Program Synthesis for Heterogenous Accelerators

Bruce Collie

Master of Science by Research
School of Informatics
University of Edinburgh
2018

Abstract

Heterogenous architectures and workload-specific accelerators can provide significant performance improvements across many problem domains. However, their use by developers is limited—integrating accelerators with a program is difficult and often produces non-portable code. Existing work to solve this problem has focused on compiler-based detection and rewriting of code to run on a particular accelerator. Despite promising results, this approach still requires significant manual effort by the developer to support new accelerators.

To reduce the effort of integrating accelerators with code, we aim to automatically learn a model of their behaviour. A model learned in this way can be used with existing tools to map code to the accelerator, without the programmer having to write accelerator-specific code to do so. Our approach to this problem is to use program synthesis to synthesise programs that behave equivalently to an accelerator.

We select linear algebra as a test domain, and show that our program synthesiser (AccSynt) is able to synthesise programs with complex nested control flow and data access patterns. Additionally, we describe an experimental case study that demonstrates the potential performance gains from using accelerators in this domain, and show how these results relate to our program synthesis techniques.

Acknowledgements

My sincerest thanks go to my supervisor, Professor Michael O'Boyle, for his ever-present help and insightful conversations throughout this project. Thanks also to Philip Ginsbach for his technical assistance and invaluable tools, and to the other Pervasive Parallelism students for excellent company throughout the year. Finally, I am forever grateful to my family and girlfriend Alice for their continual love and support of my work.

This work was supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1) and the University of Edinburgh.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Bruce Collie*)

Table of Contents

Chapter 1

Introduction

There is an increasing trend in hardware development towards heterogeneous devices and workload-specific accelerators. The slowing down of Moore's law and Dennard scaling mean that general-purpose processors are no longer the most effective choice for many problem domains; the power consumption and design constraints imposed by generality mean that dark silicon is widespread in traditional processors. Many problem domains (for example, machine learning, ubiquitous computing and signal processing) now rely on specialised accelerators to maximise their performance to power ratios. Such accelerators are available in many different forms—optimised software libraries, architecture-specific methods and specialised hardware designs are all common.

Making effective use of accelerators requires a detailed understanding of the performance characteristics of a particular problem—there are often overheads involved in using them that confound the potential benefits. For example, using a graphics processing unit (GPU) to accelerate data-parallel problems is often only worthwhile for suitably large instances; for smaller cases the time taken to copy data between the host and the device can be prohibitive. In general, using accelerators can lead to significant performance improvements, but only when used judiciously. Even allowing for these potential downfalls, accelerator usage has become widespread across a number of problem domains, a trend that shows no sign of slowing down.

Despite their newfound ubiquity, making use of accelerators from application code is not easy. Detailed knowledge of both its source code and the accelerator's behaviour are required, and even when an accelerator is integrated the resulting code is unlikely to be portable. Ideally, this task should be done by the compiler. It should be able to identify code that can be run using an accelerator at compilation time, and to replace

this code with usages of the accelerator without involving the programmer. By doing this, the programmer can separate their algorithmic intent and program design from the constraints imposed by specific accelerators. Another benefit of this approach is that the program will remain portable to different accelerator implementations in the future, providing they are compatible with the compiler integration.

There are several facets to this type of compiler integration. We must be able to identify code that matches a particular pattern—that is, given a compiled program, what sub-programs meet a set of structural requirements? Then, we must be able to transparently rewrite this code to use an accelerator interface. These must both be written manually by a programmer for each new accelerator supported. The computational patterns supported and the code rewriting required are different for each and cannot be extracted automatically from an accelerator. Recent and current work is concerned with code discovery and rewriting, and so we focus on the third problem: given an accelerator, how can we automatically determine the computational idioms it supports? Answering this question would allow programs to automatically support future accelerators using current tooling for discovery and rewriting.

Figure 1.1 shows the different parts of this problem, and the dataflow between them. Ideally, we would be able to match user code to arbitrary accelerators, while also selecting the accelerator with the best performance for a particular workload. Existing work can match code to individual accelerators, but being able to do so for arbitrary acclerators is an open question. The work in this dissertation aims to learn a model for the behaviour of arbitrary accelerators, and to build a performance model for them that can be used to select the most efficient one at run time.

In order to support any possible future accelerator, we must be able to express behaviours in a fully general way—any restriction on how behaviour is described might prevent some types of accelerator from being supported in the future. This means that techniques such as classifying an accelerator's functionality as one particular choice from a group of options using machine learning are not the most suitable. Perhaps the most natural way to express computational behaviour in a general way is in fact as a program in some language. That is, any accelerator's behaviour could in fact be expressed as a program. Then, given such a program we would be able to reason about its structure—by comparison, the original accelerator would be a "black box".

Choosing an executable program as a model we aim to learn leads us naturally to program synthesis as a methodology. A subcategory of machine learning, program synthesis aims to automatically generate programs that are correct with regard to a

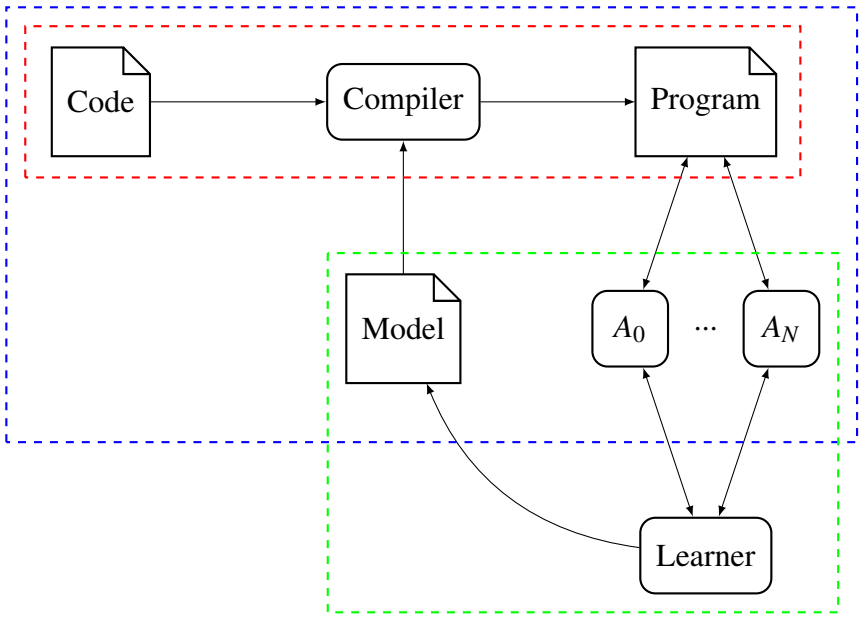

Figure 1.1: Data flow in a system performing compilation and acceleration of user programs. The components highlighted in red are a naïve system that simply compiles user code with no accleration. Recent work uses a handwritten model to match code to a set of accelerators $A_0, ..., A_N$—this is the system highlighted in blue. The work in this dissertation aims to implement the system shown in green—given a set of accelerators, how can we learn their behaviour and use this to build a model compatible with existing tools for matching code?

specification of some kind. This is in general a hard problem—the space of potential programs that could be generated is enormous, and it is often difficult to prove that a solution is in fact correct. However, much prior work on the subject is available to draw upon for a solution. While program synthesis is not an obvious solution to the problem of matching user code to available accelerators, for the problem stated above with the given constraints it is a reasonable suggestion. In this dissertation we aim to develop a program synthesis-based solution to the problem of learning a model for accelerator behaviour, and to evaluate to what extent these techniques allow for generality in the solution.

The remainder of this dissertation is organised as follows. In Chapter 2, we review related work, covering three broad areas: the development of heterogeneous hardware and accelerators, mapping code effectively onto available accelerators, and program synthesis. We describe several example domains to which accelerators can be usefully applied, giving examples from recent work to demonstrate their application. Then, we

examine current methods for mapping code onto accelerators—these methods are useful, but often still require a great deal of manual effort on the part of the programmer. Finally, we examine similar work in program synthesis, showing that applying it to learning accelerator behaviour is a new technique with little similar work in the literature. Chapter 3 provides a brief technical background to the dissertation; it describes the tools and libraries used to implement our program synthesiser, introducing important concepts and terminology. As well as this, we give a short overview of sparse linear algebra techniques that can be used as a reference throughout the rest of the dissertation.

Chapter 4 describes an experimental investigation into the performance improvements that can be achieved using accelerators for sparse linear algebra workloads. This investigation examines four CPU (central processing unit) and GPU implementations of sparse matrix-vector multiplication across different host platforms. In the best case, the accelerator implementation improves performance by more than $50\times$ compared to a naïve sequential implementation. However, these gains are not consistent—some workloads are not improved by accelerator usage. To remedy this, we describe a predictive model that is able to accurately select the best possible accelerator for a given problem at run time. This model combined with the accelerator implementations motivates the implementation work in subsequent chapters.

The primary artifact produced during the course of this dissertation is AccSynt, a program synthesiser that aims to synthesise programs that match the behaviour of an accelerator. Its implementation is split into a general framework for performing program synthesis, and a specific component that is more focused on the initial set of interesting accelerators. This implementation work is described in Chapter 5, along with descriptions of the core algorithms and ideas used to synthesise programs. Chapter 6 evaluates this work with respect to the range of programs that can be synthesised, as well as the time taken to do so. Finally, Chapter 7 summarises the contributions made in this dissertation, and identifies a number of areas for potential future work. In particular, we identify several possible ways in which the specific synthesis methods used in AccSynt can be generalised to further problem domains.

Chapter 2

Related Work

In this chapter we review existing literature with respect to three key questions related to the problem statement identified in Chapter 1. First, in Section 2.1, we examine recent work in accelerators and heterogenous computing—what form do accelerators normally take, and what interfaces do they present to host programs? Then, Section 2.2 examines existing approaches to the automatic mapping of code to accelerator devices. In particular, we look at how host programs can currently be written or adapted to use an accelerator, and the state of current work on automatic code discovery and rewriting to use an accelerator. Finally, in Section 2.3 we review existing work in program synthesis, with particular focus on how it has been used previously to learn the behaviour of hardware systems, finding that there is little or no work related to accelerators and heterogenous devices in the program synthesis literature.

2.1 Heterogenous Computing and Accelerators

There is a recent trend in computer architecture towards designing *heterogenous* systems. These systems are made up of multiple components, each of which has a different architecture or function within the system—for example, GPUs have instruction sets that are designed for very high data parallelism, while digital signal processors typically have good support for multiply-accumulate, saturating arithmetic or ring-buffer memory access. A large number of new heterogenous devices are being designed every year, with applications to many different problem domains such as cryptography [9], bioinformatics [45] and deep learning [27, 35].

Zahran [48] argues that the trend towards heterogenous computing will continue for the foreseeable future, and that our perspective towards hardware and software de-

velopment must change to incorporate heterogenous principles. There are a number of reasons for this. Perhaps the most critical is the demise of Moore and Dennard's respective laws—transistor density and power consumption are no longer scaling as quickly as they have been for the past fifty years. As a result, alternative chip designs are now the most viable option for many computationally intensive workloads. As a representative example, Jouppi et al. [27] at Google describe the implementation of their tensor processing unit (TPU) chip; an analysis showed that a small increase in their usage of deep neural network inference would double their datacenter computation demands using existing methods. They achieve an improvement of up to 80× in the performance to power ratio when compared to a typical CPU+GPU implementation of the same inference tasks.

Heterogenous devices are one instance of what we will refer to throughout this dissertation as *accelerators*—any method of improving the performance of a computational workload with regard to a particular metric. These methods could be software or hardware, and the performance metric could be speed, energy efficiency, or indeed anything applicable to the workload in consideration. This definition is intentionally broad; we examine the potential improvements from using accelerators without being overly concerned about their underlying implementation.

With these definitions in mind, the remainder of this section deals with the following two questions: when is it viable to use an accelerator to improve performance of a system, and how can software authors effectively integrate accelerators with their code?

2.1.1 Accelerator-Friendly Workloads

The primary limiting factor to the use of accelerators is whether or not the system in question has a computational bottleneck. If the work performed by a system is unpredictable or very varied, it is unlikely that an accelerator will be able to provide performance improvements. For example, deep learning inference systems are bottlenecked largely by the performance of matrix multiplication routines—suitably fast implementations of these routines are able to speed up inference while also reducing power consumption [27]. A slightly different example is the always-on motion coprocessor in the iPhone; the ability of the phone to provide constant motion analysis and recording is bottlenecked by the power consumption of the primary processor. Using the low-power coprocessor enables the efficient implementation of a workload that

would not otherwise be possible.

