comparable for some devices.

8. Discussion

The results presented in this work highlight the value of the FPGA solutions to accelerate inference of CNNs. They offer a remarkable trade-off between power consumption and

inference time, resulting in interesting solutions for on the edge computing.

It is necessary to underline that these results were possible, thanks to the use of a CNN model optimized for resource-constrained devices [12], featuring a reduced number of parameters and layers. In view of that, a full on-chip design was achievable, with strong advantages in terms of latency and power consumption. Consequently, results are pertinent for applications requiring relatively small models, such as digit and letter recognitions systems [37, 38], audio [39], and mobile vision applications [40].

Finally, the proposed full on-chip design guarantees a straightforward processing architecture (i.e., no data scheduling from external memories and no management of shared inference processing elements), further reducing the overall system design time. However, when compared with NCS and other *plug and play* solutions, the use of FPGA still requires much more design effort and competences, in view of the higher and heterogeneous design steps (i.e., model quantization and architecture definition) and of the broader design space.

9. Conclusions

This article presents a full on-chip FPGA-based hardware accelerator for on the edge keyword spotting. The KWS system is described focusing on its realization through a machine-learning algorithm and on traducing AI on the edge paradigm.

Starting from a *Keras-Python* model of a KWS based on a SCNN, the parameters of the network were quantized in order to shrink the hardware resources needed for its realization. CNNs have a large number of parameters and are characterized by multiplying and accumulating operations that make their implementation on an FPGA device challenging. Quantization analysis shows that fixed-point representation does not significantly affect model accuracy. On the contrary, it is possible to increase it for particular combinations of input words, weight, and layer output representations. Then, the accelerator architecture is described, focusing on design effort to exploit the intrinsic parallelism of these devices. The SCNN accelerator was implemented on several Xilinx and Intel FPGAs to analyse design portability on different families. The obtained results are presented in terms of maximum achievable clock frequency, hardware resources needed for the network implementation, energy per inference, and power consumption. Finally, the proposed accelerator was compared with a commercial solution for on the edge AI applications: the Intel Movidius NCS. This analysis shows that with a FPGA-based solution, it is possible to overcome NCS performances in terms of inference time and energy per inference.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

- [1] D. M. Ramík, C. Sabourin, R. Moreno, and K. Madani, "A machine learning based intelligent vision system for autonomous object detection and recognition," *Applied Intelligence*, vol. 40, no. 2, pp. 358–375, 2014.
- [2] H. Zhang, K.-F. Wang, and F.-Y. Wang, "Advances and perspective on applications of deep learning in visual object detection," *Acta Automatica Sinica*, vol. 43, no. 8, pp. 1289–1305, 2017.
- [3] L. Zhang, Z. He, and Y. Liu, "Deep object recognition across domains based on adaptive extreme learning machine," *Neurocomputing*, vol. 239, pp. 194–203, 2017.
- [4] R. Nian, B. He, and A. Lendasse, "3D object recognition based on a geometrical topology model and extreme learning machine," *Neural Computing and Applications*, vol. 22, no. 3-4, pp. 427–433, 2013.
- [5] G. Retsinas, G. Sfikas, N. Stamatopoulos, G. Louloudis, and B. Gatos, "Exploring critical aspect of CNN-based keyword spotting. A phocnet study," in *Proceedings of the 13th IAPR International Workshop on Document Analysis Systems*, pp. 13–18, Vienna, Austria, April 2018.
- [6] Y. B. Ayed, D. Fohr, J. P. Haton, and G. Chollet, "Keyword spotting using support vector machines," in *Proceedings of the 5th International Conference on Text, Speech and Dialogue*, vol. 2448, pp. 285–292, Brno, Czech Republic, September 2002.
- [7] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: keyword spotting on microcontrollers," 2017, <https://arxiv.org/abs/1711.07128>.
- [8] Q. Zhang, M. Zhang, T. Chen et al., "Recent advances in convolutional neural network acceleration," *Neurocomputing*, vol. 323, pp. 37–51, 2019.
- [9] Neural compute Stick Documentation. <https://software.intel.com/en-us/movidius-ncs>.
- [10] Google Coral Datasheet: <https://coral.withgoogle.com/tutorials/accelerator-datasheet/>.
- [11] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks," *Neural Computing and Applications*, pp. 1–31, 2018.
- [12] G. Benelli, G. Meoni, and L. Fanucci, "A low power keyword spotting algorithm for memory constrained embedded system," in *Proceedings of the 26th IFIP/IEEE International Conference on Very Large Scale Integration*, Verona, Italy, October 2018.
- [13] Keras Documentation. <https://keras.io/>.
- [14] A. Mohamed, "Deep neural network acoustic models for ASR," Doctoral thesis, Toronto University, Toronto, Canada, 2014.
- [15] S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning, ICML*, vol. 1, pp. 448–456, Lille, France, July 2015.
- [16] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural network," in *Proceedings of the FPGA 2015—2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, Monterey, CA, USA, February 2015.

- [17] J. Qiu, J. Wang, S. Yao et al., "Going deeper with embedded FPGA platform for convolutional neural network," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pp. 26–35, Monterey, CA, USA, February 2016.
- [18] E. Nurvitadhi, D. Sheffield, J. Sim et al., "Accelerating binarized neural networks: comparison of FPGA, CPU, GPU and ASIC," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pp. 77–84, Tokyo, Japan, December 2016.
- [19] S. Moini, B. Alizadeh, M. Emad, and R. Ebrahimpour, "A resource-limited hardware accelerator for convolutional neural networks in embedded vision applications," *IEEE Transactions on Circuits and Systems II Express Briefs*, vol. 64, no. 10, pp. 1217–1221, 2017.
- [20] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang, "Accelerating low bit-width convolutional neural networks with embedded FPGA," in *Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Leuven, Belgium, September 2017.
- [21] J. Park and W. Sung, "FPGA based implementation of deep neural networks using on-chip memory only," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1011–1015, Shanghai, China, March 2016.
- [22] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: a tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.
- [23] G. Feng, Z. Hu, S. Chen, and F. Wu, "Energy-efficient and high-throughput FPGA-based accelerator for convolutional neural networks," in *Proceedings of the 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pp. 624–626, Hangzhou, China, October 2016.
- [24] X. Zhang, X. Liu, A. Ramachandran et al., "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Leuven, Belgium, September 2017.
- [25] J. H. Kim, B. Grady, R. Lian, J. Brothers, and J. H. Anderson, "FPGA-based CNN inference accelerator synthesized from multi-threaded C software," in *Proceedings of the 30th IEEE International System-On-Chip Conference (SOCC)*, pp. 268–273, Munich, Germany, September 2017.
- [26] AMBA Advanced Extensible Interface 4 Specifications. <https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications>.
- [27] Zynq UltraScale+ Family Datasheet. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.
- [28] Virtex UltraScale+ Family Datasheet. https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf.
- [29] Virtex UltraScale Family Datasheet. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf.
- [30] Zynq-7000 Family Datasheet. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.
- [31] Virtex-7, Kintex-7 and Artix-7 Families' Datasheet. https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.
- [32] Arria 10 Family Overview. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_overview.pdf.
- [33] Stratix V Family Datasheet. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-v/stx5_51001.pdf.
- [34] Cyclone V Family Datasheet. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51001.pdf.
- [35] Quartus Prime Standard Edition. <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/archives/qts-qps-handbook-16.0.pdf>.
- [36] Rasberry PI 3B Datasheet. https://www.terraelectronica.ru/pdf/show?pdf_file=%252Fds%252Fpdf%252FT%252FTechicRP3.pdf.
- [37] Y. Hout and H. Zhao, "Handwritten digit recognition based on depth neural network," in *Proceedings of the International Conference on Intelligent Informatics and Biomedical Sciences (ICIIBMS)*, Shanghai, China, November 2017.
- [38] D. C. Ciresan, U. Meier, and J. Schmidhuber, "Transfer learning for Latin and Chinese characters with deep neural networks," in *Proceedings of the 2012 International Joint Conference on Neural Networks (IJCNN)*, Brisbane, Australia, June 2012.
- [39] T. Secu, C. Puhresh, B. Kingsbury, and Y. LeCun, "Very deep multilingual convolutional neural networks for LVCSR," in *Proceedings of the 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Shanghai, China, March 2016.
- [40] A. G. Howard, M. Zhu, B. Chen et al., "MobileNets: efficient convolutional neural networks for mobile vision applications," 2017, <https://arxiv.org/abs/1704.04861>.

