FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Software Loop Unrolling Specialization for Hardware Accelerators

Tomás Pacheco Pires



Mestrado em Engenharia Informática e Computação

Supervisor: Prof. João Manuel Paiva Cardoso

September 18, 2026

Software Loop Unrolling Specialization for Hardware Accelerators

Tomás Pacheco Pires

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Francisco António Ferraz Martins de Almeida Maia
Examiner: Prof. Vanderlei Bonato
Supervisor: Prof. João Manuel Paiva Cardoso

September 18, 2026

Resumo

Aceleradores de hardware, como os FPGAs, são complementos importantes a CPUs, devido aos seus baixos consumos de energia e alta performance em tarefas especializadas, e a síntese de alto nível (HLS) tem um papel importante em ligar estes dois elementos, uma vez que permite a engenheiros de software o uso destes aceleradores; no entanto, estas ferramentas requerem bastante intervenção manual para aplicar estratégias de otimização para núcleos de hardware, o que implica conhecimento em áreas fora do domínio tradicional do desenvolvimento de software. Este trabalho descreve um método semi-automático para a otimização e especialização fonte-a-fonte em C de núcleos de hardware. Este método consiste na conversão de uma função em código fonte C como entrada num grafo de fluxo de controlo e dados (CDFG) por injeção de código de instrumentação e execução, que representa o fluxo da execução e as dependências de dados do progama, em forma desenrolada; este grafo é depois convertido de volta numa função de código C que reflete esta estratégia de desenrolamento. Resultados experimentais em relação a diferentes graus de especialização e otimização com um número de núcleos de hardware distintos demonstram que o nosso método resulta num compromisso grande em que recursos de hardware são usados liberalmente de modo a obter a latência mínima possível, por vezes em detrimento da velocidade máxima de relógio, e que apesar dos nossos resultados serem geralmente iguais aos obtidos com diretivas HLS equivalentes, a nossa substituição de código condicional por operações ternárias demonstrou resultados promissores em relação a performance ao nível dos recursos de hardware, escabilidade e velocidade de execução, relativamente ao método existente.

Abstract

Hardware accelerators, such as FPGAs, act as important complements to CPUs for their low energy consumption and high performance for specialised tasks, and high level synthesis (HLS) is an important bridge for these two, as it allows for software engineers to exploit these accelerators; however, these tools often require manual intervention for the application of optimisation strategies for hardware kernels, which requires expertise uncommon within the field of software programming. This work describes a method for semi-automatic C source-to-source specialisation and optimisation of hardware kernels. This approach relies on conversion of an input C source function into a control and data flow graph (CDFG) in runtime through the injection of instrumentation code, which represents an unrolled representation of the program's execution flow and data dependencies; this graph is then converted back into a C code function that reflects this unrolling strategy. Experimental results against different degrees of specialisation and optimisation with several different hardware kernels with different properties show that our approach results in a large compromise in which FPGA hardware resources are used liberally to achieve the minimum latency possible, sometimes at the cost of maximum clock speed, and that while these results largely match with those of equivalent existing HLS directives, our substitution of conditional code with ternary operators provided promising overall improvements in hardware resource, scalability and execution performance over the existing method.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

The work developed for this dissertation aims to further some of these goals through its contributions to the larger academic community. These goals are described below, alongside how the work furthers them and how well it accomplishes that.

The specific Sustainable Development Goals mentioned have the following names:

SDG 9 Industry, Innovation and Infrastructure

SDG 12 Responsible Consumption and Production

SDG 17 Partnership for the Goals

SGD	Target	Contribution	Performance Indicators and Metrics
9	9.1	Faciliate adoption and optimsiation of FPGAs for highly parallelisable tasks, such as artificial intelligence and big data processing.	Promising avenues in optimisation of specialised hardware kernels were identified, particularly the use of ternary operators for conditional code.
12	12.1	Increase adoption of FPGAs through improvement of development technologies, which are able to perform specialised tasks with less energy consumption than typical CPUs.	The work's practical usefulness is limited in its current state, but the approaches developed within show promise for future enhancements.
17	17.1	Promote cooperation between software and hardware engineering disciplines through lowering the knowledge barrier for software engineers to contribute to the developement of hardware accelerators.	Current state of toolchain still requires too much manual intervention to be useful, though these can be automatised in the future.

Contents

1 Introduction **1**
- 1.1 Context and Motivation . . . 1
- 1.2 Problem Definition . . . 1
- 1.3 Contributions . . . 2
- 1.4 Research Questions . . . 2
- 1.5 Methodology . . . 3
- 1.6 Dissertation Structure . . . 3

2 Literature Review **4**
- 2.1 Collection Phase . . . 4
- 2.2 Trimming Phase . . . 5
- 2.3 State of the Art . . . 5
  - 2.3.1 The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs 5
  - 2.3.2 OmpSs@FPGA framework for high performance FPGA computing . . . 6
  - 2.3.3 A Source-to-Source Compiler to Enable Hybrid Scheduling for High-Level Synthesis . . . 8
  - 2.3.4 Graph-Based Code Restructuring Targeting HLS for FPGAs . . . 10
  - 2.3.5 On Data Parallelism Code Restructuring for HLS Targeting FPGAs . . . 11
  - 2.3.6 State of the Art Evaluation . . . 12
- 2.4 Summary . . . 12

3 Source-to-Source Unrolling Methodology **13**
- 3.1 Overview . . . 13
- 3.2 Source-to-Source Generation Tool Pipeline . . . 13
  - 3.2.1 dotgen: Source to Optimised Graph Conversion Tool . . . 13
  - 3.2.2 dot2c: Graph to Source Conversion Tool . . . 16
- 3.3 Testing Pipeline . . . 19
- 3.4 Intermediate Data and Control Flow Graph . . . 19
  - 3.4.1 Variable Nodes . . . 20
  - 3.4.2 Opcode Nodes . . . 22
  - 3.4.3 Special Nodes . . . 22
- 3.5 JSON Function Metadata Definition File . . . 24
- 3.6 Test Parameter Description JSON File . . . 25
- 3.7 Summary . . . 25

4 Experimental Results and Evaluation **26**
- 4.1 Evaluation Environment . . . 26
- 4.2 Benchmarks . . . 26

4.2.1 vote_kNN . . . 27
4.2.2 mac . . . 28
4.2.3 do_mean . . . 28
4.2.4 do_variance . . . 29
4.2.5 do_max . . . 30
4.3 Test Results . . . 31
4.3.1 vote_kNN . . . 31
4.3.2 mac . . . 32
4.3.3 do_mean . . . 35
4.3.4 do_variance . . . 39
4.3.5 do_max . . . 41
4.4 Summary . . . 43

5 Conclusion and Future Work **46**
5.1 Conclusion . . . 46
5.2 Future Work . . . 47
5.2.1 Issues with the Current Implementation . . . 47
5.2.2 Issues with Evaluation . . . 48
5.2.3 Other Future Improvements . . . 48

References **49**

A Conversion Process Example Sources: mac_4 **51**

List of Figures

3.1 Toolchain pipeline diagram 14
3.2 Testing pipeline diagram 19
3.3 Graphical representation of mac_4 CDFG 21

4.1 mac specialisation generation times for different data sizes 38
4.2 mac HLS times for different data sizes 38

List of Tables

2.1 List of test environments for each work 5

3.1 List of implemented opcodes 23

4.1 vote_kNN specialisation case overview 28
4.2 mac specialisation case overview 28
4.3 do_mean specialisation case overview 29
4.4 do_variance specialisation case overview 30
4.5 do_max specialisation case overview 31
4.6 vote_kNN synthesis area metrics 33
4.7 vote_kNN synthesis area metrics comparison 33
4.8 vote_kNN synthesis temporal performance metrics 34
4.9 mac synthesis area metrics 35
4.10 mac synthesis area metrics comparison 36
4.11 mac synthesis temporal performance metrics 36
4.12 mac specialised function source generation processing metrics 36
4.13 mac specialised HLS kernel synthesis and cosimulation processing time (OutMem: out of memory crash) 37
4.14 do_mean synthesis area metrics 39
4.15 do_mean synthesis area metrics comparison 40
4.16 do_mean synthesis temporal performance metrics 40
4.17 do_variance synthesis area metrics 41
4.18 do_variance synthesis area metrics comparison 42
4.19 do_variance synthesis temporal performance metrics 42
4.20 do_max synthesis area metrics 43
4.21 do_max synthesis area metrics comparison 44
4.22 do_max synthesis temporal performance metrics 44

List of Acronyms

BRAM	Block Random Access Memory
CFG	Control Flow Graph
CDFG	Control and Data Flow Graph
CGRA	Coarse-Grained Reconfigurable Array
CPU	Central Processing Unit
DFG	Data Flow Graph
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
FF	Flip-Flop
FSM	Finite State Machine
GPU	Graphics Processing Unit
HLS	High Level Synthesis
ILP	Instruction Level Parallelism
IR	Intermediate Representation
LUT	Lookup Table
NPU	Neural Processing Unit
RTL	Register Transfer Level
SDG	Sustainable Development Goals

Chapter 1

Introduction

1.1 Context and Motivation

Hardware acceleration is becoming progressively more relevant in the field of computing, as advancements in CPU execution performance have become less relevant; Paulino et al. [19] have shown that CPU clock speeds reached a plateau in 2006, even as transistor counts continue to increase, whereas FPGA technology continued to see tangible improvements as of the article's writing in 2021. This makes hardware accelerators an important part of modern computing, which may take the form of GPUs, FPGAs [15], CGRAs [18], NPUs [16], et cetera. While some of these accelerators already have widespread adoption, namely GPUs for both real time graphics and parallel compute tasks, FPGAs offer more customisability and are overall more energy and power efficient than GPUs, even if they cannot approach them in terms of raw clock speed performance, as demonstrated by Qasaimeh et al. [20].

Adoption of FPGA programming by software programmers, which is essential for the design of application-specific architectures, is mainly driven through the use of High-Level Synthesis (HLS) [8], which allows for the use of conventional computer programming languages to describe and/or generate such architectures. HLS systems are, therefore, required to analyse how to best transform the input code to take advantage of the chief benefits of FPGAs, namely the parallelisation and pipelining of operations.

1.2 Problem Definition

At the time of writing, most widespread HLS solutions rely heavily on manual optimisation of code, which requires knowledge on the part of the developer on how to design efficient logical circuits, which is beyond the field of expertise of most software developers.

Furthermore, while HLS systems are capable of optimising input code structures automatically, many of these optimisations are limited in their capabilities, namely in terms of accounting for elaborate memory dependencies.

Analysing the two current major market HLS solutions, AMD's Vitis HLS [1] largely relies on pragmas and bespoke libraries to drive FPGA specialisations in C code, whereas Intel's oneAPI compiler [14] focuses more on automatic optimisations, though it still requires the programmer to manually optimise the underlying C code and use pragmas in order to take full advantage of these.

It is with these limitations in mind that this work aims to explore and develop optimisation techniques which leverage the transformation of C and C++ code into properly optimised code that compiles into a form that results in higher efficiency and/or performance for HLS and target FPGAs.

1.3 Contributions

The work developed for this dissertation provides a simple yet novel method for optimising hardware accelerator kernels written in C for execution speed, through specialisation of a given kernel for a specific set of parameters. The toolchain developed for this purpose, along with its internal workings, are described in Chapter 3.

Furthermore, empirical analyses of this method's performance, both in terms of area and speed, as well as comparisons with other existing optimisation strategies based on total or partial specialisation, are reported in Chapter 4.

1.4 Research Questions

In order to carry out the tasks mentioned previously, it is useful to organise the core questions that the research aims to answer; these should pertain to both the identification of implementation candidates and how they perform. For this purpose, the following research questions have been identified:

- **RQ1** – Are there any optimisation opportunities relevant to area usage that have remained unexplored in the realm of HLS?
- **RQ2** – What strategies for specialisation of tasks could lead to improvements in its overall performance?
- **RQ3** – What are the best ways by which a given C code pattern be programmatically optimised for HLS or a given FPGA target?
- **RQ4** – What impact do certain code specialisation strategies have on performance/area usage/compilation time?

1.5 Methodology

This dissertation work was divided into two phases: a development phase and an experimentation phase, corresponding to the development of the optimisation method and tools, and the empirical evaluation of said method, respectively.

In the development phase, a well-defined method for converting a given function kernel into a specialised, unrolled form was devised, from which tool programs and the file structures required to accomplish this goal were written. This also includes the setup required to automate testing for the experimentation phase.

In the experimentation phase, a set of functions were chosen that would ensure thorough stress testing of real use cases in which the developed specialisation method would be employed, which would then be synthesized with varying levels of optimisation in addition to a given set of specialised versions generated by the method. Hardware area usage, latency and clock frequency metrics were then obtained through estimated values generated by the HLS tools, which were used to draw conclusions as to the effectiveness of this method relative to other optimisation strategies.

1.6 Dissertation Structure

This dissertation consists of 5 chapters in total, in addition to one appendix:

- Chapter 1 (this chapter) is the introduction for this dissertation. It contains an overview of its mission statement, including the problems it aims to solve, and what goals it tries to meet.
- Chapter 2 describes the literature review process, including how relevant papers and documents were selected for study, as well as a state-of-the-art review featuring the most relevant papers found for this work.
- Chapter 3 describes the specialisation method created by the dissertation work, featuring the toolchain developed, methods used for optimisation, and explanations of the inner workings of each tool and file format devised for this purpose.
- Chapter 4 describes the experimentation process, detailing how testing was carried out, its environment and tools, which functions were selected, optimisations and specialisations chosen, as well as test results and their analysis.
- Chapter 5 is the conclusion of this dissertation, where the total work and its findings are summarised, and future work is identified.
- Appendix A contains examples of the file formats devised for the methodology, as further reference material for understanding their structure.

Chapter 2

Literature Review

In addition to the few research papers provided in the beginning of this dissertation work, an extensive bibliography of prior research was curated for the purposes of analysing the state-of-the-art of HLS code optimisations.

For the construction of the research bibliography, an exploratory approach was used: from a prepared set of search queries, the resulting works from various sources were skimmed through by evaluating their titles and abstracts and, if deemed relevant to the study in question, added to a reference manager's database for further evaluation. From there, each selected work is given a further cursory analysis of their conclusions, methods and results, and if these prove to contain promising leads the work is admitted into the final bibliography and saved for state-of-the-art analysis.

2.1 Collection Phase

When beginning the process of literature review, the first step was to craft the query strings to use when retrieving possibly useful sources. In the end, the following two queries were settled upon (presented as a "pseudo-query" here):

```
- ("high-level synthesis" OR "high level synthesis" OR HLS)
  AND optimization

- FPGA AND compiler AND optimization
```

These queries were then adapted and submitted to various data sources, specifically SCOPUS, Web of Science, U. Porto Discovery as general databases of research material, alongside IEEE Xplore as a domain-specific data source. For SCOPUS and Web of Science, the query that specified high-level synthesis would return results outside the relevant field (namely related to bio-engineering), though this issue was easily fixed by filtering results to only include those pertinent to the field of computer science.

Table 2.1: List of test environments for each work

Name	Year	Target FPGA	Language	Compiler	HLS Tool
Huang et al. [13]	2013	Cyclone II	C	LLVM	LegUp
Haro et al. [11]	2021	Zynq ZCU102/Alveo U200	C	GCC	Vivado
She et al. [21]	2025	Zynq xc7z020clg484	C	LLVM	Vitis
Ferreira [10]	2019	Artix-7 xc7z020clg484	C	Clava	Vivado
Campos [6]	2021	Artix-7 xc7z020clg484	C	N/A	Vivado

For each query and data source, about 150 results were explored; if a title of a given result seemed relevant and its date was less than 20 years old, its abstract and publication would be consulted, and if this also points to relevant research, then the work would be stored in a library managed by the Zotero reference manager, as an "unread" work for further evaluation. In total, 72 works were gathered during this process.

2.2 Trimming Phase

After the initial collection of documents, a second curation pass was carried out by skimming through the contents contained inside each work; this process started by reading the conclusion segment, then evaluating the overall approach and obtained results of each work. The main factor being evaluated here was whether any given work's optimisation approach was applicable to the source code level, as many of these techniques concern the RTL generation step from an input IR obtained after compilation of said source code; At the result level, the criteria for acceptance was for a given work to either provide a clear improvement, or provide good insights for future work.

Should a given work pass all the aforementioned criteria, it is approved for the final study bibliography, and is marked for thorough review in the state-of-the-art section; otherwise, the work is rejected and not considered for further study. In the end, 40 research works were approved by this phase.