Secondly, an application needs to match the functionality offered by an accelerator precisely in order to take advantage of the improved performance. An example of when this may not be the case is when application code implements a task using a different data format to the one accepted by the accelerator. This type of mismatch requires extra marshalling to be performed in order to use the accelerator, which in some cases can negate any performance gain.

In general, to make use of an accelerator an application needs a consistent bottleneck expressed in a manner compatible with the accelerator.

2.1.2 Using Accelerators in Programs

Even if an application has a bottleneck which is compatible with an accelerator, integrating code with the accelerator can be challenging. The range of interfaces offered by accelerators is very broad—some offer runtime support libraries for user programs, while others provide a direct memory access (DMA) interface that needs to be wrapped with a device driver [47]. The Google TPU [27] is only accessible through a Tensorflow wrapper running on their own cloud computing platform, and many papers only present a hardware description language design that must be synthesized alongside another chip to act as a coprocessor [9, 45]. Table 2.1 gives a brief overview of how much work is required to integrate accelerators with different types of external interface into a program.

Even once a suitable accelerator interface is available, matching this to an existing program is a non-trivial task—program structure can be arbitrarily complex, and rewriting sections to use the accelerator interface entails a great deal of work on the part of the programmer responsible. Automated methods for mapping existing code [16] or high-level algorithmic designs [42] onto accelerators exist, but require detailed knowledge of the accelerator being targeted in order to map the code.

Throughout the rest of this dissertation, we can assume that a software interface to any accelerator is available. This type of interface will almost always take the form of a C function call. In general, adapting an existing application to use an accelerator can require significant engineering work beyond the initial development of the application. We provide a more detailed overview of work related to automated mapping of code to accelerators in Section 2.2.

Accelerator Interface	Work Required to Integrate
Software interface to specialised CPU features or instructions	Calls to library functions, no extra hardware required
Software interface to external hardware	Calls to library functions, appropriate interconnect (e.g. a serial bus protocol or over a network)
DMA interface to external hardware	Implementation of device driver, calls to driver interface, appropriate interconnect
Hardware description language design	Synthesis of design to a reconfigurable architecture, implementation of device drivers, calls to driver interface, appropriate interconnect

Table 2.1: Comparison of the work required to integrate different accelerator interfaces into a user program.

2.1.3 Summary

Heterogenous devices and accelerators can provide significant performance improvements across a variety of different workloads and evaluation metrics—improved throughput, energy efficiency and availability are all possible depending on the workload. Because of recent trends in the performance of traditional general-purpose processors, it is likely that a further move towards heterogenous methods is on the horizon. However, practical integration of accelerators into user programs is still not as simple as it could be—finding an appropriate accelerator and matching it to program structure requires a lot of manual engineering effort.

2.2 Mapping and Optimisation

One of the primary goals of our work is to enable software optimisation by mapping code to accelerator devices. We aim to do this by automatically learning the behaviour of accelerators through program synthesis, such that the synthesised programs can later be used with existing tools to discover similar code in programs. We now give an overview of alternative techniques related to the discovery of code suitable for accel-

eration and the automatic mapping of this code to run on accelerators.

There are several common approaches to the problem of efficiently mapping a programmer's intent to the most efficient accelerator implementation. The three most common approaches are to use runtime support libraries for the accelerator, a domain-specific library, or to automatically map code onto the accelerator. In this section we give an overview of important work using each of these approaches, and discuss how they relate to our goal of learning models for accelerator behaviour.

2.2.1 Domain-Specific Languages

A domain-specific language (DSL) is a programming language that is highly specialised for a single task or problem domain. They express a limited set of highly specialised operations, and are typically designed to make it easier for users to write programs in that problem domain. Uses for DSLs are very diverse—configuration languages, build systems and unit test descriptions are commonly built using a DSL. For performance-critical problem domains where using an accelerator is beneficial, a DSL is useful because it limits the scope of operations expressible in the language. This means that programs can be more easily mapped onto supported accelerators, and that whole-program optimisation is feasible.

Halide [37] is a DSL for expressing image processing pipelines. By using Halide, the algorithmic description of a pipeline can be decoupled completely from performance concerns such as cache-friendliness or data-dependency. The programmer is able to write their code at a high level of abstraction, while still taking advantage of domain-expert knowledge for optimisation. Another DSL for image processing is PolyMage [33], which uses an automatic parameter search to find the optimal implementation for a particular pipeline—they go on to show that this approach gives performance on par with hand-tuned Halide code.

Spampinato and Püschel [40] demonstrate LGen, a DSL for implementing linear algebra programs. They make the observation that existing linear algebra acceleration is biased towards large-scale problems, but that generating optimal code for smaller problems can also provide performance improvements. Similar approaches exist for other domains such as Fast Fourier Transforms [14].

Lift [20, 42] is a functional language designed to be compiled to OpenCL. The authors deliberately distinguish Lift from DSLs such as Halide—Lift is a general purpose language that can be applied to many distinct problem domains (for example, image

processing, machine learning and stencil programs are all well supported). However, for any particular domain Lift shares some philosophy with DSLs. The language is made up of a set of primitive operations, and it uses parameter tuning based on program rewrite rules to optimise programs.

A disadvantage of using a DSL is that they may not be easy to integrate with existing application code. Halide sidesteps this issue by writing the DSL using expressive C++ code—this is known as an EDSL (embedded DSL). This approach is popular, but typically relies on syntactic support from a host language for expressivity. Examples of EDSLs are CλaSH [2] and Chisel [3]; both are used to describe hardware designs as functional programs. In general, however, if a DSL is not designed with interoperability in mind it can be difficult to integrate them into other programs.

2.2.2 Runtime Libraries

Writing performance-critical code using a DSL allows for high-level intent to be expressed clearly in a suitable language. However, using a DSL may not be well-suited to all programs—for example, if the program requires logic that cannot be expressed using the DSL, or if it cannot be easily integrated with the DSL's interface. In these scenarios, it is beneficial to use a lower-level approach that expresses individual operations to a host program. Typically, this is achieved by using a runtime library that provides a convenient interface to well-optimised implementations of performance-sensitive operations. Doing this allows the programmer to decide precisely when to use an optimised implementation, and to integrate these operations tightly with their own code.

This approach is common for linear algebra workloads—many libraries conform to the standard BLAS interface, which describes low-level "building blocks" for linear algebra programs [6]. Other libraries such as SparseX [12] and pOSKI [8] provide highly specialised implementations of sparse linear algebra by performing automatic tuning based on characteristics of the host hardware and the problem at hand. Libraries are also made available by device manufacturers specifically for their hardware—for example, Intel's Math Kernel Library [24].

In general, any software interface to an efficient implementation of a workload falls into this category. Providing an interface of this kind is more convenient for the library developer (no need to write a new DSL) and for the programmer (no need to learn the DSL and easy integration with existing code). However, this comes at the expense of

the opportunity for holistic optimisations afforded by a DSL.

2.2.3 Automatic Mapping

The goal of automatic mapping is to take a naïve user program and automatically work out where an optimised implementation could be applied. This would allow programmers to write programs without being locked into a particular accelerator implementation, or even without knowing how to accelerate the program at all.

For obvious reasons, doing this for arbitrary programs is a hard problem. Even if we know the exact behaviour of an accelerator, finding code that can be run using it is difficult. Ginsbach and O'Boyle [15] describe their interface description language, a constraint-based method for discovering computational idioms in compiled code—they use this approach to discover several different patterns that can be run using an accelerator [16]. For a given accelerator implementation, this allows for compatible code to be easily discovered. However, mapping this code onto an accelerator requires writing a compiler pass to rewrite the target program. Recent prototype work from the same authors aims to make it easier to map discovered idioms to multiple implementations. Their work uses a new DSL to describe the mapping process.

Compared to the volume of work related to implementing DSLs and runtime libraries, there is very little work related to automatic mapping. The task is difficult, and developers are capable of writing code that targets efficient implementations. Existing work on automatically discovering opportunities for improved performance in user code have so far focused on local properties such as peephole optimisations [31, 32] or superoptimisation [36, 38]. However, these approaches do not apply to performance opportunities from accelerators—the patterns discovered using a constraint-based approach can span large non-local regions of compiled code.

2.3 Program Synthesis

Gulwani, Polozov, and Singh [19] describe in depth the difficulty of performing program synthesis—the search space of possible programs is intractibly large, and user intent can be expressed in any number of different ways. As a consequence of this, almost all research in program synthesis involves an element of search space reduction. The exact nature of this search space reduction depends on the synthesis algorithm in question. For example, the syntax-guided approach in [1] has the user supply a

syntactic template for possible solutions, while Godefroid and Taly [17] use a "smart sampling" method to select the best possible test case for disambiguating candidate solutions.

In the context of learning a model for an accelerator, the program synthesiser has no domain-specific information about the structure of potential solutions—the accelerator being learned could be performing any arbitrary computation. This means that any information that can narrow the search space of potential programs must be supplied by the user of the synthesiser, an idea that is not uncommon in the program synthesis literature. Syntax-guided synthesis [1] is an instance of this idiom, where the information supplied is the syntactic form of a potential solution. Similarly, SKETCH [39] has the user supply a series of statements that might appear in a program, alongside potential syntactic templates. A slightly different angle is taken by Leung, Sarracino, and Lerner [29], who synthesise parsers for a grammar by querying the user interactively for disambiguating examples. Even more distinct is CLGen [**Cummins2017a**], which uses recurrent neural networks to generate syntactically valid programs that can be used for compiler fuzzing.

Oracle-guided program synthesis [25, 26] describes a group of techniques that share a common style. An oracle specification refers to any interface that can answer queries about the synthesis problem or potential solutions to it. Commonly, the type of oracle found in practical examples is an input-output oracle—one that can supply the correct output for a given input. The prevalence of this type of oracle is due in part to work on programming by example, where a user supplies a set of examples by hand, which are to be generalised into a program. This has been applied successfully to string processing in spreadsheets [18], compiler fuzzing [5], processing algebraic datatypes [34], and database query synthesis [46]. In the context of this dissertation, accelerators can be viewed as input-output oracles; it is easy to get an output that corresponds to an input, but hard to get any further information. This view means that previous literature related to oracle-guided synthesis is likely to be relevant to the work done in this dissertation.

Studying the behaviour of hardware devices using program synthesis is not a common use case, especially with regard to heterogenous devices and accelerators. The closest analogue in the literature is a body of work with the common goal of learning semantics for CPU instruction sets. For example, Godefroid and Taly [17] learn bit-vector representations of x86 instructions using input-output examples. Similarly, Heule et al. [23] learn the semantics of the same instruction set by using a 'stratified'

synthesis approach that composes previous results to reach a solution at each synthesis step. Other work uses a compiler back-end to synthesise instruction semantics that can be used for decompilation [**Hasabnis2015**, 21, 22]. Lim and Reps [30] learn the semantics of machine code in order to build generic tools for analysing binaries.

Chapter 3

Background

This chapter gives an overview of the technical background to the project as follows: Section 3.1 introduces the field of program synthesis, defining key terminology and summarising important results and techniques. Then, in Section 3.2, we give an introduction to LLVM [28] and the compiler-based tooling used to implement AccSynt, the program synthesiser described in Chapter 5. Finally, Section 3.3 summarises the motivation for sparse linear algebra techniques, giving a brief description of the relevant formats and algorithms. Chapter 4 motivates and makes use of these sparse methods; this chapter acts as a technical reference for the case study.

3.1 Program Synthesis

Being able to automatically generate a program based only on a specification has been a goal of computer science research for nearly sixty years [13]. At the highest level, the problem statement is to accept a specification that a correct program should satisfy, and then to subsequently produce such a program. Different applications of program synthesis use different representations for programs and specifications. There therefore exists a wide variety of different algorithms and concepts within the scope of program synthesis. In this section we give a brief summary of important terminology and definitions from program synthesis, along with an overview of significant algorithms and techniques, and a discussion of the difficulties associated with program synthesis.

3.1.1 Definitions and Terminology

Throughout the rest of this dissertation, we will make reference to some key definitions from program synthesis. These definitions are given below along with some running examples from the literature.

Program A structured representation of a computation. For example, Heule et al. [23] synthesise bit-vector formulae as their 'programs', Srinivasan and Reps [41] deal with linear sequences of machine code instructions, and Gulwani [18] builds string-manipulation programs by composing components from a library. These three choices are all conceptually very different, but can each be seen as a computation mapping inputs to outputs. The choice of a program representation is strongly dependent on the synthesis problem in question.

