Tampere University

SAMAN PAYVAR

GPU-based Architecture Modeling and Instruction Set Extension for Signal Processing Applications

Tampere University Dissertations 733

Tampere University Dissertations 733

SAMAN PAYVAR

GPU-based Architecture Modeling and Instruction Set Extension for Signal Processing Applications

ACADEMIC DISSERTATION
To be presented, with the permission of
the Faculty of Information Technology and Communication Sciences
of Tampere University,
for public discussion in the auditorium TB109
of the Tietotalo, Korkeakoulunkatu 1, Tampere,
on 20 January 2023, at 12 o'clock.

ACADEMIC DISSERTATION
Tampere University, Faculty of Information Technology and Communication Sciences
Finland

Responsible supervisor and Custos	Professor Timo D. Hämäläinen Tampere University Finland	
Pre-examiners	Professor Juha Plosila University of Turku Finland	Docent Sebastien Lafond Åbo Akademi University Finland
Opponent	Professor Shuvra Bhattacharyya University of Maryland United States	

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

Cover design: Roihu Inc.

ISBN 978-952-03-2722-4 (print)
ISBN 978-952-03-2723-1 (pdf)
ISSN 2489-9860 (print)
ISSN 2490-0028 (pdf)
http://urn.fi/URN:ISBN:978-952-03-2723-1

Carbon dioxide emissions from printing Tampere University dissertations have been compensated.

PunaMusta Oy – Yliopistopaino
Joensuu 2022

PREFACE

This work has been carried out in the Faculty of Information Technology and Communication Sciences at Tampere University during 2017-2022.

I would like to express my appreciation to my supervisor Professor *Timo D. Hämäläinen* for his guidance. Grateful acknowledgments go also to the reviewers of my thesis, Professor *Juha Plosila* and Docent *Sebastien Lafond* for their comments and the opponent Professor *Shuvra S. Bhattacharyya*. In addition, I would like to thank Professor *Jani Boutellier* and Professor *Maxime Pelcat* for their advice. I am very grateful for the opportunity of working at the Unit of Computing Sciences and I want to thank all my colleagues and supervisors.

I am grateful to my family for supporting me throughout my education.

Tampere, December 2022
Saman Payvar

ABSTRACT

The modeling of embedded systems attempts to estimate the performance and costs prior to the implementation. The early stage predictions for performance and power dissipation reduces the more costly late stage design modifications. Workload modeling is an approach where an abstract application is evaluated against an abstract architecture. The challenge in modeling is the balance between fidelity and simplicity, where fidelity refers to the correctness of the predictions and the simplicity relates to the simulation time of the model and its ease of comprehension for the developer. A model named GSLA for performance and power modeling is presented, which extends existing architecture modeling by including GPUs as parallel processing elements. The performance model showed an average fidelity of 93% and the power model demonstrated an average fidelity of 84% between the models and several application measurements. The GSLA model is very simple: only 2 parameters that can be obtained by automated scripts.

Besides the modeling, this thesis addresses lower level signal processing system improvements by proposing Instruction Set Architecture (ISA) extensions for RISC-V processors. A vehicle classifier neural network model was used as a case study, in which the benefit of Bit Manipulation Instructions (BMI) is shown. The result is a new PopCount instruction extension that is verified in ETISS simulator. The PopCount extension of RISC-V ISA showed a performance improvement of more than double for the vehicle classifier application. In addition, the design flow for adding a new instruction extension for a re-configurable platform is presented.

The GPU modeling and the RISC-V ISA extension added new features to the state of the art. They improve the modeling features as well as reduce the execution costs in signal processing platforms.

CONTENTS

List of Figures

List of Tables

ABBREVIATIONS

API	Application Programming Interface
BMI	Bit Manipulation Instruction
CNN	Convolutional Neural Network
DSP	Digital Signal Processor
ETISS	Extendable Translating Instruction Set Simulator
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
GPU	Graphics Processing Unit
I/O	Input Output
ISA	Instruction Set Architecture
LSLA	Linear System Level Architecture
MoA	Model of Architecture
MoC	Model of Computation
OpenCL	Open Computing Language
PULP	Parallel Ultra Low Power
RISC-V	Reduced Instruction Set Computer Five

ORIGINAL PUBLICATIONS

Publication I S. Payvar, J. Boutellier, A. Morvan, C. Rubattu and M. Pelcat. Extending Architecture Modeling for Signal Processing towards GPUs. *2019 27th European Signal Processing Conference (EUSIPCO)*. 2019, 1–5. DOI: 10.23919/EUSIPCO.2019.8903094.

Publication II S. Payvar, M. Pelcat and T. D. Hämäläinen. A model of architecture for estimating GPU processing performance and power. *Design Automation for Embedded Systems*. 2021, 43–63. DOI: 10.1007/s10617-020-09244-4.

Publication III S. Payvar, M. Khan, R. Stahl, D. Mueller-Gritschneder and J. Boutellier. Neural Network-based Vehicle Image Classification for IoT Devices. *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2019, 148–153. DOI: 10.1109/SiPS47522.2019.9020464.

Publication IV S. Payvar, E. Pekkarinen, R. Stahl, D. Mueller-Gritschneder and T. D. Hämäläinen. Instruction Extension of a RISC-V Processor Modeled with IP-XACT. *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. 2019, 1–5. DOI: 10.1109/NORCHIP.2019.8906975.

1 INTRODUCTION

Computer systems are composed of software, middleware, and hardware. The software can be abstracted as a collection of commands to be executed on the hardware. Typically, hardware platforms have multiple Processing Units (PUs). The middleware provides abstraction between software and hardware, and can regard the workload scheduling where the SW commands are mapped to the PUs of the hardware.

Machine learning algorithms require high computing performance and there is a need to execute them in a wide range of platforms from Internet of Things (IoT) devices to cloud servers. Quite often there is a limited execution time budget. The challenge is that energy consumption should be minimized while the output quality and performance requirements have to be satisfied. Traditionally, the optimization process starts by modifying the application and then further system level optimizations are considered by changing the mapping or the platform. In these cases, the application workload mapping on the platform variations must be evaluated.

A model-based approach is often used to speed up the development time rather than trying out real implementations. There are multiple feasible system modeling methods. This thesis uses a modular approach, which considers the application and platform models separately. This system modeling concept was originally introduced by Pelcat et al. [34] and named Linear System Level Architecture (LSLA) and provides an obvious division and reusability for application and platform models. In LSLA, the Model of Computation (MoC) presents the application and the Model of Architecture (MoA) represents the platform. The original LSLA supports only multi-core CPU platforms, and performance modeling. To better support machine learning applications, this thesis presents an extension to cover Graphical Processing Unit (GPU) units and power modeling also.

In this way the goal of this thesis is to improve system level modeling features at a high abstraction level. The second goal is to improve real platform performance at a lower abstraction level study. Machine learning applications are often written

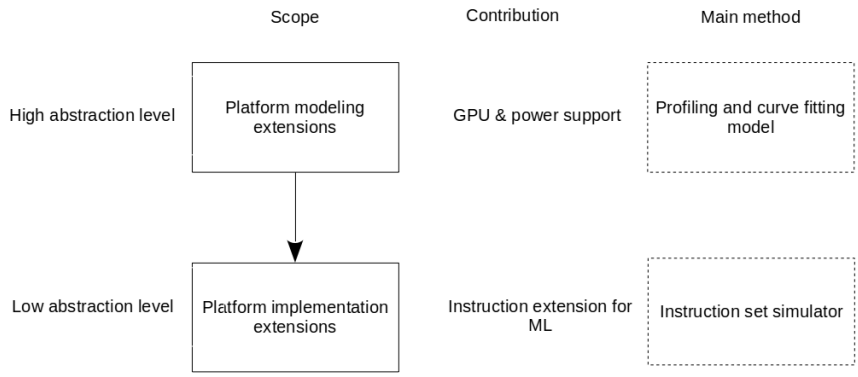

Figure 1.1 Overview of the research conducted in this thesis

in OpenCL which motivates the GPU extension for the LSLA. For platform optimization, the way that extra instructions are added is studied. It is expected that the Instruction Set Architecture (ISA) extensions will reduce the number of compiled assembly instructions, resulting in faster execution time. Reduced Instruction Set Computer Five (RISC-V) ISA is open source and a widely adopted ISA which makes it a suitable selection for this research. Common classification applications like the Convolutional Neural Network (CNN) benefit greatly from Bit Manipulation Instructions (BMI) extensions which at the beginning of this study were not standardized in RISC-V. Implementation of a non-standard RISC-V extension is another aspect of the research problem for system optimizations in this thesis.

Figure 1.1 presents an overview of the research conducted in this thesis: the abstraction levels according to the scope, novel contributions, and main methods applied in the research.

1.1 Objectives and scope of research

The objective of this thesis is to improve implementations of machine learning type applications. The focus is on the modeling of the workload mapping on the platform and ISA extensions. The target is to present performance and power models with as high fidelity as possible for the platform, i.e., at least 80 percent to be useful for early state architecture exploration. The modeling of the applications and their platform independent optimizations are beyond the scope of this thesis. The second goal is

to achieve at least double the performance improvement with ISA extensions for a neural network application on an IoT device.

1.2 Research problems

The research problems investigated in this thesis are platform modeling and ISA extensions. The modeling questions include the following:

- What is the simplest system model that achieves a reasonable fidelity value for useful modeling accuracy?
- Could the performance model also be used for power modeling to keep the model as simple as possible?
- How to expand the LSLA so that OpenCL applications running on GPUs can be modeled?

The ISA extension questions consist of the following:

- What is the performance gain and effort of adding instruction extensions to machine learning algorithms?
- What kind of custom instruction helps neural network execution on RISC-V?

1.3 Summary of contributions

Publication [26] describes the performance model for an OpenCL application on GPU equipped platforms. The model is linear and it is named GSLA. It introduces minimal computation for fast workload analysis resulting in reasonable workload balancing between processing elements. The performance studies show an average fidelity value of 93%.

Publication [31] depicts the power model for the OpenCL applications and provides a power dissipation analysis. Utilizing this model facilitates workload spreading between processing elements for optimal power dissipation. The power dissipation inspections reveal an average fidelity value of 84%.

Publication [27] introduces the Population Count(PopCount) extension for RISC-V ISA, which improves the performance of a vehicle classifier CNN application more

than double. This work includes the total instruction count and type analysis as well as their impact on performance.

Publication [29] presents a design flow for including the new instructions in the compiler and simulator of the RISC-V ISA. The effort analysis estimates around two hours for a new instruction inclusion.

1.3.1 Author's contribution to published work

The author is the main author and the main contributor in all included publications. The main tasks were performed by the author including the literature reviews, software development work, measurements, analysis of the results, and typing of the articles.

Assistant Professor Jani Boutellier helped to develop the ideas in *Publication*Payvar et al., [31] and [27], assisted with the writing, and gave valuable comments.

Prof. *Maxime Pelcat* helped to develop the ideas in *Publication*[26] and *Publication* [31] and gave instructive comments.

Dr. Antoine Morvan and Claudio Rubattu, M.Sc., helped to develop the idea in *Publication*[26] and [31].

Mir Khan, M.Sc., provided the binarized version of the vehicle classifier application and helped with the writing of *Publication* [27].

Rafael Stahl, M.Sc., and Dr. Daniel Mueller-Gritschneder helped with *Publication* [27] and *Publication* [29], and provided valuable comments.

Esko Pekkarinen, M.Sc., helped to develop the idea in Publication [29] and with the writing of the article.

Prof. Timo D. Hämäläinen helped to develop the idea in *Publication* [29] and [31] and with the writing of the articles.

1.4 Outline of the thesis

This thesis consists of four publications and an introductory part. Chapter 2 briefly presents the related work. Chapter 3 introduces the performance and power models for workload distribution. Chapter 4 explains the non-standard RISC-V ISA extensions. Chapter 5 considers the execution cost improvements of the implementations. Chapter 6 presents the main results and conclusions of the thesis.

2 RELATED WORK

System optimizations for a specific application are considered by two approaches utilized simultaneously or separately. The first approach is by efficient workload balancing between the processing elements of heterogeneous platforms. The second approach is architecture modifications, in this thesis the system's processing elements with ISA extensions. In this section, the related work for both approaches is presented.

2.1 Workload modeling

In platforms with multiple processing units, the optimal platform utilization requires balanced work allocation between them. This is achievable by many methodologies, and Table 2.1 summarizes the major works related to them.

The related work is divided into two subgroups which focus on fidelity evaluation or optimization. This thesis concentrates on fidelity evaluation while the optimization works are presented here to provide a broader view. The studies use different metrics to evaluate their proposed estimation models, i.e., the fidelity of their work. The suitability of their selected metric is beyond the scope of this thesis but could be further investigated, e.g., in [13].

2.1.1 Fidelity work

The main related work is the Linear System-Level Architecture (LSLA) [34]. LSLA presents the hardware as a model of architecture (MoA) and describes the application with a mode of computation (MoC). System optimization iteration is shown as the mapping of the MoC activity on the MoA. An LSLA MoA is composed of the processing elements and the communication nodes, and each component has a cost function where the targeted cost could differ such as the execution time or the

energy consumption. The total execution cost of the system is calculated as the sum of the execution costs of the processing elements and the communication nodes. The authors present experiments with C applications running on a multi-core CPU platform using the PREESM tool [33]. They used the *kendall tau* as the fidelity measurement to evaluate their experiments and reported a fidelity of 86% for the energy consumption modeling.

The period graph study [3] estimates the system throughput from the system simulation data and the maximum cycle mean. They experimented with a nested genetic and a simulated annealing algorithm for voltage scaling optimization. Their graph consists of four main elements including named nodes, striped nodes, directed edges, and solid small black circles on the edges. For period graph construction, first the simulation data is used to identify the period. Then, the tasks of the period are considered as the nodes and the idle time of the period is demonstrated with striped nodes. Later, the nodes connections as the edges are assigned according to their order in the period from the last node to the first node. They use the send and receive nodes for Inter Processor Communication (IPC) to show the data transmission. They are connected by the edges of the send node to the receive node. Also, a delay is applied to an edge by adding a black circle to visualize the inter-iteration dependencies. They reported a fidelity value of more than 0.65 for a system with six processors.

The ARM's big.LITTLE model [41] facilitates the core selection of heterogeneous architecture. Their model is based on a correlation between last-level data cache (LLC) misses and cycles per instruction (CPI) for efficient core selection. They use two linear equations for presenting the model where one equation is for the small cores and the other equation is for the big cores. Their model provides a soft threshold setting where the larger number of LLC misses are scheduled on little cores. They use a linear equation with two parameters for CPI calculations where MPI is multiplied by the first parameter and added to the second parameter. They experimented with HEVC and LDPC decoder applications and observed the effectiveness of application division into multi-phases. They reported a negligible error of 9.71% for A15 and 21.67% for A7, and error reduction as the LLC misses increased.

The Runtime system for User Preferences-defined Schedules (RUPS) [7] is a runtime scheduling tool for multi-core architectures, which provides a trade-off between three variables named PE, i.e., performance, E, i.e., energy consumption, and FT, i.e., fault tolerance. Their methodology depicts the best scheduling trade-off deci-

sion from the user preferences. They considered task duplication for fault tolerance where a copy of a task named duplicate was assigned to another processing element which was replaced by the initial task in case of failure. RUPS consists of four parts including system check, user preferences, schedule creation, and runtime system. First, the system check gets the system data and then the user preferences are collected. Later, the schedule is optimized according to the user requests and the system data and then the runtime system receives the schedule. They experimented using real machines supporting DVFS with four scenarios of only PE, PE and FT, only FT, and only E. They reported an average maximum deviation of 7.14% and 1.64% for energy prediction.

The thermally aware energy efficiency model of [38] utilizes environmental temperature variations and eliminates the power sensor requirement of the multiprocessor platform. Their model includes a variable for the temperature which is used for application reconfiguration. They considered the platform configuration points for energy modeling of the heterogeneous platforms with Hardware Program Counters (HPC). They experimented with the ODROID XU3 platform in three different environments where the board fan condition and temperatures were different in order to provide a cold, middle, and hot case. They observed large energy level variations with temperature changes from cold to hot and their results show 33% energy improvements for modeling with temperature awareness.

The ANNETTE [46] shows a performance estimation model for Deep Neural Network (DNN) applications. They focus on mapping of the layers according to their stacked model. Their framework has mapping and mixed layer models and generates models according to the kernel and benchmark data. They use two separate platforms, i.e., ZCU102 SoC board and Intel Neural Compute Stick 2 (NCS2) and two network applications for execution time evaluations. They used the Spearman metric to evaluate their model on 34 random models of the NASBench dataset and reported 0.988 as the fidelity value of their model.

2.1.2 Optimization work

The parallel dataflow study by [12] uses the PREESM tool to explore the energy efficient mapping of a parallel application on a platform with a quad-core CPU. Their work depicts the effect of the static power domination of contemporary platforms on

the energy consumption of parallel applications. They showed that limiting the execution speed of the application for controlling the clock frequency of the processor core by meta data input to the power management system resulted in energy-efficient mapping. They used Levenberg-Marquardt's algorithm for calculating a third degree polynomial as their power model. They experimented with three different mappings of the Sobel filter application including completely sequential, completely parallel, and mix-parallel mappings. They reported energy savings of over 50% compared to the reference.

The runtime mapping optimization research in [47] considers the energy efficiency of several parallel applications running on a multi-core platform. They utilized a hybrid approach of both mapping and frequency adjustment. Their management strategy has three steps. First, for each application some mapping options and their minimum frequencies are calculated at design time. Later, at runtime, the minimum common frequency is selected according to the design time data, which shows the relevant mappings. Finally, the mappings are combined. In the case of violation of the timing constraints, the second and third steps are redone. Their design time data consists of the execution trace which is the start time and the end time of the mapping options at the defined frequencies. Also, the Minimum Allowed Frequency (MAF) based on [12] is determined. Their results depict 2x reduction in energy consumption.

The workload type study by [11] investigates the effect of the instructions type on the big.LITTLE platform. They showed the impact of the type of the executed instruction and the stress on the micro architecture. The execution operation of different machine instructions utilizes diverse transistors, thus the power dissipation on the processor varies. Their experiments with six benchmarks depict around 40% workload type impact on power dissipation. Their workload type aware scheduling includes a power and a performance model for each workload type where the energy consumption of a schedule is calculated according to the relevant models. Their scheduler has four variations for power and performance, both including and ignoring them. The best case studies showed 31.3% energy savings.

The passive-active flow graphs (PAFGs) of [20] proposes a computation model for the signal processing systems. A PAFG is developed as an application graph based on core functional dataflow (CFDF). The PAFG expands the dataflow graphs to cover several inputs and outputs with a different presentation where the vertices

show both buffers and computational models and the edges depict their connections. The PAFG blocks are passive or active, where an active block shows both computational blocks and buffers but a passive block depicts only buffers. In addition, they introduce the passivization transformation which is the conversion of active blocks to passive ones. Their experiments with the error vector magnitude and jitter measurement applications show improvements in throughput and buffer memory requirement (BMR).

The intermediate directed acyclic graph (DAG) representation work by [2] shows an improved model for resource allocation. Its implementation in the SPIDER tool utilizes single-rate directed acyclic graph (SR-DAG) data. They use dependency equations for the numerical modeling of the SR-DAG and define additional equations for hierarchy inclusion from the πSDF MoC. Their relaxed execution model on interface-based synchronous dataflow (IBSDF) maximizes the throughput of the hierarchical application. The proposed method eliminates the SR-DAG building and storing stage but increases the complexity of the resource allocation algorithm. They experimented with the greedy scheduling of four applications including SqueezeNet, Reinforcement Learning, Stabilization, and Sobel-Morpho. They report an overhead reduction in computation time and memory footprint besides performance improvement. Their results show an average memory reduction of 97.34% for scheduling and mapping. Also, a minimum decrease of 47.11% in total execution time for the resource allocation stage is reported.

The Data flow Method for Hardware/Software Exploration (DAMHSE) [42] represents a fast hardware-software prototyping method. A compilable/synthesizable code is provided by performing a design space exploration (DSE). Their experiments with the parallel edge detection of an image processing system shows performance elevation by 21.6x. They used hardware accelerators and an edited PREESM tool for optimized tasks for actor scheduling and mapping in addition to code generation. They described a complementary application management system which dynamically distributes tasks between processing elements. Their manager has a low penalty of maximum 20% of the execution time which makes it suitable for low power systems. They report high computational performance with reasonable overhead for dynamic application mapping.

The OpenCL as abstract hardware research [36] shows a system modeling framework with multiple MoCs. They implemented a portable heterogeneous architec-

ture for system modeling. Their model-based design framework arranges a system into different directors where each one shows a semantic model. The models are based on a MoC which executes according to implemented semantics. An application scenario contains multiple directors, which is a heterogeneous combination of MoCs. They evaluated the utilization of OpenCL by mapping an application modeled by Synchronous Data Flow (SDF) on different platforms. They experimented with a palatalized 2-D image processing application for face detection. The face detection has three modules for skin detection, skin quantization, and image contouring. They conclude that OpenCL eases the targeted automatic synthesis.

The Distributed Operation Layer (DOL) by [44] is a framework for the efficient execution of parallel applications on heterogeneous platforms with multiple processors. DOL provides automation for optimized algorithm mapping on the architecture according to the targeted goal. In addition, DOL facilitates system level performance analyzing with two static and dynamic models. The static model does not consider resource sharing and assumes smooth patterns of operation and communication, whereas the dynamic model considers bursts, blocking, and synchronization. The DOL framework has four parts including application specification, architecture specification, functional simulation, mapping optimization and performance evaluation while it outputs the mapping specification. They experimented with MPEG-2 decoder mapping using a static model for performance analysis. Their experimental architecture has two identical tiles communicating through a NoC while each tile consists of a RISC and a DSP. They presented three optimal mappings and selected one of them as the best choice because of the workload variations.

ArchC [39] is an open-source processor architecture description language (ADL) based on SystemC. It aids architecture design in the early stages by providing a SystemC executable and automatically generating verification tools like a simulator. ArchC compares the model editions against the functional description by inspecting the memory hierarchy and other integration of IPs in a design. Also, it supplies multiple simulator options, i.e., interpreted or compiled and several memory selections like memory levels or cache. ArchC has two inputs for describing the targeted architecture. One is the explanation of the resources like a pipeline structure or memory hierarchy, and the other is a statement of the instruction set architecture, such as the format of the instructions or the opcodes. They simulated multiple processor architectures using ArchC. They have MIPS and Intel 8051 with both cycle-accurate and

functional models and several architectures, such as PowerPC with only functional models.

The energy model work in [37] describes the utilization of core level techniques for heterogeneous platform modeling. They experimented with variable platform configurations with the same performance levels where the platform configuration was considered as the number of application parallel instances, number and type of cores, DVFS levels, and utilization rate. Their experiments were based on the ODROID XU4 platform which has two types of core clusters of ARM Cortex A7 and ARM Cortex A15, while four distinct voltage and frequency levels for both core types are involved. They implemented two energy models: one for big cores and the other for small cores and reported different energy and performance values in a table. Their results show better energy efficiency in terms of the utilization rate factor.

The present work focuses on improving modeling features and accuracy while the majority of the reviewed related research targets the optimization of some applications. Table 2.1 summarizes the workload reviews. The summary tables present the similarities, case study and the results with respect to this work. These studies are selected based on the goal of the study, the utilized methodology, experimentation approaches, design tools, and evaluation metrics. In addition to relevance, the publication date of the paper is considered to provide an up-to-date literature review as the majority of the reviewed papers have been published during the last four years. For example, the LSLA uses the same evaluation metric to show the suitability of the model and report a high fidelity value, as in this work. The table includes five columns. From left to right the first column shows the reference, the second describes the scope of the reviewed work, the third lists the similarity score to this work within a range of one to five where five is the highest and only used for the publications of this research, and the fourth and fifth columns show the purpose and the reported results.

Table 2.1 Summary of workload reviews

Ref.	Scope	Similarity	Purpose	Results
LSLA	Methodology and evaluation	4	Modeling C applications running on multi-core CPU	86% fidelity

[12]	Polynomial utilization	3	Parallel applications energy efficient mapping	50% energy saving
[47]	Mapping optimization	2	Dynamic energy optimization by mapping and frequency adjustment	206% energy saving
[11]	Workload scheduling	2	Energy optimization by type aware workload scheduling	31.3% energy saving
[20]	Graph presentation	2	Modeling signal processing systems	31.88% throughput 25% BMR
[2]	MoC of application	2	Resource allocation modeling	97.34% mem., 47.11% exe. reduction
[3]	Fidelity evaluation	3	Systems throughput estimation	Fidelity of minimum 65%
[42]	Task mapping	2	Hardware-software prototyping	21.6x performance increase
[36]	OpenCL mapping	2	System modeling with portable architecture	Design flow
[44]	Architecture modeling	2	Automatic algorithm mapping	Optimal mapping
[39]	Architecture design	2	Early stage architecture design and verification	MIPS-I 550.91 KIPS
[41]	Multi-core modeling	2	Modeling ARM's big.LITTLE architecture	System model
[7]	Multi-core scheduling	2	Runtime scheduling of multi-core architectures	Max deviation of 7.14% and 1.64%
[37]	Multi-core modeling	2	Modeling heterogeneous platform	Energy model
[38]	Multi-core modeling	2	Energy-efficient modeling without power sensors	33% energy saving

[46]	Fidelity evaluation	3	Modeling DNN applications	0.988 performance fidelity
This work [26] & [31]	Architecture modeling	5	Modeling OpenCL applications on GPU	93% performance and 84% power fidelity

2.2 Instruction Set Architecture extensions

The RISC-V ISA extension work in [16] depicts the inclusion of Bit Manipulation Instructions (BMI) into the RISC-V ISA. They explored system performance and power improvements through hardware optimization. They considered the instructions of the ARMv8 and x86 and used them as a reference for introducing ten BMIs to the RISC-V ISA. They mentioned 13.5% less bytes encoding requirement in comparison to x86. They implemented the introduced BMIs with Berkeley's Rocket CPU and reported no overhead on the critical path. Their experiments included 13 benchmarks where instructions were manually replaced for the GNU assembler and the results show a speed-up of between 1.09x and 96.87x.

The RISC-V verification environment study by [23] presents a framework for the early stage evaluation of RISC-V cores. The demonstrated framework has a reconfigurable, portable, and user-friendly design for the easy addition of user-defined ISA extensions. Their environment communicates with the DUT through the AXI4 lite protocol, which facilitates core switching. Their implementation, based on Universal Verification Methodology (UVM), utilizes the object-oriented facilities of the UVM to provide bottom-up stimuli generation. They used internal binded signals for designing their predictor model, which provides a reconfigurable architecture.

The multi-core generation work by [40] presents an automated solution for architecture customization. It is based on the RISC-V ISA and outputs the Verilog code of the architecture for an input application described as a dataflow. They used an open source rocket chip generator with added accelerators and a NoC. The utilized tools allow architecture configuration such as memory size or number of cores. They used the CAL actor language for the application development, which includes actors as operations. They experimented with two applications including auto focus criterion calculation and convolution. Their results show decreased design time, reduced hardware resources, and improved performance.

