Universidade Estadual de Campinas
Instituto de Computação

Pedro Ferrazoli Ciambra

IaRa: Dataflow Programming with the MLIR Framework

IaRa: Programação Dataflow com a ferramenta MLIR

CAMPINAS
2022

Pedro Ferrazoli Ciambra

IaRa: Dataflow Programming with the MLIR Framework

IaRa: Programação Dataflow com a ferramenta MLIR

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Hervé Cédric Yviquel

Este exemplar corresponde à versão final da Dissertação defendida por Pedro Ferrazoli Ciambra e orientada pelo Prof. Dr. Hervé Cédric Yviquel.

CAMPINAS
2022

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

C48i

Ciambra, Pedro Ferrazoli, 1993-
IaRa : dataflow programming with the MLIR framework / Pedro Ferrazoli Ciambra. – Campinas, SP : [s.n.], 2022.

Orientador: Hervé Cédric Yviquel.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Compiladores (Computadores). 2. Fluxo de dados (Computadores). I. Yviquel, Hervé Cédric. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações Complementares

Título em outro idioma: IaRa : programação dataflow com a ferramenta MLIR
Palavras-chave em inglês:
Compilers (Electronic computers)
Data flow computing
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Hervé Cédric Yviquel [Orientador]
Karol Desnos
Lucas Francisco Wanner
Data de defesa: 19-08-2022
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-5445-9752
- Currículo Lattes do autor: http://lattes.cnpq.br/5182140742412094

Universidade Estadual de Campinas
Instituto de Computação

Pedro Ferrazoli Ciambra

IaRa: Dataflow Programming with the MLIR Framework

IaRa: Programação Dataflow com a ferramenta MLIR

Banca Examinadora:

- Prof. Dr. Hervé Cédric Yviquel
  IC/UNICAMP
- Prof. Dr. Karol Desnos
  INSA Rennes
- Prof. Dr. Lucas Francisco Wanner
  IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 19 de agosto de 2022

Acknowledgements

To CNPq, for funding[1] my studies.

To the staff and professors of the Institute of Computing, for their knowledge and support.

To my parents and brother, for bearing with me during the pandemic.

To my labmates, for their advice and camaraderie.

To Mickaël and Maxime, for their invaluable help and insight.

And to Hervé, for his infinite patience.

[1]This study was financed in part by The Brazilian National Council for Scientific and Technological Development (CNPq), grant #133933/2020-2

Resumo

Dataflow (fluxo de dados) é uma família de modelos de computação que oferece uma forma intuitiva de programação paralela. Apesar de ser considerado uma boa solução para a implementação de programas paralelos, particularmente na área de processamento de sinais digitais, ele ainda não é muito difundido em outras áreas de sistemas de computação. Este projeto almeja aproximar a pesquisa em *dataflow* do estado da arte da área de Compiladores através do uso do MLIR, um alicerce de desenvolvimento de compiladores que permite a criação de representações intermediárias modulares e intercompatíveis, na implementação de um compilador *dataflow*. A implementação atual suporta o escalonamento de grafos SDF para dispositivos single-core. Ela contém um *parser* para uma uma linguagem de descrição de grafos baseada em DIF; um novo dialeto de representação intermediária para grafos *dataflow* baseado em MLIR; e um alicerce para o desenvolvimento de cooptimizações grafo/ator, incluindo uma nova técnica de cooptimização que permite eliminação de código morto pré-escalonamento. O compilador foi testado com múltiplos algoritmos reais e demostra ganhos promissores em desempenho e uso de memória. O uso do MLIR também abre diversas oportunidades de pesquisa, já que ele tem relação com muitos projetos em computação paralela.

Abstract

Dataflow is a family of models of computation that allow for intuitive parallelism from design time. Despite being considered a good solution for the implementation of parallel programs, particularly in the field of digital signal processing, it has not yet met widespread adoption. This work seeks to bring dataflow research closer to the current state of the art in compiler technologies by using MLIR, a framework focused on the creation and manipulation of intercompatible intermediate representations, in the implementation of a dataflow compiler. The current implementation provides support for the scheduling of Synchronous Dataflow graphs for single-core targets. It contains a parser for a DIF-based graph specification language, a custom MLIR-based IR dialect for the internal representation of dataflow graphs, and a framework for the development of graph/kernel cooptimizations, including a novel co-optimization technique that enables ahead-of-schedule dead code elimination. It was tested with several real-world algorithms, showing promising results in performance and memory usage. The use of MLIR also paves the way for many research opportunities, as it is related to several parallel computing projects.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Nowadays, parallelization is the most effective way to scale the performance of computing systems. With the slowing down of improvements to processor speeds, the only way to accelerate computation is to split it up spatially among multiple devices (vectorization units, processor cores, accelerator cards, external computers in a network). However, this is a delicate and error prone task: manual parallel programming requires considerable specific knowledge and depends heavily on the application and hardware in question. Each of the common venues for parallelization has their own gotchas and pitfalls: vectorization instructions and acceleration devices are limited by bus and memory bandwidths, multi-threading and tasking solutions need to deal with deadlocks and data races, and distributed systems must account for lossy, high-latency or unreliable networks. All of these have their own specific, incompatible programming frameworks, and have differing levels of applicability for any given workload.

A programming paradigm that addresses these issues is the dataflow network. Unlike other strategies such as tasking, dataflow networks rely on rigorous mathematical descriptions of the transformations applied to the data, providing guarantees that facilitate the automatic identification of data and task parallelism. This means that, once described as a dataflow network, a program can be easily and optimally compiled for a wide variety of targets and architectures, including not only classic von Neumann architectures, but also distributed systems and acceleration hardware such as GPUs and TPUs. However, the currently available dataflow frameworks are numerous and do not offer much in terms of interoperability, having generally been designed for domain-specific purposes such as Digital Signal Processing (DSP). As there is no universally recognized industry standard for dataflow programming yet, there is still significant resistance to its adoption.

In this work, we address these adoption problems by creating a complete dataflow compilation flow that is integrated with a mainstream compiler framework (LLVM). In particular, we adopt the recently released MLIR, a toolset for the creation of modular intermediate representations that allows for powerful manipulation of abstractions inside of a compiler. MLIR's focus on compatibility and reuse makes it especially suited for our use-case, as it enables the versatility of dataflow networks to be used in many different contexts; it is also modern and actively adopted by major players in the current parallel computing landscape, particularly in machine learning and hardware acceleration contexts, despite being relatively recent.

The contributions consist of:

- A parser for a DIF-based SDF graph specification language;
- An MLIR dialect for dataflow that contains all abstractions required for the specification of SDF graphs;
- A compilation flow that generates a single-core SDF schedule;
- A co-optimization pass that performs dead code elimination on unused outputs of kernel functions specified in C;
- Experimental analysis of the performance and memory usage characteristics of the generated executables in example applications;
- A comparison with the results of the state-of-the-art in Integrated Development Environments (IDE) for static dataflow.

We chose DIF [29] as a basis for our input language for its simplicity and relative wide adoption among dataflow projects. We focus on the SDF model of computation as a first step for the fact that its performance benefits rely on static analysis, which MLIR is well equipped to tackle, and the availability of reference works to compare against, by projects such as PREESM [46].

As the compiler is designed as a proof-of-concept and starting point for future extension, it currently provides only a single-core scheduler; the development of multi-core scheduling strategies is a complex subject, and as such we prioritized the development of other key parts of the pipeline, such as the co-optimization infrastructure.

We have co-authored an article [17] that covers the dead code elimination co-optimization, already accepted and pending publication in the SiPS 2022 conference proceedings.

A brief summary of this dissertation's chapters follow. In Chapter 2, we cover the field of Dataflow, its main abstractions and state of the art. In Chapter 3, we cover the characteristics of MLIR and its place in the current research environment. In Chapter 4, our main contributions, collectively named *IaRa*, are covered in detail. In Chapter 5, we show experimental results. Chapter 6 concludes this dissertation.

Chapter 2

Dataflow Programming

Dataflow is a family of Models of Computation (MoCs) that addresses the demand for abstract representation of parallel programs, being related to the similar *stream processing* and *reactive programming* paradigms. Being agnostic to the underlying parallelization mechanism, it is applicable in a broad range of situations; nowadays, it has found application in the fields of hardware design and machine learning. The concept of representing data transformation as a flow graph is considerably old; it was a popular topic in hardware research in the 70's and 80's, and has existed as a programming paradigm as far back as 1960 [31]. Since dataflow is well-suited for graphical representation, it was also a popular subject for visual programming research [53].

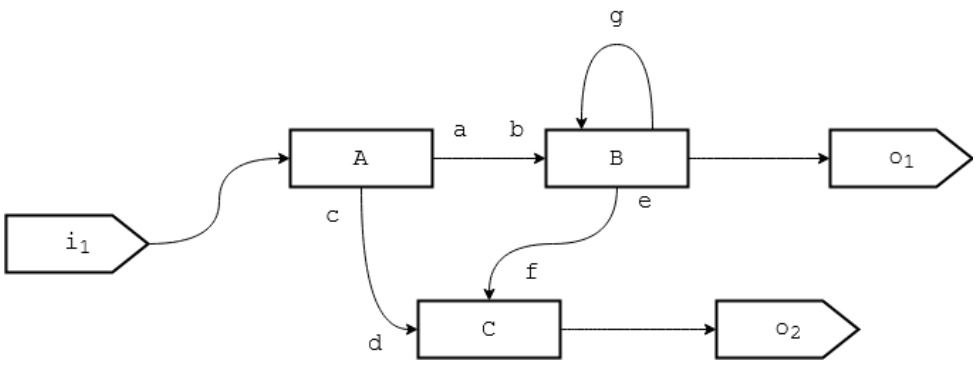

Figure 2.1: An example of dataflow network (specifically, a SDF graph, with fixed data rates on each port)

Dataflow models represent algorithms as directed graphs, as shown in Fig. 2.1, where nodes ("actors") represent processing steps and edges represent queues (first-in-first-out data structures, or FIFOs) that transport data between each processing step. Each actor represents an operation that can be activated ("fired") to do work; each firing consumes a quantity of values ("tokens") from its incoming edges and produces output tokens into its outgoing edges. These actions are regarded as black-boxes, and treated as indivisible units of work. By scheduling these actors to fire in a suitable order, the inputs flow through the graph from edge to edge as they are transformed, reaching the sink nodes as the final output of the algorithm. As this representation naturally encodes the data dependencies between each part of the algorithm, the task of selecting parallelizable regions becomes

simple enough that it can be done automatically, either by a dynamic runtime or statically at compilation time.