Specification Programs can only be evaluated as correct or incorrect with respect to a specification—a logical formula that determines the set of correct programs. Much like the representation of the program being synthesized, the representation of the specification also depends on the problem domain. Both Gulwani [18] and Heule et al. [23] use input-output pairs as their specification—a synthesized program is then judged to be correct if its behaviour matches all the examples. Srinivasan and Reps [41] use bit-vector formulae to fully describe the desired behaviour of a sequence of machine-code instructions (note that the program representation in one application can be treated as the specification in another).

Oracle Guidance The string processing examples used by Gulwani [18] as a specification are given by human users to describe their desired behaviour, while Heule et al. [23] observe the behaviour of individual machine code instructions to collect their examples. This second approach treats each instruction as an *oracle*—an interface that can be queried to produce a partial specification (in this case, input-output examples).

Jha and Seshia [25] formalise the role of oracles in program synthesis, giving definitions and analyses for the different types of response that an oracle can provide to a synthesiser (for example, providing new input-output pairs, evaluating whether a given output is correct for an input, or checking correctness for a whole program). Typically, oracle-guidance is used in situations where a synthesized program aims to match the behaviour of a 'black box' whose internal structure is not known. Knowledge of structure generally allows for more

sophisticated specifications to be given.

3.1.2 Challenges and Difficulties

At the core of program synthesis research is the difficulty of identifying a correct solution from an intractably large space of potential programs. Exhaustive searches are not practical, purely because the number of solutions to evaluate is too large—as a consequence of this, there is a focus in synthesis research on intelligent reduction of search spaces. For example, Solar-Lezama [39] reduce their search space by having the user specify a partial structure for the program being synthesized. Heule et al. [23] do so by starting with a small set of target programs, then expressing later programs in terms of the solution to earlier ones. In general, this type of search space reduction is important to ensure that reasonably large programs can be synthesized.

There are a number of different approaches to the process of actually synthesizing candidate programs. Some approaches rely on being able to encode their program structure so that it can be understood by an "off the shelf" logical solver. For example, Godefroid and Taly [17] use an SMT (satisfiability modulo theories) solver to check the correctness of their candidate programs. This approach relies on being able to encode the behaviour of a candidate as an SMT problem, but has the advantage that solutions can be conclusively proven correct. Others synthesize programs where a formal semantic model cannot be obtained, and so correctness must be checked by executing the program on test inputs. While this technique cannot give concrete proof of correctness, it is conceptually simple and can provide 'good enough' assurances when proof is not possible. Schkufza, Sharma, and Aiken [38] synthesise optimal machine code programs, then evaluate correctness using a novel method for comparing target register values to the results produced by the synthesized program. A logical encoding of the programs they synthesise is not practical because of the complexity of the instructions used to construct their candidate programs.

Random sampling methods are a common approach to the task of generating candidate programs, and weighting the probability distribution that programs are drawn from can be seen as another way of reducing the search space. Schkufza, Sharma, and Aiken [38] use Monte Carlo sampling methods to sample programs that are closer to a correct solution more frequently, while Phothilimthana et al. [36] run a stochastic search in parallel to an enumerative one. With little *a priori* knowledge of what a candidate program looks like, random sampling is an effective way of finding candidate

programs.

3.1.3 Summary

Program synthesis can be used to automatically produce correct solutions to a computational problem, or to extract a semantic model for a 'black box' where no internal structure is known. It can be applied to a number of different research domains, such as text manipulation, superoptimisation of binary code, and discovery of formal semantics. The large search spaces involved in program synthesis means that random sampling methods are common, along with techniques for reducing the search space of candidate programs.

3.2 LLVM

In Chapter 5 we describe the implementation of AccSynt, a program synthesizer that can be used to learn the behaviour of accelerators using input-output queries to their interface. The programs AccSynt synthesizes are functions expressed in LLVM [28] intermediate representation (IR). In this section we give a brief overview of LLVM and why AccSynt uses it as a target for synthesis.

Traditional compilers commonly represent their intermediate code using single static assignment (SSA) form. In this form, variables are assigned to exactly once, and are therefore immutable once assigned. This makes the code more amenable to certain analyses. LLVM IR is in SSA form, and replaces all high-level control flow with conditional tests and branches between basic blocks (linear regions of code a single entry and single exit point). However, type information such as aggregate structures and integer bitwidths is retained in the IR. As a result LLVM IR occupies a conceptual space between C and assembly code—the structure is simple enough to facilitate machine analysis, but enough information is retained that it is easily understood by humans. Figure 3.1 compares a simple C function with LLVM IR it could reasonably be compiled to.

LLVM is a large open-source project with excellent support for generating and manipulating IR programs, as well as for just-in-time (JIT) compilation. The original purpose of this was to facilitate compiler optimisations and JIT backends for programming languages, but these features are also well suited for program synthesis tasks—programs can be constructed and executed without reinventing a target programming

```c
int func(int x, int y)
{
  if(x < y) {
    return x;
  } else {
    return y + x;
  }
}
```

compile →

```
define i32 @func(i32 %x, i32 %y) {
  %cond = icmp slt i32 %x, %y
  br i1 %cond, label %if_t,
               label %if_f
if_t:
  ret i32 %x
if_f:
  %sum = add i32 %x, %y
  ret i32 %sum
}
```

Figure 3.1: Comparison between C code and equivalent LLVM IR—all values are named explicitly in SSA form and control flow has been lowered to conditional tests and branches.

language and runtime. For this reason, AccSynt targets LLVM IR with its generated programs.

3.3 Sparse Linear Algebra

Linear algebra workloads are commonly targeted by accelerators. This is due in part to their ubiquity across a variety of different problem domains—for example, scientific simulation, graph algorithms and machine learning all make use of linear algebra. In Chapter 5 we describe work done benchmarking different accelerators and building a predictive performance model for sparse linear algebra workloads. This section provides a brief introduction to these methods.

A *sparse* matrix is one where a large proportion of elements are zero. Storing such a matrix in a traditional dense format entails a lot of redundancy—sparse formats only store the non-zero elements of the matrix together with information about their location, allowing for more compact storage if there are very few non-zeros.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 6 & 8 & 0 \\ 3 & 1 & 6 & 8 \\ 3 & 0 & 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix}$$

Figure 3.2: Two 4×4 matrices—one dense and one sparse.

3.3.1 Storage

There are several different formats in which a sparse matrix can be stored, each with their own advantages and disadvantages. Perhaps the simplest of these is these is the coordinate format (**COO**), where the non-zero entries in the matrix are stored as triples $(row, column, value)$. For example, the sparse matrix in Figure 3.2 would be represented (using zero-based indexing) as:

$$\mathbf{COO} = [(0,2,1),(1,2,2),(1,3,3),(3,0,4)].$$

The coordinate format is most commonly used for input and output of matrices rather than as an efficient representation for linear algebra workloads—it is easily understood by users when represented textually and can be converted to other formats as required.

Matrix-vector multiplication is one of the most common linear algebra routines, and so a sparse matrix format that permits an efficient implementation of this is desirable. The compressed sparse row (**CSR**) meets this requirement by storing non-zero elements as a single contiguous array **A**, together with an array **JA** giving the column index of each non-zero element, and an array **IA** that stores a cumulative sum of the number of non-zeros in each row. Again referring to the sparse matrix in Figure 3.2, these arrays are:

$$\mathbf{A} = [1,2,3,4]$$
$$\mathbf{IA} = [0,1,3,3,4]$$
$$\mathbf{JA} = [2,2,3,0]$$

The first element of **IA** is always 0, and each successive element satisfies:

$$\mathbf{IA}[i] = \mathbf{IA}[i-1] + nnzInRow(i-1).$$

As a consequence of this, pairs of elements in **IA** give ranges in **A** and **JA** in which elements in a particular row can be found. **A** and **JA** both have size equal to the number of non-zero elements, while **IA** has one more element than the number of rows in the matrix.

Variations on CSR exist, such as blocked compressed sparse row storage (**BCSR**) which stores dense blocks of elements rather than individual non-zeros. This can result in higher performance for some problem domains, but is less commonly used than regular CSR. Dual to CSR is the compressed sparse column format (**CSC**) which stores non-zero elements in column-major order.

3.3.2 Matrix-Vector Multiplication

Sparse matrix-vector multiplication (SPMV) is one of the most common sparse linear algebra tasks. In Chapter 5 we describe a series of experiments investigating the performance of SPMV using different accelerators in the CSR format—in this section we introduce the baseline sequential implementation of SPMV for a CSR matrix for the purposes of comparison.

```
1  void csr_spmv(double *out_vec,
2                double *A,        // non-zero values
3                double *in_vec,
4                int *IA,          // row ranges
5                int *JA,          // column indexes
6                int n_rows)
7  {
8    for(int i = 0; i < n_rows; ++i) {
9      // Outer loop over all rows in matrix
10     double sum = 0.0;
11     for(int j = IA[i]; j < IA[i+1]; ++j) {
12       // Inner loop over non-zeros in row i
13       // JA[j] is the column index of the current non-zero
14       sum += A[j] * in_vec[JA[j];
15     }
16     out_vec[i] = sum;
17   }
18 }
```

Figure 3.3: C implementation of SPMV for a CSR matrix and dense vector.

Figure 3.3 shows how SPMV can be implemented in the CSR format using the definitions above. This algorithm is efficient because the non-zero elements in the matrix are stored contiguously, and are iterated over in order. Later sections will refer back to this implementation as a performance baseline that accelerated implementations can be compared to.

Chapter 4

Case Study

In Chapter 1 we identified the problem statement for this dissertation: we aim to automatically learn behavioural models for accelerators, such that the models can then be used with existing tooling to discover code that can be mapped to an accelerator. An ideal use-case for our work in this dissertation is to generalise learned programs to constraints that can then be used to discover and map code onto accelerators. While this process of generalisation is future work (see Chapter 7), learning programs as models for accelerators is one step towards an end-to-end solution for matching arbitrary code with arbitrary accelerators. Figure 4.1 shows the structure of an end-to-end solution.

To motivate an end-to-end solution, it is necessary to show that performance can indeed be improved by using an accelerator for a given problem—we want to know whether the difficult process of automatically matching code to accelerators is actually worthwhile. If it is, then learning the behaviour of that accelerator is a useful end result. As an example problem to be studied in this way, we considered sparse matrix-vector multiplication. This choice was made for a number of reasons: there are many different accelerator implementations available, it is a common performance bottleneck in scientific code, and the code structure in SPMV is similar to many other linear algebra problems, helping to generalise a program synthesis based solution.

This chapter describes our experimental investigation into the potential performance improvements available on sparse linear algebra workloads. We conducted a survey of publicly available scientific source code, finding that sparse matrix-vector multiplication is a common idiom in physical simulation code. Using the discovered code, we construct a set of benchmarks that can be used to evaluate the performance of an implementation of sparse matrix-vector multiplication. The results from these benchmarks show speedup of up to 50$\times$ compared to a naïve sequential implemen-

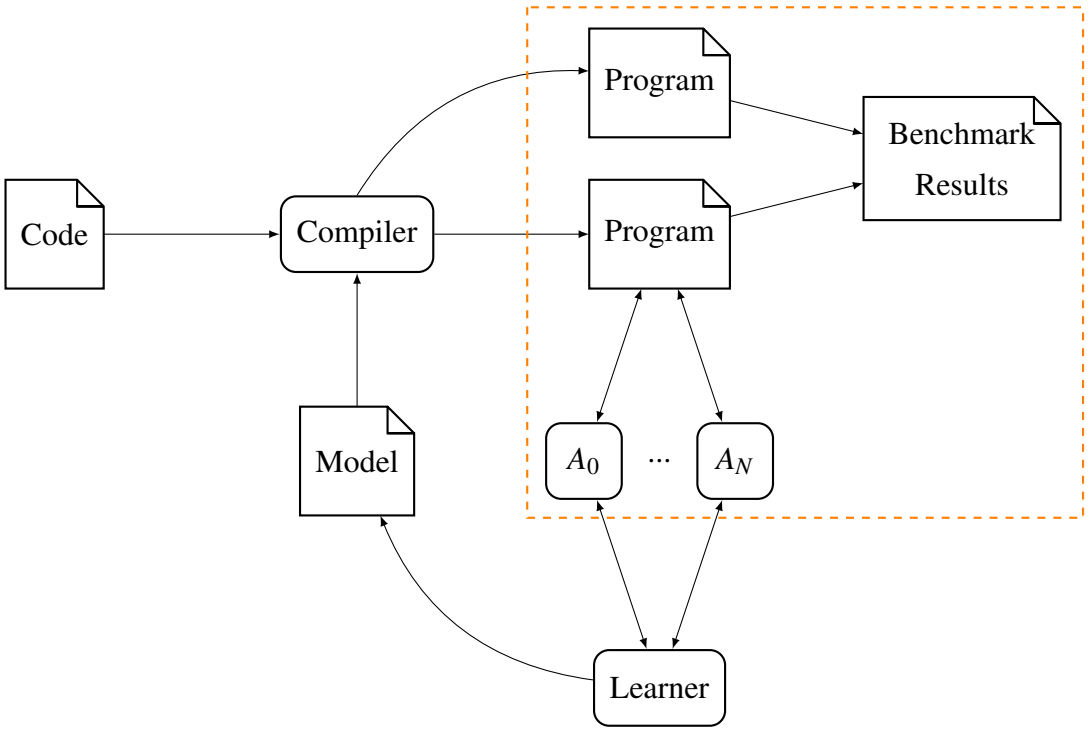

Figure 4.1: The compiler can produce multiple equivalent versions of the same program—for example, here it produces one that makes calls to accelerators, and one that does not.

tation. However, for some instances, using an accelerator causes performance to decrease. To address this problem, we demonstrate a machine learning model that can be used to accurately select the best implementation available on a platform. The section highlighted in orange in Figure 4.1 shows the work performed in this chapter in the context of an end-to-end solution.