Research Article

Implementing and Evaluating an Heterogeneous, Scalable, Tridiagonal Linear System Solver with OpenCL to Target FPGAs, GPUs, and CPUs

Hamish J. Macintosh ^{1,2}, Jasmine E. Banks ¹, and Neil A. Kelson²

¹School of Electrical Engineering and Computer Science, Queensland University of Technology, Brisbane, Queensland 4001, Australia

²eResearch Office, Division of Research and Innovation, Queensland University of Technology, Brisbane, Queensland 4001, Australia

Correspondence should be addressed to Hamish J. Macintosh; hj.macintosh@hdr.qut.edu.au

Received 27 February 2019; Revised 3 August 2019; Accepted 6 September 2019; Published 13 October 2019

Guest Editor: Sven-Bodo Scholz

Copyright © 2019 Hamish J. Macintosh et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Solving diagonally dominant tridiagonal linear systems is a common problem in scientific high-performance computing (HPC). Furthermore, it is becoming more commonplace for HPC platforms to utilise a heterogeneous combination of computing devices. Whilst it is desirable to design faster implementations of parallel linear system solvers, power consumption concerns are increasing in priority. This work presents the *oclspkt* routine. The *oclspkt* routine is a heterogeneous OpenCL implementation of the truncated SPIKE algorithm that can use FPGAs, GPUs, and CPUs to concurrently accelerate the solving of diagonally dominant tridiagonal linear systems. The routine is designed to solve tridiagonal systems of any size and can dynamically allocate optimised workloads to each accelerator in a heterogeneous environment depending on the accelerator's compute performance. The truncated SPIKE FPGA solver is developed first for optimising OpenCL device kernel performance, global memory bandwidth, and interleaved host to device memory transactions. The FPGA OpenCL kernel code is then refactored and optimised to best exploit the underlying architecture of the CPU and GPU. An optimised TDMA OpenCL kernel is also developed to act as a serial baseline performance comparison for the parallel truncated SPIKE kernel since no FPGA tridiagonal solver capable of solving large tridiagonal systems was available at the time of development. The individual GPU, CPU, and FPGA solvers of the *oclspkt* routine are 110%, 150%, and 170% faster, respectively, than comparable device-optimised third-party solvers and applicable baselines. Assessing heterogeneous combinations of compute devices, the GPU + FPGA combination is found to have the best compute performance and the FPGA-only configuration is found to have the best overall estimated energy efficiency.

1. Introduction

Given the ubiquity of tridiagonal linear system problems in engineering, economic, and scientific fields, it is no surprise that significant research has been undertaken to address the need for larger models and higher resolution simulations. Demand for solvers for massive linear systems that are faster and more memory efficient is ever increasing. First proposed in 1978 by Sameh and Kuck [1] and later refined in 2006 [2], the SPIKE algorithm is becoming an increasingly popular method for solving banded linear system problems [3–7].

The SPIKE algorithm has been shown to be an effective method for decomposing massive matrices whilst remaining numerically stable and demanding little memory overhead [8]. The SPIKE algorithm has been implemented with good results to solve banded linear systems using CPUs and GPUs and in CPU + GPU heterogeneous environments often using vendor-specific programming paradigms [6].

A scalable SPIKE implementation targeting CPUs and GPUs in a clustered HPC environment to solve massive diagonally dominant linear systems has previously been demonstrated with good computation and communication

efficiency [5]. Whilst it is desirable to design faster implementations of parallel linear system solvers, it is necessary also to have regard for power consumption, since this is a primary barrier to exascale computing when using traditional general purpose CPU and GPU hardware [9, 10].

FPGA accelerator cards require an order of magnitude less power compared to HPC grade CPUs and GPUs. Previous efforts in developing FPGA-based routines to solve tridiagonal systems have been limited to solving small systems with the serial Thomas algorithm [11–13]. We have previously investigated the feasibility of FPGA implementations of parallel algorithms including the parallel cyclic reduction and SPIKE [14] for solving small tridiagonal linear systems. This previous work utilised OpenCL to produce portable implementations to target FPGAs and GPUs. The current work again utilises OpenCL since this programming framework allows developers to target a wide range of compute devices including FPGAs, CPUs, and GPUs with a unified language.

An OpenCL application consists of C-based kernel code intended to execute on a compute device and C/C++ host code that calls OpenCL API's to set up the environment and orchestrate memory transfers and kernel execution. In OpenCL's programming model, a device's computer resources are divided up at the smallest level as processing elements (PEs), and depending on the device architecture, one or more PEs are grouped into one or many compute units (CUs) [15]. Similarly, the threads of device kernel code are called work items (WIs) and are grouped into work groups (WGs). WIs and WGs are mapped to the PE and CU hardware, respectively.

OpenCL's memory model abstracts the types of memory that a device has available. These are defined by OpenCL as global, local, and private memory. Global memory is generally hi-capacity off-chip memory banks that can be accessed by all PEs across the device. Local memory is on-chip memory and has higher bandwidth and lower capacity than global memory and is only accessible to PE of the same CU. Finally, private memory refers to on-chip register memory space and is only accessible within a particular PE.

The motivation for this paper is to evaluate the feasibility of utilising FPGAs, along with GPUs and CPUs concurrently in a heterogeneous computing environment in order to accelerate the solving of a diagonally dominant tridiagonal linear system. In addition, we aimed at developing a solution that maintained portability whilst providing an optimised code base for each target device architecture and was capable of solving large systems. As such, we present the *oclspkt* routine, an heterogeneous OpenCL implementation of the truncated SPIKE algorithm that can dynamically load balance work allocated to FPGAs, GPUs, and CPUs concurrently or in isolation, in order to solve tridiagonal linear systems of any size. We evaluate the *oclspkt* routine in terms of computational characteristics, numerical accuracy, and estimated energy consumption.

This paper is structured as follows: Section 2 provides an introduction to diagonally dominant tridiagonal linear systems and the truncated SPIKE algorithm. Section 3

describes the implementation of the *oclspkt*-FPGA OpenCL host and kernel code and the optimisation process. This is followed by the porting and optimisation of the *oclspkt*-FPGA kernel and host code to the GPU and CPU devices as *oclspkt*-GPU and *oclspkt*-CPU. Section 3 concludes with discussion of the integration of the three solvers to produce the heterogeneous *oclspkt* solver. In Section 4, the individual solvers are compared to optimised third-party tridiagonal linear systems solvers. The three solvers are further compared in terms of estimated energy efficiency, performance, and numerical accuracy in addition to an evaluation of different heterogeneous combinations of the *oclspkt*. Finally, in Section 5, we draw conclusions from the results and discuss the implications for future work.

2. Background

2.1. Tridiagonal Linear Systems. A coefficient band matrix with a bandwidth of $\beta = 1$ in the linear system $Ax = y$ is considered tridiagonal:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & & & & \\ a_{2,1} & a_{2,2} & a_{2,3} & & & \\ & \ddots & \ddots & \ddots & & \\ & & & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ & & & & a_{n,n-1} & a_{n,n} \end{bmatrix}, \quad (1)$$

$$d = \min \frac{|A_{i,i}|}{\sum_{i \neq j} |A_{i,j}|}. \quad (2)$$

For nonsingular diagonally dominant systems where $d > 1$ in equation (2), a special form of nonpivoting Gaussian elimination called the Thomas algorithm [16] can perform LU decomposition in $\Theta(n)$ operations. The Thomas algorithm provides good performance when solving small tridiagonal linear systems; however, since this algorithm is intrinsically serial, it fails to scale well in highly parallel computing environments. More advanced, inherently parallel methods must be applied if the problem requires solving large systems. Many parallel algorithms exist for solving tridiagonal and block tridiagonal linear systems and are implemented in well-established numerical libraries [17–19].

2.2. The SPIKE Algorithm. The SPIKE algorithm [2] is a polyalgorithm that uses domain decomposition to partition a banded matrix into mutually independent subsystems which can be solved concurrently. Consider the tridiagonal linear system $AX = Y$ where A is $n \times n$ in size with only a single right-hand side vector Y . We can partition the system into p partitions of m elements, where $k = (1, 2, \dots, p)$, to give a main diagonal partition A_k , off-diagonal partitions B_k and C_k , and the right-hand side partition Y_k :

$$A_k = \begin{bmatrix} a_{i,j} & a_{i,j+1} & & & \\ a_{i+1,j} & a_{i+1,j+1} & a_{i+1,j+2} & & \\ & \ddots & \ddots & \ddots & \\ & & a_{i+m-2,j+m-3} & a_{i+m-2,j+m-2} & a_{i+m-2,j+m-1} \\ & & & a_{i+m-1,j+m-2} & a_{i+m-1,j+m-1} \end{bmatrix},$$

$$[B_k, C_k, Y_k] = \begin{bmatrix} 0 & a_{mk+1,m(k-1)} & y_{mk} \\ \vdots & \vdots & \vdots \\ a_{m(k+1)-1,m(k+1)} & 0 & y_{m(j+1)} \end{bmatrix}. \quad (3)$$

