SIMDIFY: FRAMEWORK FOR APPLICATION SPECIFIC SIMD-PROCESSING WITH RISC-V SCALAR INSTRUCTION SET

by

Mehmet Alp Şarkışla

B.S., Electrical & Electronic Engineering, Boğaziçi University, 2017

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Systems & Control Engineering

Boğaziçi University

2021

ACKNOWLEDGEMENTS

I would like to thank Prof. Dr. Arda Yurdakul, who guided me throughout development of my thesis. I am also grateful to my thesis committee members for their valuable time and effort to judge my thesis.

I am also thankful to my RISC-V group-mates, namely İbrahim Taştan, Ömer Faruk Irmak and Fatih Aşağıdağ, together we developed the basics of our RISC-V core. I also would like to thank Ömer Faruk Irmak for his support in machine code generation using compilers. I am also thankful to Anılcan Çakır for all the help on the way.

I would like to express my sincere gratitude to my company, TUBITAK Bilgem. My lab, TUTEL, has supported me and kept me going throughout my thesis.

Finally, I am fully indebted to my family and my friends for helping me survive all the stress from this study.

This work is supported by the Turkish Ministry of Science, Industry and Technology under Grant No. 58135. This thesis is published under M. A. Şarkışla and A. Yurdakul, "SIMDify: Framework for SIMD-Processing with RISC-V Scalar Instruction Set," 19th Australasian Symposium on Parallel and Distributed Computing (AusPDC'21) (in Australasian Computer Science Week Multiconference (ACSW '21)), 1 - 7, February 2021, DOI: 10.1145/3437378.3444364. Core parts of this thesis consists of contents from this publication. Agreement with the publisher company, Association for Computing Machinery (ACM), on copyright is established, and the relevant copyright document can be found in the appendix.

ABSTRACT

SIMDIFY: FRAMEWORK FOR APPLICATION SPECIFIC SIMD-PROCESSING WITH RISC-V SCALAR INSTRUCTION SET

Most of the hardware accelerators communicate with the processor via custom instructions. Since custom instructions are not standardized, each accelerator requires a different compiler and user code, which can be a tedious process for the user. To reduce the user burden, we propose a parallel programming framework called SIMDify, which generates single-instruction-multiple-data (SIMD) processors that can achieve SIMD processing without using custom instructions.

SIMDify takes an application machine code compiled for scalar RISC-V ISA and simulates it to determine the SIMD processing regions. Then, SIMDify configures and generates the application-specific SIMD processor that executes scalar RISC-V instructions concurrently on the SIMD datapath. SIMD processor consists of a single master and multiple slave processing elements (PE). Slaves focus on SIMD level tasks, whereas the master is responsible for the central control. Proposed architecture is the first SIMD capable RISC-V processor designed in HLS and can operate with a faster clock frequency than the existing SISD RISC-V HLS cores. SIMDify relieves the user from using custom instructions with rigid programming models and offers a flexible solution. The processor is designed and tested in Vivado High Level Synthesis 19.2. It operates at 78 MHz on Zynq Zedboard FPGA. Master PE uses 5% and each slave uses 3.5% of FPGA resources. Test results show that execution time can be improved by 8.5x with 9 slaves and 19x with 29 slaves.

ÖZET

SIMDIFY: RISC-V SKALER KOMUT SETİ İLE UYGULAMAYA ÖZEL SIMD İŞLEME İSKELETİ

Donanım olarak tasarlanmış hızlandırıcıların büyük bir kısmı, işlemciyle özel komutlar aracılığıyla iletişim kurar. Özel talimatlar standart olmadığından, her hızlandırıcı farklı bir derleyici ve kullanıcı kodu gerektirir ve bu da kullanıcı için zorlu bir süreç olabilir. Kullanıcı yükünü azaltmak amacıyla, tek komut çoklu veri (SIMD) komutlarını kullanmadan SIMD işlemcileri üreten SIMDify adlı paralel bir programlama çerçevesi sunuyoruz.

SIMDify, skaler RISC-V komut kümesi mimarisi (ISA) için derlenen makine kodunu alır ve SIMD işleme bölgelerini belirlemek için simüle eder. Ardından, SIMD veri yolunda skaler RISC-V komutlarını eşzamanlı yürüten ve uygulamaya özel olan SIMD işlemcisini yapılandırır. Üretilen SIMD işlemcisi, bir ana ve birden çok köle işlem öğesinden oluşur. Köleler, SIMD işlemlerine odaklanırken, ana işlem öğesi kontrolden sorumludur. Önerilen mimari, yüksek seviyeli sentez (HLS) araçlarında tasarlanan ilk SIMD özellikli RISC-V işlemcisidir. Mimarinin mevcut tek komut tekli veri (SISD) RISC-V HLS çekirdeklerinden daha hızlı bir frekansta çalıştığı gösterilmiştir. SIMDify, kullanıcıyı esnek olmayan programlama modelleriyle özel komutları kullanmaktan kurtarır ve esnek bir çözüm sunar. İşlemci Vivado HLS 19.2'de tasarlanmış ve test edilmiştir. Zynq Zedboard alanda programlanabilir kapı dizisi (FPGA) üzerinde 78 MHz'de çalışır. Ana öğe, FPGA kaynaklarının %5 ini kullanır ve her köle kaynak kullanımını %3,5 arttırır. Test sonuçları, işlem süresinin 9 köle ile 8,5 kat ve 29 köle ile 19 kat hızlanabileceğini göstermektedir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

Lat_n	Latency when Unroll Factor is equal to n
$Lat_{parallel}$	Latency of the parallelized portion in the algorithm
Lat_{serial}	Latency of the non-parallelized portion in the algorithm
n	Unroll Factor

LIST OF ACRONYMS/ABBREVIATIONS

ANN	Artificial Neural Networks
ASIC	Application Specific Integrated Circuits
ASIP	Application Specific Instruction-Set Processors
BRAM	Block Ram
BTA	Branch Target Address
CPU	Central Processing Units
DCT	Discrete Cosine Transform
DSP	Digital Signal Processing
EPI	European Processor Initiative
FF	Flip-flop
FIRTL	Flexible Internal Representation for RTL
FPGA	Field Programmable Gate Arrays
GCC	GNU Compiler Collection
GDB	GNU Project Debugger
GNU	GNU's Not Unix!
GPP	General Purpose Processor
HDL	Hardware Description Language
HLS	High-Level Synthesis
HPC	High-Performance Computing
IDE	Integrated Development Environment
IP	Intellectual Property
ISA	Instruction Set Architecture
ISS	Instruction Set Simulation
IoT	Internet of Things
JIT	Just-in-time
KMN	K-means clustering
KNQ	k-nearest neighbors with qsort
KNS	k-nearest neighbors with selective sort

LUT	Look Up Table
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MMM	Matrix Matrix Multiplication
MPI	Message Passing Interface
MVM	Matrix Vector Multiplication
OpenMP	Open Multi-Processing
PC	Program Counter
PE	Processing Element
RF	Register File
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
ReLU	Rectified Linear Unit
SAD	Sum of Absolute Distances
SDR	Software-defined Radio
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SSD	Sum of Squared Distances
VCS	Verilog Compiler and Simulator
VHDL	VHSIC-HDL, Very High-Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word

1. INTRODUCTION

A processor is the main component of the most digital electronic systems. Device designers use different types of processors depending on their area, power and cost constraints. For instance, due to their flexible and low-cost nature, general purpose processors or central processing units (CPUs) are more suitable to be used in tasks with multiple applications. General purpose processors trade computer power and flexibility with energy consumption and chip area. However, for processors that are used in single type of applications, such as neural-networks [3], cryptography [4], implants [5], wearables [6] and, Internet of Things (IoT) [7] area, power, cost and performance efficient processors are required.

In terms of area and performance, best pick is to use application-specific integrated circuits (ASICs). ASICs, as the name suggests, are devices that are created with a specific purpose in mind. They are not re-programmable and can only be used for a specific task. ASICs are quite expensive in terms of cost, resources and design time, but they do offer incredible high performance and low power consumption.

In recent years the main attention has been optimizing general-purpose processors for a given application domain to make them more efficient [8]. These optimized processors are called as application specific instruction set processors (ASIPs). An ASIP utilizes special properties of applications to accommodate the desired cost, power, area and performance requirements. ASIPs are in the middle of CPU and ASIC approaches: they can be programmed with a high-level language, the software can easily be modified if a bug is found, and yet the custom instructions can accelerate the application's performance far beyond the level of a general-purpose processor at a much lower power budget.

With the adoption of ASIP design, area-efficient [9, 10], power-efficient [11–14] and performance-efficient [9, 10, 15–17] processors have been designed. Optimizing the area is done by reducing its number of registers and functional units which removes unnecessary instructions and reduces the chip area. Optimizing performance is done by adding custom instructions or hardware to accelerate execution. One frequent way to enhance performance-efficiency of ASIP is to exploit inherent data parallelism in the algorithms and execute them concurrently.

High-Performance Computing (HPC) is another area that benefits from ASIP design. HPC is used in various fields such as weather modelling [18], physics [19], and biomedical modelling [20]. HPC algorithms make use of inherent data parallelism of the algorithms to increase performance. However, processors cannot automatically recognize this parallelism. Programmers must use Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) standards to guide compiler and processors to execute in an efficient manner. On top of that, the task must be parsed in a way that there will be minimum or no data dependency between each sub-process. For instance, in weather modeling, map is parsed in grids where data dependency only occurs in the edges. Each grid can then be executed in similar parallel computations. Concurrently executing the same operation on multiple data groups is called SIMD (Single instruction, multiple data) processing.

For hardware, SIMD instructions are inherently simple to implement, since they only require the duplicated structure of the main execution unit's datapath. But not all processors have built-in instructions for SIMD processing. Traditional approaches to this problem are solved by extending standard instruction set with non-standard custom instructions (compiler retargeting) [21] or using Just-in-time (JIT) compilers [22], both of which requires a non-standard compiler as well as non-standard instructions in the custom hardware. Since custom instructions are not standardized, each individual accelerator requires a different compiler modification. On top of the compiler modification, to properly introduce new instructions, simulators and debuggers must be additionally retargeted.

Accelerators are used in various fields such as machine learning [23], speech recognition [24], raw data processing [25], cryptography [26] and image detection and recognition [27] especially after the rise of IoT. Though designed accelerators may extremely speed up the execution, using them with custom instructions and compilers is a tedious process that discourages software programmers from using these accelerators [28]. SIMDify offers a flexible parallel processing solution that reduces the user burden and removes the custom instructions.

In this thesis, we present SIMDify [29], an open-source hardware-software parallelization framework to design special purpose SIMD processors without using any just-in-time compilation, extending the default instruction set or retargeting the compiler. SIMDify takes an application machine code compiled for scalar core and SIMD parameters, and generates a customization header files. Using Vivado High-Level Synthesis (HLS) [30], SIMDify processes the generated header files and automatically synthesizes the desired SIMD capable special purpose processor architecture. Processor is compatible with the RISC-V Instruction Set Architecture (ISA), and executes the native instruction set even during SIMD execution. The main contributions of this thesis can be summarized as follows:

- A flexible parallel programming framework called SIMDify for generating, customizing and scaling SIMD capable processors with minimal software level modification and using a standard compiler. To generate the special purpose processor, user only requires to write an algorithm in C, and compile it in RISC-V compiler. SIMDify will generate custom header files for the HLS, then synthesizes the SIMD soft processor that accelerates the given algorithm. Generated processor then can be mapped to an FPGA.
- A new RISC-V soft processor architecture that enables in-memory SIMD processing is proposed. Generated processor is the first SIMD capable RISC-V core designed using HLS. Processor can achieve similar frequency with other HLS generated RISC-V cores even with 30 slaves, and it can execute applications as SIMD by using only the base RISC-V ISA without modifying the existing compiler.

Traditional ASIP design flow has five key steps according to [31]. First two is analysing and design space exploration of the architecture. Third step is the extending the instruction set. SIMDify does not extend the existing instruction set and we leave the application analysis part to the user. Other two steps (code and hardware synthesis) is automated by the SIMDify. In this project, our aim is automating ASIP processor design using Vivado High Level Synthesis [32] tool and to make the ASIP design process much easier and efficient. User codes and compiles the task in C language, and SIMDify handles the rest. Applicability of the SIMDify is tested on selected algorithms. Clock speed, area and performance-efficiency of the generated soft-processors are studied for Zynq Zedboard FPGA [33].

The rest of the thesis is organized as follows. The second chapter discusses SIMD processing, RISC-V, and HLS related works. In third and fourth chapters proposed hardware-software system is introduced. In Chapter 5, detailed experimental analysis on resource usage and performance is given. The last chapter summarizes our work and broadly provides outputs of the study.

2. RELATED WORKS

In this chapter, we will set forth related research under three sections. In first section we discuss the application specific processors that utilizes SIMD processing. In second section High Level Synthesis is explained. In the last section RISC-V ISA that we use in our design is explained.

2.1. Single Instruction Multiple Data Application Specific Processor Design

Computer Architecture consists of four widely accepted main classifications based upon the number of concurrent stream of instruction and data available [34]. Single Instruction Single Data (SISD) architectures are sequential in nature and does not exploit any parallelism. In SIMD architectures, single instruction operates on multiple data at the same time. SIMD architectures exploit inherent data parallelism of the algorithms and they require minimum or no data dependency between its data streams to be effective. In conventional SIMD, SIMD instructions are used to inform processor about when to use SIMD processing. If the SIMD instructions are not standardized in the architecture, users must introduce SIMD instructions by themselves. This can be done by either using inline assembly or modifying the compiler toolchain. SIMDify does not force users which compiler to use. Intermediate representation generated by the compiler front end can also be used to detect SIMDifiable regions with cost of forcing users to a custom compiler.

In Just-in-time (JIT), compiling binary instructions to be executed by processors are interrupted, dynamically interpreted, and modified [22]. Instructions are modified ahead of time to introduce minimum overhead and modified to best suit the custom processor. JIT compiling is mainly used to translate bytecodes of high-level languages to custom SIMD instructions, and overhead is not entirely avoidable.

In the literature, SIMD computation is achieved using custom instructions in [35–37] by extending RISC-V ISA, and in [38] by extending SimpleRISC ISA. To process these instructions, mentioned ASIPs either use inline assembly or modify the compiler toolchain. So for each individual custom SIMD processor user must modify its compiler. On top of the compiler modification, to properly introduce new instructions, simulators and debuggers must be additionally retargeted.

Automated tools like Codasip Studio [39] or ASIP Designer [40] where ISA extended processors can be generated together with SDK exist, but this also limits the user by forcing one IP ecosystem. Chipyard RoCC [41] is another commonly used framework for designing accelerators for Rocket processor. However, communicating accelerators with its RoCC interface also requires a custom software toolchain. SIMDify solution can be applied to any SIMD loop that satisfies the memory constraints, whereas, contemporary approaches might require different custom instruction for each new application.

The solution proposed in this thesis is scalable, user-friendly, open-source, and does not depend on non-standard compilers. Hardware-level parallelization is achieved without using additional instructions. The only thing dependent on the user is coding the algorithm in a partitionable way and default tools for compiling and high-level synthesis. SIMDify framework only requires four inputs in the C code. Other inputs are only for parameter configuration and are not compulsory. Inputs are compiled as data memory values using default compiler. Then, an automated framework reads these values and configures the processor.

2.2. High Level Synthesis

Best performance in application specific processors are achieved by manually customizing an application for a specific target architecture and customizing the hardware for a specific application. However, the trade-off is the cost of portability, development expenses, and time cost. Furthermore, hardwired circuits are inappropriate for devices

that need to adapt to ever-changing algorithms. FPGAs, on the other hand, are reconfigurable hardware blocks. The FPGA architecture is relatively simple; array of programmable logic blocks connected to programmable interface. Since FPGAs can be configured after manufacturing by the user, they can be used to implement wide range of logic functions (from full adder to processor core). This makes them ideal for application tailored processing. Soft ASIPs are used as a viable strategy to reduce the design time without sacrificing performance and due to its reconfigurable nature.

To configure and program FPGAs, hardware description languages (HDLs) such as Verilog HDL and VHDL (VHSIC-HDL, Very High-Speed Integrated Circuit Hardware Description Language) is used. HDLs describe combinational and sequential logic. An IDE (Integrated Development Environment) such as Vivado Design Suite from Xilinx, Intel Quartus Prime Software Suite from Intel, and HDL Designer from Mentor Graphics interprets, optimizes, and tests HDL code and synthesizes and configures the FPGA with the equivalent logic.

However, the traditional FPGA design flow demands specialized hardware design expertise and familiarity with Hardware description language (HDL), which is difficult for non-hardware designers. With the advent of the High-Level Synthesis (HLS) tools, it is possible to prototype, synthesize, and simulate hardware using a high-level languages such as C or C++. Just as HDL is interpreted to synthesize gate-level logic, HLS tool such as Vivado HLS, Intel HLS Compiler or Catapult HLS interprets C code to generate an HDL text file. In the case of Vivado HLS, the text file is formatted as Verilog and VHDL. Typical Vivado HLS design flow [30] consists of:

- Compiling, simulating and debugging the C algorithm.
- Synthesize the C algorithm as an register transfer level (RTL) implementation. Optional user directives called pragmas can be used to guide the HLS tool.
- Generate reports and analyze the design.
- Verify the RTL implementation using co-simulation.
- Package the RTL implementation into an IP.

C++ code is used to describe the behavior, and the HLS tool synthesizes the corresponding register transfer level (RTL) circuit. HLS inputs consists of function written in C, C++, or SystemC, constraints such as clock period, uncertainty and target FPGA, optional directives (pragmas) that guides the synthesis process, optimizes the system, and implements specific behavior and finally the test bench files for verification. HLS outputs consists of RTL implementation files in HDL and report files.

The tool gives designers better authority over-optimization of their design architecture. However, there are more ways than one to synthesize the C code. So, tools must be guided by the user through pragmas. Quality of the design is directly dictated by the selected pragmas. Hence, iterative design process for finding the best solution takes a considerable design effort and time. In our approach, we have already designed the template SIMD processor architecture. The SIMDify framework, which generates application specific SIMD architecture, greatly reduces the design effort and time of the user. SIMDify fully utilizes HLS and its C like header structure to reduce design time.

HLS has many built-in pragmas that correspond to design constraints such as parallel and pipelined design. Vivado HLS is chosen as the primary tool due to its fast design process, built-in pragmas, and accessible and flexible nature. In [42] it is shown that HLS can reduce the design effort compared to non-HLS RTL. In terms of area, the processor designed in HLS is %50 larger than its RTL equivalent.

Example processors designed using a HLS in the literature are Comet core [43] in RISC-V ISA and Catapult HLS, HL5 [42] in RISC-V ISA and SystemC and, approximate CPU [44] in RISC-V ISA and Vivado HLS. All mentioned processors doesn't have a SIMD support, but Comet does allow instruction extensions by modifying the HLS code. However, compiler modification must be done by the user. HL5 and Comet have stable $riscv32im$ instruction support. [45] and [46] are MIPS architecture based processors, utilizing Vivado and LegUp [47] HLS tools, respectively. In [14] SIMD processor for software-defined radio (SDR) applications is designed using OpenCL language [48].

Bespoke processor article [49] mentions HLS is costly and increases design and verification effort. However, designed SIMDify framework automates this process with the guidance from user. So user won't be needed to verify the design all the time. User can generate the application machine code and SIMDify automatically applies SIMD processing and trims the unnecessary blocks.

Another article presents Trimmed VLIW approach [50]. It trims down the HDL code depending on the application. For instance, 4:1 mux is trimmed down to 2:1 if the select signal has two constant values. HLS does that automatically for the given process. Since it exists in a high level abstraction, constants will be defined before muxes. So, HLS will trim down the constant switch case arguments in the code. Therefore, using HLS is beneficial with respect to design time, flexibility and overall control.

2.3. RISC-V Instruction Set Architecture

Most ISAs used by major companies such as ARM, Intel, and AMD are proprietary. For this reason, free, open-source ISAs like Open RISC [51] and RISC-V [1] based processors are gaining momentum in custom processor designs. RISC-V is recommended as an open-source ISA standard by [52]. Currently, RISC-V lead by the RISC-V foundation [53] and its members.

In this work, RISC-V ISA [1] is chosen due to its open-source, free, active, and well-documented nature. The base RISC-V ISA from University of California, Berkeley had been released in 2011 and it is still rising in popularity with its open source GitHub applications, public Google groups and meetings. RISC-V attracts a wide variety of researchers from both academic community [54, 55], and private companies [56–59]. RISC-V has a compiler, simulator, QEMU support, and a cycle-accurate verification suite.

The main goal of RISC-V is to create a long-lasting open-source ISA ecosystem with a wide range of uses. For this reason, it can be both scaled down using its embedded ISA and scaled up using its single, double and, quadruple precision floating-point support. Its 32, 64, and 128 bit base instruction set is suitable for all ranges of devices from IoT to warehouse scale computing systems. A number of hard and soft core designs ranging from simple single cores [60,61] to complex out-of-order superscalar cores [57, 62, 63] have been shared as open-source and many RISC-V related academic papers have been published. Some commercial products from different vendors are also available in the market [56, 58, 59].

As a part of the European processor initiative (EPI) processor with RISC-V ISA will be developed using fully European IPs [64]. Manufacturers such as Western Digital Corporation [57], Google [65], and Alibaba [66], also designed processors using RISC-V ISA. Several implementations [67–69] of RISC-V have been made in Chisel language [70]. Even though Chisel is different than the traditional HDL, it is closer to the HDL than to the HLS [71].

Even though RISC-V ISA has two extensions for parallel computation, i.e., “P” (Packed SIMD) and “V” (Vector) extensions, currently, both extensions are not ratified. Also, ”V” extension is not tailored for packed SIMD applications, and the ”P” extension is not scalable. Since these extensions are subject to change, the designed processor might be obsolete in the future. This problem can be solved using compiler retargeting and extending standard instruction set with non-standard custom instructions [21] or using Just-in-time (JIT) compilers, which compiles the program in run-time [22]. Both approaches yield non-standard compilers as well as non-standard instructions in custom hardware. So, compiler must be modified for each individual accelerator.

”V” extension can work with small scale vector lengths, however it’s intended for high performance computing with its OpenMP support. Allowing vector processing requires significant changes in the processor architecture, whereas SIMDify can unroll loops with its simple core architecture. In ”V” extension, width of a vector can only

be a power of 2 whereas SIMDify can take custom unroll factors as an input. However, custom unroll factors must exactly divide the number of iterations in the loop.

"P" extension requires a different machine code for different the number of parallel computing units. SIMDify does not have this constraint. Hence, the user can easily explore design space to optimize the overall design without recompiling the software. Currently, "P" extension is not supported by the standard RISC-V compiler. Both "V" and "P" extensions are not standardized and there are no existing open source designs with these extensions. We released SIMDify as an open source project on Github [29].

3. SIMDIFY FRAMEWORK

SIMDify can parallelize and accelerate an application with minimal software level modification and using the standard RISC-V compiler. It utilizes HLS pragmas and C like header structure of the HLS. Using HLS, SIMDify processes the RISC-V compiler machine code and HLS simulator outputs and automatically generates desired SIMD processor architecture. SIMDify is fully automated and it requires only 4 variables to configure the software, which reduces the design time.