2.1 Models of Computation

There are different levels of generality when implementing dataflow systems. Each successive level increases the expressiveness and flexibility of the system, but sacrifices some property that could be taken advantage of for compile-time analysis. These levels have been formalized as Models of Computation (MoCs), and they relate to each other as shown by Fig. 2.2.

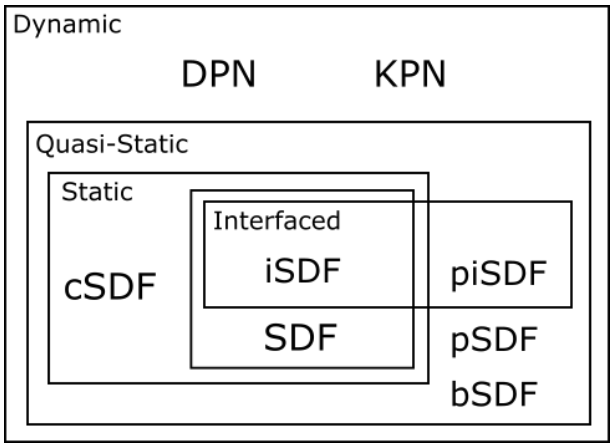

Figure 2.2: An Euler diagram showing the capabilities of each MoC; outer sections are more expressive but require more resource-intensive runtimes.

2.1.1 Static MoCs

Synchronous Dataflow is the most constrained of the Models of Computation. By requiring the number of tokens an actor consumes and produces at each port to be the same for every activation, the behavior of the system becomes predictable and an efficient schedule is deducible at compile-time. Such schedules have the advantage of not having to rely on overhead-heavy dynamic runtimes, which can be valuable in resource-constrained or real-time applications.

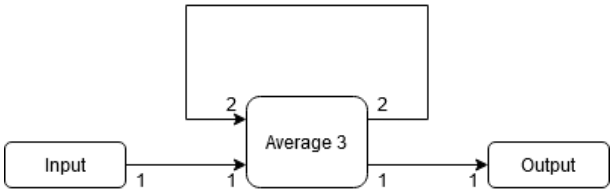

Figure 2.3: An example SDF graph implementing a low-pass filter. The numbers denote the amount of tokens consumed/produced by each edge at each firing.

Figure 2.3 shows a simple example of SDF graph that implements a smoothing filter. On each firing, the *Average 3* actor calculates the mean of the three received values, sends

the average to the output node, and sends the two most recently produced values back to itself, to be used in the next firing. These values are stored in the FIFO queue associated with the self-loop, which in this case needs to have space for at least 2 values. FIFOs that form loops must be initialized with values ("delays"), so that dependent nodes can fire the first time.

Scheduling SDF networks can be done by working with *topology matrices*, which describe the relationship between nodes and edges by encoding their consumption/production rates as positive and negative elements. For example, the network in Figure 2.1 would have a topology matrix of the following form.

$$\Gamma = \begin{bmatrix} a & -b & 0 \\ c & 0 & -d \\ 0 & e & -f \\ 0 & 0 & 0 \end{bmatrix}$$

Each column represents a node and each row represents an edge. If a node produces values into an edge, the corresponding element is positive; conversely, if a node consumes values from an edge, the element is negative. A self-loop such as the one on node B should consume and produce the same amount of tokens each firing (otherwise, the buffer would overflow or underflow), and so the corresponding row cancels out and equals zero.

Lee and Messerschmitt [35] describe a method to determine if a periodic schedule is possible, by analyzing the topology matrix, and provide an algorithm to efficiently generate both single- and multi-threaded schedules. There is generally a trade-off between period length and edge buffer size (referred to as *throughput-buffering trade-off*), as longer periods allow for more size-efficient routing when the data rates on each edge don't match nicely.

It should be noted that this algorithm does not provide an *optimal* schedule, only an approximated one, with the constraint of being periodic. The calculation of the maximum throughput for a given dataflow graph is a theoretically difficult problem, even for highly constrained MoCs such as SDF, and algorithms that find an optimal scheduling in the general case have exponential complexity [25] [52]. However, the approximations generated by this algorithm are often good enough, and competitive with hand-made parallel schedules. This model has been successful for digital signal processing (DSP) applications, as many common protocols and algorithms have fixed packet sizes that are data-independent.

Some works define further constraints to SDF. Single-rate SDF (srSDF) includes the constraint that all edges must have matching input and output rates; this means that all nodes in the network fire at the same rate, on average [11]. Homogenous SDF (hSDF) is a further constraint where the rates of all ports in the network are equal to one, which means that a single-threaded schedule consists of simply executing the actors in topological order. These variants can generally be converted between each other, with exponential size trade-offs.

Cyclo-static dataflow (cSDF) is a generalization of SDF that allows for actors to periodically change the amount of processed tokens per activation, as long as this change is

cyclic and predictable. This additional capacity is useful, for instance, for protocols that employ some sort of serial, known-length transmission; the behavior of the actor when processing headers/footers may be different to the behavior when processing the payload.

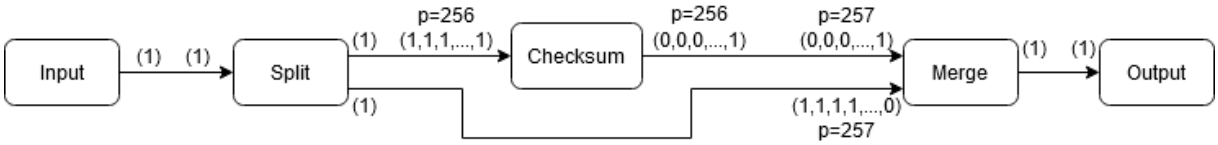

Figure 2.4: An example cSDF graph that appends a checksum after each string of 256 values. The p values denote the length of the rate lists.

In the example of Figure 2.4, each actor input and output is labeled with a list of rates. Each sequential firing of the actor consumes a number of tokens given by the values in the list, starting over from the beginning when the end is reached. Here, for every 256 input tokens, the *Checksum* actor produces a single token. The *Merge* node first passes along the 256 values unchanged, and then the checksum token. It is always possible to transform such a model into a single-rate equivalent, at the cost of higher node counts and queue sizes. A method that does not require such a reduction is described by Bodin et al [12].

2.1.2 Quasi-Static Dataflow

Static MoCs are limited to applications where the input data is homogeneous and predictable. To work around this, extensions to SDF and cSDF have been developed that break the pure static schedulability to permit occasional data-dependent tuning of each actor's data rates. This is especially useful in cases where the algorithm is *mostly* synchronous, except for some initial or final configuration phase.

Boolean Dataflow (bDF) allows for special control edges in the graph representation that can switch the behavior of connected actors. It has been proven to be Turing-complete [14], but it is not very practical to use in real life applications, as the design process can be complex and unintuitive.

Parameterized SDF (pSDF) is a generalization of bDF models that allows the control edges to have arbitrary numeric values. To be able to maintain efficiency at runtime, some knowledge of which information can change and at which times must be provided by the programmer; this information takes the form of special *parameter* edges in the graph representation of the algorithm, which do not correspond to FIFO queues but to "control channels" that are only updated sporadically, at known points of the execution. These channels are used to reconfigure the scheduler at runtime, but they otherwise remain constant and allow the schedule to execute at full efficiency.

An example would be an algorithm that can apply a transformation on an image, without knowing the size of the image *a priori*; this information is provided at the start of the process and used to automatically reconfigure the actors and queue sizes for the new image.

This MoC has been explored by a number of works, including Bhattacharya et al. [8] and Desnos et al. [20].

Parameterized/Interfaced SDF (piSDF or πSDF) is an extension to pSDF that includes a hierarchical structure to the graph; instead of representing actors as simple atomic executions of a black-box procedure, this MoC defines a sub-graph hierarchy that enables efficient design reuse. This allows the definition of interfaces between hierarchy levels, including the types and rates of inputs, outputs and parameters. This is described by Desnos et al. [20] as an instance of PiMM, a meta-model that could apply the same hierarchical structure to other MoCs.

2.1.3 Dynamic Dataflow

Dataflow Process Networks (DPN), also known as Dynamic Dataflow [36], is a more general model that allows for completely data-dependent execution. As such, static scheduling is impossible, and therefore this model depends completely on a runtime to allocate the firing of each actor.

2.2 Dataflow Programming Languages

The following is a selection of previous works that aim to represent dataflow graphs in textual form, with differing levels of expressiveness and different supported MoCs.

LUSTRE [27] is a programming language for SDF that emphasizes synchronous signals and state transitions. Each value is modelled as a signal with a defined value at each point in discrete time, and the relationships between signals are declaratively defined as functions of the values at previous times. It includes a robust static checker that accounts for the unique mathematical properties of the system (referred to as *clock calculus*). Since 1993, it has found use in the industry as the software base for critical embedded systems [7], such as aircraft and nuclear power plants. The project is still being worked on academically, by the Verimag Laboratory of Université Grenoble Alpes [30].