The coefficient matrix partitions are factorised so $A = DS$ where D is the main diagonal block matrix and S is the SPIKE matrix as shown in the following equation:

$$DS = \begin{bmatrix} A_1 & & & & \\ & A_2 & & & \\ & \ddots & \ddots & \ddots & \\ & & & A_{p-1} & \\ & & & & A_p \end{bmatrix} \cdot \begin{bmatrix} I & V_1 & & & \\ W_2 & I & V_2 & & \\ & \ddots & \ddots & \ddots & \\ & & & W_{p-1} & I & V_{p-1} \\ & & & & W_p & I \end{bmatrix}. \quad (4)$$

where $V_k = (A_k)^{-1}B_k$ for $k = 1, \dots, p-1$ and $W_k = (A_k)^{-1}B_k$ for $k = 2, \dots, p$. By first solving $DF = Y$, the solution can be retrieved by solving $SX = F$. As $SX = F$ is the same size as the original system, solving for X can be simplified by first extracting a reduced system of the boundary elements between partitions to form $\widehat{S}\widehat{X} = \widehat{F}$ as shown in equation (5), where t and b denote the top- and bottommost elements of the partition:

$$\begin{bmatrix} 1 & 0 & V_1^t & 0 & & & & \\ 0 & 1 & V_1^b & 0 & & & & \\ 0 & W_2^t & 1 & 0 & & & & \\ 0 & W_2^b & 0 & 1 & & & & \\ & \ddots & \ddots & \ddots & & & & \\ & & & & 1 & 0 & V_{p-1}^t & 0 \\ & & & & 0 & 1 & V_{p-1}^b & 0 \\ 0 & W_p^t & 1 & 0 & & & & \\ 0 & W_p^b & 0 & 1 & & & & \end{bmatrix} \cdot \begin{bmatrix} X_1^t \\ X_1^b \\ X_2^t \\ X_2^b \\ \vdots \\ X_{p-1}^t \\ X_{p-1}^b \\ X_p^t \\ X_p^b \end{bmatrix} = \begin{bmatrix} F_1^t \\ F_1^b \\ F_2^t \\ F_2^b \\ \vdots \\ F_{p-1}^t \\ F_{p-1}^b \\ F_p^t \\ F_p^b \end{bmatrix}. \quad (5)$$

The reduced system \widehat{S} is a sparse banded matrix of size $2p \times 2p$ and has a bandwidth of 2. Polizzi and Sameh [2] proposed strategies to handle solving the reduced system. The truncated SPIKE algorithm states for a diagonally dominant system where $d > 1$ (equation (2)) the reduced SPIKE partitions V_k^t and W_k^b can be set to zero [2]. This truncated reduced system takes the form of $p-1$ independent systems as shown in equation (6) which can be solved easily using direct methods:

$$\begin{bmatrix} 1 & V_k^b \\ W_{k+1}^t & 1 \end{bmatrix} \begin{bmatrix} X_k^b \\ X_{k+1}^t \end{bmatrix} = \begin{bmatrix} F_k^b \\ F_{k+1}^t \end{bmatrix}, \quad k = 1, \dots, p-1. \quad (6)$$

With \widehat{X} computed, the remaining values of X can be found with perfect parallelism using the following equation:

$$\begin{cases} A_1 X_1 = F_1 - V_1^b X_2^t, \\ A_k X_k = F_k - V_k^b X_{k+1}^t - W_k^t X_{k-1}^b, & k = 2, \dots, p-1. \\ A_p X_p = F_p - W_p^t X_{p-1}^b, \end{cases} \quad (7)$$

Mikkelsen and Manguoglu [20] conducted a detailed error analysis of the truncated SPIKE algorithm and showed that a reasonable approximation of the upper bound of the infinity norm is dependent on the degree of diagonal dominance, the partition size, and bandwidth of the matrix given by

$$|\widehat{x} - x|_\infty \approx d^{-m/\beta}. \quad (8)$$

3. Implementation

The general SPIKE algorithm consists of four steps: (1) partitioning the system, (2) factorising the partitions, (3) extracting and solving the reduced system, and (4) recovering the overall solution.

For diagonally dominant tridiagonal linear systems, the truncated SPIKE algorithm may be employed. This requires only the bottom SPIKE element v_{km}^b and top SPIKE element w_{km+1}^t in order to resolve the boundary unknown elements x_{km}^b and x_{km+1}^t [2]. This decouples the partition from the rest of the matrix and can be achieved by performing only the forward-sweep steps of LU factorisation, referred to as LU_{FS} , and forward-sweep steps of UL factorisation, referred to as UL_{FS} . LU_{FS} and UL_{FS} will be computed for partitions k and $k+1$ for $k = 1, 2, \dots, p-1$.

The factorised right-hand side elements f_{km}^b and f_{km+1}^t and SPIKE elements w_{km+1}^t and v_{km}^b are used to form and solve the reduced system $\widehat{S}_k \widehat{X}_k = \widehat{F}_k$ using equation (6) to produce x_{km}^b and x_{km+1}^t . This algorithmic step is referred to as RS.

The remaining elements of the solution X_k can then be recovered with equation (7) via the back-sweep step of LU , referred to as LU_{BS} , and the back-sweep step UL factorisation, referred to as UL_{BS} , on the top and bottom half of the partitions k and $k+1$, respectively. We use the Thomas algorithm to compute the forward- and back-sweep factorisation steps giving the overall complexity of our truncated SPIKE algorithm as $\mathcal{O}(n)$. A high-level overview of the anatomy and execution flow of our *oclspkt* routine is shown in Figure 1. The *oclspkt* solver expects the size of the system n , the RHS vector Y , and the tridiagonal matrix split into vectors of its lower diagonal L , main diagonal D , and upper diagonal U as inputs. The solution vector X is returned.

In the following subsections, we describe the truncated SPIKE algorithm implementation for the FPGA (*oclspkt*-FPGA) using OpenCL and the development considerations to obtain optimised performance. As a part of this process, we design and implement an optimised TDMA OpenCL kernel to act as a serial baseline performance comparison for

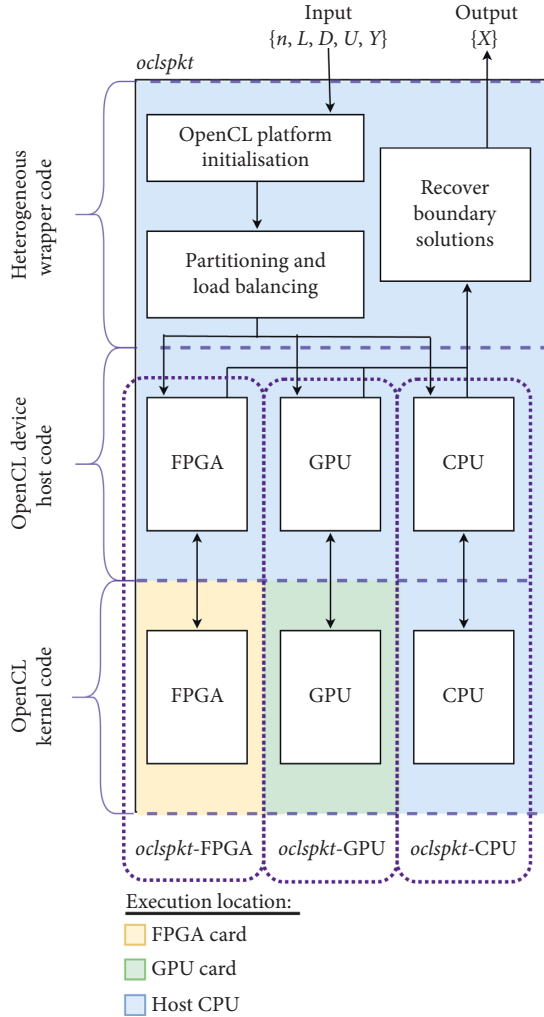


FIGURE 1: An overview of the anatomy and execution flow of the *oclspkt* solver.

the parallel truncated SPIKE kernel. Since the optimised TDMA implementation is constrained by the available global memory bandwidth, we are able to make genuine comparisons of FPGA hardware utilisation and computational complexity for these two kernels.

We then discuss the process of porting and optimising the *oclspkt* code for CPU and GPU. Finally, we describe integrating the three *oclspkt* (FPGA, GPU, and CPU) implementations as a heterogeneous solver. The specific hardware we target for these implementations are Bittware A10PL4 FPGA, NVIDIA M4000 GPU, and the Intel Xenon E5-1650 CPU.

3.1. FPGA Implementation. In order to take advantage of the FPGA's innate pipelined parallelism, we implement both the TDMA and truncated SPIKE algorithm as single work item kernels. A single work item kernel has no calls to the OpenCL API for the local or global relative position in a range of work items. This allows the Intel FPGA compiler to pipeline as much of the kernel code as possible, whilst not having to address WI execution synchronisation and access

to shared memory resources. This reduces the OpenCL FPGA resource consumption overhead, allowing more of the logic fabric to be used for computation.

3.1.1. TDMA Kernel Code. We found no suitable FPGA implementation of a tridiagonal linear system solver able to solve large systems. In order to provide a suitable performance baseline for the more complex SPIKE algorithm, we implemented the Thomas algorithm or TDMA with OpenCL. The TDMA implementation calculates the forward-sweep and backsubstitution for one block of the input system at a time, effectively treating the FPGA's on-chip BRAM as cache for the current working data. The block size m is set as high as possible and is only limited by the available resources on the FPGA. An OpenCL representation of the kernel implementation is shown in Figure 2.