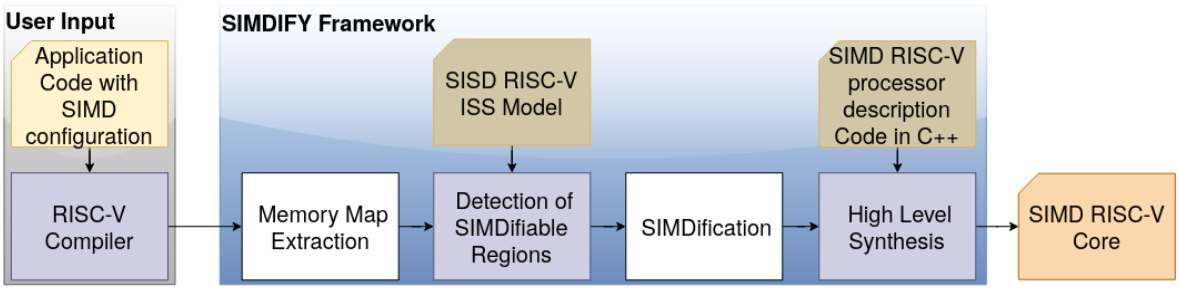

Figure 3.1. Block diagram of SIMDify Framework.

Operation of SIMDify framework is shown in Figure 3.1. It takes the compiled machine code that contains the algorithm and necessary configuration parameters. The machine code is fed to the Memory Map Extraction block to generate the Local Memory header file. Then, the Local Memory header and the SISD (Single-instruction-single-data) RISC-V ISS (Instruction Set Simulation) Model is fed to the next block to detect the regions that will be executed as SIMD (SIMDifiable Regions) and write them to the Address Header. After that, the SIMDification block generates the SIMD header file using the Address and Local Memory Header. Lastly, SIMD RISC-V processor description code in C++ and all header files are synthesized in Vivado HLS to generate SIMD RISC-V Core. All steps are automated inside the SIMDify Framework. A detailed explanation for each block in the figure is given in the rest of this section.

3.1. Application Code with SIMD configuration

Four variables must be included in the C code to generate and configure the SIMD processor successfully:

- StartPar: Determines the region which SIMD processing will be executed. The user has to set StartPar to 1 just before the loop begins and to 0 just after the loop ends.
- par_num: Unroll factor. Determines the number of SIMD processes. Number must exactly divide the loop count. Denoted by n.
- arr_str: Start local data memory address of the SIMD array. Used in SIMD slaves. Equals to &SMA[0]; where SIMD_memory_array (SMA) is the name of the array accessed in the SIMD loop with size X. Denoted by $A_{data,start}$.
- arr_end: Last local data memory address of the SIMD array. Used in SIMD slaves. Equals to &SMA[X-1] + (&SMA[X-1] - &SMA[X-2]); where SIMD_memory_array (SMA) is the name of the array accessed in the SIMD loop with a size X. Denoted by $A_{data,end}$.

In the local memory, variables have specific addresses which are generated using the "section" command. This command is a GCC variable attribute which is used for setting particular variables to appear in individual sections (address ranges). Only the unroll factor can be modified after compilation. To change the other three, code must be re-compiled. Section names and addresses are determined from the linker file.

Our SIMD processor template processing system consists of one master processing element (PE) and $n-1$ slave PEs. Master can access the complete local memory and executes the sequential code. During SIMD execution, master also executes concurrently with the slaves. So, during SIMD processing, n PEs execute concurrently. In order to fully benefit from SIMD operation, memory access range of each PE has to be contiguous as shown in Figure 3.2. To achieve this, the user must write the SIMD loop part of the C code while considering memory adjacency. For example, consider a four

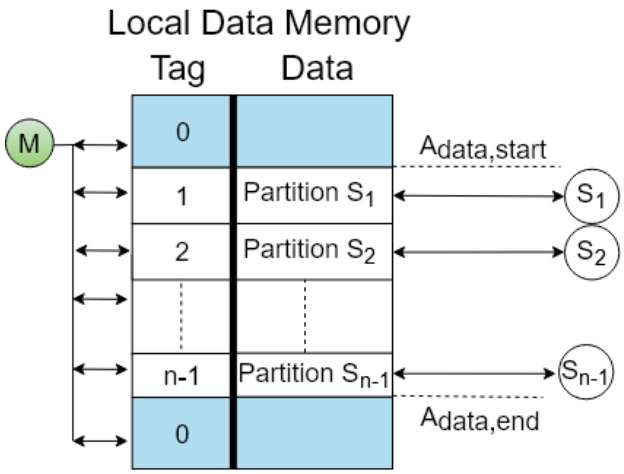

Figure 3.2. Accessible regions in the Local data memory for n-1 Slave PEs and the Master PE. Different Tag values are generated for each partition.

iteration loop for matrix vector multiplication $A[i][0..4] \cdot v[0..4] = r[i]$ like in Figure 3.3 where A is the name of the 4×5 matrix, v is the multiplied vector with size 5 and r is the result vector, Figure 3.4.a. In i-th iteration each element in i-th row of A is multiplied with elements in the vector and summed up.

```
Matrix_multiply:for(int i = 0; i < X; i++){
    for (int j = 0; j < 5; j ++) {
        r[i][5] = r[i][5] + A[i][j] * v[j];
    }
}
```

Figure 3.3. Non-SIMDifiable C code of the matrix multiplication example.

The example code results in one matrix, one vector, and one result block in the memory, Figure 3.4.b. To design a SIMDifiable C code, all addresses accessed in only one iteration in the SIMD loop, i-th row of A and r, must be adjacent in the memory. So, code shown in Figure 3.4.a, cannot be executed in our designed SIMD processor. We solve this problem by adding another column to the matrix to store the result vector by modifying the multiplication as $A[i][0..4] \cdot v[0..4] = A[i][5]$, Figure 3.4.c. In this way, all arrays that are read and written in one iteration are compiled as adjacent memory partitions 3.4.d. Hence, each SIMD slave S_i will be able to execute in its own dedicated partition *Partition* S_i, as shown in Figure 3.2. Note that the master M can access the entire local memory.

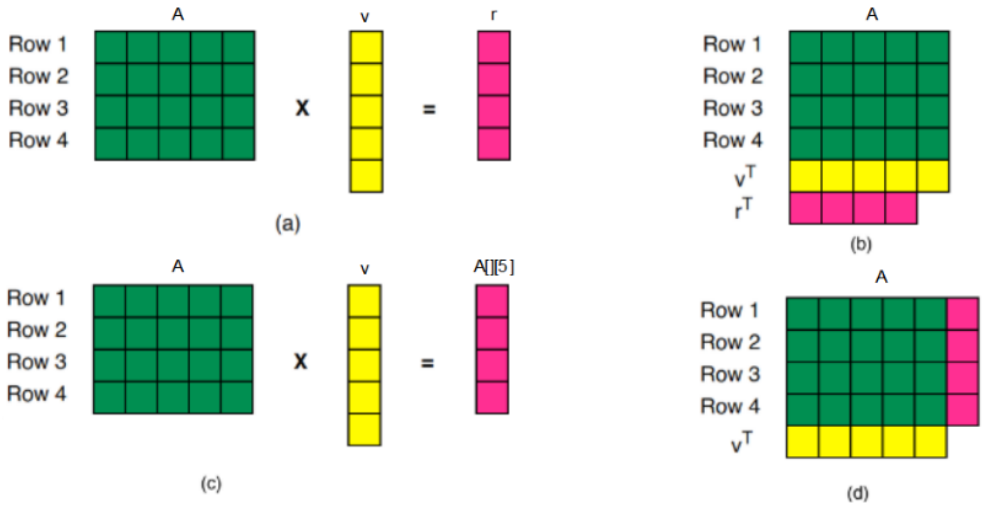

Figure 3.4. Coding example (a) Vector matrix multiplication
(b) Default memory allocation of arrays after declaration.
(c) Suggested SIMDifiable vector matrix multiplication.
(d) SIMDifiable memory allocation after suggestion.

Local data memory in Figure 3.2 consists of data and tag fields. Tag field is used for local data memory access and the data field stores the local data. It is a single block that contains address-to-partition mapping. Tag field makes a trade-off between memory access latency and area. By using tag field, area is increased. In return, the SIMD architecture does not require many comparison and multiplexer blocks, which increase the latency of the address-to-partition mapping process.

Size of the tag field is proportional to the size of the data field, and each tag contains values from 0 to $n-1$. 0 value is for regions that are only accessed by the master, and 1 to $n-1$ is for slave regions S_1 through S_{n-1}. How tag field is used for memory access is detailed in Chapter 4.1.

C code of the example matrix multiplication structure should be written as Figure 3.5. Variables that are not accessed in only one iteration in the parallelized loop need not be adjacent in the memory. So, user only has to modify its SIMD execution loop and include the four variables. Rest of the code remains the same.

```
int X=4;  // size of the iteration and the SIMD array
// three variables must set before SIMD loop:
par_num=4; //can be 1,2, and 4
arr_str= &A[0];
arr_end=&A[X-1] + (&A[X-1] - &A[X-2]);
//startPar must be set before and after the SIMD loop.
startPar=1;
SIMD_loop:for(int i = 0; i < X; i++){
    for (int j = 0; j < 5; j ++) {
        A[i][5] = A[i][5] + A[i][j] * v[j];
    }}
startPar=0;
```

Figure 3.5. SIMDifiable C code of the matrix multiplication example.

As another example, we can present matrix matrix multiplication (MMM). Figure 3.6 shows MMM for loop, $A[i][0..1] \cdot B[0..1][0..2] = A[i][2..4]$, is SIMDified and each PE executes $A[i][0..1] \cdot B[0..1][x] = A[i][x+2]$ operation. In Figure 3.6.d, B is the common memory and can be accessed by all slaves and A is the partitioned memory and can only be accessed by single slave. Master can access to the entire local memory.

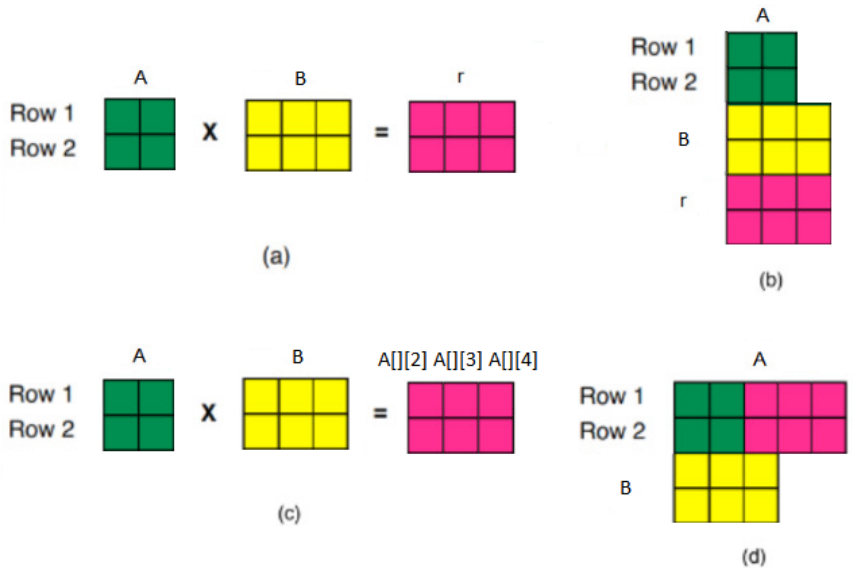

Figure 3.6. Coding example (a) Matrix matrix multiplication
(b) Default memory allocation of arrays after declaration.
(c) Suggested SIMDifiable matrix matrix multiplication
(d) SIMDifiable memory allocation after suggestion

3.2. RISC-V-compiler

In this work standard RISC-V compiler such as riscv32-unknown-elf, riscv-none-embed, riscv64-unknown-elf GCC is used. To synthesize memory in a partitionable way, the compiler optimization level must be 3. The compiler generates the machine code, which consists of the data memory before the execution, and the instructions to be executed.

3.3. Memory Map Extraction

Memory Map Extraction block reads the machine code and generates the corresponding Local Memory header file for the HLS. Local Memory header contains instruction and data array. In the instruction array, each element contains 32-bit instructions. The size of the instruction array depends on the generated machine code. The data array contains 32 bits as 4x8, partitioned as 4 dual port 8 bit sized memory arrays. The length of the data array depends on the linker file. The header also contains macros for each instruction in the instruction binary. For example, if instruction binary contains an *ADDI* instruction, header contains *#define ADDI* directive. Macros are used in HLS to remove unused instructions of RISC-V and to create an area efficient core.

3.4. SISD RISC-V Instruction Set Simulation (ISS) Model

This model is written in C++ to be simulated with the Vivado HLS. With the use of HLS-specific constructs like ap_int library and HLS directives, overall design time is reduced. SISD model is only used in HLS C simulation to read instruction and data arrays in the local memory and generate the address header, as explained in the next part.

3.5. Detection of SIMDifiable Regions

Instructions in the local memory header are simulated in HLS without Register-Transfer Level (RTL) synthesis using the SISD RISC-V ISS model. While simulating, the model constantly reads the four variables (StartPar, n, $A_{data,start}$, $A_{data,end}$) from their respective local addresses, Figure 3.7. When the StartPar is read as "1", it means that simulation is entering the SIMD loop and when it is read as "0", it means that simulation is exiting the SIMD loop. Meanwhile values $A_{data,start}$, $A_{data,end}$, and par_num, which are set before SIMD loop, are saved to the Address Header.

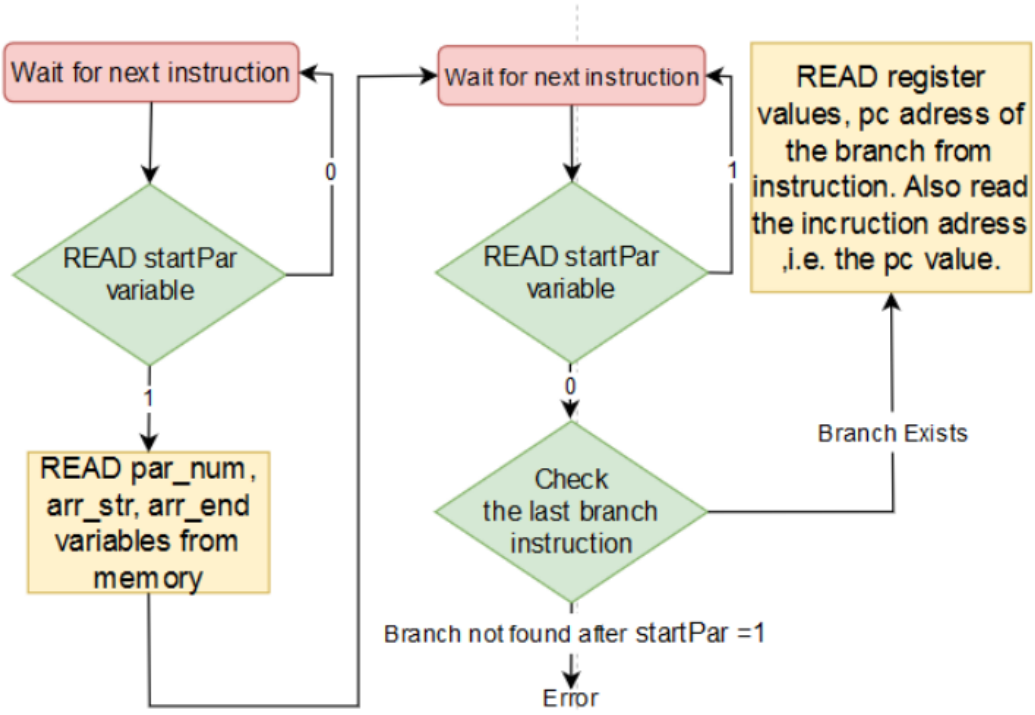

Figure 3.7. Flow diagram for Detection of SIMDifiable Regions block.

After exiting the loop, the model checks the branch instruction of the SIMD loop. The start of the SIMD loop, branch target address, is equal to the sum of sign extended immediate offset, imm[12:1], and branch program counter (PC) address (Figure 3.8). Together with the branch target address and next value after branch PC address, the register numbers and contents given in the source register (rs1 and rs2) fields of the branch instruction are saved to the Address Header. Detailed explanation about how register numbers and contents are used for transition between normal processing mode and SIMD processing mode is given in Chapter 4.1.

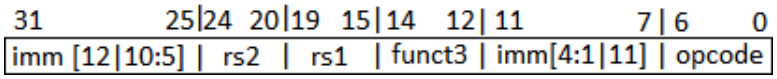

Figure 3.8. B-type instruction structure for RISC-V, rs1 and rs2 are the source registers for branch operation [1].

3.6. SIMDification

SIMDification block generates the HLS SIMD header file that consists of slave PE and cache parameters, partitions, and functions used in the SIMD execution. By default, this block uses the unroll factor determined in the C code, but it can be overwritten to reconfigure SIMD processor without re-compiling it from scratch. SIMD processing can be applied to any memory partitionable loop in the application. SIMDified local data memory is generated by allocating all the data between the $A_{data,start}$ and $A_{data,end}$ into n equisized partitions, as shown in Figure 3.9. The master PE acts as n-th slave during SIMD processing. Start and end addresses of the partitions are saved to the SIMD header. These are also used while transitioning between standard processing mode and SIMD processing mode. SIMDification block also generates constant memory tags for every word in the local data memory. CPU looks at the tags to determine which memory address belongs to which memory partition. Tagged memory architecture will be detailed in Chapter 4.1.

3.7. SIMD RISC-V processor description Code in C++

SIMD RISC-V processor description Code is a HLS code that is written in C++ and is responsible for generating processor system with dynamic branch prediction. It generates two types of datapath:

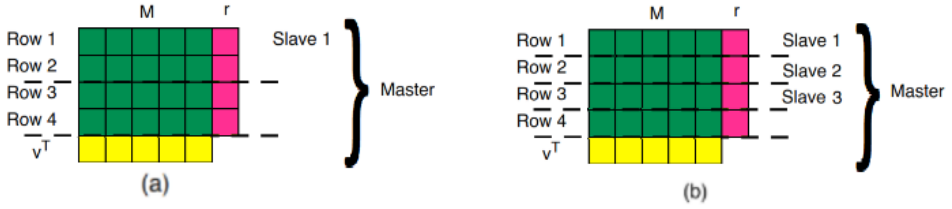

Figure 3.9. Example partitioning for a) n=2 , b) n=4

- *Master Datapath:* It is always executed, unique and responsible for branch prediction, stalls, and other control signals. Master datapath can access all the local memory (data and instruction), external memory, and register file array.
- *Slave Datapath:* There are $n-1$ slaves, which are executed only during SIMD processing. Each slave can only access its own register file and its own memory partition. In SIMD processing, slaves are not executed if the instruction is a branch or a jump, or an instruction is accessing a different memory address than its own partition (common memory).

Master and slave datapaths are entangled in the processor and not single blocks, but for the sake of clarity they will be referred as Master and Slave processing elements throughout this thesis. An illustrative partitioning is given in Figure 3.9 for n=2 and n=4 for the matrix-vector multiplication code. In a memory partitionable loop, every load or store is accessing a different part of the memory or a common memory address. So, there are no dependencies between iterations. In each iteration, SIMD loops either access to the common memory (like vector load) or they all access to a different part of the memory (like matrix load/store). In matrix multiplication, SIMD Slaves are not executed when the code is accessing the common memory (v block). Instead, master LOADs v array and writes to all n registers. Memory of the matrix is partitioned amongst PEs, the vector memory will reside in the non-partitioned common memory, and only the master PE can access it and write to all registers. If v must be STOREd inside SIMD loop, it must be a part of the partitioned matrix A.

3.8. High Level Synthesis

Using HLS, SIMDify synthesizes the processor using generated headers and PE codes written in C++. For different applications the flow must start from the beginning. For the same application with different unroll factors, starting from the SIMDification step is enough.

4. CORE ARCHITECTURE

Our soft application-specific SIMD-processor consists of two main parts: A relatively large master PE and small slave PEs. Using HLS, SIMDify can combine and connect master and slaves to generate various SIMD processors for an application depending on the unroll factor. The processor is designed in C++ and synthesized in Vivado High-Level Synthesis 2019.2.

4.1. SIMD Processor

In our system, software loop is unrolled in hardware level to be executed in parallel as SIMD. Execution results in n times the latency gain for the SIMD executed part. The user guides the SIMDify, and the framework configures the processor accordingly. This process does not require inline assembly or custom instructions. It only requires modification on the SIMDified loop itself, thus, rest of the application does not need to be modified. Also, no extra instruction overhead is added to instructions generated by the compiler.

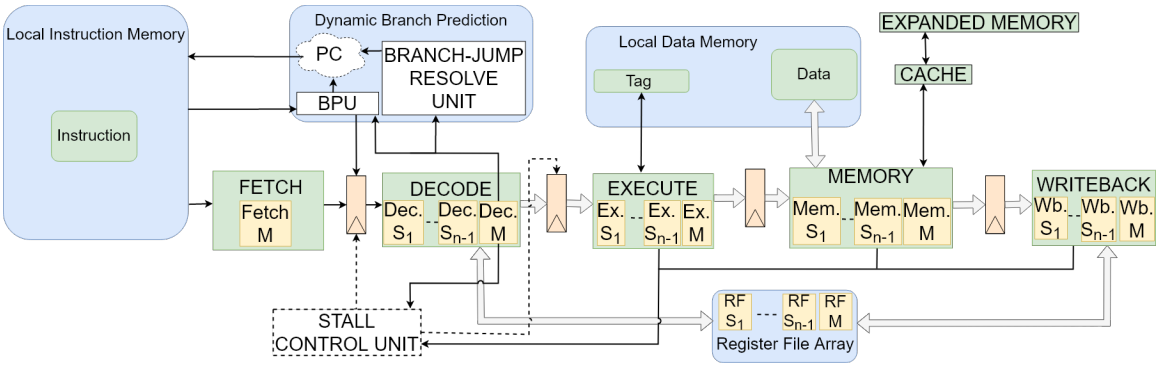

Figure 4.1. Block diagram of overall system with 1 master PE and n-1 slave PEs. Local data memory is detailed in Figure 3.2.

The overall SIMD processor architecture is shown in Figure 4.1. SIMD processor consists of a master and $n-1$ slaves. Using HLS, SIMDify combines and connects these PEs to generate different SIMD processors for each application and unroll factor. Proposed processor architecture is the first RISC-V processor with SIMD support de-

signed using HLS. In the figure, thin lines are for single data and exclusive to master. Thick lines indicate busses where both master and slaves execute. Dashed lines are stall outputs. The designed processor runs in one of two different modes at any given time:

- Standard mode where the only active PE is the master.
- Parallel mode for SIMD processing where all PEs are active.

In the fetch stage, the master checks the PC value to start or end the SIMD processing. Before beginning the SIMD processing, the master initializes all slaves by writing different values of the SIMD loop iterator to the register files. These values are pre-calculated by the SIMDify tool as explained in the previous section.

In a SIMD loop machine code, rs1 and rs2 of branch source registers are set as initial and final addresses of the memory partition. Register of the initial_address is incremented until it's the same as final_address. This is purely done by compiler and similar for every SIMDifiable loop.