The customized core paper in [48] demonstrates the analysis of a taped-out implementation of an in-order, single-issue, 64-bit RISC-V ISA core named Ariane. It has Global Foundries 22-nm FDX technology and runs up to 1.7 GHz. The core covers multiplication or division, atomic memory operations, floating point, and compressed instructions. It has six pipeline stages and a branch predictor. Its cache memories are single-port static SRAM by INVECAS while the cache size of the instructions is 16kB and the cache size of the data is 32kB. Their implementation includes ISA extensions, reducing the number of instructions by a factor of 4.6 when compared to the baseline 32-bit RISC-V. Their evaluations with a 2-D convolution show that the number of instructions decreases to 110k with the DSP ISA extensions, whereas without the extension the number of instructions is 129k for 64-bit ISA and 135k for 32-bit ISA.

The near-threshold (NT) RISC-V core work by [9] shows the implementation of an architecturally optimized in-order core which supports ISA extensions. It covers compressed instructions and has a L0 buffer which causes less cache access. Their implementation has four pipeline stages and Harvard memory architecture. For power efficiency, they considered clock gating so that each unit has separate input operand registers, resulting in a 50% decrease in power dissipation. Their evaluations of sensor processing workloads demonstrate 3.5-fold performance gain and 3.2-fold energy efficiency. They implemented several ISA extensions such as post-increment instructions, hardware loop, vector instructions, and fixed point. They compare their architecture to baseline RISC-V and OpenRISC cores and reported 10-fold speedup as a result of the implemented ISA extensions.

The vector extension study by [5] describes a 64-bit RISC-V coprocessor called Ara coupled with a RV64GC core named Ariane. Their architecture has uniform lanes of functional units and vector register files communicating through a vector load/store unit (VLSU) and slide unit (SLDU) which makes it scalable. They experimented with AXPY, convolution, and matrix multiplication and reported 97% floating point unit (FPU) usage for 256x256 matrix multiplication. Ara is implemented with GlobalFoundries 22FDX FD-SOI technology and runs at more than 1 GHz in the typical corner. They compared Ara plus Ariane with solo Ariane and noted 2.5x energy efficiency for the Ara and Ariane pair. They observed two challenges, one with a scalability bottleneck introduced by VLSU and SLDU, and the other with performance reduction of smaller size problems caused by pairing Ara

plus Ariane with the vector instruction issue rate problem.

The RISC-V soft-core process work by [15] shows a scalable vector extension for the performance optimization of parallel applications. Their vector extension eliminates the vectorized code adaptation requirement upon adjusting the number of computing units for the SIMD soft-core. They targeted ARM SVE compatibility so they used ARM's automatic vectorization compiler. First, they compiled the program to ARM SVE assembly code. Second, they translated the assembly to a RISC-V assembly. Finally, they assembled the RISC-V assembly to RISC-V machine code using the edited GNU assembler. They used 32 vector registers and 16 predicate registers with eight of them for vector-mask control and the other eight for logical operation between them. Their vector instructions have three types including control, load/store, and vector operations. They performed a functional simulation of their implementation in a gem5 simulator.

The RISC-V ISA small float SIMD extension paper by [43] presents a 32-bit RISC-V core for scalar and floating point operations with a data type width of 8-bit and 16-bit. They implemented relevant C/C++ types and GNU compiler editions. Their edition covers the real type extension of the GNU compiler, which is an internal format for floating point types. They included call back functions to convert the internal format to the small floats and tuned the machine modes and rules as well as adding new keywords and conversion rules and adjusting the GCC auto-vectorizer. Their experiments with mixed precision computing shows no accuracy degradation and the results for 8-bit demonstrate execution time improvements of 2.18x and an energy reduction of 50%, whereas for 16-bit the execution time reduction is 1.64x and the energy decrease 30%.

ASSIST [19] is a synthesis framework for a RISC-V core which receives instruction specifications and provides multiple RISC-V processors register transfer level (RTL) implementations. The multi RTL outputs balance the quality-of-result (QoR) metrics with a facilitating custom instruction set and architectural feature analyses. Also, they describe automatic optimization for specific technology and performance. ASSIST has three main parts including an instruction specification interface, architectural synthesis engine, and pipeline schedule autotuner. The instruction interface uses an architecture description language (ADL) in Python for collecting the functional behavior of the instructions. The synthesis engine uses the DAL data and pipeline schedule for architecture generation as Chisel RTL, which is later sim-

ulated. The schedule autotuner receives the measured metrics and optimizes the pipeline schedule. They evaluated more than 60 variations of RTL implementations of in-order RISC-V 32I with different features such as pipeline structures. Their experiments were with machine learning and cryptograph applications and their 32-bit cryptographic ISA extension showed 9.3X faster task execution.

SMURF by [4] is a RISC-V high precision floating point accelerator used for Variable Precision (VP) calculations and covers up to 512 bits of mantissa. They present a RISC-V ISA extension and its design as a 64-bit word-size with pipelined architecture. SMURF is generated with Rocket chip as a RISC-V coprocessor and has two floating point (FP) formats for the internal computations and the main memory, while the format conversion is performed by a load and store unit. Their FPGA implementation has a 50 MHz clock frequency and their ASIC implementation with a 28 nm FDSOI has a 600 MHz clock frequency. They compared their design with a RISC-V 64-bit FP and reported 9 times more area and 12 times more power dissipation and stated that the essential internal memories and the mantissa buffers were the main reason.

The compressed RISC-V ISA work by [25] describes a 32-bit core with a coexisting 16-bit ISA extension which includes a low overhead dual-issue hardware support. Their architecture has six pipeline stages, while the execution stage has substages with variable latency for different instructions. For example, the addition instruction takes one clock cycle but division needs 16 cycles. They observed that RISC-V ISA mainly uses 8 registers with instruction compression which increases the hazards. Thus, they reduced stalls by selective register renaming for instruction compression registers. Also, they implemented a partitioned register file and clock gating. They experimented with a cycle-accurate simulator and analyzed the hardware, reporting similar performance with area and power efficiency in comparison to a super-scalar processor.

The multi precision RISC-V ISA floating point extension study by [17] presents an architecture for IEEE 754-2008 SP numbers which provides a trade-off between accuracy and precision. Their implementation covers eight-core Pulp architecture with their proposed shared unit for the floating point square root or division operation, which supports different precision in 5 to 8 clock cycles. They divide the architecture into three stages consisting of pre-processing, iteration, and post-processing. The pre-processing and post-processing each take one clock cycle while the iteration

stage includes four parallel iteration units which output one mantissa bit. They experimented with DIV/SQRT-intensive algorithms and concluded that there was a possibility for effectiveness and energy saving through precision reduction. Their report shows 36% energy saving and 43.65% performance improvement.

The RISC-V virtual prototype (VP) research by [10] presents an instruction set simulator (ISS) with software debugging, which facilitates functional verification and design space validations. Their design covers the IMAC instruction set with interrupt controllers and peripherals for 32-bit and 64-bit single and multi-cores. In addition, it is possible to add ISA extensions by modifying the decode and execute functions. Their VP design is based on systemC and it has about 12000 lines of C++ code. It is extendable with a generic bus system and TLM 2.0 communication, and covers the GCOV and GDB of the GNU tool chain. Their VP simulation is fast as it has two performance optimizations including a direct memory interface and local time quantums. They provide a HiFive1 board configuration example and experimented with embedded applications, reporting an average of 46 MIPS.

The resource sharing multi processor system on chip (MPSoC) study by [18] describes a RISC-V ISA extension where the instructions execute on shared coprocessing units outside of the processor's data path. They implemented a NoC-based MPSoC with 3x3 mesh topology, i.e., nine tiles where each tile contains a network interface (NI), a router, and a PE. In their design the center tile is the memory surrounded by four processors and four coprocessors. The coprocessors execute the multiplication and division instructions and are accessible by the processors as needed, where the processor uses the NoC for sending the operation packet to the coprocessor and receiving the result packet. They used four scenarios for area, power and performance evaluations and reported negligible hardware and execution time overheads to trade off area and leakage power.

XpulpNN [8] is a RISC-V ISA extension designed for quantized neural network (QNN) applications. It provides SIMD instructions for low-bit-width data types of nibble, i.e., 4-bit and crumb, i.e., 2-bit, which removes the instructions required for data packing and unpacking of QNN as well as an instruction for quantization. They implemented the extension in the RISC-V RI5CY where the quantization unit was added to the execution stage of the pipeline on the PULPissimo platform with 22nm FDX technology. Their report shows insignificant overhead with around 5-fold performance improvement for 4-bit and about 8-fold improvement for 2-bit in

comparison to 8-bit SIMD. They report 2-fold energy efficiency in comparison to commercial ARM architectures and up to 9-fold when compared to the baseline.

Table 2.2 summarizes the reviews. The RISC-V BMI study has the same goal of extending the RISC-V ISA and shows a high speed-up value like the high value obtained in this work. The table has five columns for referencing, stating the scope, depicting the similarity point, presenting the purpose, and the reported results of the reviewed work.

Table 2.2 Summary of RISC-V reviews

Ref.	Scope	Similarity	Purpose	Results
[16]	RISC-V BMI extension	4	RISC-V ISA customization	Up to 96.87x speedup
[23]	Non-standard extension simulation	2	Early stage RISC-V core evaluation	Development time reduction
[40]	Application specific ISA customization	2	Customized HDL generation	3 to 4 times speedup
[48]	RISC-V ISA extension analysis	2	Taped-out RISC-V core	Reduction by a factor of 4.6
[9]	RISC-V ISA extension investigation	2	Implemented RISC-V core	10 times speedup
[5]	RISC-V ISA extension implementation	2	Implemented RISC-V co-processor	2.5x energy efficiency
[15]	Architecture extension	2	Implemented RISC-V soft-core process	Programming method
[43]	RISC-V ISA extension	2	Implemented RISC-V SIMD extension	2.18x and 1.64x speedup
[19]	RISC-V ISA extension simulation	2	RISC-V core synthesis framework	Up to 9x speedup
[4]	RISC-V ISA extension implementation	2	Variable Precision RISC-V accelerator	12x power increase
[25]	RISC-V ISA extension	2	Dual-issue RISC-V core	Performance improvement

[17]	RISC-V ISA extension	2	Multi-precision RISC-V architecture	36% energy saving
[10]	RISC-V ISA simulation	2	RISC-V instruction set simulator	46 average MIPS
[18]	RISC-V ISA extension	2	RISC-V co-processing	Area and power reduction
[8]	RISC-V ISA extension for NN	2	RISC-V QNN customization	9x energy efficiency
This work [27] & [29]	RISC-V ISA extension	5	Extending PopCount instruction	2x performance improvement

2.3 Literature review comparison

The workload modeling studies include similar work such as LSLA [34] and parallel dataflow [12] with similar methodology. Also, the period graph study [3] has related evaluation metric as this work. On the other hand, the difference between the studies lies in the application parallelization, i.e., OpenCL or the platform selection, i.e., the GPU. The ISA extension studies have similarities in the selection of RISC-V ISA although they selected different types of instructions or have different implementations. In this work, the Instruction Set Simulator (ISS) is used where a GNU RISC-V compiler is edited to cover the PopCount instruction with no physical implementation.

3 METHODOLOGY

This work targets improvements from two different aspects: improving platform modeling by the means of workload modeling, as well as implementing an RISC-V ISA extension. The applications, tools, input data, platforms, and measurement methods as well as the techniques and platforms utilized are described for each of these aspects.

3.1 Workload modeling

Four representative OpenCL applications were considered including Matrix multiplication, predistortion, Gaussian filtering, and PopCount, which are executed with random data inputs. Two different GPUs which are known as Mali on Odroid XU3 [24] and AMD RX 460, i.e., Baffin [1] on a desktop computer were selected. Two different shell scripts were used for profiling the applications on each of the platforms. One of the shell scripts was used for collecting the execution time data only on the Baffin GPU while the other was used for collecting both the execution time and power dissipation data through a wire connection on the Odroid XU3 platform, as it has built-in power sensors. The collected profiling data was used for studying the workload optimization and proposing the performance and power models presented in this work with a Matlab script. For example, Figure 3.1 shows the stages of the proposed model's utilization on an unknown platform. In this figure, the OpenCL implementation, Shell script, and Matlab script usages are demonstrated in order from stage 1 to 3.

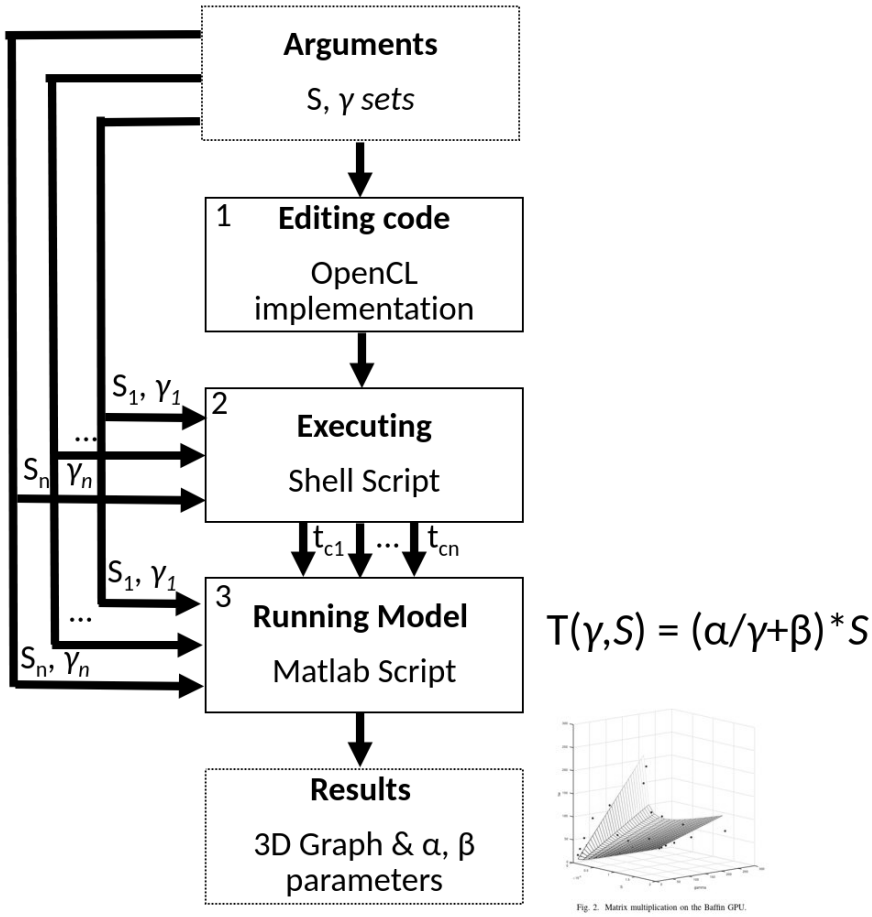

Figure 3.1 Workload modeling method.

S is the input data quantity, γ is the parallelism factor, α shows the reciprocal of the slope, and β depicts the intercept.

3.2 RISC-V ISA extension methods

The RISC-V ISA extension was studied with the PopCount [27] instruction addition to the open source GNU RISC-V compiler. One C implementation of a binarized neural network application known as the vehicle classifier [14] was selected, which is run with camera images of four vehicle types. The modified compiler was used for compiling the C code for the RISC-V ISA assembly that included the PopCount instruction. No commercial RISC-V processor with a hardware implementation of

the PopCount instruction was available thus the Extendable Translating Instruction Set Simulator (ETISS) by [22] was used for exploring the effect of the addition of PopCount, and the ETISS simulator was modified to cover the PopCount instruction.

3.2.1 ETISS simulator

ETISS has multiple ISA support, including RISC-V. It provides semi-automated instruction additions and has a straightforward procedure. Its successful execution results in a simulation and performance data display which could be edited to show more details such as the printing of all of the executed assembly instructions. In addition, as the targeted platform is PULPino [35], the relevant library files are used. The PULPino is an open source RISC-V microcontroller. It is reconfigurable for covering RISCY or the zero-riscy core. The ETISS simulator has the advantage of being open source while its codes can be edited to cover extra functionalities.

3.3 Use case applications

The selected applications for the experiments have wide usage areas such as signal processing, wireless communication, or machine learning. They have been selected by their use in other research studies. Verification of the use case is beyond the scope, but it is stated that they are sufficient for the methodology development purposes.

In this study four applications were selected: matrix multiplication, predistortion, Gaussian filtering, and the vehicle classifier. The matrix multiplication, predistortion, Gaussian filtering applications were used in the modeling studies as they are common embedded system applications. Also, the vehicle classifier was used in the ISA extension as it includes a population count instruction. It is a neural network image processing application which detects vehicle types according to pictures of them.

4 WORKLOAD MODELING OF HETEROGENEOUS PLATFORMS

The LSLA [34] concept covers multi-core CPU platforms while many current platforms also include a GPU. Such heterogeneous platforms require an extension to the LSLA in order to cover the GPU. The proposed model is named GSLA which adds a new parameter for parallelism and support to OpenCL applications. OpenCL applications use the CPU as the host for the GPU. Thus, a simple MoA was designed for the investigations. Figure 4.1 depicts the MoA, which consists of two processing elements called PE1 and GPU and one communication node named CN. The PE1 is the host device of the OpenCL applications, which communicates with the GPU through CN.

OpenCL applications include the kernel code which runs in the GPU and the main code in the CPU, including OpenCL API calls and I/O management. Figure 4.2 illustrates two computations named I/O and Kernel and presents their communication as a two-sided arrow. The execution of the application shown by MoC on the platform presented by MoA is depicted by mapping the MoC to the MoA where the dash lines show the mapped computations on the relevant processing element or communication node. The execution costs such as execution time or power dissipation are shown as tokens in the arcs on the dash lines. Figure 4.3 shows the mapping of the model of the computation on the model of the architecture.

The total execution cost is presented in Equation 4.1, which is the sum of the

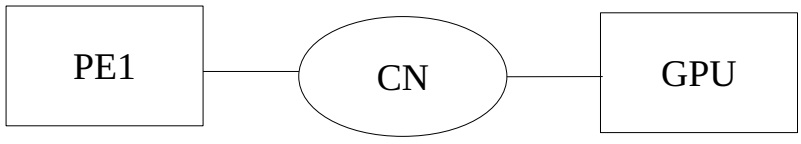

Figure 4.1 Model of Architecture.

Figure 4.2 Model of Computation.

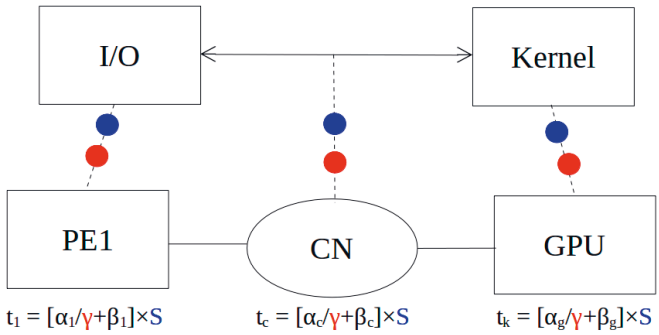

Figure 4.3 Mapping of MoC on MoA.

costs of the processing elements and the costs of the communication nodes. λ shows the scaling coefficient between the processing units and the communication cost. T_p depicts the collection of processing element tokens while T_c represents the collection of the communication node tokens. Also, t represents any token.

$$cost(A,P) = \sum_{t \in T_p} cost(t, map(t)) + \lambda \sum_{t \in T_c} cost(t, map(t)) \tag{4.1}$$

In this study, the execution costs of the OpenCL applications are examined and two models are presented: one for performance and the other for the power dissipation of the system.

4.1 Performance modeling

The execution time of the OpenCL applications can be modeled with GSLA [32] as a simple linear model as the sum of the costs of the three elements of the MoA, i.e., the cost functions of PE1, CN, and GPU. The total execution time is presented as (t_w) in Equation 4.2 as the sum of the execution costs of the kernel, i.e., (t_k) plus the execution cost of the processing elements, i.e., $(t\,1)$ plus the execution costs of the

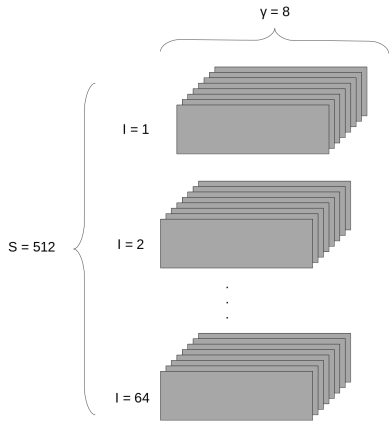

Figure 4.4 An example of the S and γ variables.

communication node, i.e., (t_c).

$$t_w = t_k + t_1 + t_c \tag{4.2}$$

The cost functions are presented in Equations 4.3, 4.4, and 4.5 where they all have two variables named S and γ and two parameters named α and β. Here, S is the input data quantity and γ is the parallelism factor. Figure 4.4 shows an example of S and γ variables with values of 512 and 8. Also, in the models the α shows the reciprocal of the slope, and β depicts the intercept. α and β are coefficients of the equations and they are calculated for each specific system.

$$t_k(\gamma, S) = (\alpha_g/\gamma + \beta_g) \times S \tag{4.3}$$

$$t_1(\gamma, S) = (\alpha_1/\gamma + \beta_1) \times S \tag{4.4}$$

$$t_c(\gamma, S) = (\alpha_c/\gamma + \beta_c) \times S \tag{4.5}$$

4.2 Power modeling

The power dissipation of the OpenCL applications are representable by our proposed power model, as shown in Equation 4.6. The total power dissipation of the

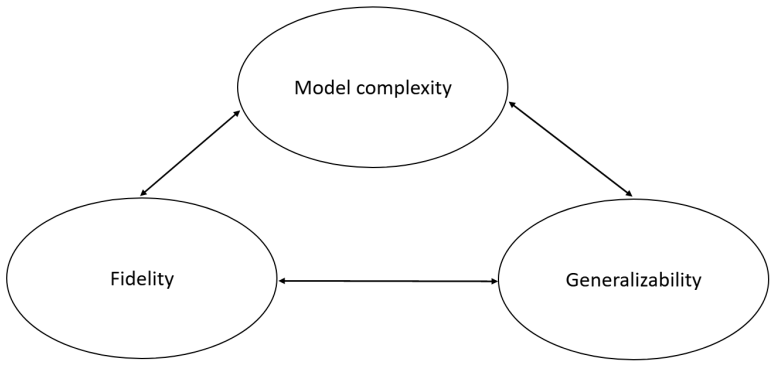

Figure 4.5 Trading off between different aspects of the modeling.

OpenCL application is denoted by P_t and has two variables named S and γ and three parameters named a_{gpu}, b_{gpu}, and c_{gpu}. a_{gpu}, b_{gpu}, and c_{gpu} are coefficients of the equations similar to the α and the β coefficients of the 4.3, 4.4, and 4.5 equations. In the experiments, the a_{gpu} parameter was a fixed constant value for all the tested applications, which suggests it is static power dissipation. On the other hand, when the a_{gpu} parameter is removed, i.e., having the same equation for both performance and power, the R-squared value decreases. The R-squared value is a metric for correlation calculation and here it is used for evaluating the proposed model. It is a value between 0 and 1 and shows a better correlation as its value gets closer to 1.

$$p_t(\gamma, S) = a_{gpu} + b_{gpu}S + c_{gpu}S/\gamma \tag{4.6}$$

4.3 Proposed models

This modeling approach has practical benefits. The simplicity of the model and its generalizability together with its evaluation with a fidelity metric provide a useful modeling approach. The concept is illustrated in Figure 4.5.

Equations 4.1-4.6 show the results of the workload modeling of this work. Here two workload models are proposed for the execution of OpenCL applications on GPUs. The selection criteria of these models is based on the complexity and fidelity trade-off, i.e., introducing powerful but simple models with the lowest possible complexity.

Figure 4.6 presents the development and the utilization of the models. The model

development section in blue shows the model creation phases step by step, whereas the model usages in green depict the two utilization approaches of the presented time and power models.

In model development, first the test data and a parallelizable application are selected, taking the data size into consideration. Also, a MoC for the application and a MoA for the platform are designed as schematics. Then, the OpenCL implementation of the parallel application is made. Later, the application is edited to receive the S and γ parameters as command line inputs that are affected by the input data size of the application, the parallelization capacity of the platform, and the relation of S to γ. The compiled application is executed and the correctness of the outputs is checked. Then, the application with considered S and γ values is run on the selected platform and the profiling data is collected. The data includes the execution time and power dissipation without any runtime error. Later, the Lsqlin function in Matlab is used to fit the average of the collected data points. After multiple use case modeling, GSLA models for time and power as well as curve fitting parameters are proposed.

In the model utilization for a known platform, first the input data is considered and the OpenCL implementation of a parallelizable application is carried out. Then, the application is edited to receive the S and the γ parameters as inputs. Later, the application is run on the known platform, thus profiling is not required and the same α and β parameters for the time equation and the same a, b, and c parameters for the power equation are used. The models show the estimated execution time and power dissipation values for the selected input data size and the parallelization factor.

In the model utilization for an unknown platform, first the OpenCL implementation of the selected parallelizable application is implemented, taking the input data size of the application and the parallelization factor into consideration. Then, the application is run on a reference platform and the profiling data is collected. The collected data about the execution time and the power dissipation is used in the Matlab script to provide the new platform specific parameters. Later, the equations with the calculated parameters are used to estimate the execution time and power dissipation values of the application. In addition, these parameters could be reused for similar applications or platforms. If only the MoC of the application is available, i.e., no physical platform or MoA to execute the code on, the previously collected data of similar platforms could provide initial guesses, e.g., expecting similar cost function

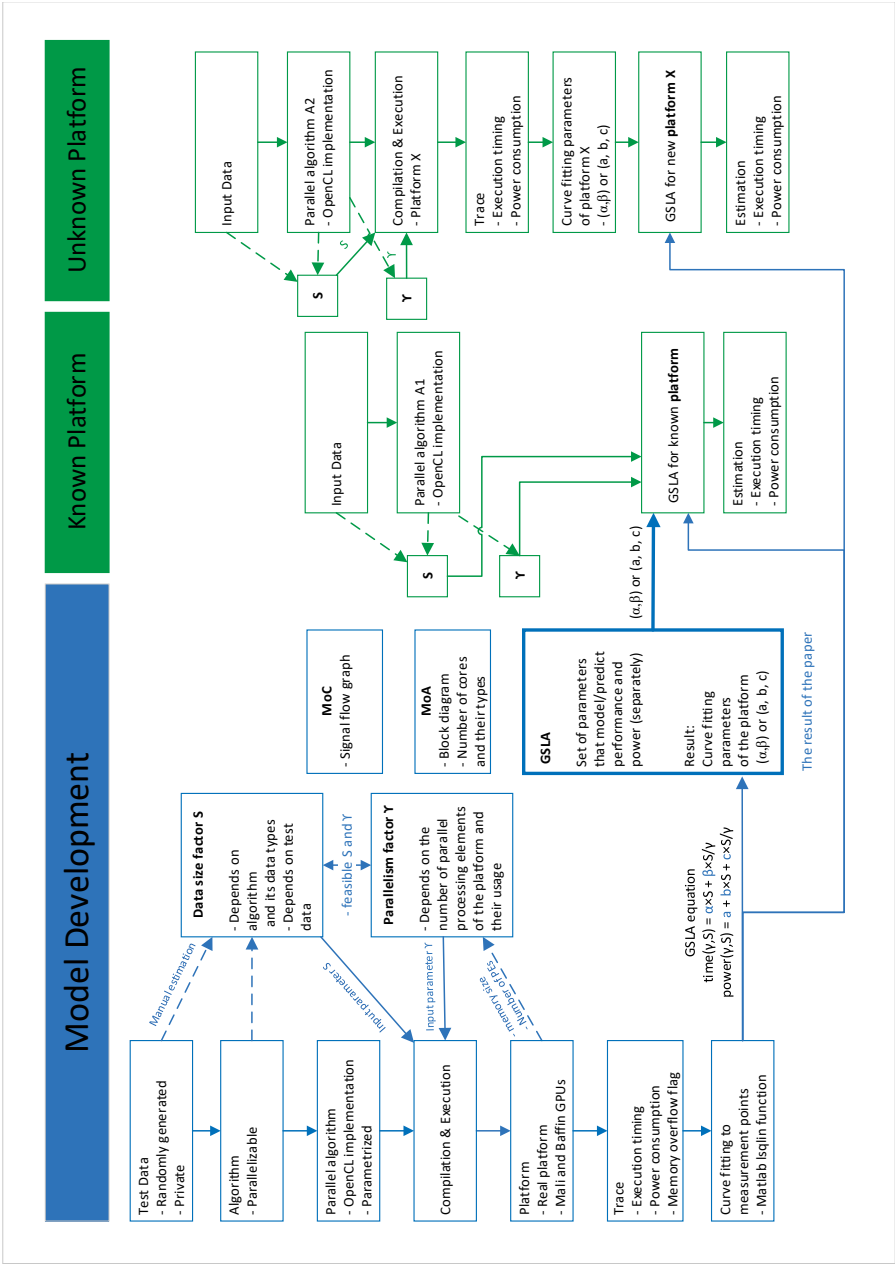

Figure 4.6 Development and utilization workflow.

parameters and similar graphs. In this case, the model utilization would be for the known platform but it would be an initial estimation rather than the actual usage of the proposed model, i.e., the S and the γ parameters and the arguments to calculate the execution time or power dissipation from the cost function formula are required.