The forward-sweep section loads m elements of the input vectors L, D, U, Y , from off-chip DDR4 RAM to on-chip BRAM. With this input, the upper triangular and modified RHS is calculated, overwriting the initial values of D and Y . D and Y are then written back to DDR4 RAM due to BRAM limitation on the FPGA. The forward-sweep section iterates over $1, \dots, p$ blocks. The back-substitution section loads m elements of D, U , and Y vectors and writes m elements of X after recovering the solution via back-substitution. The back-substitution section iterates over $p, \dots, 1$ blocks.

3.1.2. Truncated SPIKE Kernel Code. An OpenCL algorithm representation of the truncated SPIKE kernel *spktrunc* is shown in Figure 3. The FPGA *oclspkt* implementation executes p iterations of its main loop, loading one block of the linear system given, where block size is m , and we solve 1 partition per iteration of the main loop.

A partition of size m is loaded from global memory and partitioned as per equation (3). $LU_{FS}(k)$ and $UL_{FS}(k)$ are executed concurrently to compute and store half of the upper and lower triangular systems $[D'UY']_k((m/2); m)$ and $[LD'Y']_k(1; (m/2))$, and the SPIKE elements v_k^b and w_k^t , respectively. Next, using y_{k-1}^b and v_{k-1}^b from the previous iteration and y_k^t and w_k^t as inputs for $RS(k)$, the boundary elements x_{k-1}^b and x_k^t are computed.

Finally, $X_k(1; (m/2))$ and $X_{k-1}((m/2); m)$ are then recovered with $UL_{BS}(k)$ and $LU_{BS}(k-1)$ of $[LD'Y']_k(1; (m/2))$ and $[D'UY']_{k-1}((m/2); m)$. $[D'UY']_k((m/2); m)$ and v_k^b are stored for the next iteration of the main loop. The FPGA solver is initialised with an upper triangular identity matrix in $[D'UY']_{k-1}((m/2); m)$ for $k = 0$.

This results in a streaming linear system solver where loading in a block of partitions at the start of the pipeline will compute a block of the solution vector with a $-(m/2)$ element offset.

3.1.3. Host Code. On the host side, in order to interleave the PCIe memory transfers to the device with the execution of the solver kernel, we create in-order command queues for writing to, executing on, and reading from the FPGA.

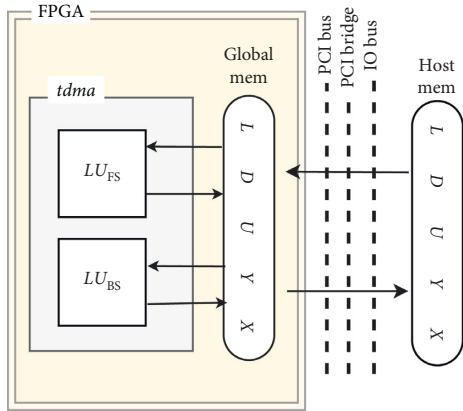


FIGURE 2: The FPGA TDMA OpenCL kernel *tdma*, with the execution path and data dependencies shown. The *tdma* executes as a single WI kernel.

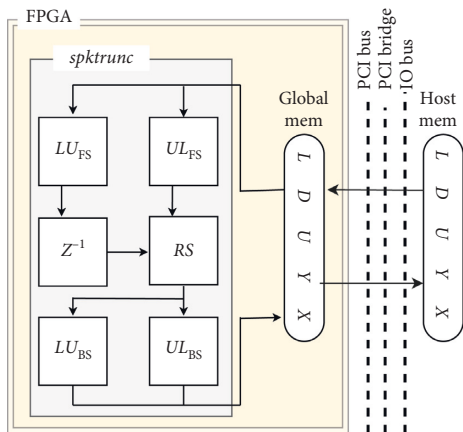


FIGURE 3: The FPGA truncated SPIKE OpenCL kernel *spktrunc*, with the execution path and data dependencies shown. The *spktrunc* executes as a single WI kernel.

We create two copies of read-only memory objects L , D , U , Y and a write-only memory object X . The *spktrunc* kernel and the FPGA's DMA controller for the PCIe bus share the total bandwidth of the DDR4 RAM bank. To maximise FPGA global memory bandwidth, the device memory objects are explicitly designated specific RAM bank locations on the FPGA card in such a way that the PCIe to device RAM, and device RAM to FPGA bandwidth, is optimised.

The execution kernel is enqueued as a 1-by-1-by-1 dimension task with arguments p , L , D , U , Y , and X , where p , the number of partitions to solve, is given by $\text{ceil}((n/m) + 1)$. The execution kernel is scheduled and synchronised with the write and read operations of the device memory objects using OpenCL event objects.

The kernel code is dependent on the partition size m , so memory buffers for the input and output vectors are created as 1-by-size vectors, where size is given by $p \times m$. The input matrix consists of lower, main, and upper diagonal vectors of A and a single right-hand side vector Y is stored in row-major order.

The memory objects L , D , U , Y are padded with an identity matrix and zeros in order to accommodate linear systems where m is not a factor of n , giving the overall memory requirement as $5 \times \text{size}$.

As the kernel is implemented as a single work item, this allows for single-strided memory access patterns when the FPGA loads partitions from global memory for processing. This means that it is not necessary to implement a pre-execution data marshalling stage as is often required for SIMD or SIMD-like processors.

3.1.4. Kernel Complexity and Hardware Utilisation. The FLOP requirements for our TDMA and truncated SPIKE OpenCL kernels are presented in Table 1. The TDMA kernel has significantly fewer FLOPs compared to the truncated SPIKE kernel. This is expected as the TDMA kernel only computes the LU factorisation and back-substitution, compared to the more computationally complex truncated SPIKE polyalgorithm described previously. However, since the TDMA kernel requires the upper triangular matrix of the entire system to be stored to global memory as an intermediate step and then subsequently reread, the TDMA kernel requires double the number of FPGA to off-chip memory transactions in comparison with the truncated SPIKE kernel.

The FLOP and memory transaction requirements shown in Table 1 are reflected in the FPGA kernel hardware utilisation presented in Table 2. OpenCL requires a static partition of the available FPGA hardware resources in order to facilitate host to FPGA memory transfers and OpenCL kernel execution. Per Table 2, this static partition is significant, consuming at least 10% of measured resource types. The total resource utilisation for each kernel is given by the addition of OpenCL static resource utilisation and the kernel-specific resource utilisation.

The more computationally complex truncated SPIKE kernel requires more lookup tables (ALUT), flip-flops (FF), and digital signal processor (DSP) tiles than the TDMA kernel. The TDMA kernel however requires more block RAM (BRAM) tiles due to implementing a greater number of load store units to cater for the extra global memory transactions. Furthermore, both kernels are constrained by the available amount of BRAM on the FPGA, with the BRAM utilisation by far the highest resource utilisation for both kernels.

3.1.5. FPGA OpenCL Optimisation Considerations. Our implementation of the truncated SPIKE algorithm is global memory bandwidth constrained. It requires large blocks of floating point data to be accessible at each stage of the algorithm. By far the largest bottleneck to computational throughput is ensuring coalesced aligned global memory transactions.

When loading matrix partitions from the global to local or private memory, a major optimisation consideration is the available bandwidth on the global memory bus. The available bandwidth per memory transaction is 512 bits, and the load-store units that are implemented by the Intel FPGA

TABLE 1: FLOP and global memory transactions required for the TDMA and truncated SPIKE FPGA kernels.

Operation	TDMA	Truncated SPIKE
ADD/SUB	$3mp$	$(5m + 3)p$
MUL	$3mp$	$(6m + 5)p$
DIV	$3mp$	$(3m + 3)p$
MEM	$10mp$	$5mp$

TABLE 2: FPGA hardware utilisation for the OpenCL static portion, TDMA, and truncated SPIKE kernels, and total utilisation for each implementation (static + kernel).

Resource	OpenCL static (%)	TDMA		Truncated SPIKE	
		Kernel (%)	Total (%)	Kernel (%)	Total (%)
ALUTs	13	14	27	18	31
FFs	13	10	23	13	23
BRAMs	16	52	68	49	65
DSPs	10	16	26	27	37

compiler are of the size 2^b bits where $b_{\min} = 9$ and b_{\max} is constrained by the available resources on the FPGA. Therefore, to ensure maximum global memory bandwidth with the aforementioned constraints, we set partition size m to 32 for single precision floating point data for both kernels resulting in 1024 bit memory transactions.

The value of m is hard-coded and known at compile time allowing for unrolling of the nested loops at the expense of increased hardware utilisation. Unrolling a loop effectively tells the compiler to generate a hardware instance for each iteration of the loop, meaning that if there are no loop-carried dependencies, the entire loop is executed in parallel. However, for loops with carried dependencies such as *LU/UL* factorisation, each iteration cannot execute in parallel. Nonetheless, this is still many times faster than sequential loop execution despite the increase in latency that is dependent on the loop size.