Consider an example where, unroll factor is 3, and SIMD loop accesses addresses 1 to 30. So, initial_address is 1 and final_address is 30. Master PE overrides "set rs1 and rs2" instruction and sets rs1 and rs2 values of the slave PEs as 1, 11 and 10, 20 and master PE as 21 and 30 respectively. This approach is similar to loop unrolling. This does not take additional time, since initialization is executed instead of "set rs1 and rs2" instruction. After the loop, master PE continues its normal operation. Since memory accessed in each iteration corresponds to different memory partitions, the system can be executed as SIMD.

The master determines execution mode by checking the PC value in fetch stage. System runs in parallel mode if the PC value corresponds to the SIMD loop and runs in standard mode if it doesn't. Additionally, only master is active if instruction is LUI, AUIPC, JUMP, or BRANCH, or accesses to non-partitioned (common) local memory.

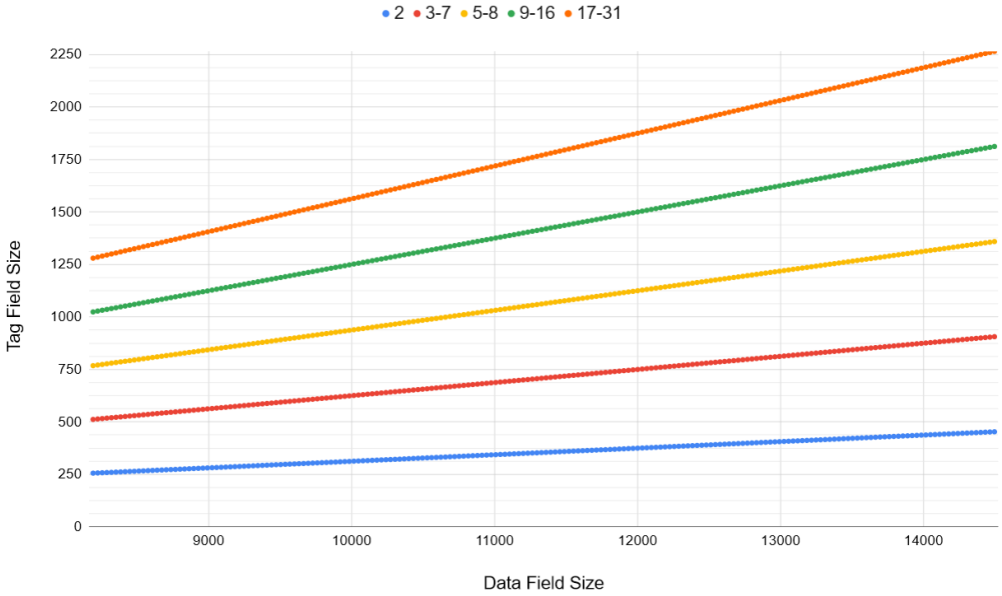

Figure 4.2. Plot of tag field size vs data field size for different unroll factors up to 32. For all unroll factors Tag field size can be calculated with Equation 4.1. Data field is given as multiples of 32.

Local data memory consists of a data field that has random access data and a tag field that identifies this data. Tag field is generated by the SIMDify and cannot be accessed by instructions. In the execute stage, the tag of the data is read from the tag field, and it is used to set the enable signals of the memory partitions. Then, in the memory stage, enable signals are used by the PEs to access the correct memory partition. There are three possible outcomes depending of the tag values and current mode:

- In standard mode: Only the master is active and tag value is used to give access to the master PE to the demanded memory partition.
- In parallel mode and all addresses are the same: That means PEs are reading from a common memory like v. In this case only the master PE accesses the memory and writes to all register files.
- In parallel mode and all addresses are different: That means core is executing as SIMD and every PE reads and writes to its own partition, by using their dedicated RFs.

Size of the tag field depends of size of the data field and the unroll factor, Equation 4.1, which can be seen in Figure 4.2. If the unroll factor is 1, tag field is not generated since there is only single memory partition and single PE.

$$TagFieldSize = \frac{DataFieldSize}{32} * \lceil log_2(n) \rceil \tag{4.1}$$

4.2. Master (Scalar) PE

The scalar PE in Figure 4.3 is master for slave PEs and supports $riscv32i$ instruction set and MUL, MULH, MULHSU, MULHU multiplication instructions. $Riscv32m$ exclusive multi-cycle instructions (REM and DIV) are not implemented. DIV and REM should be implemented in a way that does not change the behaviour of single cycle instructions. We observed that if they are implemented as a/b, HLS compiles the DIV instruction, but timing of the single cycle instructions also changes. Master has a standard five-stage pipeline [72]. Instructions are fetched and issued without changing the order of execution. Using the aforementioned directives unused instruction blocks are removed, which reduces the area. All data dependency hazards are solved via stalling.

Branch hazards are handled with dynamic one-level branch prediction with a 1-bit saturating counter, as shown in Figure 4.4. A saturating counter records the last branch result as 0 for not-taken, and 1 for taken. If taken, the branch address is also recorded. Jump instructions are resolved in the decode stage, which results in 1 cycle overhead. Branch instructions are also resolved in the decode stage to reduce misprediction penalty. Misprediction is solved with flushing fetch register and correcting the PC value. In the figure, top input of the pipe registers indicates flush, and bottom indicates stall. The core has a local memory for faster processing, and memory can be expanded with a cache connected to external memory. All instructions

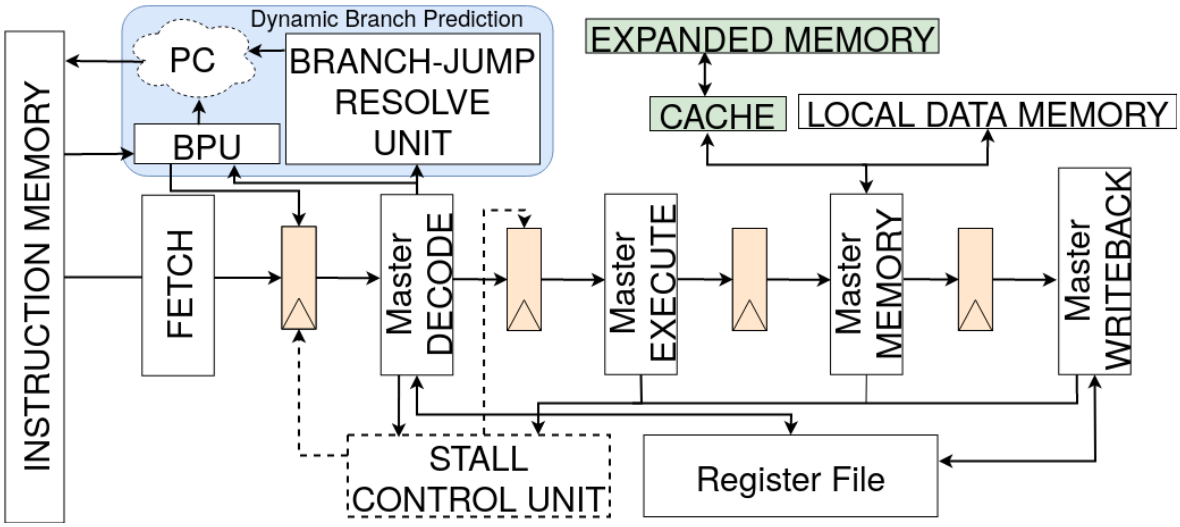

Figure 4.3. Block diagram of Scalar Datapath with a standard RISC five-stage pipeline with 1-bit branch prediction and optional cache.

are stored in the local instruction memory. Latency for memory stage is a single cycle for local memory. If the cache is implemented, the overall pipeline depth does not change, but the memory stage may take multiple cycles to execute. All local memories and cache memories are asynchronous read and synchronous write. It should be noted that block diagram is behaviorally correct, however, in HLS register file, writeback, and decode stages are written as one block. This is done to generate two port asynchronous read, one port synchronous write register file (RF). Generated RF Verilog code can be observed in Figure 4.5. Operands are sent to the decode first, and sent to the execute from there.

The master PE is also responsible for starting and ending the SIMD processing. Before beginning the SIMD processing, the master PE initializes slave PEs as explained in the Chapter 4.1.

Blocks of the unused instructions of the application are removed. This results in area and speed improvements. Normally each multiplication instruction (MUL, MULH, MULHSU, MULHU) requires 3 DSP blocks per instruction. So if an application only has MUL in its machine code, 3, if it has MUL and MULHSU 6 DSP blocks are used. If core were generated in non-application specific way, 12 DSP blocks would be used.

```
Fetch Stage
if  Instruction = Branch then
   if  SaturatingBit = Taken then
      NextPC = BTA ;
   else
      NextPC = PC + 4;
   end if
end if
Decode Stage
if  Instruction = Branch then
   Check for misprediction
   if  IsBranchTaken != SaturatingBit or BTA != calculated address then
      Flush Fetch Register;
      SaturatingBit = IsBranchTaken;
      if  IsBranchTaken  then
         BTA = calculated address;
         NextPC = calculated address;
      else
         NextPC = PC + 4;
      end if
   end if
end if
```

Figure 4.4. Algorithm for 1-bit Branch Prediction. BTA stands for branch target address.

```
assign q0 = ram[addr0];
assign q1 = ram[addr1];
always @(posedge clk)
begin
    if (we2)
    begin
        ram[addr2] <= d2;
    end
end
```

Figure 4.5. HDL of the Generated Register File

4.3. Slave PE

Slave PEs structure is shown in Figure 4.6. It only consist of decode, execute, memory, and writeback units. All slave PEs can only access to their individual 32-bit register files, and their partition in the main memory. Since it is guaranteed that all PEs will execute the same instruction at any given time, redundant signals are trimmed to reduce area. Slave PEs also do not have a stall, fetch, or branch units. They are generated only when the user demands a SIMD processing and are fully controlled by the master PE.

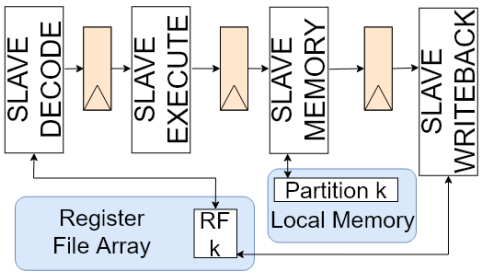

Figure 4.6. Block diagram of kth Slave PE.

Slave PEs are generated by using HLS loop unroll pragmas with case blocks. So, they use same blocks and same unified local memory as the master. SIMDify detects SIMD loops and guides HLS to generate slave PEs accordingly. This approach generates general purpose slaves and removes the need for designing custom modules per application. Slave PEs can execute most instructions of the supported ISA. Slave and master PEs are further reduced to only execute necessary instructions per application basis, which reduces area. If master PE is modified to include extra instructions, slave PEs can be easily scaled to include these instructions as well.

5. ILLUSTRATIVE EXAMPLE

5.1. Application Code with SIMD configuration

In this chapter illustrative example for the SIMDification of a matrix vector multiplication in Figure 5.1.a will be given. C code shows the multiplication of 25 by 4 matrix with 4 by 1 vector. *par_num* is the unroll factor, which is set globally before the main is called. Other three variables (*arr_str*, *arr_end* and *startPar*) are set during main call. Attribute fixes the location of the 4 variables. When compiling, matrix static variable is preloaded in data memory. We preloaded the matrix as an illustrative example. It is also possible to load the matrix using the external cache. Matrix starts at 0x4000 and ends at 0x4000+500. Data memory is given in the Figure 5.1.b.

```
#include <stdio.h>
#include <stdlib.h>

#define SIZEX 25
#define SIZEY 4
volatile int __attribute__((section(".mySection_par"))) startPar =  0x00 ;
volatile int __attribute__((section(".mySection_par_num"))) par_num =  0x03 ;
volatile int __attribute__((section(".mySection_arr_str"))) arr_str =  0x00 ;
volatile int __attribute__((section(".mySection_arr_end"))) arr_end =  0x00 ;

static  int matrix  [SIZEX][SIZEY+1] = {{0x51,0x38,0x16,0x03,0},
                                        {0x4E,0x30,0x16,0x02,0},
                                        {0x4B,0x33,0x14,0x01,0},
                                        {0x49,0x31,0x18,0x03,0},
                                        {0x59,0x2E,0x39,0x14,0},
                                        {0x6B,0x31,0x46,0x16,0},
                                        {0x59,0x30,0x48,0x18,0},
                                        {0x5C,0x2B,0x41,0x10,0},
                                        {0x63,0x23,0x48,0x18,0},
                                        {0x55,0x28,0x3E,0x11,0},
                                        {0x5E,0x33,0x04,0x1C,0},
                                        {0x61,0x2C,0x40,0x14,0},
                                        {0x64,0x28,0x4E,0x18,0},
                                        {0x61,0x2C,0x4B,0x13,0},
                                        {0x66,0x2E,0x44,0x14,0},
                                        {0x69,0x30,0x46,0x16,0},
                                        {0x6C,0x2C,0x04,0x16,0},
                                        {0x6B,0x30,0x50,0x1B,0},
                                        {0x60,0x2E,0x48,0x18,0},
                                        {0x5B,0x29,0x38,0x10,0},
                                        {0x58,0x26,0x3C,0x11,0},
                                        {0x58,0x26,0x3B,0x10,0},
                                        {0x5C,0x2B,0x3E,0x13,0},
                                        {0x60,0x2B,0x51,0x19,0},
                                        {0x56,0x30,0x48,0x18,0},};
static   int   vector [SIZEY] = {0x44,0x2E,0x1B,0x07};

int main() {
arr_str =  &matrix[0] ;
arr_end =  &matrix[SIZEX-1] + (&matrix[SIZEX-1] - &matrix[SIZEX-2]);
startPar = 0x1;
for (int i = 0; i < SIZEX; i ++) {
    for (int j = 0; j < SIZEY; j ++) {
        matrix[i][SIZEY] = matrix[i][SIZEY] + matrix[i][j] * vector[j];
    }
        //printf("result: %d = %d\n", i, matrix[i][SIZEY]);
}
startPar = 0x0;
return 0;
}
```

(a)

```
Contents of section .data:
 4000 51000000 38000000 16000000 03000000
 4010 00000000 4e000000 30000000 16000000
 4020 02000000 00000000 4b000000 33000000
 4030 14000000 01000000 00000000 49000000
 4040 31000000 18000000 03000000 00000000
 4050 59000000 2e000000 39000000 14000000
 4060 00000000 6b000000 31000000 46000000
 4070 16000000 00000000 59000000 30000000
 4080 48000000 18000000 00000000 5c000000
 4090 2b000000 41000000 10000000 00000000
 40a0 63000000 23000000 48000000 18000000
 40b0 00000000 59000000 28000000 3e000000
 40c0 11000000 00000000 5e000000 33000000
 40d0 04000000 1c000000 00000000 61000000
 40e0 2c000000 40000000 14000000 00000000
 40f0 64000000 28000000 4e000000 18000000
 4100 00000000 61000000 2c000000 4b000000
 4110 13000000 00000000 66000000 2e000000
 4120 44000000 14000000 00000000 69000000
 4130 30000000 46000000 16000000 00000000
 4140 6c000000 2c000000 04000000 16000000
 4150 00000000 6b000000 30000000 50000000
 4160 1b000000 00000000 60000000 2e000000
 4170 48000000 18000000 00000000 5b000000
 4180 29000000 38000000 10000000 00000000
 4190 58000000 26000000 3c000000 11000000
 41a0 00000000 58000000 26000000 3b000000
 41b0 10000000 00000000 5c000000 2b000000
 41c0 3e000000 13000000 00000000 60000000
 41d0 2b000000 51000000 19000000 00000000
 41e0 56000000 30000000 48000000 18000000
 41f0 00000000 00000000 00000000 00000000
```

(b)

Figure 5.1. (a) C code of Matrix-Vector Multiplication. (b) Data memory preloaded with matrix data.

5.2. RISC-V-compiler

After compiling the C code, we get Figure 5.2. Start code resets the registers, initializes the stack pointer, and calls the *main*(). *ebreak* stops the execution. There is a total of 104 instructions. In main code, addresses between 0x13c and 0x190 show the loop that will be SIMDified. Second for loop in the C code is automatically unrolled by the compiler. Before the loop, *arr_str* (0x114), *arr_end* (0x124) and *startPar* (0x130) variables are set. And after the loop, *startPar* (0x194) is set back to zero. Setting these values puts 10 additional instructions to the algorithm. Since the designed SIMD processor overrides existing instruction (0x134) to switch between SIMD processing mode and normal processing mode, transition does not cost extra cycles. Instead of executing instruction in the PC=0x134 and setting a4, it sets all a4 and a7 registers of the slave and master PE to their respective values. These values are calculated in the SIMDification step and correspond to start and end register values of the pseudo unrolled loop.

```
Disassembly of section .text:
00000000 <_start>:
   0:   00000093            li  ra,0
   4:   00000113            li  sp,0
   8:   00000193            li  gp,0
   c:   00000213            li  tp,0
  10:   00000293            li  t0,0
  14:   00000313            li  t1,0
  18:   00000393            li  t2,0
  1c:   00000413            li  s0,0
  20:   00000493            li  s1,0
  24:   00000513            li  a0,0
  28:   00000593            li  a1,0
  2c:   00000613            li  a2,0
  30:   00000693            li  a3,0
  34:   00000713            li  a4,0
  38:   00000793            li  a5,0
  3c:   00000197            auipc   gp,0x0
  40:   fc418193            addi    gp,gp,-60 # 0 <_start>
  44:   00008117            auipc   sp,0x8
  48:   fbc10113            addi    sp,sp,-68 # 8000 <_stack_start>
  4c:   00000097            auipc   ra,0x0
  50:   018080e7            jalr    24(ra) # 64 <__libc_init_array>
  54:   00000097            auipc   ra,0x0
  58:   0b4080e7            jalr    180(ra) # 108 <__DTOR_END__>
  5c:   00100073            ebreak
00000060 <_fini>:
  60:   00008067            ret
00000064 <__libc_init_array>:
  64:   ff010113            addi    sp,sp,-16
    ...
  f4:   00008067            ret
000000f8 <__CTOR_LIST__>:
    ...
00000100 <__CTOR_END__>:
    ...
```

(a)

```
Disassembly of section .text.startup:
00000108 <main>:
 108:   00004737            lui a4,0x4
 10c:   00070893            mv  a7,a4
 110:   000066b7            lui a3,0x6
 114:   0116a423            sw  a7,8(a3) # 6008 <arr_str>
 118:   000067b7            lui a5,0x6
 11c:   1f488893            addi    a7,a7,500
 120:   00006e37            lui t3,0x6
 124:   0117a623            sw  a7,12(a5) # 600c <arr_end>
 128:   000e0e13            mv  t3,t3
 12c:   00100793            li  a5,1
 130:   00fe2023            sw  a5,0(t3) # 6000 <startPar>
 134:   00070713            mv  a4,a4
 138:   02e00313            li  t1,46
 13c:   00472783            lw  a5,4(a4) # 4004
 140:   00072683            lw  a3,0(a4)
 144:   00872603            lw  a2,8(a4)
 148:   02f305b3            mul a1,t1,a5
 14c:   01072803            lw  a6,16(a4)
 150:   00469793            slli    a5,a3,0x4
 154:   00c72503            lw  a0,12(a4)
 158:   00d787b3            add a5,a5,a3
 15c:   00361693            slli    a3,a2,0x3
 160:   40c686b3            sub a3,a3,a2
 164:   00279793            slli    a5,a5,0x2
 168:   010787b3            add a5,a5,a6
 16c:   00269693            slli    a3,a3,0x2
 170:   40c686b3            sub a3,a3,a2
 174:   00b787b3            add a5,a5,a1
 178:   00351613            slli    a2,a0,0x3
 17c:   00d787b3            add a5,a5,a3
 180:   40a606b3            sub a3,a2,a0
 184:   00d787b3            add a5,a5,a3
 188:   00f72823            sw  a5,16(a4)
 18c:   01470713            addi    a4,a4,20
 190:   fae896e3            bne a7,a4,13c <main+0x34>
 194:   000e2023            sw  zero,0(t3)
 198:   00000513            li  a0,0
 19c:   00008067            ret
```

(b)

Figure 5.2. Assembly code of matrix vector multiplication.
a) Start Code, b) Main Code

5.3. Memory Map Extraction

After the memory map extraction, we get Figure 5.3. It contains the necessary information to run the algorithm without SIMDifying it. In instruction array *inst_mem*, each element contains 32-bit instructions. The size of the *inst_mem* depends on the generated machine code, which is 104 words in this case. This part is synthesized as the instruction memory of the processor. The data array *mem* contains 32 bits as 4x8, synthesized as 4 dual port 8-bit memory arrays. Size of the *mem* is taken from the text file. To reduce the occupying space in the figure *inst_mem* and *mem* variables are cut off. The header also contains definitions for existing instructions in the instruction binary. These are used to trim unnecessary blocks in HLS. For example, there isn't any MULH instruction. So we do not generate extra multiplication blocks for MULH. GLUT part trims large blocks (ALU, STORE) for non-used group of instructions. Other constants set the size of the local data and instruction memory and locations of the 4 variables.

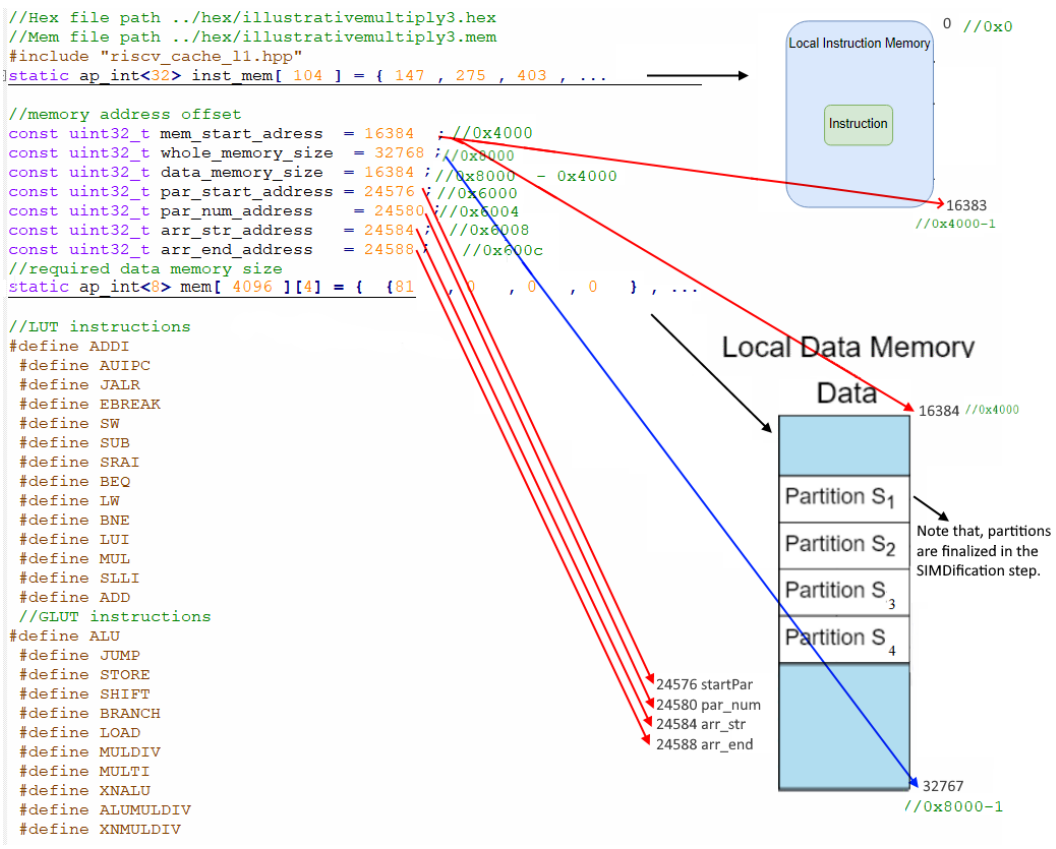

Figure 5.3. Local Memory header file of matrix vector multiplication.

