Doctoral Thesis

Building Abstractions for Staged DSLs in Performance-Oriented Program Generators

Author(s):
Stojanov, Alen

Publication Date:
2019-05

Permanent Link:
https://doi.org/10.3929/ethz-b-000372536

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Building Abstractions for Staged DSLs in Performance-Oriented Program Generators
BUILDING ABSTRACTIONS FOR STAGED DSLS
IN PERFORMANCE-ORIENTED PROGRAM
GENERATORS

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES OF ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

ALEN STOJANOV
Dipl., Eidgenössisches Polytechnikum
born on 1 September 1987
citizen of North Macedonia and Bulgaria

accepted on the recommendation of

Prof. Dr. Markus Püschel, examiner
Prof. Dr. Tiark Rompf, co-examiner
Prof. Dr. Zhendong Su, co-examiner

2019
To Gordana, my mother.

To this very day, you would still say: “Ајде Ален, што чекањ?”

I will forever remember your words. These simple words, that hold the meaning of: “Come on, Alen, what are you waiting for?”, have been the driving force throughout this journey.

Well mom, I won’t be waiting anymore.
Abstract

Developing high-performance code for numerical domains is challenging, as it requires hand-in-hand specialization with the continuous evolution of modern hardware. Program generators based on domain-specific languages (DSLs) can provide a solution to the challenge of re-specializing programs and libraries as new architectures emerge. However, such code generators are difficult to design as they require DSLs that reason about high-level mathematical domains, and employ analysis and transformation steps to map these domains to low-level hardware instructions. The difficulty comes from both the problem of designing an extensible approach to perform all the optimizations the compiler is unable to do and the actual implementation of the generator.

The Lightweight Modular Staging (LMS) framework delivers the promise of “abstracting without regret” through the use of multi-stage programming (MST). LMS implements MST based on types and has shown to be an effective tool for designing and staging DSLs, providing means to abstract DSLs and thus improve the state of generators.

The work in this thesis focuses on the design of abstractions for staged DSLs that simplify analyses and transformations to deliver high performance. First, we demonstrate a systematic approach to enable data parallelism, by automatically generating low-level, instruction set architecture (ISA) DSLs, providing complete support from MMX to AVX-512 ISAs. We then show the immediate benefit of these DSLs when used in managed language runtimes as embedded DSLs, allowing for tight integration with the host language and significant improvements in performance. Then, we use the low-level DSLs as building blocks for low-level abstractions to provide an efficient implementation of variable-precision computations on state-of-the-art CPUs. Further, we use the concept of stage polymorphism to provide abstractions for data parallelism and perform all relevant optimizations in a prototypical generator for high-performance convolutional operations. Finally, we propose the metacontainer abstraction, to stage a more general, dynamically typed numerical DSL in the LMS framework, which relies heavily on static type information. We demonstrate this work by staging a subset of MATLAB for compilation to low-level C code, and use the metacontainer abstraction to simplify type and shape inference, and handle many dynamical aspects of the language.
Résumé

Le développement de programmes de calcul numérique à haute performance est délicat, car il nécessite une spécialisation qui doit suivre l’évolution constante du matériel informatique. Les générateurs de programmes basés sur des langages dédiés (DSLs) répondent à ce problème en permettant d’adapter directement un programme ou une bibliothèque à une nouvelle architecture matérielle aussitôt qu’elle apparaît. Ces générateurs sont néanmoins complexes, car ils nécessitent des DSLs capables de représenter des concepts mathématiques de haut niveau, et différentes étapes de manipulation et de transformation pour produire les instructions matérielles de bas niveau correspondantes. Deux difficultés se posent alors ; d’une part trouver une approche modulaire pour effectuer toutes les optimisations qu’un compilateur ne serait pas capable d’effectuer, et d’autre part implémenter un tel générateur.

Lightweight Modular Staging (LMS) est une forme de programmation par étage (PPE) qui permet de réaliser une « abstraction sans regrets ». LMS implémente le PPE via le typage, et se révèle être un très bon outil pour implémenter les DSLs, permettant ainsi de faciliter le développement des différents étages des générateurs de code.

Dans cette dissertation, nous étudions la conception des abstractions des DSLs à étage, pour simplifier les analyses et les transformations qui conduisent à la production de code à haute performance. Nous présentons d’une part la génération automatique de DSLs de bas niveau, représentant des jeux d’instructions (ISA) complets, qui peuvent s’étendre du MMX à l’AVX-512, permettant ainsi la programmation en parallèle. Nous montrons ensuite les avantages directs que procurent ces DSLs lorsqu’ils sont implémentés au sein d’une machine virtuelle de haut niveau, permettant ainsi une intégration fine au langage hôte, et une augmentation significative des performances. Nous utilisons ensuite les DSLs de bas niveau comme briques de base pour l’abstraction de bas niveau, permettant ainsi une implémentation efficace de calcul numérique à précision variable sur des processeurs récents. Par ailleurs, nous utilisons le concept de polymorphisme à étage pour permettre des abstractions relatives au parallélisme et les optimisations correspondantes dans un prototype de générateur de convolutions à haute performance. Enfin, nous introduisons le concept de méta-conteneur, qui permet de proposer des DSL numériques avec un typage dynamique plus général que celui initialement proposé par LMS,
qui exploite principalement les informations obtenues par un typage stathique. Nous montrons l’intérêt de ce nouveau concept en implémentant un générateur à étage qui transforme un sous-ensemble de MATLAB en un code C de bas niveau. Les méta-conteneurs permettent alors de simplifier l’inférence de type et de forme, et permettent de gérer plusieurs aspects dynamiques du langage.
Acknowledgements

First, I like to express my utmost gratitude to my advisor Prof. Dr. Markus Püschel, for all his support, patience and guidance; for helping me grow as a scholar and for teaching me to perform scientific research with the highest level of detail and devotion.

I would like to thank my co-advisor Prof. Dr. Tiark Rompf for being supportive and always ready to give insightful feedback for my work, and for helping me improve myself as a researcher. I have greatly enjoyed our collaboration, which I hope we continue in the future.

I would like to thank Prof. Dr. Zhendong Sun for agreeing to be part of my PhD committee.

I would like to thank Prof. Dr. Michael Franz. The short collaboration, during the work of my master thesis, had a profound impact on my carrier and eventually lead to the decision of pursuing a PhD degree.

I had the pleasure and luck to work with many great collages and would like to thank all my collaborators: Francois, Georg, Daniele, Victoria, Luca, Michael, Remi, Gagandeep and Tyler. It was such a rewarding experience working with all of you.

I like to thank Ivaylo Toskov, for choosing me as his master thesis co-advisor and for the impeccable results that his work delivered.

I would also like to use this opportunity to thank my closest friends: Jana, Darko, Vladimir, Mihajlo, Manuel, Kaveh, Endri, Kaan, Rostislav, Krenar, Eray, Ivana, Viktor, Riste, Mite, Ace, Jovanche, Ilija, Petre and Sanja. Your friendship has been an immense pleasure throughout all these years.

I doubt that I will ever find enough words to thank my parents Gordana and Timko. You did an exceptional job to raise me up as a person and to set me on the right path. None of my success would ever materialized, without your presence as role models in my life.

Finally, I like to thank my wife, Katerina. Thank you for the courage that you lend me, for believing in me and for being there for me all these years. Much of my success is thanks to you.
## Contents

1 Introduction .......................................................... 1
   1.1 Goal of This Dissertation ........................................ 5
   1.2 Contributions of this Dissertation ............................... 6
   1.3 Background ........................................................ 7
      1.3.1 Scala ....................................................... 8
      1.3.2 Lightweight Modular Staging ............................... 14
   1.4 Organization of the Dissertation ................................. 17

2 Vectorization in LMS .................................................. 19
   2.1 Motivation ........................................................ 20
   2.2 Background ........................................................ 22
      2.2.1 Intrinsics .................................................... 22
      2.2.2 Java Virtual Machines ..................................... 23
   2.3 SIMD Intrinsics on Managed Language Runtimes ................. 25
      2.3.1 Automatic Generation of ISA-specific DSLs .............. 26
      2.3.2 Developing Explicitly Vectorized Code in the JVM ....... 32
   2.4 Results .......................................................... 34
      2.4.1 Evaluation .................................................. 34
      2.4.2 Limitations ................................................ 40
   2.5 Related Work ..................................................... 41
   2.6 Summary ........................................................ 42

3 Low Level Abstractions ................................................ 43
   3.1 Motivation ........................................................ 44
   3.2 Background ........................................................ 45
   3.3 Low Precision Arithmetic ....................................... 47
      3.3.1 4-bit Arithmetic .......................................... 48
      3.3.2 8, 16 and 32-bit Arithmetic ............................... 54
   3.4 Evaluation ........................................................ 55
      3.4.1 Performance of Individual Routines ....................... 59
      3.4.2 Evaluation of the Low-Level Abstractions in LMS ...... 60
      3.4.3 Evaluation of Quantized IHT and Quantized GD .......... 62
   3.5 Related Work ..................................................... 64
   3.6 Summary ........................................................ 65

4 Abstracting Data Parallelism using Staged Polymorphism .......... 67
   4.1 Motivation ........................................................ 68
   4.2 FGen ............................................................. 69
      4.2.1 Tiling ......................................................... 71
      4.2.2 Mapping to Σ-LL and Loop Optimizations .................. 77
   4.3 Abstracting Data Parallelism ..................................... 82
      4.3.1 Parametric Stage Polymorphism ............................. 82
4.3.2 ISA Abstraction ........................................... 89
4.4 Experimental Results ...................................... 90
4.5 Related Work ................................................ 93
4.6 Summary ...................................................... 93

5 object-Oriented Stage Polymorphism ....................... 97
5.1 Motivation .................................................... 98
5.2 MATLAB ..................................................... 100
5.3 Metacontainers ............................................... 107
5.4 MGen ........................................................ 111
  5.4.1 Supported Subset ....................................... 111
  5.4.2 High-level Overview .................................... 113
  5.4.3 Building the M-IR ....................................... 116
  5.4.4 Type Inference .......................................... 118
  5.4.5 Shape Inference ......................................... 124
  5.4.6 Conversion to C-IR ..................................... 136
  5.4.7 Vectorization ............................................ 145
5.5 Results ......................................................... 150
  5.5.1 Evaluation ............................................... 150
  5.5.2 Limitations .............................................. 159
5.6 Related Work ................................................ 161
5.7 Summary ...................................................... 163

6 conclusion and Future Work .................................. 165

BIBLIOGRAPHY ..................................................... 169
The continuous evolution of modern processors spans in complexity and diversity across all scales from mobile devices to data centres. To benefit these advancements and deliver highly performant programs, software must be able to take advantage of multi-level memory hierarchies, deep hardware pipelines, vector instructions, and multiple cores.

Practice shows that highly tuned code can outperform straightforward implementations by one or two orders of magnitude [1], especially for performance-critical numerical computations. In these domains, a diverse set of hand-tuned optimizations are applied. These include loop merging and multiple-level tiling to utilize memory hierarchies, loop unrolling, scalar replacement and register blocking to take advantage of instruction level parallelism and hardware pipelines, explicit vectorization to benefit from data parallelism, task parallelization to utilize multiple cores simultaneously and many more.

Unfortunately, most of these optimization are partially supported and, in many cases, beyond the capabilities of state-of-the-art compilers. As a result, some major chip manufacturers provide hand-written assembly libraries tailored to their microprocessors. Examples include Intel’s Math Kernel Library (MKL) [2], Integrated Performance Primitives (IPP) [3] libraries, AMD Core Math Library [4] and others. However, these libraries are hand-tuned to a specific platform, imposing a significant porting cost when the processor is upgraded, or a new architecture is introduced.

Automatic code generation is a promising solution to the problem of continuous re-specialization and porting of high performance libraries. Generators can capture algorithmic knowledge for computing a particular numerical kernel, or even an entire application, and generate highly tuned code for a specific platform. Whenever a new platform is introduced, the code is regenerated after possible extension to the generator, to address the features of the new platform. Building such generators can be challenging, because is requires support for a variety of devices at multiple levels of
abstraction, from scalar instructions and coarse-grain parallelism to high-level program representation. This is the reason why only few exist to date, restricted to generating numerical kernels, such as the fast Fourier transforms (FFT), filters, and matrix computations.

Domain Specific Languages (DSLs) have proven to be excellent tools for constructing generators, since they can capture the mathematics of a specific domain to carry out critical optimizations (e.g., Spiral [5], FFTW [6], FLAME [7], LGen [8]). DSLs can provide multiple layers of abstraction, ranging from the low-level computer architecture to the high-level mathematical nature of the algorithm to perform optimizations at different levels of abstraction.

High-level abstractions in DSLs are important, as they also enable expressiveness and productivity for domain experts. Engineers and machine learning (ML) researchers benefit from a range of dynamically typed numerical DSLs such as MATLAB and R, or ML frameworks built in Python, that are suitable for quick prototyping. However, none of these DSLs or frameworks offers high-performance. This is because high-level expressiveness and abstractions come at a price. The more abstraction is employed in a DSL, the more pressure it puts on the generator to resolve it. In many cases, high-level abstractions obscure semantic information, which complicates analysis and optimizations employed by the generator during compile-time. The inability to remove abstractions results in its propagation in the generated code, incurring interpretive overhead at runtime.

Multi-stage programming (MSP, or staging for short), as established by Taha and Sheard [9], has been widely used to improve the performance in many domains such as optimizing compilers, domain-specific languages and numerical computation [10–12]. Frameworks such as Lightweight Modular Staging (LMS) [10] that rely on specializing static analyses by partial evaluation have shown to be quite effective in removing the interpretive overhead generated by abstraction. Consequently, MSP delivers the slogan and the promise of “abstracting without regrets”.

Software engineering practice demands generalization and abstraction, whereas high performance demands specialization and concretization. The two goals seem to be at odds. But, with the promise of “abstracting without regret”, we can close this gap. The focus in prior research on program generation and compilers has revolved around the performance of the generated code, as a primary measure to quantify its success. On the other hand, implementing generators with extensibility, modularity, reusability and maintainability in mind, has been a secondary goal, if at all. This is because
such efforts are usually hard to quantify. However, employing state-of-the-art techniques from meta-programming and generative programming, one can improve these properties.

Spiral [5] and SpiralS [11] are good examples in this regard. Spiral is implemented in the programming language GAP [13]. The shortcomings of the language (which was designed for computer algebra system), such as the lack of standard libraries and the absence of support of many modern programming paradigms, result in a codebase that spawns over 1 million lines of code, increasing maintenance effort and impeding implementation time. SpiralS, on the other hand, is a reimplementation of a subset of Spiral in Scala and uses LMS as a staging environment. The approach of using state-of-the-art metaprogramming techniques in a staging environment to Scala, provides access to an extensive set of Java libraries and Scala features. Such features include Scala’s advanced type system and a combined functional and object-oriented paradigms that directly impact and increase productivity. Consequently, SpiralS shows that implementation of a small, but representative subset of Spiral can take as little as 1200 lines of code.

Efforts in providing an elegant solution that abstracts over and simplifies program representation can enable opportunities for more advanced analysis and optimization of programs. Such solutions, using multi-stage programming, applied in numerical domains, would require abstractions that encompasses data-types and instruction set architecture, generating code in various forms and code styles (illustration in Figure 1.1). To encode and apply domain knowledge, we require low-level abstractions to model and abstract fine-grained numerical computation, and high-level abstractions to improve DSL transformation and analysis.

This dissertation, considers the design abstraction from low-level to high-level in several domains, which is explained next.

Support for SIMD in MSP Fine-grained representation of computation primitives is imperative for high-performance in a given DSL. These low-level representations can be used directly in developing new DSLs, or can be used as building blocks for DSL abstractions. The most basic form of such primitives would be a hardware instruction in a given instruction set architecture (ISA). Of particular interest for numerical domains are instructions that take advantage of data parallelism or Single Instruction Multiple Data (SIMD) [14] instructions. However, with the growing diversity of hardware, providing this support could be a very large effort if done manually. Consequently, complete representation of SIMD ISAs has so far
been offered only as a form of a intermediate representation (IR) in state-of-the-art compilers, and has not been demonstrated in an MSP framework. We explore an automatic approach to provide a complete SIMD support for a wide set of ISAs in an MSP framework.

**Low-level abstractions using MSP** Managed language runtimes do not offer the low-level control required for delivering high performant code. For example, support for data-parallelism, if available at all, is left to the virtual machine (VM) and the built-in just-in-time (JIT) compiler to carry out automatically, which often leads to suboptimal code. MSP frameworks, such as LMS, have been used to build and compile embedded DSLs. In this scenario, the embedded DSL can interoperate with the host languages and benefit from all its high-level language features. If the DSL also supports data-parallelism, MSP can effectively provide support to the manage language runtime to use the vector API with zero overhead. We explore a metaprogramming approach to achieve this with the goal of providing full SIMD support in managed languages.

DSLs that represent data-parallelism, combined with the MSP promise of “abstraction without regret”, can also be used to create low-level abstractions
that serve as macro systems to define new vectorized APIs. This provides the prospect of creating “virtual ISAs” to support instructions and data formats that are not natively supported by hardware. As an example, we build abstractions for variable-precision computations that are efficiently implemented on state-of-the-art CPUs.