Loop unrolling is our primary computational optimisation step, thereby allowing enough compute bandwidth for our kernel to act as a streaming linear system solver. Note that in our optimisation process, we either fully unroll loops or not at all. It is possible to partially unroll loops for a performance boost when hardware utilisation limitations do not permit a full unroll. Partially unrolling a loop can however be inefficient since the hardware utilisation does not scale proportionally with the unroll factor due to the hardware overhead required to control the loop execution.

3.2. Porting the Truncated SPIKE Kernel to CPU and GPU.

In order to investigate the full potential for a heterogeneous computing implementation of the truncated SPIKE algorithm for solving tridiagonal linear systems of any size, we exploited the portability of OpenCL. We modified the host and kernel code used for the FPGA implementation to target CPU and GPU hardware. To achieve this, it was necessary to make modification to the host and kernel side memory objects and data access patterns, remap the truncated SPIKE algorithm to different kernel objects, and modify the work

group sizes and their mapping to compute units with respect to CPU and GPU hardware architecture.

3.2.1. Partitioning and Memory Mapping. The memory requirements for CPU and GPU implementations are dependent on the partitioning scheme for each device. The memory required to solve for a partition of size m multiplied by a multiple of the GPU's preferred work group size must be less than the available local memory. This constrains the size and number of partitions m , since it is preferable to maximise the occupancy of the SIMD lane whilst ensuring that sufficient local memory is available.

Unlike the GPU, all OpenCL memory objects on the CPU are automatically cached into local memory by hardware [21]. However, considering that the CPU has a lower compute unit count, we maximise the partition size m to minimise the number of partitions, thereby minimising the operation count required to recover the reduced system. The relative values for p and size in terms of m and work group size are shown in Table 3.

For our implementation of the truncated SPIKE algorithm for the CPU and GPU, the host and kernel memory requirements are five 1-by-size vectors, L, D, U, Y, X , of the partitioned system and four 1-by- $(p+2)$ vectors, V, W, Y^t, Y^b , of the reduced system. By storing the values for V, W, Y^t, Y^b in a separate global memory space, we remove the potential for bank conflicts in memory transactions that may occur if the reduced system vectors are stored in place in the partitioned system.

The reduced system memory objects are padded with zeros to accommodate the top- and bottommost partitions $j = 0$ and $j = p$ removing the need for excess control code in the kernel to manage the topmost and bottommost partitions. Further, to ensure data locality for coalesced memory transactions on both the CPU and GPU, the input matrix is transformed in a preexecution data marshalling step. The data marshalling transforms the input vectors so that data for adjacent work items are sequential instead of strided. This allows the data to be automatically cached and for

TABLE 3: *oclspkt* kernel partitioning schemes.

Device	Partitions (p)	size
FPGA	$\text{ceil}(n/m) + 1$	$p \times m$
GPU	$n / (WG_{\text{size}} \times m)$	$p \times m \times WG_{\text{size}}$
CPU	$n / (CU_s \times WG_{\text{size}})$	$p \times CU \times WG_{\text{size}}$

vector processing of work items on the CPU and for full bandwidth global memory transactions on the GPU.

3.2.2. Remapping the Kernel. In contrast to the FPGA implementation of the truncated SPIKE algorithm, for the CPU and GPU implementations, we split the code into two separate kernels for the CPU and three separate kernels for the GPU. This allows for better work group scheduling and dispatching for multiple compute unit architectures as we enqueue the kernels for execution as arrays of work items known as an NDRange in OpenCL's parlance.

In remapping the kernel to the CPU, the underlying architecture provides relatively few processing elements per compute unit and a fast clock speed. As such, in order to make best use of this architecture, the number of partitions of the truncated SPIKE algorithm should be minimised, thereby ensuring allocation of work groups of the maximum possible size to each compute unit to ensure maximum occupancy. Figure 4 shows an OpenCL representation of the CPU implementation, its execution order, and data path.

For the CPU implementation, we use the partitioning scheme proposed by Mendiratta [22]. We compute the *LU* and *UL* forward-sweep factorisation in the *spkfac_{cpu}* kernel where we apply *UL* factorisation to elements 0 to $2m_{\min}$ and apply *LU* factorisation to elements m_{\min} to m where m_{\min} is the smallest partition size required to purify the resulting factorisations of error as per equation (8). This reduces the overall operation count and is only possible when $m \gg m_{\min}$. The *spkfac_{cpu}* kernel is enqueued as a p -by-1-by-1 NDRange, in a single in-order command queue.

The reduced system and the recovery of the overall solution are handled by a second kernel *spktrrec*. The *spktrrec* kernel is enqueued as per *spkfac_{cpu}* in a single in-order command queue. The reduced system is solved, and the boundary unknown elements are recovered and used to compute the *UL* back-sweep of elements 0 to $m_{\min} - 1$ and the *LU* back-sweep of elements m to m_{\min} .

In contrast to the CPU, the GPU has many processing elements per compute unit and a relatively low clock speed. In order to optimise performance, it was important to maximise the number of partitions of the SPIKE algorithm, by reducing the partition size and thereby ensuring maximum occupancy of the processing elements. Figure 5 shows an OpenCL representation of the GPU implementation, its execution order, and data path.

For the GPU, partitioning the system and the *LU* and *UL* factorisation of the code are handled by the first kernel, *spkfact_{gpu}*. Unlike the CPU, the GPU computes the entire block size m of the *UL* and the *LU* factorisations. Only the top half of the *UL* and the bottom half of the *LU* results are then stored in global memory in order to reduce global

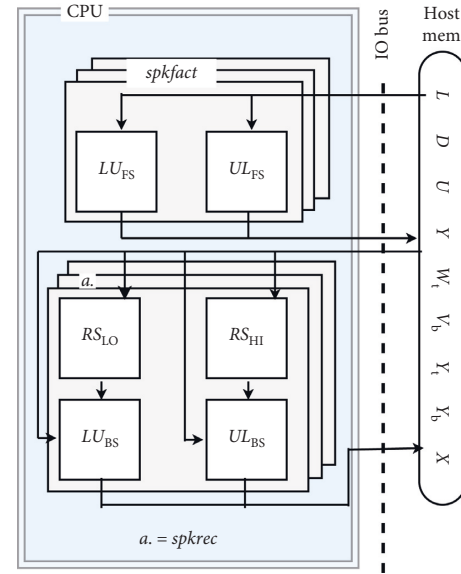


FIGURE 4: The CPU truncated SPIKE OpenCL kernels *spkfact* and *spkrec*, with the execution path and data dependencies shown. Both kernels are executed as an NDRange of work items.

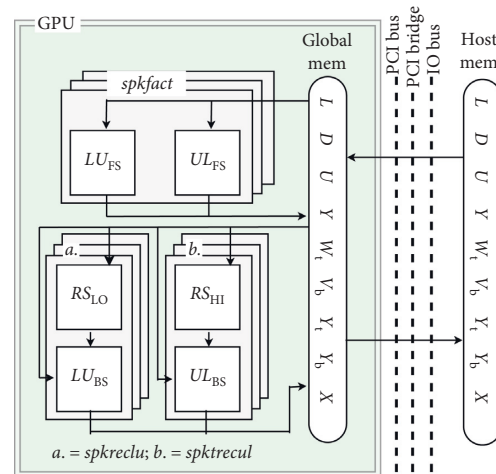


FIGURE 5: The GPU truncated SPIKE OpenCL kernels *spkfact*, *spktrrecl*, and *spktrrecul*, with the execution path and data dependencies shown. All kernels are executed as an NDRange of work items.

memory transactions and overall global memory space requirements. The reduced system and the recovery of the overall solution are handled by two kernels, *spktrrecl* and *spktrrecul*. *spktrrecl* and *spktrrecul* only load the bottom half of partition m and top half of partition m , respectively, to compute the back-sweep portions of *LU* and *UL* factorisation. The three kernels are again enqueued as p -by-1-by-1 NDRange in in-order command queues for writing to, executing on, and reading from the GPU. As with the FPGA, this effectively interleaves the PCIe data transfer with kernel execution.

3.3. The Heterogeneous Solver. We further extend our truncated SPIKE implementation to utilise all available

computation resources available on a platform, as shown in Figure 1.

This heterogeneous solver first checks for available devices on the host using OpenCL APIs and then queries if device profiling data exist for found devices. If profiling data are not available for all devices, each device will be allocated an even portion of the input system and profiling data will be collected on the next execution of the solver. Otherwise, each device will be allocated a portion of the input system determined by the percentage of the total system throughput over the individual devices’ previously recorded throughput. Throughput in this case includes data transit time across the PCIe bus, data marshalling, and compute time of the kernel.

The heterogeneous solver then asynchronously dispatches chunks of the input data to the devices, executes the device solvers, and recovers the solution. The interchunk boundary solutions recovered from the devices are cleansed of error by executing a “top” level of the truncated SPIKE algorithm on the chunk partitions.

4. Evaluation

In the following subsections, we evaluate the *oclspkt* routine in terms of compute performance, numerical stability, and estimated power efficiency. The results presented use single precision floating point and all matrices are random and nonsingular, and the main diagonal has a diagonal dominance factor $d > 3$.