5.4. Detection of SIMDifiable Regions

After Memory Map Extraction, algorithm is simulated once to extract necessary values for SIMD processing. These values are stored in a header, Figure 5.4. Variables for_init and for_end hold the start and end addresses of matrix variable - the common memory offset (memory_start_address, Figure 5.3). Variables $start\_reg$ and end_reg show the incremented (a4, 0x14 in [1]) and constant (a7, 0x17 in [1]) register numbers of the branch instruction in Figure 5.4 address 0x190, respectively. As it can be seen in Figure 5.4, a4 is incremented and a7 is constant throughout the loop. Register a4 starts at $memory\_start\_address + 0$ value and is incremented by 500, which corresponds to for_init and for_end variables. Variables Par_start_addr and Par_end_addr hold the boundary start and end PC values of the SIMDify loop. Variable $init\_simd\_offset$ refers to negative offset from Par_start_addr to last modified PC of a4 or a7 between Par_start_addr (0x13c in Figure 5.4) and $StartPar = 1$ (0x130 in Figure 5.4). This offset value corresponds to $PC = 0x134$ in Figure 5.4. Tool uses $init\_simd\_offset$ and overrides that instruction for switching SIMD processing mode, so that core won't lose extra cycles when switching between modes.

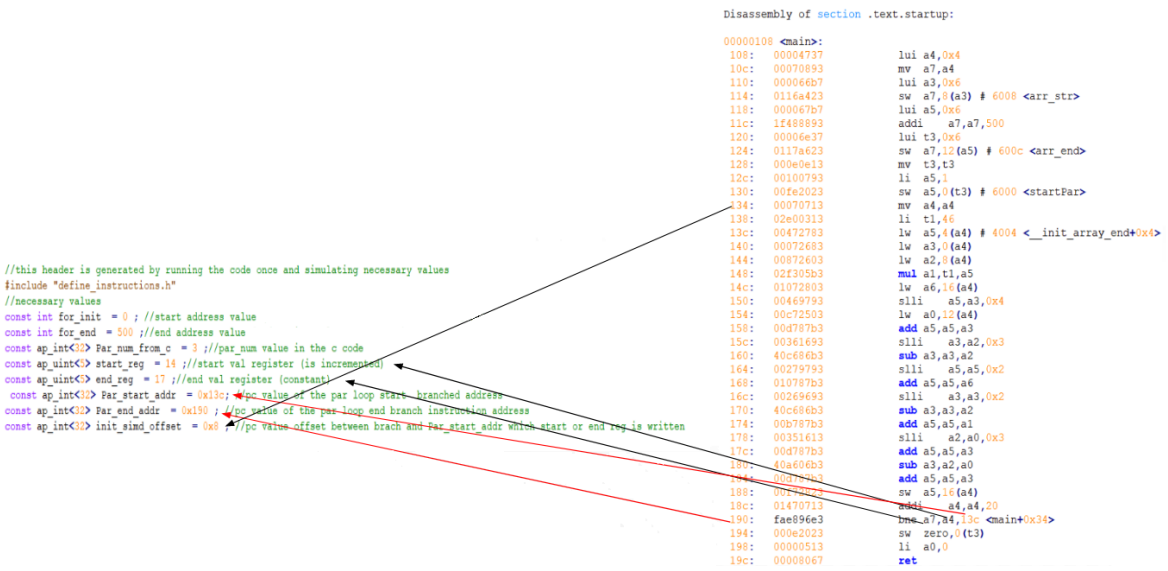

Figure 5.4. Address Header file of matrix vector multiplication.

5.5. SIMDification

After SIMDifiable regions are detected, unroll factor can be changed manually. Since 25 is not divisible by 3 (*par_num* in the C code), it will be changed to 5 in this stage, Figure 5.5. There are $n + 1 = 6$ enumerated tags, tag $ext = n$ is generated for external cache, others are used for n PEs. Since the example does not use external memory, tag field does not contain any *ext*. Variable *Addrlut* is the tag field, and *mem_par* is the partitioned memory blocks of the PEs. Register file and partition numbers are dependent on the unroll factor. This is handled with $SETRF$ and $GETSET$ macros, and they are used in writeback and memory stages respectively. When $PC = 0x134$ a7 and a4 values of PE registers set to $for\_array\_parstart$ and for_array_end values, respectively. This simulates unrolling the loop. Variable for_array_init is the offset for each memory partition.

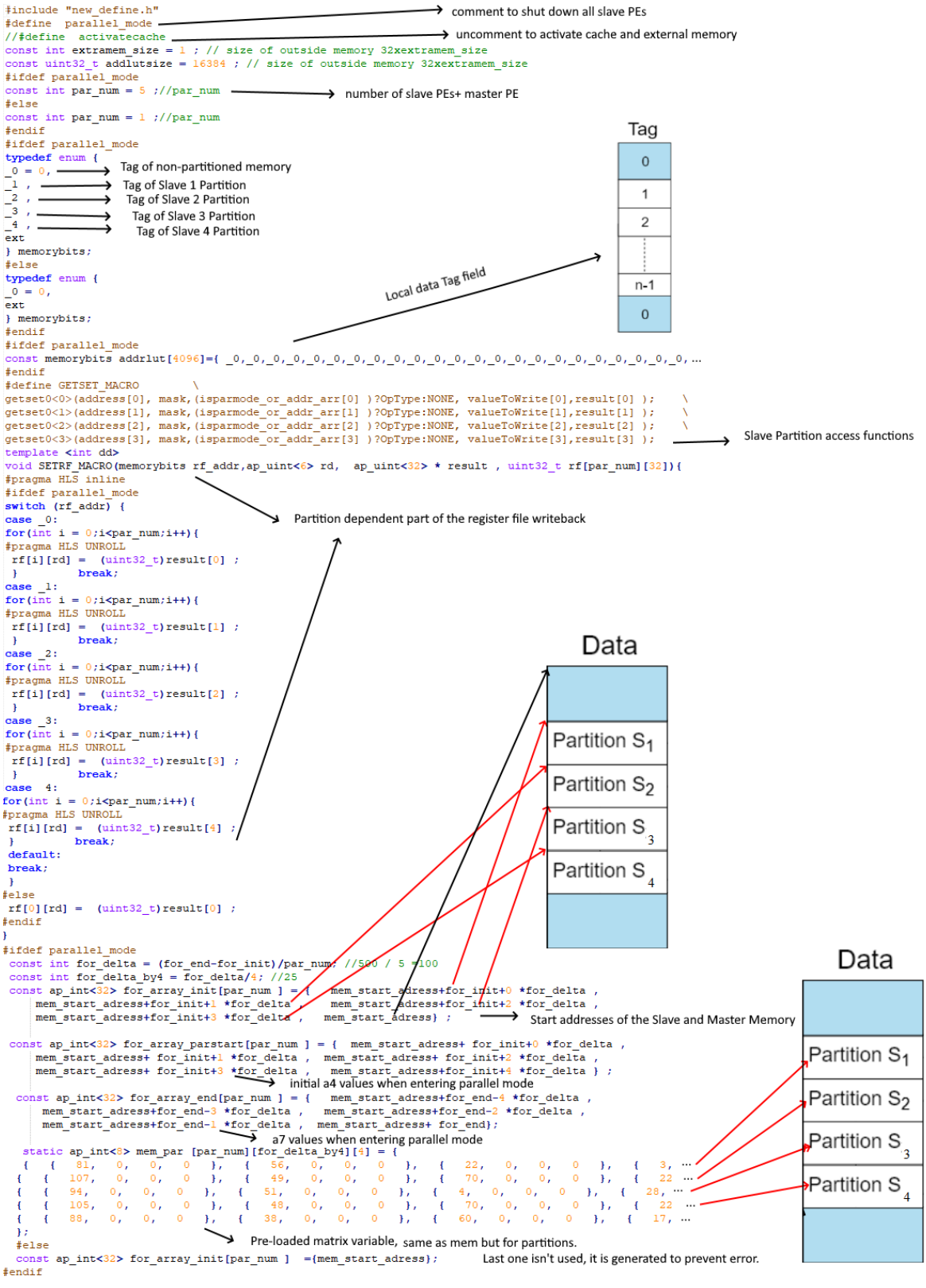

Figure 5.5. SIMD header file of matrix vector multiplication.

6. EXPERIMENTS AND RESULTS

6.1. Experimental Setup

Experiments are carried out on Zynq-7020-2CLG484-1 FPGA as hardware simulation. Each test uses less than 32KB of total memory, which fits in the local memory. C code is compiled with riscv64-unknown-elf-gcc 7.2.0 with following options $-mabi = ilp32 - g0 - O3 - march = rv32im - Wl, - - no - relax - nostartfiles$.

We implemented algorithms of matrix vector multiplication (MVM), matrix matrix multiplication (MMM), sum of absolute distances (SAD), sum of squared distances (SSD), artificial neural networks (ANN), k-nearest neighbors with selective sort (KNS), k-nearest neighbors with qsort (KNQ), K-means clustering (KMN) and Discrete Cosine Transform (DCT). Both massively parallelizable algorithms with large parallel portions (MVM, SAD, SSD, ANN, DCT) and partially parallelizable algorithms with smaller parallel portions (KNS, KNQ, KMN) are tested. Both large and small scale applications only requires user modification on the SIMD loop, rest of the application does not change. SIMDify focuses on unrolling user picked critical loops in the application.

6.2. Overall Results

We verified the HDL generated by SIMDify against SISD RISC-V ISS Model and confirmed that their outputs agree. To measure the latency, HLS cosimulation results are used, which are based on synthesized HDL code. Resource usage and clock speed values are taken from the synthesis report. Generation time is around 4 min on a four-core Intel Xeon server.

6.2.1. Clock Speed

Clock speeds are calculated when the target clock is 15ns, and uncertainty is %12.5. Three different parameters affect the clock speed: multiplication, cache and the number of PEs, i.e. n. If an algorithm uses one of multiplication instructions, MUL block is inserted, and its 11 ns slack causes the bottleneck. However, if it does not use any multiplication instructions, its period changes depending on the number of slave PEs, Table 6.1. In the algorithms mentioned above, only SAD does not use any multiplication instructions. If the unroll factor is one, only the scalar PE with one partition is used, so extra logic for slave PE routing is removed. Cached clock-speed, 11.827 ns, is faster than the non-cached core, but, it requires three times more clock cycles to complete. Different master and slave PE architectures might change the clock speed of the generated processor. However, SIMDify framework itself is independent from the master and slave PEs.

6.2.2. Latency

Speed-up and latency values for each algorithm are given in Table 6.1. Latency without SIMD processing is given in terms of the number of clock cycles, and the latency speedup is calculated as Equation 6.1.

$$Speedup = \frac{Lat_1}{Lat_n} = \frac{Lat_1}{Lat_{serial} + \frac{Lat_{parallel}}{n}} \tag{6.1}$$

150 iteration MVM, MMM, SAD, SSD, KNS, KNQ, KMN and DCT and 75 iteration ANN algorithms are SIMDified with 5, 15, and 25 unroll factors. "Max" is used to show maximum achievable parallelism where $\frac{Lat_{parallel}}{n}$ term goes to 0. KNS, KNQ and KMN algorithms have maximum speed-up around 1.5 and MVM, MMM, SAD, SSD, and DCT algorithms have maximum speed-up between 50 and 100. The

processor does not lose any clock cycles when going into or exiting the SIMD mode. So, calculated latency values are in correspondence with the Amdahl's law [73], Equation 6.2. However, since clock speed is different with and without the slave PEs, it must also be considered when calculating the true speed-up, which is also given separately in Table 6.1.

$$Lat_n = Lat_1 - Lat_{parallel}(1 - \frac{1}{n}) \tag{6.2}$$

Table 6.1. Latency (clock cycles), Clock Speed and Speed-up for unroll factor 5, 15, 25, and maximum achievable parallelism.

Algorithm		MVM	MMM	SAD	SSD	ANN	KNS	KNQ	KMN	DCT
Single PE Latency (Cycles)		6580	9134	10640	8680	14051	13189	9213	42084	13938
Clock Period (ns)	1	12.75	12.75	11.92	12.75	12.75	12.75	12.75	12.75	12.75
	5	12.78	12.78	12.69	12.78	12.78	12.78	12.78	12.78	12.78
	15	12.78	12.78	12.64	12.78	12.78	12.78	12.78	12.78	12.78
	25	12.78	12.78	12.66	12.78	12.78	12.78	12.78	12.78	12.78
Speedup (Cycles)	5	4.63	4.73	4.75	4.72	4.76	1.36	1.60	1.41	4.81
	15	11.75	12.44	12.67	12.40	12.76	1.45	1.78	1.51	9.18
	25	16.96	18.49	19.00	18.39	19.22	1.47	1.83	1.54	20.2
	Max	50.62	68.16	76.00	66.77	79.84	1.49	1.89	1.57	101
Speedup (Time)	5	4.62	4.72	4.46	4.71	4.75	1.35	1.54	1.40	4.80
	15	11.72	12.41	11.95	12.37	12.73	1.44	1.69	1.50	9.16
	25	16.92	18.47	17.89	18.34	19.17	1.46	1.72	1.53	20.1
	Max	50.50	68	71.56	66.58	79.38	1.49	1.78	1.56	100

6.2.3. Area

The resource used for MVM algorithm is given in the Table 6.2. The number of BRAM required for each algorithm is dependent on the number of instructions in the algorithm. The number of LUTs and FFs required is roughly similar for each tested algorithm. The number of DSP blocks required is dependent on the number of multiplication instructions used in the algorithm. So if all of MUL, MULH, MULHSU, MULHU instructions are used, the processor will require 12 DSPs per PE. For applications with no multiplication instructions, such as the SAD algorithm, no DSP blocks are used. DSP usage improved drastically by application specific block removal mentioned in the Scalar PE subsection. With the same technique, BRAM and FF usage does not change, and LUT usage is improved by ∼4%. In all test cases, number of LUTs was the limiting factor in deciding the maximum number of slave PEs (unroll factor). Maximum 25-30 PEs can be implemented inside the Zynq-7020-2CLG484-1 FPGA. It can be seen that DSP increase is linear w.r.t unroll factor. For MVM with unroll factor 25, the application finishes 16.9 times faster by using 8.53 times more BRAM, 25 times more DSP blocks, 12.09 times more FF, and 13.66 times more LUT.

Table 6.2. Resource usage of Matrix Vector Multiplication for unroll factor 5, 15, 25.

Type	Available	Utilization			
		1	5	15	25
BRAM	280	13	46	70	111
DSP	220	3	15	45	75
FF	106400	637	1619	4787	7699
LUT	53200	3406	9958	27649	46541

In MVM, MMM, SAD, SSD, DCT and ANN, we showed experimentally that if the latency is mainly due to partitionable loop, SIMDify can speed-up the design drastically. However, this isn't the case with KNS, KNQ and KMN, which can only be reduced to 60% of single cycle latency due to Amdahl's law. KNS, KNQ and KMN all parallelizes their distance calculation portion inside their algorithm. When using

SIMDify, users must decide if SIMDifing can improve the application, and how much.

It should also be mentioned that Comet [43] also reports around 70 MHz on Artix 7 FPGA and takes around 2 minutes to synthesize. In [43], Rocket Core [71], another core written in Chisel HDL, has been mentioned to have 76 MHZ on Artix 7 FPGA. HL5 article does not indicate its FPGA speed, but it has clock frequency between 700 MHz and 2GHz in 32 nm CMOS. Proposed SIMD processor architecture has a similar clock frequency with aforementioned HLS cores even with 24 slaves, Table 6.1. The solution proposed in this thesis is scalable, open-source, and does not depend on non-standard compilers to minimize the user workload. Using SIMDify, hardware-level parallelization is achieved without the use of additional instructions.

6.3. Algorithms in Detail

6.3.1. MVM

SIMDified part of the matrix vector multiplication algorithm is given in Figure 6.1. 150×4 and *matrix* variable is concatenation of 150×3 array that is multiplied with the 150×1 *vector* common variable and 150×1 array which is holding the result. In this example unroll factor can be dividends of 150 and its latency vs. unroll factor graph is given in Figure 6.2. Here, 150×5 matrix by 150×1 vector multiplication latency is also tested. Tested values are highlighted. Expected lines are calculated using Equation 6.2 using $Lat_1 = 6580$, 7630 and $Lat_{parallel} = 6450$, 7500 for 150×4 and 150×5 cases respectively. Resource utilization is given in Figures 6.3 and 6.4.

```
#define SIZEX 150
#define SIZEY 4
startPar = 0x1;
for (int i = 0; i < SIZEX; i ++) {
   for (int j = 0; j < SIZEY; j ++) {
      // Accumulative addition over result vector
      matrix[i][SIZEY] = matrix[i][SIZEY] + matrix[i][j] * vector[j];
   }
      //printf("result: %d = %d\n", i, matrix[i][SIZEY]);
}
startPar = 0x0;
```

Figure 6.1. SIMDified part of the matrix vector multiplication algorithm

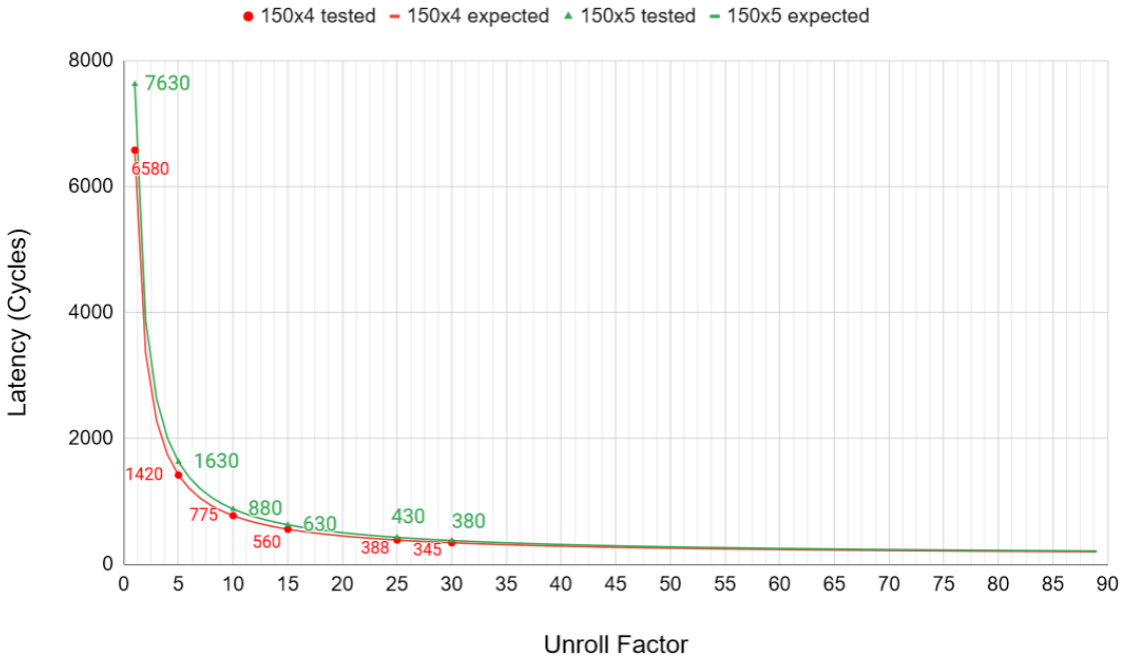

Figure 6.2. Latency vs. unroll factor graph for MVM algorithm

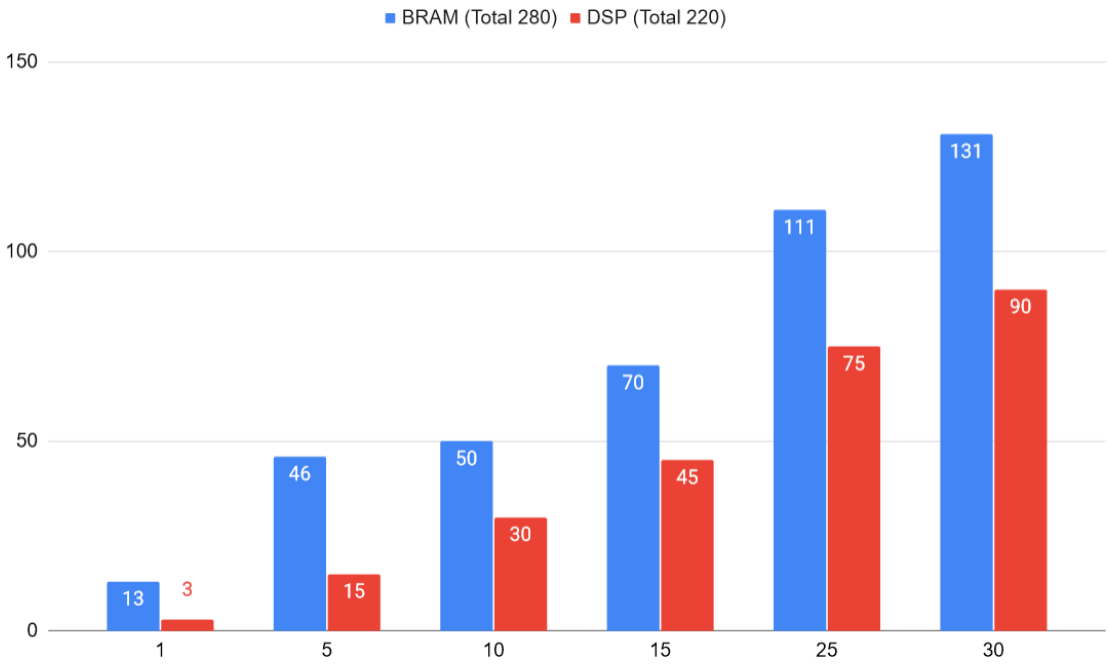

Figure 6.3. BRAM and DSP utilization of MVM algorithm for different unroll factors. All results are taken from Vivado HLS tool.

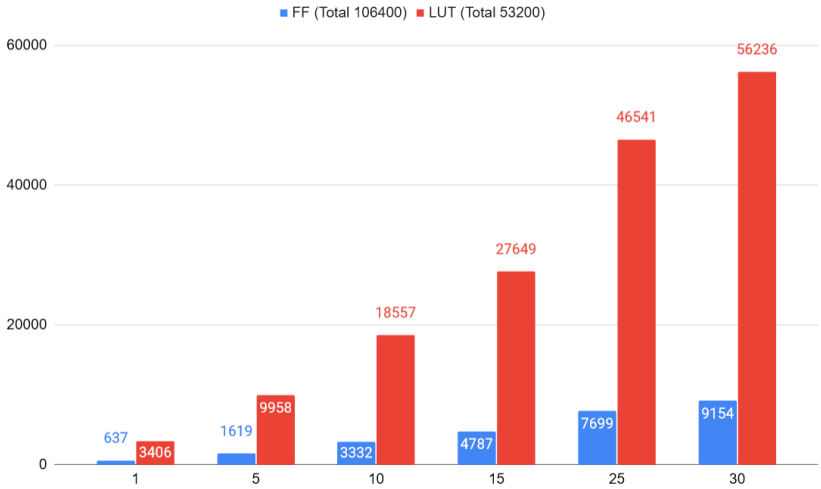

Figure 6.4. FF and LUT utilization of MVM algorithm for different unroll factors. All results are taken from Vivado HLS tool. Note that n=30 LUT result requires more than 100% utilization.