4.4 Population count analysis

Further analysis of the proposed performance and power models is investigated from the aspect of neural network applications. The popularity of the neural network applications and suggested performance improvements in recent studies, e.g., [14] motivated the selection of the PopCount algorithm in this study. The PopCount instruction has high utilization in the binary convolution of neural network applications.

4.4.1 PopCount instruction

The PopCount algorithm receives a string of zeros and ones and outputs a number showing the number of ones in that string. There are multiple implementations for the PopCount calculation. Here three of them have been selected for further investigations with OpenCL kernel code. The selected algorithms include a native OpenCL function call 1, LLVM implementation 2, and the adder tree method 3. These kernel codes are disassembled by editing the 4th parameter of the *clBuildProgram* call in the OpenCL code which resulted in five files outputs. One of the files, i.e., *.isa* contains the assembly code equivalent of the kernel code.

Algorithm 1 OpenCL Kernel Native Instruction

```
__kernel void
PopC (
  __global int * C,
  __global int * A,
{
 C[get_global_id(0)] = popcount(A[get_global_id(0)]);
}
```

The experiments were performed on the AMD Baffin GPU and the disassembling

Algorithm 2 OpenCL Kernel LLVM Implementation [21]

```
__kernel void
PopC_2 (
  __global int * C,
  __global int * A,
{
  int x = A[get_global_id(0)];
  x = x - ((x >> 1) & 0x55555555);
  x = ((x >> 2) & 0x33333333) + (x & 0x33333333);
  x = (x + (x >> 4)) & 0x0f0f0f0f;
  x = (x + (x >> 16));
  C[get_global_id(0)] = (x + (x >> 8)) & 0x0000003F;
}
```

Algorithm 3 OpenCL Kernel Adder Tree Method

```
__kernel void
PopC_3 (
  __global int * C,
  __global int * A,
{
  int x = A[get_global_id(0)];
  ushortx_l = x;
  ushortx_h = (x >> 16);
  int res_l = popcount(x_l);
  int res_h = popcount(x_h);
  C[get_global_id(0)] = res_l + res_h
}
```

results showed 23 assembly instructions 4 for the native OpenCL function call, 36 assembly instructions 5 for the LLVM implementation, and 27 assembly instructions 6 for the adder tree method. All disassembly codes have the same first 16 and 2 last instructions which are grayed out for easier comparison.

The compiler converts the C code to the assembly according to the platform. For example, Figure 4.7 shows the 14 logical operations of the LLVM implementation of the PopCount algorithm and Algorithm 5 shows its equivalent assembly on an AMD Baffin GPU. Instruction number 18 in the algorithm is the first instruction, i.e., 1_SHIFT in the graph. The instructions are the same up to number 10 in the

graph, i.e., 10_ConstAND which is the number 27 in the assembly. Then, there is an extra *v_add_u32* instruction in the assembly. Later, it continues the same up to number 12 in the graph, i.e., 12_ADD which is followed by an extra *v_add_u32* instruction again. The last two instructions in the graph, i.e., 13_SHIFT and 14_ADD are converted into the three instructions of *v_mov_b32*, *v_addc_u32*, and *v_and_b32* in the assembly. Consequently, the 14 logical operations are converted into 18 assembly instructions for the real platform.

Algorithm 4 OpenCL Kernel Native Instruction Disassembly

```
{
1 : s_load_dword s0, s[4 : 5], 0x04
2 : s_waitcnt lgkmcnt(0)
3 : s_and_b32 s0, s0, 0x0000ffff
4 : s_mul_i32 s0, s0, s8
5 : v_add_u32 v0, vcc, s0, v0
6 : s_load_dwordx2 s[0 : 1], s[6 : 7], 0x00
7 : s_load_dwordx4 s[4 : 7], s[6 : 7], 0x30
8 : s_waitcnt lgkmcnt(0)
9 : v_add_u32 v0, vcc, v0, s0
10 : v_mov_b32 v1, s1
11 : v_addc_u32 v1, vcc, 0, v1, vcc
12 : v_lshlrev_b64 v[0 : 1], 2, v[0 : 1]
13 : v_add_u32 v2, vcc, s6, v0
14 : v_mov_b32 v3, s7
15 : v_addc_u32 v3, vcc, v3, v1, vcc
16 : flat_load_dword v2, v[2 : 3]
17 : v_add_u32 v0, vcc, s4, v0
18 : v_mov_b32 v3, s5
19 : v_addc_u32 v1, vcc, v3, v1, vcc
20 : s_waitcnt vmcnt(0) & lgkmcnt(0)
21 : v_bcnt_u32_b32 v2, v2, 0
22 : flat_store_dwordv[0 : 1], v2
23 : s_endpgm
}
```

Algorithm 5 OpenCL Kernel LLVM Implementation Disassembly

```
{
1 : s_load_dword s0, s[4 : 5], 0x04
2 : s_waitcnt lgkmcnt(0)
3 : s_and_b32 s0, s0, 0x0000ffff
4 : s_mul_i32 s0, s0, s8
5 : v_add_u32 v0, vcc, s0, v0
6 : s_load_dwordx2 s[0 : 1], s[6 : 7], 0x00
7 : s_load_dwordx4 s[4 : 7], s[6 : 7], 0x30
8 : s_waitcnt lgkmcnt(0)
9 : v_add_u32 v0, vcc, v0, s0
10 : v_mov_b32 v1, s1
11 : v_addc_u32 v1, vcc, 0, v1, vcc
12 : v_lshlrev_b64 v[0 : 1], 2, v[0 : 1]
13 : v_add_u32 v2, vcc, s6, v0
14 : v_mov_b32 v3, s7
15 : v_addc_u32 v3, vcc, v3, v1, vcc
16 : flat_load_dword v2, v[2 : 3]
17 : s_waitcnt vmcnt(0) & lgkmcnt(0)
18 : v_lshrrev_b32 v3, 1, v2
19 : v_and_b32 v3, 0x55555555, v3
20 : v_sub_u32 v2, vcc, v2, v3
21 : v_lshrrev_b32 v3, 2, v2
22 : v_and_b32 v2, 0x33333333, v2
23 : v_and_b32 v3, 0x33333333, v3
24 : v_add_u32 v2, vcc, v3, v2
25 : v_lshrrev_b32 v3, 4, v2
26 : v_add_u32 v2, vcc, v3, v2
27 : v_and_b32 v2, 0x0f0f0f0f, v2
28 : v_add_u32 v2, vcc, v2, v2 src0_sel : WORD_1
29 : v_lshrrev_b32 v3, 8, v2
30 : v_add_u32 v2, vcc, v3, v2
31 : v_add_u32 v0, vcc, s4, v0
32 : v_mov_b32 v3, s5
33 : v_addc_u32 v1, vcc, v3, v1, vcc
34 : v_and_b32 v2, 63, v2
35 : flat_store_dword v[0 : 1], v2
36 : s_endpgm
}
```

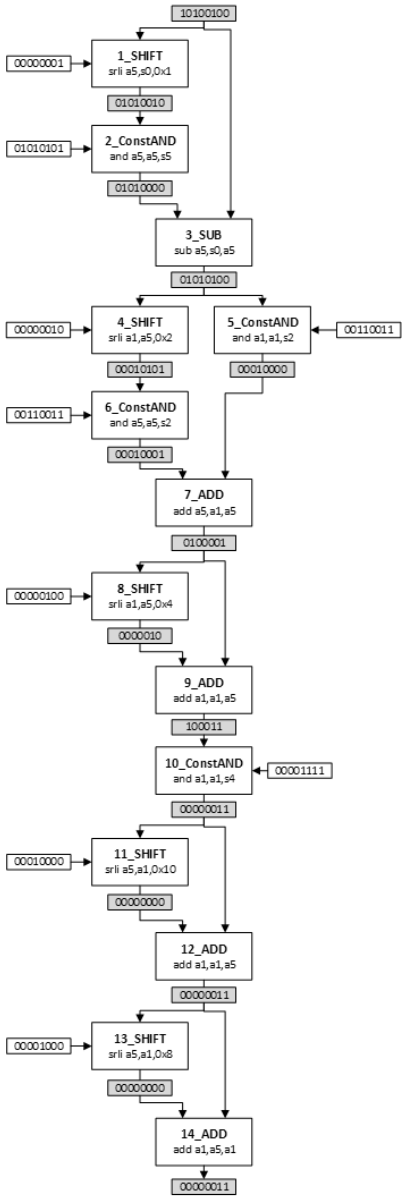

Figure 4.7 PopCount LLVM Logical Operations.

Algorithm 6 OpenCL Kernel Adder Tree Method Disassembly

```
{
1 : s_load_dword s0, s[4 : 5], 0x04
2 : s_waitcnt lgkmcnt(0)
3 : s_and_b32 s0, s0, 0x0000ffff
4 : s_mul_i32 s0, s0, s8
5 : v_add_u32 v0, vcc, s0, v0
6 : s_load_dwordx2 s[0 : 1], s[6 : 7], 0x00
7 : s_load_dwordx4 s[4 : 7], s[6 : 7], 0x30
8 : s_waitcnt lgkmcnt(0)
9 : v_add_u32 v0, vcc, v0, s0
10 : v_mov_b32 v1, s1
11 : v_addc_u32 v1, vcc, 0, v1, vcc
12 : v_lshlrev_b64 v[0 : 1], 2, v[0 : 1]
13 : v_add_u32 v2, vcc, s6, v0
14 : v_mov_b32 v3, s7
15 : v_addc_u32 v3, vcc, v3, v1, vcc
16 : flat_load_dword v2, v[2 : 3]
17 : s_waitcnt vmcnt(0) & lgkmcnt(0)
18 : v_bfe_u32 v3, v2, 0, 16
19 : v_lshrrev_b32 v2, 16, v2
20 : v_bcnt_u32_b32 v3, v3, 0
21 : v_bcnt_u32_b32 v2, v2, 0
22 : v_add_u32 v2, vcc, v2, v3 src0_sel : WORD_0src1_sel : WORD_0
23 : v_add_u32 v0, vcc, s4, v0
24 : v_mov_b32 v3, s5
25 : v_addc_u32 v1, vcc, v3, v1, vcc
26 : flat_store_dword v[0 : 1], v2
27 : s_endpgm
}
```

4.4.2 PopCount modeling

The *OpenCL* implementation of the PopCount algorithm is coded where the kernel receives an array of integer numbers and returns the PopCount results of each number. The kernel is designed as a two dimensional indexing space where the first dimension is a fixed value of 16, meaning that the kernel works on arrays which have 16 elements. The second dimension value is a variable presenting the γ,i.e., the

parallelization factor showing the number of vectors that the kernel performs the computations on. Also, S, i.e., work size is implemented as the iterations of a *for* loop calculated as S divided by γ. For example, when γ is 8 and S is 512, the kernel receives 8 arrays each with 16 values, which in total becomes 128 numbers when repeating the calculations 64 times. See Figure 4.4.

The shell scripts are used for benchmarking on the Baffin GPU and have nested loops with the program execute command with two command line arguments values for S and γ. The argument sets in the scripts have the same values as the previous experiments in order to provide comparable results. For benchmarking, the same S and γ sets were used for all applications. $S \in \{$ 512, 1024, 2048, 4096, 8192, 16384 $\}$ and $\gamma \in \{$ 8, 16, 32, 64, 128, 256$\}$. The measured values are read by the *Matlab* script of the proposed performance model which prints the parameters, the fidelity value, and the graph. See Figure 4.8. The blue dots in the figure are the average of 10 iterations for each relevant S and γ value.

Table 4.1 presents the PopCount parameters in addition to the algorithms of the three previous studies on the Baffin GPU. Table 4.2 shows the Kendall tau coefficient of the studied algorithms on the Baffin GPU. It is an association evaluation metric which shows the strength of the association with a value between -1 and 1. The zero value would show independence between the model and its measurements. As the value gets closer to 1 it shows better correlation, i.e., is more desirable. In these tables B1 is the matrix multiplication, B2 is the predistortion, B3 is the Gaussian filtering, and B4 is the PopCount algorithm.

Table 4.1 Parameters of functions.

Application	α_g	β_g	α_1	β_1	α_c	β_c
B1	0.001	0.008	0.000	0.004	0.005	0.068
B2	0.005	0.049	0.002	0.003	0.060	0.000
B3	0.000	0.051	0.002	0.000	0.004	1.074
B4	0.000	0.004	0.000	0.000	0.002	0.054

Table 4.2 Fidelity values.

Application	Fidelity GSLA	Fidelity LSLA
B1	0.88	0.75
B2	0.90	0.92
B3	0.91	0.62
B4	0.79	0.55

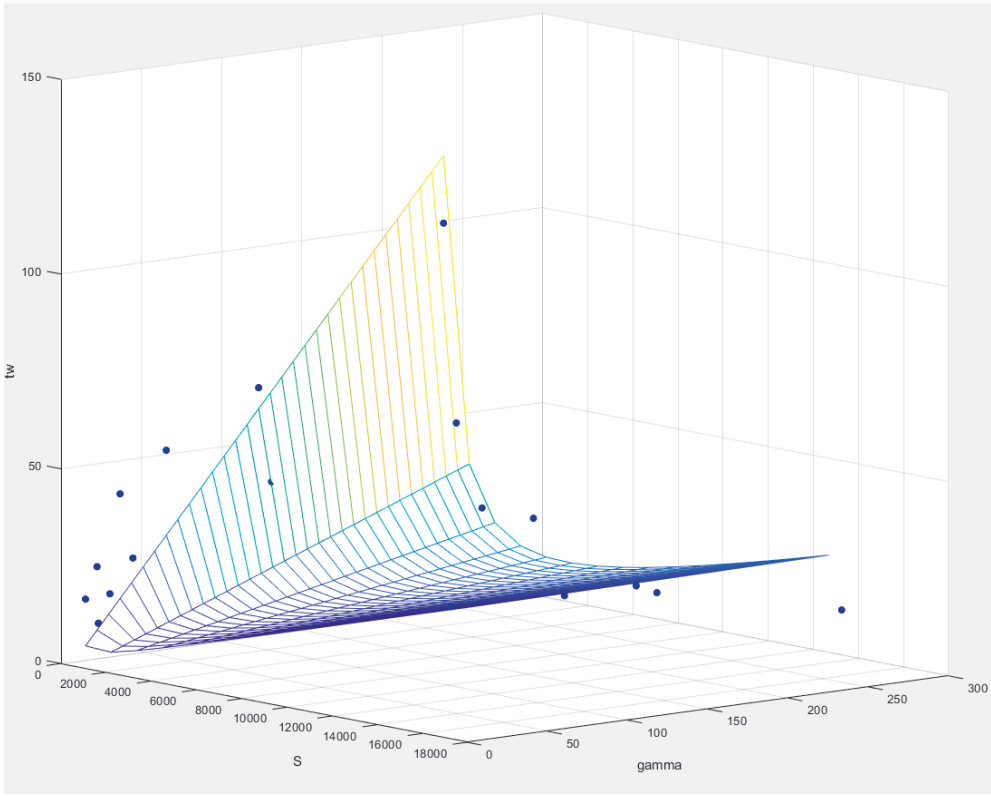

Figure 4.8 Population count graph.

γ is the parallelization factor, S is the work size, and tw is the time in milliseconds.

4.4.3 Modeling conclusion

In this study, four applications were considered in terms of performance and power dissipation. The proposed GSLA model was examined for its correctness and feasibility with the fidelity metric. The results of the experiments showed a value between 0.79 and 0.91 for performance modeling on the Baffin platform. Thus, these observations suggest that the utilization of the model is feasible. In case of parallelizable applications like matrix multiplication, predistortion, and Gaussian filtering, the fidelity results are good while in case of the worst case, i.e., PopCount, it shows an expected reduction.

The LSLA study investigated the system of C applications on a multi-core CPU platform while the present work considers OpenCL applications on the GPU. In the experiments the GPU unit provided multiple processing elements and the three algorithms of matrix multiplication, Gaussian filtering, and predistortion supply the parallel applications. During benchmarking, the execution time and the power dissipation were measured and the collected data was used for modeling. The linear regression method was used for studying the relations between the workload and the execution costs such as time and power, and the Kendall tau metric was used for selecting the appropriate model. The investigations resulted in two models: one for performance and the other for the power dissipation of the system.

The proposed GLSA modeling approach has a simple equation while providing a reasonable fidelity value. Also, the same equation of performance modeling with the addition of a constant value is used for power modeling. In addition, the cost functions of the LSLA are replaced with GSLA equations in order to cover the execution of OpenCL applications on GPU platforms.

The evaluation of the results with the Kendall tau metric for the workload modeling together with the linear regression method of the OpenCL applications on the GPU shows a fidelity value between 0.79 and 1.00 for performance, and a value between 0.74 and 0.94 for power dissipation on both the Baffin and Mali platforms.

5 RISC-V NON-STANDARD INSTRUCTION EXTENSIONS

The RISC-V ISA is a simple, widely adopted, and open source instruction set architecture. It has multiple standard extensions that add some extra instructions to the pipeline of the basic processing elements. These extra instructions reduce the total number of compiled assembly instructions as they replace the compiler's built-in function calls. The reduction of the assembly code improves performance, especially for algorithms that have a lot of data manipulation functions. Beside the standardized extensions, there are some promising non-standard extensions such as BMI, which are considered in this work. The focus of this work is on the PopCount instruction, as it has a high utilization rate in machine learning applications, suggesting performance improvement potential.

5.1 RISC-V BMI extension

One of the bit manipulation instruction extensions is the PopCount or hamming weight, which is not included in the standard RISC-V ISA extensions. The PopCount instruction calculates the number of ones, e.g., in a byte it returns how many of the eight digits are one. This instruction has considerably high utilization in the current trend applications in research such as CNNs, especially in binarization CNN algorithms implemented by packing binarized weights for suitable operations instead of executing many times per weight. The CNN applications show justifiable performance and accuracy for previously considered unjustifiable classifications applications on low power platforms like RISC-V. The energy efficiency aspect of the RISC-V platforms and the ability of CNN applications motivated the investigation of a system optimization, consisting of vehicle classification on PULPino, which is an open source RISC-V core. In addition, PopCount is not supported by any RISC-

V microcontroller on the market as it is a non-standard extension. Consequently, the RISC-V ETISS [22] with PULPino [45] support was considered in this study. Also, the RISC-V GCC compiler was selected as a mature compiler. Further, ETISS only supports the virtual prototype of the base RISC-V ISA, therefore modification was applied to ETISS in order to cover the PopCount instruction. This modification was implemented by studying the executed instructions and notation of unused instructions of the use case vehicle classification application [28]. Thus, the XORI instruction implementation on ETISS was modified to perform PopCount for XORI with a specific immediate value. The results showed an execution time reduction of more than double and 2 KB decrease in instruction memory footprint.

5.2 RISC-V extension design flow

Inclusion of the new instructions to the RISC-V ISA requires changes to the compiler, simulator, and hardware design tool. In this thesis, the compiler and simulator modifications are covered and a design flow for the RISC-V GCC compiler and ETISS simulator is presented. [30]. The first six levels of the proposed design flow named L0 to L5 show the compiler and simulator modification. Figure 5.1 shows the first three levels from initial specifications, i.e., the bit pattern of the new instruction and the standard RISC-V ISA extension data up to the outputs of L2, which are the object copy and GCC compiler tools. Figure 5.2 presents the output of L3 up to the output of L5, which is the simulation of the binary file and extraction of the execution time report from the simulator. The effort analysis of the proposed design flow showed around two hours of work effort for the inclusion of a new instruction in the RISC-V ISA. The idea is not to add a new unit to the CPU micro architecture but to use existing units by means of the new instructions.

5.2.1 Compiler modification

The addition of the new instruction to the GNU RISC-V compiler is presented in three steps. First, the bit pattern of the new instruction should be decided according to the RISC-V specification complying the instruction types. Also, the standard extension and the relevant flags for the RISC-V GCC compiler should be selected. See L0 in Figure 5.1. Second, a uniqueness check of the considered pattern for the

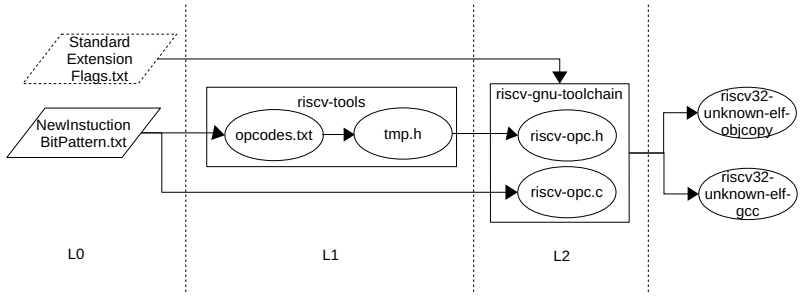

Figure 5.1 Compiler modification design flow.

new instruction should be performed, as each instruction should have its own unique and distinguishable pattern for the processor's decoder unit detection. This check could be done with some tools such as riscv tools where the opcodes text file with its own specific pattern is read and the equivalent header file for the GCC RISC-V compiler is supplied, see L1 in Figure 5.1. Third, the header file and the C files of the RISC-V opcodes are edited accordingly in the riscv-gnu-toolchain tool. Then, the standard flags from L0 are used to run the tool and provide the RISC-V compiler with the new instruction support, see L2 in Figure 5.1. The produced compiler is used to compile the application written in C for the RISC-V assembly.

5.2.2 Simulator modification

The architectural change to the ETISS simulator is implementable with the m2-isa-r tool, which is an Eclipse application [6]. First, this tool requires the algorithm of the new instruction and its bit pattern. The modification of the nML file results in a RISC-V architecture file in CPP language, see L4 in Figure 5.2. Second, the address of the compiled binary in L3 should be included in the ETISS initialization file and the new architecture files produced in L4 should be applied to the ETISS simulator. Then, running the ETISS simulator results in printing the outputs of the compiled application and ETISS reports, one of which is the CPU time, see L5 in Figure 5.2.

5.2.3 ISA extension conclusions

Extending the instruction set of the processing elements by supplying the hardware implementations reduces the number of clock cycles it takes to execute an unsup-

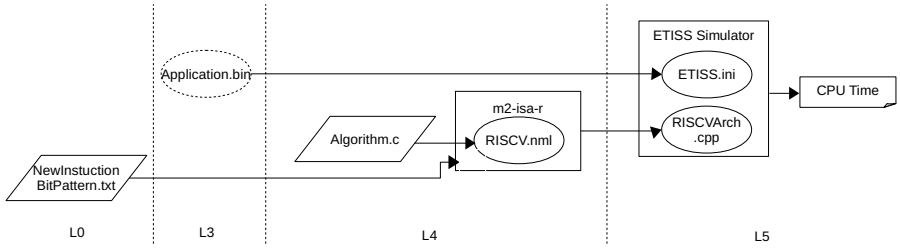

Figure 5.2 Simulator modification design flow.

ported instruction from multiple to usually one clock cycle. Consequently, the total number of assembly instructions is reduced, which improves the performance. Depending on the application and the frequency of the targeted instruction in the application's machine code a minimal hardware overhead could significantly improve the execution cost.

There are several techniques for the optimization of hardware implementations such as the inclusion of hardware accelerates. The RISC-V BMI study by [16] provides a RISC-V ISA extension which improves the performance of the system profiling. In addition, there are several evaluation techniques for effectiveness analyses such as simulating or prototyping. The ETISS [22] simulator shows the timing data and early design stage view of the targeted system.

In the study, the frequency of the PopCount instruction in the machine code of neural network applications motivated its further investigation. Consequently, the RISC-V GNU compiler was edited to cover the PopCount instruction in order to provide a comparison between the algorithm and the extended instruction. The compilations showed the replacement of 14 instructions with the expanded 1 instruction and a clear instruction memory reduction. For the timing examination, the simulation methodology with ETISS was used where the vehicle classifier application showed around double the improvement in performance.

The performance gain is around double and the effort required for editing the compiler and simulator for validation is reasonable. Also, the studied vehicle classifier application has a considerable number of PopCount instructions which suggests its RISC-V extension benefit.

The outcome analysis of the RISC-V ISA extension with the PopCount instruction for a neural network application known as a vehicle classifier using the ETISS simulator demonstrates more than 2x performance improvement.

6 CONCLUSIONS

This work addressed improvements on system modeling and execution of signal processing algorithms on RISC-V processors. Both aspects were verified using embedded platforms and representative signal processing applications.

For workload modeling, the Odroid XU3 platform featuring the ARM Mali T628 MP6 GPU and onboard sensors, in addition to a desktop machine including an AMD Baffin Radeon RX 460, were considered for four parallel algorithms with OpenCL API for matrix multiplication, Gaussian filtering, predistortion, and PopCount. A shell script supplied the commands for the broad connection, data transfer, code compilation, and execution while generating the workload size, i.e., S and parallelizing factors, i.e., γ parameters as the command line inputs for the applications. During the benchmarking, the power sensor and clock data was collected and was later sent as inputs to the Matlab script for modeling. The Matlab scripts read the collected power and time measurements for the relevant S and γ values and used a linear regressing technique for modeling the system and the Kendall tau metric for evaluations.