All results presented in this paper have been executed on a Dell T5000 desktop PC with an Intel Xeon CPU E5-1620 v4, 64 GB of RAM, a Bittware A10PL4 FPGA, and a NVIDIA M4000 GPU; full specifications are listed in Table 4.

4.1. Compute Performance. To evaluate the compute performance of the *oclspkt*, we first only consider the kernel execution time for our target devices in isolation, assuming pre-distributed memory. In Figure 6, we show the time to solve a system where $N = 256 \times 10^6$, specifically identifying the solution and data marshalling components of the overall execution time. We set N to 256×10^6 in-order to showcase the best possible performance for the computing-device only without introducing PCI memory transactions. In this experiment, the GPU kernel takes on average 78.4 ms to solve the tridiagonal system, where the FPGA and CPU are 2.6 and 4.8× slower at 200 ms and 376 ms, respectively. Furthermore, when also considering the data marshalling overheads required by the GPU and CPU kernels, the GPU is still the quickest at 152 ms with the FPGA and CPU now 1.3 and 6.1× slower.

The compute performance figures are not surprising when we consider that the performance of *oclspkt* is bound by the available memory bandwidth. For large matrices, the global memory transactions required for *oclspkt*-GPU and *oclspkt*-CPU are $\approx 13N$ where the *oclspkt*-FPGA solver requires $\approx 5N$. We can estimate the performance of the compute devices using the required memory transactions of the individual solvers and with the total available memory bandwidth of the devices. Performance estimation is calculated using MT/B , where MT is the number of required

TABLE 4: Specifications for Dell T5000 desktop PC.

Component	Specification
CPU	Intel Xeon E5-1620 v4 @ 3.50 GHz
GPU	NVIDIA M4000 8 GB GDDR5 PCIe G3 x16
FPGA	Bittware A10PL4 w/ Intel Arria 10 GX 8 GB DDR4 PCIe G3 x8
RAM	64 GB DDR4 @ 2400 MHz
OS	CentOS 7.4 ICC 18.0.3
Software	CUDA 9.0 Intel Quartus Pro 17.0 Intel OpenCL SDK 7.0.0.2568

memory transactions and B is the maximum available memory bandwidth. The estimated relative estimated performance is calculated to be 1, 2.17, and 4.57× slower for the GPU, FPGA, and CPU, respectively (normalised for the GPU). These values closely correspond to the measured relative performance of the kernel solve time in Figure 6.

In Figure 7, we compare these results to other diagonally dominant tridiagonal solver algorithms, our TDMA FPGA kernel, a CUDA-GPU implementation, *dgtsv*, [6], the Intel MKL *sdtsvb* routine [23], and a sequential CPU implementation of the TDMA. For each of the three target devices, our *oclspkt* implementation outperforms the comparison routines for solving a tridiagonal system of $N = 256 \times 10^6$. The *oclspkt* (FPGA) is 1.7× faster than the TDMA (FPGA) kernel, the *oclspkt* (GPU) implementation is 1.1× faster than the *dgtsv*, and our *oclspkt* (CPU) is 1.5 and 3.5× faster than the *sdtsvb* and TDMA CPU solvers, respectively. Note that for each of these results, we include any data marshalling overhead, but exclude host to PCIe device transfer time.

A comparison of the compute performance targeting single and heterogeneous combinations of devices executing the *oclspkt* routine is shown in Figure 8. We normalise the performance metric to rows solved per second (RSs^{-1}) to provide a fair algorithmic comparison across different device hardware architectures. Furthermore, when evaluating the heterogeneous solver performance of *oclspkt*, we use a holistic system approach, which includes the host to device PCIe data transfer times for the FPGA and GPU devices, and all data marshalling overheads. As shown in Table 5, the GPU + FPGA device combination has the best average maximum performance. The GPU + FPGA device combination performs 1.38× better than the next best device, the GPU-only, and performs 2.48× better than the worst performing device, the CPU-only implementation.

Curiously, we would expect performance metrics of the heterogeneous combinations of devices to be close to the summation of the individual device performance metrics. In fact, our results show that only the GPU + FPGA heterogeneous performance is close to the summation of the GPU and FPGA-only performance at 88% of the theoretical total. The CPU + FPGA, CPU + GPU, and CPU + GPU + FPGA average maximum performance are only 65%, 51%, and 55%, respectively.

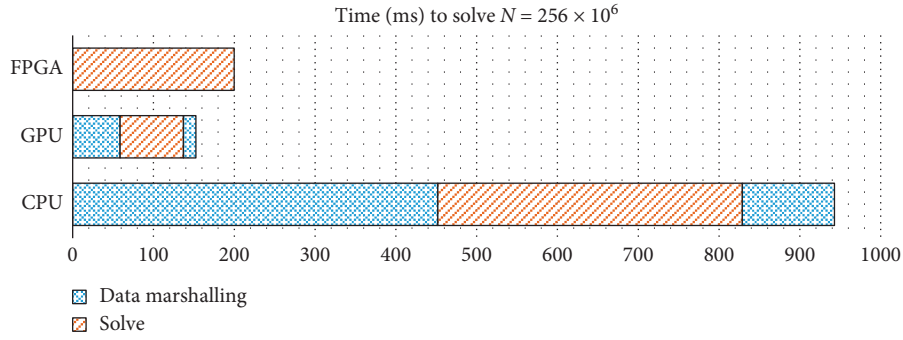


FIGURE 6: Time (ms) to solve a system of size $N = 256 \times 10^6$ using *oclspkt*, targeting CPU, GPU, and FPGA devices.

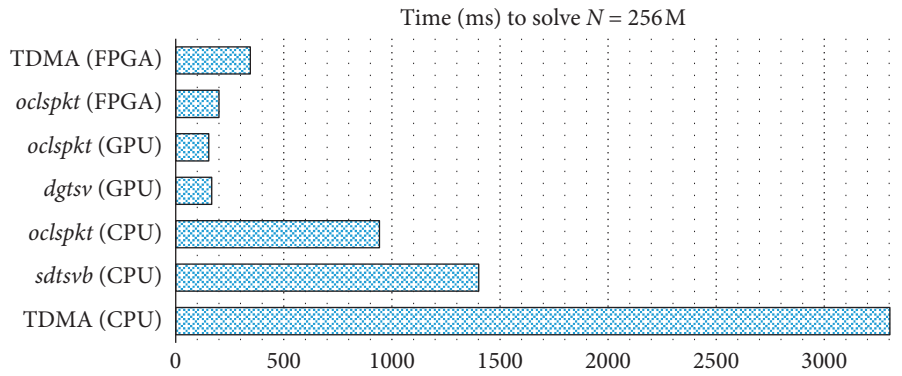


FIGURE 7: Comparing time (ms) to solve a system of size $N = 256 \times 10^6$ using *oclspkt*, *dgtsv*, *sdtsvb*, and TDMA.

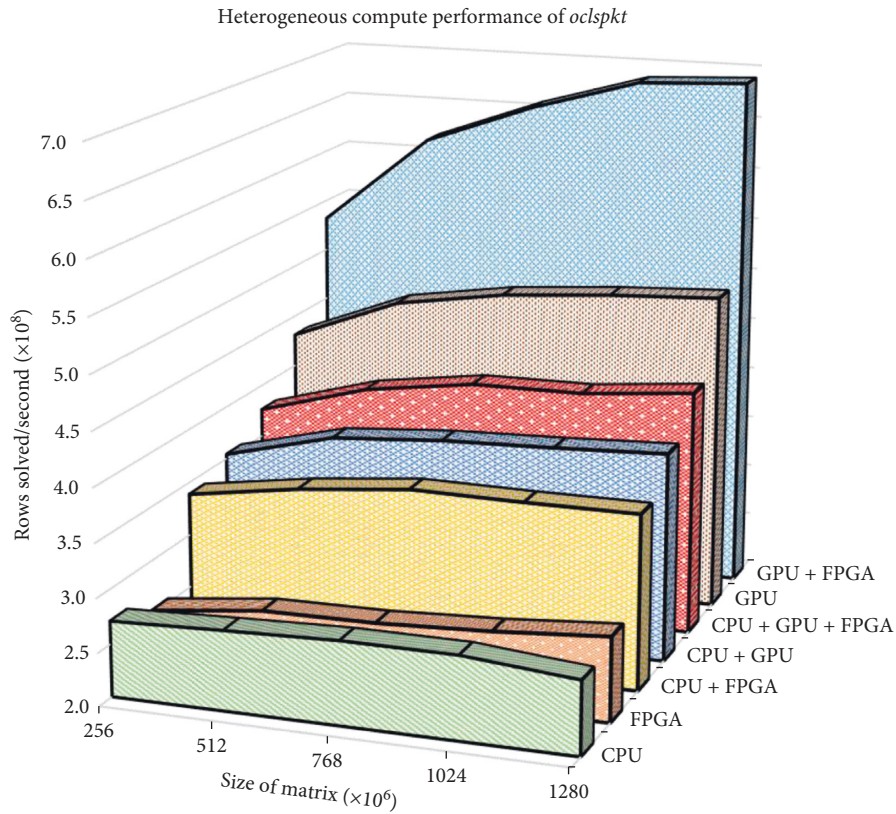


FIGURE 8: Performance comparison in rows solved per second for $N = (256 \dots 1280) \times 10^6$ when targeting CPU, GPU, FPGA, and heterogeneous combinations of devices.