6.3.2. MMM

SIMDified part of the matrix matrix multiplication algorithm is given in Figure 6.5. 150×7 and *matrix* variable is concatenation of 150×5 matrix that is multiplied with the 5×2 *secondmatrix* common variable and 150×2 array which is for holding the result. In this example unroll factor can be dividends of 150 and its latency vs. unroll factor graph is given in Figure 6.6. Here, 150×5 matrix by 5×4 multiplication latency is also tested. Tested values are highlighted. Expected lines are calculated using Equation 6.2 using $Lat_1 = 9134$, 12902 and $Lat_{parallel} = 9000$, 12750 for 5×2 and 5×4 cases respectively. Resource utilization is given in Figures 6.7 and 6.8.

```
#define SIZEX 150
#define SIZEY 5
#define SIZEZ 2
startPar = 0x1;
for (int i = 0; i < SIZEX; i++) { //rowMatrix1
 for (int k = 0; k < SIZEZ; k++){
  for (int j = 0; j < SIZEY; j++) { //ColumnMatrix1 rowMatrix2
  // Accumulative addition over result vectors
  matrix[i][SIZEY+k] = matrix[i][SIZEY+k] + matrix[i][j] * secondmatrix[j][k];
  } } }
startPar = 0x0;
```

Figure 6.5. SIMDified part of the matrix matrix multiplication algorithm

Figure 6.6. Latency vs. unroll factor graph for MMM algorithm

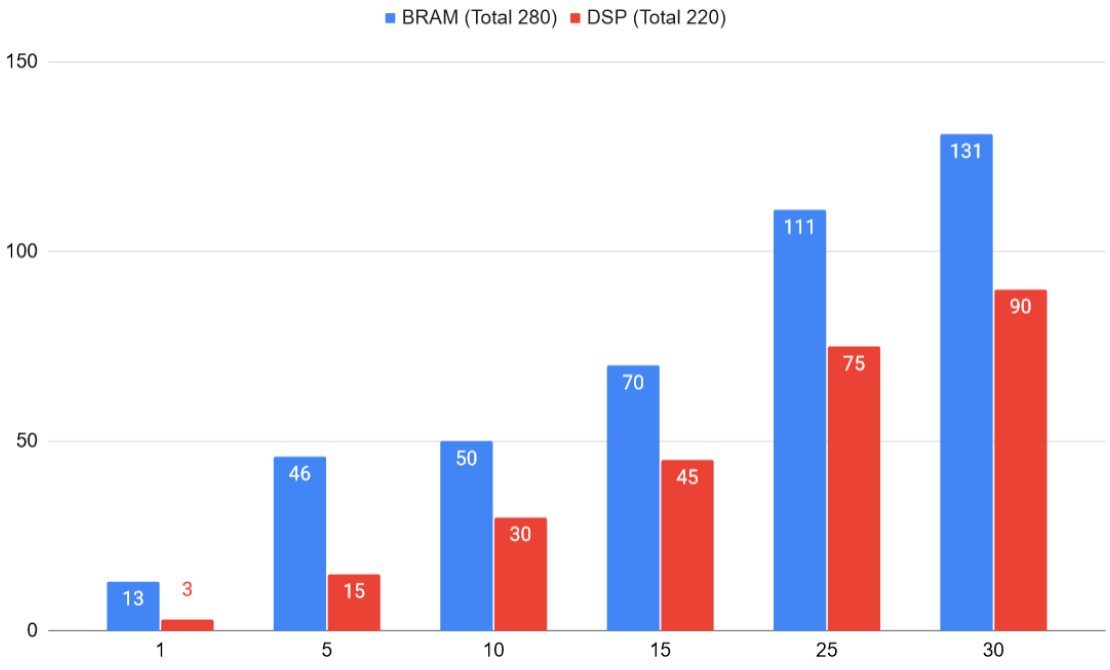

Figure 6.7. BRAM and DSP utilization of MMM algorithm for different unroll factors. All results are taken from Vivado HLS tool.

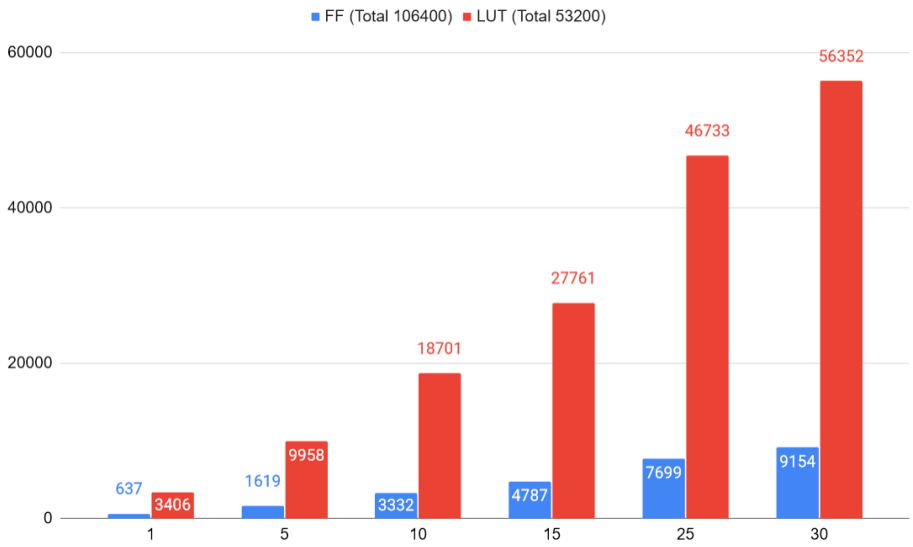

Figure 6.8. FF and LUT utilization of MMM algorithm for different unroll factors. All results are taken from Vivado HLS tool. Note that n=30 LUT result requires more than 100% utilization.

6.3.3. SAD

SIMDified part of the sum of absolute distances algorithm is given in Figure 6.9. In image processing, SAD is used to measure the correlation between two images. It is calculated by subtracting the main image from the the pattern image and getting absolute value of the result. 150×11 A variable is concatenation of 150×5 pixel image array that is compared with the 150×3 pixel pattern array. Result is written to rest of the A, a 150×3 array.

```
#define row_lenght 150
#define pattern_column_lenght 3
#define image_column_lenght 5
int result_overhead = image_column_lenght+pattern_column_lenght;
startPar = 1;
for (int i = 0; i <row_lenght; i++) {
   for (int j = 0; j <= image_column_lenght - pattern_column_lenght; j++) {
      for (int k = 0; k<pattern_column_lenght; k++) {
      A[i][j+result_overhead] += abs(A[i][k] - A[i][k+j+pattern_column_lenght]);
      }
   }
}
startPar = 0;
```

Figure 6.9. SIMDified part of sum of absolute distances algorithm

In this example unroll factor can be dividends of 150 and its latency vs. unroll factor graph is given in Figure 6.10. Here, 150×6 image 150×3 pattern latency is also tested. Tested values are highlighted. Expected lines are calculated using Equation 6.2 using $Lat_1 = 10640$, 12752 and $Lat_{parallel} = 10500$, 12600 for 150×5 and 150×6 cases respectively. Resource utilization is given in Figures 6.11 and 6.12.

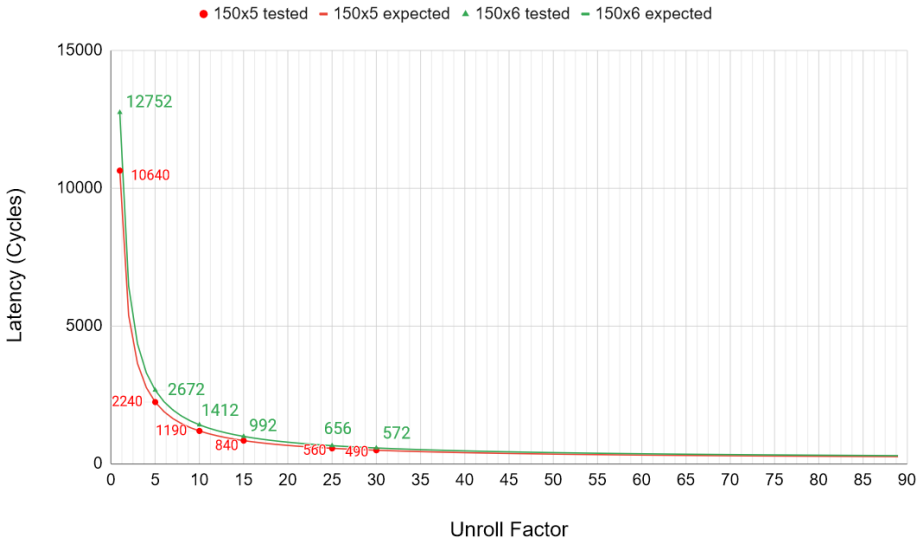

Figure 6.10. Latency vs. unroll factor graph for SAD algorithm

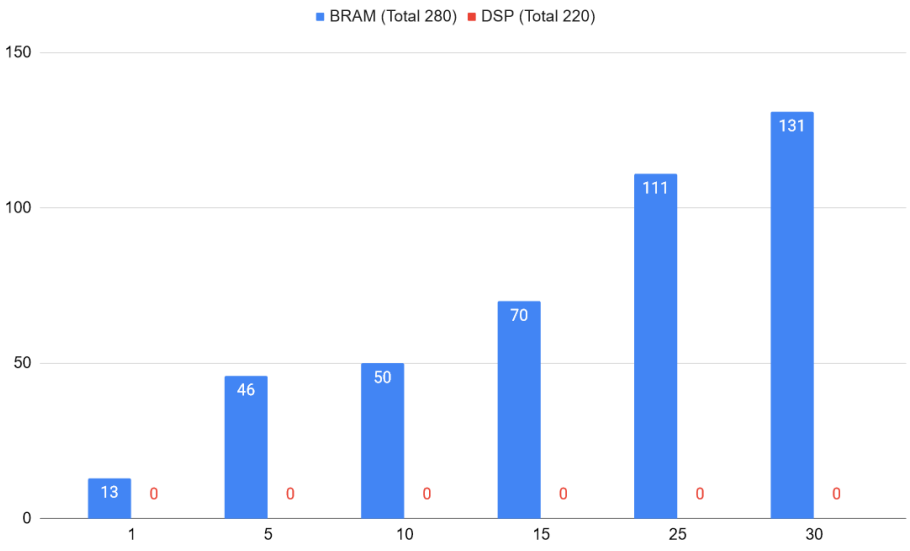

Figure 6.11. BRAM and DSP utilization of SAD algorithm for different unroll factors. All results are taken from Vivado HLS tool. Note that SAD algorithm does not use any multiplication instructions, So DSP utilization is 0%.

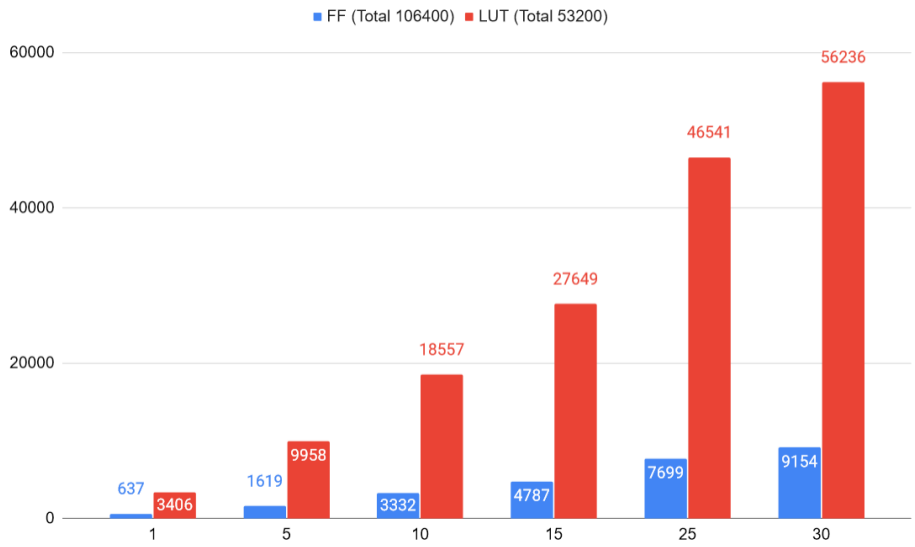

Figure 6.12. FF and LUT utilization of SAD algorithm for different unroll factors. All results are taken from Vivado HLS tool. Note that n=30 LUT result requires more than 100% utilization.

6.3.4. SSD

SIMDified part of the sum of squared distances algorithm is given in Figure 6.13. In image processing, SSD is used to measure the correlation between two images. It is calculated by subtracting the main image from the the pattern image and squaring the result. 150×11 A variable is concatenation of 150×5 pixel image array that is compared with the 150×3 pixel pattern array. Result is written to rest of the A, a 150×3 array.

```
#define row_lenght 150
#define pattern_column_lenght 3
#define image_column_lenght 5
int result_overhead = image_column_lenght+pattern_column_lenght;
startPar = 1;
for (int i = 0; i <row_lenght; i++) {
   for (int j = 0; j <= image_column_lenght - pattern_column_lenght; j++) {
      for (int k = 0; k<pattern_column_lenght; k++) {
         temp = A[i][k] - A[i][k+j+pattern_column_lenght];
         A[i][j+result_overhead] += temp * temp;
      }
   }
}
startPar = 0;
```

Figure 6.13. SIMDified part of the sum of squared distances algorithm

In this example unroll factor can be dividends of 150 and its latency vs. unroll factor graph is given in Figure 6.14. Here, 150×6 image 150×3 pattern latency is also tested. Tested values are highlighted. Expected lines are calculated using Equation 6.2 using $Lat_1 = 8680$, 11086 and $Lat_{parallel} = 8550$, 10950 for 150×5 and 150×6 cases respectively. Resource utilization is given in Figures 6.15 and 6.16.

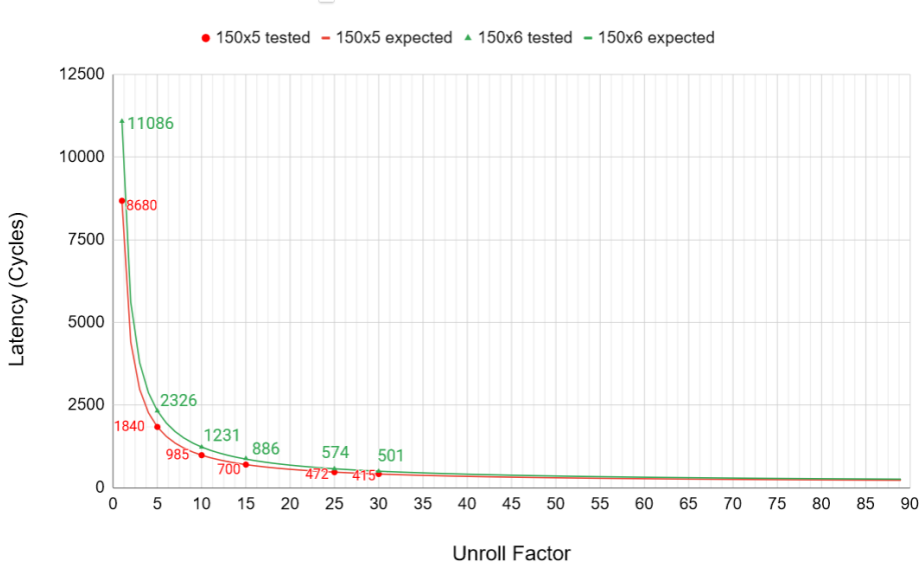

Figure 6.14. Latency vs. unroll factor graph for SSD algorithm

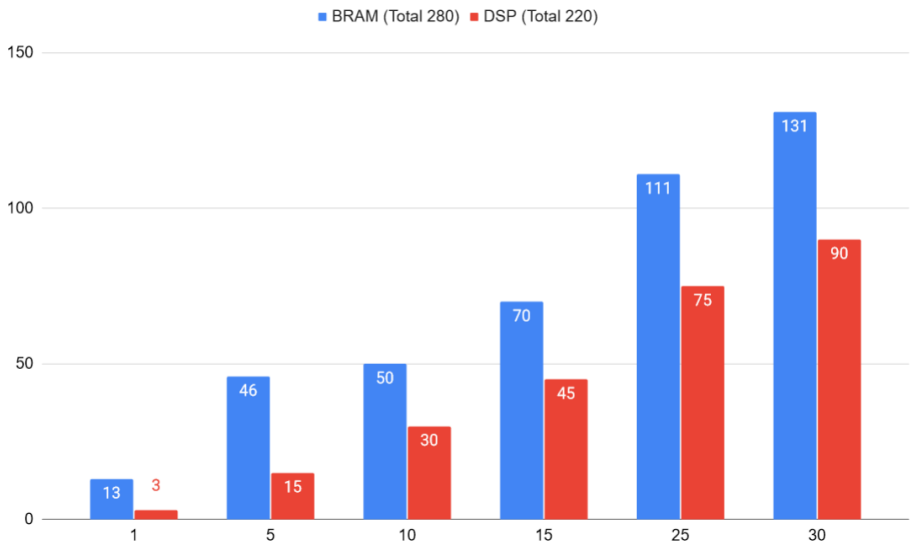

Figure 6.15. BRAM and DSP utilization of SSD algorithm for different unroll factors. All results are taken from Vivado HLS tool.

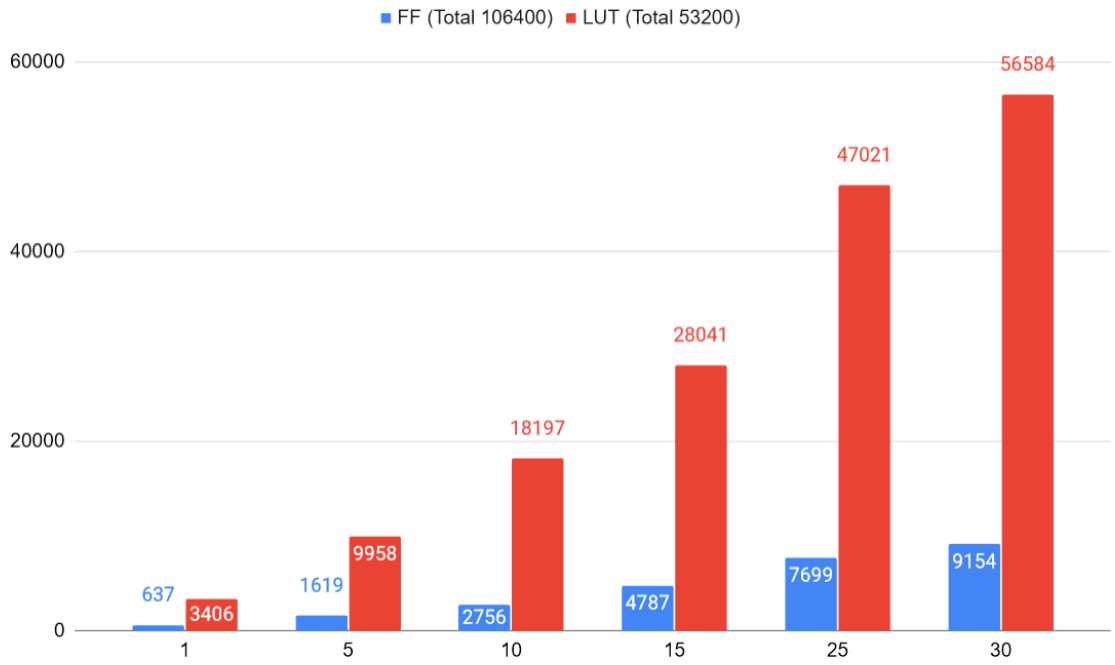

Figure 6.16. FF and LUT utilization of SSD algorithm for different unroll factors. All results are taken from Vivado HLS tool. Note that n=30 LUT result requires more than 100% utilization.

6.3.5. ANN

SIMDified part of the artificial neural networks algorithm is given in Figure 6.17. Dataset used is the banknote authentication dataset taken from UCI Machine Learning Repository [74]. Neural network consists of 4 input neurons, 1 hidden layer with 2 neurons and 2 output neurons. 75×8 *input* variable is concatenation of 75×4 input layer, 75×2 hidden layer and, 75×2 output layer. *weights* common variable is 16 length array and contains $4 \times 2 = 8$ weight and 2 bias values for the first hidden layer and $2 \times 2 = 4$ weight and 2 bias values for the output layer.

Genann_activation function is the activation function. In this example Rectified Linear Unit (ReLU) function is used. Each row of the *input* variable corresponds to different input set and result can be observed in output neuron. In this example unroll factor can be dividends of 75 and its latency vs. unroll factor graph is given in Figure 6.18.

```
#define INPUT_SIZE 75
const int input_neuron_size = 4;
const int hidden_layer_size = 1;
const int hidden_neuron_size = 2;
const int output_neuron_size = 2;
/* Total number of weights, and size of weights buffer. */
const int ann_total_weights = total_weights_g;
void genann_run(volatile int *inputs) {
    int const *w = weights;
    volatile int *o = inputs + input_neuron_size;
    volatile int *i = inputs;
    int h, j, k;
    /* Figure input layer */
    for (j = 0; j < hidden_neuron_size; ++j) {
        int sum = *w++ * -1;
            sum = sum * 256;
        for (k = 0; k < input_neuron_size; ++k) {
            sum += *w++ * i[k];
        }
        *o++ = genann_activation( sum);
    }
    i += input_neuron_size;
    /* Figure hidden layers, if any. */
    for (h = 1; h < hidden_layer_size; ++h) {
        for (j = 0; j < hidden_neuron_size; ++j) {
            int sum = *w++ * -1;
            sum = sum * 256;
            for (k = 0; k < hidden_neuron_size; ++k) {
                sum += *w++ * i[k];
            }
           *o++ = genann_activation( sum);
        }
        i += hidden_neuron_size;
    }
    /* Figure output layer. */
    for (j = 0; j < output_neuron_size; ++j) {
        int sum = *w++ * -1;
            sum = sum * 256;
        for (k = 0; k < hidden_neuron_size; ++k) {
            sum += *w++ * i[k];
        }
        *o++ = genann_activation( sum);
    }
        return ;
}

startPar = 0x1;
   for (int j = 0; j < INPUT_SIZE; j++) {
      genann_run(input[j]);
   }
startPar = 0x0;
```

Figure 6.17. SIMDified part of the artificial neural networks algorithm

In 6.18, 4 input neurons, 2 hidden layer with 2 neurons and 2 output neurons is also tested. Tested values are highlighted. Expected lines are calculated using Equation 6.2 using $Lat_1 = 14051, 20670$ and $Lat_{parallel} = 13875, 20475$ for $4-2-2$ and $4-2-2-2$ cases respectively. Resource utilization is given in Figures 6.19 and 6.20.