**Abstracting data parallelism**  SpiralS uses stage polymorphism as a principal abstraction tool to encode staging decisions in Scala’s type system. While it offers powerful abstractions to implement many transformation rules from the Spiral system, it lacks support for data-parallelism. We extend SpiralS approach and build abstractions for SIMD target architectures and data representations. We apply the abstraction in a subset of Spiral’s internal DSL, and use it to perform performance-related transformations such as loop tiling, loop merging, loop unrolling and scalar replacement, to achieve high performance.

**Staging general numerical DSLs**  Being able to abstract different data representations and vector ISAs, and applying these abstraction for DSL transformations, raises the question of whether these abstractions are restricted to the domain defined by the DSL, or whether they are applicable in more general numerical DSLs. Such DSLs include MATLAB, Wolfram, R and others, that are dynamically typed and cover wide range of domains. Particularly challenging task is to stage these languages with an MSP framework, such as LMS, that performs staging based on types, when types are determined only at the runtime of the staged code. For this purpose, we stage a subset of MATLAB, and build abstractions to handle the dynamic aspects of the language. Our abstractions aim to simplify analyses and transformations to infer types, and dimension of numerical primitives.

### 1.1 Goal of this dissertation

The overall goal in this dissertation can be summarized as follows:

Provide low-level and high-level abstractions that simplify the construction, transformation, and optimization of staged DSLs. The provided abstractions will incorporate knowledge of a wide variety of instruction set architectures (ISA), take advantage of data parallelism, integrate tightly in a staging environment and offer high performance.
To achieve this goal, the work in this thesis uses the Lightweight Modular Staging (LMS) [10] framework. LMS uses metaprogramming approach to deliver staging based on types, and is designed as a library in Scala. We design several prototypical DSLs and program generators to demonstrate our methodologies in designing low-level and high-level abstractions. In particular, we show how low-level abstractions in LMS can be used in Scala directly, or can be used as a building block for new abstraction in delivering variable precision computations. Furthermore, we design two prototypical generators, FGen and MGen. FGen focuses on generating code for convolutions, while MGen stages subset of the MATLAB language to compile to low-level C code. For each generator and DSL, we provide an evaluation of the involved abstractions in delivering the required analysis and transformation steps to generate code and performance.

1.2 CONTRIBUTIONS OF THIS DISSERTATION

This dissertation offers several contributions:

SYSTEMATIC SUPPORT FOR DATA PARALLELISM IN LMS The first contribution is a complete support of Single Instruction Multiple Data (SIMD) intrinsics in LMS. We achieve this using an automatic approach to generate the intrinsics API from the vendor-provided XML specification. For each vector ISA, we create modular DSL that can provide low-level, assembly like control to DSL developers to use SIMD instructions. Our approach supports all 5912 Intel SIMD intrinsics from MMX to AVX-512.

SIMD SUPPORT IN NATIVE LANGUAGE RUNTIMES Managed language runtimes, such as the Java Virtual Machine (JVM,) provide adequate performance for a wide range of applications, but at the same time, lack much of the low-level control that performance-minded programmers appreciate in languages like C/C++. We employ a metaprogramming approach that enables programmers to use LMS and the available SIMD support to generate and load native code at runtime. The work demonstrates that this combination of SIMD and metaprogramming enables developers to write high-performance, vectorized code on an unmodified JVM that outperforms the auto-vectorizing HotSpot just-in-time (JIT) compiler.

VARIABLE PRECISION THROUGH LOW-LEVEL ABSTRACTIONS Our support of SIMD in LMS can be used to build low-level abstractions. Program-
mers can use the entire set of vector DSLs as a kind of macro system to define new vector APIs with zero overhead. We demonstrate this use case to build a variable precision API, with a particular focus on 4-bit stochastic quantization. We provide an end-to-end implementation of our approach in LMS, as well as a standalone C/C++ library. The LMS approach offers significant improvements over a native JVM implementation and provides tight integration between vectorized native code and the managed JVM ecosystem. At the same time, our C/C++ library demonstrates that reduced precision stochastic quantized arithmetic can be implemented efficiently on modern CPUs.

DATA PARALLELISM THOUGH THE USE OF STAGED POLYMORPHISM

We present FGen, a prototypical program generator for high performance convolution operations. The generator uses an internal mathematical DSL to enable structural optimization at a high level of abstraction. We use FGen as a testbed to demonstrate how to provide modular and extensible support for modern SIMD vector architectures in a DSL-based generator. Specifically, we show how to combine staging and generic programming with type classes to abstract over both the data type (real or complex) and the target architecture (e.g., SSE or AVX) when mapping DSL expressions to C code with explicit vector intrinsics. Benchmarks shows that the generated code is competitive with commercial libraries.

STAGING MATLAB USING OBJECT-ORIENTED STAGE POLYMORPHISM

We demonstrate the use of object-oriented stage polymorphism as an abstraction tool for building code generators. For this goal, we build MGen, a prototypical program generator for a subset of the MATLAB language. We show that with the provided abstraction, we can simplify the analysis and transformation of staged DSLs, and show that we can use LMS in generating C code for a dynamically typed numerical language as MATLAB.

1.3 BACKGROUND

This section provides the necessary background for this dissertation. We assume that the reader has basic knowledge of the programming languages C, C++ and Scala, and can understand their syntax.

We give a short overview of some of the more advanced and not as commonly known features of Scala utilized in Section 1.3.1. We introduce
Lightweight Modular Staging (LMS), the metaprogramming approach upon which our work is based in Section 1.3.2.

1.3.1 Scala

We present a brief introduction to relevant generic programming and metaprogramming techniques in Scala.

**TYPE PARAMETERIZATION** Scala supports type parameters for classes and methods, allowing us to create generic classes and polymorphic methods. To define a generic class, we use:

```scala
class Stack[A] {
  private var elements: List[A] = Nil
  def push(x: A) { elements = x :: elements }
  def peek: A = elements.head
  def pop(): A = {
    val currentTop = peek
    elements = elements.tail
    currentTop
  }
}
```

This implementation of a Stack class takes any type A as a parameter. Consequently the underlying list can only store elements of type A, and the methods push, pop and peek will operate with instances of type A.

To use type parametrization, we then replace A with a concrete type. To illustrate, consider the following:

```scala
val stack = new Stack[Int]
stack.push(1)
stack.push(2)
println(stack.pop) // prints 2
println(stack.pop) // prints 1
```

**TYPE ALIASES** A type alias creates a new named type for a specific, existing type (or class). This new type alias is treated by the compiler as if it were defined in a regular class. New instances can be created from a type alias, and used in place of classes for type parameters. Type aliases can also be specified in value, variable, and function return types. If the class being aliased has type parameters, either the type parameters can be added in the type alias or it can be fixed with specific types. For example:
Type aliases can only be defined inside objects, classes, or traits. They only work on types, so objects cannot be used to create type aliases.

**Implicit Parameters** Scala features *implicit parameters* that allow the developer to omit calling methods or referencing variables directly, but instead rely on the compiler to make the connections. Parameters become implicit if they are marked by the `implicit` keyword at the start of the parameter list. If the parameters in that parameter list are not passed as usual, the Scala compiler will attempt to find an implicit value of the correct type, and if it succeeds, will pass it automatically.

To illustrate consider the following program:

```scala
abstract class Monoid[A] {
  def add(x: A, y: A): A
  def unit: A
}

object ImplicitTest {
  implicit val stringMonoid: Monoid[String] = new Monoid[String] {
    def add(x: String, y: String): String = x concat y
    def unit: String = ""
  }
  implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
  def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
    if (xs.isEmpty) m.unit else m.add(xs.head, sum(xs.tail))
  def main(args: Array[String]): Unit = {
    println(sum(List(1, 2, 3)))  // uses intMonoid implicitly
    println(sum(List("a", "b", "c")))  // uses stringMonoid implicitly
  }
}
```
In the example above we define a method `sum` which computes the sum of a list of elements using the monoid’s `add` and `unit` operations. The operation `add` combines a pair of `A`s and returns another `A`, together with `unit` that is able to create some (specific) `A`.

To show how implicit parameters work, we have the following setup:

1. First, we define monoids `stringMonoid` and `intMonoid` for strings and integers, respectively. The `implicit` keyword indicates that the corresponding object can be used implicitly.

2. The method `sum` takes a `List[A]` and returns an `A`, which takes the initial `A` from `unit`, and combines each next `A` in the list to that with the `add` method. Making the parameter `m` implicit here means we only have to provide the `xs` parameter when we call the method if Scala can find an implicit `Monoid[A]` to use for the implicit `m` parameter.

3. In our main method we call `sum` twice, and only provide the `xs` parameter.

In the main method, the Scala compiler will search for an implicit definition in the `implicit scope`. The first call to `sum` passes a `List[Int]` for `xs`, which means that `A` is `Int`. The implicit parameter list with `m` is left out, so Scala will look for an implicit of type `Monoid[Int]`. `intMonoid` is an implicit definition that can be accessed directly in `main`. It is also of the correct type, so it is passed to the `sum` method automatically. The second call to `sum` passes a `List[String]`, which means that `A` is `String`. Implicit lookup will go the same way as with `Int`, but this time will find `stringMonoid`, and pass it automatically as `m`. The program output will be:

```
1 6
2 abc
```

Implicit values can not be top-level values, and must be enclosed in a Scala class or object. An value or an implicit definition is consider to be part of the `implicit scope` if:

- is defined in, passed into, or imported to the current scope
- is defined in companion objects (i.e. an object with the same name as a given Scala class)

If the compiler will not be able to resolve the implicit value or definition, it will produce a compiler error.
implicit conversions. Apart from implicit parameters, the Scala compiler also features implicit conversions to an expected type. The rule is simple. Whenever the compiler sees a type \( X \), but needs a \( Y \), it will look for an implicit function that converts \( X \) to \( Y \). For example, normally a double cannot be used as an integer, because it loses precision:

```
val i: Int = 3.5
:5: error: type mismatch;
```

However, by defining an implicit conversion we obtain:

```
implicit def doubleToInt(x: Double) = x.toInt
doubleToInt: (Double)Int
val i: Int = 3.5
i: Int = 3
```

The compiler sees a \( \text{Double} \), specifically 3.5, in a context where it requires an \( \text{Int} \). So far, the compiler is looking at an ordinary type error. Before giving up, though, it searches for an implicit conversion from \( \text{Double} \) to \( \text{Int} \). In this case, it finds one: \( \text{doubleToInt} \), which is already in scope.

Implicit conversion can also be passed as implicit parameters in Scala. Consequently, there are three places implicits are used in the language: implicit parameters, conversions to an expected type and conversions of the receiver of a selection. To illustrate the last use case for implicits, consider the following example:

```
implicit def stringWrapper(s: String) = new RandomAccessSeq[Char] {
  def length = s.length
  def apply(i: Int) = s.charAt(i)
}
res = "abc123" exists (_.isDigit)
res: Boolean = true
```

In the example above, the expression "abc".exists is implicitly converted into stringWrapper("abc").exists because the exists method is not available on Strings but is available on RandomAccessSeqs. As of Scala 2.10 also classes can be constructed implicitly through a feature called implicit class. To use the feature we mark a class as implicit, which in turn will make its primary constructor available similar to implicit conversion.
The resolution of implicit conversion works in a similar fashion as it is the case of implicit parameters resolution. However, in both cases, their application is governed by the following rules:

- **Non-Ambiguity Rule.** An implicit conversion is only inserted if there is no other possible conversion to insert. If the compiler has multiply choices, it will refuse to choose, and return a compiler error.
- **One-at-a-time Rule.** Only one implicit is tried. The compiler will never apply or chain multiple conversion.
- **Explicits-First Rule.** Whenever code type checks as it is written, no implicits are attempted. The Scala compiler will not change code that already works.

We refer to both implicit parameters and implicit conversions as *implicits*.

**Type Classes and Implicits** Type classes [15] were first introduced in Haskell to enable a statically typed version of *ad-hoc* polymorphism. Type classes are implemented in Scala as normal classes and objects, but use implicits in order to be passed in functions. This usage pattern enables the provision of common interfaces to classes which did not declare them. It can both serve as a bridge pattern, gaining separation of concerns, and as an adapter pattern.

To illustrate take a generic method that sums all elements of a list:

```scala
def sum[T](list: List[T])(implicit n: Numeric[T]): T = {
  import n._ // import implicit parts into scope
  list.foldLeft(n.zero)(_ + _)
}
```

The same expression can be rewritten using a syntactic sugar, usually referred to as a *context bound*, that looks like this:

```scala
def sum[T:Numeric](list: List[T]): T = {
  import implicitly[Numeric[T]]._ // import implicit parts into scope
  list.foldLeft(n.zero)(_ + _)
}
```

As the method `sum` is generic, the implementation of a plus operator needs to become available to the `sum` method. The implementation can be implicitly passed by the `Numeric[T]` type class instance, providing a
type-specific implementation for a given T. A possible interface definition for Numeric could look like this:

```scala
trait Numeric[T] {
  def plus (l: T, r: T): T
  def minus (l: T, r: T): T
  def zero: T

  implicit class Ops(lhs: T) {
    def +(rhs: T) = plus (lhs, rhs)
    def -(rhs: T) = minus(lhs, rhs)
  }
}
```

Note that in the example above with the use of the implicit class Ops, the arithmetic operations such as + and - will be desugared into method calls of Numeric, i.e. plus and minus respectively.

There are two important benefits of using type classes. The first, by defining a context bound, we are capable of restricting the input type to the sum method to be a numeric type (that has at least some basic algebraic operations). The second, type classes offer benefit over inheritance, such that we can retroactively add functionality to existing data types. For example, we can provide a type class implementation of Numeric for Int or Double.

```scala
implicit object NumericInt extends Numeric[Int] {
  def plus (l: Int, r: Int): Int = l + r
  def minus (l: Int, r: Int): Int = l - r
  def zero: Int = 0
}

implicit object NumericDouble extends Numeric[Double] {
  def plus (l: Double, r: Double): Double = l + r
  def minus (l: Double, r: Double): Double = l - r
  def zero: Double = 0.0
}
```

These objects serve as evidence that both types Int and Double are in fact numeric types. Note that we have used the implicit keyword, which means that the compiler will automatically insert a reference to these objects whenever an implicit parameter of type Numeric[Int] or Numeric[Double] is required in a method application.
1.3.2 Lightweight Modular Staging

This section gives a high level introduction to Lightweight Modular Staging (LMS).

**LMS Overview**  Multi-stage programming [9] was introduced to simplify program generator development by expressing the program generator and parts of the generated code in a single program, using the same syntax. Traditionally this is done through annotations within the code, which requires a specialized compiler such as MetaOCaml [16]. LMS uses only types to distinguish the computational stages. Expressions of type `Rep[T]` in the first stage yield a computation of type `T` in the second stage. For example, the operation

```plaintext
1 def add(a: Int, b: Int) = a + b
2 val c = add(2, 3)
3 val c: 5
```

will simply execute the arithmetic, operation obtaining the result of 5, while the operation

```plaintext
1 def add(a: Rep[Int], b: Rep[Int]) = a + b
2 val c = add(2, 3)
3 val c: Rep[Int]
```
uses the higher-kinded type Rep[_] as a marker type to redirect the compiler to use an alternative plus implementation. Instead of obtaining the arithmetic result, the alternative operation will result in an abstract syntax tree (AST) node that represents this expression, as illustrated in Figure 1.2.

LMS makes pervasive use of operator overloading to make code generation blend in with normal programming. And as such, a plus operator will be overloaded with the alternative implementation such as

```scala
def infix_+(lhs: Rep[Int], rhs: Rep[Int]): Rep[Int]
```

This form of operator overloading is extended to if/else, for, while expressions and other built-in constructs [17, 18]. Staged expressions can be further processed and unparsed into source code in Scala or a different language, such as C. The key benefit of staging is that the present-stage code can be written in a high-level style, yet generate future-stage code that is very low-level and efficient. Staging is a programmatic way to remove indirection - when generating code in one step, Scala becomes a glorified macro system to generate C code.

**Propogation of Types in LMS** Scala uses a strong type system. However certain type-related features are weakened by the restrictions and limitations of its runtime environment. As Scala runs on the Java Virtual Machine (JVM), it is limited by *type erasure*. Type erasure refers to the runtime encoding of parameterized classes in Scala, in which all generic type information is removed during runtime.

To avoid the effects of type erasure, LMS uses type class Typ[T], for run-time representation of static types. Instances of Typ[T] are implicitly passed in functions, and stored at each symbol. When no implicit value of the expected type is found, the compiler synthesizes the value itself.

**The LMS Extensible Graph IR** Unlike earlier staging approaches, LMS does not directly generate code in source form but provides an extensible intermediate representation (IR). The overall structure is that of a “sea of nodes” dependency graph [19]. We provide a short overview next, while detailed explanation of LMS internals can be found in [20–23].