2.3 State of the Art

In this section, five of the most relevant works in the bibliography are summarised. All of these pertain directly to the transformation of source code for the purpose of FPGA-centric optimisation. The execution environment of each is detailed in Table 2.1.

2.3.1 The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs

This work by Huang et al. [13] aims to develop and evaluate a selection and ordering system for optimisation passes in the LLVM compiler within the domain of HLS for FPGAs, using the LegUp HLS suite and the Altera Cyclone II FPGA as targets. Four performance metrics were evaluated: maximum frequency, circuit area, execution cycle latency and wall-clock time.

The work was motivated by the uncertainty of improvements provided by default optimisation levels provided by compilation suites: prior works mentioned are Milestone, a machine learning based tool that proposes a set of passes to apply based on static analysis of source code, which achieved an average of 11% improvement in execution speed during testing, and LegUp itself, which provides basic hardware synthesis oriented optimisations such as scheduling of basic blocks into parallelised functional units.

The first step taken was to measure the impact of given compiler optimisations on HLS; the CHStone benchmark suite was used for this purpose. Each benchmark results in its own compute kernel; circuit area and maximum frequency are determined after completion of synthesis, and total execution time and wall-clock time are determined by simulating the circuit multiple times with a collection of distinct inputs.

The second round of tests concerned the impact of pass parameters relative to -O3, specifically the loop unrolling and inlining thresholds. These tests showed that maximum frequency tends to decrease and area increase as more inlining is performed due to lengthening of the critical path, while further loop unrolling improved latency at the cost of frequency. The last tests made during this phase concerned the ordering of passes relative to each other, but no meaningful pattern was found in these.

The proposed optimisation pipeline selection and ordering system can be configured to use one of the four defined performance metrics; the basic algorithm iterates over a set of given optimisation passes, chosen given on their impact both in isolation and in the -O3 optimisation pipeline, accepting passes that improve performance based on a partial simulation of the resulting kernel; this set of passes is iterated upon multiple times until an arbitrary number of iterations or if no meaningful improvements were achieved in an iteration.

Three methods for pass evaluation were devised, with each respective one trading execution time with depth of explored solution space: - Basic iteration through a sorted list of passes based on pair order evaluation; - Evaluate all possible positions of a given pass; - Evaluate all possible positions of a given pass in the top 3 best pipeline candidates.

All tests of the developed system, regardless of the evaluation method used, were always able to output a custom optimisation pipeline that outperformed the -O3 option, though the impact on circuit area in the target FPGA was minimal.

The paper proposes further work on power dissipation and energy consumption as performance metrics to consider, optimising the most computationally expensive pass evaluation method through memoisation, and finally the writing of new compiler passes geared specifically towards HLS, such as pipelining of basic blocks. Optimisation passes geared towards Very Long Instruction Word processors are also considered promising leads to investigate.

2.3.2 OmpSs@FPGA framework for high performance FPGA computing

This study by Haro et al. [11] describes and presents an improved version of the OmpSS@FPGA HLS framework, which revolves around a task-based, master-slave model in which a Symmetric Multi-Processor (SMP) host issues tasks to FPGA kernels.

Source code is written in C/C++ and compiled by Mercurium, which splits it into SMP and FPGA parts and links wrapper calls to each:

- SMP code replaces compiler directives with calls to a Nanos++ API, which manages FPGA tasks and low level communication;
- FPGA code is wrapped around a hardware runtime written for Vivado HLS, which handles the task engine and memory transfers.

Factors relevant to OmpSs@FPGA's paradigm, such as data dependencies and memory allocation, are specified through bespoke pragmas in the source code, which are then used by Mercurium to inform the data transferring required in the program.

Synthesis is handled by Accelerator Integration Tool (AIT), which feeds the high level output of the compiler to the appropriate vendor tools and assigns connections with the hardware runtime and the I/O pins of the target FPGA. In the prior incarnation of OmpSs, this hardware runtime is the Smart OmpSs Manager (SOM), which is a simple task scheduler responsible for receiving commands and forwarding them to the relevant accelerators.

Introduced in this study are multiple optimisations, one of which is the wide and unified memory port, which aims to address the deficiencies of allocating individual memory ports for each pointer argument, namely the rapid exhaustion of memory ports, by allocating a single memory port with a configurable data width, making each accelerator use a single AXI interface, but allowing for very wide data transfers, such as vector data. However, this approach requires all data to be aligned to the port's width, and data transactions must be able to occur at the same rate as the memory bandwidth to take full advantage of its benefits.

Furthermore, pipelining of load-computation-store loops can be defined, which limits their latency to solely being affected by either computation or memory transfers, instead of both. This is achieved through pragmas that detail both the arrays involved in the transfers and specifying how their addresses are incremented or decremented. In order to make this possible, the required memory and ports are doubled in order to enable double buffering, as well as special cases for the first load and the last store. Another optimisation is dynamic copy optimisation, which is able to detect and suppress local copies whenever 2 consecutive tasks operate over the same memory region, on the same accelerator by manipulating the copy flags of each task before they reach the accelerator.

Finally, the largest addition this study makes is the introduction of new internal task creation and management solutions, which attempts to incorporate as much of the task management into the FPGA part to avoid the communication latency between the SMP and the FPGA. SOM cannot handle tasks with dependencies, so another hardware runtime, the Picos OmpSs Manager (POM) was developed, which integrates the Picos Daviu hardware dependence manager, based on the Picos++ architecture, and proves itself to be faster than a software runtime. Nevertheless, it also requires more hardware resources, so it does not fully replace SOM, especially for simple or highly parallel applications, and if required SOM can now be extended with task nesting support.

POM communicates with the host through an AXI-stream interface, from which commands are received. All tasks created by accelerators pass through a scheduler before being issued, whereas SMP tasks are placed in a spawn out queue to be read by the host, which then places results in a spawn in queue. A cutoff manager dispatches tasks to either the scheduler or Picos Daviu, depending on whether dependencies are present, and handles the memory allocation for the task queue.

These optimisations were benchmarked in the Xilinx Zynq Ultrascale+ MPSoC ZCU102 and the Xilinx Alveo U200 discrete board. For benchmarking the effectiveness of these new optimisations, the following benchmarks were devised: matrix multiplication, Cholesky decomposition, n-body physics simulation and Spectra physics simulation.

In addition to individual results from these benchmarks, the average speedup of each optimisation and peak performance comparisons against SMP and GPU implementations of each benchmark were recorded. For the former measurement, both boards got good results from the wide port optimisation, with the Alveo benefitting the most due to its larger memory port; load-computation-store loop pipelining only saw visible improvements on the ZCU102, with the Alveo requiring the addition of extended SOM to benefit from it; the new internal task management runtimes, extended SOM and POM, benefitted the ZCU102 the most, though the Cholesky benchmark saw a massive speedup of 700%, compared to an average of 12.5% across the other benchmarks; the Alveo does not benefit as much from these, as it would be more suited to fine-grained tasks than those tested.

Regarding comparisons against other platforms, both FPGAs are able to outperform the SMP benchmarks with execution time improvements ranging from 36700% to 25%, and while GPU are able to outperform the FPGAs in the matrix benchmarks, the performance per watt is less for GPUs than FPGAs for all benchmarks as well.

2.3.3 A Source-to-Source Compiler to Enable Hybrid Scheduling for High-Level Synthesis

This work by She et al. [21] proposes a novel approach to hybrid scheduling by relying on generation of C code compatible with most static HLS tools. This is achieved primarily by a custom loop pipelining approach based around a Pseudo Cycle Accurate (PCA) model, alongside a custom IR dubbed the Execution Graph (EG). Through the proposed solution, the work aims to address both the deficiencies of static scheduling of operations, which is unsuitable to irregular patterns such as conditional branching and irregular memory accesses, as well as that of existing dynamic and hybrid scheduling HLS tools, which are overall less mature than static HLS tools in terms of performance.

The PCA model denotes each iteration of a loop as a "transaction", consisting of granular operation blocks dubbed Virtual Cycles (VC), which roughly correspond to the stages of a pipelined loop, though they may correspond to more than one execution cycle. Buffers are used to propagate values from VC to VC, holding the value of a given VC's output and propagating it to the input of another VC depending on the presence of a "valid" bit controlled through recurrence variables that

take the form of multiplexers, which control the execution flow of each transaction. In order to avoid race conditions where 2 execution paths join at the same operation and try to output a value to a buffer at the same time, queues are inserted into the short path, which store that path's value before the joining point is entered.

Dependencies in recurrence variables appear in the EG as Strongly Connected Components (SCC), which are handled through insertion of commit nodes that withhold valid inputs until all connected inputs are valid, at which point propagation to the output recurrence variables carries on. When multiple SCCs are found within the same loop, communication between them is handled through latency-insensitive channels, which support handshaking of valid and ready statuses; these may be either scheduled in topological order, or allow for overlapping execution of two decoupled SCCs, with the latter strategy trading better latency for a larger critical path.

The proposed framework is first compiled into LLVM IR form, with an auxiliary pass producing a Data Dependency Graph, Control Data Flow Graph, and a Control Dependence Graph, which are fed into an EG Builder to create the initial EG; from there, a Pre-Scheduler identifies boundaries for VCs, a PCA transformer applies the PCA model to the EG, an SCC Connector inserts inter-SCC communication nodes, and finally a PCA translator converts the final EG into C code. A special case is made for regular loops with no conditionals which causes them to not have the PCA model applied to them, since they do not benefit from it.

The Execution Graph itself is a modification of Gates SSA, which itself is a modification of LLVM IR, introducing alternate node types meant for exploitation of parallelism. EG extends it by adding new nodes corresponding to the previously mentioned elements that are core to the PCA model (buffers, SCC channels, etc.). When determining FIFO sizes for SCC communication channels, the framework will try to determine its maximum accumulated token number by profiling the loop within a testbench, though if one is not available it will try to estimate it instead by comparing the mean latencies of both connected SCCs. Finally, irregular memory accesses are handled through a store queue.

Due to its implementation of dynamic scheduling, many of its added factors end up adding execution overhead to the circuit, with the main factors being the control logic for buffers, the hardware FSM controlling execution flow, and the handshaking logic of inter-SCC communication.

The proposed framework was benchmarked against a set of different HLS tools, each representing a different scheduling approach: Vitis HLS [2] for static scheduling (also used for compiling the output of the proposed framework), Dynamatic HLS [17] for dynamic scheduling, and DASS HLS [7] for hybrid scheduling. A set of open source benchmarks, covering a wide range of behaviour predictability, were compiled and tested for each tool on a Zynq FPGA board, both with and without optimisation applied in compilation. For each run, the maximum frequency, execution cycles, wall-clock time and circuit area were recorded.

Compared to static scheduling, the new framework was able to successfully identify regular loops and achieve the same performance, and conditional loops were able to show clear improvements in wall-clock time; kernels with dynamic memory accesses showed a higher frequency but

lower latency. Compared to both dynamic and DASS approaches, the new framework achieved higher maximum frequencies, achieved by use of static HLS tools, less handshaking overhead and better optimisation of static components. Furthermore, it also consistently showed less resource usage.

The work also points out that the current implementation of this framework only operated over a single loop nesting level, and places optimisation of multiple loop nesting levels as further work.

2.3.4 Graph-Based Code Restructuring Targeting HLS for FPGAs

This work by Ferreira [10] covers an approach to automatically apply optimisations to software HLS code through the use of a dataflow graph representing the code's execution trace. This aims to address the inability of existing HLS tools to automatically apply most optimisation strategies to code targeted for hardware synthesis.

The approach to optimisation devised by the work is to transform execution traces of C code into DFGs through a frontend component, which are then optimised and transformed back into HLS optimised C code by a backend component. Dataflow graphs were chosen for their ability to represent not only the flow of data but also data parallelism, which are important considerations for hardware implementations.

The frontend generates graphs through instrumentation code injection into the input C code, which is then compiled and executed to create this file at runtime. The backend optimises its input graphs through 7 passes, configured through a file that specifies optimisation options and simultaneous memory operations supported by the target hardware. These pass stages are the following:

1. Pruning of unnecessary nodes;
2. Separation of dataflows for each output generated;
3. Matching between aforementioned separate dataflows for the purpose of folding repeated code patterns;
4. Sequential matching of dataflows through pipelining of the graph around the variable with the most writes, so as to find a better structure for the overall algorithm;
5. General dataflow optimisations, such as substitution of arithmetic accumulation chains with balanced trees, substitution of divisions with inverse multiplications, generation of buffers to reduce redundant memory accesses, and optional full partitioning of arrays;
6. Unfolding of loops generated in stages 3 and 4 to exploit instruction level parallelism, going back to stage 5 for each unfolded loop for further optimisation;
7. Generation of output C code with the relevant HLS directives.

Evaluation of this method against several degrees of optimisation determined that elimination of memory bottlenecks was the most important optimisation for the sake of increasing throughput, and that folding and unfolding alone already showed improvements for some test cases, with the highest speedup cases scoring between 1.4x and 81.7x depending on the target function, achieved when either all the backend optimisation steps are applied, or all but data reuse is applied due to area or critical path restraints.

The work's limitations include the inability to generate DFGs properly for conditional statements or loops that branch into distinct dataflow paths depending on their inputs, due to the aforementioned runtime generation method with instrumentation code, which is proposed to be solved through merging of execution traces and representation of ternary operators. Its full unfolding approach is also highlighted for its lack of scalability, resulting in excessively large DFGs that could be condensed in future work.

2.3.5 On Data Parallelism Code Restructuring for HLS Targeting FPGAs

This work by Campos [6] continues the work described in Section 2.3.4 by adding code restructuring for data level parallelism, introducing support for simple control flow representation and multiple outputs.

The overall approach remains the same as in the previous work: a frontend turns input C code into a DFG, and a backend restructures said DFG and turns it into optimised C code. The frontend injects instrumentation code through Clava [5]. Optimisation aims to take advantage of both data and task level parallelism, with the following tasks being described for the process:

1. Pruning of unnecessary edges and nodes;
2. Assignment of a "height" value to each node;
3. Detection and reordering of addition chains into balanced trees with minimal height to increase task level parallelism and expose potential data level parallelism;
4. Location, ordering and folding of isomorphic subgraphs through matching of all possible subgraphs by weight and operating over those with 3 or more occurrences;
5. Encapsulation of remaining data flow nodes into prologue and epilogue functions.

Evaluation of this improved method shows significant improvements in speedup when comparing the same test functions with that of the previous approach, achieving about 10x the maximum speedup over that work. Execution times for the backend also were evaluated against different values of maximum and minimum folding levels, resulting in improvements between 19% and 87% for optimal settings.

Limitations of this approach maintain the inability to properly handle conditional code (e.g. `if` and `switch` statements), as well as not being able to account for kernel parameters that alter execution flow.

2.3.6 State of the Art Evaluation

Throughout the process of reviewing the state-of-the-art of HLS optimisations, three main target fields were identified:

- Task Scheduling
- Loop Transformations (Parallelisation, Pipelining, etc.)
- Memory Optimisations (Array Transformations, etc.)

Of these, loop transformations seem to be the most well-trodden subject, with many different methods of optimising loops according to specific use cases. Meanwhile, task scheduling seems to be the most promising field, with very recent strides in the exploration of novel scheduling methods, especially in the realm on dynamic scheduling [21][11]. Finally, memory optimisations have seen the least research, and many of the advancements in that field have been relatively simple. This makes this field rather open to new discoveries, but the lack of radical discoveries in recent years makes it harder to identify possible angles by which to tackle.

2.4 Summary

This chapter covers the literature review process, describing each step taken during preparation, curation and study of each paper. Five of the most relevant works are also summarised here.

The core of the studies found during this process relate to loops, which have multiple ways of being optimised (e.g. unrolling, parallelisation), and with clear opportunities for specialisation by making their number of iterations fixed. This line of optimisation is what was ultimately chosen as the core topic explored within this dissertation work.

Chapter 3

Source-to-Source Unrolling Methodology

3.1 Overview

The approach described in this dissertation is to specialise certain kernels for specific scenarios through complete unrolling of their loops. This approach should result in a faster kernel, at the cost of occupying more area within the hardware accelerator.

The work presented in this chapter covers the automatisation of the process of converting a given function into a specialised function that turns a subset of its parameters into constant values, in addition to completely unrolling all loops within, including nested loops.

3.2 Source-to-Source Generation Tool Pipeline

The source transformation pipeline, shown in Figure 3.1, takes a C source file containing the kernel function as input, producing a specialised, loop-free C source file corresponding to a kernel that matches the original's behaviour, for constant values of a subset of its parameters.