SIGNAL [37] is a similar language that allows description of systems with multiple clocks, as opposed to the global clock of LUSTRE. It also defines signals by their relations to one another, but includes a more formal proof system to derive the implementations from the user-defined relationships. It was also designed to work with a block-diagram based graphical user interface, a relative novelty at the time of publication. It is still being worked on, as part of the Polychrony toolset [34].

CAL [22] is a dataflow language created as part of the Ptolemy project. It supports the DPN model, and is intended to be used in a wide array of applications, from heterogenous network processing to DSP.

RVC-CAL is a subset of CAL defined by the Moving Picture Experts Group (MPEG) Reconfigurable Video Coding (RVC) standard [10]. It features a more strict type system and more restricted constructs than CAL, in order to facilitate hardware and software generation. It was developed in order to provide a better medium to represent reference

implementations of media encoders and decoders, which had been traditionally represented in the C language and similar.

DIF [29], or Dataflow Interchange Format, is a text-based language intended to be used to encode dataflow network specifications, especially ones designed by graphical tools. It is not geared towards any specific MoC, being compatible with SDF, cSDF, boolean and parameterized dataflow, as well as capturing network hierarchy ("interfaced" dataflow). It also provides an unconstrained specification, that could in principle be used for dynamic dataflow MoCs such as DPN. It is intended to be easily generated and parsed by tools, and also to be easily read and written by humans. The DIF project also includes Java tools for working with the DIF language, including conversion between compatible MoCs and some implementations of established algorithms.

HoCL [49], the Higher-Order dataflow Coordination Language, is a dataflow language that supports advanced features such as polymorphic type inference. It supports many different SoCs and comes with several backends, including DIF, Preesm, SystemC and CAL. It is a successor to the CAPH [50] project, but it currently has no official academic publication of its own yet.

2.3 Dataflow Runtimes and Compilers

As dataflow is applicable in a number of contexts, several compilers and frameworks have been developed that support is. A non-exhaustive of projects relevant to the state-of-the-art follows.

The Ptolemy project [23] consists of studies in heterogenous computing, including, but not limited to, several dataflow MoCs. It is an attempt to supply an environment for the entire design and operation processes, and is built to be compatible with several different tools, systems, programming languages, frameworks, and MoCs. It sought to allow interconnection between different systems using object-oriented concepts, which were very popular at the time of the project's creation. The project includes a SDF scheduler and a DPN runtime; furthermore, it allows mixing the two (and other, non-dataflow MoCs) when designing a system, so that the pros and cons of each can be balanced as required by the application. The current iteration of the project consists of the Java-based Ptolemy II, which is being actively developed and is used in production by a number of projects, such as Kepler [2], a scientific computing framework.

Grape-II [33] consists of a rapid prototyping toolchain for DSP systems, intended to provide effortless estimation of the cost and resource usage of the design. It supports both HDL synthesis and code generation for DSP systems, and also includes a graphical editor, where the user can connect heterogenous components (such as DSP processors, FPGAs, etc.). The behavior of the prototypes can be simulated using the included SDF and cSDF schedulers.

Preesm [46] is a dataflow framework geared for rapid prototyping of DSP programs, particularly Texas Intrument's multicore devices. It includes an Eclipse-based graphical interface and optimizing compiler that generates executables starting from user-defined

piSDF actor implementations, provided in C/C++. It is bundled with Spider [28], a runtime for the piSDF MoC that is intended for the development of parallel programs on DSP processors. It is presented as a direct alternative to OpenMP, a parallelization library for the C, C++ and Fortran programming languages.

Open Dataflow [9], is a dataflow runtime and compiler for dynamic dataflow based on the RVC-CAL language. Besides offering an environment for the execution of DPN, it offers three compilation backends, generating HDL, SystemC-compatible code, and embedded C, respectively. Development on this project has gradually diminished in favor of the Orcc project, and it is currently unmaintained.

Orcc [55], the Open RVC-CAL Compiler (Orcc) is an open-source IDE for dataflow programming. It includes an Eclipse-based graphical editor for dataflow networks and a multi-target compiler that supports both software and hardware languages. It is also designed to support the use of the RVC-CAL language. The project supports a variety of targets, including general purpose processors, embed processors, HDL, LLVM, C code generation, etc. It also includes a built-in simulator that allows for easy validation of network designs.

StreamIt [54] is a programming language and compiler framework designed for modern streaming applications that defines a model very similar to a constrained version of SDF. Being designed with user productivity and industry readiness in mind, not much effort is made to relate the concepts used in the model with the formal definitions present in the literature; however, the project includes concepts similar to piSDF's reconfigurable actors and DPN's asynchronous messaging. It also defines novel concepts such as "information wavefronts" that are used in latency calculations.

The MAPS project [15] provides a compiler infrastructure for heterogenous multiprocessing SoCs. It introduces CPN (C for Process Networks), a small extension of the C language that provides the relevant dataflow abstractions, and a compiler extension for the Clang compiler that supports it.

Our compiler, IaRa, overlaps with Preesm in functionality, in that it facilitates the generation of performant executables starting from kernel code in C and a dataflow graph description. In Chapter 5, we compare the performance and memory usage of executables generated by Preesm and IaRa.

The next chapter covers MLIR, the framework upon which we've chosen to build IaRa. It is part of LLVM, a compiler framework that is widely adopted in the industry; we expect that this will be a differential that will help set IaRa apart from the works described in this chapter.

Chapter 3

The MLIR framework

The problem of optimizing code for parallel execution has been tackled multiple times and from many different directions over the years. Projects such as OpenMP [18], for instance, allow you to generate parallel code by annotating sequential C or C++ code with special markers that indicate parallelizable regions. Other projects have developed their own solutions, such as the built-in coprocedures of Go [47] and the safety-oriented thread management of Rust [38]. Similar variety can be seen in the hardware accelerator camp, with projects such as the Nvidia CUDA framework [41] for general computing on graphic cards being followed by AMD ROCm [3], Apple Metal [4] and the OpenCL standard of the Khronos group [51]. As it can be imagined, many of these projects employ common solutions that have been reimplemented at each instance, with differing levels of interoperability. This poses an issue, as this makes it very difficult to port code between different languages and target hardware platforms while maintaining performance and ease of writing.

3.1 LLVM and its limitations

One of the most relevant projects in the current compiler landscape is LLVM [32], an industrial-grade compiler infrastructure and toolset that addresses this portability issue. One of its main features is the LLVM Intermediate Representation (IR). It consists of a human-readable language that has a level of abstraction between the C language and that of the assembly languages of common processor targets such as x86 and ARM, also providing intrinsics for extensions such as SIMD intructions. Higher-level languages that wish to use LLVM may simply emit LLVM IR and let the low level compiler handle the work of producing highly-optimized machine code for supported target architectures. It is relied on by the compilers of many general-purpose programming languages, including C/C++ (through the Clang compiler), Rust, Swift, Julia and others.

One limitation of LLVM, however, is that while these languages often make full use of the LLVM toolset, and therefore avoid the reimplementation of low-level parts of the compiler flow, there is still a great deal of higher-level compiler logic that is similar between different languages, which ends up being redone by each project from scratch. Some examples are type systems, lifetime analysis, macros, generics, and meta-programming.

At the same time, the popularization of machine learning created a demand for flexible compilers that could target a large variety of hardware accelerators, which LLVM IR was not designed to support. A number of competing IR standards for hardware-accelerated machine learning arose, such as TensorFlow [1], GLOW [48] and ONNX [6]. Each hardware vendor also provides their own parallel programming interface.

3.2 MLIR as a solution for intercompatibility

These demands create a many-to-many relationship between programming languages, compilation frameworks and target architectures that invariably leads to repeated work, incompatibilities and loss of generality in the conversions between different formats and layers of abstraction. Because each conversion step is completely incompatible with the next, any potential optimization that spans more than one format needs to deal with the implementation details of both formats, greatly multiplying the effort required.

To address this, the MLIR project was created; it allows for arbitrarily-defined domain-specific IRs to operate together with each other within the same environment, implementing all of the best practices in compiler design developed through the experience of the LLVM team. Originally part of the TensorFlow project, it was deemed generally useful enough to be accepted and merged into the LLVM project.

Instead of the assembly-language-inspired *instructions* of LLVM IR, MLIR code consists of *operations*. Unlike instructions, operations are not restricted to single physical actions in a processor; they can represent any kind of abstract information, and can contain other operations in nested regions. This allows operations to represent complex concepts such as modules, functions, and structured control flow. Custom types and attributes can also be defined. To transform source code between different forms, *passes* are defined, within a powerful trait system that allows for automatic parallelization of the compilation flow.

Related operations, types, attributes and passes are grouped into namespaces, termed *dialects*. Developers are free to select which dialects are relevant and allowed to interact with their operations and passes, and to define and share their own dialects. MLIR currently provides built-in dialects for control flow, memory management, arithmetic and tensor operations, function and scope management, parallelization, and other concepts that are useful for general-purpose compilers. One of these dialects maps directly into LLVM IR, and is commonly used as a target by projects that wish to leverage the existing framework around it.

MLIR code can carry and transmit as much abstract data as is convenient, as deep into the compilation pipeline as necessary. Take, for instance, the common case of transforming an abstract matrix multiplication into real code, by either transforming it into a nested loop to be executed in a CPU or by offloading it to an accelerator. Traditional pipelines provide only two possibilities: that the code be annotated ahead of time with vendor-specific syntax or function calls, requiring the developer to decide ahead of time how to compute the multiplication and limiting portability; or to write the nested loop normally and then, further down the compilation flow, rely on some pattern-matching tool to

recognize it as a matrix multiplication and replace it with the accelerator call. Since MLIR allows the operation to remain abstract, even as the surrounding code goes through the normal compilation process, this decision can be postponed and taken much later in the compilation flow.