The framework provides an extendable interface to model IR class hierarchies. It is based on expression trees, closely resembling an abstract syntax tree (AST), representing expressions, definitions, statements and blocks:
Expressions are restricted to be atomic and extend \( \text{Exp}[T] \), representing constants and symbols:

```scala
abstract class \( \text{Exp}[T] \)
case class \( \text{Const}[T](x: T) \) extends \( \text{Exp}[T] \)
case class \( \text{Sym}[T](n: \text{Int}) \) extends \( \text{Exp}[T] \)
```

Composite IR nodes extend \( \text{Def}[T] \). Custom nodes typically are composite. They refer to other IR nodes only via symbols. As a small example, we present a definition of staged arithmetic on doubles (taken from [20]). We first define a pure interface in the trait \( \text{Arith} \) by extending the LMS trait \( \text{Base} \), which defines \( \text{Rep}[T] \) as an abstract type constructor.

```scala
trait \( \text{Arith} \) extends \( \text{Base} \) {
  def infix_+(x: \text{Rep}[\text{Int}], y: \text{Rep}[\text{Int}]): \text{Rep}[\text{Int}]
  def infix_-(x: \text{Rep}[\text{Int}], y: \text{Rep}[\text{Int}]): \text{Rep}[\text{Int}]
}
```

We continue by adding an implementation component \( \text{ArithExp} \), which defines concrete \( \text{Def}[\text{Double}] \) subclasses for plus and minus operations.

```scala
trait \( \text{ArithExp} \) extends \( \text{BaseExp with Arith} \) {
  case class \( \text{Plus}(x: \text{Exp}[\text{Int}], y: \text{Exp}[\text{Int}]) \) extends \( \text{Def}[\text{Int}] \)
  case class \( \text{Minus}(x: \text{Exp}[\text{Int}], y: \text{Exp}[\text{Int}]) \) extends \( \text{Def}[\text{Int}] \)
  def infix_+(x: \text{Exp}[\text{Int}], y: \text{Exp}[\text{Int}]) = \text{Plus}(x,y)
  def infix_-(x: \text{Exp}[\text{Int}], y: \text{Exp}[\text{Int}]) = \text{Minus}(x,y)
}
```

The trait \( \text{BaseExp} \) defines \( \text{Rep}[T]=\text{Exp}[T] \), whereas \( \text{Rep}[T] \) was left abstract in trait \( \text{Base} \).

Taking a closer look at \( \text{ArithExp} \) reveals that the expected return type of \( \text{infix}_+ \) is \( \text{Exp}[\text{Double}] \) but the result value \( \text{Plus}(x,y) \) is of type \( \text{Def}[\text{Double}] \). This conversion is performed implicitly by LMS using \( \text{toAtom} \):

```scala
implicit def \( \text{toAtom}[T](d: \text{Def}[T]): \text{Exp}[T] = ... \)
```
The method `toAtom` maintains the correct evaluation order by binding the argument `d` to a fresh symbol (on the fly conversion to administrative normal form (ANF)).

**Transformations in LMS**  Transformations in LMS work as IR interpreters. They traverse the IR, and for each transformations they create new symbols instead of using existing ones. Consequently each transformer in LMS, maintains a map, that maps old symbols with new ones. We refer to this transformation as transformation through *iterated staging*.

To perform transformation by iterated staging, LMS turns a sea-of-node representation into a structured graph. Then, transformers use a tree-like traversal over the graph structure, traversing each statement in a topological sort of its dependencies. For each statement a substitution is performed and stored in the map of old symbols to new ones. This is best depicted through the use of `ForwardTransformer` interface. For example, creating a transformer that replaces all addition operators with subtraction operators would have the following form:

```scala
trait SimpleTransformer extends ForwardTransformer {
  val IR: ArithExp; import IR._
  override def transformStm(stm: Stm): Exp[Any] = stm match {
    case TP(s, Plus(a, b)) => {
      // find the substitutions applied before
      val (a_subst, b_subst) = (apply(a), apply(b))
      // substitute the plus operator
      a_subst - b_subst
    }
    case _ => super.transformStm(stm)
  }
}
```

As shown above, the transformer traverses the `ArithExp` in a tree-like fashion, and access the substitution map through the `apply` method of the transformer.

1.4 Organization of the Dissertation

The dissertation is organized as follows. Chapter 2 explains the automatic generation of ISA specific DSLs in LMS, and their direct application in managed language runtimes. Chapter 3 discusses our methodology in building low-level abstractions and their application. Chapters 4 and 5 provide detailed overview of the design and implementation of our proto-
typical generators FGen and MGen, respectively. Finally, Chapter 6 provides summary and conclusion of this work, and proposes future work.
Vectorization in LMS

Lightweight Modular Staging (LMS) is an optimizing compiler framework, allowing programmers to tightly integrate domain-specific abstractions and optimizations into the generation process. It is built as a set of components to represent staged expressions, associated data types, and optimizations that can be both generic and domain-specific. The modular aspect of LMS comes from the fact that these components can be extended and composed in a flexible way, serving as building blocks for domain specific language (DSL) implementations.

The core implementation of LMS offers components to deal with mathematical domains, but it has limitations. Namely, it includes representations for operations that deal with Scala supported primitives, reduced to basic arithmetic computations (+, −, ·, ÷), primitive conversions, and some bit-level operations.

To reason about mathematical domains and develop fast computations, LMS needs a wider range of components of mathematical operations that closely map to the existing hardware. Modern hardware offers wide range of instructions that deal with mathematical operations, in particular SIMD instructions that take advantage of data parallelism to achieve high performance. But the vast number of these instructions, makes porting them to LMS a tedious and error prone process, if done manually.

To work around these issues, this chapter explores the following research questions:

- Can we employ a systematic and automatic inclusion of SIMD support in LMS?
- LMS is built in Scala and runs on the Java Virtual Machine (JVM). To demonstrate an immediate benefit of SIMD support in LMS, can we devise a method to support SIMD on managed language runtimes?
The organization of this chapter is as follows. Section 2.1 motivates the necessity for low-level hardware support on managed languages. Section 2.2 gives overview of intrinsics functions, as well as background for different implementation of the Java Virtual Machine. Section 2.3 shows that it is indeed possible to automatically generate a complete SIMD support from vendor specification, and demonstrates how this support can be used for explicit vectorization in the HotSpot JVM. Section 2.4 shows the obtained speed up and the limitations of our approach. Finally, Sections 2.5 and 2.6 provides overview of the related work, and summary.

The work in this chapter was published in the 2018 International Symposium on Code Generation and Optimization [24].

2.1 MOTIVATION

Managed high-level languages are designed to be portable and to support a broad range of applications. For the programmer, the price is reduced access to detailed and low-level performance optimizations. In particular, SIMD vector instructions on modern architectures offer significant parallelism, and thus potential speedup, but neither languages like Java, JavaScript, Python, or Ruby, nor their managed runtimes provide direct access to SIMD facilities. This means that SIMD optimizations, if available at all, are left to the virtual machine (VM) and the built-in just-in-time (JIT) compiler to carry out automatically, which often leads to suboptimal code. As a result, developers may be pushed to use low-level languages such as C/C++ to gain access to the intrinsics API. But leaving the high-level ecosystem of Java or other languages also means to abandon many high-level abstractions that are key for the productive and efficient development of large-scale applications, including access to a large set of libraries.

To reap the benefits of both high-level and low-level languages, developers using managed languages may write low-level native C/C++ functions, that are invoked by the managed runtime. In the case of Java, developers could use the Java Native Interface (JNI) to invoke C functions with specific naming conventions. However, this process of dividing the application logic between two languages creates a significant gap in the abstractions of the program, limits code reuse, and impedes clear separation of concerns. Further, the native code must be cross-compiled ahead of time, which is an error-prone process that requires complicated pipelines to support different operating systems and architectures, and thus directly affects code maintenance and refactoring.
To address these problems, we propose a systematic and automated approach that gives developers access to SIMD instructions in the managed runtime, eliminating the need to write low-level C/C++ code. Our methodology supports the entire set of SIMD instructions in the form of embedded domain-specific languages (eDSLs) and consists of two parts. First, for each architecture, we automatically generate ISA-specific eDSLs from the vendor’s XML specification of the SIMD intrinsics. Second, we provide the developer with the means to use the SIMD eDSL to develop application logic, which automatically generates native code inside the runtime. Instead of executing each SIMD intrinsic immediately when invoked by the program, the eDSLs provide a staged or deferred API, which accumulates intrinsic invocations along with auxiliary scalar operations and control flow, batches them together in a computation graph, and generates a native kernel that executes them all at once, when requested by the program. This makes it possible to interleave SIMD intrinsics with the generic language constructs of the host language without switching back and forth between native and managed execution, enabling programmers to build both high-level and low-level abstractions, while running SIMD kernels at full speed.

This work presented in this chapter makes the following contributions:

1. We present the first systematic and automated approach that supports the entire set of SIMD instructions, automatically generated from the vendor specification, in a managed high-level language. The approach is applicable to other low-level instructions, provided support for native code binding in the managed high-level language.

2. In doing so, we show how to use metaprogramming techniques and runtime code generation to give back low-level control to developers in an environment that typically hides architecture-specific details.

3. We provide an end-to-end implementation of our approach within the HotSpot JVM, which provides access to all Intel SIMD intrinsics from MMX to AVX-512.

4. We provide benchmarks that demonstrate significant performance gains of explicit SIMD code versus code auto-vectorized by the HotSpot JIT compiler.

Our work focuses on the JVM and Intel SIMD intrinsics functions, but would equally apply to other platforms.
2.2 BACKGROUND

In this section we provide an overview of x86 intrinsics functions and discuss the interworking of the existing implementations of the Java Virtual Machine, at the time of writing of this dissertation.

2.2.1 Intrinsics

Intrinsics are compiler-built-in functions that usually map into a single or a small number of assembly instructions. During compilation, they are inlined to remove calling overhead. This way they provide the programmer with assembly-like functionality, without having to worry about register allocation and instruction scheduling. SIMD intrinsics give access to data parallel instructions in vector ISAs, such as NEON on ARM processors, or the SSE and AVX families on Intel.

We focus on the x86 architecture and the associated SIMD intrinsics that are available in modern C/C++ compilers, such as GCC, Clang/LLVM, and Intel ICC. Specifically, these include the following ISAs:

- **MMX** - operating on 64-bit wide registers; provides integer operations only.

- **SSE / SSE2 / SSE3 / SSSE3 / SSE4.1 / SSE4.2** - operating on 128-bit wide registers; provides integer, 32-bit, and 64-bit floating point operations, string operations and cache and memory management operations.

- **AVX / AVX2** - ISAs that expand the SSE operations to 256-bit wide registers and provide extra operations for manipulating non-contiguous memory locations.

- **FMA** - an extension to SSE and AVX ISAs to provide fused multiply add operations.

- **AVX-512** - extends AVX to operate on 512-bit registers and consists of multiple parts called F / BW / CD / DQ / ER / IFMA52 / PF / VBMI / VL.

- **KNC** - the first production version of Intel’s Many Integrated Core (MIC) architecture that provides operations on 512-bit registers.

Additionally, we also include:
SVML - an intrinsics short vector math library, built on top of the ISAs mentioned above.


These ISAs yield a large number of associated intrinsics: arithmetics operations on both floating point and integer numbers, intrinsics that operate with logical and bitwise operations, statistical and cryptographic operations, comparison, string operations and many more. Table 2.1 gives a rough classification of the different classes of intrinsics. To ease the life of the developers, Intel provides an interactive tool called Intel Intrinsics Guide [25] where each available intrinsics is listed, including a detailed description of the underlying ISA instruction. Table 2.1 shows the number of available intrinsics: 5912 in total (of which 338 are shared between AVX-512 and KNC), classified into 24 groups.

2.2.2 Java Virtual Machines

There are many active implementations of the JVM including the open source IBM V9 [26], Jikes RVM [27], Maxine [28], JRockit [29], and the proprietary SAP JVM [30], CEE-J [31], JamaicaVM [32]. HotSpot remains the primary reference JVM implementation that is used by both Oracle Java and OpenJDK. Each of the JVM implementations provides support for Java Standard Edition or Micro Edition, tailored for a particular need:
<table>
<thead>
<tr>
<th>Arithmetics</th>
<th>Shuffles</th>
<th>Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm256_add_pd</td>
<td>_mm256_permutevar_pd</td>
<td>_mm_avg_epu8</td>
</tr>
<tr>
<td>_mm256_hadd_ps</td>
<td>_mm256_shufflehi_epi16</td>
<td>_mm256_cdfnorm_pd</td>
</tr>
<tr>
<td>_mm512_rem_epu8</td>
<td>_mm_blend_epi16</td>
<td>_mm512_erf_ps</td>
</tr>
<tr>
<td>_mm_rcp_ps</td>
<td>_mm256_unpacklo_pd</td>
<td>_mm_erfcinv_ps</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compare</th>
<th>String</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_cmp_epi16_mask</td>
<td>_mm_cmpestrm</td>
<td>_mm512_or_pd</td>
</tr>
<tr>
<td>_mm_cmp_eq_epi8</td>
<td>_mm_cmpistrz</td>
<td>_mm256_andnot_pd</td>
</tr>
<tr>
<td>_mm512_cmpnle_pd_mask</td>
<td>_mm_cmpistri</td>
<td>_mm512_xor_epi64</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Crypto</th>
<th>Bitwise</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_aesdec_si128</td>
<td>_mm256_bslli_epi128</td>
<td>_rdrand16_step</td>
</tr>
<tr>
<td>_mm_sha1msg1_epu32</td>
<td>_mm256_lzcnt_epi64</td>
<td>_rdseed64_step</td>
</tr>
<tr>
<td>_mm_aeskeygenassist_si128</td>
<td>_mm512_rol_epi32</td>
<td>_rdrand32_step</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loads</th>
<th>Stores</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_i32gather_epi32</td>
<td>_mm512_storenrrng0_pd</td>
<td>_mm256_castps_pd</td>
</tr>
<tr>
<td>_mm256_broadcast_ps</td>
<td>_mm_store_pd1</td>
<td>_mm256_cvtps_epi32</td>
</tr>
<tr>
<td>_mm512_loadunpackhi_epi32</td>
<td>_mm256_stream_ps</td>
<td>_mm_cvtph_ps</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

| Table 2.1: Simplified classification of the x86 SIMD intrinsics set. |
either a particular target machine or microarchitecture, embedded systems, operating system, provides additional garbage collector, resource control or parallelism model. However, none of the active JVM implementation provides any support for explicit vectorization nor intrinsics, nor permits inlining of assembly code directly in the Java source due to portability specifications.

The HotSpot JVM, which is the focus of this study, provides JIT compilation of Java bytecode as a black box. The developer has no control over, nor receives any feedback on the compilation phases except through coarse-grained command-line and debug options [33]. There are two flavors of the VM: a client mode focused on latency, and a server mode tuned for throughput. We only focus on the Server VM, as it is tuned to maximize peak operating speed. The Server VM offers a tiered compilation of bytecode using the C1 and C2 compilers. C1 is a fast, lightly optimizing bytecode compiler, while C2 performs more aggressive optimizations. When JVM applications are started, the HotSpot VM starts interpreting bytecode. It detects computation-intensive hot spots in the code via profiling, and proceeds to compile the bytecode of frequently used functions with C1. Once further thresholds are reached, functions may be compiled using C2. C2 supports autovectorization, using Superword Level Parallelism (SLP) [34]. SLP detects groups of isomorphic instructions and replaces them with SIMD instructions, which results in a lightweight vectorization. The SLP approach is limited and cannot optimize across loop iterations, nor can it detect idioms such as reductions.

2.3 SIMD INTRINSICS ON MANAGED LANGUAGE RUNTIMES

In this section, we present our two-tier approach for making the Intel SIMD intrinsics available in the JVM. First we automatically generate SIMD eDSLs, each implemented as a Scala class that corresponds to one of the 13 vector ISAs in Figure 2.1. Then, we show how to use these eDSLs to generate high-performance SIMD code with high-level language constructs inherited by the host language. We show examples of end-user code in Scala, but any other JVM language could be used.
2.3.1 Automatic Generation of ISA-specific DSLs

Before we start, we have to establish a corresponding type system between the JVM and the SIMD intrinsics functions to represent the SIMD vector types, that is required for the generation of the eDSLs and their usage.

**Type System for SIMD Intrinsics in the JVM.** The JVM has no notion of SIMD vector types, thus we build abstract classes to mark the type of DSL expressions that represent SIMD intrinsics functions in LMS as shown in Figure 2.2.

```
1 Rep[___m64] // MMX integer types
2 Rep[___m128] // SSE 4x32-bit float
3 Rep[___m128d] // SSE 2x64-bit float
4 Rep[___m128i] // SSE 2x64 / 4x32 / 8x16 / 16x8-bit integer
5 Rep[___m256] // AVX 8x32-bit float
6 Rep[___m256d] // AVX 4x64-bit float
7 Rep[___m256i] // AVX 4x64 / 8x32 / 16x16 / 32x8-bit integer
8 Rep[___m512] // AVX512 16x32-bit float
9 Rep[___m512d] // AVX512 8x64-bit float
10 Rep[___m512i] // AVX512 8x64 / 16x32 / 32x16 / 64x8-bit integer
```

**Figure 2.2: Vector Primitive Types in LMS**

SIMD intrinsics functions take primitive arguments that correspond to low-level C/C++ primitive types. The primitive types in the JVM exhibit a fixed width, and therefore a direct mapping can be established with C/C++ primitives. Some intrinsics however, require the use of unsigned types that are not supported natively in the JVM:

```
1 unsigned int _mm_crc32_u16 (unsigned int, unsigned short)
```

To mitigate this problem, we use the Scala Unsigned [35] package, which implements unsigned types and operations on top of the signed types available in the JVM. Table 2.2 shows the type mapping between the 12 primitives, which in most cases is straightforward, except for JVM Char that maps to int16_t to support UTF-8. Arrays of primitive types in the JVM and C/C++ code are isomorphic and both represent continuous memory space of a certain primitive type. Therefore Array[T] maps to a memory pointer T* in the low-level SIMD intrinsics.
2.3 SIMD Intrinsics on Managed Language Runtimes

JVM Types ↔ C/C++ Types

<table>
<thead>
<tr>
<th>JVM Type</th>
<th>C/C++ Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float</td>
<td>float</td>
</tr>
<tr>
<td>Char</td>
<td>int16_t</td>
</tr>
<tr>
<td>Double</td>
<td>double</td>
</tr>
<tr>
<td>Boolean</td>
<td>bool</td>
</tr>
<tr>
<td>Byte</td>
<td>int8_t</td>
</tr>
<tr>
<td>UByte</td>
<td>uint8_t</td>
</tr>
<tr>
<td>Short</td>
<td>int16_t</td>
</tr>
<tr>
<td>UShort</td>
<td>uint16_t</td>
</tr>
<tr>
<td>Long</td>
<td>int64_t</td>
</tr>
<tr>
<td>ULong</td>
<td>uint64_t</td>
</tr>
<tr>
<td>Int</td>
<td>int32_t</td>
</tr>
<tr>
<td>UInt</td>
<td>uint32_t</td>
</tr>
</tbody>
</table>

Table 2.2: Type mappings between JVM and C/C++ types.