The input source is first integrated into the source of a C utility named `dotgen`, which outputs an Intermediate Representation (IR) of the target kernel's Data and Control Flow Graph (CDFG) in a human-readable graph described in Graphviz's DOT language [4], with unrolling optimisations ; this, alongside a JSON file containing metadata describing all remaining information not present in the graph, are fed into a Python tool named `dot2c`, which parses both files and builds the internal graph to generate the final C source output, containing the specialised function.

The following sections aim to explain how each of the pipeline's tools, `dotgen` (3.2.1) and `dot2c` (3.2.2), operate internally.

3.2.1 dotgen: Source to Optimised Graph Conversion Tool

The `dotgen` tool automatically generates a DOT language file representing a given function's data and control flow graph by running an instrumentised version of said function's source. The

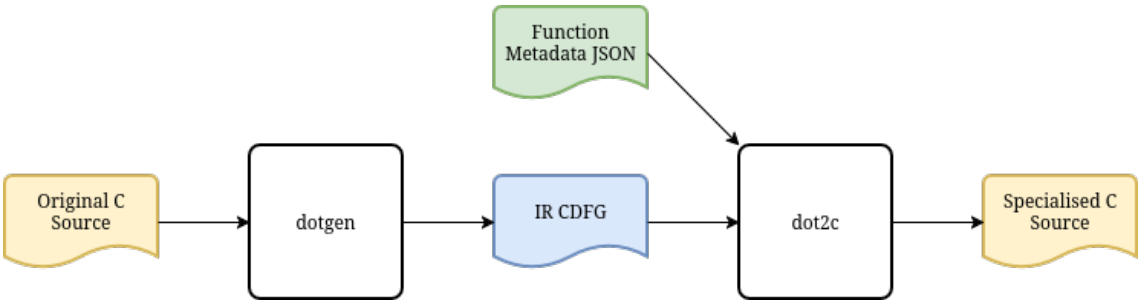


Figure 3.1: Toolchain pipeline diagram

generated graph must match the control flow of the original function, but with all loops completely unrolled, and any variable values that affect loop count converted into constant values.

Instrumentation consists of insertion of standard library functions for string generation and printing text to be output into the graph file, alongside a collection of helper functions and macros that make the instrumentation itself less verbose and repetitious on the source side. An example of the `mac` function after instrumentation is presented in Listing A.2. Listing 3.1 shows a code snippet corresponding to the loop within the `mac` function's C source after inclusion of the instrumentation code; the complete source of the instrumentised function can be seen in Listing A.2.

These instrumentation functions take care of creating or accessing variable or opcode nodes, as well as creating edges between nodes. In addition, macros for creating opcode nodes were defined for each mnemonic, as the information required by its relevant functions (mnemonic, inputs, outputs) to define each of them is verbose.

It should be noted that this tool currently only automates the process of graph generation through execution of the instrumentised function; the process of instrumentation itself, as well as setting up the tool's source itself to run the function with the correct parameters, is all handled manually. The tool's approach also cannot account for nested function calls, therefore these must be replaced by inlining the nested function's body and modifying its instrumentation to link into its position within the parent function.

Organisation of the instrumentation code follows a set pattern: Prepended to the start of the function body is `DOT_start`, which begins the graph definition and initialises both `start` and `end`, along with definitions of an array of string buffers `_buf` and string pointer `_ptr`; following that, each basic block is identified (a basic block in this case is any straight sequence of code with no branches caused by loop iteration flow except to its beginning or after its end) and instrumentation code is appended to each, executing the following operations, in order:

- Construction of dynamic node names through `sprintf` calls that output into `_buf` entries (e.g. variables or values that change with loop iterations);
- Creation of variable nodes through `DOT_addvar` calls, with their references stored in `_ptr`;
- Creation of opcode nodes through opcode macros, with their references stored in `_ptr`;

- Creation of control flow edges through `DOT_vtoo` (variable to opcode), `DOT_otov` (opcode to variable) and/or `DOT_otoo` (opcode to opcode) calls;
- Creation of data dependency edges through `DOT_vtov` (variable to variable) calls;
- Caching of variable node references into `_ptr` for next block's data dependency definitions.

This continues until the function reaches its termination point, to which the edges going to the `end` node are created and `DOT_end` is called, closing the graph definition.

Listing 3.1: Code snippet of instrumentised loop for mac C function source

```
1    for (int i = 0; i < n; i++)
2    {
3        // Temporary string construction
4        sprintf(_buf[0], "%d.c.f32", i);
5        sprintf(_buf[1], "b1_%d_0", i);
6        sprintf(_buf[2], "b1_%d_1", i);
7        sprintf(_buf[3], "b1_%d_2", i);
8        sprintf(_buf[4], "b1_%d_3", i);
9        sprintf(_buf[5], "b1_%d_4", i);
10
11       // Variable nodes
12       _ptr[0] = DOT_addvar("a.ptr.f32");
13       _ptr[1] = DOT_addvar(_buf[0]);
14       _ptr[2] = DOT_addvar("b.ptr.f32");
15       _ptr[7] = DOT_addvar("acc.f32");
16       _ptr[9] = DOT_addvar("acc.f32");
17
18       // Opcode nodes
19       _ptr[3] = OP_LDARRAY_PTR_F32(_buf[1]);
20       _ptr[4] = OP_LDARRAY_PTR_F32(_buf[2]);
21       _ptr[5] = OP_MUL_F32(_buf[3]);
22       _ptr[8] = OP_ADD_F32(_buf[4]);
23
24       // Control flow edges
25       DOT_vtoo(_ptr[0], _ptr[3], 0);
26       DOT_vtoo(_ptr[1], _ptr[3], 1);
27       DOT_vtoo(_ptr[2], _ptr[4], 0);
28       DOT_vtoo(_ptr[1], _ptr[4], 1);
29       DOT_otoo(_ptr[3], _ptr[5], 0, 0);
30       DOT_otoo(_ptr[4], _ptr[5], 0, 1);
31       DOT_otoo(_ptr[5], _ptr[8], 0, 0);
32       DOT_vtoo(_ptr[7], _ptr[8], 1);
33       DOT_otov(_ptr[8], _ptr[9], 0);
34
35       // Data dependency edges
36       DOT_vtov("start", _ptr[0]);
```

```
37          DOT_vtov("start", _ptr[1]);
38          DOT_vtov("start", _ptr[2]);
39          DOT_vtov(_ptr[15], _ptr[7]);
40
41          // Node caching for next block's data dependencies
42          _ptr[15] = _ptr[9];
43          acc += a[i] * b[i];
44      }
```

3.2.2 dot2c: Graph to Source Conversion Tool

The `dot2c` tool takes the DOT graph generated with `dotgen` (previously described in Subsection 3.2.1), as well as a JSON metadata file, and outputs a C source file corresponding to the information given.

Internally, both the DOT and JSON files are run through a parser that converts their information into text data, which is then aggregated and formatted to form the C source; the DOT parser builds the CDFG, from which the tool generates the function's sequence of instructions, while the JSON parser turns the metadata into the function's definition and local variable declarations.

The DOT parser makes use of the NetworkX library [9] to transform the plain text input file into an internal format that may be manipulated by the program. The main traversal process is given in pseudocode in Listing 3.2, starting with the `parse` function: In lines 2-4, the graph itself is initialised to accommodate graph colouring by giving all nodes a "visited" boolean parameter. Lines 6-8 find the first nodes to parse by locating all children of the `start` node and adds them to a queue. Lines 10-18 represent the graph traversal loop proper: the first node in the queue is retrieved, and if the node has not been coloured yet then it is parsed through the `parse_node` function, which takes care of redirecting each node type to their relevant processing function in lines 30-33, if its parent nodes were all already coloured: node processing relies on information generated by processing of its parent nodes, and this condition also ensures that data dependencies are respected. Should processing be successful, the node is coloured and its children are added to the queue. If the `end` node wasn't reached, then the graph is considered invalid and an error is raised in line 21. Otherwise, the end result of this function consists of a list of C statements that represent the function's body, excluding variable definitions.

Processing of variable nodes (described in Section 3.4.1) is described in Listing 3.3: the node's name and C type are extracted from the node's text label in lines 3-4. Lines 7-10 check for whether this node is the output of an opcode node, in which case a new C statement is generated where the variable is assigned the result of the parent's operation, optionally cast to the variable's type, which is added to the statement list. Finally, line 13 colours the node, thus finishing processing.

Processing of opcode nodes (described in Section 3.4.2) is described in Listing 3.4: lines 3-5 retrieve the node's input nodes (which exist as its parent nodes), the mnemonic that defines it, and its output nodes; the input nodes' names and mnemonic are used to generate the opcode's equivalent C operation and store it as the node's name in line 8. Lines 9-14 determine whether the

opcode has outputs at all, which determines the rest of the processing step: if an output exists, then the current node's type is derived from the output node's type (since the output node has not been processed yet) and stored for said output node's later usage; if no output exists, then the generated operation is treated as its own final C statement, which is added to the statement list.

The JSON parser extracts the function's name, return type and parameters' names and types, converting the type information to their C type equivalents, and stores them in a dedicated structure; afterwards, it reads a list of the variables referenced by the code, including their scope, type and, if applicable, array size, and generates a corresponding list of C declaration statements.

Finally, the generated data from both the DOT and JSON parsers is aggregated into a complete C source function and its corresponding header file, both of which are output into their respective output files.

Listing 3.2: Node traversal algorithm pseudocode

```
def parse(graph):
        # Initialise node colouring
        for node in graph.nodes:
                node.visited = False

        # Begin traversal at start node
        graph.nodes['start'].visited = True
        queue = list(graph.children('start'))

        # Traverse graph
        while(len(queue) > 0):
                # Get next node in queue
                node = queue.pop(0)
                # Stop traversing if endpoint is found
                if node = 'end':
                        return
                if node.visited == False:
                        queue += parse_node(graph, node)

        # Did not find endpoint, therefore graph is malformed
        raise DOTError('Unexpected end of traversal')

def parse_node(graph, node):
        # Ensure all parents have been parsed before
        for parent in graph.parents(node):
                if parent.visited == False:
                        return []

        # Determine node type and parse it accordingly
        if node is variable:
                parse_variable(graph, node)
        if node is opcode:
```

```
33                  parse_opcode(graph, node)
34
35          # If node could be parsed, colour it and send its children to the queue
36          if node.visited == True:
37                  return graph.children(node)
38          else:
39                  return []
```

Listing 3.3: Variable node parsing pseudocode

```
1   def parse_variable(graph, node):
2           # Read node label and extract its C name and type
3           node.name = get_var_name(node)
4           node.ctype = get_var_ctype(node)
5
6           # If node is the output of an opcode, generate C assignment statement
7           for parent in graph.parents(node):
8                   if parent is opcode:
9                           src = conv_type(parent, node.ctype)
10                          add_c_stmt(f'{node.ctype} {node.name} = {src};')
11
12          # Colour node
13          node.visited = True
```

Listing 3.4: Opcode node parsing pseudocode

```
1   def parse_opcode(graph, node):
2           # Get input and output C names and types, as well as mnemonic from node's
                text label
3           inputs = get_opcode_inputs(graph, node)
4           mnemonic = get_opcode_mnemonic(node)
5           output = get_opcode_output(graph, node)
6
7           # Transform opcode into a C operation
8           node.name = get_c_operation(mnemonic, inputs)
9           if output != null:
10                  # If opcode has an output, get its C type
11                  node.ctype = output.ctype
12          else:
13                  # If opcode has no output, add its C operation as a statement
                        directly
14                  add_c_stmt(f'{node.name};')
15
16          # Colour node
17          node.visited = True
```

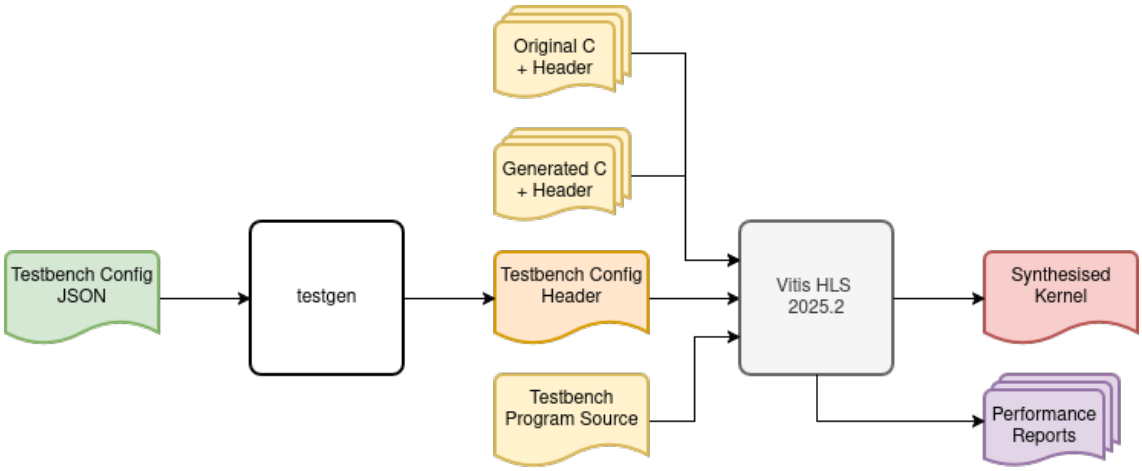


Figure 3.2: Testing pipeline diagram

3.3 Testing Pipeline

Functions are tested through integration within a simple, general purpose testbench program, which calls both the original and transformed functions, then compares their returned values. Its flow is depicted in Figure 3.2.

This integration process consists of putting both their source and header files within the test bench project's source structure (the original function files go into a `ref` directory, whereas the transformed function files go inside a `tgt` directory), then generating a `test_params.h` header file through a Python utility named `testgen`, which includes definitions and macros that modify the behaviour of the testbench program to work properly with their given functions. This tool is configured through a JSON file that is passed to it, which is described in more detail in Section 3.6.

After integration is carried out, the source files for both functions need to be manually configured within Vitis by adding them into the project, defining which function to be used as the top function for kernel synthesis, and adding pragmas with port definitions for the purpose of C/RTL cosimulation.

Finally, the testbench is run locally to ensure operation correctness, then synthesis is performed for both functions to obtain area usage metrics and the achieved clock period, then finally C/RTL cosimulation is run to obtain the latency of the kernel.

3.4 Intermediate Data and Control Flow Graph

In order to facilitate the prototyping of the toolchain, the intermediate graph is represented through Graphviz's DOT language [4], since it is a human-readable graph representation language with robust visualisation tools, which makes the process of manually iterating over the design of the graph's structure, as well as being able to process said graph into a visual format to aid in visualising its flow for evaluation.

To better exemplify this graph's structure, an example function `mac` (Listing 3.6) and the graphical representation of an IR CDFG generated from it (Figure 3.3) are provided within this section; the actual DOT language description of the graph can be found in Listing A.3.

The intermediate DCFG is a directed graph which consists of 3 types of nodes: variable nodes, opcode nodes and special nodes.

The text field of a given node is defined in the abstract grammar present in Listing 3.5, with variable nodes following the `variable` definition, opcode nodes following the `opcode` definition, and special nodes following the other definitions of `label`.

Listing 3.5: Abstract grammar for node text fields

```
label           : 'start'
                | 'end'
                | variable
                | opcode
variable        : var_name ['.c' | ('.local_' block_id)] [type] '.' INT
var_name        : NAME ['[' INT ']'] [('.' var_name) | ('->' var_name)]
type            : ['.ptr'] '.' ('i' | 'f') INT
block_id        : INT ['_' block_id]
opcode          : '{' (opcode_input | ('{' opcode_inputs '}')) '|' opcode_mnemonic
    '|' (opcode_output | ('{' opcode_outputs '}')) '}'
opcode_mnemonic : 'mov' ['.cond'] type
                | 'add' ['.cond'] type
                | 'sub' ['.cond'] type
                | 'mul' ['.cond'] type
                | 'div' ['.cond'] type
                | 'ldarray' ['.cond'] type
                | 'conv' ['.cond'] type type
                | 'cmp_' ('gt' | 'lt' | 'eq' | 'ne' | 'ge' | 'le') type
                | 'ret' type
opcode_input    : 'i' INT type
opcode_inputs   : opcode_input '|' (opcode_inputs | opcode_input)
opcode_output   : 'o' INT type
opcode_outputs  : opcode_output '|' (opcode_outputs | opcode_output)

NAME            : <any valid C symbol name>
INT             : <0 or any positive integer>
```

3.4.1 Variable Nodes

Variable nodes are used to represent not only variables, but also constants. They are connected to opcode nodes to identify them as either inputs or outputs for that opcode, and are connected to other variable nodes to indicate a data dependency in the execution flow.