This allows for improved separation of concerns, as the hardware optimization of the matrix multiplication itself can be deferred to each vendor, and the compiler's high-level part can focus on the semantics of the problem instead. Meanwhile, all of the tools used for the transformation and manipulation of the source code and dialects remain the same across all levels of the compilation, allowing for high-quality, easily maintainable and extendable compilers. And, as the user no longer depends on a specific vendor's toolset to express a generically parallel program, they are enabled to adopt a more expressive declarative programming style.

This is particularly useful for domain-specific languages, as each domain can define its own dialect and operations and defer the implementation of different levels of abstraction to different existing dialects. A popular example is the domain of neural networks; by using a dialect, the modeling of the neural network's architecture can be decoupled from the implementation of the computations themselves.

3.3 Standard dialects

MLIR encourages separation of concerns as often as possible. As such, each different set of common compiler abstractions supported out of the box is separated into a different dialect. As a group, these are referred to as "standard" MLIR. All of these dialects provide conversion passes into LLVM IR, and as such can be used to implement general purpose programming languages. A non-exhaustive list follows.

The Builtin dialect provides common types and attributes (such as strings, integers, dictionaries and source code locations), and is globally visible by all other dialects. It defines the `module` operation, a generic region that defines a symbol scope.

arith, **math** and **complex** provide numeric types and mathematical operations.

scf and **cf** provide control flow directives. **scf** provides structured control flow abstractions, such as if-else, for and while blocks that may yield values, and **cf** provides simple assembly-like labels, jumps and branches.

affine provides affine loops, affine maps and other structures that can benefit from polyhedral analysis.

bufferization and **memref** provide memory-related operations and passes.

tensor and **linalg** provide operations on data of arbitrary dimensionality.

vector and **x86vector** provide intrinsics for SIMD operations.

omp and **gpu** provide operations that define concurrency. **omp** matches OpenMP directives, while **gpu** codifies generic kernel launching for programming models such as CUDA and OpenCL.

emitc provides operations that allow the translation of standard MLIR into C++, created with machine learning models in mind.

3.4 Third-party projects

Despite being a relatively recent effort, a number of tools have already populated MLIR's ecosystem. A non-exhaustive list of other examples follows.

TensorFlow [1], a project backed by Google, is the current industry standard in neural network frameworks. It defines its neural networks as a graph of tensor transformations, sharing many similarities with dataflow MoCs. Since it is offered as a library, many language front-ends are available, Python being one of the most popular; as its usage is somewhat complex and boilerplate-heavy, a number of more user-friendly wrapper interfaces have been developed, such as Keras [16]. Before being merged into the LLVM project, MLIR was part of TensorFlow.

Open Neural Network Exchange (ONNX) [6] is an effort to provide an open standard for all existing neural network frameworks, including TensorFlow [1], Keras [16], JAX [13], GLOW [48], and PyTorch [44]. This makes it well aligned to MLIR's objective of improving compatibility between compiler projects; as such, ONNX provides a MLIR dialect that implements the ONNX standard and generates efficient implementations with minimal runtime.

Circuit IR Compilers and Tools (CIRCT) [24] is an experimental framework that uses MLIR to tackle the problem of hardware design. It provides integrations to hardware design tools such as Chisel [5] and Calyx [42], and hardware description languages such as VHDL and Verilog. It also provides schedulers and generators that automatically generate hardware from standard MLIR, which, besides being useful for prototyping, also provides a path for programs written in general-purpose languages to target Field Programmable Gate Arrays (FPGAs).

Flang [43] is a Fortran implementation, developed from scratch in modern C++ using MLIR as the basis for its high level IR.

Verona [39] is a research programming language project led by Microsoft that explores concurrent ownership.

Polygeist [40] is an advanced C compiler that specializes in polyhedral optimizations. It features a robust parser of the C language, and, as part of its front-end, converts it into standard MLIR with little loss of information.

3.5 Syntax

MLIR's textual representation adopts a syntax that is reminiscent of its LLVM IR origins. The sequential instructions in MLIR dialects are denominated "operations", and they follow the following format.

```
%ret_val:3 = "dialect_name.op_name"(%arg){attr = true}
    : !dialect_name<"custom_type"> loc(callsite("ctx" at "filename":30:1))
```

Simply put, a typical MLIR operation is composed of:

- the operation's name (which, for user-defined dialects, is prefixed with the dialect's name, so that the symbol can be reused between different dialects);
- a list of dynamically-known parameters, such as function arguments, that are references to the results of other operations;
- a list of statically-known attributes, which consist of arbitrary constant data. This could be used, for instance, to represent C++'s template arguments;
- a list of zero or more results (differently from LLVM IR, an operation may return more than one result);
- the operation's affine shape (the types and shapes of its parameters and results);
- the operation's source location. As the code is transformed, the location is automatically propagated; it can then be used when generating debug information. While it doesn't need to be printed when generating the textual representation of the IR, it must be provided when creating operations programmatically.

Additionally, an operation can contain *regions*, which themselves may contain *blocks*, as presented in Listing 3.1. Blocks are nested containers for sub-operations that can represent a structured hierarchy without requiring scope markers. These can be used to represent, for instance, the contents of a loop or if-else statement, but also the implementation of a function or the contents of a module.

While MLIR provides a human-readable universal notation for all dialects and operations, it also supports the implementation of custom parsers and printers that can improve the legibility of a custom dialect.

MLIR is implemented in C++14 and relies heavily on LLVM's codebase. As such, while bindings for C and Python exist, C++ is the primarily supported language. It makes heavy use of C++'s template system to ensure type strictness when transforming the language, and of LLVM's TableGen tool that automates the generation of boilerplate for custom dialects and operations. Some code transformation facilities are provided out of the box, such as pattern matching and tree rewriting.

```
"my_dialect.some_nested_operation" {
    ^block1{
        "my_dialect.some_nested_operation" {
            ^block1 {
                "my_dialect.some_nested_operation" {
                    ^block1 {
                    }
                    ^block2 {
                    }
                    ...
                }
            }
        }
    }
}
```

Listing 3.1: Example of nested operations

Chapter 4

The IaRa compiler and dialect

IaRa is a compiler for SDF built on top of MLIR. It accepts a dataflow network in a format based on the Dataflow Interchange Format (DIF) [29], and generates an executable by scheduling kernel function calls in an appropriate order and amount. It currently provides a simple memory pool-based single-core scheduler. Its structure also allows the implementation of co-optimizations that take advantage of simultaneous access to dataflow graph structure and internal kernel information. The compiler currently includes one such co-optimization, a dead code elimination pass.

4.1 Overview of the compilation flow

The structure of the compiler is shown in Figure 4.1. The DIF source is parsed and translated into the IaRa dialect. If a kernel's source code is available in the C language, Polygeist can be used to translate it into standard MLIR and it can be added to the same module as the dataflow graph, enabling the co-optimization pass and potentially enabling low-level LLVM optimizations such as inlining. IaRa also allows kernels to be externally linked, enabling implementation in other C-compatible languages, such as C++ or Rust, and the usage of precompiled libraries.

After the graph is transformed by any co-optimizations, it is flattened and scheduled, generating a MLIR module with only standard dialects. This is translated into LLVM IR, which can be compiled into an executable.

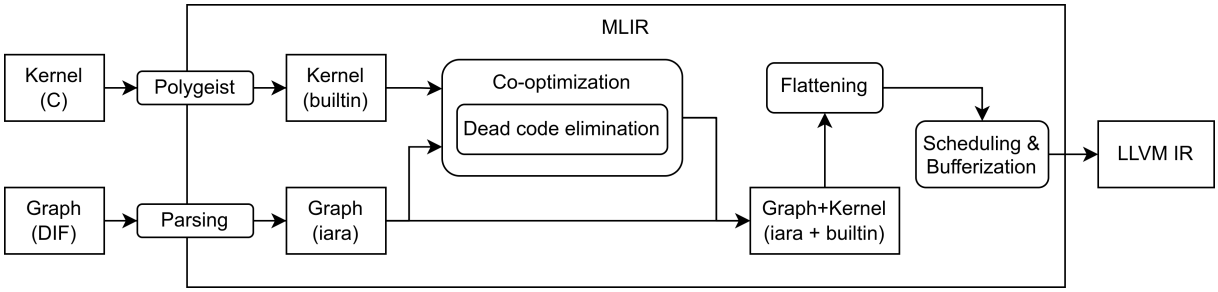

Figure 4.1: Structure of the IaRa compiler

4.2 DIF parser

The Dataflow Interchange Format [29] is designed to be a human-readable textual format for the specification of dataflow graphs, aimed to improve intercommunication between tools. It provides syntax for the representation of actors, edges, parameters, delays, actor implementations ("actortypes", or kernels) and other abstractions.

As a C++-compatible parser of the DIF language is not readily available, IaRa provides a custom parser, which includes some extensions to the DIF specification. Considering that DIF is a relatively simple language, we opted to write the parser from scratch, without relying on external parsing tools such as Lex/Yacc or ANTLR. This also keeps the project dependencies to only LLVM and Polygeist, which simplifies the build process.

To be able to generate function calls of the correct interface when there is no access to kernel source code, the dataflow graph description must be annotated with typing information for the actor's ports. While DIF is flexible enough to represent arbitrary information through the use of its "attribute" directive, there is no widely accepted format for the specification of type information. Furthermore, as this is currently the only user-facing interface of IaRa, it is important to keep the syntax easy to use; in our use-case, the attribute block syntax causes information about an actor's interface to be spread over several points in the source code.