TABLE 5: Maximum observed average ($n = 16$) compute performance and estimated energy efficiency of *oclspkt* for devices and heterogeneous combinations.

Device	Compute performance	Estimated energy efficiency
CPU	279	1.39
GPU	501	12.9
FPGA	280	28.6
CPU + GPU	396	1.67
CPU + FPGA	365	1.81
GPU + FPGA	691	15.6
CPU + GPU + FPGA	431	2.06

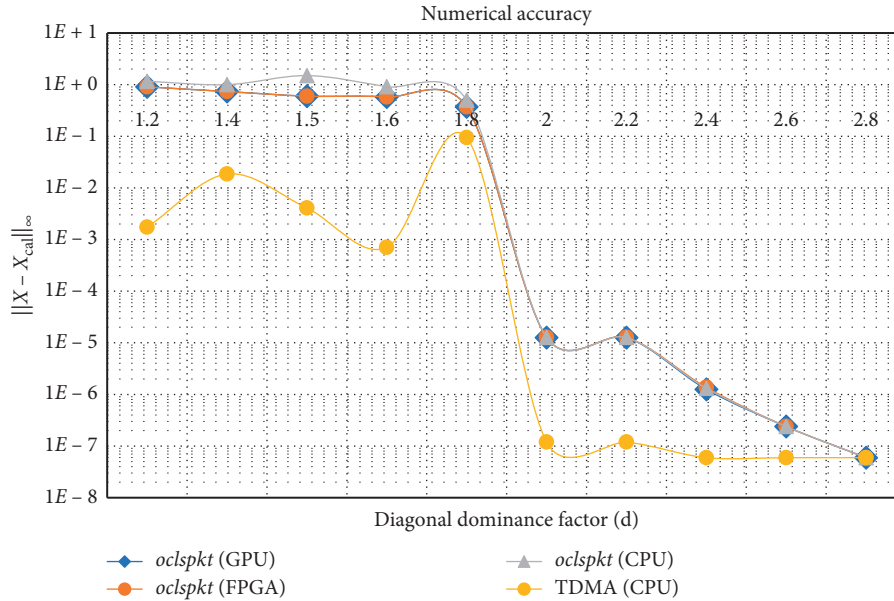


FIGURE 9: Numerical accuracy of *oclspkt* for varying diagonal dominance compared to CPU TDMA solver.

For the PCIe attached devices, the GPU and FPGA performance metrics are determined by the available PCIe bus bandwidth. We have provisioned primary and secondary memory spaces in the GPU and FPGA global memory space. This allows the *oclspkt* routine to execute the OpenCL kernel on the primary memory space whilst writing the next input chunk to the secondary memory space. This ensures the PCIe bus, and available memory bandwidth is being used in an efficient way. That is to say, the OpenCL kernel execution time (Figure 6) is completely interleaved with the host to device memory transfers. As such, the M4000 GPU card with 16 PCIe Gen 3.0 lanes available will outperform the A10PL4 FPGA card with 8 PCIe Gen 3.0 lanes regardless of the kernel compute performance. Similarly, the CPU performance is determined by the available host RAM bandwidth.

Using the de facto industry standard benchmark for measuring sustained memory bandwidth, STREAM [24, 25], our desktop machine, specified in Table 4, has a maximum measured memory bandwidth of 42 GBs^{-1} . Profiling our CPU implementation of *oclspkt* using Intel VTune Amplifier shows very efficient use of the available memory bandwidth with sustained average 36 GBs^{-1} and peak 39 GBs^{-1} memory bandwidth utilisation. This saturation of host memory

bandwidth by the CPU solver creates a processing bottleneck and negatively affects the PCIe data transfer to the FPGA and GPU. This, coupled with the heterogeneous partitioning scheme described in subsection 3.3, will favour the increase in the chunk size of the input system allocated for CPU computation on each successive invocation of the *oclspkt* routine and subsequently decrease the performance of GPU and FPGA devices.

One thing that may increase the performance of the CPU + [GPU | FPGA | GPU + FPGA] combinations of solvers is to change the workload partitioning scheme to only look at data marshalling and kernel execution times, excluding the PCIe data transfer times. This would mean on successive calls of the routine the host would be tuned to send more workload to the GPU and FPGA and minimise the workload allocated to the CPU, thus improving performance.

4.2. Numerical Accuracy. In Figure 9, we show the numerical accuracy of the *oclspkt* in terms of the infinity norm of the known and calculated results, varied by the diagonal dominance of the input matrix compared to the TDMA CPU implementation. The TDMA approaches the machine

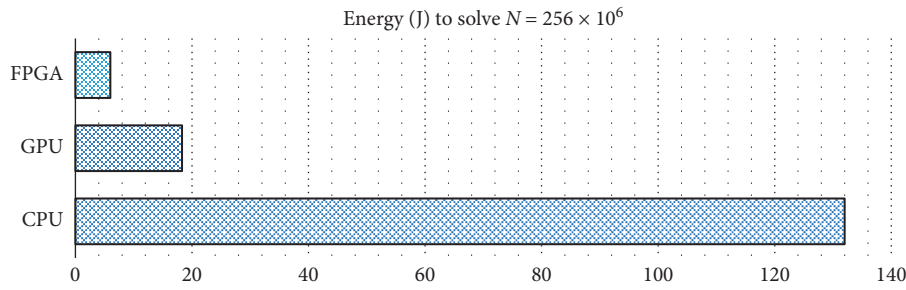


FIGURE 10: Joules required to solve a tridiagonal system of $N = 256 \times 10^6$ per device using the *oclspkt* routine.

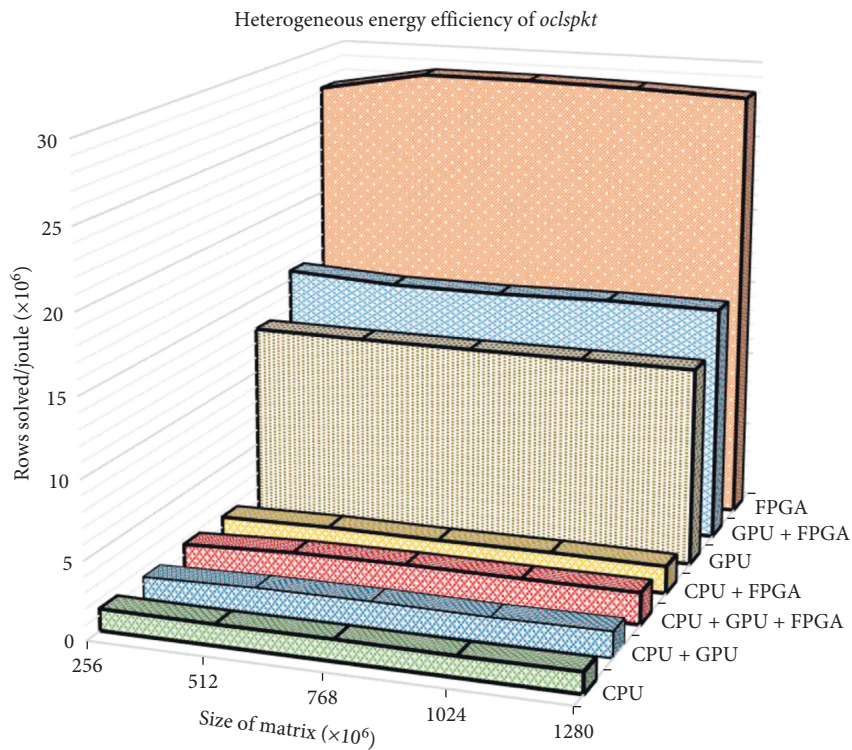


FIGURE 11: Estimated energy efficiency: comparison in rows solved per second for $N = (256 \dots 1280) \times 10^6$ for *oclspkt* when targeting CPU, GPU, FPGA, and heterogeneous combinations of devices.

epsilon value for single precision floating point numbers when the diagonal dominance of the input system is 2, whereas the *oclspkt* for the CPU, GPU, and FPGA requires a diagonal dominance of 2.8 to achieve a similar accuracy. Equation (8) shows that an approximation of the upper bound of the infinity norm error is dependent on the SPIKE partition size, the bandwidth, and the degree of diagonal dominance. As the GPU and FPGA partition sizes and the small CPU partition sizes are equal, that is, $m_{\text{GPU}} = m_{\text{FPGA}} = m_{\text{CPU}_{\text{min}}}$, the numerical accuracy for all implementations is expected to be very similar.