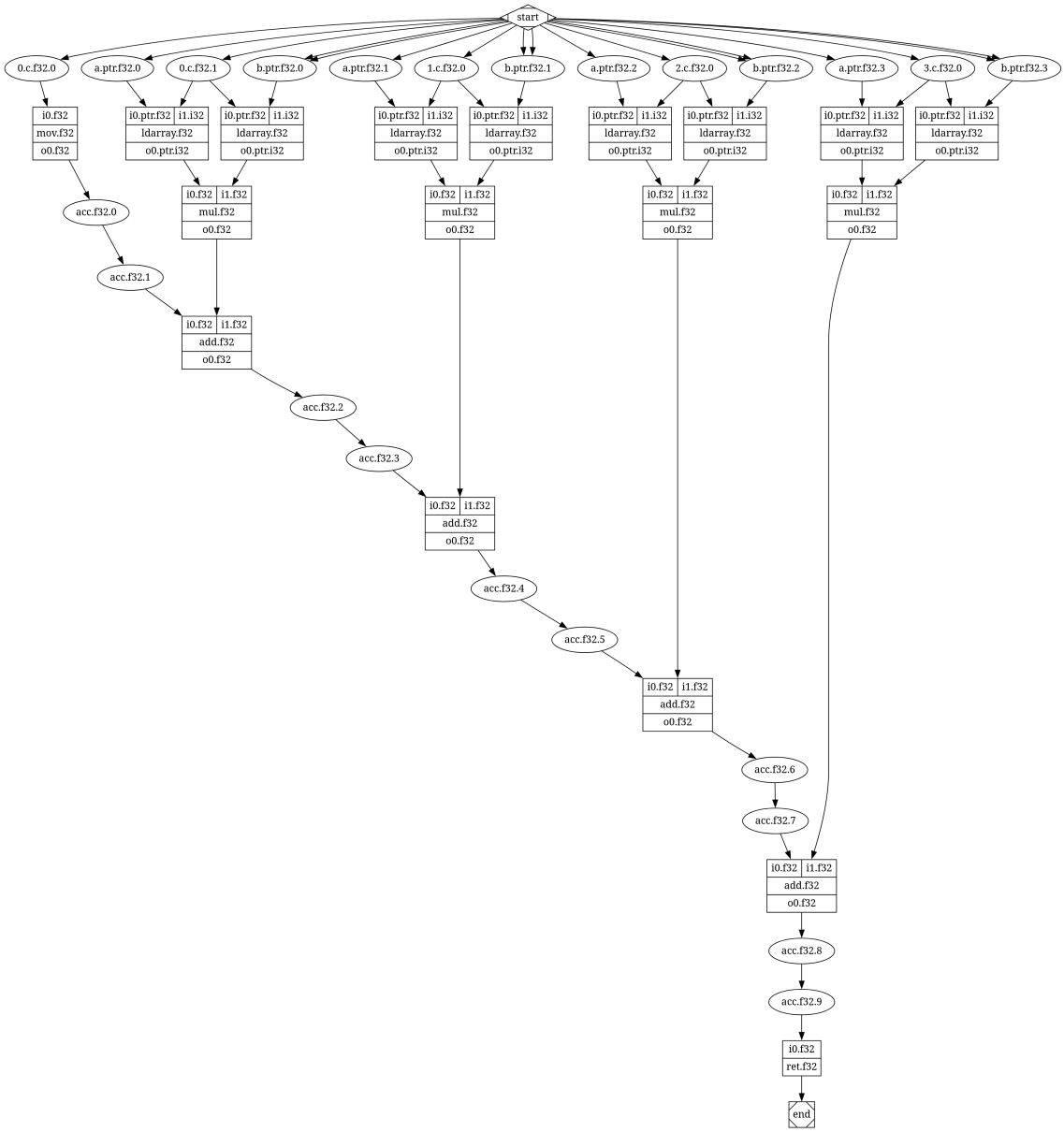


Figure 3.3: Graphical representation of mac_4 CDFG

Within the DOT language, these nodes are depicted as plain nodes, with their text field containing a descriptor, optional flags denoting either a constant value (`c`) or a block scope variable (`local_<NUM>`), its type, and an index for distinguishing between multiple instances of the same variable.

3.4.2 Opcode Nodes

Opcode nodes represent a basic operation, which includes a mnemonic identifying the kind of operation and fields for inputs and outputs, which are then linked into either variable nodes, or the inputs/outputs of other opcode nodes when applicable to not overcomplicate the graph with superfluous temporary variable nodes.

To distinguish opcode nodes, their internal name is given according to an identification pattern denoted as `(block_no)_(opcode_id)`, where `block_no` refers to an abstract basic code block number to which this opcode corresponds, as well as its iteration number(s) if applicable; a basic block, in this context, refers to any sequence of instructions that is always executed linearly. Both this basic code block number and the iteration numbers are separated by underscores (_). The opcode ID is simply a sequence number which corresponds to this opcode node's position within the basic code block's execution order. All numbers referenced in this description are zero-indexed.

This node's DOT language representation is a record node, divided into 3 rows containing, from top to bottom, the inputs, the opcode name, then the outputs. Each node's internal name corresponds to its code block and iteration number. The name text field's parameters consist of its mnemonic, followed by its type, with both parts being able to consist of multiple parameters if necessary (for example, a type casting operation from `int` to `float` has the name field `conv.i32.f32`). Input and output nodes follow a more rigid pattern: the first parameter is a single character, either `i` for an input or `o` for an output, followed by a sequence number; after this follow one or more parameters describing its type, in similar fashion to variable nodes.

For this project, only a subset of opcodes was implemented that were relevant to the functions that this work used. These cover basic operations such as data transfers (`mov`), arithmetic (`add`, `sub`, `mul`, `div`), comparisons (`cmp`), type casting (`conv`), array access (`ldarray`) and returning values (`ret`). A short description of their structure and function is listed in Table 3.1.

A notable limitation of the current opcode implementation is that conditional branches are not fully implemented, instead only appearing as a family of opcodes with a `cond` name suffix, which only update their output value if an added input boolean value is true. This leads to certain features, namely multiple return points, to be impossible to represent with the current graph definition; it also could lead to wasted time for large conditional blocks.

3.4.3 Special Nodes

Special nodes are nodes that do not have a direct analogue in the source code. 2 designations of special nodes exist:

Table 3.1: List of implemented opcodes

Mnemonic	Inputs	Output	Pseudocode
mov.type	src.type	dst.type	dst = src
ldarray.type	a.ptr.type, idx.i32	e.ptr.type	e = &(a[idx])
conv.type1.type2	src.type1	dst.type2	dst = (type)src
mov.cond.type	s.type, c.i1	d.type	d = c ? s : d
add.type	op1.type, op2.type	out.type	out = op1 + op2
sub.type	op1.type, op2.type	out.type	out = op1 - op2
mul.type	op1.type, op2.type	out.type	out = op1 * op2
div.type	op1.type, op2.type	out.type	out = op1 / op2
cmp_eq.type	op1.type, op2.type	out.i1	out = op1 == op2
cmp_ne.type	op1.type, op2.type	out.i1	out = op1 != op2
cmp_gt.type	op1.type, op2.type	out.i1	out = op1 > op2
cmp_ge.type	op1.type, op2.type	out.i1	out = op1 >= op2
cmp_lt.type	op1.type, op2.type	out.i1	out = op1 < op2
cmp_le.type	op1.type, op2.type	out.i1	out = op1 <= op2
ret.type	v.type	N/A	return v

- Start node: a unique node that represents the starting point of the program's control flow. Every node with no prior dependencies is pointed to by this node. Represented in the DOT language as a diamond shaped node labelled `start`.
- End node: represents the end of the control flow for the function program. Upon finding every connection to the end node, the parser stops parsing the graph. Represented in the DOT language as a square shaped node labelled `end`.

Listing 3.6: mac function C source code

```
1 float mac(float *a, float *b, int n)
2 {
3     float acc = 0;
4     for (int i = 0; i < n; i++)
5     {
6         acc += a[i] * b[i];
7     }
8     return acc;
9 }
```

Listing 3.7: mac_4 function C source code

```
1 float mac_4(float * a, float * b)
2 {
```

```
3   float acc;
4
5   acc = 0;
6   acc = (float)(*(&(a[(int)(0)]))) * (float)(*(&(b[(int)(0)]))) + acc;
7   acc = (float)(*(&(a[(int)(1)]))) * (float)(*(&(b[(int)(1)]))) + acc;
8   acc = (float)(*(&(a[(int)(2)]))) * (float)(*(&(b[(int)(2)]))) + acc;
9   acc = (float)(*(&(a[(int)(3)]))) * (float)(*(&(b[(int)(3)]))) + acc;
10  return acc;
11 }
```

3.5 JSON Function Metadata Definition File

The JSON file passed to the `dot2c` tool contains all function information not present within the CDFG DOT file. Since the CDFG only covers the data and control flow of the function, this JSON file contains data pertaining to the function's definition and the variables used within. An example of such a file can be found in Listing A.4.

The object described by the metadata file contains two strings, `name` and `return_type`, followed by an array of objects `variables`. `name` contains the name of the function, and `return_type` contains the return type of the function, written in the format described in Listing 3.5 within the `type` token.

This file is manually written in the work's current incarnation, though a tool could be made to generate it automatically with a similar method to the one used by `dotgen`.

The objects contained within the `variables` array have the following parameters:

- `name`: string containing the variable's name.
- `scope`: string describing the scope of the variable. May either be "`fn_parameter`" for function parameters, "`local`" for local variables, or "`local_loop`" for variables enclosed within a loop.
- `type`: string containing the variable's value type, in the same aforementioned format as `return_type`.
- `array_size`: optional integer array that specifies that the variable is an array, and thus can only be present if said variable is defined as a pointer in `type`. Each element of the array denotes the size of each dimension (e.g. the first integer specifies the size of the first dimension, the second integer specifies the size of the second dimension, et cetera).
- `block_id`: integer value that specifies that the variable is present in the corresponding code block. Only present if `scope` is equal to "`local_loop`".
- `iterations`: array of integer values that specify how many iterations this variable goes through; this tells `dot2c` how many separate variables it should generate in place of the current variable. Only present if `scope` is equal to "`local_loop`".

3.6 Test Parameter Description JSON File

The JSON file passed to `testgen` contains all information required to configure the testbench program for a given set of reference and target functions; this file is separate from the one passed to `dot2c`. It contains an array of objects `inputs` describing what input variables are required to be defined for both function, two objects `reference` and `target` describing how their respective function should be called, and an `output_type` string containing the C type of both functions' output. An example of this file can be found in Listing A.4.

Each object in the `inputs` array contains the following parameters:

- `name`: String containing the name of the variable.
- `type`: String containing the C type of the variable.
- `array_size`: Optional string containing the numeric size of the variable. This tells `testgen` that this variable is an array of the type given in `type`.
- `definition`: A string (or an array of strings if the variable is an array) containing the value declaration of the variable.

Both `reference` and `target` follow the same structure, thus containing the following parameters:

- `name`: String containing the name of the function.
- `file`: String containing the file path of the function's header file.
- `args`: Array of strings containing the parameters to pass to the function for testing.

3.7 Summary

This chapter has described a new method by which a given C source function may be transformed into a specialised and optimised version through a semi-automated pipeline, as well as the evaluation toolchain developed for ensuring the correct behaviour of said method's output. All file formats created for this process, such as a control and data flow graph for intermediate representation and configuration metadata for the tools themselves, were also described in detail.

The generation toolchain manages to implement full unrolling of the target function as its chief optimisation, thus allowing it to theoretically achieve maximum cycle efficiency when synthesised through high level synthesis, the impact of which is explored in detail in the next chapter.

Chapter 4

Experimental Results and Evaluation

4.1 Evaluation Environment

Evaluation was carried out using the Vitis 2025.02 [3] high-level synthesis development environment, with a clock period target of 5ns (clock frequency of 200 MHz). The FPGA model used is a Zynq-7000 XC7Z020-CLG484 [23], corresponding to the model used in the ZedBoard development board [22].

The computer used to run all tools required by the evaluation process is an MSI Katana 15 B13V laptop, containing an Intel i7-13620H and 32GiB of RAM, running Fedora Linux 44; the process ran at maximum performance settings for all steps.

4.2 Benchmarks

For the purposes of evaluation, a selection of small, simple function kernels were chosen with regard to characteristics that cover distinct cases. In the end, 5 functions were chosen, each tested against multiple specialisation cases. 4 of these functions were adapted from the JACABench benchmark suite [12], with the `mac` function being the sole function created during the dissertation work. `do_mean`, `do_variance` and `do_max`, in specific, are part of the same feature classification group of functions, and thus are similar in structure.

In addition to the original general purpose function and its tool-generated specialised counterparts, the specialisation of the original function by simply swapping all occurrences of the constant parameters with constant values within the C source was also tested, as well as automatic unrolling of said case by using the `HLS unroll` pragma. Since these optimisations are simple to implement, each was written manually during testing. It should be noted that limitation of loop iterations through Vitis's `HLS loop_tripcount` pragma was also generated and tested for most functions, though it only resulted in negligible or no changes in latency in relation to the original kernel.

The following subsections describe each function, detailing how it operates, the reasons for its inclusion in testing, and what specialised kernels were generated for that purpose. These are coupled with their respective C source listing and a table of all the generated kernels, which specifies the parameter values they specialise towards (Constants), The number of lines of code generated for the C source (LoC), and the number of variable nodes (#VarNodes), opcode nodes (#OpNodes) and edges (#Edges) contained within the corresponding intermediate representation CDFG.

4.2.1 vote_kNN

`vote_kNN` (shown in Listing 4.1) is an implementation of the voting part of the k-Nearest Neighbour algorith. It consists of a classification algorithm which takes an array of floating point numbers which range from 0 to the given number of classes, then outputs the predominant class value it came across. This function was chosen due to its high reliance on loops (thus a good candidate for unrolling optimisations) and its coverage of core basic operations, containing both arithmetic and conditional operations.

Listing 4.1: vote_kNN function C source code

```
1  float vote_kNN(float *ids_shortest_dist, int k, int num_classes)
2  {
3      float histogram[num_classes];
4      for (int i = 0; i < num_classes; i++) {
5          histogram[i] = 0;
6      }
7
8      // build histogram
9      for (int i = 0; i < k; i++) {
10         float p = ids_shortest_dist[i];
11         histogram[(int)p] += 1;
12     }
13
14     float max = 0; // maximum is K
15     float classification_id = 0;
16     for (int i = 0; i < num_classes; i++) {
17         if (histogram[i] > max) {
18             max = histogram[i];
19             classification_id = (float)i;
20         }
21     }
22
23     return classification_id;
24 }
```

For the tests themselves, cases were generated and tested for all combinations of this function for cases where k=3 or 5, as well as with 6 or 22 classes.

Table 4.1: vote_kNN specialisation case overview

Name	Constants	LoC	#VarNodes	#OpNodes	#Nodes	#Edges
vote_kNN_3_n6	k=3, n=6	55	70	39	111	139
vote_kNN_5_n6	k=5, n=6	65	82	47	131	163
vote_kNN_3_n22	k=3, n=22	135	198	103	303	395
vote_kNN_5_n22	k=5, n=22	145	210	111	323	419

4.2.2 mac

`mac` (shown in Listing 3.6) is a simple multiply-and-accumulate function that receives two floating point number arrays `a` and `b` of the same size `n`, then returns the result of following formula:

$$\sum_{i=0}^{n-1} a_i b_i$$

It works by iterating through each pair of array values, multiplying them and adding them to an accumulator variable for each iteration, then returning said accumulator in the end. This function was chosen due to it being a tight loop with a multiplication operation inside, which is known to be expensive to implement in FPGAs. This should provide a case where resource usage can increase massively with our specialisation method.

For evaluation, specialised cases for array sizes of 4, 16 and 64 were generated. In addition, this function was chosen for evaluation of generation and synthesis performance, for array sizes of 4, 8, 16, 32, 64, 128, 256, 512 and 1024.

4.2.3 do_mean

`do_mean` calculates the mean value of a list of values from an array of floating point feature values. Given the feature window's size, it accumulates the values of each element in the feature window (different types of features are interleaved by a value defined at compile time), then divides the accumulated value by the window's size.