Automatic Generation of ISA-Specific eDSLs

LMS provides a relatively simple interface to define eDSLs, but adding more than 5000 functions by hand would be tedious and error-prone. Our approach generates the LMS eDSLs automatically from the XML specification provided by the Intel Intrinsics Guide; these are then packed as a jar file that is later published in the Maven Central Repository [36] for deployment. At the time of writing, we use the latest version of the intrinsics specifications stored as data-3.3.16.xml file, and build the generator such that it anticipates future extensions in the specifications. Figure 2.3 shows a high-level overview of the generation process, which we explain step-by-step next.

Parse XML Intrinsics Specification. The first step in the generation process extracts the information from the XML file. As shown in an example (Figure 2.4), for each intrinsics, the XML file contains a name that defines the intrinsics function, return type, ordered list of each argument of the function with the corresponding type, CPUID parameter, that correspond to the ISA set, and a category parameter.

Generate ISA-Specific DSL in LMS. For each intrinsic function we implement four building blocks that define the eDSL. These are represented in terms of implementation classes provided by the LMS framework. The classes Def[T] and Exp[T] together define a computation graph. Subclasses of Def[T] implement graph nodes that represent individual computations, e.g., Plus(a, b). Here, a and b are values of type Exp[T]: either constants Const(.) or symbols Sym(id) that refer to other graph nodes through a numeric index id. The four necessary building blocks are as follows:
Figure 2.3: Generating SIMD intrinsics eDSLs from vendor specification.
Figure 2.4: XML specification of the _mm256_add_pd intrinsic.

1. Definition of the intrinsic function represented as a subclass of Def[T].
2. Implicit conversion from expression Exp[T] to definition Def[T], looking up a computation node given a symbolic reference.
3. Mirroring function that converts a Def[T] into expression Exp[T], potentially applying a transformation.
4. Unparsing routine that converts each Def[T] into C/C++ string.

To complete the first part, we define IntrinsicDef[T], an abstract class that each intrinsics definition will inherit:

```scala
abstract class IntrinsicDef[T:Manifest] extends Def[T] {
  val category : List[IntrinsicsCategory]
  val intrinsicType : List[IntrinsicsType]
  val performance : Map[MicroArchType, Performance]
  val header : String
}
```

Then for each intrinsic function, we define a Scala case class that corresponds to the intrinsics function’s name, its input arguments and return type. Each case class contains the category, the type of the intrinsics and performance informations when available. Additionally, we also include the header where the C/C++ intrinsics is defined:
case class MM256_ADD_PD(a: Exp[__.m256d], b: Exp[__.m256d]) extends IntrinsicsDef[__.m256d] {
  val category = List(Arithmetic)
  val intrinsicType = List(FloatingPoint)
  val performance = Map.empty[MicroArchType, Performance]
  val header = "immintrin.h"
}

With the current definition we allow a particular intrinsics to pertain to several categories. The header information gives us the control to include the correct header when unparsing the code to C/C++ code. Performance information is included but is not used in the staging process.

Next we generate Scala code for the implicit conversion from intrinsics expressions to definitions. This routine is essential in LMS, as it provides automatic conversion of the staged code into static single assignment (SSA) form. In most cases it is sufficient to rely on the Scala compiler to automatically perform the implicit conversion:

def _mm256_add_pd(a: Exp[__.m256d], b: Exp[__.m256d]) : Exp[__.m256d] = MM256_ADD_PD(a, b)

The LMS framework supports DSL transformations by substitution. Once a substitution is defined, LMS creates new definitions. However, when no substitution is available, a definition has to be converted to an expression through a routine of mirroring that converts a Def[T] back to Exp[T], potentially creating a new definitions for subexpressions as part of the transformation:

override def mirror[A:Typ](e: Def[A], f: Transformer)(implicit pos: SourceContext): Exp[A] = (e match {
  case MM256_ADD_PD (a, b) => _mm256_add_pd(f(a), f(b))
  case MM256_ADD_PS (a, b) => _mm256_add_ps(f(a), f(b))
  // ... a lot more patterns to match
  case _ => super.mirror(e, f)
})

Once code is generated for all these routines and for each intrinsic, the final step is to generate code to perform the unparsing of the DSL into C code. The unparsing routine is done similarly to the mirroring routine, by pattern matching each DSL definition to produce the corresponding C expression:
override def emitNode(s: Sym[Any], r: Def[Any]) = r match {
  case iDef@MM256_ADD_PD(a, b) => {
    headers += iDef.header
    emitValDef(sym, s"..mm256_add_pd(${a}, ${b})")
  }
  // ... a lot more patterns to match
  case _ => super.emitNode(sym, rhs)
}

Infer intrinsic mutability  As mentioned before, when code is generated for the implicit conversion of an intrinsics expression to the intrinsics definition, we can rely on the Scala compiler to match the correct implicit method. This works correctly for immutable expressions, but not all intrinsics are immutable. For example, each intrinsics that loads and stores from/to memory creates effects that have to be handled by LMS. The semantic of these effects is essential in scheduling the DSL.

To resolve this problem, we use the category information of each intrinsics (see Figure 2.4), and implement a conservative heuristic to generate the effects:

- Each time an intrinsics is discovered with a load category, we generate a read effect on each argument that is a memory location.
- Each time an intrinsics is discovered with a store category, we generate a write effect on each argument that is a memory location.

For example, an AVX load of 4 doubles has the form of:

```scala
def _mm256_load_pd[A[_], U:Integral]{
  (implicit cont: Container[A]): Exp[__.m256d] = {
    cont.read(mem_addr)
    (MM256_LOAD_PD(mem_addr, mem_addrOffset)
    (implicitly[Integral[U]], cont))
  }
```

The heuristics is invoked on each intrinsics that performs loads, stores, maskstores, maskloads, gather, scatters and other intrinsics that perform memory-related operations.

Split each isa specific DSL into subclasses  The JVM has a hard limit of 64KB on the size of each method, which is an obstacle in generating
the unparsing and mirroring routines for large ISA, such as AVX-512 or KNC. To avoid this obstacle, and still keep the LMS design pattern, we decided to split the ISA specific DSLs into subclasses that inherit each other.

2.3.2 Developing Explicitly Vectorized Code in the JVM

Figure 2.5 gives a high-level overview of how to use explicit vectorization in the JVM. The process consists of two parts: compile-time tasks, done by the high-performance code developer, and runtime tasks that are done automatically by LMS and our compiler pipeline. Specifically, the compile-time tasks of the developer comprise four steps:

1. Implement a native function placeholder that will represent the vectorized code.

2. Create a DSL instance by instantiating one or mixing several ISA-specific eDSLs.

3. Implement the SIMD logic as a staged function.

4. Call the provided compile routine to generate, compile and link the code in the JVM.

After the four steps are completed, and the JVM program is started, the compiler pipeline is invoked with the compile routine. This will perform system inspection, search for available compilers and opportunistically pick the optimal compiler available on the system. In particular, it will attempt to find icc, gcc or llvm/clang. After a compiler is found, the runtime will determine the target CPU, as well as the underlying micro-architecture to derive available ISAs. This allows us to have full control over the system, as well as to be able to pick the best mix of compiler flags for each compiler.

Once this process is completed, the user-defined staged function is executed, which assembles a computation graph of SIMD instructions. From this computation graph, LMS generates vectorized C code. This code is then automatically compiled as a dynamic library with the set of derived compiler flags, and linked back into the JVM. To link the native code into the JVM, JNI requires the C functions header to contain the Java_ prefix, followed by package name, class name and name of the native function. The compile routine automates this process using JVM reflection and some lightweight use of Scala macros. By this automation, we ensure the interoperability between the native function and the staged function, creating
Start developing SIMD code

1. Implement a native function placeholder
2. Mixin one or several ISA-specific eDSLs
3. Implement the SIMD staged function
4. Call compile to generate native code

Start the Java Virtual Machine (JVM)

- Detect available C/C++ compilers
- Inspect the system through CPUID
- Infer available ISAs and compiler flags
- LMS: remove abstraction & generate C code
- Compile and link the code to the JVM

Use high performance SIMD code

Figure 2.5: Developing explicit vectorized code in the JVM using SIMD eDSLs.
code robust to modifications and refactoring and eliminate the need for the developer to recompile the native code each time major code revision are performed on the low-level code or the class container.

Figure 2.6 illustrates a complete and self-contained implementation of a BLAS 1 routine called SAXPY [37], which computes $y = y + \alpha x$ for given vectors $x, y$ and scalar $\alpha$. The expression `forloop(...)` creates a staged loop in the LMS computation graph.

2.4 RESULTS

To evaluate the robustness of the automatic SIMD eDSL generator, we test it against older versions of the hardware specifications provided by the vendor. To assess the viability of our approach in the JVM, we consider two ubiquitous kernel functions: the aforementioned SAXPY and matrix multiplication. Finally, we discuss the limitation of our approach, when applied to managed language runtimes.

2.4.1 Evaluation

**Experimental setup.** We perform the tests on a Haswell enabled processor Intel Xeon CPU E3-1285L v3 3.10GHz with 32GB of RAM, running Debian GNU/Linux 8 (jessie), kernel 3.16.43-2+deb8u3. The available compilers are `gcc 4.9.2-10` and Intel `iccc 17.0.0`. The installed JVM is HotSpot 64-Bit Server 25.144-b01, supporting Java 1.8. To avoid the effects of frequency scaling and resource sharing on the measurements, Turbo Boost and Hyper-Threading are disabled.

We use ScalaMeter [38] to perform the benchmarks. To obtain precise results, we select a pre-configured benchmark that forks a new JVM virtual machine and performs measurements inside the clean instance. The new instance has a compilation threshold of 100 (`-XX:CompileThreshold=100`) and we perform at least 100 warm-up runs on all test cases to trigger the JIT compiler. Each test case is performed on a warm cache. Tests are repeated 30 times, and the median of the runtime is taken. We show the results as performance, measured in flops per cycle.

We use `-XX:UnlockDiagnosticVMOptions` to unlock the diagnostic JVM options, which enables us to inspect the JIT compilation of the HotSpot JVM, and `-XX:CompileCommand=print` to output the generated assembly. In all test cases we observe the full-tiered compilation starting from the C1 compiler to the last phase of the C2 compiler. For a fair comparison between
import ch.ethz.acl.commons.cir.IntrinsicsIR
import com.github.dwickern.macros.NameOf_

class NSaxpy {

  // Step 1: Placeholder for the SAXPY native function
  @native def apply {
    a: Array[Float], b: Array[Float], scalar: Float, n: Int
  } Unit

  // Step 2: DSL instance of the intrinsics
  val cIR = new IntrinsicsIR; import cIR._

  // Step 3: Staged SAXPY function using AVX + FMA
  def saxpy_staged(
    a_imm : Rep[Array[Float]], b : Rep[Array[Float]],
    scalar : Rep[Float] , n : Rep[Int]
  ): Rep[Unit] = {
    import ImplicitLift._
    val a_sym = a_imm.asInstanceOf[Sym[Array[Float]]]
    val a = reflectMutableSym(a_sym) // make array ‘a’ mutable
    // start with the computation
    val n0 = (n >> 3) << 3
    val vec_s = _mm256_set1_ps(scalar)
    forloop(0, n0, fresh[Int], 8, (i : Rep[Int]) => {
      val vec_a = _mm256_loadu_ps(a, i)
      val vec_b = _mm256_loadu_ps(b, i)
      val res = _mm256_fmadd_ps(vec_b, vec_s, vec_a)
      _mm256_storeu_ps(a, res, i)
    })
    forloop(n0, n, fresh[Int], 1, (i : Rep[Int]) => {
      a(i) = a(i) + b(i) * scalar
    })
  }

  // Step 4: generate the saxpy function,
  // compile it and link it to the JVM
  compile(saxpy_staged _, this, nameOf(apply _))
}

Figure 2.6: A complete implementation of the BLAS 1 routine SAXPY in the JVM using AVX and FMA SIMD intrinsics.
the JVM and our generated intrinsics code, we consider the C2 compiled version of the bytecode only, excluding the JIT warm-up time and the LMS generation overhead.

**SAXPY**  We compare the generated SAXPY vector code, shown in Figure 2.6, against an equivalent Java implementation.

```java
public class JSaxpy {
    public void apply(float[] a, float[] b, float s, int n) {
        for (int i = 0; i < n; i += 1)
            a[i] += b[i] * s;
    }
}
```

Figure 2.7 shows the performance comparison. First we note the similarity in performance, which is not surprising since SAXPY has low operational intensity and the simplicity of the code enables efficient autovectorization.

The assembly diagnostics shows that, even though the Haswell is equipped with AVX and FMA instructions, the JVM will autovectorize the computation using SSE and ignore FMA, but will load and store the data using AVX instructions:

```
0x00007f0dc07064f0: vmovdqu 0x10(%rcx,%r9,4),%xmm2
```
For small sizes that are L1 cache resident the Java implementation does better. This is because JNI methods are not inlined and incur additional cost to be invoked.

**Matrix-Matrix Multiplication (MMM).** For the second benchmark we chose MMM, which has a high operational intensity and is known to benefit from various optimizations such as blocking and vectorization [39]. We consider three versions. The first is a standard Java implementation of a triple MMM loop. The two other versions are a blocked version of MMM, with block size of 8, the first implemented in Java, and the second implemented using AVX intrinsics in Scala. For simplicity, we assume that the matrix has size \( n = 8k \), and provide the implementation in Figure 2.9.

Our Scala implementation uses the SIMD intrinsics with high level constructs of the Scala language, including pattern matching (lines 5, 10, 19, 20, etc), lambdas (lines 4, 10, 34), Scala collections (lines 4, 34, etc), closures (line 45) and others that are not available in low-level C code. Once LMS removes the abstraction overhead, the MMM function results in a high-

---

**Figure 2.8: Java implementation vs LMS intrinsics generated code: MMM (Matrix-Matrix-Multiplication)**

---

```plaintext
0x00007f0dc07064f7: vmulps %xmm1,%xmm2,%xmm2
0x00007f0dc07064fb: vaddps 0x10(%rdx,%r9,4),%xmm2,%xmm2
0x00007f0dc0706502: vmovdqu %xmm2,0x10(%rdx,%r9,4)
```
// take 8x__m256 vectors, transpose their values & return 8x__m256 vectors.

// Take 8x__m256 vectors, transpose their values & return 8x__m256 vectors.

def transpose(row: Seq[Exp[__m256]]) = {
    val __tt = row.grouped(2).toSeq.flatMap{
        case Seq(a, b) => Seq(_mm256_unpacklo_ps(a, b), _mm256_unpackhi_ps(a, b))
    }).grouped(4).toSeq.flatMap{
        case Seq(a, b, c, d) => Seq(
            _mm256_shuffle_ps(a, c, 68), _mm256_shuffle_ps(a, c, 238),
            _mm256_shuffle_ps(b, d, 68), _mm256_shuffle_ps(b, d, 238)
        )
    }
    val zip = __tt.take(4) zip __tt.drop(4)
    val f = _mm256_permute2f128_ps _
    zip.map({ case (a, b) => f(a, b, 0x20) }) ++
    zip.map({ case (a, b) => f(a, b, 0x31) })
}

// Perform Matrix-Matrix-Multiplication

// Perform Matrix-Matrix-Multiplication

// Perform Matrix-Matrix-Multiplication

Figure 2.9: Implementation of MMM in the JVM using AVX intrinsics
Table 2.3: Intel Intrinsics Guide XML specifications.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-3.2.2.xml</td>
<td>03.09.2014</td>
</tr>
<tr>
<td>data-3.3.1.xml</td>
<td>17.10.2014</td>
</tr>
<tr>
<td>data-3.3.11.xml</td>
<td>27.07.2015</td>
</tr>
<tr>
<td>data-3.3.16.xml</td>
<td>26.01.2016</td>
</tr>
<tr>
<td>data-3.4.xml</td>
<td>07.09.2017</td>
</tr>
</tbody>
</table>

performance implementation. The performance comparison in Figure 2.8 shows that the use of explicit vectorization through SIMD intrinsics can offer improvements up to 5x over the blocked Java implementation, and over 7.8x over the baseline triple loop implementation.