4.3. Estimated Energy Consumption and Efficiency. To determine estimated energy consumption in Joules for each device, we used the manufacturer's rated thermal design power (TDP) and multiplied it by the kernel execution time for data marshalling and solved steps of the *oclspkt*. TDP

represents the average power in watts used by a processor when the device is fully utilised. Whilst this is not a precise measurement of power used to solve the workload, it nevertheless provides a relative interdevice benchmark. The TDP of the M4000 GPU is 120 W, the Xeon E-1650 is 140 W, and the A10PL4 FPGA is 33 W.

When solving a system of $N = 256 \times 10^6$ as shown in Figure 10, the FPGA implementation uses 2.8 \times less and 20 \times less energy than the GPU and CPU implementations, respectively. Further, in Figure 11, we can see the estimated energy efficiency of each hardware configuration of *oclspkt* in rows solved per Joule. Across the range of the experiment, each solver shows consistent results with the FPGA-only solver estimated to be the most energy-efficient peaking at 28×10^6 rows solved per Joule.

The FPGA-only solver is estimated to be on average 1.8 \times more energy efficient than the next best-performing solver, the GPU + FPGA, and is 20.0 \times more energy efficient than the

poorest performing CPU-only solver. This is not surprising since the TDP for the FPGA is an order of magnitude smaller than the other devices. Similarly to the heterogeneous results in subsection 4.1, the addition of the CPU solver significantly constrains the available bandwidth to host memory slowing down the PCIe data transfer rates. In turn, this pushes more work to the CPU solver and slows down the overall compute, and since the CPU has the highest TDP, this exacerbates the poor energy efficiency.

5. Conclusion

In this paper, we presented a numerically stable heterogeneous OpenCL implementation of the truncated SPIKE algorithm targeting FPGAs, GPUs, CPUs, and combinations of these devices. Our experimental case has demonstrated the feasibility of utilising FPGAs, along with GPUs and CPUs concurrently in a heterogeneous computing environment in order to accelerate the solving of a diagonally dominant tridiagonal linear system. When comparing our CPU, GPU, and FPGA implementation of *oclspkt* to a suitable baseline implementation and third-party solvers specifically designed and optimised for these devices, the compute performance of our implementation showed 150%, 110%, and 170% improvement, respectively.

Profiling the heterogeneous combinations of *oclspkt* showed that targeting the GPU + FPGA devices gives the best compute performance and targeting FPGA-only will give the best estimated energy efficiency. Also adding our highly optimised CPU implementation to a heterogeneous device combination with PCIe attached devices significantly reduced the expected performance of the overall system. In our experimental case, with a compute environment that has CPUs, GPUs, and FPGAs, it is advantageous to relegate the CPU to a purely task orchestration role instead of computation.

Under our experimental conditions, all device compute performance results are memory bandwidth constrained. Whilst the GPU kernel compute performance is several times faster than the FPGA kernel, the FPGA test hardware has several times less available memory bandwidth. As high-bandwidth-data transfer technology is introduced to the new generations of FPGA accelerator boards, this performance gap between devices is expected to close. Given the significantly lower power requirements, incorporation of FPGAs has the potential to reduce some of the power consumption barriers currently faced by HPC environments as we move towards exascale computing.

A natural progression of this work would be to extend the *oclspkt* routine to be able to solve nondiagonally dominant and block tridiagonal linear systems. Further, it would be advantageous to extend the heterogeneous partitioning routine to be able to tune the solver to maximise energy efficiency where desired. A part of this extension would involve a more detailed power analysis and direct in-line monitoring of the host power usage.

An extension of this work may also seek to account for memory bottlenecks detected on successive invocations of the solver further enhancing performance in heterogeneous applications.

Data Availability

The source code and data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This project utilised the High-Performance Computing (HPC) Facility at the Queensland University of Technology (QUT). The facility is administered by QUT's eResearch Department. Special thanks go to the eResearch Department for their support, particularly for providing access to specialist FPGA and GPU resources.

References

- [1] A. H. Sameh and D. J. Kuck, "On stable parallel linear system solvers," *Journal of the ACM*, vol. 25, no. 1, pp. 81–91, 1978.
- [2] E. Polizzi and A. H. Sameh, "A parallel hybrid banded system solver: the SPIKE algorithm," *Parallel Computing*, vol. 32, no. 2, pp. 177–194, 2006.
- [3] E. Polizzi and A. Sameh, "SPIKE: a parallel environment for solving banded linear systems," *Computers & Fluids*, vol. 36, no. 1, pp. 113–120, 2007.
- [4] M. Manguoglu, F. Saied, A. Sameh, and A. Grama, "Performance models for the SPIKE banded linear system solver," in *Proceedings of the 2010 Ninth International Symposium on Parallel and Distributed Computing (ISPDC)*, IEEE, Istanbul, Turkey, July 2010.
- [5] X. Wang, Y. Xu, and W. Xue, "A hierarchical tridiagonal system solver for heterogenous supercomputers," in *Proceedings of the 2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, IEEE, New Orleans, LA, USA, November 2014.
- [6] L.-W. Chang, J. A. Stratton, H.-S. Kim, and W.-M. W. Hwu, "A scalable, numerically stable, high-performance tridiagonal solver using GPUs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE Computer Society Press, Salt Lake City, UT, USA, November 2012.
- [7] H. Gabb, "Intel® adaptive SPIKE-based solver," Technical report, Intel, Santa Clara, CA, USA, 2010.
- [8] L. W. Chang and W. M. Hwu, *A Guide for Implementing Tridiagonal Solvers on GPUs*, Springer, Berlin, Germany, 2014.
- [9] J. Mair, Z. Huang, D. Eyers, and Y. Chen, "Quantifying the energy efficiency challenges of achieving exascale computing," in *Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 943–950, IEEE, Shenzhen, China, May 2015.
- [10] M. U. Ashraf, F. Alburaei Eassa, A. Ahmad Albeshri, and A. Algarni, "Performance and power efficient massive parallel computational model for HPC heterogeneous exascale systems," *IEEE Access*, vol. 6, pp. 23095–23107, 2018.
- [11] D. J. Warne, N. A. Kelson, and R. F. Hayward, "Solving tridiagonal linear systems using field programmable gate arrays," in *Proceedings of the 4th International Conference on*

- Computational Methods (ICCM 2012)*, Gold Coast, QLD, Australia, November 2012.
- [12] D. J. Warne, N. A. Kelson, and R. F. Hayward, "Comparison of high level FPGA hardware design for solving tri-diagonal linear systems," *Procedia Computer Science*, vol. 29, pp. 95–101, 2014.
 - [13] S. Palmer, *Accelerating Implicit Finite Difference Schemes Using a Hardware Optimised Implementation of the Thomas Algorithm for FPGAs*, Cornell University, Ithaca, NY, USA, 2014.
 - [14] H. Macintosh, D. Warne, N. A. Kelson, J. Banks, and T. W. Farrell, "Implementation of parallel tridiagonal solvers for a heterogeneous computing environment," *The ANZIAM Journal*, vol. 56, pp. 446–462, 2016.
 - [15] B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, "Chapter 2—introduction to OpenCL," in *Heterogeneous Computing with OpenCL*, B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, Eds., pp. 15–38, Morgan Kaufmann, Burlington, MA, USA, 2nd edition, 2013.
 - [16] B. P. Flannery, S. Teukolsky, W. H. Press, and W. T. Vetterling, *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, Cambridge University Press, Cambridge, UK, 1992.
 - [17] P. Arbenz, A. Cleary, J. Dongarra, and M. Hegland, "A comparison of parallel solvers for diagonally dominant and general narrow-banded linear systems II," in *Euro-Par'99 Parallel Processing*, pp. 1078–1087, Springer, Berlin, Germany, 1999.
 - [18] C. R. Dun, M. Hegland, and M. R. Osborne, "Parallel stable solution methods for tridiagonal linear systems of equations," in *Proceedings of the Computational Techniques and Applications Conference (CTAC95)*, pp. 267–274, World Scientific Publishing, River Edge, NJ, USA, August 1996.
 - [19] M. Hegland, "On the parallel solution of tridiagonal systems by wrap-around partitioning and incomplete LU factorization," *Numerische Mathematik*, vol. 59, no. 1, pp. 453–472, 1991.
 - [20] C. C. K. Mikkelsen and M. Manguoglu, "Analysis of the truncated SPIKE algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 4, pp. 1500–1519, 2008.
 - [21] Intel Corporation, "Developer guide for Intel® Sdk for openCL™ applications," 2018, <https://software.intel.com/en-us/openclsdk-devguide-2017>.
 - [22] K. Mendiratta, "a banded SPIKE algorithm and solver for shared memory architectures," Masters' thesis, University of Massachusetts, Amherst, MA, USA, 2011.
 - [23] Intel Corporation, "?dtsvb," 2018, <https://software.intel.com/en-us/mkl-developer-reference-c-dtsvb>.
 - [24] J. D. McCalpin, "STREAM: sustainable memory bandwidth in high performance computers," Technical report, University of Virginia, Charlottesville, VA, USA, 1995.
 - [25] J. McCalpin, *Memory Bandwidth and Machine Balance in High Performance Computers*, IEEE Technical Committee on Computer Architecture Newsletter, 1995.