IaRa provides an extension that allows the type annotation to be included immediately after the declaration of a port or parameter, using the colon notation that is common in modern programming languages such as TypeScript, Rust and Swift. An example can be seen in line 3 of Listing 4.1.

IaRa also allows the connections between ports and edges to be expressed with the arrow notation (->) inside the actor definition. While this deviates from the publicly available DIF definition, the fact that there is no standardized format for the expression of types means that this change does not impact any potential compatibility with existing dataflow graphs written in DIF, as some manual changes to the source code would be required by the user either way.

4.3 The IaRa Dialect for dataflow

The IaRa dialect is similar to DIF in level of abstraction. The top-level IR generated from the DIF source shown in Listing 4.1 can be seen in in Listing 4.2.

The dialect defines two custom Attributes: Ports and Parameters. Ports consist of a name (a string), a datatype (a standard MLIR type) and a data rate (an integer). Parameters consists of a name, a datatype and an optional initialization value of that type.

4.3.1 Operations

IaRa's operations consist of the concepts that make up a dataflow graph:

- *GraphOp* represents a single hierarchical level of a graph. GraphOp encodes the name, MoC, graph-level parameters and, in the case of a sub-graph, its outside

interface input and output ports. It contains KernelOps, NodeOps and EdgeOps within its single block, and may coexist with other GraphOps in the same MLIR module.

- *KernelOp* represents an external function that constitutes the implementation for one or more nodes. It encodes the function name of the implementation and its interface (parameters and input/output ports). It is also possible to define default values for the parameters.

- *NodeOp* encodes a single actor in the graph topology. It provides a name for the node and values for the actor parameters. It also contains a reference to an implementation, which can either be an KernelOp that represents a concrete function or a GraphOp that represents an interface to a sub-graph.

- *EdgeOp* encodes the edges of the graph. It consists simply of two node-port pairs that represent the input and output ports within the graph. Optionally, one of the node names may be replaced by the containing graph name, which represents an interface port for hierarchical graphs. It may also accept SDF delay duration and value attributes.

```
dif main {
  parameter {
    width : i32 = 640;
    height : i32 = 480;
  }
  actortype read_rgb_frame {
    param width, height;
    production rgb:u8=921600;
  }
  actortype rgb_to_hsl {
    param width;
    param height;
    consumption rgb : u8 = 921600; // 640 * 480 * 3
    production h : f32 = 307200; // 640 * 480
    production s : f32 = 307200; // 640 * 480
    production l : f32 = 307200; // 640 * 480
  }
  actortype l_to_rgb {
    param width;
    param height;
    consumption l: f32 = 307200;
    production rgb : u8 = 921600;
  }
  actortype write_rgb_frame {
    param width;
    param height;
    consumption rgb : u8 = 921600;
  }
  actor n1 {
    type: read_rgb_frame;
    interface rgb -> e1;
  }
  actor n2 {
    type: rgb_to_hsl;
    interface e1 -> rgb;
    interface l -> e2;
  }
  actor n3 {
    type: l_to_rgb;
    interface e2 -> l;
    interface rgb -> e3;
  }
  actor n4 {
    type: write_rgb_frame;
    interface e3 -> rgb;
  }
}
```

Listing 4.1: DIF source for RGB to Grayscale algorithm

```
module {
  iara.graph @main : "dif"
   param_defaults [
    #iara.param<"width" = 640 : i32>,
    #iara.param<"height" = 480 : i32>] {
    iara.kernel @read_rgb_frame
     params [#iara.param<"width" = <<NULL ATTRIBUTE>>>, #iara.param<"height" = <<NULL ATTRIBUTE>>>]
     outputs [#iara.port<"rgb" : i8[921600 : i64]>]
    iara.kernel @rgb_to_hsl
     params [#iara.param<"width" = <<NULL ATTRIBUTE>>>, #iara.param<"height" = <<NULL ATTRIBUTE>>>]
     inputs [#iara.port<"rgb" : i8[921600 : i64]>]
     outputs [#iara.port<"h" : f32[307200 : i64]>, #iara.port<"s" : f32[307200 : i64]>, #iara.port<"l" : f32[307200 : i64]>]
    iara.kernel @l_to_rgb
     params [#iara.param<"width" = <<NULL ATTRIBUTE>>>, #iara.param<"height" = <<NULL ATTRIBUTE>>>]
     inputs [#iara.port<"l" : f32[307200 : i64]>]
     outputs [#iara.port<"rgb" : i8[921600 : i64]>]
    iara.kernel @write_rgb_frame
     params [#iara.param<"width" = <<NULL ATTRIBUTE>>>, #iara.param<"height" = <<NULL ATTRIBUTE>>>]
     inputs [#iara.port<"rgb" : i8[921600 : i64]>]
    iara.node @n2 : @rgb_to_hsl
    iara.node @n3 : @l_to_rgb
    iara.node @n4 : @write_rgb_frame
    iara.node @n1 : @read_rgb_frame
    iara.edge @e1 : @n1::"rgb" -> @n2::"rgb"
    iara.edge @e2 : @n2::"l" -> @n3::"l"
    iara.edge @e3 : @n3::"rgb" -> @n4::"rgb"
  }
}
```

Listing 4.2: IaRa dialect for RGB to Grayscale algorithm

4.4 Polygeist as a C translator

For IaRa's co-optimization feature, actor source code must be provided in MLIR form. While MLIR does provide an experimental, limited form of conversion from LLVM IR, it does not completely cover the features needed for actor analysis.

As Polygeist offers a reasonably complete C front-end, we rely on it to produce high-level MLIR, as shown in Listings 4.3 and 4.4. Since Polygeist is an ongoing project with frequent updates, it also dictates the version of the LLVM project IaRa is built against.

```
void l_to_rgb(int width, int height, float l[307200], unsigned char *rgb)
    {
  unsigned long long num_pixels = width * height;
  for (unsigned long long i = 0; i < num_pixels; i++) {
    unsigned char L = (unsigned char)(l[i] * 255.0);
    rgb[i * 3] = rgb[i * 3 + 1] = rgb[i * 3 + 2] = L;
  }
}
```

Listing 4.3: Example of actor source in C

```
  func.func @l_to_rgb(%arg0: i32, %arg1: i32, %arg2: !llvm.ptr<f32>, %arg3
   : !llvm.ptr<i8>) attributes {llvm.linkage = #llvm.linkage<external>} {
    %c0 = arith.constant 0 : index
    %c1 = arith.constant 1 : index
    %c2_i64 = arith.constant 2 : i64
    %c3_i64 = arith.constant 3 : i64
    %cst = arith.constant 2.550000e+02 : f64
    %c1_i64 = arith.constant 1 : i64
    %0 = arith.muli %arg0, %arg1 : i32
    %1 = arith.index_cast %0 : i32 to index
    scf.for %arg4 = %c0 to %1 step %c1 {
      %2 = arith.index_cast %arg4 : index to i64
      %3 = llvm.getelementptr %arg2[%2] : (!llvm.ptr<f32>, i64) -> !llvm.
   ptr<f32>
      %4 = llvm.load %3 : !llvm.ptr<f32>
      %5 = arith.extf %4 : f32 to f64
      %6 = arith.mulf %5, %cst : f64
      %7 = arith.fptoui %6 : f64 to i8
      %8 = arith.muli %2, %c3_i64 : i64
      %9 = llvm.getelementptr %arg3[%8] : (!llvm.ptr<i8>, i64) -> !llvm.
   ptr<i8>
      %10 = arith.addi %8, %c1_i64 : i64
      %11 = llvm.getelementptr %arg3[%10] : (!llvm.ptr<i8>, i64) -> !llvm.
   ptr<i8>
      %12 = arith.addi %8, %c2_i64 : i64
      %13 = llvm.getelementptr %arg3[%12] : (!llvm.ptr<i8>, i64) -> !llvm.
   ptr<i8>
      llvm.store %7, %13 : !llvm.ptr<i8>
      %14 = llvm.load %13 : !llvm.ptr<i8>
      llvm.store %14, %11 : !llvm.ptr<i8>
      %15 = llvm.load %11 : !llvm.ptr<i8>
      llvm.store %15, %9 : !llvm.ptr<i8>
    }
    return
  }
```

Listing 4.4: MLIR output of Polygeist for source in Listing 4.3

4.5 Co-optimizing graph and kernel

Using Polygeist to convert the C implementation of actors into higher-level MLIR enables access and modification to their internal operations through MLIR's robust API. This enables the development of fine-grained optimization strategies that make use of both coordination and kernel information that have not been explored in source-to-source dataflow projects, due to previously described challenges.

MLIR functions are already dataflow-like, in that they follow a Single Static Assignment Control Flow Graph (SSACFG) scheme. This means that the results of each operation are given unique names that are not used more than once in a given context, forming a Directed Acyclic Graph (DAG) that can be analyzed and altered with similar techniques as dataflow graphs themselves. This has similarities with the Single-Rate Dataflow MoC.

These co-optimizations may improve performance, memory usage or parallelism, with the potential downside of increasing code size, as they rely on creating altered copies of each kernel depending on their usage within the topology of the graph. It is a similar trade-off as the constant propagation optimization in classic compilers. However, the number of copies is bounded to the number of nodes in the graph, and does not generate higher-than-linear increase in the executable size, as something like a recursive C++ template has the potential to do.

4.5.1 Dead code elimination

Usually, dataflow-level dead code elimination consists of removing actors whose outputs do not contribute to a useful result. If a subset of the outputs is necessary, the whole actor would be computed and a temporary buffer would have to be provided as a location for the unused results. Without access to the internals of the function, this unused memory and computation cannot be avoided.