For the second part, RISC-V ISA was selected for the bit manipulation instruction (BMI) exploration of neural network applications. The GNU RISC-V compiler was edited to cover the PopCount instruction and the compilation of the inline assembly generated the relevant machine code. Comparison of the edited compiler's output with the PopCount algorithm shows a clear reduction of the number of instructions, i.e., 14 instructions. The ETISS simulator was used for the simulation of the vehicle classifier application on the Pulpino platform.

6.1 Addressing the research problem

The research problems on platform modeling and ISA extensions included five main questions. In the platform modeling section there were three issues. First, the in-

troduced GSLA provides a trade-off between simplicity and usefulness. Second, the same GSLA is shown to be usable for both performance and power modeling with the introduction of a constant as the static power dissipation of a system. Third, the GSLA expands the LSLA for single kernel OpenCL applications. The ISA extension research had two research questions. First, the PopCount shows a performance gain of around a double for vehicle classifier applications, whereas the addition of PopCount to an ISA requires a compiler and in our case, editing of the simulator code. Second, the PopCount instruction is frequently utilized in the convolution calculations of neural network applications, which means significant benefit in its hardware realization.

6.2 Future work

The original LSLA model covers multi-core CPUs, whereas the presented performance and power models consider the GPU. The workload study could be further investigated for GPU and CPU co-utilization where the workload is balanced between CPU and GPU simultaneously. In addition, the OpenCL support of the Preesm tool could facilitate model exploration and much faster statistical analysis.

The PopCount instruction extension of the RISC-V ISA was evaluated here with the ETISS simulator which describes the cycle timing data. Further examination could be made by including a special accelerator HW for the instruction and exploring the effect on FPGA prototyping. Neural network applications could be further explored for other instruction extensions.

REFERENCES

[1] *ARM Radeon RX 460*. URL: https://www.amd.com/en/products/graphics/radeon-rx-460 (visited on 06/03/2021).

[2] F. Arrestier, K. Desnos, E. Juarez and D. Menard. Numerical representation of directed acyclic graphs for efficient dataflow embedded resource allocation. *ACM Transactions on Embedded Computing Systems (TECS)* 18.5s (2019), 1–22.

[3] N. Bambha and S. S. Bhattacharyya. A joint power/performance optimization algorithm for multiprocessor systems using a period graph construct. *Proceedings 13th International Symposium on System Synthesis*. IEEE. 2000, 91–97.

[4] A. Bocco, Y. Durand and F. De Dinechin. SMURF: Scalar Multiple-precision Unum Risc-V Floating-point Accelerator for Scientific Computing. *Proceedings of the Conference for Next Generation Arithmetic 2019*. 2019, 1–8.

[5] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner and L. Benini. Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multi-precision Floating-Point Support in 22-nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2019).

[6] *Eclipse Application*. URL: https://www.eclipse.org/ (visited on 06/11/2021).

[7] P. Eitschberger, S. Holmbacka and J. Keller. Trade-Off Between Performance, Fault Tolerance and Energy Consumption in Duplication-Based Taskgraph Scheduling. *International Conference on Architecture of Computing Systems*. Springer. 2018, 3–17.

[8] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi and L. Benini. XpulpNN: accelerating quantized neural networks on RISC-V processors through ISA extensions. *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, 186–191.

[9] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak and L. Benini. Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), 2700–2713.

[10] V. Herdt, D. Große, P. Pieper and R. Drechsler. RISC-V based virtual prototype: An extensible and configurable platform for the system-level. *Journal of Systems Architecture* (2020), 101756.

[11] S. Holmbacka and J. Keller. Workload type-aware scheduling on big. LITTLE platforms. *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2017, 3–17.

[12] S. Holmbacka, E. Nogues, M. Pelcat, S. Lafond, D. Menard and J. Lilius. Energy-awareness and performance management with parallel dataflow applications. *Journal of Signal Processing Systems* 87.1 (2017), 33–48.

[13] H. Javaid, A. Ignjatovic and S. Parameswaran. Fidelity metrics for estimation models. *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2010, 1–8.

[14] M. Khan, H. Huttunen and J. Boutellier. Binarized Convolutional Neural Networks for Efficient Inference on GPUs. *2018 26th European Signal Processing Conference (EUSIPCO)*. IEEE. 2018, 682–686.

[15] Y. Kimura, T. Kikuchi, K. Ootsu and T. Yokota. Proposal of Scalable Vector Extension for Embedded RISC-V Soft-Core Processor. *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. 2019, 435–439.

[16] B. Koppelmann, P. Adelt, W. Mueller and C. Scheytt. RISC-V Extensions for Bit Manipulation Instructions. *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE. 2019, 41–48.

[17] L. Li, M. Gautschi and L. Benini. Approximate DIV and SQRT instructions for the RISC-V ISA: an efficiency vs. accuracy analysis. *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE. 2017, 1–8.

[18] P. Lima, C. Vieira, J. Reis, A. Almeida, J. Silveira, R. Goerl and C. Marcon. Optimizing RISC-V ISA Usage by Sharing Coprocessors on MPSoC. *2020 IEEE Latin-American Test Symposium (LATS)*. IEEE. 2020, 1–5.

[19] G. Liu, J. Primmer and Z. Zhang. Rapid Generation of High-Quality RISC-V Processors from Functional Instruction Set Specifications. *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2019, 1–6.

[20] Y. Liu, L. Barford and S. S. Bhattacharyya. Generalized graph connections for dataflow modeling of DSP applications. *2018 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE. 2018, 1–6.

[21] *LLVM PopCount*. URL: `https://github.com/sifive/riscv-llvm/blob/master/compiler-rt/lib/builtins/popcountsi2.c` (visited on 11/03/2022).

[22] D. Mueller-Gritschneder, M. Dittrich, M. Greim, K. Devarajegowda, W. Ecker and U. Schlichtmann. The extendable translating instruction set simulator (ETISS) interlinked with an MDA framework for fast RISC prototyping. *2017 International Symposium on Rapid System Prototyping (RSP)*. IEEE. 2017, 79–84.

[23] A. Munir, M. Magdy, S. Ahmed, S. Nasr, S. El-Ashry and A. Shalaby. Fast Reliable Verification Methodology for RISC-V Without a Reference Model. *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE. 2018, 12–17.

[24] *Odroid XU3 board*. URL: `https://www.hardkernel.com/shop/odroid-xu3/` (visited on 06/03/2021).

[25] K. Patsidis, D. Konstantinou, C. Nicopoulos and G. Dimitrakopoulos. A low-cost synthesizable RISC-V dual-issue processor core leveraging the compressed Instruction Set Extension. *Microprocessors and Microsystems* 61 (2018), 1–10.

[26] S. Payvar, J. Boutellier, A. Morvan, C. Rubattu and M. Pelcat. Extending Architecture Modeling for Signal Processing towards GPUs. *2019 27th European Signal Processing Conference (EUSIPCO)*. 2019, 1–5. DOI: `10.23919/EUSIPCO.2019.8903094`.

[27] S. Payvar, M. Khan, R. Stahl, D. Mueller-Gritschneder and J. Boutellier. Neural Network-based Vehicle Image Classification for IoT Devices. *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*. 2019, 148–153. DOI: 10.1109/SiPS47522.2019.9020464.

[28] S. Payvar, M. Khan, R. Stahl, D. Mueller-Gritschneder and J. Boutellier. Neural network-based vehicle image classification for iot devices. *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE. 2019, 148–153.

[29] S. Payvar, E. Pekkarinen, R. Stahl, D. Mueller-Gritschneder and T. D. Hämäläinen. Instruction Extension of a RISC-V Processor Modeled with IP-XACT. *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. 2019, 1–5. DOI: 10.1109/NORCHIP.2019.8906975.

[30] S. Payvar, E. Pekkarinen, R. Stahl, D. Mueller-Gritschneder and T. D. Hämäläinen. Instruction Extension of a RISC-V Processor Modeled with IP-XACT. *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*. IEEE. 2019, 1–5.

[31] S. Payvar, M. Pelcat and T. D. Hämäläinen. A model of architecture for estimating GPU processing performance and power. *Design Automation for Embedded Systems*. 2021, 43–63. DOI: 10.1007/s10617-020-09244-4.

[32] S. Payvar, M. Pelcat and T. D. Hämäläinen. A model of architecture for estimating GPU processing performance and power. *Design Automation for Embedded Systems* 25.1 (2021), 43–63.

[33] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan and S. Aridhi. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. *2014 6th european embedded design in education and research conference (EDERC)*. IEEE. 2014, 36–40.

[34] M. Pelcat, A. Mercat, K. Desnos, L. Maggiani, Y. Liu, J. Heulot, J.-F. Nezan, W. Hamidouche, D. Ménard and S. S. Bhattacharyya. Reproducible evaluation of system efficiency with a model of architecture: From theory to practice. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.10 (2017), 2050–2063.

[35] *PULPino*. URL: https://github.com/pulp-platform/pulpino (visited on 11/08/2022).

[36] O. Rafique and K. Schneider. Evaluating OpenCL as a Standard Hardware Abstraction for a Model-based Synthesis Framework: A Case Study. *MODELSWARD*. 2019, 386–393.

[37] H. Rexha, S. Holmbacka and S. Lafond. Core level utilization for achieving energy efficiency in heterogeneous systems. *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE. 2017, 401–407.

[38] H. Rexha and S. Lafond. Energy Efficiency Platform Characterization for Heterogeneous Multicore Architectures. *ICT4S*. 2019.

[39] S. Rigo, G. Araujo, M. Bartholomeu and R. Azevedo. ArchC: A SystemC-based architecture description language. *16th Symposium on Computer Architecture and High Performance Computing*. IEEE. 2004, 66–73.

[40] S. Savas, Z. Ul-Abdin and T. Nordström. A framework to generate domain-specific manycore architectures from dataflow programs. *Microprocessors and microsystems* 72 (2020), 102908.

[41] S. Stepanovic, G. Georgakarakos, S. Holmbacka and J. Lilius. An efficient model for quantifying the interaction between structural properties of software and hardware in the ARM big. LITTLE architecture. *Concurrency and Computation: Practice and Experience* 32.10 (2020), e5230.

[42] L. Suriano, F. Arrestier, A. Rodrguez, J. Heulot, K. Desnos, M. Pelcat and E. de la Torre. DAMHSE: programming heterogeneous MPSocS with hardware acceleration using dataflow-based design space exploration and automated rapid prototyping. *Microprocessors and Microsystems* 71 (2019), 102882.

[43] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu and L. Benini. Design and Evaluation of SmallFloat SIMD extensions to the RISC-V ISA. *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, 654–657.

[44] L. Thiele, I. Bacivarov, W. Haid and K. Huang. Mapping applications to tiled multiprocessor embedded systems. *Seventh International Conference on Application of Concurrency to System Design (ACSD 2007)*. IEEE. 2007, 29–40.

[45] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak and L. Benini. PULPino: A small single-core RISC-V SoC. *3rd RISCV Workshop*. 2016.

[46] M. Wess, M. Ivanov, C. Unger, A. Nookala, A. Wendt and A. Jantsch. Annette: Accurate neural network execution time estimation with stacked models. *IEEE Access* 9 (2020), 3545–3556.

[47] S. Yang, S. Le Nours, M. mendez Real and S. Pillement. Mapping and Frequency Joint Optimization for Energy Efficient Execution of Multiple Applications on Multicore Systems. *2019 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE. 2019, 29–34.

[48] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), 2629–2640.

PUBLICATIONS

PUBLICATION I

Extending Architecture Modeling for Signal Processing towards GPUs

S. Payvar, J. Boutellier, A. Morvan, C. Rubattu and M. Pelcat

*2019 27th European Signal Processing Conference (EUSIPCO)*2019, 1–5
DOI: 10.23919/EUSIPCO.2019.8903094

Extending Architecture Modeling for Signal Processing towards GPUs

Saman Payvar
Tampere University
Tampere, Finland
saman.payvar@tuni.fi

Jani Boutellier
Tampere University
Tampere, Finland
jani.boutellier@tuni.fi

Antoine Morvan
IETR/INSA Rennes
Rennes, France
antoine.morvan@insa-rennes.fr

Claudio Rubattu
IETR/INSA Rennes and UNISS
Rennes, France and Sassari, Italy
crubattu@uniss.it

Maxime Pelcat
IETR/INSA Rennes - Institut Pascal
Rennes and Clermont Ferrand, France
maxime.pelcat@insa-rennes.fr

***Abstract*—Efficient usage of heterogeneous computing architectures requires distribution of the workload to available processing elements. Traditionally, this mapping is done based on information acquired from application profiling. To reduce the high amount of manual work related to mapping, statistical application and architecture modeling can be applied for automating mapping exploration. Application modeling has been studied extensively, whereas architecture modeling has received less attention. Originally developed for signal processing systems, Linear System Level Architecture (LSLA) is the first architecture modeling approach that clearly distinguishes the underlying computation hardware from software. Up to now, LSLA has covered the modeling of multicore CPUs. This work proposes extending the LSLA model with GPU support, by including the notion of parallelism. The proposed GPU modeling extension is evaluated by performance estimation of three signal processing applications with various workload distributions on a desktop GPU, and a mobile GPU. The measured average fidelity of the proposed model is 93%.**

***Index Terms*—modeling, architecture, design space exploration, signal processing systems**

I. Introduction

Heterogeneous computing platforms that contain GPUs and DSPs alongside general-purpose processors, have become mainstream platforms for many signal processing applications, such as image, video and audio processing. One of the design decisions that should be made in the early stage of programming such systems is the mapping of the application to the platform i.e. workload consideration for processing elements. Unfortunately, the exploration of mapping alternatives is nowadays still mostly performed manually, which is a work-intensive and time consuming task. An approach that considerably reduces the effort is building models of the target platform and the application and exploiting them with automatic techniques or tools. With a suitable modeling approach combined with design space exploration, the efficiency of hundreds or thousands of mapping alternatives can be approximated within seconds.

In statistical system modeling, the application and the architecture are often considered together. Originally introduced for modeling of signal processing systems [1], Linear System Level Architecture (LSLA) [2], however, is the first Model of Architecture (MoA) that clearly separates the underlying architecture from the software running on top of it. LSLA specifically models the architecture and distinguishes the concepts of Model of Computation (MoC) from the MoA. The MoA and MoC separation reduces the modeling effort by formulating the system modeling as mapping of MoC activity to the MoA, so that the MoA and the MoC can be treated independently when needed. In LSLA it is possible to map different types of MoC to the LSLA, such as Synchronous Data Flow (SDF) [3] that is especially popular in signal processing.

In LSLA, an application described by a MoC is mapped to a processing architecture modeled by the LSLA MoA, and by considering the activity of the application, a cost function is computed for each processing element in the platform. For estimating the performance of various mapping alternatives, the cost functions of the processing elements are summed while varying the mapping parameters. In the original LSLA work, Pelcat et al. [2] have modeled the energy consumption of the Odroid XU3 platform with a graph. In this particular case, eight processing elements interconnected by three communication nodes model the asymmetric eight-core CPU of the Odroid platform. LSLA provides a model for parallel programing for CPU cores, while most contemporary platforms include also a GPU, which motivates the proposed work.

The contributions of this work are:

- An extension, called LSLAG, of LSLA is presented for covering GPU units. The proposed GPU extension to the model is linear, similar to the original LSLA that only covers CPU cores.
- For experimental evaluation of the proposed model, three applications are implemented in OpenCL and are executed on two different GPU-equipped platforms. Based on these experiments, the average fidelity of the model is 93%, which is similar to the original LSLA model

proposed for the CPU cores of the same platform.

This paper is structured as follows: Section 2 introduces related work and provides a comparison to the proposed work. Section 3 introduces the MoA concept. Section 4 explains the proposed LSLA extension. Section 5 explains the parameters. Section 6 elaborates the performed experiments. Finally, Section 7 explains the conclusions and outlines the future work.

II. Related Works

There are different methodologies in performance modeling studies. One of them, which provides a general model is the statistical analysis method. For example, Moren, et al. [4] present a statistical approach for work load scheduling on heterogeneous platforms consisting of CPU and GPU. They have modified the OpenCL API code for dynamic code feature collection which is used for performance prediction. In modeling methods, it is common to use a graph to present a software or a hardware system, or a system of systems. These methods are divided into two different categories: data flow graphs and non data flow graphs. In a data flow graph, a vertex is used to model a run-to-completion block of computation called an *actor*. *Edges* are used to model data token communications between actors, realized by FIFO queues (First In First Out). In addition, weights on FIFOs, called delays, are used to represent initial data present on edges. The execution of a data flow actor is called *firing* and is triggered when an actor has sufficient data on each input edge. Table I lists modeling approaches and the graph semantics used in related works.

SDF (Synchronous Data Flow) [3] is a well-known static MoC. In SDF, a system is modeled with a data flow graph where the firing rules specify the constant token consumption and production rates for all actors. These constant rates introduce limitations in terms of algorithmic behavior representation.

CFDF (Core Functional Data Flow) [5] is a form of EIDF (Enable Invoke Data Flow) [6] where a limited set of modes influence token consumptions and productions. CFDF limits mode transitions to only one alternative, making the model deterministic.

BSP (Bulk Synchronous Parallel) [7], unlike SDF or CFDF is a system modeling method, and it has its own graph implementation. In BSP, there are processing units with local memories connected over a router. Processing elements access each other's memories by remote access messages.

DAL (Distributed Application Layer) [8] has a dynamic mapping methodology. It employs Kahn process networks to explore application mappings and a finite state machine to represent execution scenarios. Multiple scenarios are precomputed at design-time and the suitable one is selected at runtime.

Bezati et al. [9] present a data flow modeling method according to the CAL language [10]. Their method has six steps. First, two different models for application and architecture are designed. Second, simulation and profiling results are collected. Third, the application is mapped to the architecture. Fourth, C++ and HDL codes are generated from CAL. Fifth, the code is compiled and synthesized. Finally, compiled code is executed.

LSLA [2] is a MoA, separate from the MoC. The LSLA MoA includes Processing Elements (PE) and Communication Nodes (CN). PEs and CNs of the LSLA MoA has cost functions including parameters that may be retrieved from representative platform benchmarking. In that case, the calculated cost functions are obtained from measured application executions and the cost function parameters can be used to predict system efficiency for a set of comparable applications.

The proposed work i.e. LSLAG provides a system modeling approach that as an extension to LSLA has the benefit of reduced modeling effort due to its re-usability. In addition, GPU coverage of LSLAG provides more complete coverage of modern heterogeneous platforms.

TABLE I
Modeling approaches

Method	Target	Graph
SDF	Application	Dataflow
CFDF	Application	Dataflow
BSP	System	Non Dataflow
DAL	System	Non Dataflow
[9]	System	Dataflow
LSLA	Architecture	Non Dataflow
LSLAG	Architecture	Non Dataflow

III. Models of Architecture

The MoA concept [2] is used to distinguish the processing architecture from the MoC, which should only address applications. A MoA is defined as a graph that can be used for reproducible execution cost (time, energy, etc.) calculations. A MoA is designed for each specific processing architecture and it covers the processing elements and their interconnect.

Each element in the MoA graph has a cost function whose parameters can be estimated statistically according to measurement results. The calculated parameter values depend on the application and the application's configuration.

A. Linear System Level Architecture

LSLA is a specific type of MoA that uses *linear* cost functions for each MoA graph element. The total cost of the modeled platform is calculated according to the Equation 1, which depicts the total cost of application activity A on the LSLA graph P. In this equation, the total cost is equal to the sum of the processing cost, and of the communication cost, λ being a scaling coefficient between processing and communication cost units. Tp depicts set of all mapped tokens to the processing elements and Tc shows set of all mapped tokens to the communication nodes. The activity of the mapped MoC is calculated as *tokens*, consisting of *quanta*, resulting in an affine cost model per communication and per processing. The quanta are an application-independent unit of execution cost.

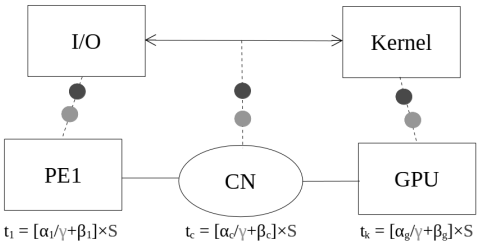

Fig. 1. Mapping of the application to the architecture model.

$$cost(A,P) = \sum_{t \in Tp} cost(t, map(t)) + \lambda \sum_{t \in Tc} cost(t, map(t)) \quad (1)$$

In LSLA, the application and its activity (i.e. the pressure it puts on hardware) are mapped as activity tokens to the LSLA model of the platform. Activity of the application includes *processing tokens* and *communication tokens*. These tokens are mapped to their associated elements in the platform model: processing tokens are mapped to processing elements and communication tokens are mapped to interconnection nodes that are used to transfer data between PEs.

IV. LSLAG: the Proposed Extension to LSLA

The proposed extension to LSLA of this work covers the GPU that can be present in a heterogeneous platform. Figure 1 shows an LSLAG graph that includes a CPU core *PE1*, the GPU, and the interconnection *CN* between the CPU core and the GPU. *PE1* is assumed to act as the host processor that communicates with the GPU. This simple LSLAG model has three elements including two processing elements and one communication node. Each element has its own cost function (presented beneath the nodes) that has two variables named γ and S, as well as two linear parameters α and β whose values are estimated for modeling purposes. The presented LSLAG is used to model the execution time of the platform, thus time samples are used in parameter calculations. As presented in Equation 1, the total cost is a sum of all cost elements, i.e. the execution time of the GPU (t_k), the execution time of the host processor (t_1) and the execution time of the interconnect (t_c). The hypothesis of Equation 2 is justified by consideration that (t_k), (t_1) and (t_c) do not overlap in time, i.e. the kernels of the GPU application are managed by the host device, then executed by the GPU at separate time intervals.

$$t_w = t_k + t_1 + t_c \quad (2)$$

$$t_k(\gamma, S) = (\alpha_g/\gamma + \beta_g) \times S \quad (3)$$

$$t_1(\gamma, S) = (\alpha_1/\gamma + \beta_1) \times S \quad (4)$$

$$t_c(\gamma, S) = (\alpha_c/\gamma + \beta_c) \times S \quad (5)$$

In these equations γ and S are variables, where γ is the parallelism factor, and S is the input data quantity. As it can be seen, increasing the parallelism factor γ decreases the total execution cost asymptotically. Conventional LSLA does not deal with parallelism, which limits its use to CPU cores. LSLAG adds the parallelism factor γ that enables including parallel processing elements. For each GPU-related MoA graph entity (i.e., the GPU itself, the host PE and the interconnect) there are separate parameters α and β, where α can be regarded as the reciprocal of slope, and β as intercept. In designing the model for the factors t_1, t_c, and t_k, simplicity was one of the driving motivations. To this end, t_1, t_c and t_k all have identical equations, and the model fitting will make the parameters settle to values that reflect the real trends of the factors. The next section describes the proposed approach of estimating each α and β.

V. Estimation of Parameters

Acquiring an accurate execution time model for an application running on a GPU requires reliable profiling data. The proposed estimation approach assumes three accurate factors that can be profiled on the platform

- Application total *wall-clock time* t_w,
- Host code execution time t_1, and
- GPU kernel execution time t_k.

The remaining factor t_c, in contrast, is derived using t_w, t_1 and t_k. The proposed procedure for acquiring accurate measurements for the factors are as follows: t_w is measured using the operating system clock, and t_k is read from the profiling data available from the GPU application programming interface. The measurement of t_1 is performed by modifying the application so that all GPU-related calls are disabled and the application only performs data I/O. Finally, t_c is derived from the other factors by subtracting t_1 and t_k from t_w.

VI. Experiments

The experiments presented below serve to illustrate the suitability of the proposed model and Equations 2-5 for real-life GPU-equipped platforms. Typical signal processing applications were used as case studies: matrix multiplication, digital predistortion and Gaussian image filtering. The applications were written in OpenCL and were executed on two GPU-equipped platforms: the Odroid XU3 containing a Mali T628 GPU and a desktop workstation with the AMD RX 460 (Baffin) GPU.

The α and β parameters of the cost functions were estimated with a Matlab script that invoked a least squares fitting algorithm (see Section 2 of [11]). In the Matlab script, the `lsqlin` function was used with a positive solution constraint.

Each application was profiled with two application variables, i.e., S and γ. Each variable had six values where S = { 512, 1024, 2048, 4096, 8192, 16384 } and γ = { 8, 16, 32, 64, 128, 256}. The global work size of OpenCL applications

was set application dependently. For matrix multiplication and predistortion, the work size set was calculated by 256*gamma, while for gaussian filtering, it is 1024*γ. The reason for this variation is in the input data types i.e. gaussian filtering reads 1-byte data, while matrix multiplication and predistortion read 4-byte data items. For each (γ, S) combination the execution time was measured 10 times, giving a total of 360 samples per application/architecture combination.

A. Application-architecture mapping

In OpenCL, when computations are performed on a GPU, the CPU works as the host device that reads data from I/O, sends it to the GPU for processing, receives the computed result and stores it back to I/O. Based on the dataflow [3] MoC, a generic model for OpenCL applications was created. Data reading and writing of the CPU is mapped to an *I/O* node (see Figure 1). The *Kernel* node represents the computations performed on the GPU, whereas the communication between the *I/O* and *Kernel* nodes is presented with a bidirectional arrow in Figure 1.

In each actor firing of the application graph, actors and the communication FIFO provide a token, which is mapped to their associated PE or CN node of the model. In other words, the tokens of the node *I/O* are mapped to the *PE1* architecture node, the tokens of *Kernel* are mapped to the *GPU* architecture node, and the communication FIFO tokens to the *CN* node. The cost functions shown below the architecture nodes have two variables, thus two tokens on the mapping lines in Figure 1 are used to present the number of quanta for each variable.

B. Results and Discussion

This section shows how the proposed GPU execution time model fits with the measured execution time samples. In Figure 2, Figure 3, and Figure 4 the bottom axes depict the variables S and γ, whereas the vertical axis depicts execution time. The dots represent the average of individual measured execution time samples. *The measured execution time samples are t_w (wall-clock time) values, and the mesh depicts the model-based sum of $t_k + t_1 + t_c$.* For clarity, the measured time samples depict the average of the 10 measurements for each (γ, S) coordinate.

Table II depicts the calculated α and β parameter values for each application on Baffin and Mali GPUs. These parameters are used in the Equation 2 for calculating t_w. The α value represents the cost of a token and equals to the slope of the mesh. The β value represents the constant time offset of the relevant LSALG element and is the *tw* intercept of the mesh graph. Due to technical difficulties, values for the digital predistortion application were not acquired on the Mali platform. App 1 is matrix multiplication, App 2 is digital predistortion and App 3 is Gaussian filtering. M stands for Mali and B for Baffin platforms.

Table III demonstrates the fitting error between the model and measured samples for each application/platform combination as *fidelity* values. To highlight the improvement of the proposed LSLAG model over conventional LSLA for GPU targets, Table III also shows the fidelity value for LSLA. Fidelity is Kendall's Tau Coefficient value calculated by the `corr` function of Matlab. For computing the fidelity, the 360 execution time samples were randomly divided into a training set of 288 samples, and a test set of 72 samples. Fidelity calculations are performed similarly to [12] and present a value between 0 and 1 where 1 would represent a perfect match between model and samples.