The assembly analysis shows that the C2 compiler will unroll the hot loops in both Java versions, but does generate SIMD instructions, which explains the low performance.

**Automatic SIMD eDSL Generator.** The predecessor of the Intel Intrinsics Guide web application was a Java application sharing the same name. The older versions of both Java and the web application contained older version of the intrinsics specifications, e.g., without AVX-512. However, Intel does not offer these versions, and continuously updates the XML specifications, improving the description / performance of each intrinsic function.

Using tools such as the Wayback Machine, a digital archive that mirrors web-site states at a given date, we were able to salvage older, pre-captured iterations of the intrinsics specifications, shown in Table 2.3. Then we instructed our eDSL generator to re-generate each ISA-specific eDSL.

Our results show that our eDSL generator is robust towards minor changes on the XML specifications, being able to retrospectively generate eDSLs for recent years. We believe that if Intel uses the same XML schema for new releases, our generator should be robust to new ISA updates, as long as the new ISA has similar properties than its predecessor.
2.4.2 Limitations

Our approach provides low-level control for performance optimizations to the Java developer but comes at a price. We discuss a number of technical issues that would be good to resolve to further improve ease-of-use and maintainability.

Currently, there is no mechanism to ensure the isomorphism between the native function placeholder and the staged function. As a result, it is the responsibility of the developer to define this isomorphic relation upon compile time. The current use of Scala macros makes the code robust in terms of refactoring and modifications, which is quite convenient compared to manually maintaining isomorphism between native C/C++ and JVM code. A more diligent use of Scala macros could potentially resolve this problem and ensure complete isomorphic binding of JNI and staged functions.

LMS does not provide any mechanism to deal with exceptions such as segfaults from generated code. Therefore it is the responsibility of the developer to write valid SIMD code. LMS is also not optimized for fast code generation, which might result in an overhead surpassing the HotSpot interpretation speed when used to generate functions that are computationally light.

Another limitation is a consequence of the complex memory model in the HotSpot JVM. Once arrays are used in the native code, API calls such as `GetPrimitiveArrayCritical` must be invoked to obtain the memory space of the array. Depending on the state of the garbage collector (GC), the array might end up on different segments on the heap, which could result in a copy once the native code tries to access the memory space, or the JVM could decide to temporary disable the GC. Although we did not experience an array copy in any test case performed, we believe that the use of LMS intrinsics is best suited for compute-bound problems, where the copy overhead can be leveraged by the fast runtime of SIMD instructions upon each JNI invocation. Some of the issues with JVM arrays can be avoided by using Java NIO buffers or off-heap memory allocated with the `sun.misc.Unsafe` package.

Finally, to effectively use the explicit vectorization in the JVM, we need a compiler installed on the target machine, as well as the JDK version of the JVM to compile and link the native code.
2.5 RELATED WORK

EXPLICIT VECTORIZATION IN THE JVM. The first approach to expose data parallelism was the implementation of the Java Vectorization Interface (JVI) as part of the Jitrino JIT compiler [40]. JVI is designed as an abstract vector interface that provides set of methods as vector operators. These methods are later compiled to different vector instructions, such as SSE and AVX. The approach offers competitive results in some case, but is limited in the SIMD instructions it supports and subsequent iterations of post-AVX ISAs. Similarly to JVI, Oracle has ongoing research developing cross-platform APIs that can leverage SIMD instructions. Implemented as part of an experimental JVM called Panama [41], SIMD instructions are used in immutable vector types, parameterized by element type and size. Similarly to JVI, Panama also suffers from limited support of vector ISAs, and requires a specific JVM. Both approaches abstract SIMD instructions, which limits the ability of a developer to tune the code for a particular microarchitecture.

AUTOVECTORIZATION IN THE JVM. Initially introduced in the Jikes RVM [27], the HotSpot JVM uses SLP [34] based autovectorization. SLP is limited and is only able to vectorize basic blocks consisting of groups of isomorphic instructions, generating SSE and AVX code. Partial support of FMA and AVX-512 is only planned for Java 9 [41].

SUPPORT OF LOW-LEVEL CODE IN THE JVM. Sulong [42] is a system to execute low-level languages such as C/C++, Fortran, Ada, and Haskell in the JVM. Sulong is capable of handling low-level languages that compile to LLVM, using LLVM IR interpreter built on top of the Truffle framework [43], running on the Graal VM. While this approach can bring a support of low-level instructions in the JVM, it does not support SIMD instructions, as Graal does not provide sufficient analysis for vectorization. Furthermore, due to interpretation, Sulong is shown to be outperformed by native compilers such as gcc.

AUTOMATIC GENERATION OF DSLS IN LMS. DSLs have been generated into LMS before. Yin-Yang [44] automatically generates deep DSL embeddings from their shallow counterparts by reusing the core translation. Forge [45] generates DSLs from a declarative specification. None of the approaches have been challenged to generate DSLs of the scale imposed by the
large amount of SIMD intrinsics, nor were they designed to automatically infer effects of mutability.

SIMD INTRINSICS IN LMS. A limited support of SIMD instructions has been introduced while abstracting vectors architectures [46]. This approach has been used in generating libraries for high-performance code, and integration with the JVM has not been demonstrated. On an even lower level, LMS has been used to define domain-specific ISAs by generate specialized hardware [47, 48].

2.6 SUMMARY

This chapter shows how metaprogramming techniques can be used to bridge the gap between high-level managed languages and the need to access low-level instructions in high performance code development. Specifically, we showed how to provide access to SIMD intrinsics in the HotSpot JVM, thus eliminating the need to write C/C++ code. Two key techniques underlie our approach. First is the use of embedded DSLs to express intrinsics inside JVM languages such as Scala. These are generated directly from the vendor XML specification, which enables complete intrinsics support and fast updates in the future. Second is the use of staging to convert SIMD intrinsics interspersed with Scala code into high-performant C kernels, which are then compiled and linked via JNI. The challenge in our work is in the systematic handling of large sets of functions, converting them into sets of DSLs, automatically inferring their side effects, and creating a compiler and code generation pipeline for convenient and productive development. We show how the SIMD support in the JVM can interact with existing high-level language features, and offer speed over autovectorized HotSpot JIT compiler.
In the previous chapter, we provided a complete support of SIMD instructions in LMS and demonstrated their usage in developing efficient code in conjunction with high-level language features and constructs. Apart from being used as embedded DSLs only, the new SIMD intrinsics can be composed with other DSLs, or can be used as basic blocks in building new abstractions on top of them. In that regard, this chapter explores the following question:

• How can we use the SIMD intrinsics in LMS to build new low-level abstractions and provide functionality that is typically implemented by low-level compilers?

To demonstrate the use of data-parallelism in LMS, we focus on prototypical study in abstracting low-precision arithmetic based on stochastic quantization. To evaluate the approach, we use the designed abstractions to implement gradient descent (GD) and compressive sensing algorithms (CS). As a side effect of this practical application, we also explore the question:

• Can these applications obtain performance benefits from low-precision stochastic quantization on state-of-the-art CPUs?

The chapter is organized as follows. Section 3.1 motivates the rationale of using low-precision data representation. Section 3.2 provides the mathematical background on implementing GD and CS algorithms, as well as stochastic quantization. Section 3.3 provides multiple implementations of the mathematical routines for the GD and CS algorithms, operating on low precision data: native implementation and LMS implementation that is used to build virtual, ISA-like, low-level abstraction that operates on top of existing ISA instructions. Section 3.4 evaluates the approach, while Sections 3.5 and 3.6 provide an overview of the related work and conclusion.
Part of the work in this chapter, focusing on low-level abstractions, was published at the 2018 International Symposium on Code Generation and Optimization [24]. The part focusing on the performance benefits on state-of-the-art CPUs was published at 2018 IEEE Workshop on Signal Processing Systems [49].

3.1 MOTIVATION

Recent progress in data processing and machine learning is due not only to the availability of large datasets and the introduction of better algorithms, but also to vastly improved system support for efficient computation. Software frameworks such as Google’s TensorFlow [50] leverage multi-core and multi-node parallelism, as well as single-processor pipelining techniques.

Low-precision data representation has emerged as an important design point for speeding up both computation and communication in machine learning applications. For instance, NVIDIA’s latest GPU family natively supports computation at 16-bit and 8-bit precisions, while TensorFlow can perform inference over models quantized to as little as 4-bit precision per component [50]. This shift is not entirely surprising: fundamental methods in optimization and sparse recovery are well equipped to dealing with noise in data and measurements; low-precision provides an additional source of quantization noise, which such methods may be able to support as well.

Recent theoretical advances [51–53] support this intuition: gradient methods can support low-precision data and computation and still converge [52]; similar results are known for compressive sensing via specific recovery techniques [53–55]. However, these results require stochastic quantization (defined in detail later), needing more careful implementation than deterministic quantization (e.g., as used in [50]), which is not known to provide convergence guarantees.

CONTRIBUTIONS. With the aim to reason about the implementation of low-level abstractions in LMS, and whether applications can take advantage of low precision stochastic quantization on state-of-the-art CPUs, we provide two types of contributions:

1. A native implementation, as a C/C++ library, called Clover, that supports the basic mathematical routines needed in optimization and sparse recovery with guaranteed convergence.
An LMS implementation of a stochastic quantization library, that is build using virtual, ISA-like, low-level abstractions, that runs on the HotSpot JVM, where access to low-level instructions is not available.

In particular, we provide efficient implementations of the dot product, scale and add, and matrix-vector multiplication in 4-bit quantized, 8-bit quantized, 16-bit half-precision and 32-bit floating point. For the native version, we also provide an efficient mixed 4,8-bit precision MVM, with the matrix represented in 4-bit quantized and the vectors represented in 8-bit quantized. Our low-level abstraction and native library faithfully implement the variants of stochastic quantization defined in e.g. [52, 53], and therefore enjoys the analytical guarantees provided by these methods.

Another, incidental, but significant contribution is the efficient implementation of the quantized 4-bit computations on Intel AVX since there is no native support for this data format. (In fact, recent work [56] posited that 4-bit precision could not be supported efficiently on x86 in the absence of native support.)

In a first set of benchmarks, we show that our native library and low-level LMS abstraction provide significant speed-ups for the low precision versions in memory bound situations. In particular for MVM, we achieve a speed-up linear in the precision reduction, e.g., 8x for 4-bit over 32-bit, due to reduced data movement.

In a second set of benchmarks we evaluate our library on two important applications: convex optimization via gradient descent (GD), and reconstruction in compressive sensing via iterated hard thresholding (IHT). First, we show that they inherit the linear (in the precision reduction) speed-up of MVM when run with reduced precision. Then, we show that also an absolute, end-to-end speed-up is attainable with precision reduction, i.e., the 4- and 8-bit versions can converge faster in terms of wall clock time to a given target error, since the number of extra iterations needed to convergence is typically smaller than any slowdown due to lowering precision.

3.2 BACKGROUND

Quantization. We work with stochastic quantization to generate a low-precision version of an arbitrary vector $v$. In brief, stochastic quantization works as follows. Given the vector $v$, let $M(v)$ be a scaling factor such that:

$$-1 \leq v / M(v) \leq 1$$

(3.1)
Without loss of generality, let us set

\[ M(v) = \max_i |v_i| \]  

(3.2)

We partition the interval \([-1, 1]\) using \(s + 1\) separators:

\[-1 = l_0 \leq l_1 \ldots \leq l_s = 1\]  

(3.3)

which will be mapped to integer levels for efficient coding. Each component \(z\) in the normalized vector \(v/M(v)\), is quantized to one of its two nearest separators: \(l_i \leq z \leq l_{i+1}\). More precisely, we map \(z\) to \(l_{i+1}\) with probability \((z - l_i)/(l_{i+1} - l_i)\), and to \(l_i\) with probability \((l_{i+1} - z)/(l_{i+1} - l_i)\).

We denote the stochastic quantization function by \(Q(v,s)\) and choose the probability of quantizing to different separators such that in expectation values are preserved, i.e., \(E[Q(v,s)] = v\). We use \(Q(v)\) when \(s\) is clear from context.

**Compressive Sensing (CS)** is a standard technique [57–59] in sparse signal recovery. It offers a solid mathematical foundation and a range of efficient algorithms for the problem of recovering sparse signals from noisy, high-dimensional, incomplete samples, with a wide range of applications, such as imaging, spectroscopy, and data analysis.

Mathematically, CS assumes a sparse or sparsely-approximable signal \(x \in \mathbb{R}^n\), which is sampled via a linear operator \(\Phi^{m \times n}\) with \(m < n\). That is, we are given a vector of measurements \(y \in \mathbb{C}^m\), which can be expressed as

\[ y = \Phi x + e, \]  

(3.4)

where \(e\) is the observation noise. Under certain assumptions on the matrix \(\Phi\), as in [58], the sparsity constraint imposed on \(x\) can overcome this ill-posed problem, and allows approximate recovery of the original signal.

**Iterative Hard Thresholding.** At a high level, recovery algorithms for CS work by iteratively building up a sparse estimate \(x\) such that \(\Phi x\) approximates \(y\), i.e., \(\|y - \Phi x\|_2\) becomes small. There has been a tremendous amount of work on developing efficient recovery algorithms; for a survey, we refer the reader to [60]. In this work, we focus on a popular recovery algorithm called iterative hard thresholding (IHT) [61, 62].

Given the current iterate \(x^t\), where \(x^0\) is initialized to 0, the IHT update rule is

\[ x^{t+1} = H_s(x^t + \mu \Phi^\top (y - \Phi x^t)), \]  

(3.5)
where $H_s(x) : \mathbb{C}^n \to \mathbb{C}^n$ is the nonlinear thresholding operator that preserves the largest $s$ (in magnitude) entries of $x$ and sets the rest to zero, and $\mu > 0$ is a parameter we will call the step size. The convergence properties of this method and its variants are well studied [63], and the method can provide state-of-the-art recovery in some settings.

**Gradient Descent (GD)** is a fundamental method for convex optimization problems of the form

$$\min_{x \in \mathbb{R}^n} f(x). \quad (3.6)$$

GD solves problems of the form (3.6) by forming a sequence of estimators $(x^k)_{k \in \mathbb{N}}$ such that

$$x^{k+1} = x^k - \mu \nabla f(x^k), \quad (3.7)$$
given the initial point $x^0$ and fixed positive step size $\mu$, where $\nabla f(x^k)$ is the gradient of the function $f$ at point $x^k$. Theoretical results [64] guarantee that GD reaches $\epsilon$ accuracy after at most $O(1/\epsilon)$ iterations if $f$ is convex and differentiable and has a Lipschitz-continuous gradient.

In this chapter, we consider GD for solving least squares problems of the form (3.6) with $f(x) = \frac{1}{2} \|Ax - b\|^2$, where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

### 3.3 Low Precision Arithmetic

In this section we describe the implementation of the low-precision routines. Our implementation provide the linear algebra operations necessary for GD and IHT in 4-bit quantized, 8-bit quantized, 16-bit half precision, and 32-bit floating point arithmetic. Furthermore, it implements MVM using mixed 4,8-bit quantized arithmetic, where the matrix is represented in 4-bit quantized and the vectors are implemented in 8-bit quantized. We describe the implementation of the 4-bit quantized arithmetic in depth and then briefly describe the differences for the other datatypes.

The native implementation is implemented as a library called Clover. The low-level abstraction in LMS follows the same implementation principles as the native implementation, however it differs from the native implementation, as it is implemented in Scala, in a form of a variable ISA-like instructions, and runs on the HotSpot JVM. For effective comparison in the JVM, we also provide a pure Java implementation.
3.3.1 4-bit Arithmetic

Functionality for quantized 4-bit linear algebra operations is provided by the CloverVector4 and CloverMatrix4 containers. CloverVector4 contains a vector represented in a quantized 4-bit format. The values and the scaling factors are stored in separate contiguous blocks of memory, such that sub-vectors of length 64 share the same scale.

An illustration of the memory layout of a CloverVector4 container is shown in Fig. 3.4. Each value in the vector is stored as a two’s complement in a 4-bit nibble. The x86 architecture uses byte addressing, so addressing the nibbles must be done in software. We accomplish this explicitly in the application logic, storing two consecutive nibbles in a byte. As a result, the values in a CloverVector4 of length $n$ are represented as an array of $n/2$ bytes.

CloverMatrix4 contains a matrix represented similarly. The values and scales are stored as separate contiguous blocks of memory, both in row-major order. Sub-matrices of size $64 \times 64$ share the same scale.

We now describe the implementation of the individual operations on the CloverVector4 and CloverMatrix4 containers, providing more detail for the dot-product implementation as it is the most important for our applications.

Packing and unpacking of 4-bit values. The x86 architecture does not support 4-bit integer arithmetic. Thus if we have an array of 4-bit values, we must unpack, representing them as larger (i.e., 8, 16 or 32-bit) integers for computation. Then, for efficient storage and data movement, we pack these larger integers, representing them as 4-bit values stored within an array. We now describe how to efficiently pack and unpack nibbles to
Figure 3.2: Illustration of placing a two’s complement nibble in the low ordered four bits of a 32-bit integer.

and from a 32-bit integer, using the example in Fig. 3.2. Nibbles can be packed and unpacked into integers of other sizes similarly.