Most traditional compilers implement some form of Dead Code Elimination (DCE), but they are usually restricted to a single compilation unit, and only to cases where there is no memory aliasing. This cannot be applied when using common SDF memory optimization strategies such as shared memory pools [21], which rely on memory recycling and thus create strong aliasing before DCE can be applied by the compiler.

MLIR provides the tools to overcome this early in the compilation flow. By using MLIR to find the internal function operations that contribute only to unused ports, we can automatically remove them. MLIR provides built-in methods for traversing value definitions and usages. By following the dependency chain of values as they are created and consumed inside the function, each operation can be flagged with all inputs that they depend on and all outputs that depend on its results. The operations whose results are required only for unused results can be safely removed, saving instructions and memory in the final executable.

If an actor with unused outputs is the only instance of a kernel in the dataflow graph, this optimization can be done with no drawbacks. In the general case, however, this optimization is a kind of code specialization, potentially trading off additional program size for improved program performance. In this work we choose to perform systematic

specialization, always performing the optimization if possible.

```
void cartesian_to_polar(
    float *x, float *y,     // input ports
    float *r, float *theta) // output ports
{
  *r = sqrt(*x * *x + *y * *y);
  *theta = atan2f(*y, *x);
}
```

Listing 4.5: Example kernel implementation in C.

```
func @cartesian_to_polar(
  %arg0: llvm.ptr<f32>, %arg1: llvm.ptr<f32>,
  %arg2:  llvm.ptr<f32>, %arg3:  llvm.ptr<f32>)
{
  %1 = llvm.load %arg0 : !llvm.ptr<f32>
  %2 = arith.mulf %1, %1 :  f32
  %3 = llvm.load %arg1 : !llvm.ptr<f32>
  %4 = arith.mulf %3, %3 :  f32
  %5 = arith.addf %2, %4 :  f32
  %6 = math.sqrt %5 :  f32
  llvm.store %6, %arg2 :  !llvm.ptr<f32>
  %9 = call @atan2(%3, %1)
   :  (f32, f32) -> f32
  llvm.store %9, %arg3 :  !llvm.ptr<f32>
  return
}
```

Listing 4.6: MLIR Polygeist output from source code in Listing 4.5. Operations that flow into specific output ports are highlighted.

Consider the C function in Listing 4.5 and its MLIR representation (Listing 4.6), where the operations that flow exclusively into **r** and theta are highlighted in **bold** and underlined, respectively. These represent the sets of operations that can be removed if the corresponding output port is unused. Finding this separation consists of following the value DAG backwards from the port-writing `llvm.store` operations, and keeping a table of which output values depend on each operation.

As MLIR automatically keeps a graph data structure that tracks the references and definitions of all intermediary values, this can be implemented with a simple recursive algorithm that walks this graph. Unlike LLVM IR, MLIR provides a structured control flow dialect that simplifies the tracking of values entering and leaving control flow regions; reasoning about branches and jumps is not necessary. However, since we're relying on LLVM to generate optimized machine code, we can skip this recursive step; it is enough to just delete all operations that directly access the memory of the output we want to remove. We can leverage the built-in function-scope dead code elimination implemented in LLVM to guarantee that all unused intermediary values and their associated operations will be removed in the final binary.

There are a couple of caveats. This method does not apply for operations that access global variables or call functions with side effects, as they may be difficult to trace between

different sections of the function. It also assumes that there is no aliasing between the pointers of each port, i.e., the memory assigned to one port won't be accessed by offsetting the pointer of another port. In the case of SDF, it is assumed that a correct kernel implementation conforms to this.

The compiler then creates a copy of the kernel, removes the relevant operations, and updates the C interface to exclude the removed output parameters. All nodes that reference this kernel with this specific subset of unused ports have their implementation fields updated. If there are no remaining nodes that reference the original function, it is deleted.

4.5.2 Graph validation

Once we have finished applying all of the kernel optimizations, we proceed with the traditional SDF pipeline. This requires flattening the graph into a single hierarchical level, recursively copying any sub-graphs and inserting them into the top-level.

Once the graph is flattened, it assumes a normal SDF form. The ports of actors, in an SDF graph, are assigned constant data rates. These are positive integers that represent the amount of tokens that flow through that port, for each actor firing. The amount of firings per graph iteration, for the producer and consumer actors, must respect the ratio between their connected ports. For instance, if node A has an output port with rate 2 that is connected to node B through a port of rate 3, we know that node A must execute 3/2 times as often as node B. This is necessary to prevent the size of the FIFO from growing without bound (if A is fired too often) and to prevent B from being starved of tokens (if A is not fired often enough).

A schedule that prevents these situations and guarantees that all activations have the correct ratios is said to be admissible. If the SDF network is a tree, it is guaranteed that such a schedule exists.

However, as this ratio relationship between nodes that share an edge travels in both directions, it may find a conflict on graphs that are not trees (either a cyclic directed graph or a DAG with paths that diverge and then converge). Therefore, the admissibility of the graph needs to be tested.

Lee et al. [35] have described a mathematical method to determine if a graph is admissible. We use an equivalent method that involves an iterative traversal of the graph.

Our method consists of attributing a rational number to each edge of the graph, based on the rate of its input and output ports, and an activation number to each node.

Starting from an arbitrary node, we assign it an activation number equal to one. From this node, we walk through the whole graph, ignoring the direction of the edges. Whenever we arrive at an unassigned node, we assign its activation number by multiplying or dividing the activation number of the previous node with the ratio of the traversed edge. If we encounter a node with an already assigned activation, we make sure it is consistent with the value that would be assigned otherwise. If we find an inconsistency, the graph is not admissible.

Provided that no inconsistencies were found, we normalize all admissions by making sure they are all positive integers. This is done by multiplying all of them with the least common multiple of all of the activation's denominators. After this step is done, the

activation numbers correspond to each actor's total number of firings per graph iteration, which will be used in the next step.

4.6 Scheduling and Bufferization

Once we have the activation numbers for every actor, scheduling and bufferization take place. Scheduling consists of determining a valid order in which to fire each actor, and bufferization consists of assigning a location in memory to each token, which will be passed as input and output arguments to the kernel function of each actor. Many strategies require both processes to be done at the same time. IaRa provides two bufferization strategies: a single-assignment strategy and a memory pool strategy.

4.6.1 Single Assignment Bufferization

This strategy consists of assigning an exclusive memory location for every token in a graph iteration, in the order that they are generated by the schedule. The amount of memory used is proportional to

$$\sum_{n \in N} A_n \sum_{o \in O(n)} R(o) S_o$$

for a graph with actors $n \in N$, activation numbers A_n, output ports $O(n)$, output data rates $R(o)\ \forall\ o \in O(n)$ and token sizes S_o. There are upsides for having a single memory location for every token, such as predictable cache behavior and compatibility with existing low-level optimizations like dead code elimination and register promotion. However, the memory consumption scales badly with the size and complexity of graphs, making it generally unfeasible for memory-constrained applications.

The implementation provided by IaRa uses the same scheduling strategy described in the next section, only overriding the memory assignment phase.

4.6.2 Memory Pool Bufferization

The optimization of memory in dataflow compilation is a well-researched problem. When minimizing memory usage, an useful abstraction is the Memory Exclusion Graph (MEG), which consists of Memory Objects that represent each produced token in an SDF period [19] and contain references to their memory allocation. Tokens that are involved in the same actor activation (and which, therefore, must not share the same space in memory) are connected together in the MEG.

There are several strategies to bufferize a MEG, particularly when it comes to parallel schedules. It can be proven that the minimum memory footprint for an SDF network is equivalent to the MEG's Maximum Weight Clique (its largest fully connected sub-graph), which can be computed by exact algorithms or approximated with an heuristic. The memory requirements depend on the shape and rates of the graph, but, as memory can be reused, it scales much better with the graph's size than the single assignment strategy does, particularly in graphs with long linear dependency chains. However, devising a schedule

for an optimal MEG is a computationally complex problem. It has been experimentally determined [19], however, that there are simple heuristics that achieve similar results with little compromise in memory usage and performance.

The current single-core pipeline of IaRa provides a post-scheduling, first-fit allocation strategy and a self-timed greedy scheduling strategy, for their simplicity and high memory efficiency. Through a process called symbolic execution [26], the size and location of tokens in a shared memory pool is determined and the MEG and the schedule are simultaneously constructed. The strategy is post-scheduling, because the schedule itself is independent of the bufferization step, even if they are constructed simultaneously for the sake of ease of implementation; it is first-fit, which means that the lowest-address empty space of sufficient size will be used for each allocation; it is self-timed, which means that each node executes as soon as possible after its inputs become available, and it is greedy, because no deep analysis is made when selecting a candidate actor to execute next.