TABLE II
CALCULATED COST FUNCTION PARAMETERS

App.	α_g	β_g	α_1	β_1	α_c	β_c
B1	0.001	0.008	0.000	0.004	0.005	0.068
M1	0.003	0.009	0.000	0.009	0.016	1.554
B2	0.005	0.049	0.002	0.003	0.060	0.000
B3	0.000	0.051	0.002	0.000	0.004	1.074
M3	0.003	0.023	0.022	0.000	0.082	0.460

TABLE III
FIDELITY OF THE TEST SETS ON EXECUTION PLATFORMS.

Application	Platform	Fidelity LSLAG (proposed)	Fidelity LSLA
1	Baffin	0.88	0.75
1	Mali	1.00	0.70
2	Baffin	0.90	0.92
3	Baffin	0.91	0.62
3	Mali	1.00	0.92

In the Table III results, it can be seen that conventional LSLA yields considerably worse fidelity than the proposed LSLAG for GPU architectures. The reason for this is evident: LSLA does not capture parallelism (γ), which is an integral part of GPU processing. An exception to this is the predistortion application on the Baffin GPU, where LSLA and LSLAG yield almost identical fidelity. The reason for this is that on this platform, communication time dominates over parallelized kernel execution, making the whole application behave almost similar to a sequential application. Dominance of communication can be seen in Table II as the high value of coefficient α_c for application B2.

The measured fidelity values also show that the proposed linear LSLAG model fits better the Mali platform than the Baffin platform. The difference is likely related to the different memory architectures; Mali uses a shared memory between the CPU and the GPU, whereas the Baffin GPU is connected over PCI Express.

VII. CONCLUSION AND FUTURE WORK

In this work, LSLAG which is a GPU extension to the LSLA model, was proposed. The proposed model is linear, like the original LSLA model that is intended for multicore CPU platforms. The validity of the proposed model was evaluated by profiling three OpenCL applications on two GPU-equipped platforms, and the achieved model fidelity was 93% for the considered set of signal processing use cases.

Similar to the LSLA model, LSLAG can be used for design space exploration and performance prediction of signal

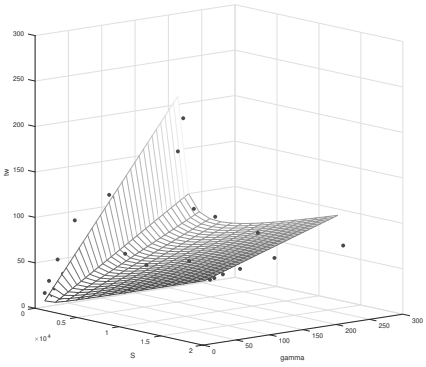

Fig. 2. Matrix multiplication on the Baffin GPU.

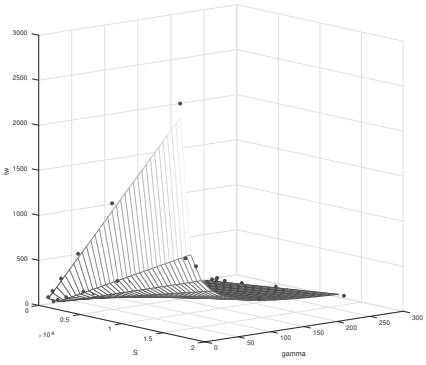

Fig. 3. Gaussian filtering on the Baffin GPU.

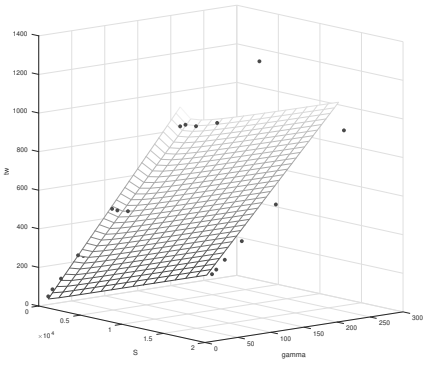

Fig. 4. Predistortion on the Baffin GPU.

processing systems, with the difference that the proposed model also covers GPU-equipped architectures.

Future work involves extending the modeling approach to cover energy measurements, and connecting the GPU extension to larger application graphs.

VIII. Acknowledgment

This work is partially supported by the ITEA3 project 16018 COMPACT and by the European Union Horizon 2020 research grant 732105 CERBERO.

References

[1] Maxime Pelcat, Karol Desnos, Luca Maggiani, Yanzhou Liu, Julien Heulot, Jean-François Nezan, and Shuvra S. Bhattacharyya, "Models of architecture: Reproducible efficiency evaluation for signal processing systems," in *IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2016, pp. 121–126.

[2] Maxime Pelcat, Alexandre Mercat, Karol Desnos, Luca Maggiani, Yanzhou Liu, Julien Heulot, Jean-François Nezan, Wassim Hamidouche, Daniel Ménard, and Shuvra S. Bhattacharyya, "Reproducible evaluation of system efficiency with a model of architecture: From theory to practice," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 10, pp. 2050–2063, 2018.

[3] Edward A. Lee and David G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[4] Konrad Moren and Diana Göhringer, "Automatic mapping for OpenCL-programs on CPU/GPU heterogeneous platforms," in *International Conference on Computational Science*. Springer, 2018, pp. 301–314.

[5] William Plishker, Nimish Sane, Mary Kiemb, Kapil Anand, and Shuvra S. Bhattacharyya, "Functional DIF for rapid prototyping," in *IEEE/IFIP International Symposium on Rapid System Prototyping (RSP)*. IEEE, 2008, pp. 17–23.

[6] William Plishker, Nimish Sane, Mary Kiemb, and Shuvra S. Bhattacharyya, "Heterogeneous design in functional DIF," in *International Workshop on Embedded Computer Systems*. Springer, 2008, pp. 157–166.

[7] Leslie G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[8] Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin-Haeng Kang, and Lothar Thiele, "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," in *International conference on compilers, architectures and synthesis for embedded systems*. ACM, 2012, pp. 71–80.

[9] Endri Bezati, Richard Thavot, Ghislain Roquier, and Marco Mattavelli, "High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms," *Journal of real-time image processing*, vol. 9, no. 1, pp. 251–262, 2014.

[10] Johan Eker and Jorn Janneck, "Cal language report," Tech. Rep., Tech. Rep. ERL Technical Memo UCB/ERL, 2003.

[11] Richard C. Aster, Brian Borchers, and Clifford H. Thurber, *Parameter estimation and inverse problems*, Elsevier, 2018.

[12] Neal K. Bambha and Shuvra S. Bhattacharyya, "A joint power/performance optimization algorithm for multiprocessor systems using a period graph construct," in *International symposium on system synthesis*. IEEE Computer Society, 2000, pp. 91–97.

PUBLICATION

II

A model of architecture for estimating GPU processing performance and power

S. Payvar, M. Pelcat and T. D. Hämäläinen

*Design Automation for Embedded Systems*2021, 43–63
DOI: 10.1007/s10617-020-09244-4

Noname manuscript No.
(will be inserted by the editor)

A Model of Architecture for estimating GPU processing performance and power

Saman Payvar · Maxime Pelcat · Timo D. Hämäläinen

Received: date / Accepted: date

Abstract Efficient usage of heterogeneous computing architectures requires distribution of the workload on available processing elements. Traditionally, the mapping is based on information acquired from application profiling and utilized in architecture exploration. To reduce the amount of manual work required, statistical application modeling and architecture modeling can be combined with exploration heuristics. While the application modeling side of the problem has been studied extensively, architecture modeling has received less attention. Linear System Level Architecture (LSLA) is a Model of Architecture (MoA) that aims at separating the architectural concerns from algorithmic ones when predicting performance. This work builds on the LSLA model and introduces non-linear semantics, specifically to support GPU performance and power modeling, by modeling also the degree of parallelism. The model is evaluated with three signal processing applications with various workload distributions on a desktop GPU and mobile GPU. The measured average fidelity of the new model is 93% for performance, and 84% for power, which can fit design space exploration purposes.

Keywords Modeling · Model of Architecture · Design space exploration · Signal processing systems

F. Author
first address
Tel.: +123-45-678910
Fax: +123-45-678910
E-mail: fauthor@example.com

S. Author
second address

1 Introduction

Heterogeneous platforms that contain GPUs and DSPs alongside general-purpose processors have become the mainstream for many signal processing applications, such as image, video and audio processing. One of the design decisions that should be made in the early stage is mapping of the application to the platform i.e. resource allocation for processing elements. Unfortunately, the exploration of mapping alternatives is still mostly performed case by case, which is a work-intensive and time consuming task. An approach that considerably reduces the effort is building models of the target platform and the application, and exploiting them with automatic tools. There are different approaches to the system modeling and workload mapping. For example, the Distributed Operation Layer (DOL)[21] is a framework for automatically optimizing parallel algorithm mapping on heterogeneous platforms. ArchC[19] is an architecture description language (ADL) for architecture design which provides early stage system verification. In contrast, rather than jointly designing optimization methods and architecture representations, this paper concentrates on learning a model from structure hypotheses and platform measurements, with the objective to obtain an abstract, repeatable and application decorrelated model usable in a wide set of optimization contexts.

In statistical system modeling, the application and the architecture are often considered together. Originally introduced for modeling of signal processing systems [14], the Linear System Level Architecture (LSLA) [15], Model of Architecture (MoA) separates the underlying architecture from the algorithmic aspects following a Y-chart approach [8]. LSLA specifically models the architecture and distinguishes the concepts of Model of Computation (MoC) from the MoA. The MoA and MoC separation reduces the modeling effort by formulating the system modeling as mapping of MoC activity to the MoA, so that the MoA and the MoC can be treated independently when needed. In LSLA it is possible to map different types of MoC to the LSLA, such as Synchronous Data Flow (SDF) [9] that is popular in signal processing.

In LSLA, an application described by a MoC is mapped to the architecture modeled by the LSLA MoA. Considering the activity of the application, a cost function is computed for each processing element in the platform. For estimating the performance of various mapping alternatives, the cost functions of the processing elements are summed up while varying the mapping parameters. For example, the energy consumption of the Odroid XU3 platform was modeled in [15]. In this particular case, eight processing elements interconnected by three communication nodes model the asymmetric eight-core CPU of the platform. The LSLA experiments model the energy consumption of a Stereo Matching application that computes a depth map from a pair of views of a single scene, while the GSLA experiments in this paper cover the execution time and the power consumption costs of matrix multiplication, digital predistortion and Gaussian filtering applications.

LSLA provides a model for linear Key Performance Indicators (KPIs). However, most contemporary platforms include a GPU, in which the perfor-

mance with respect to the application activity is non-linear. This is the key motivation to extend the LSLA model. Our initial work was presented in [13] with a GPU performance model. Consequently, in this work we introduce power modeling and collect all the results for an LSLA model extended to GPU modeling.

The key contributions of this work are:

- An extension of the LSLA model with non-linear GPU processing modeling called GSLA (GPU-oriented System-Level Architecture). The model includes both performance and power.
- Prototype tooling implemented as shell scripts and Matlab code for both execution of the application for model creation and costs prediction in exploration use.
- Proof-of-Concept with three representative applications that are implemented in OpenCL and executed on two different GPU-equipped platforms for setting the model parameters and comparing the measured and predicted values for fidelity.

Experimental results show that the proposed GSLA model can help predicting with low complexity the performance and the power consumption of an application with varying parameters. On the other hand, modeling the key performance indicators of strongly differing applications or platform configurations is shown to require model parameter recalculation. Even in cases where parameters cannot be reused, experimental results show that the model lightweight structure can be kept, and retrained through a lightweight procedure and with good accuracy. Two different utilization examples of GSLA are discussed in Section 2 to clarify the intended usage of the MoA.

This paper is structured as follows: Section 2 introduces related work and provides a comparison to the proposed work. Section 3 introduces the MoA concept. Section 4 explains the proposed LSLA extension. Section 5 explains the parameters. Section 6 elaborates the performed experiments. Section 7 presents the novel power model for GPUs. Section 8 compares the fitness of the models. Finally, Section 9 explains the conclusions.

2 Model Creation and Potential Usage

The design and usage of the proposed performance and power models in practice are represented in Figure 1. The blue stages depict the steps of the model construction while the green steps show two examples of using the models. Here the dashed lines demonstrate the manual estimations while the continuous lines present the flow of the work.

Model development starts by selecting the test data and a target parallelizable algorithm i.e. matrix multiplication, Gaussian filtering and predistortion for execution on Mali and Baffin GPUs. Then, the algorithms are coded with OpenCL function calls and receive two command line argument inputs named S and γ which impact the size of the input data and the number of their parallel executions, offering variations in their structure. Later, two value sets for

these parameters are defined where the S values set is inferred from the test data and the algorithm input data size and the γ values set is extracted from the platforms number of processing elements and memory sizes. In addition, the values for both S and γ sets were checked for feasibility e.g. execution of the OpenCL implementation of the parallel algorithm with a large S and a small γ values could take up to some days while multiple S and γ combinations are executed for profiling which makes it impractical. Shell scripts are used for passing the values of the S and γ parameters as command line arguments for each iteration of the execution and storing the execution time and power consumption values in files while the memory overflow errors are monitored. Linear regression is used to fit the GSLA model. The MoC and MoA for OpenCL application on GPU is then developed for showing the application mapping on the platform. The results of this paper show the suitable parameters of the models equations for Mali and Baffin platforms i.e. α,β or a, b, c. These values in conjunction with the model equations could be used to estimate the power and performance of similar applications on Mali and Baffin GPUs.

The produced models could be used with or without the presented parameters which are shown as two utilization cases in Figure 1. In the first case, a similar algorithm to the selected three algorithms is executed on Mali or Baffin platforms while in the second case a different algorithm is executed on a platform X. The first case does not require any compilation or execution of a training code and the performance and the power estimations are done merely by simple equation calculations using the reported parameters values. The S and γ values of an algorithm implemented in OpenCL are computed by checking the code files for the global work size and number of kernel calls which is usually defined as constant values in a header file and considering the input data of the application. In the second case where the algorithm or the platform vary much from the presented training, values of parameters shall be determined by the execution of a representative algorithm on the platform and measuring the execution time or power consumption and using the presented models equations for the parameters calculations. Later, the calculated parameters could be reused for performance and power estimations of similar cases. These equations reduces the manual work of a researcher using the OpenCL for parallelizing where the appropriate values are typically determined by trial and error i.e. editing the code and benchmarking the application.

3 Related Work

There are different methodologies in performance modeling studies. One of them, which provides a general model is the statistical analysis method. For example, Moren, et al. [11] present a statistical approach for work load scheduling on heterogeneous platforms consisting of CPU and GPU. Authors have modified the OpenCL API code for dynamic code feature collection which is used for performance prediction. In modeling methods, it is common to use

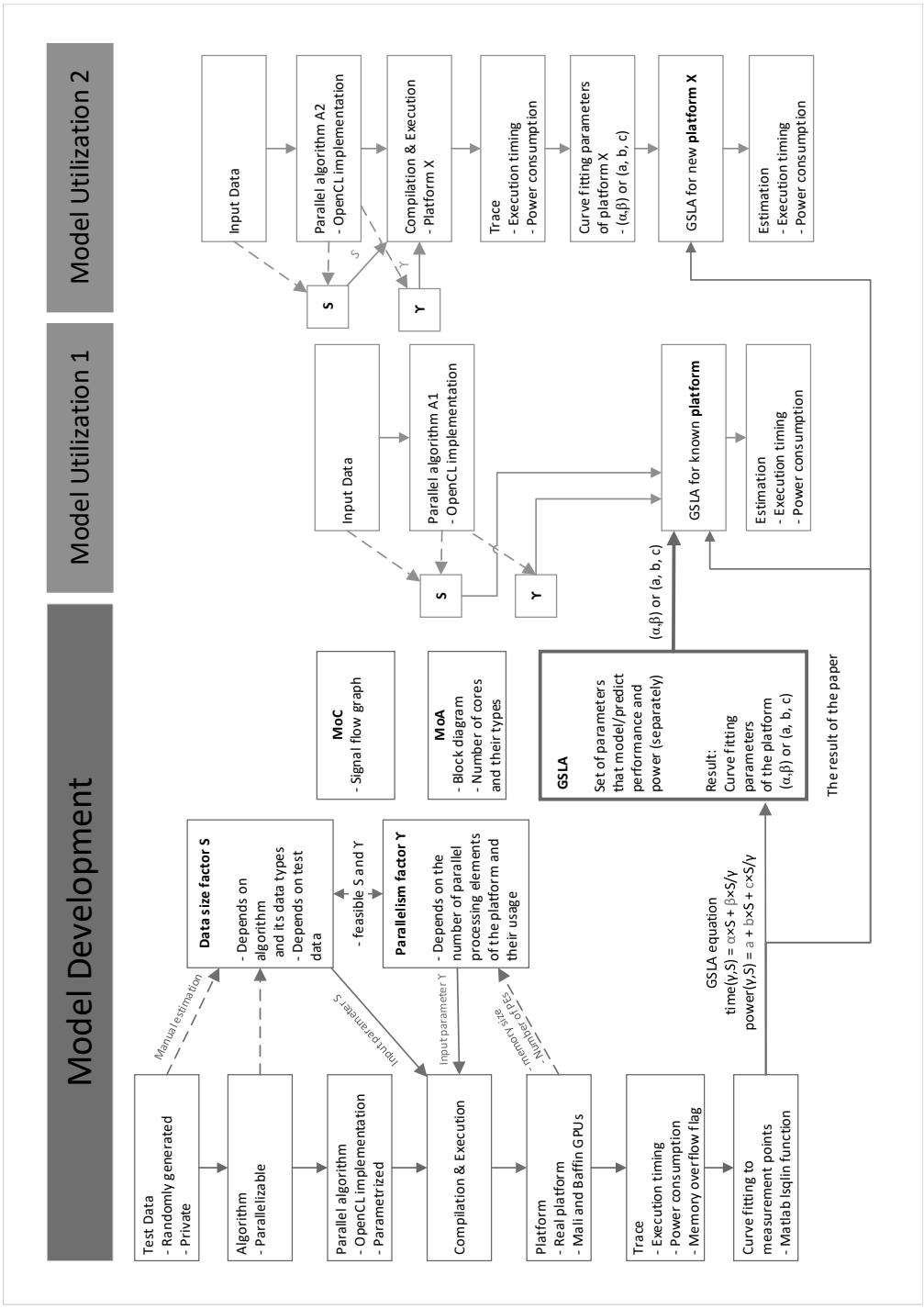

Fig. 1 Development and utilization work flow.

a graph to present a software or a hardware system, or a system of systems. These methods are divided into two different categories: data flow graphs and non data flow graphs. In a data flow graph, a vertex is used to model a run-to-completion block of computation called an *actor*. *Edges* are used to model data token communications between actors, realized by FIFO queues (First In First Out). In addition, weights on FIFOs, called delays, are used to represent initial data present on edges. The execution of a data flow actor is called *firing* and is triggered when an actor has sufficient data on each input edge. Table 1 lists modeling approaches and the graph semantics used in related works.

SDF (Synchronous Data Flow) [9] is a well-known static MoC. In SDF, a system is modeled with a data flow graph where the firing rules specify the constant token consumption and production rates for all actors. These constant rates introduce limitations in terms of algorithmic behavior representation.

CFDF (Core Functional Data Flow) [17] is a form of EIDF (Enable Invoke Data Flow) [18] where a limited set of modes influence token consumptions and productions. CFDF limits mode transitions to only one alternative, making the model deterministic.

BSP (Bulk Synchronous Parallel) [22], unlike SDF or CFDF, is a system modeling method rather than an application modeling method, and it has its own graph representation. In BSP, there are processing units with local memories connected over a router. Processing elements access each other's memories by remote access messages.

DAL (Distributed Application Layer) [20] has a dynamic mapping methodology. It employs Kahn process networks to explore application mappings and a finite state machine to represent execution scenarios. Multiple scenarios are precomputed at design-time and the suitable one is selected at run-time.

Bezati et al. [4] present a data flow modeling method according to the CAL language [5]. Their method has six steps. First, two different models for application and architecture are designed. Second, simulation and profiling results are collected. Third, the application is mapped to the architecture. Fourth, C++ and HDL codes are generated from CAL. Fifth, the code is compiled and synthesized. Finally, compiled code is executed.

LSLA [15] is a MoA, modeling hardware architecture separately from the MoC. The LSLA MoA includes Processing Elements (PE) and Communication Nodes (CN). PEs and CNs of the LSLA MoA have cost functions including parameters that may be retrieved from representative platform benchmarking. In that case, the calculated cost functions are obtained from measured application executions and the cost function parameters can be used to predict system efficiency for a set of comparable applications.

Holmbacka et al. [7] studied the energy consumption of different phases of the applications on multi-core CPUs. For utilizing the Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) of parallel platforms, they ran the parallel phases with as low as possible clock frequency on multiple cores without missing any deadline and sequential phases with higher clock frequency on a single core. For controlling the hardware features, they introduced two parameters including the level of parallelism

and the quality of service and call it QP-aware (QoS and Parallel) strategy. Running a program with lower clock frequency instead of a race-to-idle strategy provides an energy efficient solution by reducing the frequent frequency switching overhead. Authors used PREESM [16] for compiling the applications and extracting the level of parallelism and deployed a non linear programming solver for the QoS handling. In addition, they presented a platform specific power model as a function of DVFS and DPM usage.

The energy efficiency survey [10] classifies the utilized techniques for improving GPU energy efficiency and compares them with methods applied to other computing units such as FPGAs. Authors use five categories including dynamic voltage frequency scaling (DVFS), division-based CPU-GPU, architectural techniques, dynamic workload variation and application-specific programming-level approaches. A conclusion to this work is that the power consumption of the GPU should be considered at multiple design phases with several techniques to achieve desirable efficiency. The proposed GSLA model falls under this objective, as it aims at making power estimates available early during the design phase.

The proposed work on the GSLA model provides a system modeling approach as an extension to LSLA. It has the benefits of a reduced modeling effort due to its re-usability. Table 1 summarizes the modeling approaches in order to compare to this work. As can be seen, our work is focusing on the architecture and supports wide range of applications.

Table 1 Modeling approaches

Method	Target	Application Graph
SDF	Application	Dataflow
CFDF	Application	Dataflow
BSP	System	Non Dataflow
DAL	System	Non Dataflow
[4]	System	Dataflow
LSLA	Architecture	Not restricted to dataflow
GSLA	Architecture	Non Dataflow

3.1 Polynomial modeling comparisons

The power model developed in [7] uses Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) platform variables. Authors use Levenberg-Marquardts algorithm and aim at a high modeling precision which resulted in a third degree polynomial with seven terms. As an MoA, GSLA is not specialized to power modeling and has a lower complexity with three terms, keeping its complexity minimal. The usage of the fidelity metric for evaluation results in a model with lower complexity differentiates this work from studies using similar methodology, as the objective is not to have an accurate model but rather to take the right design decisions.

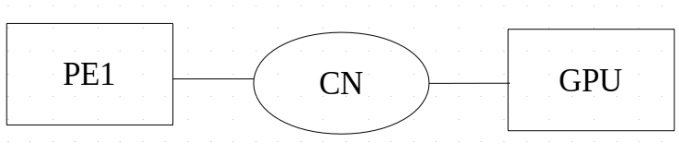

Fig. 2 Model of Architecture.

The energy model [6] presents the number of active cores and frequency as variables. Authors have considered three possibilities for the workload processing and depict three variations of their model. The variations of the terms number in their model is at least two and is impacted by the number of active cores. The proposed model is designed for static scheduling and requires timing data such deadlines and power consumption values of the active cores for predicting energy consumption. Compared to this tailored model, GSLA presents a simpler formula and we show that it still can capture several key performance indicators.

The energy per cycle model introduced in [12] uses normalized frequency variable. Authors use the Levenberg-Marquardts algorithm for calculating their model equations which has three terms and it is in degree three. This model targets power and frequency data for the energy computation while the experiments are depicted for a limited set of measurements. On the other hand, GSLA with lower complexity is demonstrated as a two dimensional model fitting thirty six average measurement points.

4 Models of Architecture

The MoA concept [15] is used to distinguish the processing architecture from the MoC, which should only address applications. Consequently, in this concept a system of an application running on a platform is presented with a MoC mapping on a MoA. A MoA is defined as a graph that in conjunction with the mapped MoC can be used for reproducible execution cost (time, energy, etc.) calculations. A MoA is designed for each specific processing architecture and it covers the processing elements and their interconnect. Figure 2 depicts an MoA which contains two processing elements and one interconnection.

Each element in the MoA graph has a cost function whose parameters can be estimated statistically according to measurement results of the mapped MoC element to the considered MoA element. The calculated parameter values depend on the application and the application configuration, but the search of the parameters is automated and requires only executable application(s) and test data from profiling.

4.1 Linear System Level Architecture

LSLA is a specific type of MoA that uses *linear* cost functions for each MoA graph element. The total cost of the modeled platform is calculated according to the Equation 1, which depicts the total cost of application activity A on the LSLA graph P. In this equation, the total cost e.g. execution time cost is equal to the sum of the processing cost, and of the communication cost, λ being a scaling coefficient between processing and communication cost units. Tp depicts set of all mapped tokens to the processing elements and Tc shows set of all mapped tokens to the communication nodes. The activity of the mapped MoC is calculated as *tokens*, consisting of *quanta*, resulting in an affine cost model per communication and per processing. The quanta are an application-independent unit of execution cost.

$$cost(A, P) = \sum_{t \in Tp} cost(t, map(t)) + \lambda \sum_{t \in Tc} cost(t, map(t)) \tag{1}$$

In a system running an application with multiple dependent tasks on a platform with multiple processing elements, parallel application mapping and scheduling are required. While mapping refers to assigning tasks to processing elements, scheduling refers to ordering task execution on each processing element. On the modeling side, mapping an activity to a platform modeled with an MoA refers to the assignment of a unique processing element or communication node to each token in the application activity. The activity abstracts the *pressure* the application puts on hardware, resulting in physical ,properties such as time and energy consumption. In our experiments, GPU and CPU tasks are mapped manually, but the designed model can feed automated mapping processes.

In LSLA, the application and its activity (i.e. the pressure that it puts on hardware) are mapped as activity tokens to the LSLA model of the platform. Activity of the application includes *processing tokens* and *communication tokens*. These tokens are mapped to their associated elements in the platform model: processing tokens are mapped to processing elements and communication tokens are mapped to interconnection nodes that are used to transfer data between PEs.

5 GSLA: Execution Time Modeling

This work adds GPUs that can be present in a modern heterogeneous platform. Figure 3 shows a simple GSLA graph that includes a CPU core *PE1*, the GPU, and the interconnection *CN* between the CPU core and the GPU. *PE1* is assumed to act as the host processor that communicates with the GPU. This simple GSLA model has three elements including two processing elements and one communication node.