In Section 4.1 we discuss the prevalence of SPMV and similar algorithms in real-world software and how they can be discovered, and in Section 4.2 we give details of the benchmark suite for SPMV we assembled to evaluate accelerated implementations, with results given in Section 4.3. Finally, in Section 4.4 we discuss our predictive model for choosing an appropriate implementation of SPMV at runtime.

4.1 Finding SPMV in Code

Finding instances of source code that implement a particular computation is not easy. A great deal of source code is never made freely available online, and the code that is available is often not in a centralised, searchable location. This (at least anecdotally)

is especially true for research-oriented software from the physical sciences, meaning that our search for instances of SPMV in real code was a frustratingly manual search. LLVM-based tooling for idiom search due to Ginsbach and O'Boyle [15] was a useful aid to this process, but the many inconsistencies between build systems and the prevalence of Fortran code limited its applicability.[1]

Discussion with colleagues associated with the Wales group[2] at the University of Cambridge suggested that chemical simulation and molecular dynamics software would be a fruitful subdomain to focus our search on. By manually examining freely available source code in this field, we were able to find a number of packages that implemented SPMV operations in their internal routines. Of these, the most common format used was CSR, with some using CSC, BCSR or other variations. For the purposes of this case study we considered only those using CSR. Then, from the CSR workloads, we found two that we could easily and reproducibly benchmark against real data. These came from the Wales group's energy minimisation application, PATHSAMPLE [43, 44].

While the scope of this case study meant that we only identified two easily reproducible benchmarks from real world code, there are many more potential instances that could be similarly evaluated given more time. Further guidance from experienced users of identified libraries would give more real-world workloads to run against, and application to more sparse formats would provide further examples we identified but was not able to analyse so far.

4.2 SPMV Benchmark Suite

Based on our work identifying SPMV instances in real-world chemical simulation code, we assembled a benchmark suite comprising five different instances. Each benchmark is an instance of SPMV discovered in a code base, where the core multiplication code was identified and replaced by calls to a harness that wrapped the different accelerator implementations we tested. The tool performing the automated search and replacement of code was developed by Ginsbach et al. [16]. A brief description of each benchmark is as follows:

Wales-PFold The PATHSAMPLE application, running a workload that simulates interactions between a large group of atoms. This benchmark uses options that

[1] Fortran code cannot be as easily compiled to LLVM IR as C or C++ code can.

[2] `https://www.ch.cam.ac.uk/group/wales/index`

disable an optional preprocessing step, which causes the SPMV computations to be a significant bottleneck.

Wales-NGT A different workload for PATHSAMPLE that requires a preprocessing step to be applied to the system before the core SPMV loop is run, meaning that SPMV instances are less of a bottleneck here.

Netlib The Netlib benchmarks [11] were developed by to test the performance of systems performing sparse linear algebra workloads. This particular benchmark is a Fortran implementation of SPMV.

Netlib-C Similar to the Netlib benchmark, but implemented in C rather than Fortran.

NAS-CG We use the conjugate gradient benchmark from the NAS parallel benchmark suite [4]. This benchmark performs SPMV operations to solve an iterative system.

Each of the five benchmarks described above can be compiled against a shared library containing the function `spmv_harness_` shown in Figure 4.2, then run to produce a performance result.

```
1 void spmv_harness_(double *out, double *A, double *in,
2                    int *IA, int *JA, int *n_rows);
```

Figure 4.2: Interface to accelerated SPMV implementations.

4.3 Benchmark Results

This section specifies the platforms and accelerator implementations we used to evaluate SPMV performance on our benchmark suite. We then show the performance results for these accelerators, which show clearly that significant improvements are available on SPMV workloads.

4.3.1 Experimental Setup

We compared benchmark performance across four accelerated implementations on two platforms. The two platforms were:

Intel Dual 16-core Intel Xeon E5-2620 processors with an Nvidia Titan X GPU.

AMD Single quad-core AMD A10-7850K processor with an AMD Radeon R7 integrated GPU and an Nvidia Titan X GPU.

On these two platforms, the accelerated implementations used were:

Intel-MKL Optimised mathematical libraries provided by Intel, run on the CPU using 8 hardware threads.

Intel-GPU Written using the cuSPARSE library for sparse linear algebra, and run on the external GPU.

AMD-integrated Using the clSPARSE library for sparse linear algebra, run on the integrated GPU.

AMD-GPU Also using clSPARSE, but run on the external GPU.

Additionally, each platform ran the sequential implementation of SPMV shown in Figure 3.3 as a baseline for performance comparisons. Results were obtained by running each benchmark, then recording either the time reported by the benchmark application or the total wall clock execution time as appropriate. Results were recorded 10 times per benchmark.

4.3.2 Results

The results from running our benchmark suite are shown in Figure 4.3. It is clear that large performance improvements are possible across a variety of benchmarks and platforms—up to 55× for the NAS-CG benchmark being run on the Intel platform's GPU.

However, it is also possible that switching to an accelerated implementation causes a decrease in performance. We see this on the Wales-PFold benchmark being run on the Intel platform's GPU. The problem here is due to more memory transfers being required between the GPU and the host than in the other benchmarks—on the AMD platform with the same GPU, the CPU is slower in comparison which compensates for this effect. Additionally, on the Wales-NGT benchmark the SPMV operations are less of a bottleneck, and so only modest speedups are observed.

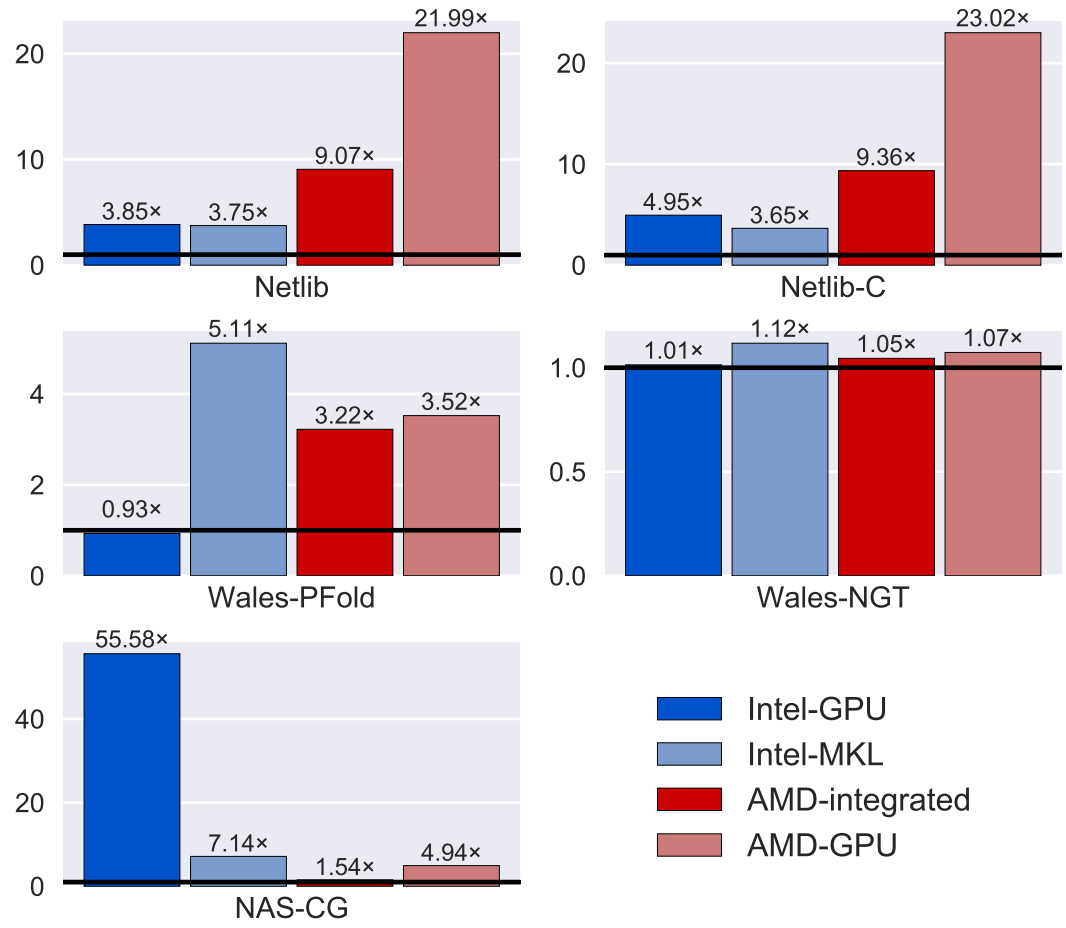

Figure 4.3: Speedup results for four accelerated implementations of sparse matrix-vector multiplication. Blue and red bars ran on the Intel and AMD platforms respectively, and all speedups are quoted relative to a sequential CPU implementation running on the same platform.

4.4 Predicting Performance

While the results in Figure 4.3 paint a positive picture for SPMV acceleration, the variance between the sizes of matrices used is small. All the scientific code and benchmark suites typically use large problem sizes, but to get a more accurate view of SPMV performance we require more data points.

To gather this data, we used the University of Florida matrix collection [10]—the number of non-zero elements in its matrices ranges from one to over 100,000,000. We developed a synthetic benchmark for this dataset that loads a matrix from a file, then performs iterated SPMV on it to measure performance. Running this benchmark over all matrices in the collection shows that not all SPMV problems run faster using

an accelerator.[3] To alleviate this problem, we developed a predictive model that can accurately select the best possible SPMV implementation at run-time.

The code in Figure 3.3 is a good approximation for CSR SPMV implementations found in code. Its running time depends on the number of rows in the matrix, as well as the number of non-zeros. It is therefore a reasonable starting point to assume that these values can be used to predict the potential speedup available by using an accelerator. Additionally, the values are available directly from the arguments passed to the SPMV interface—no additional computation is required to extract the features.

We trained a linear support vector machine (SVM) model [7] on the sparse matrix benchmark results, using the number of rows and number of non-zeros as features and the best implementation label as the output. The result of this training was a model that when given a pair $(rows, nnz)$ as input, returns an implementation label (either sequential, integrated GPU or external GPU) corresponding to the predicted best implementation. The model was trained on 2,290 examples, and tested on a disjoint set of 255 examples, with a test set accuracy of 99.8%.

To evaluate the model, we added a runtime check to our SPMV benchmark implementation that calls the predicted best accelerator after a prediction is made. We then re-ran the set of benchmarks for the AMD platform with this test enabled—the metrics we examined for comparison were the number of matrices that experienced a slowdown (i.e. the implementation chosen is not optimal), the mean speedup over the whole dataset, and the mean slowdown over instances that were slowed down. Our results showed noisy results for very small matrices, caused by the execution time of the model itself. To resolve this effect we added a manual heuristic to the model that chooses the sequential implementation when there are less than 500 non-zero elements in the matrix.