The goal of unpacking from a 32-bit integer (right part of Fig. 3.2) is as follows: We start with a 32-bit entity that stores eight nibbles. We wish to extract a single 4-bit nibble and represent it as a 32-bit integer. This can be done with two bit shifts: (a) a logical left shift is used to shift the nibble so that it occupies the highest-order 4-bits of the 32-bit entity. (b) an arithmetic right shift is used to shift the nibble to the lowest order 4-bits of the 32-bit entity. The arithmetic right shift has sign extension, filling the high-order 28 bits with the sign bit of the nibble, yielding a 32-bit integer with the same value as the two’s complement 4-bit value.

The goal of packing (left part of Fig. 3.2) is to revert the unpacking operation. Two bit shifts are used to place the lowest order 4 bits of a 32-bit integer anywhere within a 32-bit entity. (a) a logical left shift is used to shift the nibble so that it occupies the highest-order 4-bits of the 32-bit entity. (b) a logical right shift is used to shift the nibble to the desired location within the 32-bit entity. The first sets the bits lower-ordered than the nibble to zero, and the second sets the bits higher-ordered than the nibble to zero. A bitwise OR operation is then used to store up to eight nibbles in the 32-bit entity.

**Quantization.** Quantization converts a matrix or vector of 32-bit floating point values into a CloverVector4 or CloverMatrix4. To illustrate, we
consider the process of quantizing the vector container. After quantization
an array of 32-bit floating point value into a CloverVector4, each 32-bit
floating point value will be represented as the product of a 4-bit signed in-
teger in the range $[-7, 7]$ and a 32-bit floating point scale, with 64 elements
sharing the same scale. The steps are defined as follows:

1. Traverse the array into steps of 64, loading 64 single precision floats
   into 8 __m256 variables.

2. Calculate the scale for the block by finding the absolute max of the
   64 elements and calculating 7 times its reciprocal. Thus the 4-bit
   quantized value of the absolute maximum element of each block will
   be either 7 or $-7$.

3. For stochastic rounding purposes, generate random numbers using
   XORshift [65] pseudo random number generator (PRNG). The al-
gorithm is designed to generate __m256i values of 32-bit unsigned
   packed integers. However, every 8-bit chunk of the 32-bit unsigned
   packed integers also exhibits normal distribution when XORshift is
   invoked. Therefore, from one invocation of the PRNG, we can obtain
   32 numbers that fit in the $[0, 1)$ range. To obtain 64 random numbers,
   we execute this sub-routine twice, as illustrated in Figure 3.3.

4. Perform the scaling and the stochastic rounding by using fused-
multiply-add instructions to scale the initial 8 __m256 floating point
   values with the reciprocal scale, and add the random numbers.

5. Convert the obtained 8 packed floats into 8 packed unsigned 32-bit
   integers by truncation.

6. Sign the integers using vpsigned with the initial 8 AVX values such that
   each chunk is within $[-7, 7]$ range.

7. Perform in-register transposition of the $8 \times 8$ elements.

8. Pack the 4-bit quantized nibbles as described above to obtain a single
   AVX register of all 64 quanized values.

9. Store the scale and the quantized values and proceed to the next
    block.
XORShift (Pseudo Random Number Generator)

Every 8-bit chunk also has uniform distribution

Copy it 4 times, and shift left the 1\textsuperscript{st}, 2\textsuperscript{nd}, 3\textsuperscript{rd} and 4\textsuperscript{th} copy for 0, 1, 2 and 3 places respectively, obtaining random numbers in the range $[0, 2^{32} - 1]$

Convert to float and multiply by $\frac{1}{2^{32} - 1}$, obtaining random numbers in the range $[0, 1)$

Execute the operation twice, and obtain 8 registers

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{xorshift_diagram.png}
\caption{Generate 64 random numbers in $[0, 1)$ range with only two invocation of the XORShift algorithm}
\end{figure}
DOT PRODUCT. The dot product performs the operation $\gamma := a^T b$, where $a$ and $b$ are column vectors. We implement this operation by unpacking nibbles into 8-bit integers first. Then, we use the `vpmaddubsw` instruction to multiply each unsigned 8-bit integer from the first operand pairwise with the corresponding signed 8-bit integer from the second operand, producing intermediate signed 16-bit integers, and storing the sum of adjacent intermediate 16-bit integers in the destination. Finally, we sum the intermediate 16-bit integers, storing the results as 32-bit integers, eventually resulting in a partial sum that is represented as a 32-bit integer, which is then converted to a 32-bit float. The procedure is described as follows:

1. Create a single `__m256` of 8 32-bit floats that accumulates the result and initialize it to zero. Then start traversing the vectors by blocks of 64 elements.

2. For each block, load 64 values of $a^T$ into one `__m256i` called $A$ and 64 values of $b$ into a `__m256i` called $B$.

3. Unpack $A$ into two `__m256i` variables $A_1$ & $A_2$, and $B$ into $B_1$ & $B_2$, each storing 32 8-bit integers.

4. Extract the signs from the elements in $A_1$ and $A_2$ and use `vpsignb` to transfer them to $B_1$ and $B_2$. Now, $A_1$ and $A_2$ store unsigned 8-bit integers, and $B_1$ and $B_2$ store signed 8-bit integers.

5. Use `vpmaddubsw` to multiply $A_1$ and $B_1$ and to multiply $A_2$ and $B_2$, giving us two `__m256i` variables, each storing 8 signed 16-bit integers.

6. Over several instructions, successively sum the results until we have a single `__m256i` of 32-bit integers.

7. Convert the 32-bit integers into floats and multiply them with the scales, accumulating it in the result variable.

8. Once done iterating over all blocks, perform horizontal sum reduction on the result variable and return.

This dot product routine returns a single 32-bit floating point value, which may be later re-quantized. This re-quantization has some cost, but it is amortized by the rest of the dot product.
Array block $v$ with scale $s_v$

High 4-bits of $v$ (shifted by 4)

Low 4-bits of $v$ (shifted by 4)

$u = \text{sign}(u, v)$

$v = \text{sign}(v, v)$

psignb + vpmaddubsw

$u = \text{sign}(u, v)$

$v = \text{sign}(v, v)$

psignb + vpmaddubsw

$u = \text{sign}(u, v)$

$v = \text{sign}(v, v)$

psignb + vpmaddubsw

vpsraw + vpaddw

_vmm256_set1_epi16(1)

Convert to float

FMA: multiply by $s_v \cdot s_u$ and accumulate

Figure 3.4: Illustration of a dot product operation
**Mixed-Precision Dot Product.** In the 4-bit dot product, the values in each vector are unpacked to 8-bit before they are used for computation. We use this property to implement a mixed-precision dot product routine where one operand is an 8-bit quantized vector and the other is a 4-bit quantized vector. This is accomplished by unpacking the 4-bit operand into 8-bit integers and loading the 8-bit operand without unpacking. As explained next, the mixed-precision routine offers an advantage in MVM as explained next.

**Matrix-Vector Multiplication (MVM).** We implement MVM by traversing the matrix row by row, computing the dot product of each row with the corresponding vector. Once 64-rows are completed, we re-quantize the 64 values into a single 256-bit AVX variable, storing it in the resulting vector. This avoids the performance penalty from re-quantization associated with the scale and add routine.

We also implemented a mixed-precision MVM by casting its computation in terms of the mixed-precision dot product routine. In this mixed-precision MVM, the rows of the matrix are kept at 4-bit since they constitute the bulk of data movement. The vector is in 8-bit, since it is only loaded once (and reused) and thus there is little extra data movement but a gain in precision. The overall effect is a mixed-precision 4,8-bit MVM that is nearly as fast as the pure 4-bit MVM.

**Scale-and-Add.** The scale-and-add routine multiplies an instance of CloverVector4 by a scalar, and adds the result to another CloverVector4. For space reasons, we do not describe it in detail. The scale-and-add routine is not as efficient as the dot product routine. This is because for dot product, there is only a single value to re-quantize, and for scale and add, the re-quantization must be performed for every element of the output array.

**Parallelization.** We use OpenMP for multithreaded execution. The scale-and-add routine is embarrassingly parallel: one loop with independent iterations. For MVM, the outer loop that defines the blocked traversal along the rows of the matrix is parallelized; thus we do not need a parallel dot product.

### 3.3.2 8, 16 and 32-bit Arithmetic

We now discuss our implementation of the other datatypes.
8-bit arithmetic The implementation of the 8-bit version follows the same design decisions of the 4-bit version, i.e. blocks of 64 elements with a corresponding scale in the vector case, and tile of $64 \times 64$ elements with a corresponding scale in the matrix case. While the implementation is almost identical, one important difference is that the 8-bit dot product does not use steps 2) and 3), as the quantized values are already represented as 8-bit integers, and instead loads 64 values into two __m256i variables, each containing 32 8-bit integers.

16-bit arithmetic The implementation of the 16-bit version uses the native support for half-precision floats, defined in IEEE 754-2008 as the 16-bit base-2 format. Both the vector container and the matrix container have no notion of blocks; instead they are continuous arrays of type uint16_t (C intrinsics functions use uint16_t instead of a dedicated half-precision type). We use the vcvtps2ph instruction to perform quantiation. When performing scale and add, dot product, and MVM, we use vcvtph2ps to convert the packed 16-bit half precision floats into 32-bit single precision floats and perform the computation. Depending on the operation, we usually traverse the arrays in blocks of 16, 32, or 64 elements, such that we load, convert to 32-bit and compute while keeping most computation in registers and maintaining spatial and temporal locality. To benefit from ILP, we unroll the loops and use several accumulators to maximize performance. Our MVM implementation is based on our 16-bit dot product routine, and we use OpenMP to parallelize the outer loops.

32-bit arithmetic The 32-bit version uses single-precision floating point arrays. We use floating-point for this precision because computation with 32-bit floats is much faster than with 32-bit integers in AVX. Our implementation of the dot product as well as the scale and add routine is almost identical with the 16-bit version, except that we do not perform conversions. For MVM we use the (parallel) Intel Math Kernel Library (MKL) and use OpenMP to parallelize the other routines.

3.3.3 Low-Level Abstraction in LMS

We build a virtual variable-precision ISA that implements the quantization operation and dot product operator, operating on arrays of 32, 16, 8 and 4-bit precision. Similarly to the native implementation in Clover, we use floating point representation for 32 and 16-bit, and for the lower precision formats,
we use quantized arrays. In this particular version, we skip parallelization, and provide a single threaded version.

We consider a functionality similar to the SVML short vector math library, provided by Intel C/C++ compilers. Namely, we design functions that operate with continuous blocks of memory, are inlined directly (avoiding function calls) and are “CPU dispatched”, i.e., the runtime decides which hardware instructions to use. For example, if the runtime detects absence of the F16C ISA, it will automatically switch to a 16-bit version with quantized arrays.

We abstract the precision as a number that reflects the bit length of the primitive type, and provide a virtual intrinsic function that performs quantization:

```scala
1 def quantize_block (bits : Int,
2 src : Rep[Array[Float]],
3 dst : Rep[Array[Int]],
4 scales : Rep[Array[Float]],
5 offset : Rep[Int]
6 ): Rep[Unit]
```

The staged function `quantize_block` will load from the source array, perform quantization, and store the result into the destination array. When scales become available, they will be stored in the scales array. The function uses the `bits` parameter and selects the corresponding bit format for quantization, dispatching an implementation for that format.

```scala
1 def quantized_block_size (bits: Int): Rep[Int]
```

Taking advantage of data parallelism, several values of the array are processed at once. We design the function `quantized_block_size` that returns the number of elements processed for a given bit length. For example, in the case of 32, 16 and 8-bit versions, 32 elements are processed at a time and in the case of 4-bit, 128 elements at a time. This allows us to define a single quantization routine for all formats:

```scala
1 val increment = quantized_block_size(bits);
2 for (0, n, fresh[Int], increment, i => {
3 quantize_block(bits, src, dst, scales, i)
4 })
```
3.3 Low Precision Arithmetic

All other individual routines are done similarly. We illustrate the design of the dot product, implemented through the `dot_block` intrinsic:

```plaintext
```

dot_block takes the bit length and two memory addresses. According to the bit length, it will select the corresponding bit format, load from the data array and the scales array, compute the dot product and return the result. Assuming that the length of the arrays are padded with their corresponding quantized_block_size value, the two intrinsics allow us to easily build for loops with an increment defined by the quantized_block_size, such that dot_block is invoked at each iteration. Finally, the resulting dot product with variable precision is a sum reduction of 8 floats stored in the acc variable:

```plaintext
var acc = _mm256_setzero_ps();
val increment = dot_ps_step(bits);
for (0, len, fresh[Int], increment, i => {
  acc += dot_ps(dot_block, x, scales_x, y, scales_y, i)
})
reduce_sum(acc)
```

It is possible to abstract the individual routines to an ISA-agnostic implementation as later discussed in Chapter 4, and performance could be improved using methods as prefetching, particularly in the 8-bit [56] and 4-bit version, however we did not apply these techniques.

JAVA IMPLEMENTATION. To reason about the speed and performance of the LMS abstractions, we also provide a pure Java version that runs on the JVM. We implement each case as a separate class corresponding to the 32, 16, 8 and 4 bit versions. The simplest dot product is the 32-bit version. Java does not support bit manipulation or numerical expression on types lower than 32-bits. Instead, 16-bit and 8-bit types are promoted to 32-bit integers before operations are performed. To provide a fair comparison, we write the 16, 8 and 4-bit versions of the dot product such that we block the loop, and accumulate into integer values, avoiding unnecessary type promotion.
as much as possible. An illustration of a 4-bit dot product in Java is given below:

```java
public float dot (JVector4 other) {
    final int len = size_pad;
    final byte [] u = this.values, v = other.values;
    final float [] su = this.scales, sv = other.scales;

    float sum = 0.0f;
    for (int idx = 0; idx < len; idx += 64) {
        int acc = 0;
        for (int j = 0; j < 64; j += 2) {
            final int i = (idx + j) >> 1;
            final int qu = (int) u[i];
            final int qv = (int) v[i];

            final int v1 = (qv >> 4);
            final int u1 = (qu >> 4);
            final int v2 = (qv << 28) >> 28;
            final int u2 = (qu << 28) >> 28;

            acc += v1 * u1 + v2 * u2;
        }
        float scale = su [idx / 64] * sv [idx / 64];
        sum += scale * (float) acc;
    }
    return sum;
}
```

In the Java version we also skip parallelization, and provide a single threaded implementation only.

### 3.4 Evaluation

We evaluated our implementation on an Intel Xeon CPU E3-1285L v3 3.10 GHz Haswell with 32 GB of RAM and 25.6 GB/s bandwidth to main memory, running Debian GNU/Linux 8 (jessie), kernel 3.16.43-2+deb8u3. We use the Intel icc compiler 17.0.0, Intel IPP 2017.0.0 (r52494), and Intel MKL 2017.0.0 (Build 20160801). We use RDTSC to measure the cycle count for each test, performing 15 repetitions with warm cache, and reporting the median result. To avoid the effects of frequency scaling and resource sharing on the measurements, Turbo Boost and Hyper-Threading are disabled.
Figure 3.5: Performance of single threaded dot product (native implementation).

The installed JVM is HotSpot 64-Bit Server 25.144-b01, supporting Java 1.8. We use ScalaMeter [38] to perform the benchmarks. To obtain precise results, we select a pre-configured benchmark that forks a new JVM virtual machine and performs measurements inside the clean instance.

3.4.1 Performance of Individual Routines

First, we evaluate the performance of the individual routines. For each routine we derive a pseudo flop count, using the number of additions and multiplications required to perform the mathematical operations, and report our results in flops per cycle (f/c). The flop count for dot product and scale-and-add is $2n$ and for MVM it is $2n^2$.

**Dot Product.** Fig. 3.5 shows the performance profile of the dot product. When the data fits in the L2 cache the 32-bit version is much faster than 4-bit because the entire computation is done using native instructions without unpacking. Once the data exceeds the size of the L3 cache, 4-bit is fastest since the dot product is memory bound: data movement from RAM becomes the bottleneck. The speed-up is up to 6x over the 32-bit version. Since the dot product is only used within (a parallel) MVM, we do not need a parallel version.

**Scale-and-Add.** The results in Fig. 3.6a show that the 32-bit and 16-bit implementations are faster than 4-bit and 8-bit within cache for the same reasons. However, even outside L3 cache, 4- and 8-bit are no faster than
32-bit due to the overhead of re-quantization. This is reflected in the low bandwidth used. As a result, parallelization yields near-linear speed-up for 4-bit and near none for 32-bit, so parallel 4-bit is about 3x faster overall (Fig. 3.6b).

**MVM.** In Figs. 3.6c and 3.6d, we compare sequential and parallel implementations of MVM for each datatype, including the mixed 4,8-bit MVM explained in Section 3.3.2. For the sequential case, for problems that do not fit in cache, we see that pure 4-bit is about 4.6x faster than 32-bit but uses only one third of the available bandwidth, and the mixed 4-bit and 8-bit MVM is noticeable slower. However, once parallelized, all version exhaust the available bandwidth and thus reach a speedup linear in the precision reduction. The mixed 4,8-bit MVM is now as fast as the 4-bit version since the bottleneck is loading the matrix.

### 3.4.2 Evaluation of the Low-Level Abstractions in LMS

We use the same experimental setup as in section 2.4. For each benchmark, we use the identical flop (or op for 8 and 4 bit) count of $2n$ for the dot-product, where $n$ is the size of the quantized array.