Due to the very different characteristics of power and execution time modeling, we use different parameters and a slightly different model for both; the

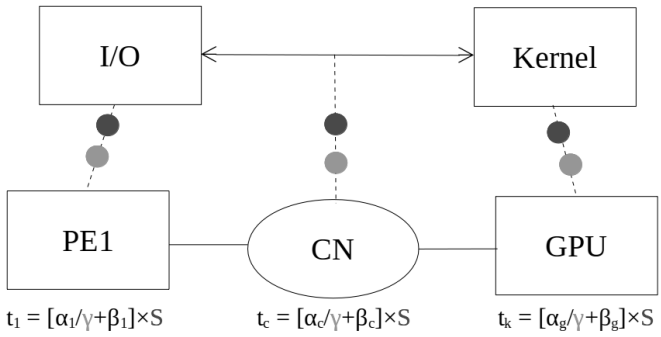

Fig. 3 Mapping of the application to the architecture model.

power model is presented in Section 8, whereas the execution time model is presented as follows: each element has its own cost function (presented beneath the nodes) that has two variables named γ and S, as well as two linear parameters α and β whose values are estimated for modeling purposes. One may note that the γ and S symbols refer to variables while bold characters represent the sets of values used during the profiling phase. The presented GSLA is used to model the execution time of the platform, thus time samples are used in parameter calculations. As presented in Equation 1, the total cost is a sum of all cost elements, i.e. the execution time of the GPU (t_k), the execution time of the host processor (t_1) and the execution time of the interconnect (t_c). The model of Equation 2 is justified by the consideration that times (t_k), (t_1) and (t_c) do not overlap in time, i.e. the kernels of the GPU application are managed by the host device, then executed by the GPU during separate time intervals.

$$t_w = t_k + t_1 + t_c \tag{2}$$

$$t_k(\gamma, S) = (\alpha_g/\gamma + \beta_g) \times S \tag{3}$$

$$t_1(\gamma, S) = (\alpha_1/\gamma + \beta_1) \times S \tag{4}$$

$$t_c(\gamma, S) = (\alpha_c/\gamma + \beta_c) \times S \tag{5}$$

In these equations γ and S are variables, where γ is the parallelism factor, and S is the input data quantity. As it can be seen, increasing the parallelism factor γ decreases the total execution cost asymptotically. Each Equation 3, 4 and 5 follows an Amdahl's law [1] with β representing the incompressible time cost of a sequential section and α representing the compressible time cost of a perfectly parallel region. Conventionally, LSLA does not deal with internal processing element parallelism, which limits its usage to cores with limited concurrency. GSLA adds the parallelism factor γ that makes it possible to

include parallel processing elements. For each GPU-related MoA graph entity (i.e., the GPU itself, the host PE and the interconnect) there are separate parameters α and β, where α can be regarded as the reciprocal of slope, and β as intercept. The section 6 describes the proposed approach of estimating each α and β.

5.1 Usage of the methodology

Figure 4 depicts the tool flow steps for creating the cost prediction using the models either for performance or power. Applications are characterized with the **S** and $\boldsymbol{\gamma}$ sets, written in OpenCL source code and compiled for execution and measurements. The employment of the models is provided by a shell script and a Matlab script which is depicted in Figure 4. First, the **S** and $\boldsymbol{\gamma}$ sets are selected and the shell script is edited accordingly. Then, the scripts perform the mapping, compilation and execution of the OpenCL application. The application receives S and γ as command line arguments. Pseudocode 1 presents an example of S and γ implementation in the matrix multiplication. Later, the Matlab script is used to extract the parameters α and β. Finally, the parameters are used to estimate the appropriate workload.

Pseudocode 1 The S and γ Implementation in Matrix Multiplication

```
INPUT: S, γ
OUTPUT: execution time print
int main(int argc, char * argv[ ])  {
  ...
  size_t globalWorkSize[3];
  int parMtx = atoi(argv[1]);
  int globalWorkSize[0] = MTX_SIDE;
  int globalWorkSize[1] = MTX_SIDE;
  int globalWorkSize[2] = parMtx;
  ...
  int workSize = globalWorkSize[0] * globalWorkSize[1] * globalWorkSize[2];
  int totalLen = (atoi(argv[2]) * 256 * 4)/4;
  int iterations = totalLen/workSize;
  ...
  for(int i = 0; i < iterations; i + +){
  ...
  clEnqueueNDRangeKernel(commands, krnMatMul, 3, NULL, globalWorkSize, ...);
  ...
  }
  ...
}
```

The parameters can be used for similar applications on equivalent platforms, or recalculated for other kind of platforms and applications.

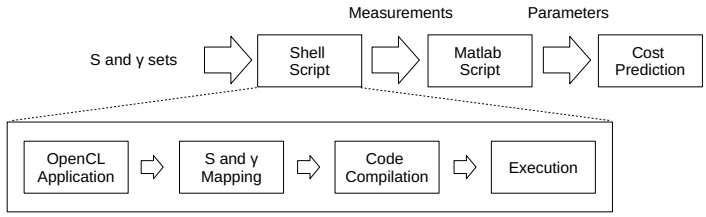

Fig. 4 Steps in the tool flow.

6 Estimation of Parameters

Acquiring an accurate execution time model for an application running on a GPU requires reliable profiling data. The proposed estimation approach assumes three accurate terms that can be profiled on the platform

- Application total *wall-clock time* t_w,
- Host code execution time t_1, and
- GPU kernel execution time t_k.

The remaining term t_c, in contrast, is derived using t_w, t_1 and t_k. The proposed procedure for acquiring accurate measurements for the terms are as follows: t_w is measured using the operating system clock, and t_k is read from the profiling data available from the GPU application programming interface. The measurement of t_1 is performed by modifying the application so that all GPU-related calls are disabled and the application only performs data I/O. Finally, t_c is derived from the other terms by subtracting t_1 and t_k from t_w.

The Pseudocode 2 presents the Matlab script used for modeling the performance of the applications according to the proposed Equation 2, 3, 4 and 5. The $\mathbf{S}$ and $\boldsymbol{\gamma}$ sets have the same number of elements and their values are considered according to the input data size of the applications and the memory of the platforms.

Pseudocode 2 Matlab Script of Performance Model

INPUT: $t_w = [...],\ t_k = [...],\ t_1 = [...]$
OUTPUT: $\alpha_g,\ \beta_g,\ \alpha_1,\ \beta_1,\ \alpha_c,\ \beta_c$
PerformanceModel {
  $\mathbf{S} = [...]$;
  $\boldsymbol{\gamma} = [...]$;
  $\mathrm{t_c} = \mathrm{t_w} - \mathrm{t_1} - \mathrm{t_k}$;
  $\alpha_\mathrm{g},\ \beta_\mathrm{g} = \mathrm{lsqlin}(\mathbf{S},\ \mathbf{S}/\boldsymbol{\gamma},\ \mathrm{t_k})$;
  $\alpha_\mathrm{c},\ \beta_\mathrm{c} = \mathrm{lsqlin}(\mathbf{S},\ \mathbf{S}/\boldsymbol{\gamma},\ \mathrm{t_c})$;
  $\alpha_1,\ \beta_1 = \mathrm{lsqlin}(\mathbf{S},\ \mathbf{S}/\boldsymbol{\gamma},\ \mathrm{t_1})$;
}

7 Experiments: Execution Time Modeling

The experiments presented below serve to illustrate the suitability of the proposed model and Equations 2-5 for real-life GPU-equipped platforms. Typical signal processing applications were used as case studies: matrix multiplication, digital predistortion and Gaussian image filtering. The applications were written in OpenCL and were executed on two GPU-equipped platforms: the Odroid XU3 containing a Mali T628 GPU and a desktop workstation with the AMD RX 460 (Baffin) GPU. The test input data was randomly generated for matrix multiplication and predistortion, and private data for Gaussian filtering.

The α and β parameters of the cost functions were obtained with a Matlab script that invoked a least squares fitting algorithm (see Section 2 of [2]). In the Matlab script, the `lsqlin` function was used with a positive solution constraint as a standard method to perform linear regression.

Each application was profiled while varying two application variables, i.e., S and γ. For setting parameters values, the applications input data sizes, the memory of the platform, and the profiling duration have been considered. As the applications are simple, setting their data parallelism γ is straightforward and corresponds to the number of parallel fired kernels. Each obtained variable had six values where $S \in \{$ 512, 1024, 2048, 4096, 8192, 16384 $\}$ and $\gamma \in \{$ 8, 16, 32, 64, 128, 256$\}$. The global work size of OpenCL applications was set application dependently. For matrix multiplication and predistortion, the work size set was calculated by 256*γ, while for gaussian filtering, it is 1024*γ. The reason for this variation is in the input data types i.e. gaussian filtering reads 1-byte data, while matrix multiplication and predistortion read 4-byte data items. For each (γ, S) combination the execution time was measured 10 times, giving a total of 360 samples per application/architecture combination.

7.1 Application-architecture mapping

In OpenCL, when computations are performed on a GPU, the CPU works as the host device that reads data from I/O, sends it to the GPU for processing, receives the computed result and stores it back to I/O. Based on the dataflow [9] MoC, a generic model for OpenCL applications was created. Data reading and writing of the CPU is mapped to an *I/O* node (see Figure 3). The *Kernel* node represents the computations performed on the GPU, whereas the communication between the *I/O* and *Kernel* nodes is presented with a bidirectional arrow in Figure 3.

In each actor firing of the application graph, actors and the communication FIFO provide a token, which is mapped to their associated PE or CN node of the model. In other words, the tokens of the node *I/O* are mapped to the *PE1* architecture node, the tokens of *Kernel* are mapped to the *GPU* architecture node, and the communication FIFO tokens to the *CN* node. The cost functions shown below the architecture nodes have two variables, thus two tokens on the

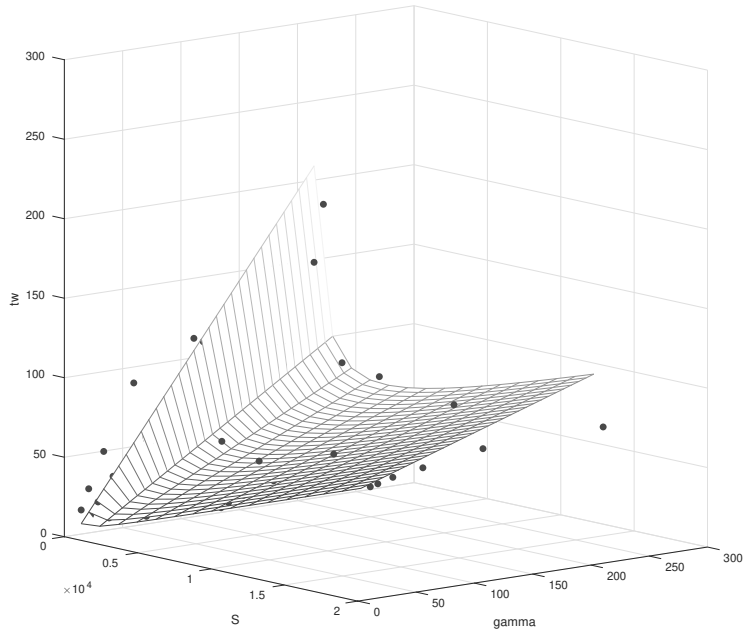

Fig. 5 Execution time of matrix multiplication on the Baffin GPU.

mapping lines in Figure 3 are used to present the number of quanta for each variable.

7.2 Execution Time Results

This section shows how the proposed GPU execution time model fits with the measured execution time samples. In Figure 5, Figure 6, and Figure 7 the bottom axes depict the variables S and γ, whereas the vertical axis depicts execution time. The dots represent the average of individual measured execution time samples.

The measured execution time samples are t_w (wall-clock time) values, and the mesh depicts the model-based sum of $t_k + t_1 + t_c$. For clarity, the measured time samples depict the average of the 10 measurements for each (γ, S) coordinate.

Table 2 depicts the calculated α and β parameter values for each application on Baffin and Mali GPUs. These parameters are used in the Equation 2 for calculating t_w. The α value represents the cost of a token and equals to the slope of the mesh. The β value represents the constant time offset of the relevant GSLA element and is the t_w intercept of the mesh graph. Due to technical difficulties, values for the digital predistortion application were not acquired on the Mali platform. App 1 is matrix multiplication, App 2 is digital predistortion and App 3 is Gaussian filtering. M stands for Mali and B for Baffin platforms.

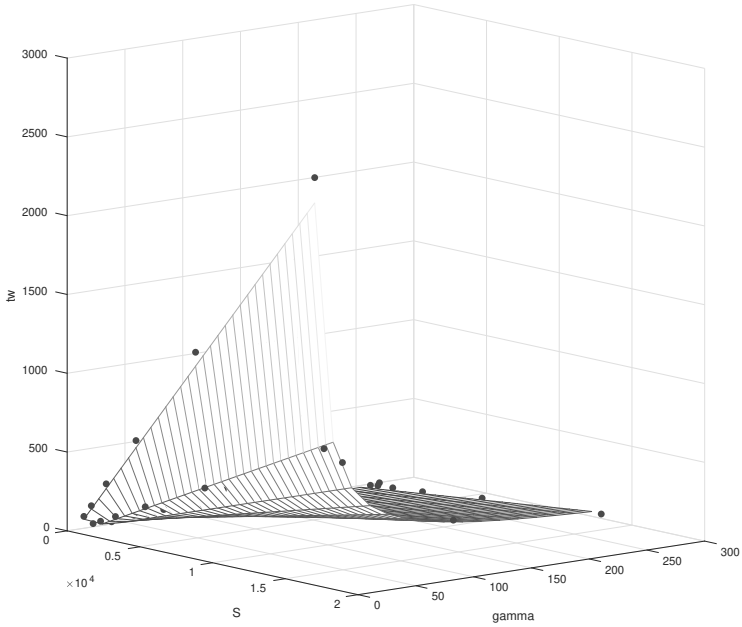

Fig. 6 Execution time of Gaussian filtering on the Baffin GPU.

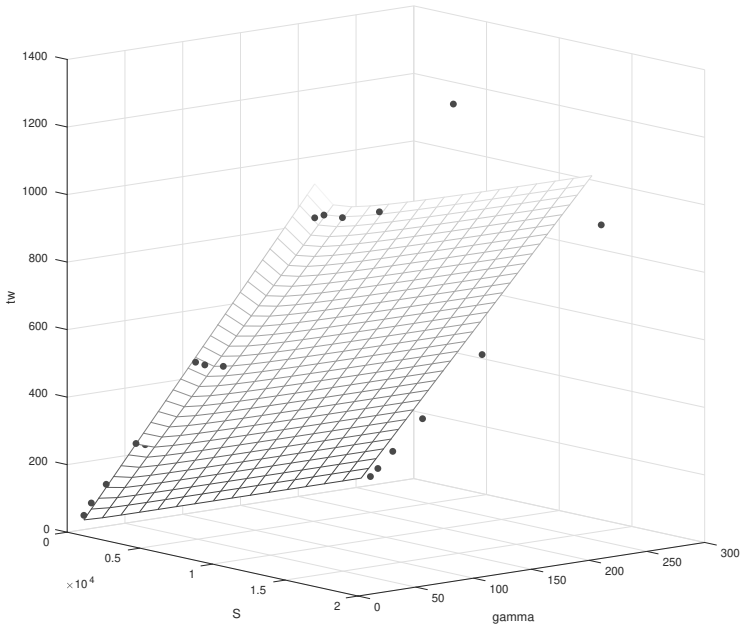

Fig. 7 Execution time of Predistortion on the Baffin GPU.

Table 2 Calculated cost function parameters

App.	α_g	β_g	α_1	β_1	α_c	β_c
B1	0.001	0.008	0.000	0.004	0.005	0.068
M1	0.003	0.009	0.000	0.009	0.016	1.554
B2	0.005	0.049	0.002	0.003	0.060	0.000
B3	0.000	0.051	0.002	0.000	0.004	1.074
M3	0.003	0.023	0.022	0.000	0.082	0.460

Table 3 Fidelity of the test sets on execution platforms (Kendall tau coefficient between −1 and 1, 1 is the best).

Application	Platform	Fidelity GSLA (proposed)	Fidelity LSLA
1	Baffin	0.88	0.75
1	Mali	1.00	0.70
2	Baffin	0.90	0.92
3	Baffin	0.91	0.62
3	Mali	1.00	0.92

Table 3 demonstrates the fitting error between the model and measured samples for each application/platform combination as *fidelity* values. To highlight the improvement of the proposed GSLA model over conventional LSLA for GPU targets, Table 3 also shows the fidelity value for LSLA. Fidelity is computed similarly to [3] with the Kendall Tau Coefficient value, as calculated by the `corr` function of Matlab when configured for it. Fidelity assesses the capacity of the model to correctly order samples, 1 corresponding to a perfect order and −1 a perfectly reverse order. Indeed, a *good* model is a model that feeds good decisions more than a model with good absolute accuracy. A value of zero, as a worst case, would suggest independence in ranking between model and measurements. For computing the fidelity, the 360 execution time samples were randomly divided into a training set of 288 samples, and a test set of 72 samples.

In the Table 3 results, it can be seen that conventional LSLA yields considerably worse fidelity than the proposed GSLA for GPU architectures. The reason for this is evident: LSLA does not capture parallelism (γ), which is an integral part of GPU processing. An exception to this is the predistortion application on the Baffin GPU, where LSLA and GSLA yield almost identical fidelity. The reason for this is that on this platform, communication time dominates over parallelized kernel execution, making the whole application behave almost similar to a sequential application. Dominance of communication can be seen in Table 2 as the high value of coefficient α_c for application B2.

The measured fidelity values also show that the proposed non-linear GSLA model fits better the Mali platform than the Baffin platform. The difference is likely related to the different memory architectures; Mali uses a shared memory between the CPU and the GPU, whereas the Baffin GPU is connected over PCI Express.

8 GPU Power Modeling

Besides GSLA for performance modeling, a power model is proposed for predicting the average power consumption of an OpenCL applications. The same Odroid XU3 platform is used for power profiling as it includes power sensors for CPU, GPU and memory. In our measurements we noticed almost constant CPU power which is expected from OpenCL applications running on GPU. Also, we ignored memory power consumption. Consequently, we only considered the GPU power dissipation as seen in Equation 6. In this equation the p_t is the total power of the GPU.

In our experiments, we recognized that power consumption modeling with reasonable accuracy requires a third constant term in comparison to execution time modeling. In order to keep the complexity at reasonable levels, we tried to come up with the simplest possible model for capturing the GPU power. Consequently, the proposed power model has three parameters.

$$p_t(\gamma, S) = a_{GPU} + b_{GPU}S + c_{GPU}S/\gamma \tag{6}$$

As experiments, matrix multiplication and the Gaussian filtering are executed on the Mali platform and power values are read from the sensors. Profiling was similar to performance model i.e. with two application variables S and γ with the same values where $S \in \{$ 512, 1024, 2048, 4096, 8192, 16384 $\}$ and $\gamma \in \{$ 8, 16, 32, 64, 128, 256$\}$. Figure 8 presents power modeling of matrix multiplication and Figure 9 for Gaussian filtering. In both of these figures *pt* axis demonstrates only GPU power consumption. Table 4 shows the power model's parameters of these applications on the Mali GPUs. The values of the b_{GPU} and c_{GPU} are very small in comparison to other parameters. The a_{GPU} is almost constant at 0.12 representing a static power consumption of $120mW$ i.e. the intercept of equation and its larger values effect the total power consumption.

Table 5 depicts the fidelity of the proposed power model. These values could improve slightly with the cost of increasing the complexity of the model. For example, Equation 7 was tested and rejected for the power model with R-squared value of 0.862 for matrix multiplication and 0.917 for Gaussian application, which does not justify the increased complexity.

$$p_t(\gamma, S) = a_{GPU} + b_{GPU}S + c_{GPU}/\gamma + d_{GPU}S/\gamma \tag{7}$$

The S and γ variables depicted in Equation 6 show the input data quantity and parallelism factor. From the observations we noticed similar variable relations like the performance cost for the measured power samples. We observed that, logically, the input data quantity increases the power consumption while an increase of the parallelism reduces the power consumption. In addition, the Pseudocode 3 depicts the Matlab script of the power modeling according to the proposed Equation 6. The values of the S and the γ sets are considered according to the memory capacity of the targeted hardware and the input data size of the applications.

Pseudocode 3 Matlab Script of Power Model

INPUT: $p_t = [...]$
OUTPUT: a_{GPU}, b_{GPU}, c_{GPU}
PowerModel {
S = [...];
γ = [...];
a_{GPU}, b_{GPU}, c_{GPU} = lsqlin(p_t, [ones(S), S, S/γ]);
}

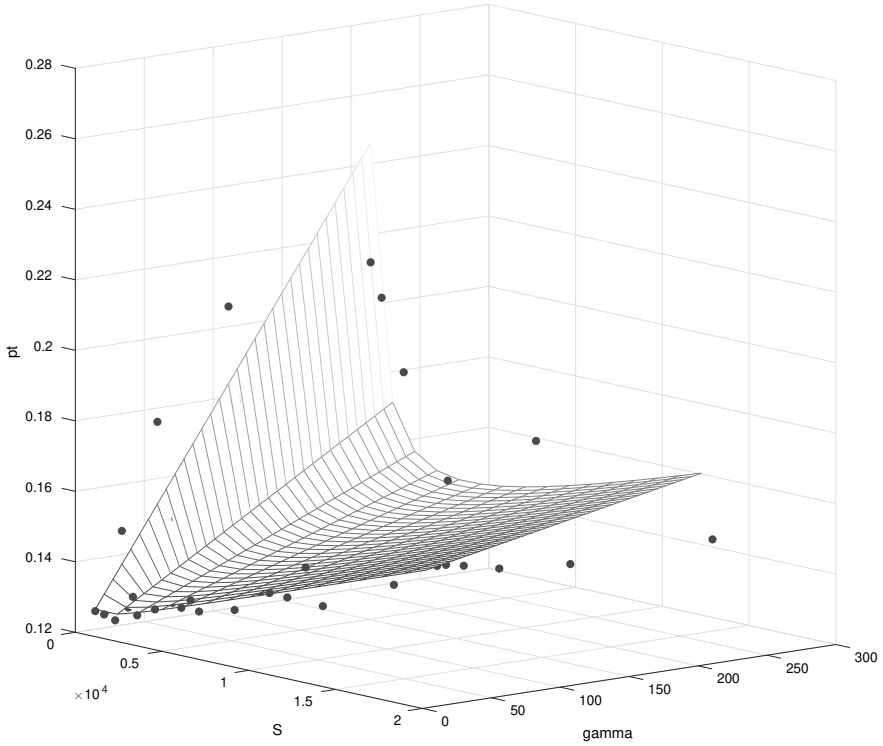

Fig. 8 Power modeling of Matrix multiplication on the Mali GPU (power in Watts).

Table 4 Cost function parameters of power modeling.

Application	a_{GPU}	b_{GPU}	c_{GPU}
M1	0.121747	0.000003	0.000055
M3	0.121422	0.000052	0.000191

Table 5 Fidelity of the power model (Kendall tau coefficient between -1 and 1,1 is the best).

Application	Platform	Fidelity power model (proposed)	Fidelity LSLA
1	Mali	0.740	0.628
3	Mali	0.943	0.916

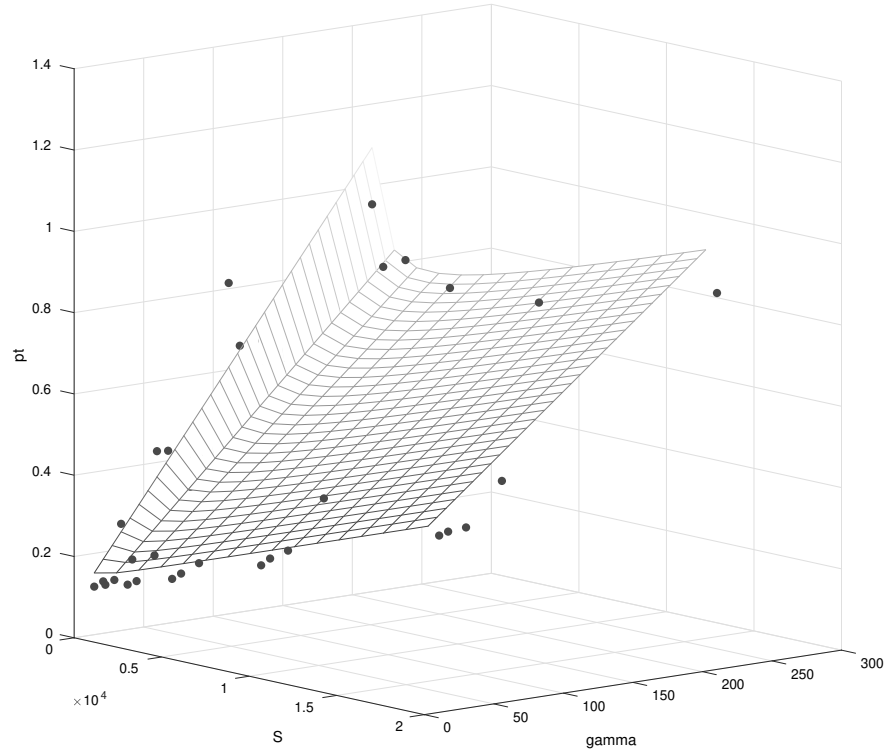

Fig. 9 Power modeling of Gaussian filtering on the Mali GPU (power in Watts).

9 Fitness of the models

The proposed LSLA extension aims to model parallel execution on the GPU of the considered platform. With the following fitness study, we show that the created GSLA model is capable of fitting both GPU performance and GPU power consumption. The performance model shown in Equation 8 and the power model presented in Equation 6 have the same terms with exception of an extra constant term for Equation 6. This constant logically models the static power of the platform while timing is null when no computation is requested. The fitness of the performance model and the presented power model are compared using the R-squared for the power measurements calculated with Matlab *regress* function. Table 6 depicts the fitness comparison of the models.

The R-squared values of 0.850 and 0.916 for Equation 6 in comparison to values for the Equation 8 justifies the selection of the Equation 6 as the power model. The horizontal line has a better fit than Equation 8 for power samples of the matrix multiplication application so the R-squared value. This suggests the requirement to add a constant value in Equation 6.

The conclusion is that GSLA as defined by a sum of contributions obtained with equation 6, is capable of modelling both GPU power and performance with only 3 parameters.

$$t_w(\gamma, S) = \alpha\ S/\gamma + \beta\ S \tag{8}$$

Table 6 Fitness of Models

Application	Equation	R-squared
1	8	-5.415
1	6	0.850
3	8	0.854
3	6	0.916

10 Conclusion

We presented a new Model of Architecture called GSLA (GPU-oriented System-Level Architecture). GLSA is tailored to GPU modeling but is capable of modeling both performance and average power of the targeted GPU. Contrary to the preexisting LSLA model, GSLA includes non-linear constructs, but reasonably fits the power consumption of a complex GPU with only 3 parameters. The validity of the proposed model is evaluated by profiling three OpenCL applications on two GPU-equipped platforms. The achieved model fidelity is 93% for execution latency and 84% for power. Such performances can be considered sufficient for design space exploration purposes.

In future, other lightweight machine learning techniques will be investigated for building models from platform measurements, expecially in more heterogeneous contexts combining e.g. CPU and GPU.

Acknowledgements This work was partially supported by the ITEA3 project 16018 COMPACT and by the European Union Horizon 2020 research grant 732105 CERBERO. We thank Jani Boutellier, Tapio Nummi, Antoine Morvan and Claudio Rubattu for their helpful guidance.