The results are shown in Table 4.1. Without using the predictive model, the external GPU is always selected as the best implementation. This causes a large proportion of the benchmark to perform worse than the sequential implementation, with an average slowdown of $0.165\times$. Using the predictive model improves on this performance dramatically—far fewer instances are slowed down, and those that are are slowed down far less than before. The expected speedup over the whole dataset is also higher. Finally, we see that the manual heuristic improves performance again, but less dramatically so. The SVM+heuristic model achieves a mean speedup within 1.5% of that of perfect oracle for the same prediction—while close to 40% of instances are predicted

[3] Using the AMD platform only due to resource availability issues beyond our control.

Model	Accuracy	Speedup	Slowdown
None	34.2%	1.048×	0.165×
SVM	59.8%	1.595×	0.934×
SVM+Heuristic	61.2%	1.619×	0.946×
Oracle	100.0%	1.640×	1.000×

Table 4.1: Performance results for SPMV workloads when using a predictive model to select the best implementation at runtime. Speedup is calculated over the whole dataset, and slowdown only over the matrices that exhibited a slowdown. The oracle results refer to an ideal scenario in which no instances are ever slowed down, and its results are the ideal limiting values for the other models.

incorrectly, they are slowed down so little that the overall effect is not significant. This effect is largely because the decision boundary is very noisy.

4.5 Summary

In this chapter we have described a case study examining to what extent accelerators can be used to improve the performance of SPMV workloads. Over a collection of benchmarks comprising real-world scientific code, well-accepted benchmark suites and a synthetic benchmark over publicly available datasets, we have shown that accelerated implementations of SPMV offer very large performance increases over sequential implementations. However, these increases do not apply equally to all datasets—to address this, we developed a predictive model that is able to accurately select the best accelerator implementation at runtime.

An interesting direction for future work would be to examine other accelerator implementations—currently, the choices available on each platform are somewhat limited. Autotuning approaches such as SparseX [12] or POSKI [8] are promising, and may improve performance for more matrix instances. This would potentially also require reworking the predictive model to perform better on more classes.

Chapter 5

Implementation

In Chapter 1 we introduced the core problem addressed by this dissertation: in order to match arbitrary user code to arbitrary accelerators, we need a way of automatically learning a model of accelerator behaviour. This model can then be consumed by existing tools to discover and match code to the available accelerators. Additionally, we proposed the use of program synthesis to solve this problem. Then, Chapter 4 showed an experimental evaluation of the performance available by using linear algebra accelerators—the results from these experiments show promising performance improvements. Because of this, implementing a program synthesis based tool to automatically learn accelerator behaviour is a useful step towards the end-to-end solution shown in Figure 1.1.

This chapter discusses the implementation of a framework for performing oracle-guided program synthesis; it allows for different methods of program synthesis to be implemented easily by abstracting common details, and we use it to implement a program synthesizer for synthesizing looping programs based on user annotations. The synthesizer (AccSynt) can be used to discover an executable model for an accelerator's behaviour, with the aim of then being able to discover similar instances in a code base.

This chapter is laid out as follows: in Section 5.1 we describe the underlying technical detail in my synthesizer framework. Then, in Section 5.2 we go on to give details of how we used this framework to build AccSynt, a program synthesizer for automatically learning the behaviour of accelerators.

5.1 Framework

This section describes the implementation of a framework for developing program synthesizers that target LLVM intermediate representation. We describe the conceptual interface that a synthesizer developed in this framework should satisfy, then go on to provide technical details implemented by the framework that allow for synthesizers to be developed more easily.

5.1.1 Synthesizer Interface

It is worth examining exactly what is meant by "synthesizer" in the context of this framework—that is, what end product should a programmer be able to develop by using it, and what customisation points are available to them to do so? The stated goal of my project is to develop a synthesizer that can synthesize an executable program that is behaviourally equivalent to an accelerator—design decisions about the framework are therefore made with respect to this goal.

For an arbitrary accelerator, if all we know is how to make calls to it (i.e. we do not know any of its internal implementation details), the only practical way to decide whether a program is equivalent to it or not is to evaluate both over a large set of example inputs. If no inputs lead to different outputs, then we can be reasonably certain that the program is a model for the accelerator.

This means that the developer of a synthesizer does not have to provide a method of deciding equality between the programs they generate and the accelerator—it can be automatically implemented by the framework. This leaves the primary customisation point for the developer as the generation of individual candidate programs. These candidates can then be compiled and tested automatically by the framework.

To generate a candidate program, the developer implements a construction method that takes an LLVM IR function as input. This function is then populated with instructions by a synthesis algorithm, and can then be checked for correctness by the framework. A simplified version of this interface is shown in Figure 5.1.

5.1.2 Technical Details

Any implementation of the interface in Figure 5.1 can be used with the synthesis framework to generate programs that are automatically tested for correctness against a large set of example inputs. The framework handles several implementation details that

```
1 struct CustomSynth : Synthesizer {
2   void construct(llvm::Function *F, llvm::IRBuilder<> &B) const override;
3 };
```

Figure 5.1: C++ interface for a custom synthesis algorithm compatible with my framework.

make the development of synthesizers easier:

LLVM Compilation The framework provides code that treats LLVM functions as if they were native C++ code. There is a lot of repetitive boilerplate code associated with JIT compilation and execution of LLVM IR at runtime, and wrapping functions as C++ callable objects allows for more generic algorithms to be written. Additionally, ownership and object copying between the code generation and execution phases is complex and easy to get wrong.

Error Handling Convention It is possible for synthesized programs to terminate with an error of some kind. The synthesis framework allows for this by adding an extra output parameter to every synthesized function that can be set to a particular value to signal an error. This means that errors can be signalled out-of-band from actual return values through a simple interface. The LLVM function wrappers described above handle these errors transparently by throwing an exception if the error value is set.

The framework also contains an LLVM optimisation pass that removes error-handling code from synthesized programs—this is useful because the signature of generated code that handles errors is different to the accelerator's interface.

Handling errors in this way means that developers can report errors within their synthesized programs in a standard way using a simple interface. In AccSynt, we use this mechanism to perform bounds checking for code that accesses memory.

Multithreading Using LLVM's code generation features in a multi-threaded context is possible, but difficult to implement properly. The synthesis framework handles multithreaded code generation and compilation appropriately, meaning that developers can write their synthesis algorithms without explicitly considering how to parallelise them safely.

Some algorithms might require internal multithreading or parallelism, and to

support these cases the framework allows access to the underlying LLVM threading contexts so that code generation can still be done safely.

Allowing multithreading in this way means that the same synthesizer can be easily scaled up to run on a multi-core machine without rewriting the code generator.

Correctness Checks As described above, the use case for this framework is writing synthesizers that are compared against the behaviour of accelerators for correctness. This means that the primary method of checking their correctness is by comparing input-output behaviour against a large set of inputs.

The framework handles this checking for synthesizers, which are able to access the set of examples used for checking correctness (if the examples are relevant for code generation).

5.1.3 Summary

Writing a program synthesizer entails a great deal of boilerplate code that must be implemented before any actual synthesis can be performed. In this section we have given a brief overview of some examples of where this boilerplate occurs, and described the implementation of a framework that abstracts it away. This allows for new program synthesizers to be written more easily—the only customisation point that needs to be implemented by a developer is a method that constructs a candidate program.

5.2 AccSynt

In Section 5.1 we described the boilerplate code that we implemented as part of a framework for developing program synthesizers. This framework offers a single customisation point—a method that generates a single candidate program based on the set of input-output examples. This section describes our implementation of AccSynt, a program synthesizer that aims to synthesize programs that are behaviourally equivalent to an accelerator. In particular, it initially targets accelerators that perform linear algebra and nested reduction operations. However, this is largely due to time restrictions during development—in further work we hope to expand the scope of accelerators targeted.

The case study in Chapter 4 examines sparse matrix-vector multiplication as a candidate for acceleration—it can be found in real-world code in the form of hand-coded implementations, and significant performance gains are available by choosing an appropriate accelerator to run the multiplications on. We therefore identified matrix-vector multiplication as an initial running example to work towards—Chapter 6 addresses this bias and discusses how the range of programs targeted can be improved.

In the remainder of this section we describe how AccSynt synthesizes linear algebra programs, and the assumptions its implementation makes to do so. The two most important ideas that allow this are adding "human" information to a type signature in order to better describe program intent, and enumerating possible loop structure given a set of iterators. We describe these ideas in more detail in Section 5.2.1 and Section 5.2.2 respectively.

5.2.1 Parameter Annotations

Synthesizers developed using our framework check correctness by comparing two callable objects with respect to their input-output behaviour—this means that the accelerators targeted should have a C-like function call interface.

```
1 void operation(double *, int, double *, double *);
```

Figure 5.2: C type signature for an operation where we do not know the semantics of any of the arguments.

For the purposes of program synthesis, the type signature alone of an accelerator interface is a very weak prior on the space of candidate programs—we have no knowledge of the semantics associated with any of the arguments. This means that we cannot make any assumptions about the structure of synthesised programs, which leads to an intractably large search space.

However, programmers often know informal information about the arguments to such a call site—for example, they may know that one parameter serves as an output, while another represents a size. These facts are often present in documentation, and can be known even without knowing what the call site's overall behaviour is. The code in Figure 5.2 shows a call site without semantic annotations, while Figure 5.3 has semantic annotations. From these annotations we can restrict the space of possible programs with this signature—for example, any program that stores to `b` or `c` or accesses

```
void operation(
  double * a{output},
  int s,
  double * b{size = s},
  double * c{size = s}
);
```

Figure 5.3: C type signature for an operation with semantic annotations on its parameters. The code between braces after each parameter name is a hypothetical syntax for these annotations—they mark the first parameter as being an output, and the last two as having size `s`.

them at indexes greater than `s` will be incorrect.

For these parameter annotations, it is important to strike a balance between giving useful information to the synthesizer and being difficult for the programmer to work out. If they have too much information, then the usefulness of the synthesis process is decreased. If they have too little, then it may not be possible to perform the synthesis.

AccSynt implements these tags by means of a small "type system" that encodes both the C types and parameter annotations for a signature. The user of a synthesiser then supplies a signature in this new type system along with the callable interface to the accelerator. In Figure 5.4 we give a simplified specification for the tags of interest to us when synthesizing linear algebra programs—it allows integer and double values, and pointers to those values. Pointer types can be annotated as being outputs, or as having a compile- or run-time size specified by a constant value or another argument respectively.

This system of tagging types can be easily extended to more complex signatures (for example, structures or variadic functions), but for the purposes of this chapter the definition given is sufficient. Similarly, any information easily obtainable by the programmer can and should be included in new tags.

5.2.2 Loops

At the core of any linear algebra routine is an iteration or loop of some kind over the data in a matrix or vector. In order to learn the behaviour of accelerators that perform linear algebra workloads, a way of synthesizing looping programs is required. This is not a commonly studied program synthesis task, and so in Section 2.3 we examine

⟨*size*⟩ ::= *non-negative constant integer*

⟨*param*⟩ ::= *name of one of the function arguments*

⟨*base-type*⟩ ::= integer
| double

⟨*aggregate-type*⟩ ::= pointer ⟨*base-type*⟩

⟨*sized-type*⟩ ::= fixed-size ⟨*size*⟩ ⟨*aggregate-type*⟩
| sized ⟨*param*⟩ ⟨*aggregate-type*⟩

⟨*type*⟩ ::= ⟨*base-type*⟩
| ⟨*aggregate-type*⟩
| ⟨*sized-type*⟩
| output ⟨*sized-type*⟩

Figure 5.4: Specification for tagged type constructors used in annotated type signatures.

similar work to put the techniques from this section into context.

The key idea that underpins our synthesis of loops is that if one of an interface's arguments is a pointer whose size is known, it is far more likely than not that the computation can be expressed as a loop involving the size. This is the first assumption that AccSynt makes—if there is a data parameter with a known size, it should try to synthesize a loop over it. An example of what this loop looks like is shown in Figure 5.5.

```
1 double func(
2   int x,
3   double* a{size = x}
4 );
```

synthesize ⟶

```
1 double func(int x, double *a)
2 {
3     for(int i = 0; i < x; ++i) {
4       // use a[i]
5     }
6 }
```

Figure 5.5: Example of how AccSynt synthesizes a loop skeleton from a data parameter with a known size. For clarity, this listing uses C as the synthesis target rather than LLVM.