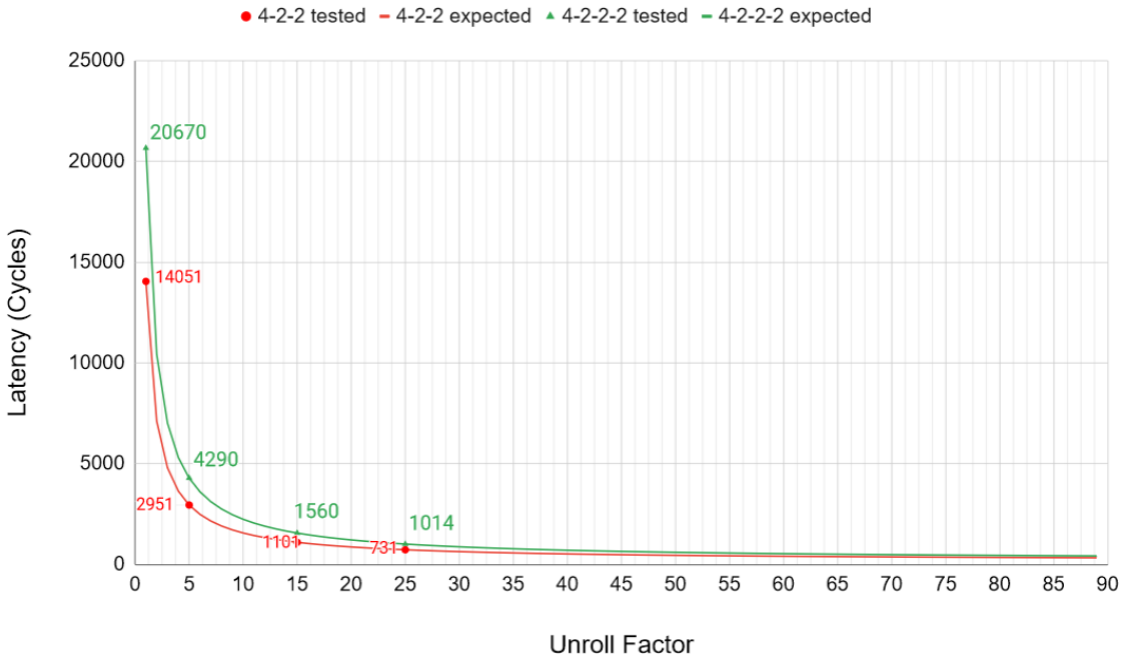

Figure 6.18. Latency vs. unroll factor graph for ANN algorithm

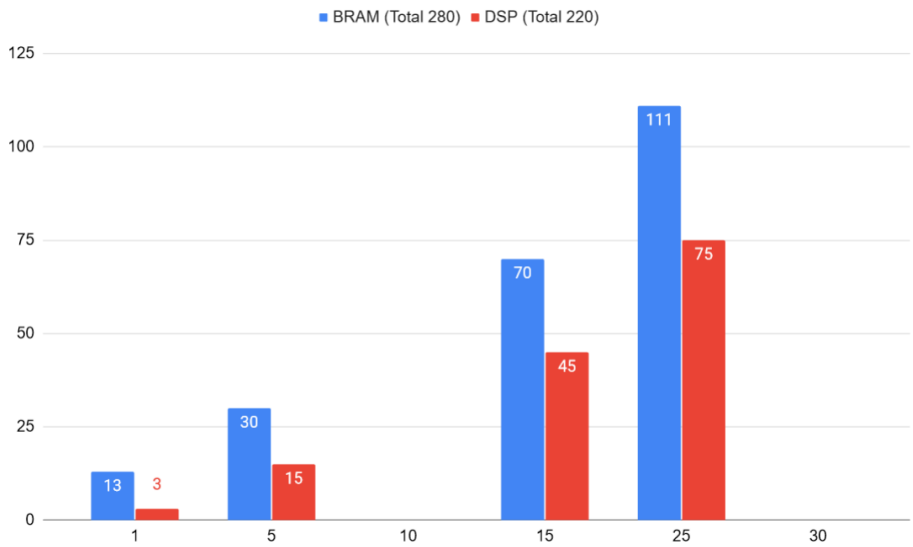

Figure 6.19. BRAM and DSP utilization of ANN algorithm for different unroll factors. All results are taken from Vivado HLS tool.

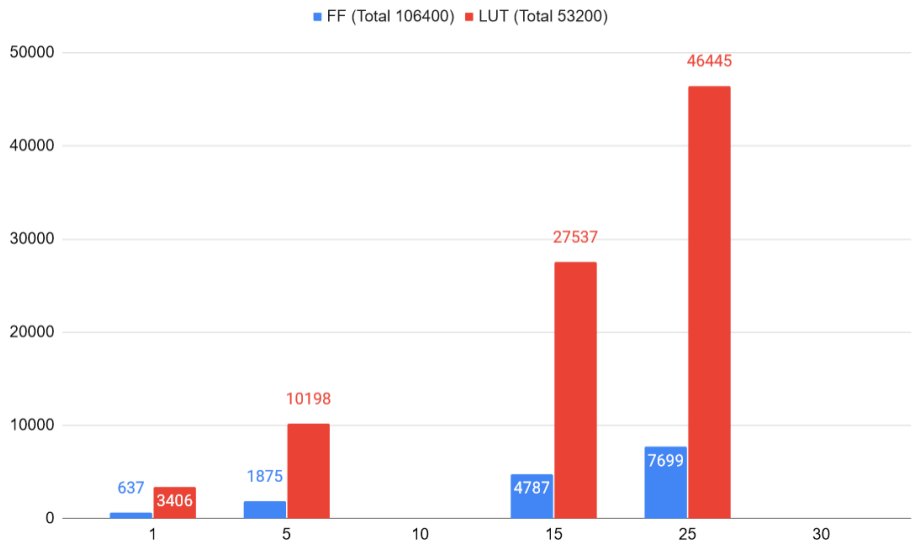

Figure 6.20. FF and LUT utilization of ANN algorithm for different unroll factors. All results are taken from Vivado HLS tool.

6.3.6. KNS

SIMDified part of the k-nearest neighbors with selective sort algorithm is given in Figure 6.21. KNN algorithm classifies a new case using the feature difference with available cases. 150×6 *arr* variable is concatenation of 150×1 feature difference vector, 150×4 feature matrix, and 150×1 class vector. New case p is classified by checking the feature difference of all available cases (arr). In this example unroll factor can be dividends of 150 and its latency vs. unroll factor graph is given in Figure 6.22.

```
void selectionsort(Point* arr, int size, int amount){
    for (int i = 0; i < amount; i++){
        int min = i;
        for (int j = i; j <= size-1; j++){
            if (arr[j].distance < arr[min].distance){
                min = j;
            }
        }
        Point temp = arr[i] ;
        arr[i] = arr[min] ;
        arr[min]  = temp;
    }
}
const int n = 150; // Number of data points
startPar = 0x1;
for (int i = 0; i < n; i++){
    arr[i].distance = (arr[i].x - p.x) * (arr[i].x - p.x)+(arr[i].z - p.z) * (arr[i].z - p.z)
          +(arr[i].k - p.k) * (arr[i].k - p.k) + (arr[i].y - p.y) * (arr[i].y - p.y);
}
startPar = 0x0;
selectionsort(arr, n,k);
```

Figure 6.21. SIMDified part of the k-nearest neighbors with selective sort algorithm

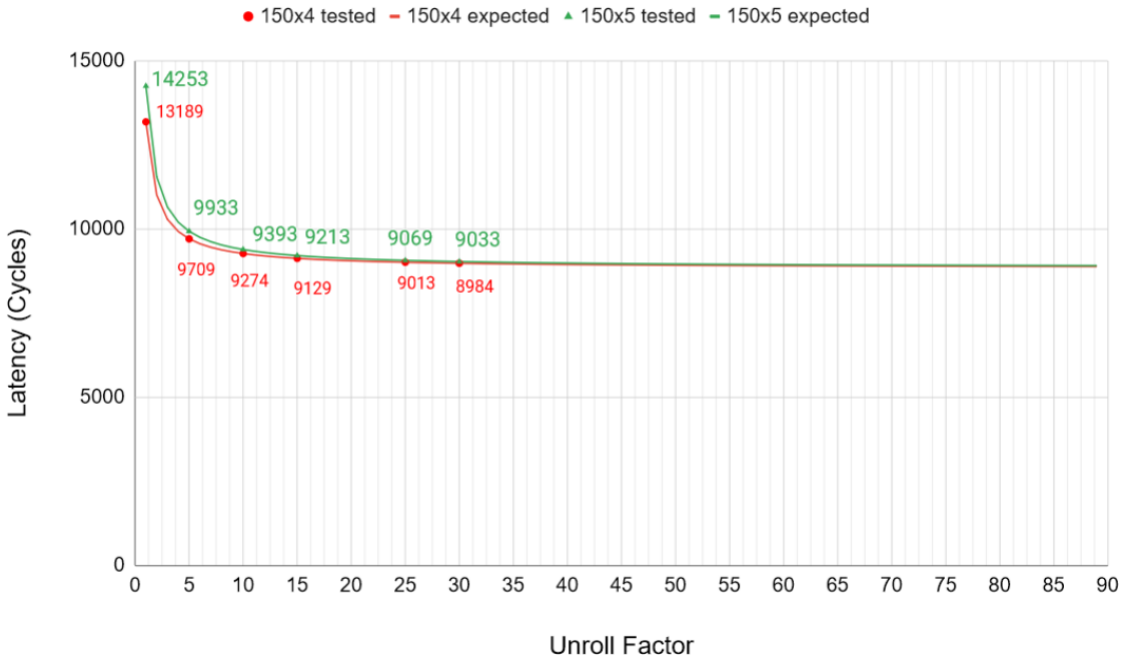

Figure 6.22. Latency vs. unroll factor graph for KNS algorithm

In 6.22, 150×5 feature matrix is also tested. Tested values are highlighted. Expected lines are calculated using Equation 6.2 using $Lat_1 = 13189$, 14253 and $Lat_{parallel} = 4350$, 5400 for 150×4 and 150×5 cases respectively. Resource utilization is given in Figures 6.23 and 6.24.

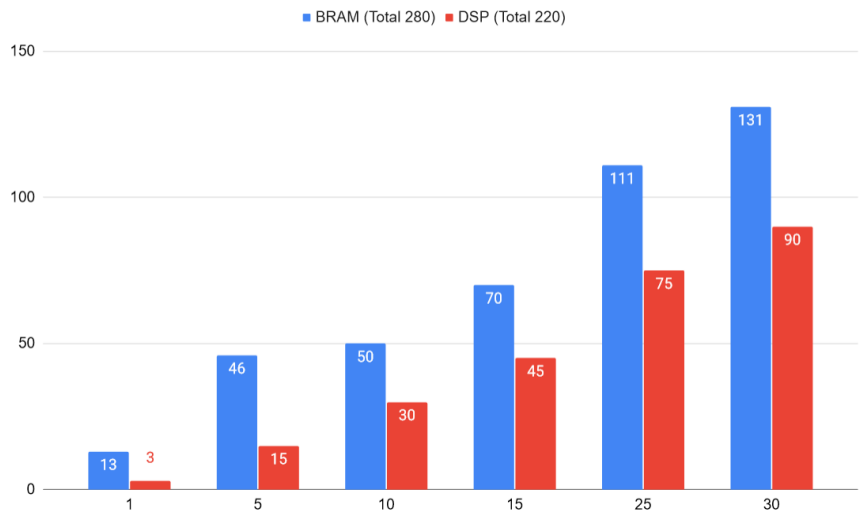

Figure 6.23. BRAM and DSP utilization of KNS algorithm for different unroll factors. All results are taken from Vivado HLS tool.

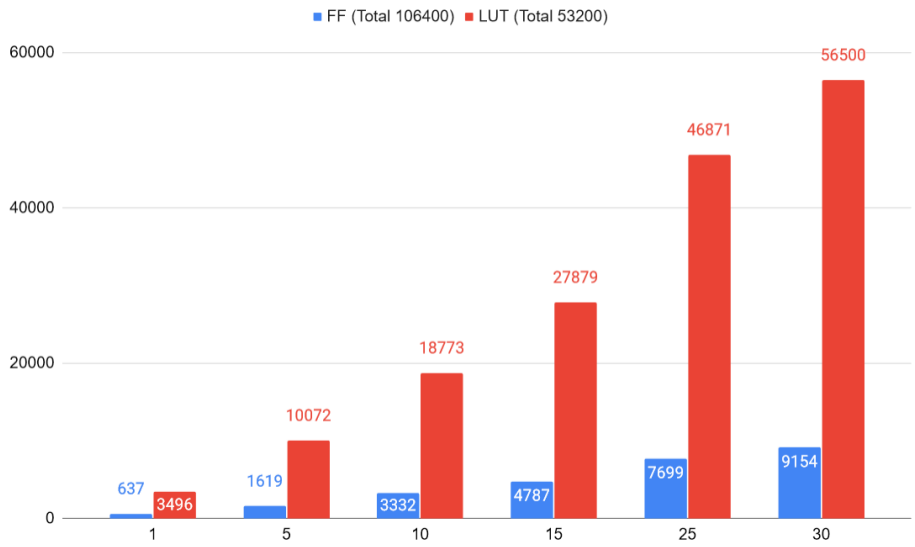

Figure 6.24. FF and LUT utilization of KNS algorithm for different unroll factors. All results are taken from Vivado HLS tool. Note that n=30 LUT result requires more than 100% utilization.

6.3.7. KNQ

SIMDified part of the k-nearest neighbors with qsort algorithm is given in Figure 6.25. Both KNS and KNQ SIMDifies the only distance calculation (SSD part). Only difference between KNS and KNQ is, KNQ uses the C library function *qsort* to sort a calculated array. 150×6 *arr* variable is concatenation of 150×1 distance vector, 150×4 feature matrix, and 150×1 classification vector. New case p is classified by checking the distance of all available cases (arr). In this example unroll factor can be dividends of 150 and its latency vs. unroll factor graph is given in Figure 6.26.

```
int comparison(const void* a, const void* b){
    const Point* p1 = a;
    const Point* p2 = b;
    return (p1->distance - p2->distance);
}
const int n = 150; // Number of data points
startPar = 0x1;
for (int i = 0; i < n; i++){
    arr[i].distance = (arr[i].x - p.x) * (arr[i].x - p.x)+(arr[i].z - p.z) * (arr[i].z - p.z)
             +(arr[i].k - p.k) * (arr[i].k - p.k) + (arr[i].y - p.y) * (arr[i].y - p.y);
}
startPar = 0x0;
// Sort the Points by distance from p
qsort(arr, n, sizeof(Point), &comparison);
```

Figure 6.25. SIMDified part of the k-nearest neighbors with qsort algorithm

Figure 6.26. Latency vs. unroll factor graph for KNQ algorithm

In 6.26, 150×5 feature matrix is also tested. Tested values are highlighted. Expected lines are calculated using Equation 6.2 using $Lat_1 = 9213$, 10641 and $Lat_{parallel} = 4350$, 5400 for 150×4 and 150×5 cases respectively. Resource utilization is given in Figures 6.27 and 6.28.

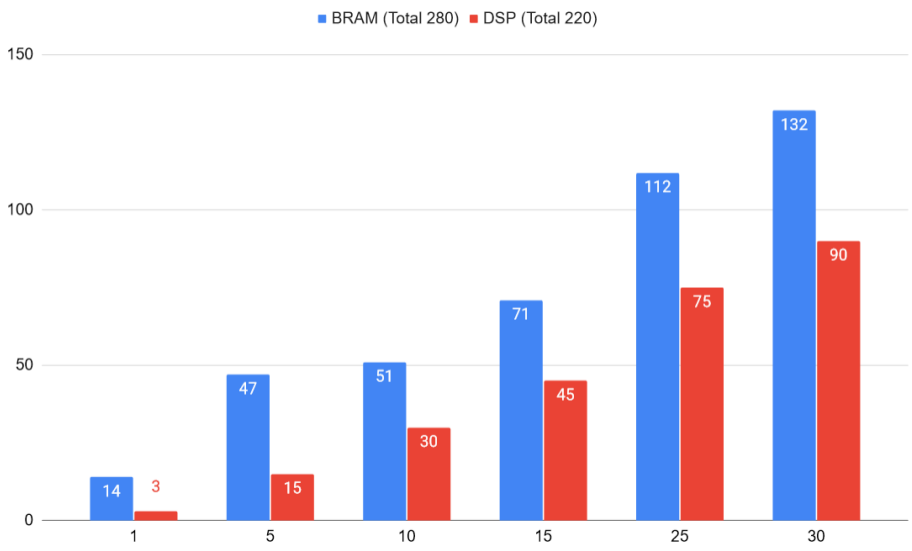

Figure 6.27. BRAM and DSP utilization of KNQ algorithm for different unroll factors. All results are taken from Vivado HLS tool.

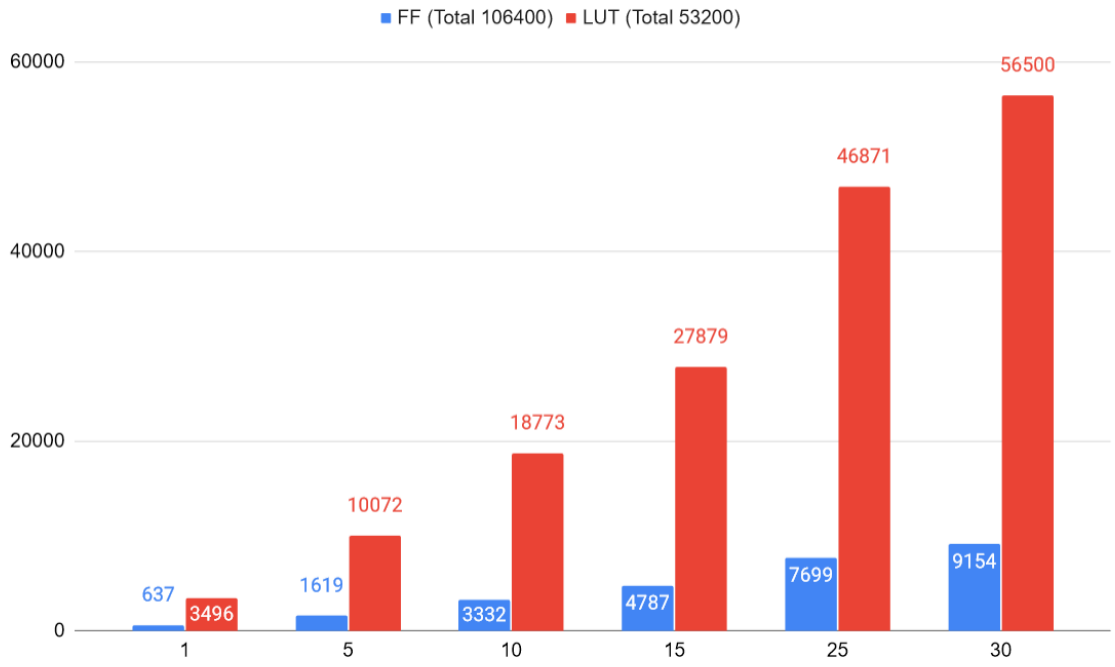

Figure 6.28. FF and LUT utilization of KNQ algorithm for different unroll factors. All results are taken from Vivado HLS tool. Note that n=30 LUT result requires more than 100% utilization.

6.3.8. KMN

In K-Means, 150 item array with 4 features is assigned to 2 clusters. This assignment is done base on their distances. Full code is omitted since it was too long, only SIMDified part ,distance calculation (SSD part) is shown. *arr* is 150×6 length, consists of 150×4 features and 150×1 cluster number and 150×1 distance. In this example unroll factor can be dividends of 150 and its latency vs. unroll factor graph is given in Figure 6.30.

In 6.30, 150 item array with 5 features and 2 clusters is also tested. Tested values are highlighted. Expected lines are calculated using Equation 6.2 using $Lat_1 = 42084$, 51323 and $Lat_{parallel} = 15300$, 15600 for 150×4 and 150×5 cases respectively. Resource utilization is given in Figures 6.31 and 6.32.

```
pick_k_points_as_initial_centroids ();
while (error < old_error  &&  error > tolerance){
startPar = 0x1;
for (int h = 0; h < num_points; h++) {
    for (int i = 0; i < number_of_clusters; i++) {
   for (int j = 0; j < feature_number; j++){
           arr[h][distance_offset+i] = arr[h][distance_offset+i]
           + (arr[h][j] - c[i][j])*(arr[h][j] - c[i][j]);
     }}}
startPar = 0x0;
update_all_centroids_based_on_distances() ;
}
```

Figure 6.29. SIMDified part of the k-means clustering algorithm

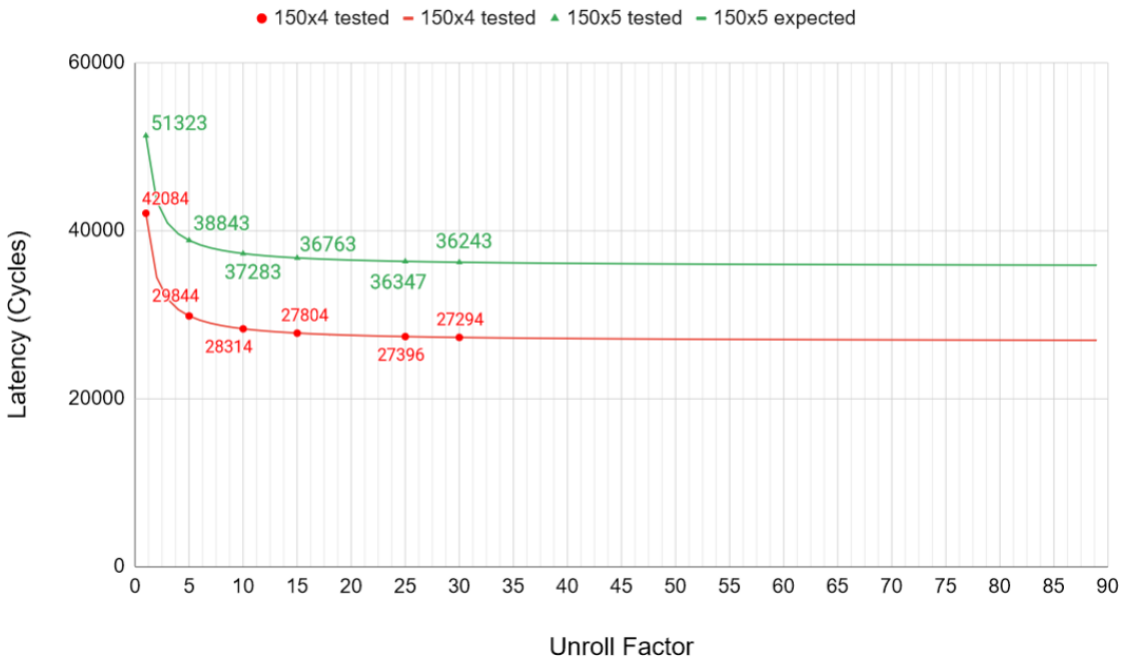

Figure 6.30. Latency vs. unroll factor graph for KMN algorithm

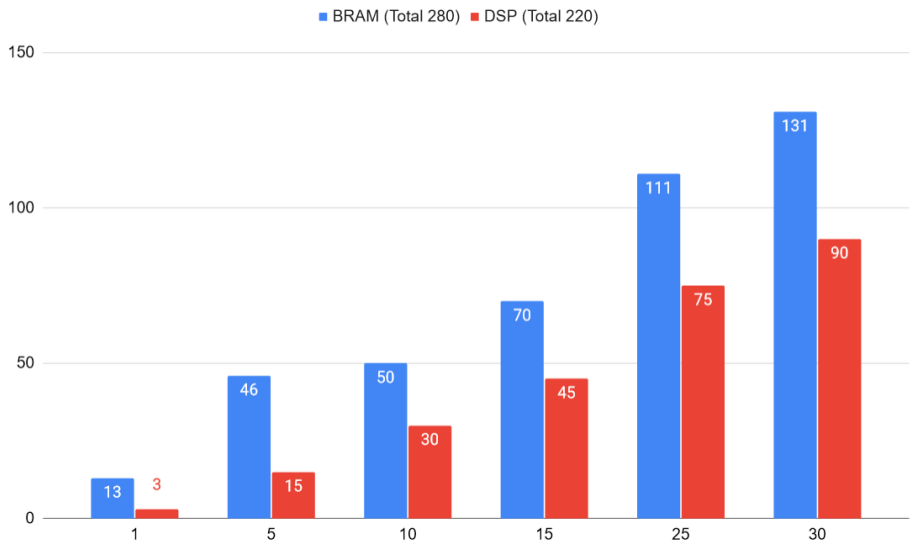

Figure 6.31. BRAM and DSP utilization of KMN algorithm for different unroll factors. All results are taken from Vivado HLS tool.