This acts as a less demanding counterpart to `mac`, as the single loop present only features addition, with a single division operation appearing only after said loop is completed.

For evaluation, window sizes of 200, 300 and 400 were chosen, as is the case with the rest of the feature classification functions.

Table 4.2: mac specialisation case overview

Name	Constants	LoC	#VarNodes	#OpNodes	#Nodes	#Edges
mac_4	n=4	11	23	18	43	62
mac_16	n=16	23	83	66	151	230
mac_64	n=64	71	323	258	583	902

Table 4.3: do_mean specialisation case overview

Name	Constants	LoC	#VarNodes	#OpNodes	#Nodes	#Edges
do_mean_200	wsize=200	207	807	402	1211	1609
do_mean_300	wsize=300	307	1207	602	1811	2409
do_mean_400	wsize=400	407	1607	802	2411	3209

Listing 4.2: do_mean function C source code

```
1 void do_mean(float *mean, float *window, int wsize)
2 {
3   float sum = (float) 0.0;
4   for(int i=0; i < wsize; i++)
5   {
6     sum += window[i*3];
7   }
8   *mean = sum / wsize;
9 }
```

4.2.4 do_variance

`do_variance` calculates the variance of a set of floating point feature values within a given window of values within a feature array. This is defined by the formula:

$$\frac{\sum_{i=0}^{size-1}(window_i - mean)^2}{size}$$

where size denotes the feature window's size and mean is the mean value calculated by the `do_mean` function.

Given this function incorporates the result of another function, while also featuring a more demanding calculation loop, similar to the `mac` loop but with further arithmetic instructions to execute, this function acts as a stress test for kernel synthesis, both in terms of code size and resources needed.

For evaluation, window sizes of 200, 300 and 400 were considered for specialisation. Since the tools developed for specialised kernel source generation do not support nested functions, all specialised modifications of the original C source code (both manual and through the use of the dissertation work's tools) use a version of `do_variance` with the contents of the `do_mean` inlined within.

Listing 4.3: do_variance function C source code

```
1 #include "do_mean.h"
2
3 void do_variance(float *variance, float *window, int wsize)
```

Table 4.4: do_variance specialisation case overview

Name	Constants	LoC	#VarNodes	#OpNodes	#Nodes	#Edges
do_variance_200	wsize=200	410	1812	1204	3018	4217
do_variance_300	wsize=300	610	2712	1804	4518	6317
do_variance_400	wsize=400	810	3612	2404	6018	8417

```
4  {
5    float mean;
6
7    do_mean(&mean, window, wsize);
8
9    *variance = (float) 0.0;
10   for(int i=0; i < wsize; i++)
11     {
12     *variance += (window[i*3]-mean)*(window[i*3]-mean);
13   }
14   *variance = *variance / (wsize-1);
15 }
```

4.2.5 do_max

`do_max` returns the maximum value from a window of feature floating point values within a feature array. This simply keeps track of the maximum value, replacing it if a given value within the value set is higher than the recorded value.

This function was chosen as a test focused on performance of conditional operations, as it is the core operation of this kernel, aside from a fixed value multiplication that can be easily optimised away with additions. Branching is known to be hard to implement in hardware efficiently, therefore this case allows for isolation of that behaviour.

For evaluation, window sizes of 200, 300 and 400 were considered for specialisation.

Listing 4.4: do_max function C source code

```
1  #include "float.h"
2
3  void do_max(float *max, float *window, int wsize)
4  {
5    *max = FLT_MIN;
6    for(int i=0; i < wsize; i++)
7      {
8      if (*max < window[i*3]) *max = window[i*3];
9    }
10 }
```

Table 4.5: do_max specialisation case overview

Name	Constants	LoC	#VarNodes	#OpNodes	#Nodes	#Edges
do_max_200	wsize=200	205	804	601	1407	2004
do_max_300	wsize=300	305	1204	901	2107	3004
do_max_400	wsize=400	405	1604	1201	2807	4004

4.3 Test Results

The following subsections contain analyses of the test results for each evaluated function, coupled with tables listing the metrics obtained from said tests. Analysis of these results is split into two general performance areas: FPGA area usage and execution speed.

Tables are split in a similar fashion, with each function being given two tables for area usage metrics and one for execution speed metrics, which compare each of the generated specialised kernels with their repsective run of the original general-purpose kernel with matching parameters. The first area table contains the number of lookup tables (#LUT), flip-flops (#FF), Block RAMS (#BRAM) and digital signal processors (#DSP), with a second table containing the relative percentages for each element type relative to the base case. ; the speed table contains the total number of clock cycles (#Cycles), the obtained clock period (alongside its respective frequency), the execution time (Exec. Time) and the speedup relative to the original. Execution time is determined from the measured number of cycles and clock period through the formula $Cycles * ClockPeriod$, while speedup is obtained from the execution time of the specialised function (t_{spec}) and the original function (t_{orig}) by calculating t_{orig}/t_{spec}. For each metric, the best relative result is highlighted in bold. Every non-derived metric, with the exception of the number of clock cycles, was given by the synthesis report generated by Vitis; the clock cycle count was given by Vitis's cosimulation report.

For the `mac` function, execution times for the `dotgen` and `dot2c` tools, as well as memory allocated by `dot2c`, were collected for each generated specialisation case through performance counters built into each tool. Total processing times for Vitis HLS simulation, synthesis, and cosimulation were also measured for all kernel variants of each specialisation case.

It should be noted that, unlike the generated specialised functions, the original, general purpose function remains unchanged between test parameters, therefore the synthesised circuit remains the same and area usage metrics do not change between synthesis runs.

4.3.1 vote_kNN

Experimental results for `vote_kNN` are given in Tables 4.6 and 4.7 for area usage metrics and in Table 4.8 for execution speed metrics.

Analysis of area relevant performance metrics between the original `vote_knn` function and its generated counterparts shows immediately that all the specialised functions use more of each element than the original function, aside from DSPs. Regarding the influence of each hardcoded

parameter, it can be seen that the k parameter has more influence over flip-flop count than the number of classes, as well as being the only factor influencing the number of block RAM units used. Conversely, the number of classes seems to affect LUT count more than k. The loop specialised kernel sees a modest decrease in LUT and FF usage, which is only affected slightly by increasing values of `k` and `num_classes`. The unrolled specialised kernel shows slight deviations in resource usage, though largely negligible; this may be due to each source function's handling of conditional code, as the generated code substitutes conditional flow by ternary operations.

Block RAM usage can be explained by the fact that the high number of simultaneous operations the specialised kernel is trying to do at once requires more memory to process at the fastest speed possible, which does not occur when using simple loop parallelisation. Since the function only features a single addition operation in its source code, its arithmetic load is very low, which may explain why the synthesised kernel only requires 2 DSPs, no matter how big the function's source code is. Likewise, the fact that a fixed number of iterations is well defined in the specialised loop kernel also means the synthesiser knows in advance how many iterations each loop will take, thus allowing it to optimise for that specific number of iterations, which results in the decrease in used resources.

In terms of speed, an universal improvement in all categories is seen when comparing the specialised functions against their respective instance of the original. In fact, the original function is unable to meet the target period of 5ns due to memory dependencies, only achieving 6.312ns at most, while all specialised functions clear the requirement, while also taking less clock cycles to execute. The overall speedup between both versions tends to increase as `num_classes` increases, while decreasing as `k` increases. That being said, even in the worst tested case, the time taken by the optimised version is only 60% of the time taken by its general purpose counterpart. The pragma unrolled specialised kernel performs the same as the generated kernel for k=3, though for k=5 it suffers from a slightly higher cycle count, thus suggesting that the synthesised kernel is able to handle ternary operators better than conditional flow.

This influence the k parameter has over performance may be due to its corresponding loop block being the most computationally expensive, requiring a type conversion from floating point to integer, pointer lookups in memory and arithmetic. This all operates within a single memory area, meaning it is heavily subjected to memory dependencies on top of the expensive operations, thus requiring a higher clock period. This could also explain why the original kernel fails to meet its clock period target, as its main optimisation method is the parallelisation of its loops, which are even more heavily subjected to memory dependency related latency.

4.3.2 mac

Experimental results for `vote_kNN` are given in Tables 4.9 and 4.10 for area usage metrics and in Table 4.11 for execution speed metrics. In addition, this function was also used to measure time and memory taken during the entire generation and synthesis process for a wider range of kernels, the results of which are shown in Tables 4.12 and 4.13, including charts with generation and synthesis times in Figures 4.1 and 4.2, respectively.

Table 4.6: vote_kNN synthesis area metrics

Function	#LUT	#FF	#BRAM	#DSP
`vote_kNN`	3105	2443	**2**	**2**
`vote_kNN` (constants: k=3, n=6)	**2785**	**2116**	**2**	**2**
`vote_kNN` (`unroll`: k=3, n=6)	4275	2834	8	**2**
`vote_kNN_3_n6`	4276	2834	8	**2**
`vote_kNN`	3105	2443	**2**	**2**
`vote_kNN` (constants: k=5 n=6)	**2787**	**2118**	**2**	**2**
`vote_kNN` (`unroll`: k=5 n=6)	5295	3798	15	**2**
`vote_kNN_5_n6`	5256	3790	15	**2**
`vote_kNN`	3105	2443	**2**	**2**
`vote_kNN` (constants: k=3, n=22)	**2807**	**2176**	**2**	**2**
`vote_kNN` (`unroll`: k=3, n=22)	7528	3600	8	**2**
`vote_kNN_3_n22`	7528	3600	8	**2**
`vote_kNN`	3105	2443	**2**	**2**
`vote_kNN` (constants: k=5 n=22)	**2809**	**2178**	**2**	**2**
`vote_kNN` (`unroll`: k=5 n=22)	8552	4582	15	**2**
`vote_kNN_5_n22`	8518	4574	15	**2**

Table 4.7: vote_kNN synthesis area metrics comparison

Function	#LUT(%)	#FF(%)	#BRAM(%)	#DSP(%)
`vote_kNN`	100%	100%	**100%**	**100%**
`vote_kNN` (constants: k=3, n=6)	**90%**	**87%**	**100%**	**100%**
`vote_kNN` (`unroll`: k=3, n=6)	138%	116%	400%	**100%**
`vote_kNN_3_n6`	138%	116%	400%	**100%**
`vote_kNN`	100%	100%	**100%**	**100%**
`vote_kNN` (constants: k=5, n=6)	**90%**	**89%**	**100%**	**100%**
`vote_kNN` (`unroll`: k=5 n=6)	171%	155%	750%	**100%**
`vote_kNN_5_n6`	169%	155%	750%	**100%**
`vote_kNN`	100%	100%	**100%**	**100%**
`vote_kNN` (constants: k=3, n=22)	**90%**	**89%**	**100%**	**100%**
`vote_kNN` (`unroll`: k=3, n=22)	242%	147%	400%	**100%**
`vote_kNN_3_n22`	242%	147%	400%	**100%**
`vote_kNN`	100%	100%	**100%**	**100%**
`vote_kNN` (constants: k=5, n=22)	**90%**	**89%**	**100%**	**100%**
`vote_kNN` (`unroll`: k=5 n=22)	275%	188%	750%	**100%**
`vote_kNN_5_n22`	274%	187%	750%	**100%**

Table 4.8: vote_kNN synthesis temporal performance metrics

Function	Latency	Clock Period (Frequency)	Exec. Time	Speedup
`vote_knn` (k=3, n=6)	180	6.312ns (158.428 MHz)	1136.16ns	1.00
`vote_knn` (constants)	160	6.312ns (158.428 MHz)	1009.92ns	1.13
`vote_knn` (`unroll`)	**152**	**3.650ns (273.972 MHz)**	**554.8ns**	**2.05**
`vote_knn_3_n6`	**152**	**3.650ns (273.972 MHz)**	**554.8ns**	**2.05**
`vote_knn` (k=5, n=6)	201	6.312ns (158.428 MHz)	1268.712ns	1.00
`vote_kNN` (constants)	182	6.312ns (158.428 MHz)	1148.784ns	1.11
`vote_kNN` (`unroll`)	178	**4.524ns (221.043 MHz)**	805.272ns	1.58
`vote_knn_5_n6`	**168**	**4.524ns (221.043 MHz)**	**760.032ns**	**1.67**
`vote_knn` (k=3, n=22)	292	6.312ns (158.428 MHz)	1843.104ns	1.00
`vote_kNN` (constants)	257	6.312ns (158.428 MHz)	1622.184ns	1.14
`vote_kNN` (`unroll`)	**232**	**3.650ns (273.972 MHz)**	**846.8ns**	**2.18**
`vote_knn_3_n22`	**232**	**3.650ns (273.972 MHz)**	**846.8ns**	**2.18**
`vote_knn` (k=5, n=22)	313	6.312ns (158.428 MHz)	1975.656ns	1.00
`vote_kNN` (constants)	279	6.312ns (158.428 MHz)	1761.048ns	1.12
`vote_kNN` (`unroll`)	**258**	**4.524ns (221.043 MHz)**	1167.192ns	1.69
`vote_knn_5_n22`	**248**	**4.524ns (221.043 MHz)**	**1121.952ns**	**1.76**

Regarding area, the expected increase in overall element usage with the specialised methods is recorded, though now the number of DSPs also increase. The increase in BRAM and DSP usage seems exponential, as by `mac_16` it seems the maximum amount of BRAM and DSP resources is hit, and LUT and FF resource usage increases exponentially. This trend continues in `mac_64`, though the increase in LUT and FF usage is not as great. In contrast to what was observed with `vote_kNN` (Subsection 4.3.1), the element usage of the loop specialised kernel is larger than that of the original; of special mention is that while it does not use any more DSPs than the original kernel, it uses as many BRAMs as the generated kernel, meaning it too runs into the same bottleneck as the former, which results in a large increase in LUTs and FFs. This may happen due to unrolling optimisations made by the HLS synthesiser. Given the absence of conditionals, the pragma unrolled kernel performed exactly the same as the generated kernel.

The DSP and BRAM bottleneck is reflected in the execution speed metrics, where `mac_4` achieves a respectable speedup of 2.12, but then dives as `mac_16` only is able to achieve a speedup of 1.15, and `mac_64` does not manage to be faster than the general purpose kernel, at a speedup of 0.94. This is chiefly due to the increase in the obtained clock period that occurs due to the aforementioned bottleneck, which for both cases is higher than in the original kernel, and gets higher as the size parameter the kernel is generated against is increased. `tripcount` optimised versions have no effect over performance at all, while loop specialisation, despite it suffering from a BRAM bottleneck, does not suffer relative speedup decreases at nearly the same rate as the generated kernel and maintains the same clock period, which makes it the fastest kernel for both n=16 and n=64.

This experiment conveys that this kind of optimisation is not effective when dealing with

Table 4.9: mac synthesis area metrics

Function	#LUT	#FF	#BRAM	#DSP
`mac`	**2319**	**2121**	**2**	**5**
`mac` (constants: n=4)	2523	2243	8	**5**
`mac` (`unroll`: n=4)	3660	2803	8	14
`mac_4`	3660	2803	8	14
`mac`	**2319**	**2121**	**2**	**5**
`mac` (constants: n=16)	3327	4197	29	**5**
`mac` (`unroll`: n=16)	9058	7261	29	50
`mac_16`	9058	7261	29	50
`mac`	**2319**	**2121**	**2**	**5**
`mac` (constants: n=64)	3369	11024	29	**5**
`mac` (`unroll`: n=64)	11458	10912	29	50
`mac_64`	11458	10912	29	50

limited resources, as this method relies on heavy use of hardware resources to achieve its high speed improvements; once these resources run out, performance heavily decreases, be it in regard to clock period increase or compensating for the lack of scarce hardware resources with a larger volume of other types of resources. The nature of multiplication being an expensive operation in terms of digital circuit resources means that it must make use of more DSPs than usual to ensure the best performance possible. The fact that the simple loop-based specialisation approach works best for large n values due to it not requiring as many DSP resources corroborates this theory.

Analysis of function generation and synthesis times shows that the time taken to run the source-to-source pipeline is very small compared to the time taken by the HLS tools to synthesise the corresponding kernel. `dotgen`'s speed results in its time taken being largely negligible,and even `dot2c`, running under the less performant Python runtime environment, still represents a fraction of the time taken by the equivalent synthesis process. It also seems that `dot2c` does not allocate much memory, achieving a little over 1MiB in most cases, which also is constant, possibly due to preemptive memory allocation by the Python runtime. In terms of synthesis, unrolling seems to only affect the time taken significantly at the 128 iteration mark, though afterwards it exponentially increases until it surpasses 6 minutes when synthesising for 1024 iterations; when not unrolling, synthesis time remains constant at about 9 seconds.