Figure 3.7 shows the obtained results. Our 4-bit implementation outperforms HotSpot by a factor of up to 40x, the 8-bit up to 9x, the 16-bit up to 4.8x, and the 32-bit version up to 5.4x. There are several reasons for the speedups obtained with the use of SIMD intrinsics. In the 32-bit case, we see the limitation of SLP to detect and optimize reductions. In the 16-bit, there is no way to obtain access in Java to an ISA such as FP16C. And in the 8-bit and 4-bit case, Java is severely outperformed since it does type promotion when dealing with integers. However, the largest speedup of 40x in the 4-bit case is due to the domain knowledge used for the implementing the dot product, that the HotSpot compiler cannot synthesize with a lightweight autovectorization such as SLP.

Comparing the results in Figure 3.5 and Figure 3.7, we can observe difference in the performance of the two implementations. This is no surprise, as the constant overhead imposed by the JNI call to execute the dot product, will be added to the measurements, diminishing the overall performance. But once data is in L2, L3, or RAM the overhead gets amortized with the rest of the computations, having almost identical results with the native implementation. Note that the JVM version will always underperform compared to the native implementation, as it does not employ any prefetching.
Figure 3.6: Performance evaluation of quantized linear algebra routines: MVM and AXPY
3.4.3 Evaluation of Quantized IHT and Quantized GD

We implemented the IHT and GD algorithms with the routines described in Section 3.3 and our basic library functions. We show relative performance, comparing the 8-bit, 16-bit, and mixed 4,8-bit versions to the 32-bit implementation of these algorithms. The mixed 4,8-bit version uses 4-bit matrices and 8-bit vectors, using a mixed 4-bit and 8-bit MVM routine and pure 8-bit vector-vector routines.

**Experimental Setup.** In the case of IHT, we set up $\Phi$ to have twice as many columns as rows ($n = 2m$), with a sparsity of $m/4$. For GD, we set up $A$ to have $1.5x$ as many columns as rows ($n = 1.5m$). We define the recovery error as the norm of the difference between the recovered $x$ and the original $x$, divided by the norm of the original $x$. For both algorithms IHT and GD, we use grid search for the hyperparameter optimization of the $\mu$ parameter to decrease recovery error and minimize the number of iterations. We compute $\Phi^T$ needed in (3.5) offline and feed it as an argument along with $\Phi$ to both the IHT and GD routines. We use the parallel version of all operations within the IHT and GD algorithms.
**Figure 3.8:** Relative speed up of quantized GD and quantized IHT compared to 32-bit IHT IEEE-745 running on various matrix sizes

The first important observation concerns the absolute gain in runtime for the selected target error. For GD, 4-bit is about 5x faster as 32-bit. Since the per-iteration speed-up is about 8x, this means it needs about 3/5 more iterations, typically around 24 versus 15 for 32-bit. Interestingly, 8-bit can offer more than 4x speedup in some cases, meaning that it actually requires fewer iterations than 32-bit. This can happen as stochastic quantization...
adds noise, which can speed up convergence in some cases, e.g. [52, 66]. Also, the runtime is very sensitive to the value of $\mu$.

For IHT, the runtime behaviour is more erratic, likely due to the dependency on $\mu$ and due to the small number of iterations (typically around 3–5). For 4-bit we also observe that usually more iterations are needed but also, in rare cases, fewer, achieving speedups of up to 7.6x. Overall, 4-bit offers an absolute speedup in many cases. The delicate relationship between problem specification, data precision, hyperparameter, and target error invites further investigation.

3.5 RELATED WORK

There has recently been considerable interest in low-precision computation and communication in the context of machine learning, e.g. [66–68]. Due to space constraints, we will focus on work that is immediately related to ours. One of the first to consider low-precision computation with convergence guarantees in the context of (stochastic) gradient descent has been Buckwild! [51]. They provide convergence analysis for the restricted case where only the gradients are quantized, and provide CPU implementation results via intrinsics for four and eight bits in the more general case where the gradients are quantized. We note that their implementation departs from the theory, and therefore is not analytically guaranteed to converge.

The ZipML project [52] presents an algorithmic framework for end-to-end low-precision stochastic gradient descent (quantized gradients, model, and data), complete with convergence analysis in the convex case including FPGA implementation results for various precisions [69]. We implement their algorithmic framework here in the context of gradient descent. To our knowledge, this is the first efficient such implementation for CPUs. Our framework removes their technical requirement that the data has to be quantized before the execution of the algorithm, as we also implement fast quantization and dequantization. Recent work [56] performed an in-depth study of low-precision and asynchronous SGD computation on CPUs and FPGAs, with an eye towards architectural implications. Due to the lack of native instruction support for 4-bit (or lower) operations in, e.g., AVX2, they focused on 8-bit and higher precisions in their implementation, and obtained 4-bit results by emulating the existence of native instructions in a simulator. Here, we overcome this limitation by replacing native 4-bit instructions through careful use of existing AVX2 intrinsics. This allows us
to obtain results for 4-bit instructions on existing CPUs, while at the same time preserving computational efficiency.

Low-precision recovery algorithms for compressive sensing, and their FPGA implementations, have been considered by several references, e.g. [53–55]. Our CPU implementation of low-precision IHT is adapted from this last reference. To our knowledge, ours is the first low-precision CPU implementation of this technique.

3.6 Summary

The work presented in this chapter first shows that reduced precision stochastic quantized arithmetic can be implemented efficiently on modern CPUs. In particular, this holds for 4-bit on Intel AVX even though there is no native support for operations with this data format. This is possible since the extra operations needed are overcompensated by the reduced data movement in the memory-bound situations that are typical in many applications. In these cases a speed-up linear in the precision reduction is possible. As examples where this is the case, we considered gradient descent and iterative hard thresholding, for which we exhibited absolute speed-ups (time to target error) with reduced precision.

But most importantly, this work shows that we can use metaprogramming techniques and staging to develop low-level abstractions on top of existing instructions. Our benchmarks demonstrate that this combination of SIMD and metaprogramming enables developers to write high-performant, explicitly vectorized code on an unmodified JVM that outperforms the auto-vectorizing HotSpot just-in-time (JIT) compiler and provides tight integration between vectorized native code and the managed JVM ecosystem.
Abstracting Data Parallelism using Staged Polymorphism

The work presented in the previous chapters demonstrates that with the use of low-level abstraction, staging and meta-programming, we can develop highly tuned ISA-specific code. However, fast computations require deployment in many different systems, ranging from embedded systems and mobile devices to data-centers. Therefore, relying on low-level abstraction to develop code for multiple ISAs, could potentially result in increasing the development effort, or could hamper the portability of the given computation. To work around these issues, the existing low-level abstractions can be abstracted even further, providing high-level and ISA-agnostic abstractions.

High-level abstractions are crucial for mathematical domains in two aspects: when mapping mathematical domains to a given microarchitecture and when domain specific optimizations are required. The approach of stage polymorphism \cite{70}, or more precisely - *parametric stage polymorphism*, shows us that we can build DSLs with high-level abstractions such that staging decisions are encoded into the type system. Taking into consideration that vectorized computation are also represented with vector types, this chapter explores the question:

- Can we extend the approach of parametric stage polymorphism to provide modular and extensible high-level abstraction for modern SIMD vector architectures in a DSL-based generator?

To answer this question, we build a prototypical code generator for finite input response (FIR) filters with an internal DSL based on LMS. The chapter is organized as follows. Section 4.1 provides the motivation of using high-level abstraction as building blocks in developing mathematical DSLs. Section 4.2 gives an overview of the design of the generator, explains the tiling of convolutional expressions, and high-level loop optimizations. Section 4.3 provides overview of parametric staged polymorphism and
explains the extensions for data parallelism support and ISA-abstractions. Section 4.4 demonstrates how our approach is competitive with commercial libraries, and 4.6 provides the summary.

The work presented in this chapter was published in the 2014 International Workshop on Libraries, Languages, and Compilers for Array Programming [46].

4.1 Motivation

Numerical libraries need to be highly tuned to the platform’s architecture and microarchitecture to reach highest performance. This tuning requires expensive programming effort, and conflicts with portability, since it often has to be repeated for every new processor generation. Over the last decade, one solution that has emerged to solve this problem are program generators that use domain-specific languages (DSLs) to express mathematical algorithms at a high level of abstraction, which is then compiled into platform-specific high performance code [7, 71–78]. In some of these generators, DSLs are also used internally to perform difficult optimizations such as loop fusion or vectorization at a high level of abstraction through rewriting to overcome compiler limitations. Examples of this idea include Spiral [5] (a generator for linear transforms) which uses a DSL called Σ-SPL [79], and LGen [78] which uses an extension called Σ-LL.

These languages can be viewed as a domain-specific extension of the array programming paradigm, augmented with explicit data access objects and higher level mathematical operators. Intuitively, this representation makes it possible to restructure the computation to achieve the above optimizations.

An orthogonal question, investigated for example in [11, 80] is how to efficiently build such generators in an effective and maintainable way using modern programming language features. In particular, the concept of staging [9] has been proposed to build generators within a host language [76, 81]. Modern staging frameworks such as LMS (Lightweight Modular Staging) [22] go beyond primitives for emitting code and have become popular for implementing generators based on one or multiple levels of DSLs [11, 21, 82, 83]. However, only few existing generators target SIMD vector architectures, i.e., emit code that uses the so-called intrinsics interface to directly and efficiently use vector instructions.

Contributions The main contribution presented in this chapter is a new library generator called FGen for a very narrow but important operation: convolution, or, as it is called in media processing, finite-impulse-
response (FIR) filters. The generator takes as input a mathematical convolution expression including the size of two arrays involved and outputs an optimized library function. Internally a variant of the above-mentioned DSL $\Sigma$-LL is used to structure the computation and to facilitate the mapping to a vector architecture.

The second main contribution is a generic support layer for targeting vector architectures from DSL-based program generators. Specifically, we combine staging and generic programming using type classes to abstract at the DSL level over both the data representation (e.g. real or complex numbers) and the vector architecture (e.g., SSE or AVX). Extensions to new data types and new vector architectures thus become completely modular in the backend translation engine. This SIMD support layer is not specific to convolution but designed to be applicable to a large set of possible generators built with suitable DSLs.

Finally we show benchmarks comparing our automatically generated convolution code with commercial high performance libraries to demonstrate the viability of our approach.

4.2 FGen

We present FGen, a program generator for performance-optimized functions implementing convolutions, or FIR filters. FGen follows [11] in design and implementation but extends the work to include filters, and to support vector ISAs in a modular way. The mathematic of our approach build on the FIR generation as done by Spiral [84]. Figure 4.1 gives a high-level overview of the generator. The input to FGen specifies the desired function to be generated. It consists of

1. the desired convolution, written mathematically as $y = h \ast x$ with specified sizes for the vectors $x, h, y$;

2. the data type and memory layout (e.g., real, interleaved or split complex); and

3. the vector ISA (e.g., Intel SSE or Intel AVX).

We give a short overview on the inner workings next and follow up with more details in Section 4.2.1. In the first step, FGen formally tiles the computation for better locality using the mathematical language $\Sigma$-LL that slightly extends the aforementioned $\Sigma$-SPL [79] and $\Sigma$-LL [78]. In essence, the language consists of vectors, matrices, and data access objects as operands, as
well as linear algebra operations including addition, multiplication, and different forms of convolution. The chosen tile size is a search parameter but is chosen to be a multiple of the desired vector length of the specified ISA. Σ-LL makes loops and access patterns explicit, thus enabling loop optimizations like merging in the next step.

After the proper mathematical structure is established, the data type and memory layout parameters are introduced. The code style (e.g., to what degree unrolled) is a search parameter that is also fixed at this time. These choices are implemented using abstractions as explained in [11]. Next, the desired vector ISA is introduced and code is generated in an intermediate representation (IR) for vector code. The choice of vector ISA and the data type and layout are mutually dependent. For instance, when dealing with interleaved complex arrays, we must specialize to suitable shuffles that interleave and de-interleave complex numbers. Handling the interaction
of data layout and vector ISA in a modular fashion is very complex and a main contribution of this work (see Section 4.3).

Finally, the code is unparsed to C including vector intrinsics for SSE or AVX. The performance is measured and informs a feedback-driven search to find the best choices for all the parameters and choices mentioned before. The best code found is returned as final result.

4.2.1 Tiling

The convolution $y = x \ast h$ of two vectors $x = (x_0, x_1, \ldots, x_{n-1})^T$ and $h = (h_0, h_1, \ldots, h_{k-1})^T$ is defined as

$$y_m = \sum_{i=0}^{k-1} x_{m+(k-1)-i} \cdot h_i, \quad m = 0, \ldots, n - 1. \quad (4.1)$$

We use the language of signal processing and call $h$ the filter, $k$ the number of taps, and the computation finite-impulse-response filter (FIR). The filter can also be viewed as a matrix vector product:

$$y = x \ast h \iff y = M_n(h) \cdot x, \quad (4.2)$$

where $M_n(h)$ is the $n \times n$ matrix

$$
\begin{bmatrix}
  h_{k-1} & h_{k-2} & \cdots & h_0 \\
  \vdots & \ddots & \ddots & \ddots \\
  h_{k-1} & h_{k-2} & \cdots & h_0 \\
  h_{k-1} & \cdots & h_1 & \ddots \\
  & \ddots & \ddots & \ddots \\
  & & & h_{k-1}
\end{bmatrix}. \quad (4.3)
$$

To improve locality and thus performance, FGen tiles the initial filter into a sum of smaller FIR filters. To express this mathematically, we use the notion of \\textit{gathers} and \\textit{scatters}, which are functions, parameterized by a \\textit{data access function} [79]. To describe the basic idea of the gathers and scatters, we use MATLAB-like notation. We assume two vectors $x$ and $y$, having sizes $n$ and $N$ respectively, and parameters $a$ and $b$. Figure 4.2 shows the MATLAB equivalents of the used gathers and scatters.

---

1 We note that there are different variants of convolution depending on the handling of the boundaries.
<table>
<thead>
<tr>
<th>MATLAB notation</th>
<th>Σ-LL notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y = x(a:b) )</td>
<td>( y = G(h_{a}^{b-a+1 \rightarrow n}) \cdot x )</td>
</tr>
<tr>
<td>( y(a:b) = x )</td>
<td>( y = S(h_{a}^{b-a+1 \rightarrow N}) \cdot x )</td>
</tr>
</tbody>
</table>

**Figure 4.2:** Gathers and Scatters using MATLAB notation

**Gathers** Gathers represent matrices, such that, when multiplied with a vector, will extract (gather) parts of the initial vector. They operate with data access function \( f^{n \rightarrow N} \) defined on a domain \( \{0, 1, \ldots, n - 1\} \) and range \( \{0, 1, \ldots, N - 1\} \), that is used to rearrange the indices of the sub-vector:

\[
G(f^{n \rightarrow N}) = \begin{bmatrix}
  e_{f(0)}^{T} \\
  e_{f(1)}^{T} \\
  \vdots \\
  e_{f(n-1)}^{T}
\end{bmatrix}.
\] (4.4)

**Scatters** Scatter is defined as the left inverse matrix of a gather matrix.

\[
S(f^{n \rightarrow N}) = (G(f^{n \rightarrow N}))^{T}.
\] (4.5)

**Stride permutation** represents a specific data access function used in both scatters and gathers, that performs a linear index mapping of the vector indices.

\[
h_{b,s}^{n \rightarrow N} : i \mapsto b + is.
\] (4.6)

**FIR decomposition** Using this notation we can now decompose the FIR filter \( y = x * h \). First, we observe that in (4.3), the first \( n - k + 1 \) rows form a “parallelogram” shape matrix, and the last \( k - 1 \) rows form an upper triangular matrix. Correspondingly, we can decompose the filter into a concatenation (⊕) of two specialized filters that we call parallelogram shape filter (PFIR) and upper triangular filter (UTFIR):

\[
x * h = x \parallel h + x' \bigtriangleup h',
\] (4.7)

such that

\[
x' = x \cdot G(h_{n-k+1,1}^{k-1 \rightarrow n}) \quad \text{and} \quad h' = h \cdot G(h_{1,1}^{k-1 \rightarrow k}).
\]

In the following these two filters are decomposed separately. Each decomposition has degrees of freedom which are searched over by FGen.
The PFIR filter is represented by a sparse matrix with a parallelogram shape. Tiling this matrix to rectangular shapes would also result in sparse and dense sub-matrices, depending on the boundary conditions. However, we can decompose the PFIR matrix into tiles of parallelogram shapes, as illustrated in Figure 4.3.