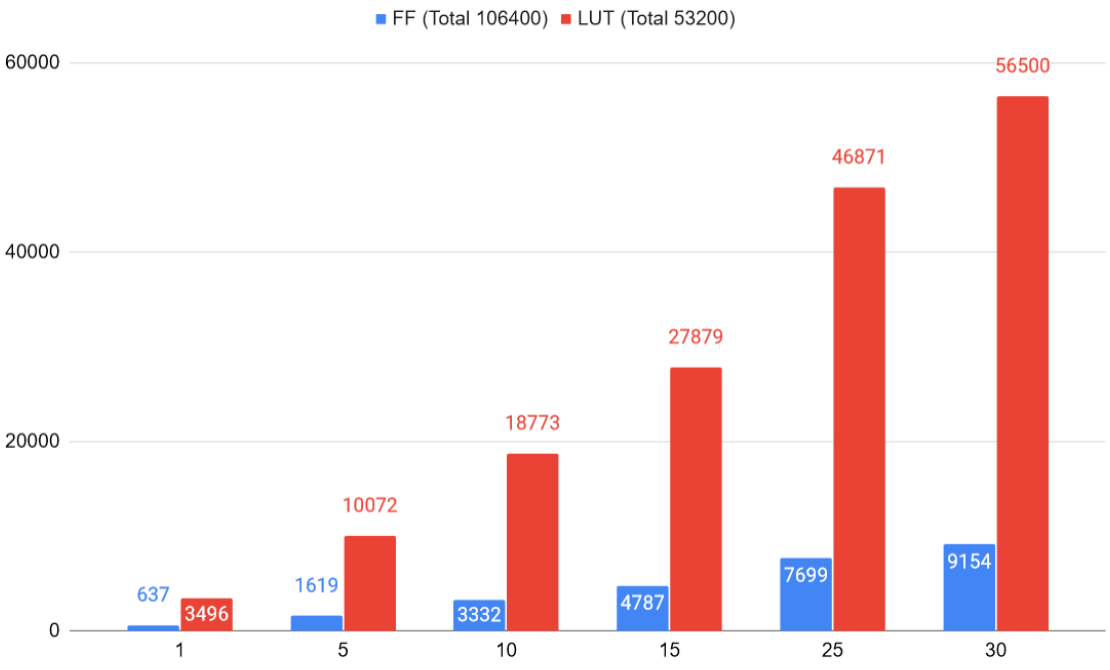

Figure 6.32. FF and LUT utilization of KMN algorithm for different unroll factors. All results are taken from Vivado HLS tool. Note that n=30 LUT result requires more than 100% utilization.

6.3.9. DCT

In DCT, 8 point one dimensional fixed point DCT algorithm is applied to 150 item array with 8 values.

```
startPar = 0x1;
for (int ctr = 0 ; ctr < SIZEX; ctr++) {
    tmp0 = data[ctr][0] + data[ctr][7];
    tmp7 = data[ctr][0] - data[ctr][7];
    tmp1 = data[ctr][1] + data[ctr][6];
    tmp6 = data[ctr][1] - data[ctr][6];
    tmp2 = data[ctr][2] + data[ctr][5];
    tmp5 = data[ctr][2] - data[ctr][5];
    tmp3 = data[ctr][3] + data[ctr][4];
    tmp4 = data[ctr][3] - data[ctr][4];
    /* Even part */
    tmp10 = tmp0 + tmp3; tmp11 = tmp1 + tmp2;/* phase 2 */
    tmp13 = tmp0 - tmp3;  tmp12 = tmp1 - tmp2;
    data[ctr][0] = tmp10 + tmp11; /* phase 3 */
    data[ctr][4] = tmp10 - tmp11;
    z1 = MULTIPLY(tmp12 + tmp13, FIX_0_707106781); /* c4 */
    data[ctr][2] = tmp13 + z1;        /* phase 5 */
    data[ctr][6] = tmp13 - z1;
    /* Odd part */
    tmp10 = tmp4 + tmp5;        /* phase 2 */
    tmp11 = tmp5 + tmp6;    tmp12 = tmp6 + tmp7;
    /* The rotator is modified from fig 4-8 to avoid extra negations. */
    z5 = MULTIPLY(tmp10 - tmp12, FIX_0_382683433); /* c6 */
    z2 = MULTIPLY(tmp10, FIX_0_541196100) + z5; /* c2-c6 */
    z4 = MULTIPLY(tmp12, FIX_1_306562965) + z5; /* c2+c6 */
    z3 = MULTIPLY(tmp11, FIX_0_707106781); /* c4 */
    z11 = tmp7 + z3; z13 = tmp7 - z3;/* phase 5 */
    data[ctr][5] = z13 + z2; data[ctr][3] = z13 - z2;/* phase 6 */
    data[ctr][1] = z11 + z4; data[ctr][7] = z11 - z4;
}
startPar = 0x0;
```

Figure 6.33. 1-dimensional 8 element Discrete Cosine Transform algorithm

We didn't compile generalized DCT algorithm since the generated SIMD processor does not use DIV instruction. *data* is 150×8 length, consists of 150×8 data. In this example unroll factor can be dividends of 150 and its latency vs. unroll factor graph is given in Figure 6.34. Tested values are highlighted. Expected lines are calculated using Equation 6.2 using $Lat_1 = 13938$ and $Lat_{parallel} = 13800$. Resource utilization is given in Figures 6.35 and 6.36.

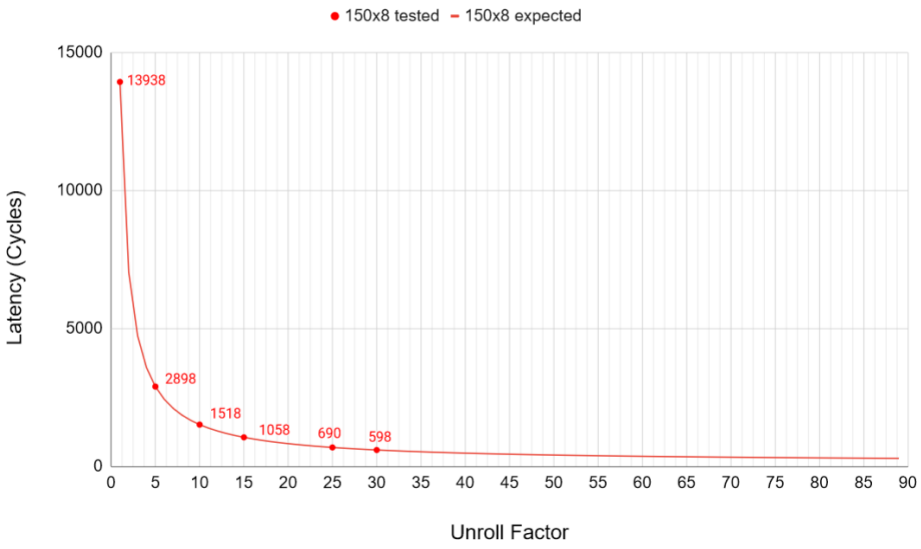

Figure 6.34. Latency vs. unroll factor graph for DCT algorithm

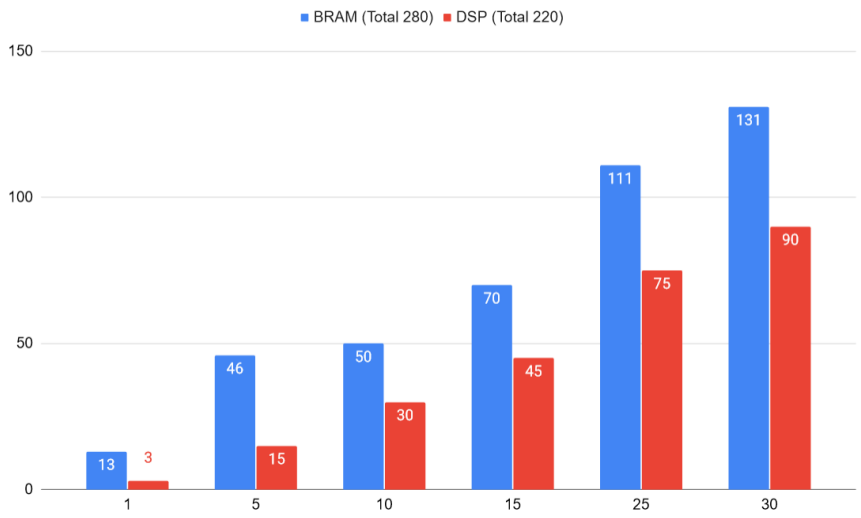

Figure 6.35. BRAM and DSP utilization of DCT algorithm for different unroll factors. All results are taken from Vivado HLS tool.

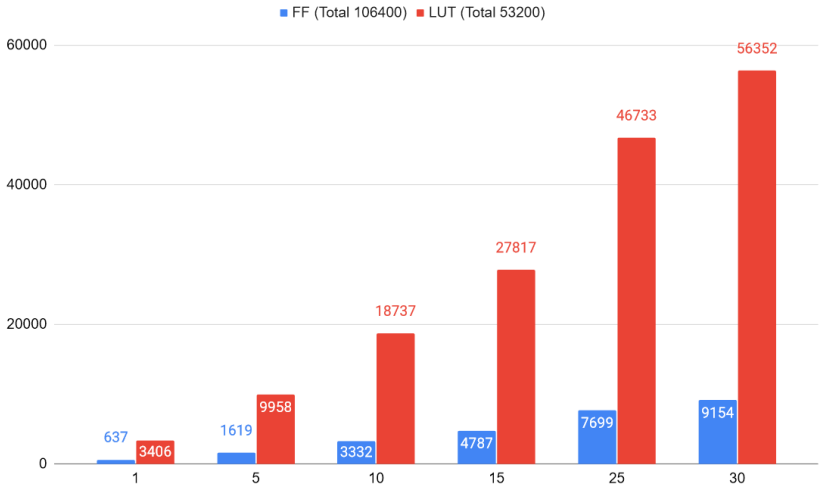

Figure 6.36. FF and LUT utilization of DCT algorithm for different unroll factors. All results are taken from Vivado HLS tool. Note that n=30 LUT result requires more than 100% utilization.

6.3.10. Multiple Parallel Loops

With SIMDify, it is possible to parallelize multiple loops. Clock period of multiple loops is 12.84 ns, which is slightly larger than the single loop period. SIMDified loops cannot be nested, data in the loops cannot be dependent, and if the same data is SIMDified they must have same unroll factor. As a proof of concept multiple parallel loops is tested with artificial neural networks algorithm. ANN consists of an input layer with 4 neurons, 2 hidden layers with 3 neurons each and an output layer with 3 neurons, Figure 6.37. Each layer will be parallelized in itself. So there will be three loops with factor 3 each, Figure 6.38.

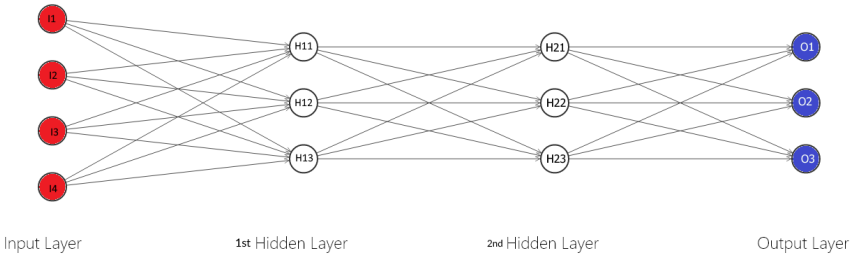

Figure 6.37. Diagram of the Neural Network. Drawn using free tool [2].

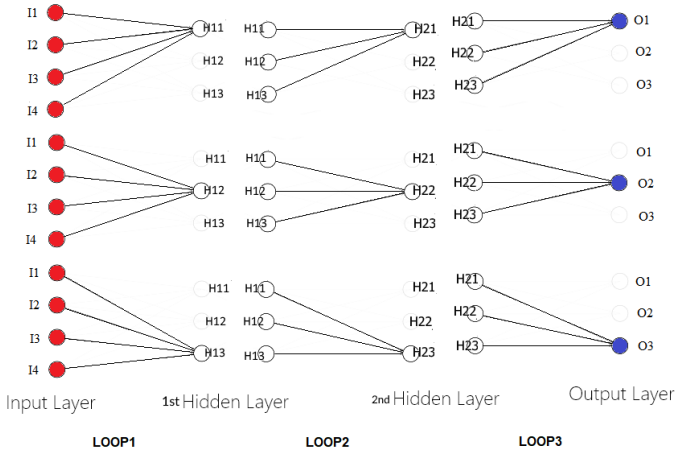

Figure 6.38. Parsed diagram of the Neural Network. All three loops and all 9 PEs viewpoint is shown.

In this example unroll factor can be dividends of 3 and its latency vs. unroll factor graph is given in Figure 6.39. Note that, in Chapter 6.3.5, 75 inputs are parallelized and each PE was executing the same ANN for different input sets. In this Chapter, ANN itself is parallelized and there is only single input set and three partitioned loops with separate Startpar's, Figure 6.40. In code we used pragma "#GCC unroll 0" to stop compiler from unrolling SIMDified loops. In the third loop, nop operation inserted since there was no suitable Init_simd_offset PC value for switching to SIMD processing mode. So processor overrides the "nop" operation instead. Resource utilization is given in Figures 6.41 and 6.42.

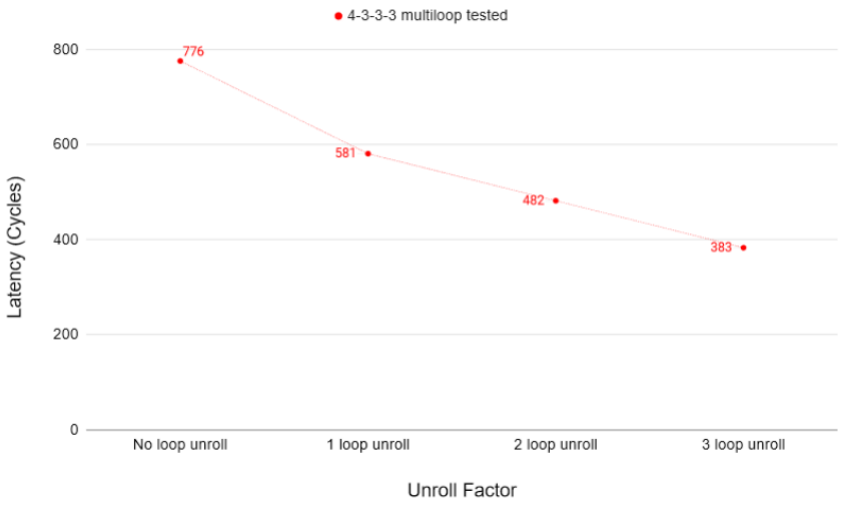

Figure 6.39. Latency vs. unroll factor graph for multiloop ANN algorithm

```
void genann_run(volatile int *inputs) {
    volatile int *i = inputs;
    if (!ann_hidden_layers) {//Input layer if no hidden layers
        for (int j = 0; j < ann_outputs; ++j) {
            output_weights_arr[j][1] = output_weights_arr[j][0]  * -1;
            output_weights_arr[j][1] = output_weights_arr[j][1] * 256;
            for (int k = 0; k < ann_inputs; ++k) {
                output_weights_arr[j][1] += output_weights_arr[j][k+1] * i[k];
            }
            output_weights_arr[j][1]= genann_activation( output_weights_arr[j][1]);
        }
        return ;
    }

    arr_str =  &input_weights_arr[0] ;
    arr_end =  &input_weights_arr[hidden_g-1] + (&input_weights_arr[hidden_g-1] - \\
        &input_weights_arr[hidden_g-2]);
    startPar = 0x1;
    /* Input layer */
    #pragma GCC unroll 0
    for (int j = 0; j < ann_hidden; ++j) {
        input_weights_arr[j][1] = input_weights_arr[j][0] * -1;
        input_weights_arr[j][1] = input_weights_arr[j][1] * 256;
        for (int k = 0; k < ann_inputs; ++k) {
            input_weights_arr[j][1] += input_weights_arr[j][k+1] * i[k];
        }
        input_weights_arr[j][1] = genann_activation( input_weights_arr[j][1]);
    }
    startPar = 0x0;

    for (int j = 0; j < ann_hidden; ++j) {
        hidden_weights_arr[0][j][1] = input_weights_arr[j][1];
    }
 /* Hidden layers, if any. */
 for (int h = 0; h < ann_hidden_layers-1; ++h) {
     arr_str =  &hidden_weights_arr[h][0] ;
     arr_end =  &hidden_weights_arr[h][hidden_g-1] + (&hidden_weights_arr[h][hidden_g-1] - \\
         &hidden_weights_arr[h][hidden_g-2]);
     startPar = 0x1;
     #pragma GCC unroll 0
     for (int j = 0; j < ann_hidden; ++j) {
         hidden_weights_arr[h][j][2] = hidden_weights_arr[h][j][0] * -1;
         hidden_weights_arr[h][j][2] = hidden_weights_arr[h][j][2] * 256;
         for (int k = 0; k < ann_hidden; ++k) {
             hidden_weights_arr[h][j][2] += hidden_weights_arr[h][j][k+1] * \\
             hidden_weights_arr[h][k][1];
         }
        hidden_weights_arr[h][j][2] = genann_activation( hidden_weights_arr[h][j][2]);
     }
     startPar = 0x0;
 }

 /* Output layer. */
 arr_str =  &output_weights_arr[0] ;
 arr_end =  &output_weights_arr[outputs_g-1] + (&output_weights_arr[outputs_g-1] -\\
     &output_weights_arr[outputs_g-2]);
 startPar = 0x1;
 asm volatile ("nop");
 #pragma GCC unroll 0
 for (int j = 0; j < ann_outputs; ++j) {
     output_weights_arr[j][1] = output_weights_arr[j][0] * -1;
         output_weights_arr[j][1] = output_weights_arr[j][1] * 256;
         for (int k = 0; k < ann_hidden; ++k) {
         output_weights_arr[j][1] += output_weights_arr[j][k+2] * \\
         hidden_weights_arr[hidden_layers_g-2][k][1];
     }
     output_weights_arr[j][1] = genann_activation( output_weights_arr[j][1]);
 }
 startPar = 0x0;
 return ;
```

Figure 6.40. 1-dimensional 8 element Discrete Cosine Transform algorithm

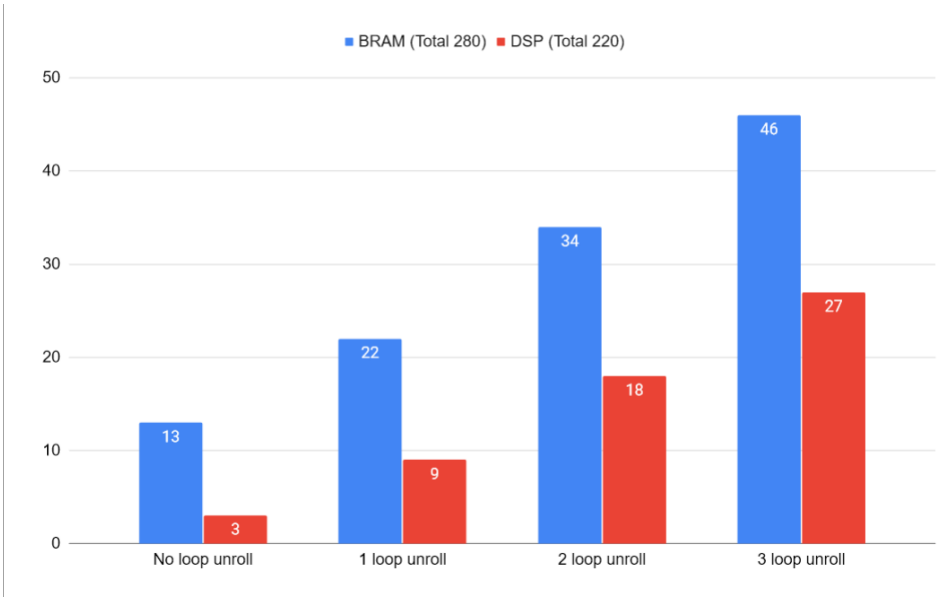

Figure 6.41. BRAM and DSP utilization of multiloop ANN algorithm for different unroll factors. All results are taken from Vivado HLS

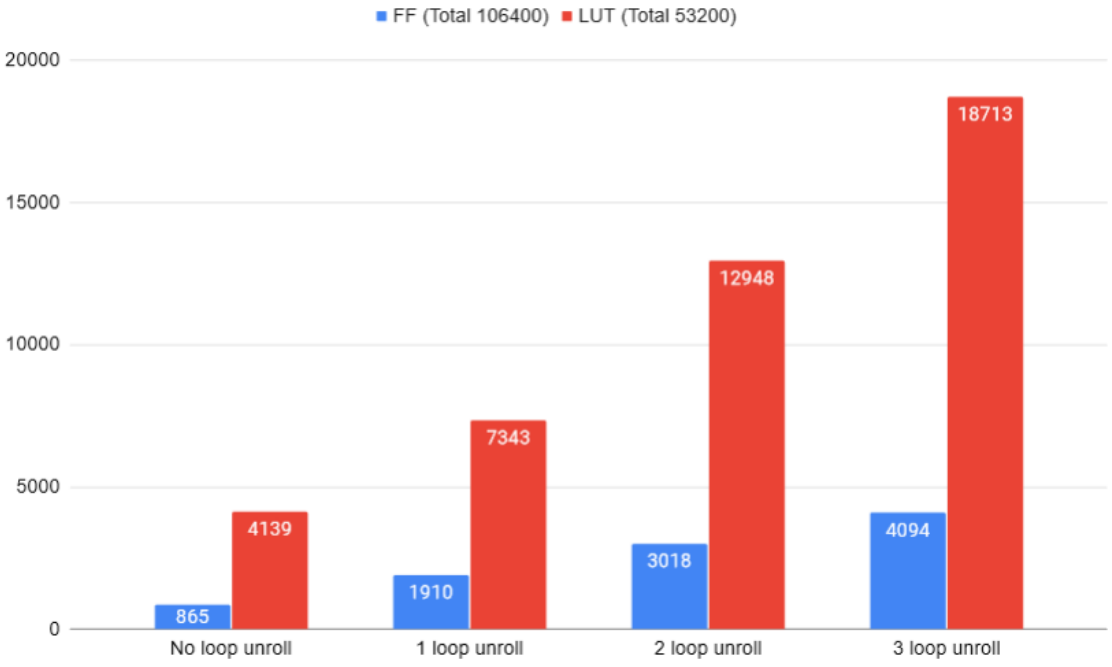

Figure 6.42. FF and LUT utilization of multiloop ANN algorithm for different unroll factors. Note that all results are taken from Vivado HLS tool, and n=30 LUT result requires more than 100% utilization.