The algorithm itself (1) consists of following the scheduling strategy as if real data were being processed, creating and erasing Memory Objects for each token that is produced or consumed, but without executing the kernels. One by one, nodes are selected to be symbolically executed and a size and location for the memory of their ports is allocated in a memory pool, which grows as necessary if empty space of suitable size is not found. This ensures that any patch of memory is only allocated while its contents are live, and that it can be freed and reused after the contents are consumed by an actor.

```
1  Pool = new MemoryPool;
2  Schedule = new List<Firing>;
3  EmptyAllFIFOs();
4  while there are actors with remaining firings do
5  |   Node = SelectReadyNode();
6  |   Node.RemainingFirings -= 1;
7  |   Firing = new ScheduleFiring(Node);
8  |   for Output in Node.Outputs do
9  |   |   M = new MemoryObject;
10 |   |   M.size = Output.Rate;
11 |   |   Output.FIFO.Push(M);
12 |   |   Firing.AddOutput(M);
13 |   |   Pool.AllocateOrGrow(M);
14 |   end
15 |   for Input in Node.Inputs do
16 |   |   Objs = Input.FIFO.TakeObjs(Input.Rate);
17 |   |   Firing.AddInput(Objs);
18 |   |   Pool.Free(Objs);
19 |   end
20 |   Schedule.Push(Firing);
21 end
```

Algorithm 1: Joint scheduling and bufferization algorithm

Here, $SelectReadyNode()$ returns a node that contains enough tokens in its input FIFOs. $Firing$s contain a reference to the actor's kernel and the Memory Objects for that particular execution's inputs and outputs, which can be translated into memory pointers to be passed to the kernel. $AllocateOrGrow()$ creates a new Memory Object in the pool, in the lowest offset that can accept it. $FIFO.TakeObjs(size)$ retrieves an

amount of Memory Objects from the FIFO that matches the size; if the Memory Object at the front of the queue is too large, it is broken up into smaller pieces. For instance, if it was a Memory Object that was created by an output port with rate 10, and the current input only consumes 4 tokens, the large Memory Object will be broken into two Memory Objects of sizes 4 and 6, and only the first will be consumed.

Once the algorithm finishes, the schedule contains an ordered list of firings and the memory offsets for its function call arguments, as well as the total size of the shared memory pool, to be statically allocated.

There is some flexibility when choosing heuristics for node selection and Memory Object allocation, which can affect the performance of the schedule and the final pool size, as explored by [19]. A schedule that favors firing the same node sequentially as many times as possible has better cache characteristics than code that fires available nodes in a round-robin fashion; likewise, it is preferable to fire a node whose input tokens have been recently produced, as it is likely that they will still be cached. When optimizing for memory, the compiler can instead prioritize nodes that will cause the smallest increase in the pool size, or maximize the biggest contiguous empty gap.

Given the possibility space when choosing these strategies, we opted to implement a baseline scheduling strategy that does not differentiate between candidate nodes; advanced strategies are kept for a future work. Our pool memory allocation scheme uses a first-fit strategy that has been experimentally determined [19] to be close in effectiveness to the theoretical optimum.

Since this strategy employs shared memory, some memory aliasing is unavoidable: the same memory offset may refer to different data at different points during the schedule's execution. This obstructs some commonly-applied low-level optimizations further down the compiler pipeline, such as LLVM's built-in function-level dead code elimination. This has motivated the development of the high-level DCE pass mentioned in section 4.5.1.

4.6.3 Code generation

The scheduling pass transforms the IaRa dialect into standard MLIR, which contains the buffer allocation and the ordered function calls. The IR, through the use of built-in conversions, can be lowered into the LLVM dialect and then to LLVM IR. The LLVM IR output can be compiled into objects or executable files using LLVM's `opt` command, or through the use of Clang.

Chapter 5

Experimental Results

To demonstrate that the compiler is functional, we have implemented a number of applications that have equivalent versions available for the Preesm compiler. In this chapter, we describe the algorithms, provide performance and memory measurements with Preesm and our compiler, and discuss the results.

5.1 Experimental setup

Experiments were conducted on a desktop computer running Ubuntu 22.04 with an Intel i5-4460 3.7GHz CPU, 8GB of RAM, and SSD storage. The project is built against development versions of LLVM (commit `89525cbf`) and Polygeist (commit `745d6841`). Preesm version 3.21 is used.

Both IaRa and Preesm are configured to use Clang 15 as a backend, as compiled from this LLVM version, with the `-O3` flag. In the Preesm experiments, it is used to compile its generated C sources, and in the IaRa experiments, it is used as an LLVM interface to compile its LLVM IR output and externally-linked source files.

Timing and memory information was obtained by using the GNU `time` command on a Linux machine. Memory usage is measured using the maximum resident set size (RSS), and timing information relates to wall time.

5.2 Applications

All chosen applications except RGB to Grayscale have publicly available Preesm implementations. Table 5.1 shows the number of nodes, edges, individual C kernels and lines of code for each application.

	RGB to GS	Sobel	S. Detection	SIFT
Node count	4	3	8	57
Edge count	3	4	7	108
Kernel count	4	3	8	33
Lines of C code	68	644	2743	5460

Table 5.1: Number of nodes, edges, kernels and lines of code per application

5.2.1 RGB to Grayscale

This algorithm (Fig.5.1) takes as input a raw rgb24-encoded color video file and sets its hue and saturation levels to zero, before outputting it in the same rgb24 format, resulting in a grayscale version. Here, we use a general-purpose RGB to HSL actor, and discard the H and S outputs; the L to RGB actor reverses the conversion, setting the H and S channels to 0. Both actors operate on an entire 640x480 video frame at a time, as opposed to single pixel values. Results are arranged in Table 5.2.

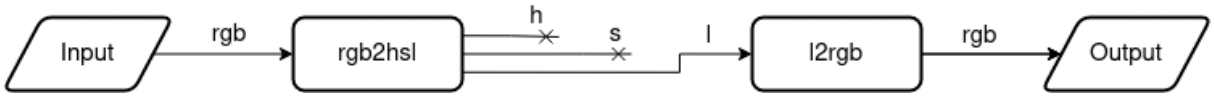

Figure 5.1: SDF dataflow network for the algorithm. One data token firing corresponds to an entire frame of uncompressed 480p video.

To measure the impact of the DCE pass, we generate two versions of the executable. Both versions are compiled by converting the C implementation of the actors into MLIR using Polygeist, guaranteeing that all of LLVM's built-in optimizations will be equally applied (such as automatic inlining and constant propagation). The optimized version differs from the baseline version only on the enabling of the DCE pass.

	Preesm	IaRa (No DCE)	IaRa (With DCE)
Time [s]	4.151	4.020	2.689
Max RSS [KB]	11084	6300	3838
Speedup	1.0×	1.03×	1.54×
Mem. Usage	1.0×	0.56×	0.34×

Table 5.2: Results for RGB to GS algorithm, compared against Preesm

Since Preesm does not deal automatically with unused ports, an equivalent Preesm project was created that writes the unused values into a static buffer. It can be seen that the non-DCE-optimized version achieves a similar time, but better memory characteristics than Preesm. This can be explained by the extra unused buffer. The version that uses DCE achieves considerable gains in both performance and memory.

Comparing the results of the DCE and No DCE versions against each other yields Table 5.3. The expectation is that our DCE pass will remove the calculations related to the H and S outputs of the actor, which constitute a considerable share of the computation.

	IaRa (No DCE)	IaRa (With DCE)
Speedup	1.0×	1.49×
Mem. Usage	1.0×	0.60×

Table 5.3: Improvements of DCE pass

The DCE pass removes kernel operations related to the computation of the H and S values. This can be observed as a speedup of 1.49. The optimized version also only

consumes a fraction of the memory of the unoptimized version; this corresponds to the amount of memory required to receive the tokens of the H and S unused outputs.

This application was chosen to highlight the potential of this optimization strategy. The hue and saturation outputs of the RGB to HSL algorithm are relatively expensive to compute. This sort of general-purpose conversion function is common in real-life applications, and we are confident that the chosen example faithfully represents a real use-case.

5.2.2 Sobel filter

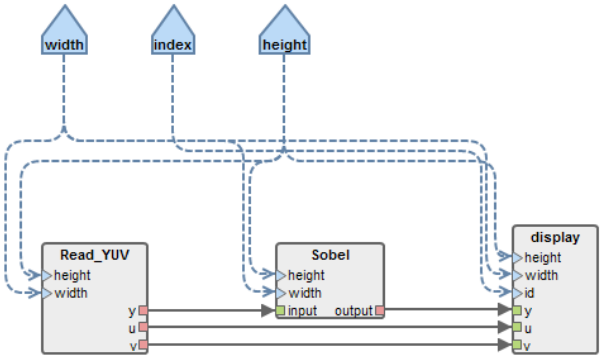

Figure 5.2: Diagram of the Sobel algorithm, as seen in the Preesm website

.

This algorithm is provided as an example implementation in Preesm's website (Fig.5.2). It applies the Sobel filter onto a video, isolating outlines and hard edges. There are 3 nodes, which process an entire video frame per iteration. Results are shown in Table 5.4

	Preesm	IaRa
FPS	1398	1402
Max RSS [KB]	33404	33232
Speedup	1.0×	1.0×
Mem. Usage	1.0×	0.99×

Table 5.4: Measurements for Sobel Filter

IaRa did not achieve a considerable improvement in performance or memory. This is due to the simplicity of the graph; the optimal schedule is trivially derived, with no room for improvement. The small memory difference can be attributed to the boilerplate that is automatically generated by Preesm, intended for multicore schedules.

5.2.3 Swimmer Detection

This algorithm processes an underwater video of a swimmer by converting it to HSV (hue, saturation and value), and draws a bounding box around the swimmer (Figure 5.3. The video has 7 seconds of 640x480 H.264 encoded footage, at 25 frames per second, with a bitrate of 3930 Kbps. IaRa was able to achieve a considerably better schedule than Preesm, despite requiring more memory. Results are shown in Table 5.5.