References

1. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, spring joint computer conference, pp. 483–485 (1967)
2. Aster, R.C., Borchers, B., Thurber, C.H.: Parameter estimation and inverse problems. Elsevier (2018)
3. Bambha, N.K., Bhattacharyya, S.S.: A joint power/performance optimization algorithm for multiprocessor systems using a period graph construct. In: Proceedings of the 13th international symposium on System synthesis, pp. 91–97. IEEE Computer Society (2000)
4. Bezati, E., Thavot, R., Roquier, G., Mattavelli, M.: High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms. Journal of real-time image processing **9**(1), 251–262 (2014)

5. Eker, J., Janneck, J.: Cal language report. Tech. rep., Tech. Rep. ERL Technical Memo UCB/ERL (2003)
6. Holmbacka, S., Keller, J., Eitschberger, P., Lilius, J.: Accurate energy modelling for many-core static schedules. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 525–532. IEEE (2015)
7. Holmbacka, S., Nogues, E., Pelcat, M., Lafond, S., Lilius, J.: Energy efficiency and performance management of parallel dataflow applications. In: The 2014 Conference on Design & Architectures for Signal & Image Processing (2014)
8. Kienhuis, B., Deprettere, E.F., Van der Wolf, P., Vissers, K.: A methodology to design programmable embedded systems. In: International Workshop on Embedded Computer Systems, pp. 18–37. Springer (2001)
9. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987)
10. Mittal, S., Vetter, J.S.: A survey of methods for analyzing and improving gpu energy efficiency. ACM Computing Surveys (CSUR) **47**(2), 1–23 (2014)
11. Moren, K., Göhringer, D.: Automatic mapping for opencl-programs on cpu/gpu heterogeneous platforms. In: International Conference on Computational Science, pp. 301–314. Springer (2018)
12. Nogues, E., Pelcat, M., Menard, D., Mercat, A.: Energy efficient scheduling of real time signal processing applications through combined dvfs and dpm. In: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 622–626. IEEE (2016)
13. Payvar, S., Boutellier, J., Morvan, A., Rubattu, C., Pelcat, M.: Extending architecture modeling for signal processing towards gpus. In: 2019 27th European Signal Processing Conference (EUSIPCO), pp. 1–5. IEEE (2019)
14. Pelcat, M., Desnos, K., Maggiani, L., Liu, Y., Heulot, J., Nezan, J.F., Bhattacharyya, S.S.: Models of architecture: Reproducible efficiency evaluation for signal processing systems. In: IEEE International Workshop on Signal Processing Systems (SiPS), pp. 121–126. IEEE (2016)
15. Pelcat, M., Mercat, A., Desnos, K., Maggiani, L., Liu, Y., Heulot, J., Nezan, J.F., Hamidouche, W., Ménard, D., Bhattacharyya, S.S.: Reproducible evaluation of system efficiency with a model of architecture: From theory to practice. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2017)
16. Pelcat, M., Piat, J., Wipliez, M., Aridhi, S., Nezan, J.F.: An open framework for rapid prototyping of signal processing applications. EURASIP journal on embedded systems **2009**, 11 (2009)
17. Plishker, W., Sane, N., Kiemb, M., Anand, K., Bhattacharyya, S.S.: Functional dif for rapid prototyping. In: Rapid System Prototyping, 2008. RSP'08. The 19th IEEE/IFIP International Symposium on, pp. 17–23. IEEE (2008)
18. Plishker, W., Sane, N., Kiemb, M., Bhattacharyya, S.S.: Heterogeneous design in functional dif. In: International Workshop on Embedded Computer Systems, pp. 157–166. Springer (2008)
19. Rigo, S., Araujo, G., Bartholomeu, M., Azevedo, R.: Archc: A systemc-based architecture description language. In: 16th Symposium on Computer Architecture and High Performance Computing, pp. 66–73. IEEE (2004)
20. Schor, L., Bacivarov, I., Rai, D., Yang, H., Kang, S.H., Thiele, L.: Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In: Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems, pp. 71–80. ACM (2012)
21. Thiele, L., Bacivarov, I., Haid, W., Huang, K.: Mapping applications to tiled multiprocessor embedded systems. In: Seventh International Conference on Application of Concurrency to System Design (ACSD 2007), pp. 29–40. IEEE (2007)
22. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM **33**(8), 103–111 (1990)

PUBLICATION III

Neural Network-based Vehicle Image Classification for IoT Devices

S. Payvar, M. Khan, R. Stahl, D. Mueller-Gritschneder and J. Boutellier

*2019 IEEE International Workshop on Signal Processing Systems (SiPS)*2019, 148–153
DOI: 10.1109/SiPS47522.2019.9020464

Neural Network-based Vehicle Image Classification for IoT Devices

Saman Payvar
Unit of Computing Sciences
Tampere University
Tampere, Finland
saman.payvar@tuni.fi

Mir Khan
Unit of Computing Sciences
Tampere University
Tampere, Finland
mir.markhan@tuni.fi

Rafael Stahl
Chair of Electronic Design Automation
Technical University of Munich
Munich, Germany
r.stahl@tum.de

Daniel Mueller-Gritschneder
Chair of Electronic Design Automation
Technical University of Munich
Munich, Germany
daniel.mueller@tum.de

Jani Boutellier
Unit of Computing Sciences
Tampere University
Tampere, Finland
jani.boutellier@tuni.fi

***Abstract*—Convolutional Neural Networks (CNNs) have previously provided unforeseen results in automatic image analysis and interpretation, an area which has numerous applications in both consumer electronics and industry. However, the signal processing related to CNNs is computationally very demanding, which has prohibited their use in the smallest embedded computing platforms, to which many Internet of Things (IoT) devices belong. Fortunately, in the recent years researchers have developed many approaches for optimizing the performance and for shrinking the memory footprint of CNNs. This paper presents a neural-network-based image classifier that has been trained to classify vehicle images into four different classes. The neural network is optimized by a technique called binarization, and the resulting binarized network is placed to an IoT-class processor core for execution. Binarization reduces the memory footprint of the CNN by around 95% and increases performance by more than 6×. Furthermore, we show that by utilizing a custom instruction 'popcount' of the processor, the performance of the binarized vehicle classifier can still be increased by more than 2×, making the CNN-based image classifier suitable for the smallest embedded processors.**

***Index Terms*—model compression, convolutional neural networks, image classification, internet-of-things**

I. INTRODUCTION

Convolutional neural networks (CNNs) have enabled a significant advance in automatic image analysis, such as image classification [1], image segmentation [2], image captioning [3] and object detection [4]. Unfortunately, up to recently the computational requirements of CNNs have restricted their use to server or desktop class computers, although their deployment to *edge devices* could open up a variety of new applications [5]. In the Internet-of-Things (IoT), the network edge refers to devices that are within immediate connection to sensors that provide input data for the whole IoT system. Such an edge device can be a smartphone [6], or a tiny sensor node commonly equipped with less than a megabyte of RAM [7].

A CNN consists of a sequence of *layers*, of which the most common types are *fully-connected layers* and *convolutional layers*. Once a CNN has been trained [8], e.g. for image classification, the *parameters* and *weights* of the layers are fixed for deployment to a target device. On the target device, the process that evaluates given input data is called *inference*, where the input data flows through the layers of the CNN, providing the requested output (e.g. classification result) from the last layer.

In terms of computation, convolutional layers consist of repeated 2D convolutions, where the input data of the layer is convolved by 2D kernels with common sizes of 5×5, 3×3 or 1×1 [9]. The computational effort of convolutional layers grows rapidly as the size of input images or kernels grows [10]. However, it has been well-known for some time that 2D convolution can also be interpreted and computed as a 2D matrix multiplication [11]. The inference of a fully-connected layer is also commonly performed by 2D matrix multiplication.

Optimization of CNN processing can be performed by optimizing software, hardware, or both [12]. Examples for software-based optimizations are model compression [9][13] or reduction of arithmetic precision [14][12]. Software-based optimizations that target convolutional layers include separable convolution [15] and depthwise convolution [16], whereas fully-connected layers can be optimized by weight pruning [13]. All of these optimizations have some negative impact on the CNN accuracy.

Reduction of arithmetic precision, on the other hand, is not limited to separate types of layers, but can be applied to the whole CNN. Arithmetic precision can be reduced from floating-point to, e.g., 16-bit fixed point [12] with minimal degradation of CNN (classification) accuracy, or by extreme quantization down to two [17] bits or one bit [18][14] of weight precision. When the precision of weights (and possibly also input data) is reduced to a single bit, the CNN is *binarized*. Binarization dramatically reduces the memory footprint of a CNN, as the original weights, which are normally expressed in 32-bit floating point, can be represented with a single bit. This evidently has an impact on the CNN's accuracy [18]. However, besides shrinking the size of the

network, binarization also enables CNN inference on devices that have no support for floating-point arithmetic, such as microcontrollers and FPGAs [19].

This paper presents a CNN for vehicle image classification [20] that has been binarized including the weights of all layers, as well as the input data, following the principles of our recent work [14]. However, unlike our recent work that concentrated on CNN inference on graphics processing units, in this paper we focus on microcontroller-class devices that can be found on edge nodes of an IoT system. As the target microcontroller, we have selected PULPino [21], which is based on the open-source instruction-set architecture RISC-V [22], which is gaining interest in both academia and industry.

The contributions of this paper are as follows:

- Performance and memory footprint measurements of our binarized CNN-based image classifier on a RISC-V microcontroller, and
- Optimization of binarized CNN computations by the custom instruction 'popcount' found in a proposal for RISC-V instruction set extensions [23].

The structure of this paper is as follows: Section II introduces other works related to optimization of CNNs; Section III describes the PULPino microcontroller that we use as the target device for our image classifier; Section IV covers the structure and binarization process of our CNN; Section V presents our experimental results, and Section VI concludes the paper.

II. RELATED WORK

This section describes previous works related to acceleration of CNNs, some also considering acceleration by hardware. Table I present a summary of these works and what target platforms they consider.

TABLE I
RELATED NEURAL NETWORK OPTIMIZATION WORKS

Work	Type	Optimization	Platform
Courbariaux et al. [18]	SW only	Binarization (fc layers only)	NVidia GPU
Rastegari et al. [24]	SW only	Binarization (conv and fc layers)	64-bit CPU
Khan et al. [14]	SW only	Binarization (conv and fc layers)	NVidia and OpenCL GPUs
ESPRESSO [25]	SW only	Binarization (conv and fc layers)	NVidia GPU, CPU
Park et al. [26]	HW SW	Zero skipping, Data reuse (conv layers only)	Nvidia GPU, GPU simulation
Conti et al. [27]	HW SW	Binarization (conv and fc layers)	HW accelerator for MCUs
Proposed	HW SW	Binarization (conv and fc layers)	RISC-V MCU (simulation)

Binarized neural networks (BNN) were originally introduced in [18]: network weights and activations are restricted to $+1$ and -1, which enables replacing multiplications and additions with bit-wise operations. Experiments have been performed on MNIST and CIFAR-10 datasets. The authors demonstrate a speedup of $7\times$ for a multi-layer perceptron network trained for MNIST handwritten digit classification. Experimental results are limited to GPU acceleration of binarized fully-connected layers.

Somewhat later the binarization optimization was extended to the large-scale ImageNet image classification challenge [24]. The authors of [24] concentrate on CPU targets and report up to $58\times$ execution time reduction on 64-bit CPUs for binarized convolution and fully-connected layers. Also, the authors claim an accuracy improvement of 16% compared to [18] in the ImageNet top-1 classification challenge.

Our previous work [14] was among the first ones to present GPU acceleration of both binarized convolution and fully-connected layers. Experimental results are presented for two mobile GPUs (NVidia Jetson and ARM Mali-T860), as well as for a desktop GPU (NVidia GTX1080). Layer implementations have been written from scratch in OpenCL and CUDA and made available open source. Additionally, the accuracy impact of various input image binarization approaches are analyzed.

In [25] a self-contained library ESPRESSO for binarized neural networks is presented. The library provides layer implementations in C and CUDA for both CPU and NVidia GPU targets. ESPRESSO [25] uses an optimization called *unrolling* (similar to *im2col* used in our previous work [14] and the proposed work) for reshaping tensors prior to computing convolution.

Optimization of CNN convolution operations is studied in [26]. The authors have observed that Winograd convolutions can involve a high number of multiplications by zero, especially if weight pruning (see, e.g. [13]) has been applied. This redundancy is avoided by skipping zero weights by a software-only and by a hardware-assisted approach. Additionally, the authors present a data reuse approach for reducing the number of additions. Both optimizations target NVidia GPUs.

In [27] the XNOR Neural Engine (XNE) is presented, a hardware accelerator for binary neural networks to be closely coupled with an MCU (micro controller unit) system. The XNE is capable of executing both binarized convolutional and fully-connected layers. The authors provide post-layout results where the accelerator has been placed on the same chip and same clock domain with a RISC-V microcontroller that acts as the host processor for the accelerator.

The proposed work is similar to the work of Conti et al. [27] in the sense that both consider an IoT edge computing scenario, build on binarized CNNs, and consider RISC-V MCU cores. However, a substantial difference is that the XNE accelerator of [27] is a dedicated datapath for CNNs next to the MCU core, whereas our proposed solution builds on a basic microcontroller architecture with just one custom processor instruction ('popcount') for accelerating BNNs. Evidently, the specialized circuit of [27] can achieve much higher energy efficiency than our proposed solution, whereas our solution only requires a tiny modification to a basic RISC-V MCU system, and otherwise remains very generic and capable of accelerating other types of applications as well.

Fig. 1. From left to right, a 'bus', 'normal car', 'truck', and a 'van'.

III. THE PULPINO RISC-V PROCESSOR FOR IOT APPLICATIONS

RISC-V is an open source instruction set architecture (ISA) that is gaining interest in both academia and industry [22]. The ISA is open and standardized, such that it is free to use for both academia and industry. To promote adoption of the new ISA, another goal was to design a modern ISA: it is designed in a modular way by providing a small base instruction set with optional extensions. Additionally, certain instruction opcodes are reserved for custom extensions. This flexibility allows to design RISC-V processors that are customized for special workloads, which makes the ISA interesting for specialized IoT devices.

While the open standard is just referring to the ISA itself and not any micro-architecture, the community around RISC-V has provided many open-source cores. An important motivation for open hardware is security, especially with recent micro-architecture bugs Spectre and Meltdown appearing in popular media [28][29]. Kerckhoff's principle and a long history of research suggests that open systems provide certain advantages over closed systems in terms of security [30][31][32].

The Parallel Ultra-Low-Power (PULP) project has developed several RISC-V-based microcontrollers that are suitable for IoT applications [21]. The PULPino is particularly suited for low cost, low power tasks, because it is a simple in-order single-core microcontroller with many configuration options. Due to these advantages, the custom processor used in this work was derived from the PULPino-based SoC (System-on-Chip).

IV. NEURAL NETWORK DESIGN

A. Network for Vehicle Classification

The neural network model we use is that of the vehicle classifier network presented in [20]. The network has five layers in total, starting with two convolutional layers, each one with 32 output feature maps, and kernel sizes 5×5. Each of the convolutional layers is followed by a 2×2 maxpooling operation. The second convolutional layer is followed by three fully-connected layers. The first fully-connected layer (the 3rd layer in the network) has 100 neurons, resulting in the shape 24×24×32×100. The two layers that follow have shapes 100×100, and 100×4, in that order.

The dataset we use for training the network has 6555 photos of vehicles from four categories: *bus*, *normalcar*, *truck*, and *van*. Each vehicle image is a full-color image of size 96×96. Example images from each class in the data set are shown in Fig. 1. We split the data into a training set (80%), validation set (10%) and a test set (10%). Our test-set accuracy reports are the recorded accuracy reports that correspond to the best validation set accuracy.

B. Neural Network Binarization

We implement a binarized version of the vehicle classifier network introduced in [20] reducing the precision of CNN weights and their activations to 1-bit. This concept was first introduced in [18], with reports of substantial reductions of model execution time and size. In this work, we replace all ReLU activations in the network with the sign function, which is given as

$$sign(x) = \begin{cases} -1 & \text{if } x \leq 0 \\ +1 & \text{if } x > 0 \end{cases} \tag{1}$$

We binarize the weights of the network using the $sign$ function as well. During training, the gradient of *sign* activations are explicitly defined to be the identity function in the backward pass so that $\frac{\partial sign(x)}{\partial x} = x$. The full-precision version of the network (non-binarized) is trained using the RMSprop optimizer, and the binarized version is trained with the ADAM optimizer. For the binarized version of the network, only the binarized weights, where all have a value of either -1 or $+1$, are used for inference on the target device. The network is trained from scratch using binarization in a separate training process.

We use the terms *packing* or *bit-packing* to denote the encapsulation of an array of 1-bit values ($+1$'s and -1's) into one 32-bit unsigned integer. For example, if we wish to *pack* a vector $\mathbf{x} \in \{-1, +1\}^{32}$, its *packed* representation, x^p, is given by

$$x^p = \sum_{i=0}^{31} (x_i + 1) 2^{i-1}, \tag{2}$$

where x_i is the ith element of $\mathbf{x}$. This then allows operations such as vector-summations and dot products to be performed using binary (bit manipulation) operations. The dot-product, for example, can be represented as

$$\mathbf{a} \cdot \mathbf{b} = 32 - 2 \times \texttt{popcount}(\texttt{xor}(A, B)), \tag{3}$$

where both A and B are 32-bit unsigned integers holding the packed representations of the vectors $\mathbf{a}$, $\mathbf{b} \in \{-1, +1\}^{32}$. The operation 'popcount' (also known as Hamming weight calculation) is a function for computing the number of bits set to 1, which can essentially simulate vector summation. The operation `xor` in Eq. 3 is the bit-wise 'xor' operation.

C. Acceleration by Bit Manipulation Instructions

Looking at Eq. 3 we see that both 'xor' and 'popcount' are used in inference of binarized CNNs to perform an operation that emulates multiplication for packed weights; this means that both for fully-connected and convolutional layers 'xor' and 'popcount' are in heavy use and offer a clear optimization target.

The hardware implementation of 'xor' can be found on any programmable processor, whereas a hardware implementation for 'popcount' is mostly available on graphics processing units or CPU SIMD extensions such as ARM NEON. For our target processor, the PULPino microcontroller, the base ISA does not include 'popcount' – this instruction is only present in the *bit manipulation extension* of RISC-V that is still under development [23].

In our experiments, in cases where the target processor did not have a hardware instruction for 'popcount', the LLVM C language description[1] shown in Algorithm 1 was called through __builtin_popcount().

Algorithm 1 LLVM 'popcount', i.e. Hamming weight

```
int32 popcountsi2 (int32 a) {
  uint32 x = (uint32) a;
  x = x − ((x >> 1) & 0x55555555);
  x = ((x >> 2) & 0x33333333) + (x & 0x33333333);
  x = (x + (x >> 4)) & 0x0F0F0F0F;
  x = (x + (x >> 16));
  return (x + (x >> 8)) & 0x0000003F;
}
```

V. EXPERIMENTS

The experimental evaluation of this work consisted of two parts: 1) evaluating the effect of the software-based binarization optimization for our image classifier, and 2) evaluating the effect of the 'popcount' custom instruction on the binarized classifier. Unfortunately, as our ultimate target platform was the PULPino microcontroller for IoT devices, it was not possible to benchmark the original *non-binarized* vehicle classifier on this device as it has no hardware support for floating point computations. Hence it was necessary to use two different target platforms to complete our experiments, and these platforms are summarized in Table II.

The ARM Cortex A53 core is a powerful mobile processor and in our experiments the processor was used under Linux for benchmarking a C language implementation of the original vehicle classifier [20], as well as for the C language implementation of the binarized vehicle classifier.

Experiments on the PULPino microcontroller platform were performed in a simulation environment, which is described next.

A. The ETISS Simulator

The RISC-V ISA is still in a phase of development, as for example the specification is not officially standardized yet. Still, the central components of the specification have matured and have been used to fabricate various chips such as the SiFive FE310 SoC [33]. The application being evaluated in this work however requires the *bit manipulation instruction* extension ('B extension') of the RISC-V ISA. This extension is still in active development [23] and not part of the current specification. Therefore, there is no RISC-V chip available that could be used for evaluating our results, however an alternative way to estimate the performance gain achievable through custom instructions is by simulation.

An RTL (Register-Transfer Level) hardware simulation would not be suitable for fast prototyping as the micro-architecture should be modified to enable the execution of the chosen custom instructions. Additionally, for a time-consuming workload such as our CNN application, the RTL simulation time would be prohibitively high.

The Extensible Translating Instruction Set Simulator (ETISS) focuses on extensibility [34] to support fast prototyping. As ETISS already supports the standard RISC-V base instruction sets, contains a virtual prototype of the PULPino [21] SoC, and allows profiling the application execution time, the use of this simulator was a natural decision our binarized image classifier application.

B. Implementation of the Popcount Instruction

As the PULPino virtual prototype of ETISS currently only supports the RISC-V base ISA, a temporary modification of the virtual prototype was required to enable profiling with support for 'popcount'. From ETISS execution traces it was discovered that the 'xori' instruction of the RISC-V base ISA remained almost unused throughout the whole execution of the binarized vehicle classifier. Therefore, in the PULPino virtual prototype the functional description of 'xori' was modified to provide alternative functionality, i.e. 'popcount', toggled by the value of the 2nd instruction operand.

In the software implementation of the binarized vehicle classifier, the calls to 'popcount' were then replaced with inline assembly calls to 'xori' with the specific operand value that would invoke 'popcount' behavior.

C. Execution Time and Memory Footprint Analysis

Table III shows the experimental results for both A53 and PULPino. From top to bottom the table rows report execution time on A53, execution time on PULPino, data memory footprint, PULPino instruction memory footprint, and CNN classification accuracy.

Looking at the A53 results it can be seen that binarization alone reduced the execution time by more than 80%, and dropped the data memory usage close to 95% when compared to the original floating point C version.

Acceleration by the hardware 'popcount' instruction reduced the computation time of the binarized vehicle classifier by around 55% on the PULPino platform, and also reduced the instruction memory footprint by around 2 kB. The reason for the 55% reduction in execution time can be seen from Table IV that shows the count of executed instructions on the PULPino platform for the binarized vehicle classifier with and without the hardware 'popcount' instruction: the code version that calls the hardware 'popcount' instruction has respectively 55% less executed instructions. This is because if there is no hardware support for 'popcount', the functionality must be

[1] https://github.com/sifive/riscv-llvm/blob/master/compiler-rt/lib/builtins/popcountsi2.c

TABLE II
PLATFORMS USED FOR EXPERIMENTS.

Tag	CPU	Platform type	Compiler	Operating System
A53	ARM Cortex A53 (1416 MHz)	Silicon SoC	g++ 5.4.0	Linux Firefly 4.4
PULPino	PULPino (33 MHz)	Virtual prototype on ETISS	riscv32-unknown-elf-gcc 7.1.1	*n/a*

implemented by means of several regular instructions, which can be seen in increased execution counts of 'srli', 'and', 'sub' and 'add' instructions for the binarized version without the hardware 'popcount' instruction. Algorithm 1 shows that these instructions are needed for the software implementation of 'popcount'

The accuracy results shown in Table III are identical to our previous work on binarization that targeted graphics processing units [14].

TABLE III
EXECUTION TIME, MEMORY FOOTPRINT AND ACCURACY

Application version Arithmetic	Baseline float32	Binarized int32	Bin+pop int32
A53 Execution time	0.362 s	0.057 s	-
PULPino Exec. time	-	2.62 s	1.18 s
Data Memory	7.2 MB	369 kB	369 kB
Pulpino Instr. Memory	-	21 kB	19 kB
Accuracy [14]	97.09%	92.52%	92.52%

TABLE IV
NUMBER OF EXECUTED INSTRUCTIONS

Instruction name	Binarized int32	Bin+pop int32
lw	8797430	8797417
lbu	272	272
addi	6372539	6354083
slli	2801668	2801668
popcount/xori[2]	4	**3302052**
srli	**16510241**	1
srai	4	4
ori	1	1
andi	3302062	14
sb	268	268
sh	4	4
sw	782165	782109
add	**16704267**	3496013
mul	**0**	0
sub	**3670893**	368845
sll	18632	18632
slt	2553032	2553032
xor	3302048	3302048
or	2451656	2451656
and	**13208192**	0
bne	3232555	3232555
blt	0	0
bge	370058	370058
bltu	4	4
jalr	39	39
jal	57	57
csrrw	1	1
Total	***84078092***	***37830833***

VI. CONCLUSIONS

In this paper we have presented a convolutional neural network based vehicle image classifier that has been optimized for real-time execution and small memory footprint by a technique called *binarization*. We show that by using 'popcount', a custom instruction in our target processor, the runtime of the binarized image classifier can be reduced by 55%. This result is important due to the fact that 'popcount' has been proposed to be included to a standardized instruction set extension ('B extension') of the recently introduced open source RISC-V instruction set architecture. Besides RISC-V, 'popcount' is already supported in graphics processing units and e.g. in the NEON SIMD extension of ARM processors.

Our work shows that the software-based *binarization* transformation coupled with the hardware-based 'popcount' instruction yields an extremely powerful combination for optimizing inference of convolutional neural networks. Together, the memory footprint is reduced by close to 95%, and execution time is reduced by a magnitude while maintaining an acceptable loss in accuracy. As a results, image classification is performed in 1.18 seconds on the tiny 33 MHz RISC-V microcontroller that is well suited for IoT applications.

ACKNOWLEDGMENT

This work was partially funded by the Academy of Finland project 309903 CoEfNet, and by the ITEA3 project 16018 COMPACT (Business Finland diary number 3098/31/2017, German ministry of education and research reference number 01IS17028).

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE conference on computer vision and pattern recognition (CVPR)*, 2016, pp. 770–778.

[2] J. Dai, K. He, Y. Li, S. Ren, and J. Sun, "Instance-sensitive fully convolutional networks," in *European Conference on Computer Vision (ECCV)*. Springer, 2016, pp. 534–549.

[3] J. Johnson, A. Karpathy, and L. Fei-Fei, "DenseCap: Fully convolutional localization networks for dense captioning," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4565–4574.

[4] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *IEEE conference on computer vision and pattern recognition (CVPR)*, 2017, pp. 7263–7271.

[5] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[6] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[7] M. Alioto and M. Shahghasemi, "The Internet of Things on its edge: Trends toward its tipping point," *IEEE Consumer Electronics Magazine*, vol. 7, no. 1, pp. 77–87, 2018.

[2] 'popcount' implemented as 'xori' alternative behavior

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[9] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and <0.5 MB model size," *arXiv preprint arXiv:1602.07360*, 2016.

[10] J. Shen, Y. Huang, Z. Wang, Y. Qiao, M. Wen, and C. Zhang, "Towards a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2018, pp. 97–106.

[11] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *International Workshop on Frontiers in Handwriting Recognition*, 2006.

[12] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[13] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," in *International Conference on Learning Representations (ICLR) Workshops*, 2018.

[14] M. Khan, H. Huttunen, and J. Boutellier, "Binarized convolutional neural networks for efficient inference on GPUs," in *European Signal Processing Conference (EUSIPCO)*. IEEE, 2018, pp. 682–686.