The synthesis process is simple when there is only a single data parameter with a

known size—there is only one possible loop over all the data elements. When there are two or more parameters with known sizes, the process is more complex. There are multiple different ways in which we can iterate over several objects of a known size—for example, we could iterate over one then the other, or we could nest the iteration over one inside the other's iteration. These two cases are shown in Figure 5.6.

```
double func(
  int x,
  int y,
  double* a{size = x},
  double* b{size = y}
);
```

synthesize →

```
double func_1(int x, int y,
              double *a, double *b)
{
    for(int i = 0; i < x; ++i) { ... }
    for(int j = 0; j < y; ++j) { ... }
}

double func_2(int x, int y,
              double *a, double *b)
{
    for(int i = 0; i < x; ++i) {
      for(int j = 0; j < y; ++j) { ... }
    }
}
```

Figure 5.6: An example of how AccSynt can synthesize multiple loops over data—either one after the other or nested.

As well as arranging the structure of the loops differently, the order in which the objects are iterated over can be varied as well. The approach taken by AccSynt to synthesizing these loops is combinatorial—it enumerates all the possible loop structures, then synthesizes a candidate with each structure in turn.

Generating potential loop structures is a two-step process. First, the potential 'shapes' for loops are generated. For N data objects, this corresponds to generating the set of all possible lists of n-ary trees with N total nodes in the list. This equivalence follows from the fact that a loop nest where the iterators are not yet known can be seen as a tree with unlabelled nodes, and it is possible to have multiple loops in sequence at the top level. Generating these lists can be done by a simple brute force algorithm as in practice, the value of N is very small. Figure 5.7 shows the five possible loop shapes when there are three iterators.

Once we have generated all the unlabeled loop shapes, we can easily enumerate all the labellings of these shapes with the iterators $i_1, \ldots, i_N$. Each of these labellings corresponds to a different control flow for the synthesized program—AccSynt stores

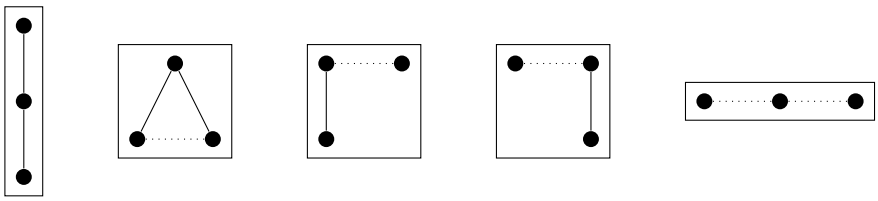

Figure 5.7: The five possible loop shapes (in the form of tree sequences) when there are three iterators. Solid lines indicate loop nesting, while dotted lines indicate sequences of loops.

all the possible structures, and synthesizes a program with each one in turn.

This loop technique will generate all the possible loops that iterate linearly over all the elements in each sized data pointer passed to the interface. However, the space of programs that this permits is somewhat restricted. Some common patterns that it cannot generate are:

- Multiple arrays of the same size are iterated simultaneously in the same loop.
- The size of pointed-to data is not known, or the information that encodes the size cannot be expressed using parameter annotations.
- Access to data is a function of the iterator variable (e.g. accessing data at index `i*j + k` inside a nested loop).

AccSynt accounts for the first case given above by adding simultaneous loops over identically-sized data to the set of structures it can generate. The second and third cases are handled by synthesizing index expressions inside loop bodies, then using them to index into unsized data. Full details of the methods used to perform loop body synthesis are given in the next section.

5.2.3 Synthesis

Previously in this section we described how parameter annotations can be used to convey programmer intent better than C types alone can, as well as how knowledge of data sizes can be used to synthesize the structure of regular loops over that data. Based on this, we now describe how actual programs can be synthesized using these techniques. The steps taken by AccSynt to generate a single candidate program which can be compiled and tested by the synthesis framework are given below, along with example code where appropriate:

1. Generate boilerplate common to all generated functions. This means constructing a function with the appropriate parameter and return types, along with entry and exit blocks that set up a return value for the function. The code below shows a minimal example of this:

```
define i64 @candidate(i64* %err, i64, double*) {
entry:
    %return_loc = alloca i64, align 4
    store i64 0, i64* %return_loc
    br label %exit
exit:
    %return = load i64, i64* %return_loc
    ret i64 %return
}
```

2. Based on the loop structure algorithm above, create the control flow code for the chosen loops. Every loop has four components: header, exit, and two halves of its body—if a loop has child loops, they are constructed between the two halves of its body, which allows setup and teardown code to be synthesized before and after child loops are run.

   The loop induction variable is set up at this point, and at the end of the body it is compared to the loop upper bound (which is known from parameter annotations). Abbreviated control flow for a single loop with no children is shown in the code sample below:

```
define i64 @candidate(i64* %err, i64, double*) {
...
header:
    br label %body
body:
    %iter = phi i64 [0, %header], [%nextit, %body]
    %nextit = add i64 %iter, 1
    %cond = icmp eq i64 %nextit, %upper
    br i1 %cond, label %loop-exit, label %body
loop-exit:
    br label %exit
...
}
```

3. Once the loop structure has been built, the next synthesis step is to load data from pointers, using indexes based on the loop iterator. In every loop we have a pointer being iterated over directly, and we load the data at the current index into this pointer in every loop. Then, for pointers with unknown sizes, we synthesise a new index and perform a load of that data as well.

   AccSynt synthesises new indexes by generating arithmetic combinations of loop indexes and constant values in the program. For example, in a nest of two loops, the inner loop would have both the inner and other indices available to combine with constant values. Addition, subtraction and multiplication of indices with constants are all supported operations. The code below shows an example of the loads performed in a loop body:

```
define i64 @candidate(i64* %err, i64, double* %data, double* %otherdata) {
...
body:
    %data_ptr = getelementptr double, double* %data, double %iter
    %val_0 = load double, double* %data_ptr ; direct load from index
    %idx = add i64 %iter, %other_iter ; synthesize new index
    %other_ptr = getelementptr double, double* %other, double %idx
    %val_1 = load double, double* %other_ptr ; computed index load
...
}
```

4. Loop bodies now perform loads of data at several different indices. However, because the synthesised program will operate on real memory, bounds checking is required to ensure memory safety. We do this by checking loads to sized pointers against their known size, and loads to unsized pointers against a large fixed size—the synthesis framework ensures that memory passed to synthesized programs is always valid up to this maximum size.

   Bounds checking is performed by adding tests to every computed index to ensure they do not exceed the maximum valid size for the relevant pointer. If they do, a branch to an error handling block is made—this block sets the error output and returns from the function. Shown below is an example of this bounds checking and error handling.

```
define i64 @candidate(i64* %err, i64 %size, double*) {
...
body:
    %err_cond = icmp sgt i64 %iter, %size
    br i1 %err_cond, label %error_handler, label %body2
body2:
    ...
error-handler:
    store i64 1, i64* %err
    ret i64 0
...
}
```

5. At this stage, the function is set up to perform control flow but not to actually compute any results. The next stage is to populate basic blocks with instructions. We maintain a set of LLVM values that are 'available', and generate new instructions by combining available values. For example, we might generate a new value by adding two existing ones together. The operations chosen to combine instructions are randomly sampled from a set of possibilities.

   The set of available values starts with the values loaded from memory inside the loop bodies, together with the value of generated loop indices and some small constant values. Each basic block is then populated with instructions by combining live values (by adding, multiplying etc.). Values computed in a basic block are available in its successors, meaning that values can flow from parent loops into child loops, and out of a child loop into the second body block of its parent.

   Loops are able to track state between iterations by creating phi nodes in their bodies—these nodes are a construction used in SSA form that allow a variable to have a different value depending on how control flow reached it. These nodes are initialised to an available value when the block is populated, then their 'looping' value is selected from the set of instructions that reference the node. This prevents useless phi nodes from being constructed. The code below shows an example of generated code:

```
define i64 @candidate(i64* %err, i64 %size, double*) {
...
body:
    %phi1 = phi double [ 0.0, %entry ], [ %val1, %body ]
    %val0 = fadd double %load1, %load2
    %val1 = fadd double %val0, 1.0
    ...
...
}
```

6. Once the basic blocks have been populated with instructions, the final synthesis step is to create outputs—this means storing to parameters flagged as outputs, as well as storing a value to the function return location created at the start of the synthesis process.

   If an output parameter has a size associated with it, stored values are selected from the available values inside the loop over that parameter's elements. Similarly, the return value for the function is selected from all the available values in the final basic block. Example code performing output is shown below:

```
define i64 @candidate(i64* %err, i64, double*) {
...
body:
    %data_ptr = getelementptr double, double* %data, double %iter
    store double %data0, %data_ptr
    ...
exit:
    %return = load i64, i64* %return_loc
    ret i64 %return
}
```

These steps produce a function that can be called identically to the accelerator interface in question. Once the candidate function has been synthesised, it is passed back to the synthesis framework to be tested for behavioural equality. If it appears to be correct, a series of LLVM passes are applied to remove error handling code and remove redundancy (while still preserving the overall code structure to aid programmer understanding). This final version is returned to the user as a successful result.

5.2.4 Summary

In this section we have described the process by which AccSynt generates candidate programs during the synthesis process. To do this, it makes use of information in parameter annotations given by the user to synthesise loop control flow structures that are likely to yield correct programs. Synthesising control flow explicitly reduces the synthesis problem to a series of linear ones, which is a more tractable problem. In Chapter 6 we examine the success of these techniques.

Chapter 6

Evaluation

In Chapter 5 we described the implementation of a framework for easily building oracle-guided program synthesisers that generate LLVM IR, along with the design of AccSynt—a program synthesiser that aims to learn the behaviour of accelerators by synthesising programs that are behaviourally equivalent to them. This chapter evaluates the success of AccSynt with respect to several criteria, and is laid out as follows: first, in Section 6.1 we examine the expressive capability of AccSynt—that is, how rich is the space of potential programs it can synthesise? In Section 6.2 we discuss issues related to the performance of AccSynt, and show how potentially long synthesis times can be avoided during development.

6.1 Expressivity

No program synthesiser can synthesise every possible program, and so an important part of their evaluation is to establish the space of programs that can be synthesised. This section examines the expressivity of AccSynt with respect to its initial target of linear algebra accelerators. We apply AccSynt to a series of progressively more difficult problems in linear algebra—scalar sum, vector sum, dot product, dense matrix-vector multiplication and sparse matrix-vector multiplication. Using these problems as examples, we identify sources of complexity that AccSynt finds 'difficult' as well as some that are easy for it to learn. Finally, we give a summary of computational patterns that are not yet supported by AccSynt.

6.1.1 Linear Algebra Problems

In this subsection we give example implementations of five progressively more difficult linear algebra problems, and apply AccSynt to learning them.

Scalar Addition

```
1 int interface(int x, int y) {
2   return x + y;
3 }
```

This is perhaps the simplest possible task that AccSynt could be applied to—AccSynt is able to synthesise it trivially, and no parameter annotations are necessary (or even possible). Addition of two values is one of the primitive operations supported by the synthesis process, and the number of instructions generated in each candidate is small because no loops are required.

Vector Addition

```
1 void interface(int s, double* x, double* y, double* o) {
2   for(int i = 0; i < s; ++i) {
3     o[i] = x[i] + y[i];
4   }
5 }
```

AccSynt requires several parameter annotations to correctly synthesise this program—the output vector must be marked as such, as well as selecting `s` as the size for all three vectors.

With these annotations in place, AccSynt can easily synthesise this program. The code generated in the loop body is no more complex than the scalar addition example given above—because control flow is generated before loop bodies, it is just as likely to be synthesised as scalar addition.

Dot Product

```
double interface(int s, double *x, double *y) {
  double dot = 0.0;
  for(int i = 0; i < s; ++i) {
    dot += x[i] * y[i];
  }
  return dot;
}
```

This example is similar to vector addition, and uses the same parameter annotations. However, it is more complex because it requires state to be tracked between iterations. AccSynt can generate two distinct but equivalent programs for this example—one that performs a load and store of the function return value at each iteration, and one that stores the final value of a phi node after the loop has finished executing.

Once the synthesiser is able to track state between loop iterations in this way, the synthesis of the loop body is similar to the previous two examples.

Dense Matrix-Vector Multiplication

```
void interface(int r, int c, double* a, double* x, double* y) {
  for(int row = 0; row < r; ++row) {
    double sum = 0.0;
    for(int col = 0; col < c; ++col) {
      sum += a[row * c + col] * x[col];
    }
    y[row] = sum;
  }
}
```

Several issues with AccSynt were diagnosed and resolved while trying to synthesise this example—it is more complex in a number of ways than the previous programs.

The first of these is that the size of the matrix `a` cannot be given as a parameter annotation. This means that accesses to it happen only through synthesised indexes, rather than directly in a loop of its own. The index synthesis process described in Chapter 5 is able to generate the row-major indexing pattern because

the row and column iterators are both available in the inner loop, and the number of columns is an available constant.