7. CONCLUSION AND THE FUTURE WORK

In this thesis, SIMDify, hardware-software parallelization framework for generating SIMD capable application-specific RISC-V instruction set processors, and the generated application specific SIMD processor structure are presented. SIMDify combines HLS with standard RISC-V compiler to generate a five-stage pipelined SIMD processor written in C++. SIMD processor consists of master and slave PEs. Using HLS, SIMDify combines and connects these PEs to generate different SIMD processors for each application. SIMD processor architecture is the first HLS designed RISC-V processor with SIMD support. System runs on Zynq-7020-2CLG484-1 FPGA and it operates in approximately 78 MHz.

Applicability of the SIMDify is tested on selected algorithms. System runs on Zynq-7020-2CLG484-1 FPGA and it operates at approximately 78 MHz. Processor is designed for an FPGA as the target hardware, so it can be combined with other applications as an accelerator. Since it's designed in HLS, it can be easily modified and improved by many users.

In terms of scalar PE, cache implementation can be improved. Also, forwarding structure can be implemented to reduce the number of stalls, and multi-cycle instructions such as DIV and REM can be implemented for full $riscv32im$ support. The main bottleneck of the core is 11 ns single cycle 32x32 multiplication instruction, which can be improved by using a custom multiplication block or supporting multi-cycle multiplication operation.

Existing external memory and cache structure can be used to increase the total data memory size. However, data in the external memory cannot be used in SIMD processing. So, to increase the size of the SIMD processed memory, tag field size must also increase. Since, SIMDification block generates constant memory tags for every word in the local data memory, generated tag field size increases proportionally with

the local data field size. This problem can be solved by decreasing tag field size per word or by changing the memory structure to extend local memory without increasing the tag size.

Designed ASIP and SIMDify framework can be applied to any iterative loop if the loop does not include any conditional branching and if the loop satisfies the memory constraints. We believe SIMDify solution is better and more comprehensive than the alternative: modifying an each application to make it compatible with each custom instruction. SIMDify automates processor generation and creates open source framework that can easily be used by anyone to achieve SIMD computation. Even though there are some limitations in current HLS tools, the time to design the custom SIMD processor has significantly decreased compared to traditional RTL flow.

REFERENCES

1. Waterman, A., Y. Lee, D. A. Patterson and K. Asanovi, "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA", *Electrical Engineering*, Vol. I, pp. 1–34, 2011.

2. *NN Drawing Tool*, `http://alexlenail.me/NN-SVG/`, accessed in December 2020.

3. Dey, S. and P. D. Franzon, "An Application Specific Processor Architecture with 3D Integration for Recurrent Neural Networks", *20th International Symposium on Quality Electronic Design (ISQED)*, pp. 183–190, 2019.

4. Bikos, A. N. and N. Sklavos, "Architecture Design of an Area Efficient High Speed Crypto Processor for 4G LTE", *IEEE Transactions on Dependable and Secure Computing*, Vol. 15, No. 5, pp. 729–741, 2018.

5. Gerrish, P., E. Herrmann, L. Tyler and K. Walsh, "Challenges and constraints in designing implantable medical ICs", *IEEE Transactions on Device and Materials Reliability*, Vol. 5, No. 3, pp. 435–444, 2005.

6. Park, C., P. H. Chou, Y. Bai, R. Matthews and A. Hibbs, "An ultra-wearable, wireless, low power ECG monitoring system", *IEEE 2006 Biomedical Circuits and Systems Conference Healthcare Technology, BioCAS 2006*, pp. 241–244, 2006.

7. Kansakar, P. and A. Munir, "A Two-Tiered Heterogeneous and Reconfigurable Application Processor for Future Internet of Things", *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 690–696, 2018.

8. Malkowsky, S., H. Prabhu, L. Liu, O. Edfors and V. Öwall, "A programmable 16-lane SIMD ASIP for massive MIMO", *Proceedings - IEEE International Symposium on Circuits and Systems*, Vol. 2019-May, pp. 1–5, 5 2019.

9. Wang, Y. and Y. Ha, "A Performance and Area Efficient ASIP for Higher-Order DPA-Resistant AES", *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, Vol. 4, No. 2, pp. 190–202, 6 2014, http://ieeexplore.ieee.org/document/6800115/.

10. Figuli, P., C. Tradowsky, N. Gaertner and J. Becker, "ViSA: A highly efficient slot architecture enabling multi-objective ASIP cores", *2013 International Symposium on System on Chip (SoC)*, pp. 1–8, IEEE, 10 2013, http://ieeexplore.ieee.org/document/6675270/.

11. Sugiura, T., S. Nakatsuka, J. Yu, Y. Takeuchi and M. Imai, "An efficient data compression method for artificial vision systems and its low energy implementation using ASIP technology", *2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings*, pp. 81–84, IEEE, 10 2014, http://ieeexplore.ieee.org/document/6981650/.

12. Li, M., F. Naessens, M. Li, P. Debacker, C. Desset, P. Raghavan, A. Dejonghe and L. Van der Perre, "A processor based multi-standard low-power LDPC engine for multi-Gbps wireless communication", *2013 IEEE Global Conference on Signal and Information Processing*, pp. 1254–1257, IEEE, 12 2013, http://ieeexplore.ieee.org/document/6737136/.

13. Lee, C.-M., Y.-J. Huang, C.-W. Liu and Y. Hsu, "DeAr: A framework for power-efficient and flexible embedded digital signal processor design", *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 658–661, IEEE, 10 2016, http://ieeexplore.ieee.org/document/7804083/.

14. Kultala, H., T. Viitanen, H. Berg, P. Jaaskelainen, J. Multanen, M. Kokkonen, K. Raiskila, T. Zetterman and J. Takala, "LordCore: Energy-Efficient OpenCL-Programmable Software-Defined Radio Coprocessor", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 27, No. 5, pp. 1029–1042, 5 2019, https://ieeexplore.ieee.org/document/8653500/.

15. Shanlin Xiao, D. Li, H. Kunieda and T. Isshiki, "Design of an efficient ASIP-based processor for object detection using AdaBoost algorithm", *2016 7th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES)*, pp. 96–99, IEEE, 3 2016, `http://ieeexplore.ieee.org/document/7467129/`.

16. Huo, Y. and D. Liu, "High-Throughput Area-Efficient Processor for Cryptography", *Chinese Journal of Electronics*, Vol. 26, No. 3, pp. 514–521, 5 2017.

17. Chidambaram, S., A. Riviello, J. PierreLanglois and J.-P. David, "Accelerating the Inference Phase in Ternary Convolutional Neural Networks Using Configurable Processors", *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 94–99, IEEE, 10 2018, `https://ieeexplore.ieee.org/document/8596860/`.

18. Michalakes, J., "HPC for Weather Forecasting", A. Grama and A. H. Sameh (Editors), *Parallel Algorithms in Computational Science and Engineering*, pp. 297–323, Springer International Publishing, Cham, 2020.

19. Porcù, R., E. Miglio, N. Parolini, M. Penati and N. Vergopolan, "HPC simulations of brownout: A noninteracting particles dynamic model", *The International Journal of High Performance Computing Applications*, Vol. 34, No. 3, pp. 267–281, 5 2020.

20. Garcia-Gasulla, M., F. Mantovani, M. Josep-Fabrego, B. Eguzkitza and G. Houzeaux, "Runtime mechanisms to survive new HPC architectures: A use case in human respiratory simulations", *The International Journal of High Performance Computing Applications*, Vol. 34, No. 1, pp. 42–56, 1 2020.

21. Liu, G., J. Primmer and Z. Zhang, "Rapid Generation of High-Quality RISC-V Processors from Functional Instruction Set Specifications", *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, ACM, NY, USA, 6 2019.

22. Becker, A., S. Sirowy and F. Vahid, "Just-in-time compilation for FPGA processor cores", *2011 Electronic System Level Synthesis Conference (ESLsyn)*, c, pp. 1–6, IEEE, 6 2011.

23. Tsai, T.-H., Y.-C. Ho and M.-H. Sheu, "Implementation of FPGA-based Accelerator for Deep Neural Networks", *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pp. 1–4, IEEE, 4 2019, https://ieeexplore.ieee.org/document/8724665/.

24. Gao, C., S. Braun, I. Kiselev, J. Anumula, T. Delbruck and S.-C. Liu, "Real-Time Speech Recognition for IoT Purpose using a Delta Recurrent Neural Network Accelerator", *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 5 2019, https://ieeexplore.ieee.org/document/8702290/.

25. Chen, Y., K. Zhang, C. Gong, C. Hao, X. Zhang, T. Li and D. Chen, "T-DLA: An Open-source Deep Learning Accelerator for Ternarized DNN Models on Embedded FPGA", *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 13–18, IEEE, 7 2019, https://ieeexplore.ieee.org/document/8839554/.

26. Lara-Nino, C. A., A. Diaz-Perez and M. Morales-Sandoval, "Lightweight elliptic curve cryptography accelerator for internet of things applications", *Ad Hoc Networks*, Vol. 103, p. 102159, 6 2020, https://linkinghub.elsevier.com/retrieve/pii/S1570870519306924.

27. Liu, Z., J. Jiang, G. Lei, K. Chen, B. Qin and X. Zhao, "A Heterogeneous Processor Design for CNN-Based AI Applications on IoT Devices", *Procedia Computer Science*, Vol. 174, pp. 2–8, 2020, https://linkinghub.elsevier.com/retrieve/pii/S1877050920315611.

28. Hao, Y., Z. Fang, G. Reinman and J. Cong, "Supporting Address Translation for Accelerator-Centric Architectures", *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 37–48, IEEE, 2 2017.

29. *SIMDify Framework*, https://github.com/alpsark/SIMDify, accessed in December 2020.

30. *Vivado High-Level Syntesis*, https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, accessed in December 2020.

31. Jain, M., M. Balakrishnan and A. Kumar, "ASIP design methodologies: survey and issues", *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, pp. 76–81, IEEE Comput. Soc, 2000.

32. *Vivado Design Suite User Guide: High-Level Synthesis*, www.xilinx.com/support/documentation/sw_manuals/xilinx2013_2/, accessed in December 2020.

33. *Zedboard FPGA*, https://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board/, accessed in December 2020.

34. Flynn, M. J., "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, Vol. C-21, No. 9, pp. 948–960, 9 1972.

35. Kimura, Y., 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)*, pp. 435–439, IEEE, 11 2019, https://ieeexplore.ieee.org/document/8951530/.

36. Garofalo, A., 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)*, pp. 186–191, IEEE, 3 2020, https://ieeexplore.ieee.org/document/9116529/.

37. Tagliavini, G., 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)*, pp. 654–657, IEEE, 3 2019, https://ieeexplore.ieee.org/document/8714897/.

38. Ganesh, V. and B. V. Sandilya, "Implementation of SIMD Instruction Set Extension for BLAKE2", *2019 10th International Conference on Computing, Communication and Networking Technologies, ICCCNT 2019*, 2019.

39. *Codasip Studio*, `https://codasip.com/codasip-studio/`, accessed in December 2020.

40. *ASIP Designer*, `https://www.synopsys.com/designware-ip/processor-solutions/asips-tools.html`, accessed in December 2020.

41. *Chipyard*, `https://chipyard.readthedocs.io/en/latest/Customization/RoCC-Accelerators.html`, accessed in December 2020.

42. Mantovani, P., R. Margelli, D. Giri and L. P. Carloni, "HL5: A 32-bit RISC-V Processor Designed with High-Level Synthesis", *Proceedings of the Custom Integrated Circuits Conference*, Vol. 2020-March, 2020.

43. Rokicki, S., D. Pala, J. Paturel and O. Sentieys, "What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications", *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Vol. 2019-Novem, pp. 1–8, IEEE, 11 2019.

44. Taştan, I., M. Karaca and A. Yurdakul, "Approximate CPU Design for IoT End-Devices with Learning Capabilities", *Electronics*, Vol. 9, No. 1, p. 125, 1 2020.

45. Skalicky, S., T. Ananthanarayana, S. Lopez and M. Lukowiak, "Designing customized ISA processors using high level synthesis", *2015 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015*, pp. 0–5, 2016.

46. Ahmed, T., N. Sakamoto, J. Anderson and Y. Hara-Azumi, "Synthesizable-from-C embedded processor based on MIPS-ISA and OISC", *Proceedings - IEEE/IFIP 13th International Conference on Embedded and Ubiquitous Computing, EUC 2015*, pp. 114–123, 2015.

47. Canis, A., J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown and T. Czajkowski, "LegUp", *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11*, p. 33, ACM Press, New York, USA, 2011.

48. Munshi, A., "The OpenCL specification", *2009 IEEE Hot Chips 21 Symposium (HCS)*, pp. 1–314, IEEE, 8 2009.

49. Cherupalli, H., H. Duwe, W. Ye, R. Kumar and J. Sartori, "Bespoke Processors for Applications with Ultra-low Area and Power Constraints", *ACM SIGARCH Computer Architecture News*, Vol. 45, No. 2, pp. 41–54, 2017, `http://dl.acm.org/citation.cfm?doid=3140659.3080247`.

50. Matai, J., J. Oberg, A. Irturk, T. Kim and R. Kastner, "Trimmed VLIW: Moving application specific processors towards high level synthesis", *Electronic System Level Synthesis Conference (ESLsyn), 2012*, pp. 11–16, IEEE, 2012.

51. Lampret, D., C.-M. Chen, M. Mlinar, J. Rydberg, M. Ziv-Av, C. Ziomkowski, G. McGary, B. Gardner, R. Mathur and M. Bolado, "OpenRISC 1000 architecture manual", *Description of assembler mnemonics and other for OR1200*, 2003.

52. Asanović, K. and D. A. Patterson, *Instruction Sets Should Be Free: The Case For RISC-V*, Tech. Rep. UCB/EECS-2014-146, EECS Department, University of California, Berkeley, 2014.

53. *RISC-V Foundation (ISA)*, `https://riscv.org/`, accessed in December 2020.

54. *Shakti-Open Source Processor Development Ecosystem*, `http://shakti.org.in/`, accessed in December 2020.

55. *PULP Platform*, `https://pulp-platform.org/`, accessed in December 2020.

56. *SiFive*, `https://www.sifive.com/`, accessed in December 2020.

57. *SweRV*, github.com/chipsalliance/Cores-SweRV, accessed in December 2020.

58. *Open, Lowest Power, Programmable RISC-V Solutions*, www.microsemi.com/product-directory/mi-v-embedded-ecosystem/4406-risc-v-cpus, accessed in December 2020.

59. *Rumble Development Corp.*, www.rumbledev.com, accessed in December 2020.

60. *A 32-bit Microcontroller featuring a RISC-V core*, https://github.com/onchipuis/mriscv, accessed in December 2020.

61. *DarkRISCV*, github.com/darklife/darkriscv, accessed in December 2020.

62. *C910 High-performance 64-bit RISC-V architecture multi-core processor with AI vector acceleration engine*, https://www.t-head.cn/product/c910?spm=a2ouz.12987052.0.0.5c5c6245WIbjoG, accessed in December 2020.

63. *BOOM: Berkeley Out-of-Order Machine*, https://github.com/riscv-boom/riscv-boom, accessed in December 2020.

64. *EPI: Accelerator Stream*, https://www.european-processor-initiative.eu/accelerator/, accessed in December 2020.

65. *OpenTitan*, https://github.com/lowRISC/opentitan, accessed in December 2020.

66. Chen, C., X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, C. Li, Y. Pu, J. Meng, X. Yan, Y. Xie and X. Qi, "Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product", *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 52–64, IEEE, 5 2020, https://ieeexplore.ieee.org/document/9138983/.

67. Celio, C., P. F. Chiu, K. Asanović, B. Nikolić and D. Patterson, "BROOM: An Open-Source Out-of-Order Processor with Resilient Low-Voltage Operation in 28-nm CMOS", *IEEE Micro*, Vol. 39, No. 2, pp. 52–60, 2019.

68. Wang, A., W. Bae, J. Han, S. Bailey, O. Ocal, P. Rigge, Z. Wang, K. Ramchandran, E. Alon and B. Nikolic, "A Real-Time, 1.89-GHz Bandwidth, 175-kHz Resolution Sparse Spectral Analysis RISC-V SoC in 16-nm FinFET", *IEEE Journal of Solid-State Circuits*, Vol. 54, No. 7, pp. 1993–2008, 7 2019.

69. *Rocket Chip Project*, `https://github.com/freechipsproject/rocket-chip`, accessed in December 2020.

70. Bachrach, J., H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language", *Proceedings - Design Automation Conference*, pp. 1216–1225, 2012.

71. Asanovic, K., R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo and A. Waterman, "The Rocket Chip Generator", *EECS Department, University of California, Berkeley, Technical Report*, 2016.

72. Harris, D. M. and S. L. Harris, *Digital Design and Computer Architecture (2nd edition)*, Morgan Kaufmann, 2012.

73. Rodgers, D. P., "Improvements in multiprocessor system design", *ACM SIGARCH Computer Architecture News*, Vol. 13, No. 3, pp. 225–231, 6 1985.

74. *UCI Machine Learning Repository*, `https://archive.ics.uci.edu/ml/`, accessed in December 2020.

APPENDIX A: ACM PUBLISHING LICENCE

ACM Publishing License and Audio/Video Release

Title of the Work: SIMDify: Framework for SIMD-Processing with RISC-V Scalar Instruction Set: SIMDify
Submission ID:62

Author/Presenter(s): Mehmet Alp Sarkisla:Bogazici University;Arda Yurdakul:Bogazici University

Type of material: Full Paper

Publication and/or Conference Name: Australasian Computer Science Week Multiconference Proceedings

1. Glossary

2. Grant of Rights

(a) Owner hereby grants to ACM an exclusive, worldwide, royalty-free, perpetual, irrevocable, transferable and sublicenseable license to publish, reproduce and distribute all or any part of the Work in any and all forms of media, now or hereafter known, including in the above publication and in the ACM Digital Library, and to authorize third parties to do the same.

(b) In connection with software and "Artistic Images and "Auxiliary Materials, Owner grants ACM non-exclusive permission to publish, reproduce and distribute in any and all forms of media, now or hereafter known, including in the above publication and in the ACM Digital Library.

(c) In connection with any "Minor Revision", that is, a derivative work containing less than twenty-five percent (25%) of new substantive material, Owner hereby grants to ACM all rights in the Minor Revision that Owner grants to ACM with respect to the Work, and all terms of this Agreement shall apply to the Minor Revision.
(d) If your paper is withdrawn before it is published in the ACM Digital Library, the rights revert back to the author(s).

☑ A. Grant of Rights. I grant the rights and agree to the terms described above.

☐ B. Declaration for Government Work. I am an employee of the national government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are you a contractor of your National Government? ○ Yes ◉ No

Are any of the co-authors, employees or contractors of a National Government? ○ Yes ◉ No

---

3. Reserved Rights and Permitted Uses.

(a) All rights and permissions the author has not granted to ACM in Paragraph 2 are

reserved to the Owner, including without limitation the ownership of the copyright of the Work and all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM in Paragraph 2(a), Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, (3) any repository legally mandated by an agency funding the research on which the Work is based, and (4) any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work.

When preparing your paper for submission using the ACM TeX templates, the rights and permissions information and the bibliographic strip must appear on the lower left hand portion of the first page.

The new ACM Consolidated TeX template Version 1.3 and above automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.

NOTE: For authors using the ACM Microsoft Word Master Article Template and Publication Workflow, The ACM Publishing System (TAPS) will add the rights

statement to your papers for you. Please check with your conference contact for information regarding submitting your source file(s) for processing.

Please put the following LaTeX commands in the preamble of your document - i.e., before \begin{document}:

\copyrightyear{2021}
\acmYear{2021}
\setcopyright{acmlicensed}\acmConference[ACSW '21]{Australasian Computer Science Week Multiconference}{February 1--5, 2021}{Dunedin, New Zealand}
\acmBooktitle{Australasian Computer Science Week Multiconference (ACSW '21), February 1--5, 2021, Dunedin, New Zealand}
\acmPrice{15.00}
\acmDOI{10.1145/3437378.3444364}
\acmISBN{978-1-4503-8956-3/21/02}

NOTE: For authors using the ACM Microsoft Word Master Article Template and Publication Workflow, The ACM Publishing System (TAPS) will add the rights statement to your papers for you. Please check with your conference contact for information regarding submitting your source file(s) for processing.

If you are using the ACM Interim Microsoft Word template, or still using or older versions of the ACM SIGCHI template, you must copy and paste the following text block into your document as per the instructions provided with the templates you are using:

ACSW '21, February 1–5, 2021, Dunedin, New Zealand

ACM ISBN 978-1-4503-8956-3/21/02...$15.00
https://doi.org/10.1145/3437378.3444364

NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library. Once you have your camera ready copy ready, please send your source files and PDF to your event contact for processing.

4. ACM Citation and Digital Object Identifier.

(a) In connection with any use by the Owner of the Definitive Version, Owner shall include the ACM citation and ACM Digital Object Identifier (DOI).
(b) In connection with any use by the Owner of the Submitted Version (if accepted) or the Accepted Version or a Minor Revision, Owner shall use best efforts to display the ACM citation, along with a statement substantially similar to the following:

> "© [Owner] [Year]. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in {Source Publication}, https://doi.org/10.1145/{number}."

5. Audio/Video Recording

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? ○ Yes ◉ No

6. Auxiliary Material

Do you have any Auxiliary Materials? ○ Yes ◉ No

7. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

◉ We/I have not used third-party material.
○ We/I have used third-party materials and have necessary permissions.

8. Artistic Images

If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part IV and be sure to include a notice of copyright with each such image in the paper.
◉ We/I do not have any artistic images.
○ We/I have any artistic images.

9. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☑ I agree to the Representations, Warranties and Covenants.

10. Enforcement.

At ACM's expense, ACM shall have the right (but not the obligation) to defend and enforce the rights granted to ACM hereunder, including in connection with any instances of plagiarism brought to the attention of ACM. Owner shall notify ACM in writing as promptly as practicable upon becoming aware that any third party is infringing upon the rights granted to ACM, and shall reasonably cooperate with ACM in its defense or enforcement.

11. Governing Law

This Agreement shall be governed by, and construed in accordance with, the laws of the state of New York applicable to contracts entered into and to be fully performed therein.

Funding Agents

1. TUBITAK award number(s):58135

DATE: **12/18/2020** sent to alp.sarkisla@boun.edu.tr at **14:12:41**