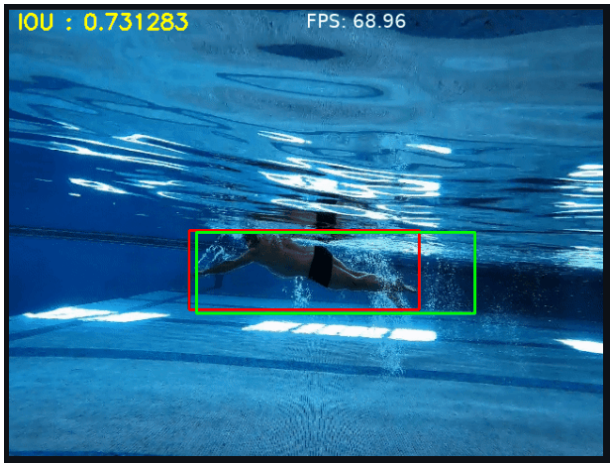

Figure 5.3: Screenshot of the algorithm at work (red box is ground truth, green box is estimate)

	Preesm	IaRa
Time [s]	2.814	1.867
Max RSS [KB]	269776	357244
Speedup	1.0×	1.51×
Mem. Usage	1.0×	1.32×

Table 5.5: Measurements for Swimmer Detection algorithm

The reason for the performance improvement is not clear. One possible explanation is that Preesm's optimization for broadcast nodes causes it to fire nodes in a round-robin manner, which may cause worse cache behavior than IaRa's default strategy of firing the same node as many times as possible; this may also explain the increased memory usage, as the maximum size of FIFOs increases.

5.2.4 SIFT

SIFT is a widely used algorithm for computer vision applications that automatically detects features in images. It is relatively complex: the graph contains 57 nodes and 108 edges, and the 33 kernel implementations span 4789 lines of C code.

	Preesm	IaRa
Time [s]	0.538	0.977
Max RSS [KB]	198448	857540
Speedup	1.0×	0.55×
Mem. Usage	1.0×	4.32×

Table 5.6: Measurements for SIFT algorithm

IaRa underperformed Preesm, generating an executable half as fast and using over four times more memory. This can be explained by the fact that SIFT contains several broadcast nodes, which generate copy operations. Preesm provides specific optimizations

that are able to optimize broadcasts, which are not yet supported in IaRa. Results can be seen in Table 5.6

5.3 Discussion

The data show that, while IaRa is functional and viable for small-scale algorithms, it suffers when it comes to scalability. Since IaRa lacks many of the sophisticated memory optimizations of Preesm, applications with many nodes and many broadcast operations do not achieve the same performance or memory economy.

However, there is no fundamental reason these same optimizations could not be adapted to IaRa's scheduler; MLIR's modular nature allows for an alternate scheduling pass to be developed independently of IaRa's other features. There is also potential for the future development of a compatibility layer between Preesm and IaRa, which would allow for the use of the features of both projects in the development of the same application.

Chapter 6

Conclusion

Dataflow models of computation have been originally developed to assist in the development of portable, high performance applications, and they have proven useful for signal processing systems. When it comes to the applicability of dataflow MoCs advances in compiler research, the strict separation that Dataflow enforces between coordination and kernel code has been problematic.

With our experiments, we have demonstrated that MLIR makes it possible to bridge this gap. By retrieving information from the graph topology description, we have combined two optimization strategies that would be incompatible otherwise: alias-creating memory pools and dead code elimination. To achieve this, we have used existing tools such as DIF and Polygeist, and implemented traditional dataflow algorithms in a cutting edge compilation framework in the form of the IaRa dialect, showing that there is potential for further integration between projects. With the implementation of an early dead code elimination pass in this framework, we have achieved a 149% speedup and a 60% memory usage improvement in a video processing application.

MLIR made it possible to implement IaRa within 1 person-year without any previous experience of LLVM/MLIR. A similar optimization would be feasible using a custom IR, but this would also require the reimplementation of a whole C compiler, which is far from trivial. Not relying on LLVM/MLIR infrastructure and existing dialects would require the designer to build a novel IR target, to reimplement all of the fine grain optimizations available natively in LLVM, such as DCE, and to emit correct machine code. Meanwhile, modifying an existing source to source compiler like Preesm [45] to allow for this functionality would require extensive modification of the internal graph representation and code generation, as it is not prepared to modify the internals of a C function in a flexible way.

6.1 Future research directions

There is a wide set of possibilites to explore from here, many of them enabled by the large variety of projects that are already found within the MLIR ecosystem. For instance, efforts can be made in integrating Polygeist polyhedral optimization passes, making further use of the access to kernel implementations; multi-core schedulers can be developed with MLIR's built-in dialects for parallelism, such as OpenMP; and support for the highly-optimized

implementations of the existing machine learning dialects could be integrated into the scheduler, improving the portability of algorithms across multiple targets.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.*, pages 423–424. IEEE, 2004.

[3] AMD. The ROCm framework. `https://rocmdocs.amd.com/en/latest/`, 2016. [Online; accessed 31-July-2022].

[4] Apple. The Metal framework. `://developer.apple.com/metal/`, 2014. [Online; accessed 31-July-2022].

[5] John Bachan. Constructing hardware in a scale embedded language. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2014.

[6] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

[7] Gérard Berry. Synchronous design and verification of critical embedded systems using scade and esterel. *Lecture Notes in Computer Science*, 4916:2–2, 2008.

[8] Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, 2001.

[9] Shuvra S Bhattacharyya, Gordon Brebner, Jörn W Janneck, Johan Eker, Carl Von Platen, Marco Mattavelli, and Mickaël Raulet. Opendf: a dataflow toolset for reconfigurable hardware and multicore systems. *ACM SIGARCH Computer Architecture News*, 36(5):29–35, 2009.

[10] Shuvra S Bhattacharyya, Johan Eker, Jörn W Janneck, Christophe Lucarz, Marco Mattavelli, and Mickaël Raulet. Overview of the mpeg reconfigurable video coding framework. *Journal of Signal Processing Systems*, 63(2):251–263, 2011.

[11] Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology*, 21(2):151–166, 1999.

[12] Greet Bilsen, Marc Engels, Rudy Lauwereins, and JA Peperstraete. Static scheduling of multi-rate and cyclo-static dsp-applications. In *Proceedings of 1994 IEEE Workshop on VLSI Signal Processing*, pages 137–146. IEEE, 1994.

[13] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[14] Joseph Tobin Buck and Edward A Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *1993 IEEE international conference on acoustics, speech, and signal processing*, volume 1, pages 429–432. IEEE, 1993.

[15] Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. Maps: Mapping concurrent dataflow applications to heterogeneous mpsocs. *IEEE Transactions on Industrial Informatics*, 9(1):527–545, 2011.

[16] François Chollet et al. Keras. `https://keras.io`, 2015.

[17] Pedro Ciambra, Mickaël Dardaillon, Maxime Pelcat, and Hervé Yviquel. Co-optimizing dataflow graphs and actors with mlir. In *2022 IEEE Workshop on Signal Processing Systems (SiPS)*, 2022.

[18] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[19] Karol Desnos. *Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs*. PhD thesis, Rennes, INSA, 2014.

[20] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S Bhattacharyya, and Slaheddine Aridhi. Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 41–48. IEEE, 2013.

[21] Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi. Memory Analysis and Optimized Allocation of Dataflow Applications on Shared-Memory MPSoCs. *Journal of Signal Processing Systems*, 80(1):19–37, July 2015.

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

[23] Johan Eker, Jörn W Janneck, Edward A Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[24] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. Mlir as hardware compiler infrastructure. In *Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[25] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij. Throughput analysis of synchronous data flow graphs. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 25–36, 2006.

[26] A.H. Ghamarian, M.C.W. Geilen, T. Basten, B.D. Theelen, M.R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In *2006 Formal Methods in Computer Aided Design*, pages 68–75, 2006.

[27] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[28] Julien Heulot, Maxime Pelcat, Karol Desnos, Jean-Francois Nezan, and Slaheddine Aridhi. Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps. In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pages 167–171. IEEE, 2014.

[29] Chia-Jui Hsu, Fuat Keceli, Ming-Yung Ko, Shahrooz Shahparnia, and Shuvra S Bhattacharyya. Dif: An interchange format for dataflow-based design tools. In *International Workshop on Embedded Computer Systems*, pages 423–432. Springer, 2004.

[30] Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. The lustre v6 reference manual. *Verimag, Grenoble, Dec*, 2016.

[31] John L Kelly, Carol Lochbaum, and Victor A Vyssotsky. A block diagram compiler. *The Bell System Technical Journal*, 40(3):669–678, 1961.

[32] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[33] Rudy Lauwereins, Marc Engels, and JA Peperstraete. Grape-ii: A tool for the rapid prototyping of multi-rate asynchronous dsp applications on heterogeneous multiprocessors. In *The Third International Workshop on Rapid System Prototyping*, pages 24–25. IEEE Computer Society, 1992.

[34] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers*, 12(03):261–303, 2003.

[35] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[36] Edward A. Lee and Thomas Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[37] Paul LeGuernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[38] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[39] Microsoft. Project verona. `https://github.com/microsoft/verona`, 2019.

[40] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Affine c in mlir, 2021.

[41] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008.

[42] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 804–817, 2021.

[43] Paul Osmialowski. How the flang frontend works. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC 2017)*, 2017.

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[45] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *Education and Research Conference (EDERC), European Embedded Design in*, pages 36–40, Sept 2014.

[46] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *2014 6th european embedded design in education and research conference (EDERC)*, pages 36–40. IEEE, 2014.

[47] Rob Pike. The go programming language. *Talk given at Google's Tech Talks*, 14, 2009.

[48] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[49] Jocelyn Sérot. Hocl: High level specification of dataflow graphs. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, pages 11–22, 2020.

[50] Jocelyn Sérot and François Berry. The caph language, ten years after. In *International Conference on Embedded Computer Systems*, pages 336–347. Springer, 2019.

[51] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[52] Sander Stuijk, Marc Geilen, and Twan Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008.

[53] William Robert Sutherland. *The on-line graphical specification of computer procedures.* PhD thesis, Massachusetts Institute of Technology, 1966.

[54] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196. Springer, 2002.

[55] Herve Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickaël Raulet. Orcc: Multimedia development made easy. In *Proceedings of the 21st ACM international conference on Multimedia*, pages 863–866, 2013.