The second complexity is that the program requires nested loops, with the inner loop tracking state between iterations—no previous example required nested loops.

However, despite being more complex, the inner loop's arithmetic operations are no more complex once the correct values are loaded from memory and made available to the synthesiser—the core operation is still a multiply-accumulate.

Sparse Matrix-Vector Multiplication

```
1  void interface(int rows, double* A, int* IA, int* JA,
2                 double* x, double* y) {
3    for(int row = 0; row < rows; ++row) {
4      double sum = 0.0;
5      for(int j = IA[row]; j < IA[row+1]; ++j) {
6        sum += a[j] * x[JA[j]];
7      }
8      y[row] = sum;
9    }
10 }
```

AccSynt as described could not synthesise this example. Its view of loop structure is not able to express the inner loop—it is not bounded by zero and a fixed value, but rather by values that depend on the current iteration. While the outer loop will be constructed easily, the inner loop will never be generated by AccSynt. To resolve this problem, we manually extended AccSynt with a set of features that would allow it to synthesise sparse matrix-vector multiplication. These features are:

- A new parameter annotation that is applied to parameters that contain indices—in the CSR format, the **IA** and **JA** arrays both contain indices that can be used to perform loads or bound loops.
- Indirect loads are performed inside loop bodies for these parameters—that is, one load is performed to get an index, which is then used to seed an index generation as described previously.

- A loop between two loaded indices is always synthesised to generate the inner loop in SPMV.

The modified synthesiser is able to synthesise sparse matrix-vector multiplication, but because the assumptions it makes are hard-coded into the synthesis process we have omitted it from the benchmarks and evaluation in this chapter. However, it serves as a useful proxy to demonstrate the potential expressivity of a more general synthesis process. As well as this, it closes the gap more neatly with the case study from Chapter 4—by learning a model for SPMV acceleration, the only step left in an end-to-end solution is to generalise from learned programs to constraint descriptions.

A general pattern that emerges from examining these examples is that the synthesis of loop control flow separately to loop bodies means that computations over scalar values can be efficiently lifted into loops. However, complex scalar operations increase the difficulty of synthesis for AccSynt dramatically—deeply nested arithmetic expressions are hard for it to synthesise, but nesting them inside a loop does not make them much harder.

It is worth noting that the expressivity of AccSynt does currently rely on parameter annotations from the programmer in order to guess loop structures. However, the amount of information given to the synthesiser is small and easy to provide based on documentation.

6.1.2 Unsupported Patterns

It is clear from examining the synthesis of dense and sparse matrix-vector multiplication that AccSynt's model of control flow is not adequate to express certain useful programs. In the case of sparse matrix-vector multiplication, the inner loop depends on values computed in its parent. There are other obvious examples of where AccSynt's control flow model is not adequate. For example, any computation that requires a conditional statement cannot be synthesized, or indeed any form of loop that is not a "for loop" in the form currently supported by AccSynt.

This suggests that a more general synthesis process would be a useful modification for AccSynt. Instead of hard-coding the ability to synthesize certain types of loop into AccSynt, it could sample from a library of control flow structures. This more flexible method would still be compatible with the current system of parameter annotations—instead of always generating nested loops, observing a sized parameter would bias the

component sampling towards those components (while still allowing other ones to be generated for flexibility).

6.2 Performance

Previously, we examined the expressive capability of AccSynt with respect to linear algebra problems of varying complexity. In this section we evaluate its performance on these problems—in particular, the time required to synthesise each example, and how this increases with the complexity of the program. As well as this, we discuss the use of stochastic synthesis methods and how they affect performance.

6.2.1 Synthesis Time

The time taken for AccSynt to successfully synthesise a program is an important metric for its usefulness. If it takes too long to learn an example successfully, then it cannot be applied easily during a development process. We created a number of example programs similar to the ones described previously. These examples perform computations involving scalar and vector arithmetic, with varying arithmetic expression depth.

Synthesis times for these problems are shown in Table 6.1. From these results we can see that increasing the number of loops required for a program does not dramatically increase the time taken to reach a solution. However, increasing the arithmetic complexity of the program and requiring synthesised indexes to access data both cause large increases in the synthesis time. This observation validates our observations from the previous section when examining the expressivity of AccSynt.

For all the benchmark results in Table 6.1, the variance in synthesis times is very high. This is because random sampling of instructions and loop structure is at the core of the synthesis process, meaning that accurately predicting synthesis performance is difficult.

The time taken to synthesise these programs is consistent with the use of AccSynt as an infrequently run tool—once a model for accelerator behaviour is discovered, it can then be reused. However, the use of stochastic methods makes it difficult to predict how long AccSynt will take to reach a solution.

Example	Loops	Expression Depth	Mean Synthesis Time (s)
`identity`	—	0	0.013 ± 0.011
`addition`	—	1	0.021 ± 0.013
`2d-dot`	—	3	4.20 ± 2.46
`vector-copy`	1	0	0.016 ± 0.002
`vector-add`	1	1	0.023 ± 0.006
`dot-product`	1	2	0.800 ± 0.478
`nested`	2	2	0.309 ± 0.158
`gemv`	2	3 + indexing	110 ± 52.2
`gemv-bias`	2	4 + indexing	323 ± 94.7
`spmv`	2	3	224 ± 38.3

Table 6.1: Time taken for AccSynt to synthesise a number of example programs. All synthesis was performed using 3 hardware threads for program generation, on an Intel i5-6500 CPU with 16GB of RAM. Statistics are given over 5 runs of each benchmark. Note that the results for the `spmv` benchmark are reported using the modifed synthesis method specific to SPMV synthesis.

6.2.2 Stochastic Methods

The use of stochastic methods and random sampling for program synthesis is common (we give a review of related techniques in Section 2.2). AccSynt achieves acceptable performance results by using a simple random sampling method to generate linear blocks of code. However, the performance and predictability of this technique require improvement so that more diverse programs with complex control flow structure can be synthesised.

When randomly sampling instructions to populate functions, AccSynt does not bias its search at all during or between iterations. This allowed the synthesiser to be developed easily, but means that all programs are generated with equal probability at each synthesis step. We propose two ways in which the search could be weighted towards generating correct programs:

Domain Similarity One potentially useful prior we can establish on generated programs is the problem domain their accelerator relates to. Given this domain information, a model could be built to describe the probability distribution of control flow or instructions in that domain.

This information is similar to that provided by parameter annotations, in that it is easily provided by the programmer without detailed knowledge of the accelerator's behaviour.

Building a model for the distribution of instructions or program structure would require a suitably large corpus of labelled programs that could be compiled and analysed.

Feedback Direction Currently, AccSynt only makes use of binary feedback for each candidate program (that is, whether it passes or fails). Making use of more detailed feedback such as a similarity metric on input-output example would allow for it to bias future programs to be more similar to partial successes already observed.

6.3 Summary

On its initial target of linear algebra workloads and accelerators, AccSynt is able to express useful programs as complex as dense matrix-vector multiplication, with synthesis times short enough on average to ensure practical application. However, sparse matrix-vector multiplication cannot yet be expressed by AccSynt. To overcome this limitation, we propose a more general approach to synthesising control flow structures. Within this more general approach, it would be possible to express the difficult inner loop from SPMV. Future work on AccSynt will focus on closing the gap between learned models and discovering potential sources of acceleration, as well as on applying program synthesis techniques to more diverse accelerators.

Chapter 7

Conclusion

This dissertation has described the motivation, design and implementation of AccSynt, a program synthesiser that aims to learn executable models for the behaviour of accelerators. As a first area of application for AccSynt, we conducted an investigation into the available performance improvements on sparse linear algebra workloads. The results of this investigation showed that significant performance gains are possible by using multiple different accelerators, and that a machine learning model can be used to select the best possible version with high accuracy.

AccSynt's implementation includes a number of novel ideas in program synthesis—for example, annotating parameters with human information to influence control flow synthesis, and combinatorially enumerating loop control flow structures to find the correct solution to a synthesis problem. We evaluated AccSynt's ability to synthesise a series of increasingly complex linear algebra workloads, demonstrating that it is able to synthesise programs with complex nested loops, such as dense matrix-vector multiplication. This synthesis can be achieved in a short enough time that AccSynt can be practically used as a development tool.

The limiting factor for AccSynt's ability to synthesise programs is the range of control flow it can generate—currently, it special-cases a particular form of loop structure. This restriction prevented it from synthesising sparse linear algebra, but we propose a method by which this restriction can be lifted. Additionally, we implemented a hard-coded modification of AccSynt that is able to synthesise sparse linear algebra; this modification demonstrates the potential of future versions of AccSynt.

By learning a model for the behaviour of an accelerator, we can close the gap between user code and arbitrary future accelerators while minimising future programmer effort. AccSynt provides a prototype implementation for this technique that shows how

more general solutions can be developed in the future.

7.1 Future Work

Future work on AccSynt will focus on three primary areas. First, we will aim to improve the expressivity of the synthesis process by implementing a component-based method for generating control flow based on parameter annotations from the user. The work we describe that extends the synthesiser to sparse matrix-vector multiplication can be seen as a first prototype of this generalisation. Then, we aim to evaluate it against more diverse accelerator devices to validate the program synthesis methodology in different contexts. Finally, the most important piece of future work is to close the gap in the toolchain by generalising code to constraints that can then be searched for using existing tools.

7.1.1 Control Flow and Components

The approach currently taken by AccSynt is to special-case the ability to generate a specific form of loop when it is aware of data with a known size. This works well for computation with a fixed, regular iteration pattern—for example, dense matrix-vector multiplication. However, attempting to apply this pattern to sparse matrix-vector multiplication revealed that it is not a general enough approach. Many control-flow patterns are not expressible using these loops.

However, the idea of synthesising control flow separately to computation is a useful one. To preserve this idea, we propose that future iterations of AccSynt make use of a library of control flow components rather than hard-coding one particular idiom. These components would fit well with improvements to the way in which incorrect results are analysed by AccSynt—for example, conditional control flow could be generated if a very small number of results in an output vector are incorrect, or different loop bounds generated if a range of results are incorrect. Doing this means that basic block bodies can still be generated separately to control flow as is the case with AccSynt currently.

Being able to easily add new control flow structures to the synthesis algorithm is important for future development, and so it may be prudent to develop a DSL or library for expressing these components—AccSynt itself is then only responsible for instantiating these in a generic way.

7.1.2 Code Search

AccSynt's primary goal is not to develop novel program synthesis techniques, but rather to facilitate the understanding of diverse accelerators so that they can be used more widely in programs. Currently, the model AccSynt learns for an accelerator is an LLVM IR program—this can be understood by programmers and used as a reference to manually discover similar instances. However, it is not easy to search automatically for similar code given a program.

Work from Ginsbach and O'Boyle [15] deals with discovering instances of computational idioms in source code, given a constraint-based description of the idiom as input. However, the problem of inducing descriptive constraints based on an example does not have an obvious solution—many different programs with varied structure may all be required to map to the same constraints.

Despite the difficulty of this problem, it is perhaps the most important future direction for work related to AccSynt. If constraint descriptions can be generated from example programs, code suitable for any accelerator we can learn using program synthesis can be discovered automatically.

7.1.3 Generalising AccSynt

This dissertation has dealt primarily with linear algebra as a working example for program synthesis applied to learning accelerator behaviour. However, as we discussed in Section 2.1, there are an increasing number of accelerators available in a huge number of problem domains. For AccSynt to be a truly useful tool, it should be able to learn accelerator behaviour in a more general way.

Expressing more general control flow as described previously is one technique that will allow AccSynt to target more diverse accelerators, assuming that enough components and heuristics are developed. Another direction to investigate in the future is interfaces that do not fit neatly into the C type signature model currently used—for example, accelerators that use memory mapped IO to communicate with the host. New program heuristics and annotations will be necessary to integrate these interfaces with program synthesis techniques.

A potential next step for AccSynt in this direction is to select several diverse accelerators and perform a detailed analysis of how AccSynt could synthesise programs to match them, and what assumptions from linear algebra must be relaxed to do so.

Bibliography

[1] R. Alur et al. “Syntax-Guided Synthesis”. In: *2013 Formal Methods in Computer-Aided Design*. Oct. 2013, pp. 1–8. DOI: 10.1109/FMCAD.2013.6679385.

[2] C. Baaij. “CλasH : From Haskell to Hardware”. Info:Eu-Repo/Semantics/masterThesis. Dec. 2009. URL: http://essay.utwente.nl/59482/ (visited on 10/12/2017).