4.3.3 do_mean

Tables 4.14 and 4.15 list the area usage metrics for each evaluated case in the `do_mean` function; Table 4.16 lists execution performance metrics for the same cases.

Lookup table and flip-flop usage for kernels where unrolling was selected exhibit a massive increase relative to the other cases, exhibiting as much as 1540% for the largest number of iterations unrolled, and even reaching 812% on the lowest case. This large difference relative to previously tested functions can be explained by the usage of much larger iteration counts while

Table 4.10: mac synthesis area metrics comparison

Function	#LUT	#FF	#BRAM	#DSP
`mac`	**100%**	**100%**	**100%**	**100%**
`mac` (constants: n=4)	109%	106%	400%	**100%**
`mac` (`unroll`: n=4)	158%	132%	400%	280%
`mac_4`	158%	132%	400%	280%
`mac`	**100%**	**100%**	**100%**	**100%**
`mac` (constants: n=16)	143%	198%	1450%	**100%**
`mac` (`unroll`: n=16)	391%	342%	1450%	1000%
`mac_16`	391%	342%	1450%	1000%
`mac`	**100%**	**100%**	**100%**	**100%**
`mac` (constants: n=64)	145%	520%	1450%	**100%**
`mac` (`unroll`: n=64)	494%	514%	1450%	1000%
`mac_64`	494%	514%	1450%	1000%

Table 4.11: mac synthesis temporal performance metrics

Function	#Cycles	Clock Period (Frequency)	Exec. Time	Speedup
`mac` (n=4)	194	**3.650ns (273.972 MHz)**	708.1ns	1.00
`mac` (constants: n=4)	132	**3.650ns (273.972 MHz)**	481.8ns	1.47
`mac` (`unroll`: n=4)	**126**	**3.650ns (273.972 MHz)**	**333.9ns**	**2.12**
`mac_4`	**126**	**3.650ns (273.972 MHz)**	**333.9ns**	**2.12**
`mac` (n=16)	334	**3.650ns (273.972 MHz)**	1219.1ns	1.00
`mac` (constants: n=16)	264	**3.650ns (273.972 MHz)**	**963.6ns**	**1.26**
`mac` (`unroll`: n=16)	**246**	4.309ns (232.072 MHz)	1060.014ns	1.15
`mac_16`	**246**	4.309ns (232.072 MHz)	1060.014ns	1.15
`mac` (n=64)	887	**3.650ns (273.972 MHz)**	3237.55ns	1.00
`mac` (constants: n=64)	801	**3.650ns (273.972 MHz)**	**2923.65ns**	**1.11**
`mac` (`unroll`: n=64)	**729**	4.738ns (211.060 MHz)	3454.002ns	0.94
`mac_64`	**729**	4.738ns (211.060 MHz)	3454.002ns	0.94

Table 4.12: mac specialised function source generation processing metrics

Function	`dotgen` time (ms)	`dot2c` time (ms)	`dot2c` memory (B)
`mac_4`	0.043	14.885	1309502
`mac_8`	0.121	24.156	1309502
`mac_16`	0.250	29.213	1309502
`mac_32`	0.478	48.261	1309502
`mac_64`	1.293	92.358	1309502
`mac_128`	5.561	169.035	1309502
`mac_256`	17.042	347.954	1309502
`mac_512`	66.135	673.458	1309502
`mac_1024`	279.188	1388.090	2130880

Table 4.13: mac specialised HLS kernel synthesis and cosimulation processing time (OutMem: out of memory crash)

Function	Simulation	Synthesis	Cosimulation	Total
`mac` (4)	0.86s	9.14s	17.91s	27.91s
`mac` (constants)	0.81s	9.05s	31.38s	41.24s
`mac` (`unroll`)	0.81s	9.13s	31.39s	41.33s
`mac_4`	1.05s	9.57s	30.66s	41.29s
`mac` (8)	0.86s	9.16s	18.13s	28.15s
`mac` (constants)	0.78s	9.26s	31.18s	41.22s
`mac` (`unroll`)	0.76s	9.25s	33.83s	43.84s
`mac_8`	0.84s	9.26s	31.32s	41.44s
`mac` (16)	0.86s	9.25s	18.13s	28.16s
`mac` (constants)	0.82s	9.26s	33.75s	43.83s
`mac` (`unroll`)	0.79s	9.35s	35.32s	45.46s
`mac_16`	0.85s	9.31s	31.65s	41.84s
`mac` (32)	0.85s	9.26s	18.14s	28.25s
`mac` (constants)	0.79s	9.23s	30.54s	40.56s
`mac` (`unroll`)	0.79s	9.47s	31.50s	41.76s
`mac_32`	0.88s	9.54s	33.87s	44.34s
`mac` (64)	0.85s	9.13s	18.26s	28.25s
`mac` (constants)	0.82s	9.28s	33.41s	40.56s
`mac` (`unroll`)	0.77s	9.88s	34.13s	41.76s
`mac_64`	0.86s	9.99s	31.95s	44.34s
`mac` (128)	0.83s	9.09s	17.95s	27.87s
`mac` (constants)	0.82s	9.32s	35.06s	45.20s
`mac` (`unroll`)	0.83s	11.65s	42.01s	54.49s
`mac_128`	0.84s	11.47s	39.47s	51.95s
`mac` (256)	0.88s	9.29s	18.99s	29.16s
`mac` (constants)	0.85s	9.53s	36.29s	46.67s
`mac` (`unroll`)	0.82s	18.73s	62.30s	81.85s
`mac_256`	0.93s	18.63s	60.76s	80.68s
`mac` (512)	1.18s	19.31s	19.52s	40.01s
`mac` (constants)	1.43s	9.78s	37.98s	49.19s
`mac` (`unroll`)	0.87s	66.82s	155.63s	223.32s
`mac_512`	0.89s	67.00s	157.27s	225.90s
`mac` (1024)	1.57s	9.83s	21.03s	32.43s
`mac` (constants)	1.07s	9.44s	34.15s	44.66s
`mac` (`unroll`)	0.82s	390.51s	OutMem	OutMem
`mac_1024`	0.98s	394.77s	OutMem	OutMem

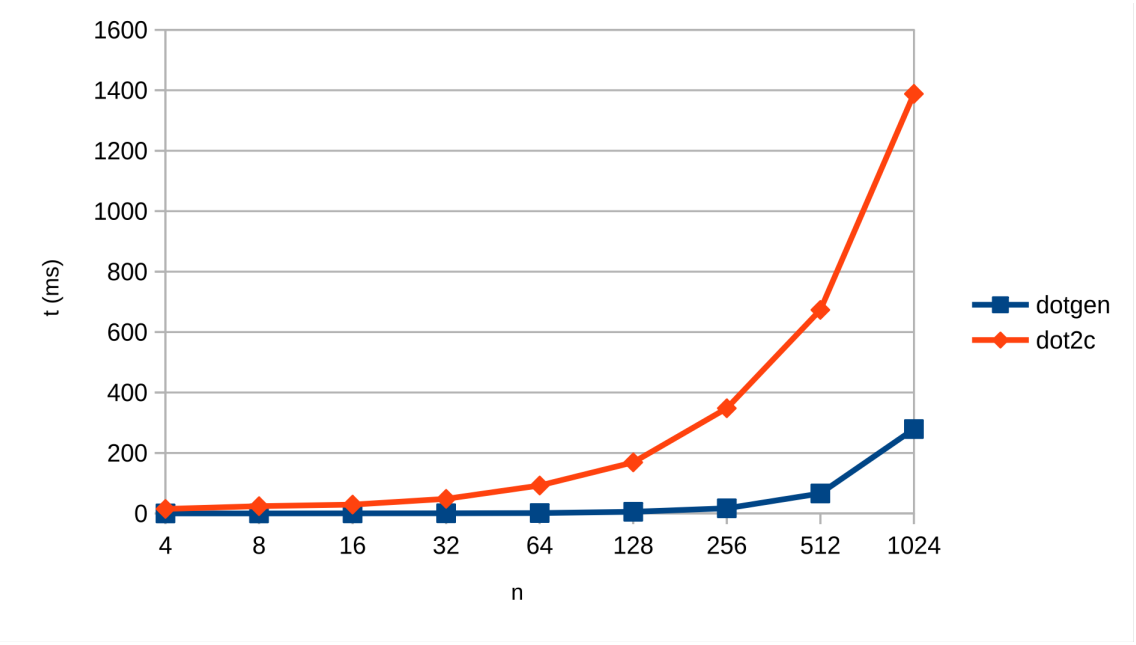


Figure 4.1: mac specialisation generation times for different data sizes

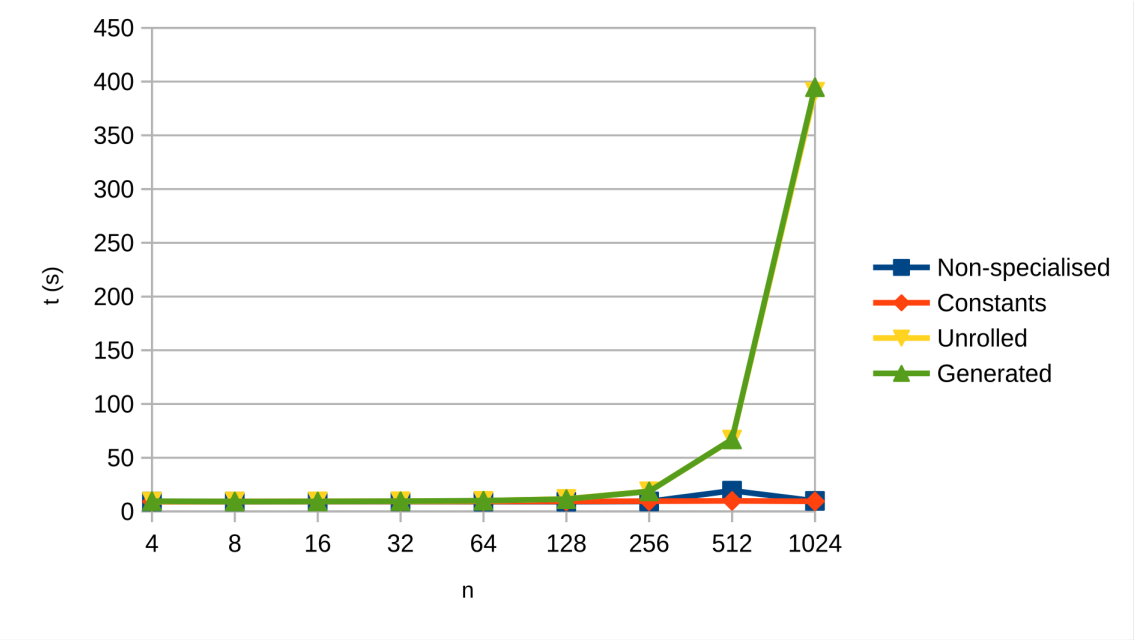


Figure 4.2: mac HLS times for different data sizes

Table 4.14: do_mean synthesis area metrics

Function	#LUT	#FF	#BRAM	#DSP
`do_mean` (wsize=200)	2535	2033	**2**	**2**
`do_mean` (constants: wsize=200)	**2374**	**1857**	**2**	**2**
`do_mean` (`unroll`: wsize=200)	20495	16511	**2**	**2**
`do_mean_200`	20495	16511	**2**	**2**
`do_mean` (wsize=300)	2535	2033	**2**	**2**
`do_mean` (constants: wsize=300)	**2374**	**1860**	**2**	**2**
`do_mean` (`unroll`: wsize=300)	28643	23911	**2**	**2**
`do_mean_300`	28643	23911	**2**	**2**
`do_mean` (wsize=400)	2535	2033	**2**	**2**
`do_mean` (constants: wsize=400)	**2374**	**1860**	**2**	**2**
`do_mean` (`unroll`: wsize=400)	36783	31311	**2**	**2**
`do_mean_400`	36783	31311	**2**	**2**

evaluating this function, but it helps demonstrate how much area usage can balloon when relying on extreme unrolling. Furthermore, similarly to previous findings, both unrolled kernels synthesise to effectively the same result, due to largely identical resulting code. Simple specialisation with no unrolling continues to be the least demanding in terms of resource usage, though it can only achieve 94% LUT usage and about 91% flip-flop usage relative to the base case. Block RAM and DSP usage does not change at all between optimisation strategies, which is a likely byproduct of the small amount of data to retain and the low arithmetic requirements of the function.

In terms of performance, `do_mean` seems to perform better in every case when not employing unrolling, as the constant specialised kernel ends up with a significant speedup compared to all other cases when running at the maximum possible clock period; the unrolled kernels, both identical, manage to run in less cycles, but suffer by requiring higher clock periods than the stipulated target of 5 ns. Curiously, this increase in clock speed is not linear, being highest for 300 iterations and lowest for 400 iterations; this likely points to the HLS tools using a different strategy for synthesising the larger kernel, resulting in a smaller critical path and, consequently, a smaller minimum clock frequency attainable.

4.3.4 do_variance

Tables 4.17 and 4.18 list the area usage metrics for each evaluated case in the `do_variance` function; Table 4.19 lists execution performance metrics for the same cases.

Analysis of resource usage across kernels reveals that the unrolled kernels feature much larger area usage ratios when compared to even `do_mean`: 5645% LUT and 7628% flip-flop usage relative to the base case in the largest scenario, and 2879% LUT and 3842% flip-flop usage in the smallest scenario, which beats the largest `do_mean` scenario's ratios. Additionally, unrolling leads to massive increases in DSP usage, going from 5 to 403-803 DSPs needed for the kernel. This can be explained through the presence of many multiplication operations within the variance

Table 4.15: do_mean synthesis area metrics comparison

Function	#LUT	#FF	#BRAM	#DSP
`do_mean` (wsize=200)	100%	100%	**100%**	**100%**
`do_mean` (constants: wsize=200)	**94%**	**91.3%**	**100%**	**100%**
`do_mean` (`unroll`: wsize=200)	808%	812%	**100%**	**100%**
`do_mean_200`	808%	812%	**100%**	**100%**
`do_mean` (wsize=300)	100%	100%	**100%**	**100%**
`do_mean` (constants: wsize=300)	**94%**	**91.5%**	**100%**	**100%**
`do_mean` (`unroll`: wsize=300)	1130%	1176%	**100%**	**100%**
`do_mean_300`	1130%	1176%	**100%**	**100%**
`do_mean` (wsize=400)	100%	100%	**100%**	**100%**
`do_mean` (constants: wsize=400)	**94%**	**91.5%**	**100%**	**100%**
`do_mean` (`unroll`: wsize=400)	1451%	1540%	**100%**	**100%**
`do_mean_400`	1451%	1540%	**100%**	**100%**

Table 4.16: do_mean synthesis temporal performance metrics

Function	#Cycles	Clock Period (Frequency)	Exec. Time	Speedup
`do_mean` (wsize=200)	2556	6.312ns (158.428 MHz)	16120.848ns	1.00
`do_mean` (constants)	2446	**3.650ns (273.972 MHz)**	**8927.9ns**	**1.81**
`do_mean` (`unroll`)	**2252**	5.279ns (189.430 MHz)	11888.308ns	1.36
`do_mean_200`	**2252**	5.279ns (189.430 MHz)	11888.308ns	1.36
`do_mean` (wsize=300)	3712	6.312ns (158.428 MHz)	23430.144ns	1.00
`do_mean` (constants)	3608	**3.650ns (273.972 MHz)**	**13169.2ns**	**1.78**
`do_mean` (`unroll`)	**3291**	6.375ns (156.863 MHz)	20980.125ns	1.12
`do_mean_300`	**3291**	6.375ns (156.863 MHz)	20980.125ns	1.12
`do_mean` (wsize=400)	4870	6.312ns (158.428 MHz)	30739.44ns	1.00
`do_mean` (constants)	4753	**3.650ns (273.972 MHz)**	**17348.45ns**	**1.77**
`do_mean` (`unroll`)	**4327**	5.262 (190.042 MHz)	22768.674ns	1.35
`do_mean_400`	**4327**	5.262 (190.042 MHz)	22768.674ns	1.35

Table 4.17: do_variance synthesis area metrics

Function	#LUT	#FF	#BRAM	#DSP
`do_variance` (wsize=200)	3709	2712	**2**	**5**
`do_variance` (constants: wsize=200)	**3435**	**2434**	**2**	**5**
`do_variance` (`unroll`: wsize=200)	106789	104193	**2**	403
`do_variance_200`	106789	104193	**2**	403
`do_variance` (wsize=300)	3709	2712	**2**	**5**
`do_variance` (constants: wsize=300)	**3435**	**2440**	**2**	**5**
`do_variance` (`unroll`: wsize=300)	158086	155493	**2**	603
`do_variance_300`	158086	155493	**2**	603
`do_variance` (wsize=400)	3709	2712	**2**	**5**
`do_variance` (constants: wsize=400)	**3435**	**2440**	**2**	**5**
`do_variance` (`unroll`: wsize=400)	209380	206863	**2**	803
`do_variance_400`	209380	206863	**2**	803

calculation accumulator loop, which demands the allocation of DSPs for each individual iteration. the constant specialisation kernel yields the best results again, with better but similarly modest improvements in resource consumption relative to what was seen in `do_mean`. Again, both the tool generated kernel and the pragma unrolled kernel compile to effectively the same result.