[15] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," in *British Machine Vision Conference (BMVC)*. BMVA Press, 2014.

[16] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[17] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[18] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[19] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2017, pp. 65–74.

[20] H. Huttunen, F. S. Yancheshmeh, and K. Chen, "Car type recognition with deep neural networks," in *IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2016, pp. 1115–1120.

[21] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak, and L. Benini, "PULPino: A small single-core RISC-V SoC," in *RISC-V Workshop*, 2016.

[22] *The RISC-V Instruction Set Manual*, RISC-V Foundation, 2017, version 2.2.

[23] *RISC-V Bitmanip Extension*, Clifford Wolf, 2019, version 0.37.

[24] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision (ECCV)*. Springer, 2016, pp. 525–542.

[25] F. Pedersoli, G. Tzanetakis, and A. Tagliasacchi, "Espresso: Efficient forward propagation for binary deep neural networks," in *International Conference on Learning Representations (ICLR)*, 2018.

[26] H. Park, D. Kim, J. Ahn, and S. Yoo, "Zero and data reuse-aware fast convolution for deep neural networks on GPU," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2016, pp. 1–10.

[27] F. Conti, P. D. Schiavone, and L. Benini, "XNOR neural engine: A hardware accelerator IP for 21.6-fJ/op binary neural network inference," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2940–2951, 2018.

[28] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[29] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[30] J.-H. Hoepman and B. Jacobs, "Increased security through open source," *arXiv preprint arXiv:0801.3924*, 2008.

[31] B. Witten, C. Landwehr, and M. Caloyannides, "Does open source improve system security?" *IEEE Software*, vol. 18, no. 5, pp. 57–61, 2001.

[32] C. Cowan, "Software security for open-source systems," *IEEE Security & Privacy*, vol. 99, no. 1, pp. 38–45, 2003.

[33] *SiFive FE310-G000 Manual*, SiFive, Inc., 2017, version v2p3.

[34] D. Mueller-Gritschneder, M. Dittrich, M. Greim, K. Devarajegowda, W. Ecker, and U. Schlichtmann, "The extendable translating instruction set simulator (ETISS) interlinked with an MDA framework for fast RISC prototyping," in *International Symposium on Rapid System Prototyping (RSP)*. IEEE, 2017, pp. 79–84.

PUBLICATION
IV

Instruction Extension of a RISC-V Processor Modeled with IP-XACT

S. Payvar, E. Pekkarinen, R. Stahl, D. Mueller-Gritschneder and T. D. Hämäläinen

*2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*2019, 1–5
DOI: 10.1109/NORCHIP.2019.8906975

Instruction Extension of a RISC-V Processor Modeled with IP-XACT

Saman Payvar
Computing Sciences
Tampere University
Tampere, Finland
saman.payvar@tuni.fi

Esko Pekkarinen
Computing Sciences
Tampere University
Tampere, Finland
esko.pekkarinen@tuni.fi

Rafael Stahl
Chair of Electronic Design Automation
Technical University of Munich
Munich, Germany
r.stahl@tum.de

Daniel Mueller-Gritschneder
Chair of Electronic Design Automation
Technical University of Munich
Munich, Germany
daniel.mueller@tum.de

Timo D. Hämäläinen
Computing Sciences
Tampere University
Tampere, Finland
timo.hamalainen@tuni.fi

***Abstract*—Short time-to-market and cost consideration of hardware design promotes reuse of ever more complex intellectual property even up to processors. In processor design, the instruction set architecture (ISA) selection is a major design decision driven largely by application requirements. Extendable ISAs enable application-specific adjustments and improved performance at the cost of more complex design. Adding a custom instruction introduces a choice of either utilizing existing hardware or adding new dedicated hardware.**

This work presents an instruction extension flow for a RISC-V processor core modeled in IP-XACT. We demonstrate the workflow by adding three bit manipulation instructions "popcnt", "parity" and "bswap" in the instruction set that executes on an extended processor platform and evaluate their performance in simulation. The simulated instruction count and performance are used to evaluate the benefit of adding dedicated hardware.

The effort analysis of the design flow shows approximately 110 minutes work for adding a new instruction to the RISC-V core. This suggests a straightforward and easy to follow approach that can be extended to other instructions as well. In addition, we propose the work flow to cover adding dedicated hardware in IP-XACT for improved re-usability and design consistency.

***Index Terms*—RISC-V, GCC back end, ISA extension, bit manipulation, IP-XACT**

I. Introduction

The transistor shrinking trend of the CMOS technology introduces higher design complexity of System-on-Chips (SoCs) while competition pressures for short time-to-market at low cost. Modularity addresses the complexity by composing the system of reusable components and their connections. The components are often provided by different vendors which necessitates a standard exchange format to facilitate Intellectual Property (IP) reuse. The most promising is the IEEE 1685 standard (IP-XACT) [1] that is an XML format widely used in the industry for IP description and integration.

Beside increased complexity, transistor size reduction increases the leakage power and thus static power dissipation. Consequently, energy efficiency becomes much more significant. Hardware design energy efficiency is achieved by application specific adjustments which increase the performance and reduce the power consumption. In processor core design, ISA extensions and the hardware design adjustments are the two available options for application specific optimizations.

RISC-V [2] is an open source ISA whose availability, simplicity and adaptability has made it popular in both academia and industry. Studies like [3] motivate further research on the potential of non-standard RISC-V extensions. In this work, we consider adding three bit manipulation instructions to the *PULPino* [4] platform that is an open source implementation of a 32-bit RISC-V single-core microprocessor.

Modeling a processor in IP-XACT facilitates the configurations for different versions such as deciding the standard and non-standard ISA extensions in the hardware design flow. The software compiler must produce compatible binary code for the target hardware, extending the configurability to the compiler, which has lead to *retargetable* compilers [5]. To this end the ISA extensions should be captured in an early phase for consistency in both hardware and software design flows using the IP-XACT model as a single point of entry.

The contributions of this work are:

- Instruction extension design flow of RISC-V ISA modeled with IP-XACT,
- Evaluation of the extensions with RISC-V GCC compilation and simulation, and
- Effort analysis of the proposed design flow

The rest of the paper is organized as follows: Section II introduces the related works. Section III explains the RISC-V non-standard instructions extensions. Section IV depicts the experiments and shows the results. Section V concludes the study and discusses future work.

TABLE I
COMPILER EXTENSION STUDIES

Study	Platform	Compiler	Extension
Tagliavini et al. [3]	PULPino	GCC	SmallFloat SIMD
Sen et al. [8]	ARMv8	N.A.	SPARCE
Murray and Franke [9]	ENCORE	GCC	App. Specific
Sedaghati et al. [10]	x86-64	GCC & ICC	StVEC
Proposed	PULPino	GCC	Bit Manipulation

II. RELATED WORK

The RISC-V ISA has a modular structure where the base ISA covers the minimal instructions and the standard instruction extensions add options for extra functionality. At the time of writing this paper, some standard extensions like bit manipulation [6] and vector extensions [7] are still a work-in-progress. Additionally, the ISA allows adding non-standard extensions, and thus custom instructions, and promising studies and proposals have already emerged. In this section, we compare the current ISA extension works summarized in Table I to our experiments. For the comparison we have considered the platform, the compiler, and the implemented extensions.

A. RISC-V ISA Extensions

The small floating point types for RISC-V ISA study [3] presents extensions for 16-bit and 8-bit floating point types. The extensions are implemented for register size of 32-bit for small floating point (FP) data types. The experiments are performed on RISCY core and show 1.64 times performance boost and 30% energy saving for 16-bits and 2.18 times execution time increase and 50% energy reduction for 8-bits.

The sparsity extension work [8] is targeted for deep neural network (DNN) execution on general purpose processors. The sparsity aware general purpose core extensions dynamically track registers with zero value and prevents fetching redundant instructions. The experiments on image recognition DNNs shows up to 31% performance improvements.

Compiler support for automatic instruction set extension study [9] applies the GCC extension in the middle-end for the ENCORE which is a customizable application specific instruction-set processor. They have reported an average performance improvement of 1.26 for experiments on 179 benchmarks.

The vector instruction extension study [10] focuses on addressing mode of vector instructions for stencil computations. The overhead estimations is considered with a hardware implementation and depicted with the optimistic and the pessimistic simulations. The experiments are performed on four x86-64 platforms using GCC and ICC compiler vectorization options. They reported performance improvement between 20% and 2.47x with optimistic instruction emulation and between 7% and 2.26x with pessimistic instruction emulation for both GCC and ICC.

B. IP-XACT Extensions

Inherently IP-XACT is extendable by vendor extension that contain supplementary information. Extendability is mandatory for wide applicability and industrial acceptance. However, extensive use of vendor extensions will result in exactly what the standard was originally intended to avert, vendor-dependence. In the following we consider proposals for vendor extensions that are not only relevant for this work, but also considered broadly applicable.

The IP-XACT extension targeted for smart systems [11] covers power, temperature and reliability concerns. All standard IP-XACT components are identified by a Vendor, Library, Name, Version (VLNV) 4-tuple, so an optional C tag is proposed to distinguish different concerns resulting in VLNVC identifiers. A smart system is modeled as multiple communicating views where each presents an extra-functional (i.e. non-functional) concern. The IP-XACT views are used for SystemC code generation where instances of all the views are instantiated at the top level. The influence of all concerns of a system are now covered when all views are simultaneously simulated.

IP-XACT is originally intended for hardware description but could be used for both hardware and software data inclusion. In [12], a methodology for both hardware and software IP reuse is proposed for IP-XACT based FPGA design flow. This approach facilitates board support package and software file creations, and results in reducing the design time to one third while doubling automation. In [13] IP-XACT extensions are proposed for covering hardware dependent software.

C. RISC-V IP-XACT Model

The PULPino platform SystemVerilog (SV) implementation is considered for creating IP-XACT models from the SV source code repositories in [14]. The modeling is applied by using the Kactus2 [15] tool to automatically import the modules and further refined manually by the designer by e.g. defining the memory maps. The resulting model covers all four RISC-V configuration options of the PULPino platform and provides the basis for configuration automation which simplifies the rest of the design flow for the different processor variants.

The related works are presented in three subsections covering three aspects of this work. The reported results of RISC-V extension studies promote further investigations of our RISC-V bit manipulation instruction extension work. The IP-XACT studies show that it can be successfully applied to many design topic and promote our core expansion study. The IP-XACT model of RISC-V study facilities our RISC-V core extension explorations.

III. RISC-V INSTRUCTION EXTENSION DESIGN FLOW

The design flow utilized in this study for adding the non-standard instructions to the RISC-V ISA is presented in Fig. 1. This flow has eight levels from top down depicting the stages from inferring the bit pattern of instructions until running the compiled binary on a processor implementation on an FPGA.

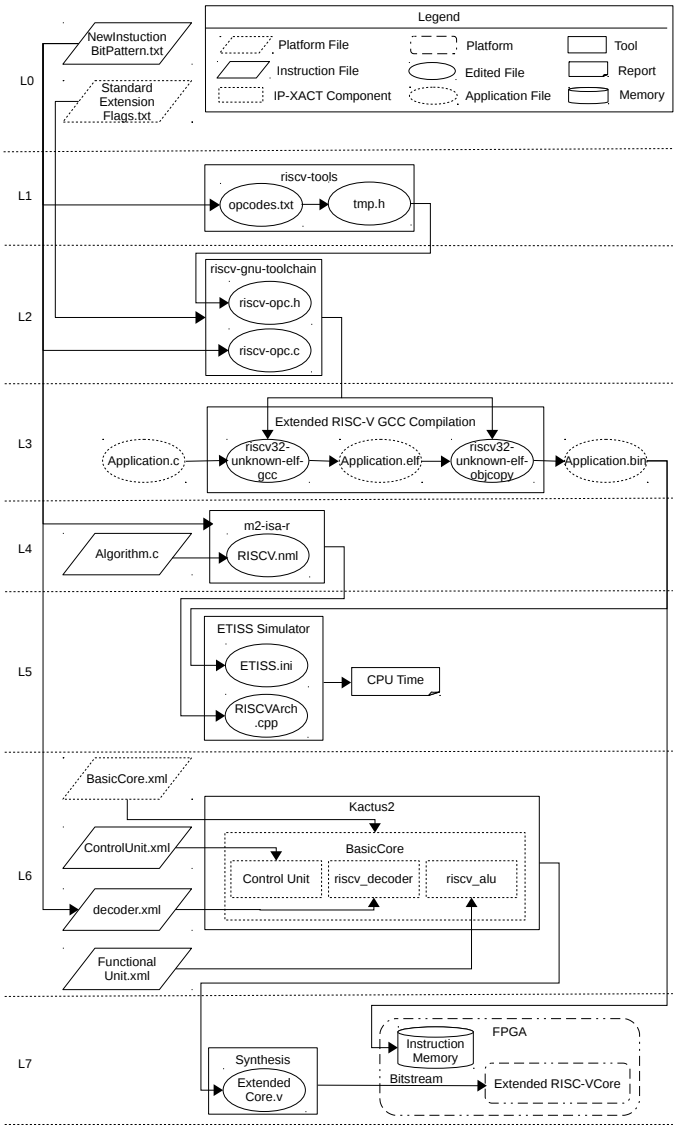

Fig. 1. The proposed design flow of RISC-V instruction extensions targeting simulation and prototyping on FPGA.

A. Standard ISA Selection and Custom Instructions Inclusion

The basic RISC-V core with its standard extensions is determined at the initial stage at level L0 in Fig. 1. The command line flags i.e. *with-arch* and *with-abi* for the compiler generation by *riscv-gnu-toolchain* are stored in a text file. Beside the standard extensions, the bit patterns of the targeted non-standard instructions are stored in another text file using the same format as in the *opcodes.txt* file of *riscv-tools* on L1.

B. RISC-V GCC Compiler Extension

The RISC-V GCC back-end can be edited to cover non-standard RISC-V instructions. Utilizing inline assembly for new instructions requires only minor modification to GCC binutils. Alternatively, the instruction pattern could be added to the automatic code generation in the compiler back-end and the compiler would attempt to utilize it whenever appropriate. However, the latter is vastly more complex requiring in-depth knowledge of the optimizer internals and thus considered outside the scope of this paper.

First, the RISC-V opcodes header file should be edited to include a new instruction's *MATCH* and *MASK* definitions and its *DECLARE_INSN* macro declaration in *riscv-opc.h*. The *MATCH* defines the binary equivalent of an instruction with zeros for the registers while the *MASK* defines a pattern for detecting the instruction. The *MATCH* pattern must be unique for each instruction, so addition of new instruction's *MASK* requires overlap checking which can be performed by tools like *riscv-tools* [16]. This first stage is shown as level L1 where the instruction specification i.e. the binary equivalent of each field of the instruction is considered as input from level L0. The output of level L1 is a temporary header *tmp.h* containing the instructions in compatible format with *riscv-opc.h*.

Second, the instruction format must be added to the *riscv_opcodes* structure in *riscv-opc.c* file. The format includes data like the instruction's name, type and registers which are inferred from the new instruction's bit pattern text file. Also, the temporary header file *tmp.h* should be applied to *riscv-opc.h*. Then, the RISCV GCC compiler is built with appropriate platform specification data i.e. *with-arch* and *with-abi* configurations received from L0. The second stage is demonstrated as level L2 where the new instruction bit pattern and standard extension flags generated in L0 are considered for header and C file modifications and the compiler build.

Third, the Extended RISC-V GCC compiler is used to compile the application C code to RISC-V assembly. The ETISS [17] project provides a *Makefile* template for compiling the C code using the previously built RISC-V GCC compiler which now recognizes the new instruction's inline assembly. Beside the C code, the *Makefile* accesses the necessary libraries for providing the platform i.e. PULPino compatible binary file. The third stage is demonstrated as level L3 where the C code input for the RISCV GCC compiler is presented on the left and the binary output file on the right.

C. ETISS Simulator Extension

As we are exploring non-standard instructions, no hardware in the market supports them out-of-the-box. Consequently, the ETISS simulator is selected for performance analysis of the studied instructions. ETISS is an ISA-independent Instruction Set Simulator (ISS) based on Dynamic Binary Translation (DBT). ETISS can be used to estimate CPU time of the compiled C code. The simulation is used to evaluate the performance of the processor using base instructions against using custom instructions to evaluate whether the performance gain justifies adding dedicated hardware.

ETISS can support different ISAs using plugins. It already provides a plugin for the basic RISC-V ISA. To extend the RISC-V ISA, the new instruction must be supported by the DBT of the ETISS simulator. This is conveniently done using the *m2-isa-r* tool, which is an Eclipse Modeling Framework (EMF) application that can generate plugins for different ISAs using a model-based flow based on nML description. nML

is a modeling language for compact ISA description. *m2-isa-r* reads the nml file *RISCV.nml* and supplies the ETISS plugin file *RiscvArch.cpp*. The plugin file contains an *InstructionDefinition* for each available instruction and defines the binary encoding and the instruction behavior which enables DBT. The nml file *RISCV.nml* is extended according to the *Algorithm.c* and the new instructions bit pattern text files. The RISC-V plugin *RiscvArch.cpp*, which is extended with the new instructions, is generated using the model-based flow resulting in ETISS supporting the new instructions. The fourth stage is demonstrated as level L4.

In a second step, the ETISS simulator is configured for estimating the performance. The ETISS simulator contains a shell script which accesses the initialization configurations file i.e. *ETISS.ini*. Beside the simulator configuration options, the path to the compiled binary is given in this file. ETISS outputs the text prints of the code and reports CPU time, simulation time, CPU Cycles and MIPS estimation. This is shown as level L5 where the ETISS simulator receives the binary and provides the CPU time report. The experiments in Section IV cover levels L0 to L5 of the presented design flow.

D. Hardware Support

Executing the new instructions is possible by either utilizing existing instructions i.e. a function call to equivalent algorithm or by adding dedicated hardware functional unit (FU) for it in the pipeline. The the latter is presented in the work flow as L6 and L7 where the FU of the new instruction is added at the IP-XACT model level.

First, the IP-XACT model of the selected basic core is packaged in the Kactus2 tool. In the case of PULPino, this is already done. Next, the FU of the new instruction is added in the pipeline and the control unit of the core is replaced with a compatible implementation. In PULPino the FU is added into the ALU module *riscv_alu* and wired to the ALU output selection multiplexer. Then, the module *riscv_decoder* is replaced with one recognizing the new instruction bit pattern and driving the control signals for ALU and optionally the rest of the core. After these editions, the basic core covers the extensions.

Second, the RTL code for the extended processor is automatically created using the generators in Kactus2. The RTL is synthesized using the chosen synthesis tool for the target FPGA platform. Then, the FPGA is programmed with the extended core including the FU of the new instruction. Finally, the binary of the compiled software code which includes the new instructions is loaded into the instruction memory for execution. In Fig. 1 the synthesis and FPGA run stage is depicted as L7.

IV. Experiments

We have studied three non-standard RISC-V bit manipulation instructions including *popcnt*, *parity*, and *bswap*, and their impact on the performance.

A. Algorithms of Instructions

We have compared the selected instruction's *LLVM* algorithm implementation [1] to their inline assembly equivalents. Population count algorithm returns the number of one bits in a given value as presented in Algorithm 1. Parity algorithm returns one if the number of ones is odd. Byte swap algorithm performs conversion between big-endian and little-endian.

Theses algorithms are compiled to several existing RISC-V ISA instructions. Consequently, the extended equivalent as only one instruction results in an obvious smaller instruction memory footprint. The disassembler of the *RISC-V GCC* compiler shows 14 *RISC-V* assembly instructions for populations count algorithm, 8 *RISC-V* assembly instructions for parity algorithm and, 6 *RISC-V* assembly instructions for byte swap algorithm. Due to coherency only the population count algorithm is presented. Algorithm 2 depicts disassembly of Algorithm 1 that all together are equivalent to one *popcnt* instruction. For example, the first instruction of Algorithm 1 at line number 2 is compiled to three instructions of *srli*, *and* and, *sub* in the first three lines of Algorithm 2.

Algorithm 1 Population Count

```
1: unsigned int popcount (unsigned int x) {
2:   x = x − ((x >> 1) & 0x55555555);
3:   x = ((x >> 2) & 0x33333333) + (x & 0x33333333);
4:   x = (x + (x >> 4)) & 0x0F0F0F0F;
5:   x = (x + (x >> 16));
6:   return (x + (x >> 8)) & 0x0000003F;
7: }
```

Algorithm 2 Population Count Assembly

```
1:  srli a5, s0, 0x1
2:  and a5, a5, s5
3:  sub a5, s0, a5
4:  srli a1, a5, 0x2
5:  and a1, a1, s2
6:  and a5, a5, s2
7:  add a5, a1, a5
8:  srli a1, a5, 0x4
9:  add a1, a1, a5
10: and a1, a1, s4
11: srli a5, a1, 0x10
12: add a1, a1, a5
13: srli a5, a1, 0x8
14: add a1, a5, a1
```

B. Results and Discussion

Table II shows the *CPU Time* results i.e. the output report of the *ETISS* simulator for running a *for loop* iterating one million times for one function call (see L5 in Fig. 1). The function has three variations i.e. the built-in function, the *LLVM* algorithm, and the inline assembly of the new instruction presented in

[1] https://github.com/sifive/riscv-llvm/tree/master/compiler-rt/lib/builtins

TABLE II
BIT MANIPULATION INSTRUCTIONS PERFORMANCE

Instruction	Built-in A.	Algorithm	Extension	Reduction
popcnt	8.8312 s	8.5187 s	8.0812 s	5.13 %
parity	8.56768 s	8.34893 s	8.09893 s	2.99 %
bswap	8.56493 s	8.25243 s	8.06493 s	2.27 %

TABLE III
NEW INSTRUCTIONS EFFORT ESTIMATION

Effort	L1	L2	L3	L4	L5	Total
Edit	30 min	30 min	10 min	30 min	10 min	110 min

Table II in the second, third and fourth column respectively. The fifth column depicts the performance improvement of the inline assembly compared to the *LLVM* algorithm. The simulation shows up to 5% improvement for new instructions extension in comparison to the *LLVM* algorithms. These results justify the implementation of the expanded *RISC-V* processors.

Table III shows the effort estimation for adding a new instruction according to the presented design flow. These estimations are done with the assumption of already installed and available tools for a first time users who is familiar with compilers and simulators. The estimations show the time required for modifying the files. On L1, one should be familiar with *RISC-V ISA* and instructions bit pattern as each instruction should have a unique bit combination. For L2, one should consider the instructions pattern and should know the targeted *RISC-V* platform. On L3 the only required modification is the Makefile of the *ETISS* compiler to include the new compiler path. For L4 the *RISCV.nml* file should be modified to include the algorithm of the new instruction. Consequently, the *RISCVArch.cpp* file is created automatically. On L5 the *RISCVArch.cpp* file is applied to the *ETISS* tool chain and after re-installation of both the *GCC RISC-V* compiler and the *ETISS RISC-V* simulator, the newly added instruction is recognized by compiler.

V. CONCLUSION AND FUTURE WORK

Utilizing IP-XACT format as higher level of hierarchy for RISC-V ISA implementation facilitates the creation of different variations of the RISC-V ISA extensions. The variations of the processor implementations forces the compiler and simulator to adjustment accordingly. In this study, we have presented our RISC-V instruction extension design flow and analyzed its utilization effort. Also, we have considered the bit manipulation extension of the RISCV ISA for the PULPino platform and shown how to modify the GCC RISCV compiler accordingly. We have applied the necessary changes to the ETISS RISC-V simulator and experimented with the three instructions. We have compared our implementations with the equivalent algorithms and presented the performance results.

The future work will be to synthesize the extended RISC-V core for FPGA using the different configurations and to run the compiled RISC-V binaries that platform to verify the improvements predicted with simulation.

VI. ACKNOWLEDGMENT

This work is partially supported by the ITEA3 project 16018 COMPACT (Business Finland diary number 3098/31/2017, German ministry of education and research reference number 01IS17028).

REFERENCES

[1] IEEE, "IEEE standard for IP-XACT, standard structure for packaging, integrating, and reusing ip within tool flows," *IEEE Std 1685-2014 (Revision of IEEE Std 1685-2009)*, pp. 1–510, Sep. 2014.

[2] *The RISC-V Instruction Set Manual*, RISC-V Foundation, 2017, version 2.2.

[3] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "Design and evaluation of smallfloat simd extensions to the risc-v isa," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 654–657.

[4] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak, and L. Benini, "PULPino: A small single-core RISC-V SoC," in *RISC-V Workshop*, 2016.

[5] L. Ghica and N. Tapus, "Optimized retargetable compiler for embedded processors-gcc vs llvm," in *2015 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE, 2015, pp. 103–108.

[6] *RISC-V Bitmanip Extension*, Clifford Wolf, 2019, version 0.90.

[7] *RISC-V Vector Extension*, Andrew Waterman, 2019, version 0.7.1.

[8] S. Sen, S. Jain, S. Venkataramani, and A. Raghunathan, "Sparce: Sparsity aware general-purpose core extensions to accelerate deep neural networks," *IEEE Transactions on Computers*, vol. 68, no. 6, pp. 912–925, 2018.

[9] A. Murray and B. Franke, "Compiling for automatically generated instruction set extensions," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 13–22.

[10] N. Sedaghati, R. Thomas, L.-N. Pouchet, R. Teodorescu, and P. Sadayappan, "Stvec: A vector instruction extension for high performance stencil computation," in *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2011, pp. 276–287.

[11] S. Vinco, M. Lora, E. Macii, and M. Poncino, "Ip-xact for smart systems design: extensions for the integration of functional and extra-functional models," in *2016 Forum on Specification and Design Languages (FDL)*. IEEE, 2016, pp. 1–8.

[12] A. Kamppi, L. Matilainen, J.-M. Määttä, E. Salminen, and T. D. Hämäläinen, "Extending ip-xact to embedded system hw/sw integration," in *2013 International Symposium on System on Chip (SoC)*. IEEE, 2013, pp. 1–8.

[13] F. Herrera, H. Posadas, E. Villar, and D. Calvo, "Enhanced ip-xact platform descriptions for automatic generation from uml/marte of fast performance models for dse," in *2012 15th Euromicro Conference on Digital System Design*. IEEE, 2012, pp. 692–699.

[14] E. Pekkarinen and T. D. Hämäläinen, "Modeling risc-v processor in ip-xact," in *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 2018, pp. 140–147.

[15] A. Kamppi, E. Pekkarinen, J. Virtanen, J.-M. Määttä, J. Järvinen, L. Matilainen, M. Teuho, and T. D. Hämäläinen, "Kactus2: A graphical eda tool built on the ip-xact standard," *The Journal of Open Source Software*, vol. 2, no. 13, p. 151, 5 2017. [Online]. Available: http://dx.doi.org/10.21105/joss.00151

[16] "RISCV Tools," https://github.com/riscv/riscv-tools.

[17] D. Mueller-Gritschneder, M. Dittrich, M. Greim, K. Devarajegowda, W. Ecker, and U. Schlichtmann, "The extendable translating instruction set simulator (ETISS) interlinked with an MDA framework for fast RISC prototyping," in *International Symposium on Rapid System Prototyping (RSP)*. IEEE, 2017, pp. 79–84.