[3] J. Bachrach et al. “Chisel: Constructing Hardware in a Scala Embedded Language”. In: *DAC Design Automation Conference 2012*. June 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[4] D. H. Bailey et al. “The NAS Parallel Benchmarks—Summary and Preliminary Results”. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing ’91. New York, NY, USA: ACM, 1991, pp. 158–165. ISBN: 978-0-89791-459-8. DOI: 10.1145/125826.125925. URL: http://doi.acm.org/10.1145/125826.125925 (visited on 07/06/2018).

[5] Osbert Bastani et al. “Synthesizing Program Input Grammars”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 95–110. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062349. URL: http://doi.acm.org/10.1145/3062341.3062349 (visited on 09/18/2017).

[6] L. S. Blackford, Roldan Pozo, and Et Al. “An Updated Set of Basic Linear Algebra Subprograms (BLAS)”. en. In: *Acm Transactions on Mathematical Software* 2 (June 2002). URL: https://www.nist.gov/publications/updated-set-basic-linear-algebra-subprograms-blas (visited on 07/23/2018).

[7] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. “A Training Algorithm for Optimal Margin Classifiers”. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT ’92. New York, NY, USA: ACM, 1992, pp. 144–152. ISBN: 978-0-89791-497-0. DOI: 10.1145/130385.130401. URL: http://doi.acm.org/10.1145/130385.130401.

[8] Jong-Ho Byun et al. *Autotuning Sparse Matrix-Vector Multiplication for Multicore*. Tech. rep. UCB/EECS-2012-215. EECS Department, University of California, Berkeley, Nov. 2012. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-215.html.

[9] Yajing Chen et al. "A Programmable Galois Field Processor for the Internet of Things". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. New York, NY, USA: ACM, 2017, pp. 55–68. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080227. URL: http://doi.acm.org/10.1145/3079856.3080227 (visited on 11/27/2017).

[10] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 1:1–1:25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. URL: http://doi.acm.org/10.1145/2049662.2049663 (visited on 07/06/2018).

[11] Jack Dongarra, Victor Eijkhout, and Henk van der Vorst. "An Iterative Solver Benchmark". In: *Sci. Program.* 9.4 (Dec. 2001), pp. 223–231. ISSN: 1058-9244. DOI: 10.1155/2001/527931. URL: https://doi.org/10.1155/2001/527931 (visited on 07/06/2018).

[12] Athena Elafrou et al. "SparseX: A Library for High-Performance Sparse Matrix-Vector Multiplication on Multicore Platforms". In: *ACM Trans. Math. Softw.* 44.3 (Jan. 2018), 26:1–26:32. ISSN: 0098-3500. DOI: 10.1145/3134442. URL: http://doi.acm.org/10.1145/3134442 (visited on 06/05/2018).

[13] Joyce Friedman. "Review: Alonzo Church, Application of Recursive Arithmetic to the Problem of Circuit Synthesis". In: *J. Symbolic Logic* 28.4 (Dec. 1963), pp. 289–290. URL: https://projecteuclid.org:443/euclid.jsl/1183734749.

[14] Matteo Frigo and Steven G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the Ieee*. 2005, pp. 216–231.

[15] Philip Ginsbach and Michael F. P. O'Boyle. "Discovery and Exploitation of General Reductions: A Constraint Based Approach". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 269–280. ISBN: 978-1-5090-4931-8. URL: http://dl.acm.org/citation.cfm?id=3049832.3049862 (visited on 09/18/2017).

[16] Philip Ginsbach et al. “Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’18. New York, NY, USA: ACM, 2018, pp. 139–153. ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3173182. URL: http://doi.acm.org/10.1145/3173162.3173182 (visited on 05/17/2018).

[17] Patrice Godefroid and Ankur Taly. “Automated Synthesis of Symbolic Instruction Encodings from I/O Samples”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. New York, NY, USA: ACM, 2012, pp. 441–452. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254116. URL: http://doi.acm.org/10.1145/2254064.2254116 (visited on 10/17/2017).

[18] Sumit Gulwani. “Automating String Processing in Spreadsheets Using Input-Output Examples”. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’11. New York, NY, USA: ACM, 2011, pp. 317–330. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926423. URL: http://doi.acm.org/10.1145/1926385.1926423 (visited on 10/10/2017).

[19] Sumit Gulwani, Alex Polozov, and Rishabh Singh. *Program Synthesis*. Vol. 4. NOW, Aug. 2017. URL: https://www.microsoft.com/en-us/research/publication/program-synthesis/.

[20] Bastian Hagedorn et al. “High Performance Stencil Code Generation with Lift”. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. New York, NY, USA: ACM, 2018, pp. 100–112. ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168824. URL: http://doi.acm.org/10.1145/3168824 (visited on 07/02/2018).

[21] Niranjan Hasabnis and R. Sekar. “Extracting Instruction Semantics via Symbolic Execution of Code Generators”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. New York, NY, USA: ACM, 2016, pp. 301–313. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950335. URL: http://doi.acm.org/10.1145/2950290.2950335 (visited on 10/03/2017).

[22] Niranjan Hasabnis and R. Sekar. “Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers”. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’16. New York, NY, USA: ACM, 2016, pp. 311–324. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872380. URL: http://doi.acm.org/10.1145/2872362.2872380 (visited on 10/02/2017).

[23] Stefan Heule et al. “Stratified Synthesis: Automatically Learning the X86-64 Instruction Set”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’16. New York, NY, USA: ACM, 2016, pp. 237–250. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908121. URL: http://doi.acm.org/10.1145/2908080.2908121 (visited on 10/16/2017).

[24] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, 2009. ISBN: 630813-054US.

[25] Susmit Jha and Sanjit A. Seshia. “A Theory of Formal Synthesis via Inductive Learning”. In: *arXiv:1505.03953 [cs]* (May 2015). arXiv: 1505.03953 [cs]. URL: http://arxiv.org/abs/1505.03953 (visited on 11/07/2017).

[26] Susmit Jha et al. “Oracle-Guided Component-Based Program Synthesis”. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE ’10. New York, NY, USA: ACM, 2010, pp. 215–224. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806833. URL: http://doi.acm.org/10.1145/1806799.1806833 (visited on 11/07/2017).

[27] Norman P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA ’17. New York, NY, USA: ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246. URL: http://doi.acm.org/10.1145/3079856.3080246 (visited on 11/27/2017).

[28] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. PhD thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, 2002. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.331 (visited on 04/06/2017).

[29] Alan Leung, John Sarracino, and Sorin Lerner. “Interactive Parser Synthesis by Example”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15. New York, NY, USA: ACM, 2015, pp. 565–574. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2738002. URL: http://doi.acm.org/10.1145/2737924.2738002 (visited on 10/10/2017).

[30] Junghee Lim and Thomas Reps. “TSL: A System for Generating Abstract Interpreters and Its Application to Machine-Code Analysis”. In: *ACM Trans. Program. Lang. Syst.* 35.1 (Apr. 2013), 4:1–4:59. ISSN: 0164-0925. DOI: 10.1145/2450136.2450139. URL: http://doi.acm.org/10.1145/2450136.2450139 (visited on 10/12/2017).

[31] Nuno P. Lopes et al. “Provably Correct Peephole Optimizations with Alive”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15. New York, NY, USA: ACM, 2015, pp. 22–32. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737965. URL: http://doi.acm.org/10.1145/2737924.2737965 (visited on 07/23/2018).

[32] David Menendez and Santosh Nagarakatte. “Alive-Infer: Data-Driven Precondition Inference for Peephole Optimizations in LLVM”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 49–63. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062372. URL: http://doi.acm.org/10.1145/3062341.3062372 (visited on 09/18/2017).

[33] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. “PolyMage: Automatic Optimization for Image Processing Pipelines”. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’15. New York, NY, USA: ACM, 2015, pp. 429–443. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694364. URL: http://doi.acm.org/10.1145/2694344.2694364 (visited on 07/02/2018).

[34] Peter-Michael Osera and Steve Zdancewic. “Type-and-Example-Directed Program Synthesis”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15. New York, NY, USA: ACM, 2015, pp. 619–630. ISBN: 978-1-4503-3468-6. DOI: 10.1145/

2737924.2738007. URL: http://doi.acm.org/10.1145/2737924.2738007 (visited on 10/10/2017).

[35] Angshuman Parashar et al. “SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA ’17. New York, NY, USA: ACM, 2017, pp. 27–40. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080254. URL: http://doi.acm.org/10.1145/3079856.3080254 (visited on 09/18/2017).

[36] Phitchaya Mangpo Phothilimthana et al. “Scaling Up Superoptimization”. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’16. New York, NY, USA: ACM, 2016, pp. 297–310. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872387. URL: http://doi.acm.org/10.1145/2872362.2872387 (visited on 10/02/2017).

[37] Jonathan Ragan-Kelley et al. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. New York, NY, USA: ACM, 2013, pp. 519–530. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462176. URL: http://doi.acm.org/10.1145/2491956.2462176 (visited on 06/05/2018).

[38] Eric Schkufza, Rahul Sharma, and Alex Aiken. “Stochastic Superoptimization”. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. New York, NY, USA: ACM, 2013, pp. 305–316. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451150. URL: http://doi.acm.org/10.1145/2451116.2451150 (visited on 10/17/2017).

[39] Armando Solar-Lezama. “The Sketching Approach to Program Synthesis”. en. In: *Programming Languages and Systems*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Dec. 2009, pp. 4–13. DOI: 10.1007/978-3-642-10672-9_3. URL: https://link.springer.com/chapter/10.1007/978-3-642-10672-9_3 (visited on 07/04/2018).

[40] Daniele G. Spampinato and Markus Püschel. “A Basic Linear Algebra Compiler”. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’14. New York, NY, USA: ACM, 2014, 23:23–23:32. ISBN: 978-1-4503-2670-4. DOI: 10.1145/2544137.2544155. URL: http://doi.acm.org/10.1145/2544137.2544155 (visited on 07/23/2018).

[41] Venkatesh Srinivasan and Thomas Reps. “Synthesis of Machine Code from Semantics”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15. New York, NY, USA: ACM, 2015, pp. 596–607. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737960. URL: http://doi.acm.org/10.1145/2737924.2737960 (visited on 10/10/2017).

[42] Michel Steuwer et al. “Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. New York, NY, USA: ACM, 2015, pp. 205–217. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784754. URL: http://doi.acm.org/10.1145/2784731.2784754 (visited on 06/05/2018).

[43] Semen A. Trygubenko and David J. Wales. “Graph Transformation Method for Calculating Waiting Times in Markov Chains”. In: *The Journal of Chemical Physics* 124.23 (June 2006), p. 234110. ISSN: 0021-9606. DOI: 10.1063/1.2198806. URL: http://aip.scitation.org/doi/full/10.1063/1.2198806 (visited on 01/17/2018).

[44] Semen A. Trygubenko and David J. Wales. “Kinetic Analysis of Discrete Path Sampling Stationary Point Databases”. In: *Molecular Physics* 104.9 (May 2006), pp. 1497–1507. ISSN: 0026-8976. DOI: 10.1080/00268970600556659. URL: https://doi.org/10.1080/00268970600556659 (visited on 01/17/2018).

[45] Yatish Turakhia, Gill Bejerano, and William J. Dally. “Darwin: A Genomics Co-Processor Provides Up to 15,000X Acceleration on Long Read Assembly”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’18. New York, NY, USA: ACM, 2018, pp. 199–213. ISBN: 978-1-4503-4911-6.

DOI: 10.1145/3173162.3173193. URL: http://doi.acm.org/10.1145/3173162.3173193 (visited on 06/05/2018).

[46] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. "Synthesizing Highly Expressive SQL Queries from Input-Output Examples". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 452–466. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062365. URL: http://doi.acm.org/10.1145/3062341.3062365 (visited on 09/18/2017).

[47] Ke Wang et al. "An Overview of Micron's Automata Processor". In: *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES '16. New York, NY, USA: ACM, 2016, 14:1–14:3. ISBN: 978-1-4503-4483-8. DOI: 10.1145/2968456.2976763. URL: http://doi.acm.org/10.1145/2968456.2976763 (visited on 11/27/2017).

[48] Mohamed Zahran. "Heterogeneous Computing: Here to Stay". In: *Queue* 14.6 (Dec. 2016), 40:31–40:42. ISSN: 1542-7730. DOI: 10.1145/3028687.3038873. URL: http://doi.acm.org/10.1145/3028687.3038873 (visited on 07/03/2018).