The massive increase in synthesis complexity described previously leads to the first cases in which cosimulation was not possible due to out of memory crashes from the HLS tools for all but the smallest case for the unrolled kernels. Thus, only the clock period could be estimated for many of these cases, which still falls short of meeting the 5 ns constraint in all situations, though it remains shorter than that of the base case. It still can be verified, at least for the 200 iteration case, that the unrolled method still demonstrates the smallest cycle count, with good results relative to constant unrolling in that field. However, the constant unrolling kernel still wins in overall speedup due to its small minimum clock period.

4.3.5 do_max

Tables 4.17 and 4.18 list the area usage metrics for each evaluated case in the `do_variance` function; Table 4.19 lists execution performance metrics for the same cases.

Resource usage for LUTs and flip-flops resides in a middle ground between `do_mean` and `do_variance`: the constant specialised kernel features decent improvements on usage of these hardware resources, whereas the unrolled cases still represent a massive increase in usage over the non specialised kernel, still surpassing the relative usage at 200 iterations over `do_mean` at 400 iterations; of special note is that the two unrolled kernels are not identical for this function, with the tool generated function synthesising with larger LUT usage, but lower flip-flop usage than the pragma unrolled version. This can be explained by the fact the tools cannot handle true conditional program flows, substituting those with ternary operations, which the HLS tools seem to interpret

Table 4.18: do_variance synthesis area metrics comparison

Function	#LUT	#FF	#BRAM	#DSP
`do_variance` (wsize=200)	100%	100%	**100%**	**100%**
`do_variance` (constants: wsize=200)	**93%**	**89.7%**	**100%**	**100%**
`do_variance` (`unroll`: wsize=200)	2879%	3842%	**100%**	8060%
`do_variance_200`	2879%	3842%	**100%**	8060%
`do_variance` (wsize=300)	100%	100%	**100%**	**100%**
`do_variance` (constants: wsize=300)	**93%**	**90%**	**100%**	**100%**
`do_variance` (`unroll`: wsize=300)	4262%	5734%	**100%**	12060%
`do_variance_300`	4262%	5734%	**100%**	12060%
`do_variance` (wsize=400)	100%	100%	**100%**	**100%**
`do_variance` (constants: wsize=400)	**93%**	**90%**	**100%**	**100%**
`do_variance` (`unroll`: wsize=400)	5645%	7628%	**100%**	16060%
`do_variance_400`	5645%	7628%	**100%**	16060%

Table 4.19: do_variance synthesis temporal performance metrics

Function	#Cycles	Clock Period (Frequency)	Exec. Time	Speedup
`do_variance` (200)	5002	6.312ns (158.428 MHz)	31572.624ns	1.00
`do_variance` (constants)	4875	**3.650ns (273.973 MHz)**	**17793.75ns**	**1.77**
`do_variance` (`unroll`)	**4300**	5.262ns (190.042 MHz)	22626.6ns	1.40
`do_variance_200`	**4300**	5.262ns (190.042 MHz)	22626.6ns	1.40
`do_variance` (300)	7312	6.312ns (158.428 MHz)	46153.344ns	1.00
`do_variance` (constants)	**7159**	**3.650ns (273.972 MHz)**	**26130.35ns**	**1.77**
`do_variance` (`unroll`)	N/A	5.585ns (179.051 MHz)	N/A	N/A
`do_variance_300`	N/A	5.585ns (179.051 MHz)	N/A	N/A
`do_variance` (400)	9634	6.312ns (158.428 MHz)	60809.808ns	1.00
`do_variance` (constants)	**9449**	**3.650ns (273.972 MHz)**	**34488.85ns**	**1.76**
`do_variance` (`unroll`)	N/A	5.817ns (171.910 MHz)	N/A	N/A
`do_variance_400`	N/A	5.817ns (171.910 MHz)	N/A	N/A

Table 4.20: do_max synthesis area metrics

Function	#LUT	#FF	#BRAM	#DSP
do_max (wsize=200)	1928	1650	**2**	**0**
do_max (constants: wsize=200)	**1788**	**1492**	**2**	**0**
do_max (unroll: wsize=200)	61915	39895	**2**	**0**
do_max_200	44111	33458	**2**	**0**
do_max (wsize=300)	1928	1650	**2**	**0**
do_max (constants: wsize=300)	**1788**	**1495**	**2**	**0**
do_max (unroll: wsize=300)	57950	92339	**2**	**0**
do_max_300	64283	49558	**2**	**0**
do_max (wsize=400)	1928	1650	**2**	**0**
do_max (constants: wsize=400)	**1788**	**1495**	**2**	**0**
do_max (unroll: wsize=400)	75986	122731	**2**	**0**
do_max_400	84443	65658	**2**	**0**

differently during synthesis. Block RAM and DSP usage remains constant for all kernels, with no DSPs being used at all due to the lack of arithemtic operation within the function.

All kernels managed to achieve the same clock period of 3.65ns, which leads the tool generated unrolled function to provide the best speedup in every case due to its usual lead in cycle count over other cases, though for this function the achievable speedup is much lower than what was achieved for do_mean and do_variance.

Of special note is the pragma unrolled kernel, which failed to run cosimulation for every evaluated iteration count, and whose synthesis process took much longer than any other kernel evaluated in this work; while concrete measurements were not taken for most cases, most kernels would synthesise in less than a minute, with the more demanding unrolled kernels taking no more than 10 minutes, as demonstrated with mac's case as shown in Table 4.13. For do_max, however, synthesis took 48 minutes for 200 iterations, 2 hours and 12 minutes for 300 iterations, and 5 hours and 5 hours and 23 minutes for 400 iterations. Cosimulation failed due to out of memory crashes, as discussed in earlier sections.

4.4 Summary

This chapter described the protocol carried out for evaluation of the method developed for this work, and laid out its results and observations extracted from those. The developed specialisation method was compared against other existing specialisation methods, across 5 functions representing a wide range of cases for HLS kernel synthesis, with each being tested against different specialisation parameters.

The results obtained were largely predictable: the existing tools only implement the loop unrolling optimisation strategy, so using existing pragmas for that effect yielded the same results, except in cases where conditional code was heavily involved, since the developed tool does not implement conditional branching. In those cases, this new tool exhibits an advantage over the HLS

Table 4.21: do_max synthesis area metrics comparison

Function	#LUT	#FF	#BRAM	#DSP
`do_max` (wsize=200)	100%	100%	**100%**	**100%**
`do_max` (constants: wsize=200)	**93%**	**90%**	**100%**	**100%**
`do_max` (`unroll`: wsize=200)	3211%	2418%	**100%**	**100%**
`do_max_200`	2288%	2028%	**100%**	**100%**
`do_max` (wsize=300)	100%	100%	**100%**	**100%**
`do_max` (constants: wsize=300)	**93%**	**91%**	**100%**	**100%**
`do_max` (`unroll`: wsize=300)	3006%	5596%	**100%**	**100%**
`do_max_300`	3334%	3004%	**100%**	**100%**
`do_max` (wsize=400)	100%	100%	**100%**	**100%**
`do_max` (constants: wsize=400)	**93%**	**91%**	**100%**	**100%**
`do_max` (`unroll`: wsize=400)	3941%	7438%	**100%**	**100%**
`do_max_400`	4380%	3979%	**100%**	**100%**

Table 4.22: do_max synthesis temporal performance metrics

Function	#Cycles	Clock Period (Frequency)	Exec. Time	Speedup
`do_max` (200)	1342	**3.650ns (273.973 MHz)**	4898.3ns	1.00
`do_max` (constants)	1199	**3.650ns (273.973 MHz)**	4376.35ns	1.12
`do_max` (`unroll`)	N/A	**3.650ns (273.973 MHz)**	N/A	N/A
`do_max_200`	**1189**	**3.650ns (273.973 MHz)**	**4339.85**	**1.13**
`do_max` (300)	1864	**3.650ns (273.973 MHz)**	6803.6ns	1.00
`do_max` (constants)	1717	**3.650ns (273.973 MHz)**	6267.05ns	1.086
`do_max` (`unroll`)	N/A	**3.650ns (273.973 MHz)**	N/A	N/A
`do_max_300`	**1707**	**3.650ns (273.973 MHz)**	**6230.55**	**1.091**
`do_max` (400)	2392	**3.650ns (273.973 MHz)**	8730.8ns	1.00
`do_max` (constants)	2238	**3.650ns (273.973 MHz)**	8168.7ns	1.069
`do_max` (`unroll`)	N/A	**3.650ns (273.973 MHz)**	N/A	N/A
`do_max_400`	**2228**	**3.650ns (273.973 MHz)**	**8132.2**	**1.074**

tool pragma in terms of resources used during synthesis of the hardware kernel, and potentially modest improvements in execution time as well. Furthermore, this unrolling method was largely characterised by an extreme tradeoff of hardware resource usage for cycle count, with its largest bottleneck being the length of the critical path impacting the maximum clock frequency possible.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Through the work carried out for this dissertation, a new graph based method for specialisation-based optimisation of HLS hardware kernels written in the C language was devised. It is believed that this methodology has great potential for implementation of various optimisation strategies, and thus it is hoped that the work laid out here inspires further exploration of its possibilities for hardware acceleration.

That being said, the current state of the work is admittedly minimal: the only optimisation strategy employed thus far has been loop unrolling, which is already offered by existing HLS tools as a simple option and provides largely similar results, the devised graph format is limited in functionality and does not account for certain use cases, the developed tools do not fully automatise the specialisation process, requiring manual intervention, and further evaluation would be needed to confidently assert some of its findings, such as synthesis times and the influence of replacement of conditionals with ternary operators.

Finally, the following list answers the research questions posited in Section 1.4:

- **RQ1** – The developed methodology is one that explores the extreme case of trading as many hardware resources as possible for maximum speed gains. This methodology, while effective, is limited by the scarcity of the resources it requires.
- **RQ2** – This work largely explored how specialisation can be used for optimisation of hardware kernels. Multiple methods were compared, and it was concluded that specialisation through substitution of loop iteration counts with constants largely provided the best area-to-speed performance ratio, while the method devised in this work relies on large trade-offs of area for speed.
- **RQ3** – The automatic generation pipeline devised for the specialisation methodology is able to mostly programmatically generate a highly specialised C language hardware kernel from a given general-purpose C source function; it is, however, clear that this process can be completely automatised.

- **RQ4** – Several code optimisation strategies were tested - loop trip count limitation, fixed-iteration loop specialisation, and the method created by this work - and all featured distinct results: limiting trip count had negligible gains in performance in some cases, while being completely identical in others; loop-based specialisation provided good improvements in both area and speed performance, except in cases where it overused scarce hardware resources; the devised method used as many hardware resources as possible to guarantee the fastest performance possible, which may fail to scale if scarce resources are exhausted. Furthermore, loop unrolling introduces more overhead in terms of synthesis time, possibly becoming unreasonable if too much conditional code must be accounted for.

5.2 Future Work

This work leaves many further ideas to be explored, which are listed below.

5.2.1 Issues with the Current Implementation

- The intermediate CDFG representation does not support true conditional blocks, therefore complex conditional logic is not effectively supported (such as conditional return statements).
- The CDFG has no support for function calls, which could be implemented under a function hierarchy system.
- `dotgen` requires manual intervention to instrumentise and set up the tool for proper usage. In the short term, a scheme similar to how testing is set up with the `testgen` tool could be used, but the ideal solution would be to generate the intermediate representation from either the source C file itself (e.g. with a tool that automates instrumentation such as Clava [5]), or an intermediate representation of said file generated by a compiler.
- The current scheme regarding configuration files for each tool introduces a lot of redundancy, as function metadata must be repeated for both `dot2c` and `testgen`'s configuration JSON files. This could be consolidated into a single file in the future.
- The current source-to-source pipeline cannot account for kernels consisting of multiple functions nested within a top function.
- Many fields defined within the IR CDFG are left unused by the tools that operate over it, thus further refinement of its format is possible to further optimise its structure, in addition to defining more opcodes it could use (such as modulo operations, function calls, et cetera).

5.2.2 Issues with Evaluation

- The impact of individual synthesised logic blocks in latency was not taken into account during analysis of each kernel; this information would allow for more detailed insights into how latency varies between implementations.
- A wider range of test functions could have been tested, such as kernels that feature both intensive arithmetic and memory storage use. A promising blind spot is that of specialisation cases which rely on more than iteration counts being fixed, such as value tables that can be constant for given specialisations.
- The performance effect of replacing conditional code with ternary operators should be evaluated further in the future, as while promising the data collected for it is insufficient to indicate accurately how it affects overall performance.

5.2.3 Other Future Improvements

- A hybrid solution, where both the general purpose and specialised versions are included within the same kernel and the datapath is chosen based on the parameters given, could also be implemented and tested.
- Further optimisations could be implemented by the generation tool, guided by templates, such as the folding of repeating patterns and culling of redundant memory accesses covered in Ferreira [10] and Campos [6].
- The CDFG could be extended with a hierarchical structure, capable of representing function calls.

References

[1] AMD. *Vitis High-Level Synthesis User Guide (UG1399)*. 2025. URL: https://docs.amd.com/r/en-US/ug1399-vitis-hls/ (visited on 09/10/2025).

[2] AMD. *Xilinx Vitis 2022.2*. 2026. URL: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis/2022-2.html (visited on 01/31/2026).

[3] AMD. *Xilinx Vitis 2025.2*. 2026. URL: https://www.amd.com/en/support/downloads/adaptive-socs-and-fpgas/development-tools/2025-2.html (visited on 09/11/2026).

[4] The Graphviz Authors. *DOT Language | Graphviz*. 2026. URL: https://graphviz.org/doc/info/lang.html (visited on 08/16/2026).

[5] João Bispo and João M.P. Cardoso. "Clava: C/C++ source-to-source compilation using LARA". In: *SoftwareX* 12 (2020), p. 100565. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100565. URL: https://www.sciencedirect.com/science/article/pii/S2352711019302122.

[6] Cardoso Campos. "On Data Parallelism Code Restructuring for HLS Targeting FPGAs". In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, pp. 144–151. DOI: 10.1109/IPDPSW52791.2021.00029.

[7] Jianyi Cheng et al. "DASS: Combining Dynamic & Static Scheduling in High-Level Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.3 (2022), pp. 628–641. DOI: 10.1109/TCAD.2021.3065902.

[8] Philippe Coussy et al. "An Introduction to High-Level Synthesis". In: *IEEE Design & Test of Computers* 26.4 (2009), pp. 8–17. DOI: 10.1109/MDT.2009.69.

[9] NetworkX developers. *NetworkX - NetworkX documentation*. 2026. URL: https://networkx.org/en/ (visited on 09/18/2026).

[10] Cardoso Ferreira. "Graph-Based Code Restructuring Targeting HLS for FPGAs". In: *Applied Reconfigurable Computing - 15th International Symposium, ARC 2019, Darmstadt, Germany, April 9-11, 2019, Proceedings*. Ed. by Christian Hochberger et al. Vol. 11444. Lecture Notes in Computer Science. Springer, 2019, pp. 230–244. DOI: 10.1007/978-3-030-17227-5_17. URL: https://doi.org/10.1007/978-3-030-17227-5%5C_17.

[11] Juan Miguel de Haro et al. "OmpSs@FPGA framework for high performance FPGA computing". In: *IEEE Transactions on Computers* 70.12 (June 2, 2021), pp. 2029–2042. ISSN: 0018-9340, 1557-9956, 2326-3814. DOI: 10.1109/TC.2021.3086106. URL: https://ieeexplore.ieee.org/document/9445632/ (visited on 11/30/2025).

[12] José A. M. de Holanda et al. “JACABench - The JACA Benchmark Suite for Embedded Computing”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by Gianluca Leone et al. Cham: Springer Nature Switzerland, 2027, pp. 100–115. ISBN: 978-3-032-29365-7.

[13] Qijing Huang et al. “The Effect of Compiler Optimizations on High-Level Synthesis-Generated Hardware”. In: *ACM Transactions on Reconfigurable Technology and Systems* 8.3 (May 19, 2015), pp. 1–26. ISSN: 1936-7406, 1936-7414. DOI: 10.1145/2629547. URL: https://dl.acm.org/doi/10.1145/2629547 (visited on 11/30/2025).

[14] Intel. *Intel oneAPI DPC++/C++ Compiler Handbook for Intel FPGAs*. 2025. URL: https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/developer-guide/2025-0/fpga-handbook.html (visited on 05/05/2025).

[15] Ian Smalley Josh Schneider. *What is a field programmable gate array (FPGA)? | IBM*. URL: https://www.ibm.com/think/topics/field-programmable-gate-arrays (visited on 10/05/2026).

[16] Ian Smalley Josh Schneider. *What is a neural processing unit (NPU)? | IBM*. URL: https://www.ibm.com/think/topics/neural-processing-unit (visited on 10/05/2026).

[17] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. “From C/C++ Code to High-Performance Dataflow Circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.7 (2022), pp. 2142–2155. DOI: 10.1109/TCAD.2021.3105574.

[18] Leibo Liu et al. “A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications”. In: *ACM Comput. Surv.* 52.6 (Oct. 2019). ISSN: 0360-0300. DOI: 10.1145/3357375. URL: https://doi.org/10.1145/3357375.

[19] Nuno Paulino et al. “A Binary Translation Framework for Automated Hardware Generation”. In: *IEEE Micro* 41.4 (2021), pp. 15–23. DOI: 10.1109/MM.2021.3088670.

[20] Murad Qasaimeh et al. “Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels”. In: *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. 2019, pp. 1–8. DOI: 10.1109/ICESS.2019.8782524.

[21] Yuhan She et al. “A Source-to-Source Compiler to Enable Hybrid Scheduling for High-Level Synthesis”. In: *Electronics* 14.23 (Jan. 2025). Publisher: Multidisciplinary Digital Publishing Institute, p. 4578. ISSN: 2079-9292. DOI: 10.3390/electronics14234578. URL: https://www.mdpi.com/2079-9292/14/23/4578 (visited on 12/19/2025).

[22] *ZedBoard Hardware User’s Guide*. Version 2.2. Avnet. 2014. URL: https://files.digilent.com/resources/programmable-logic/zedboard/ZedBoard_HW_UG_v2_2.pdf.

[23] *Zynq-7000 SoC Data Sheet: Overview*. DS190. v1.11.1. AMD. 2018. URL: https://docs.amd.com/v/u/en-US/ds190-Zynq-7000-Overview.

Appendix A

Conversion Process Example Sources: mac_4

This section contains the sources to the files involved throughout the process of specialising the input `mac` function for n=4, resulting in the output `mac_4` function. These are included to better illustrate how each of the formats defined formthe purpose of the dissertation work are structured.

For Figure 3.3, it should be noted that the way edges are connected to opcode nodes' inputs and outputs is technically not part of the DOT language's syntax, therefore the graphical representation does not accurately point these to their respective inputs.

Listing A.1: Original mac function source and header

```
// --- mac.h ---

#ifndef MAC_H
#define MAC_H

float mac(float *a, float *b, int n);

#endif

// --- mac.c ---

float mac(float *a, float *b, int n)
{
    float acc = 0;
    for (int i = 0; i < n; i++)
    {
        acc += a[i] * b[i];
    }
    return acc;
}
```

Listing A.2: mac function source with dotgen instrumentation

```
#include "stdio.h"

#include "dotgen.h"
#include "opmacros.h"

float mac(float *a, float *b, int n)
{
        char _buf[16][256];
        char *_ptr[16];
        DOT_start();

        _ptr[0] = DOT_addvar("0.c.f32");
        _ptr[1] = DOT_addvar("acc.f32");

        _ptr[2] = OP_MOV_F32("b0_0");

        DOT_vtoo(_ptr[0], _ptr[2], 0);
        DOT_otov(_ptr[2], _ptr[1], 0);

        DOT_vtov("start", _ptr[0]);

        _ptr[15] = _ptr[1];
    float acc = 0;

    for (int i = 0; i < n; i++)
    {
            sprintf(_buf[0], "%d.c.f32", i);
            sprintf(_buf[1], "b1_%d_0", i);
            sprintf(_buf[2], "b1_%d_1", i);
            sprintf(_buf[3], "b1_%d_2", i);
            sprintf(_buf[4], "b1_%d_3", i);
            sprintf(_buf[5], "b1_%d_4", i);

            _ptr[0] = DOT_addvar("a.ptr.f32");
            _ptr[1] = DOT_addvar(_buf[0]);
            _ptr[2] = DOT_addvar("b.ptr.f32");
            _ptr[7] = DOT_addvar("acc.f32");
            _ptr[9] = DOT_addvar("acc.f32");

            _ptr[3] = OP_LDARRAY_PTR_F32(_buf[1]);
            _ptr[4] = OP_LDARRAY_PTR_F32(_buf[2]);
            _ptr[5] = OP_MUL_F32(_buf[3]);
            _ptr[8] = OP_ADD_F32(_buf[4]);

            DOT_vtoo(_ptr[0], _ptr[3], 0);
            DOT_vtoo(_ptr[1], _ptr[3], 1);
            DOT_vtoo(_ptr[2], _ptr[4], 0);
```

```
48              DOT_vtoo(_ptr[1], _ptr[4], 1);
49              DOT_otoo(_ptr[3], _ptr[5], 0, 0);
50              DOT_otoo(_ptr[4], _ptr[5], 0, 1);
51              DOT_otoo(_ptr[5], _ptr[8], 0, 0);
52              DOT_vtoo(_ptr[7], _ptr[8], 1);
53              DOT_otov(_ptr[8], _ptr[9], 0);
54
55              DOT_vtov("start", _ptr[0]);
56              DOT_vtov("start", _ptr[1]);
57              DOT_vtov("start", _ptr[2]);
58              DOT_vtov(_ptr[15], _ptr[7]);
59
60              _ptr[15] = _ptr[9];
61          acc += a[i] * b[i];
62      }
63
64      _ptr[0] = DOT_addvar("acc.f32");
65
66      _ptr[1] = OP_RET_F32("b2_0");
67
68      DOT_vtoo(_ptr[0], _ptr[1], 0);
69
70      DOT_vtov(_ptr[15], _ptr[0]);
71
72      DOT_end();
73      return acc;
74  }
```

Listing A.3: DOT formatted IR CDFG generated by dotgen (indented and commented for clarity)

```
1   digraph G {
2
3       # Special Nodes
4
5       "start" [shape=Mdiamond];
6       "end" [shape=Msquare];
7
8       # Block 0 (before loop)
9
10      "0.c.f32.0";
11      "acc.f32.0";
12      b0_0 [
13          shape=record,
14          label="{ i0.f32 | mov.f32 | o0.f32 }"
15      ];
16      "0.c.f32.0" -> b0_0:"i0.f32";
17      b0_0:"o0.f32" -> "acc.f32.0";
```

```
    "start" -> "0.c.f32.0";

    # Block 1 Iteration 0 (loop)

    "a.ptr.f32.0";
    "0.c.f32.1";
    "b.ptr.f32.0";
    "acc.f32.1";
    "acc.f32.2";
    b1_0_0 [
        shape=record,
        label="{ { i0.ptr.f32 | i1.i32} | ldarray.f32 | o0.ptr.i32 }"
    ];
    b1_0_1 [
        shape=record,
        label="{ { i0.ptr.f32 | i1.i32} | ldarray.f32 | o0.ptr.i32 }"
    ];
    b1_0_2 [
        shape=record,
        label="{ { i0.f32 | i1.f32} | mul.f32 | o0.f32 }"
    ];
    b1_0_3 [
        shape=record,
        label="{ { i0.f32 | i1.f32} | add.f32 | o0.f32 }"
    ];
    "a.ptr.f32.0" -> b1_0_0:"i0.ptr.f32";
    "0.c.f32.1" -> b1_0_0:"i1.i32";
    "b.ptr.f32.0" -> b1_0_1:"i0.ptr.f32";
    "0.c.f32.1" -> b1_0_1:"i1.i32";
    b1_0_0:"o0.ptr.i32" -> b1_0_2:"i0.f32";
    b1_0_1:"o0.ptr.i32" -> b1_0_2:"i1.f32";
    b1_0_2:"o0.f32" -> b1_0_3:"i0.f32";
    "acc.f32.1" -> b1_0_3:"i1.f32";
    b1_0_3:"o0.f32" -> "acc.f32.2";
    "start" -> "a.ptr.f32.0";
    "start" -> "0.c.f32.1";
    "start" -> "b.ptr.f32.0";
    "acc.f32.0" -> "acc.f32.1";

    # Block 1 Iteration 1

    "a.ptr.f32.1";
    "1.c.f32.0";
    "b.ptr.f32.1";
    "acc.f32.3";
    "acc.f32.4";
    b1_1_0 [
        shape=record,
        label="{ { i0.ptr.f32 | i1.i32} | ldarray.f32 | o0.ptr.i32 }"
```

```
    ];
    b1_1_1 [
        shape=record,
        label="{ { i0.ptr.f32 | i1.i32} | ldarray.f32 | o0.ptr.i32 }"
    ];
    b1_1_2 [
        shape=record,
        label="{ { i0.f32 | i1.f32} | mul.f32 | o0.f32 }"
    ];
    b1_1_3 [
        shape=record,
        label="{ { i0.f32 | i1.f32} | add.f32 | o0.f32 }"
    ];
    "a.ptr.f32.1" -> b1_1_0:"i0.ptr.f32";
    "1.c.f32.0" -> b1_1_0:"i1.i32";
    "b.ptr.f32.1" -> b1_1_1:"i0.ptr.f32";
    "1.c.f32.0" -> b1_1_1:"i1.i32";
    b1_1_0:"o0.ptr.i32" -> b1_1_2:"i0.f32";
    b1_1_1:"o0.ptr.i32" -> b1_1_2:"i1.f32";
    b1_1_2:"o0.f32" -> b1_1_3:"i0.f32";
    "acc.f32.3" -> b1_1_3:"i1.f32";
    b1_1_3:"o0.f32" -> "acc.f32.4";
    "start" -> "a.ptr.f32.1";
    "start" -> "1.c.f32.0";
    "start" -> "b.ptr.f32.1";
    "acc.f32.2" -> "acc.f32.3";

    # Block 1 Iteration 2

    "a.ptr.f32.2";
    "2.c.f32.0";
    "b.ptr.f32.2";
    "acc.f32.5";
    "acc.f32.6";
    b1_2_0 [
        shape=record,
        label="{ { i0.ptr.f32 | i1.i32} | ldarray.f32 | o0.ptr.i32 }"
    ];
    b1_2_1 [
        shape=record,
        label="{ { i0.ptr.f32 | i1.i32} | ldarray.f32 | o0.ptr.i32 }"
    ];
    b1_2_2 [
        shape=record,
        label="{ { i0.f32 | i1.f32} | mul.f32 | o0.f32 }"
    ];
    b1_2_3 [
        shape=record,
        label="{ { i0.f32 | i1.f32} | add.f32 | o0.f32 }"
```

```
    ];
    "a.ptr.f32.2" -> b1_2_0:"i0.ptr.f32";
    "2.c.f32.0" -> b1_2_0:"i1.i32";
    "b.ptr.f32.2" -> b1_2_1:"i0.ptr.f32";
    "2.c.f32.0" -> b1_2_1:"i1.i32";
    b1_2_0:"o0.ptr.i32" -> b1_2_2:"i0.f32";
    b1_2_1:"o0.ptr.i32" -> b1_2_2:"i1.f32";
    b1_2_2:"o0.f32" -> b1_2_3:"i0.f32";
    "acc.f32.5" -> b1_2_3:"i1.f32";
    b1_2_3:"o0.f32" -> "acc.f32.6";
    "start" -> "a.ptr.f32.2";
    "start" -> "2.c.f32.0";
    "start" -> "b.ptr.f32.2";
    "acc.f32.4" -> "acc.f32.5";

    # Block 1 Iteration 3

    "a.ptr.f32.3";
    "3.c.f32.0";
    "b.ptr.f32.3";
    "acc.f32.7";
    "acc.f32.8";
    b1_3_0 [
        shape=record,
        label="{ { i0.ptr.f32 | i1.i32} | ldarray.f32 | o0.ptr.i32 }"
    ];
    b1_3_1 [
        shape=record,
        label="{ { i0.ptr.f32 | i1.i32} | ldarray.f32 | o0.ptr.i32 }"
    ];
    b1_3_2 [
        shape=record,
        label="{ { i0.f32 | i1.f32} | mul.f32 | o0.f32 }"
    ];
    b1_3_3 [
        shape=record,
        label="{ { i0.f32 | i1.f32} | add.f32 | o0.f32 }"
    ];
    "a.ptr.f32.3" -> b1_3_0:"i0.ptr.f32";
    "3.c.f32.0" -> b1_3_0:"i1.i32";
    "b.ptr.f32.3" -> b1_3_1:"i0.ptr.f32";
    "3.c.f32.0" -> b1_3_1:"i1.i32";
    b1_3_0:"o0.ptr.i32" -> b1_3_2:"i0.f32";
    b1_3_1:"o0.ptr.i32" -> b1_3_2:"i1.f32";
    b1_3_2:"o0.f32" -> b1_3_3:"i0.f32";
    "acc.f32.7" -> b1_3_3:"i1.f32";
    b1_3_3:"o0.f32" -> "acc.f32.8";
    "start" -> "a.ptr.f32.3";
    "start" -> "3.c.f32.0";
```

```
    "start" -> "b.ptr.f32.3";
    "acc.f32.6" -> "acc.f32.7";

    # Block 2 (after loop)

    "acc.f32.9";
    b2_0 [
        shape=record,
        label="{ i0.f32 | ret.f32 }"
    ];
    b2_0 -> "end";
    "acc.f32.9" -> b2_0:"i0.f32";
    "acc.f32.8" -> "acc.f32.9";
}
```

Listing A.4: JSON formatted mac_4 function metadata file passed to dot2c

```
{
    "name": "mac_4",
    "variables": [
        {
            "name": "a",
            "scope": "fn_parameter",
            "type": "ptr.f32"
        },
        {
            "name": "b",
            "scope": "fn_parameter",
            "type": "ptr.f32"
        },
        {
            "name": "acc",
            "scope": "local",
            "type": "f32"
        }
    ],
    "return_type": "f32"
}
```

Listing A.5: Final specialised mac_4 function source and header generated by dot2c

```
// --- mac_4.h ---

#ifndef TEST_TGT_MAC_4_C
#define TEST_TGT_MAC_4_C
```

```
5
6  float mac_4(float * a, float * b);
7
8  #endif
9
10 // --- mac_4.c ---
11
12 float mac_4(float * a, float * b)
13 {
14   float acc;
15
16   acc = 0;
17   acc = (float)(*(&(a[(int)(0)]))) * (float)(*(&(b[(int)(0)]))) + acc;
18   acc = (float)(*(&(a[(int)(1)]))) * (float)(*(&(b[(int)(1)]))) + acc;
19   acc = (float)(*(&(a[(int)(2)]))) * (float)(*(&(b[(int)(2)]))) + acc;
20   acc = (float)(*(&(a[(int)(3)]))) * (float)(*(&(b[(int)(3)]))) + acc;
21   return acc;
22 }
```

Listing A.6: JSON test parameter description source for mac_4

```
1  {
2      "inputs":
3      [
4          {
5              "name": "a",
6              "type": "float",
7              "array_size": "4",
8              "definition": [ "1.172981656", "2.363468844", "3.619922019", "
                   0.324149589" ]
9          },
10         {
11             "name": "b",
12             "type": "float",
13             "array_size": "4",
14             "definition": [ "11.210489158", "10.489829568", "10.236639576", "
                   2.986973288" ]
15         }
16     ],
17     "reference":
18     {
19         "name": "mac",
20         "file": "ref/mac.h",
21         "args": ["a", "b", "4"]
22     },
23     "target":
24     {
```

```
        "name": "mac_4",
        "file": "tgt/mac_4.h",
        "args": ["a", "b"]
    },
    "output_type": "float"
}
```