This essentially means that a PFIR filter is decomposed into one or several smaller PFIR filters. To achieve that, we need to consider vertical and horizontal decompositions.
abstracting data parallelism using staged polymorphism

We now combine \((n)\). Assume that \(k = p + q\). If the PFIR matrix is sliced diagonally at position \(p\), the initial PFIR matrix can be broken down into a sum of two PFIR matrices (Fig. 4.4a). Thus:

\[
x || h = x \cdot G(h_{0,1}^{n-p\rightarrow k}) || h \cdot G(h_{q,1}^{p\rightarrow k}) \\
+ x \cdot G(h_{p,1}^{n-p\rightarrow q}) || h \cdot G(h_{0,1}^{q\rightarrow k}).
\]

Now assume that \(k = b \cdot t\). The PFIR matrix can be broken down into \(t\) smaller matrices (Fig. 4.4b):

\[
x || h = \sum_{i=0}^{t-1} x \cdot G(h_{b+i,1}^{n-k+b\rightarrow n}) || h \cdot G(h_{b(i+1),1}^{k-b(i+1),1}).
\]

The general case of \(k = b \cdot t + s\) can be derived by combining (4.8) and (4.9):

\[
x || h = \sum_{i=0}^{t-1} x \cdot G(h_{b+i,1}^{n-k+b\rightarrow n}) || h \cdot G(h_{b(i+1),1}^{k-b(i+1),1}) \\
+ x \cdot G(h_{bt,1}^{n-bt\rightarrow n}) || h \cdot G(h_{0,1}^{s\rightarrow k}).
\]

PFIR vertical decomposition

Similarly to the analysis of the vertical breakdowns, we assume that \(n - k + 1 = p + q\). Observing Fig. 4.5a, we derive:

\[
x || h = S(h_{0,1}^{p\rightarrow n-k+1})(x \cdot G(h_{0,1}^{p+k-1\rightarrow n}) || h) \\
+ S(h_{p,1}^{q\rightarrow n-k+1})(x \cdot G(h_{p,1}^{q+k-1\rightarrow n}) || h).
\]

Assume that \(n - k + 1 = b \cdot t\). We can derive \(t\) smaller PFIR matrices (illustrated in Fig. 4.5b):

\[
x || h = \sum_{i=0}^{t-1} S(h_{b+i,1}^{n-k+1})(x \cdot G(h_{b+i,1}^{n-k+b\rightarrow n}) || h).
\]

We now combine (4.11) and (4.12) and infer the general case \(n - k + 1 = b \cdot t + s\):

\[
x || h = \sum_{i=0}^{t-1} S(h_{b+i,1}^{n-k+1})(x \cdot G(h_{b+i,1}^{n-k+b\rightarrow n}) || h) \\
+ S(h_{bt,1}^{s\rightarrow n-k+1})(x \cdot G(h_{bt,1}^{s+k-1\rightarrow n}) || h).
\]
To construct the PFIR tile, we need to perform blocking in both horizontal and vertical direction. Let's assume that $n - k + 1 = b^H \cdot t^H + s^H$ and $k = b^V \cdot t^V + s^V$. By merging equations (4.10) and (4.13) we obtain:

$$\begin{align*}
x \parallel h &= \sum_{i=0}^{t^H-1} S(h_{b^H,i,1}^{b^H \rightarrow n-k+1}) \sum_{j=0}^{t^V-1} x_1 \parallel h_1 \\
&+ \sum_{i=0}^{t^H-1} S(h_{b^H,i,1}^{b^H \rightarrow n-k+1})(x_2 \parallel h_2) \\
&+ S(h_{b^H t^H,1}^{s^H \rightarrow n-k+1}) \sum_{j=0}^{t^V-1} x_3 \parallel h_1 \\
&+ S(h_{b^H t^H,1}^{s^H \rightarrow n-k+1})(x_4 \parallel h_2),
\end{align*}$$

(4.14)

where

$$\begin{align*}
x_1 &= x \cdot G(h_{b^H,i+b^V,j,1}^{b^H+b^V-1 \rightarrow n}), \\
x_2 &= x \cdot G(h_{b^H,i+b^V,j,1}^{b^H+b^V-1 \rightarrow n}), \\
x_3 &= x \cdot G(h_{b^H t^H+b^V,j,1}^{s^H+b^V-1 \rightarrow n}), \\
x_4 &= x \cdot G(h_{b^H t^H+b^V,j,1}^{s^H+b^V-1 \rightarrow n}), \\
h_1 &= h \cdot G(h_{k,b^V(i+1),j,1}^{b^V \rightarrow k}), \\
h_2 &= h \cdot G(h_{0,1}^{s^V \rightarrow k}).
\end{align*}$$

UTFIR BREAKDOWN The remaining upper triangular matrix has a size of $k - 1$ rows and columns. Figure 4.6 suggests that UTFIR can be broken
down into two types of tiles, the parallelogram shape tiles, introduced in the PFIR matrix, and upper triangular tiles. Let’ assume that $k - 1 = b \cdot t + s$. Each $P_i$ matrix can be represented as

$$P_i = M_b^\parallel \left[ h \cdot G(h_{1,1}^{k-1\rightarrow k}) \cdot G(h_{b(i+1),1}^{k-1-b\cdot(i+1)\rightarrow k-1}) \right],$$ (4.15)

and each $Q_i$ matrix can be represented as

$$Q_i = M_b^\triangle \left[ h \cdot G(h_{1,1}^{k-1\rightarrow k}) \cdot G(h_{bi,1}^{b\rightarrow k-1}) \right].$$ (4.16)

Similar analysis as used before in the PFIR matrix for horizontal and vertical breakdown is performed on the UTFIR matrix. We omit the details and just show the final breakdown equation:

$$x \triangle h = \sum_{i=0}^{t-1} S(h_{b,i,1}^{b\rightarrow k}) \sum_{j=0}^{t-i-2} x_1 \parallel h_1 + \sum_{i=0}^{t-1} S(h_{b,i,1}^{b\rightarrow k}) (x_2 \parallel h_2) + \sum_{i=0}^{t-1} S(h_{b,i,1}^{b\rightarrow k}) (x_3 \triangle h_3) + S(h_{b,i,1}^{s\rightarrow k}) (x_4 \triangle h_4),$$ (4.17)

where

$$x_1 = x \cdot G(h_{bi+by,1}^{2b-1\rightarrow k}), \quad h_1 = h \cdot G(h_{k-b(j+1),1}^{b\rightarrow k}),$$

$$x_2 = x \cdot G(h_{b(t-1),1}^{s+b-1\rightarrow k}), \quad h_2 = h \cdot G(h_{b(i+1),1}^{s\rightarrow k}),$$

$$x_3 = x \cdot G(h_{k-b,1}^{b\rightarrow k}), \quad h_3 = h \cdot G(h_{b(i+1),1}^{b\rightarrow k}),$$

$$x_4 = x \cdot G(h_{k-s,1}^{s\rightarrow k}), \quad h_4 = h \cdot G(h_{k-s,1}^{s\rightarrow k}).$$

**LTFIR breakdown**  In the case of a streaming FIR filter, we need to also consider the case of a lower triangular matrix. The LTFIR matrix is
symmetric to the UTFIR matrix. Therefore the derivation of the breakdown rules, correspond to the one of the UTFIR:

\[
\begin{align*}
\hat{x} \triangledown \hat{h} &= S(h_{s \rightarrow k}^{s_{0,1}})(\hat{x}_1 \triangledown \hat{h}_1) \\
&+ \sum_{i=0}^{t-1} S(h_{s+b \cdot i,1}^{b_{s+b \cdot i,1}})(\hat{x}_2 \triangledown \hat{h}_2) \\
&+ \sum_{i=0}^{t-1} S(h_{s+b \cdot i,1}^{b_{s+b \cdot i,1}}) \sum_{j=0}^{i-1} \hat{x}_3 \parallel \hat{h}_3 \\
&+ \sum_{i=0}^{t-1} S(h_{s+b \cdot i,1}^{b_{s+b \cdot i,1}})(\hat{x}_4 \parallel \hat{h}_4),
\end{align*}
\] (4.18)

where

\[
\begin{align*}
\hat{x}_1 &= \hat{x} \cdot G(h_{s \rightarrow k}^{s_{0,1}}), & \hat{h}_1 &= \hat{h} \cdot G(h_{s \rightarrow k}^{s_{0,1}}), \\
\hat{x}_2 &= \hat{x} \cdot G(h_{s+b \cdot i,1}^{b_{s+b \cdot i,1}}), & \hat{h}_2 &= \hat{h} \cdot G(h_{s+b \cdot i,1}^{b_{s+b \cdot i,1}}), \\
\hat{x}_3 &= \hat{x} \cdot G(h_{b \cdot j,1}^{b_{b \cdot j,1}}), & \hat{h}_3 &= \hat{h} \cdot G(h_{b \cdot j,1}^{b_{b \cdot j,1}}), \\
\hat{x}_4 &= \hat{x} \cdot G(h_{b \cdot i,1}^{b_{b \cdot i,1}}), & \hat{h}_4 &= \hat{h} \cdot G(h_{b \cdot i,1}^{b_{b \cdot i,1}}).
\end{align*}
\]

**ISA Tiling**  
Equations (4.14) and (4.18) include three important parameters for the tiling, namely \(b^H\), \(b^V\), and \(b\). If a vector ISA and hence an associated (ISA) vector length \(v\) is specified, we make sure that the inner-most tiling is a multiple of \(v\) for efficient mapping to intrinsics.

### 4.2.2 Mapping to \(\Sigma\)-LL and Loop Optimizations

Given an FIR filter expression, FGen will use the breakdown rules explained in Section 4.2.1 to decompose a given FIR filter into PFIR and UTFIR filters. This process can be done iteratively, decomposing tiles into even smaller tiles. This allows FGen to decompose the FIR filters to fit the cache hierarchy, and then decompose the tiles to correspond to the given ISA.

FGen uses \(\Sigma\)-LL to perform tiling of the FIR expression as well as to perform loop optimizations. The \(\Sigma\)-LL DSL uses the symbol

\[
\sum_{j=0}^{N-1} A_j
\] (4.19)

to encode loops: \(j\) is the loop index which can be utilized within the parameterization of \(\Sigma\)-LL constructs such as, e.g., gather and scatter matrices within the loop, denoted here as \(A_j\). The restriction on the sum is that each iteration of \(A_j\) produces a vector that contains results within a restricted chunk and outside this chunk. Consequently, the rules of FIR decomposition can be directly mapped into \(\Sigma\)-LL constructs.
LMS REPRESENTATION OF Σ-LL  LMS represents Σ-LL using sea-of-nodes representation that correspond to DSL constructs, such as gathers, scatters, index mapping functions, as well as PFIR and URFIR nodes. To represent data access functions (or index mapping functions), FGen uses

abstract class IM
abstract class DefIM (range: Int, domain : Int) extends Def[IM] {
  val m_domain: Int = domain
  val m_range : Int = range
}

// Representation of a stride permutation
case class IM_H (fragsize: Exp[Int], stride: Exp[Int], range: Int, domain: Int ) extends DefIM(range, domain)

// Syntactic sugar to create LMS definitions
def H(f:Rep[Int], s:Rep[Int], r:Int, d:Int): Rep[IM] = IM_H(f,s,r,d)

Composition of two index mapping functions can be represented with

case class IM_Compose (x: Exp[IM], y: Exp[IM], range: Int, domain : Int ) extends DefIM(range, domain)

And gathers / scatters with:

case class Gather (im: Exp[IM], x: Exp[Vector]) extends Def[Vector]
case class Scatter (im: Exp[IM], x: Exp[Vector]) extends Def[Vector]
def G (im: Rep[IM], x: Exp[Vector]): Exp[Vector] = Gather (im, x)
def S (im: Rep[IM], x: Exp[Vector]): Exp[Vector] = Scatter (im, x)

Finally, Σ-LL will be represented with

case class Sigma (start : Exp[Int], end : Exp[Int], i : Sym[Int], body : Block[Vector] ) extends Def[Vector]

Note that in this case body represents the expression inside the Σ-LL construct. Finally, the FIR filter constructs are given by

case class FIR (x: Exp[Vector], y: Exp[Vector]) extends Def[Vector]
Therefore, decomposition of the FIR filter is represented as

```scala
class Decomposition extends ForwardTransformer { self =>
  val IR: SigmaLL_DSL; import IR._
  override def transformStm(stm: Stm): Exp[Any] = stm match {
    case TP(s, FIR(x1, h1)) if (b != 0) => {
      val (x, h) = (apply(x1), apply(h1))
      val (n, k) = (x.size, h.size)
      if (n == k) (x ^^ h) else {
        val pfir = G(H(0, 1, n - 1, n), x) || h
        val bdPFIR = S(H(0, 1, n - k, n), pfir)
        val utfir = G(H(n - k, 1, k, n), x) ^^ h
        val bdUTFIR = S(H(n - k, 1, k, n), utfir)
        (bdPFIR ++ bdUTFIR).getRep
      }
    } else {
      case _ => super.traverseStm(stm)
    }
  }
}
```

The decomposition is done using the transformer interface of LMS, namely the ForwardTransformer. This transformation process is executed, such that the sea of nodes representing the Σ-LL expressions are topologically sorted by their dependencies, and then are traversed forward, replacing each symbol with the given transformation in the transformStm method. The PFIR decomposition into smaller PFIR filters is given in Figure 4.7.

**Loop Optimizations**  The decomposition process results in generated code with nested loops that can be further optimized using techniques such as loop merging, or loop unrolling with scalar replacement. FGen does this at the Σ-LL level using a set of predefined rules. To illustrate, consider the rules shown in Table 4.1 to fuse data access and loops. The full set of rewrite rules can be found in [79].

**Rewrite Engine**  The rewrite rules in Table 4.1 suggest that LMS would needs to match a whole subgraph in order to determine whether to transform two gathers. The LMS transformers are not designed for this use case, as they work in a peephole fashion, based on pattern matching LMS
case TP(s, PFIR(x, h)) if (bH > 0 && bV > 0) => {
  val (n, k) = (x.size, h.size)
  if (bV < k && bH < (n - k + 1)) {
    val (TV, SV) = (k / bV, k % bV)
    val (TH, SH) = ((n - k + 1) / bH, (n - k + 1) % bH)
    sigma(Const(tH), (i: Rep[Int]) => S(H(bH * i, 1, bH, n - k + 1), sum(
      sigmaSum(Const(tV), (j: Rep[Int]) => {
        val x1 = (H((bH * i) + (bV * j), 1, bH + bV - 1, n), x)
        val h1 = (H(k - (bV * (j + 1)), 1, bV, k), h)
        PFIR(x1, h1)
      }), {
        val x2 = (H((bH * i) + (bV * TV), 1, bH + SV - 1, n), x)
        val h2 = (H(0, 1, SV, k), h)
        PFIR(x2, h2)
      }))) ++ (if (SH == 0) {
        NullVector()
      } else S(H(bH * tH, 1, SH, n - k + 1), sum(
        sigmaSum(Const(tV), (j: Rep[Int]) => {
          val x3 = (H((bV * j) + (bH * tH), 1, SH + bV - 1, n), x)
          val h1 = (H(k - (bV * (j + 1)), 1, bV, k), h)
          PFIR(x3, h1)
        }), {
          val x4 = (H((bH * tH) + (bV * TV), 1, SH + SV - 1, n), x)
          val h2 = (H(0, 1, SV, n), h)
          PFIR(x4, h2)
        })))
    })
  } else if (bH < (n - k + 1)) {
    var (tH, SH) = ((n - k + 1) / bH, (n - k + 1) % bH)
    while (SH <= k) { SH += bH; tH -= 1}
    if (tH <= 0) {
      PFIR((H(0, 1, n, n), x), (H(0, 1, k, k), h))
    } else {
      val h1 = (H(0, 1, k, k), h)
      sigma(Const(tH), (i: Rep[Int]) => S(H(bH * i, 1, bH, n - k + 1), {
        val x1 = (H(bH * i, 1, bH + k - 1, n), x)
        microPFIR(x1, h1)
      })) ++ S(H(bH * tH, 1, SH, n - k + 1), {
        val x2 = (H(bH * tH, 1, SH + k - 1, n), x)
        microPFIR(x2, h1)
      }
    }
  }
  else {
    PFIR((H(0, 1, n, n), x), (H(0, 1, k, k), h))
  }
}

Figure 4.7: PFIR Decomposition in LMS
\[
\left( \sum_{j=0}^{m-1} A_j \right) M \rightarrow \left( \sum_{j=0}^{m-1} A_j M \right) \quad (4.20)
\]

\[
M \left( \sum_{j=0}^{m-1} A_j \right) \rightarrow \left( \sum_{j=0}^{m-1} MA_j \right) \quad (4.21)
\]

\[
G(s^{n\rightarrow N_1})G(r^{N_1\rightarrow N}) \rightarrow G(r \cdot s) \quad (4.22)
\]

\[
S(v^{N_1\rightarrow N})S(w^{n\rightarrow N_1}) \rightarrow S(v \cdot w) \quad (4.23)
\]

\[
\sum_{i=0}^{p-1} S(h_{b_{N_1\rightarrow N}}^{i}) \sum_{j=0}^{q-1} S(h_{br}^{n\rightarrow N_1}) \rightarrow \sum_{i=0}^{p-1} \sum_{j=0}^{q-1} S(h_{b+b_{br}}^{i\rightarrow N_1}) \quad (4.24)
\]

\[
\sum_{i=0}^{p-1} G(h_{b_{N_1\rightarrow N}}^{i}) \sum_{j=0}^{q-1} G(h_{br}^{n\rightarrow N_1}) \rightarrow \sum_{i=0}^{p-1} \sum_{j=0}^{q-1} G(h_{b+b_{br}}^{i\rightarrow N_1}) \quad (4.25)
\]

Table 4.1: Data access and loop merging rules.

definition nodes. Thus using this mechanism would imply matching a subgraph that can only be matched in a single forward iteration of the DAG, backtracking the dependencies. While this is a sound approach, can become tedious in LMS, as all matching patterns must be hardcoded. To work around this issue, FGen uses a designated rewrite engine for staged expression [85].

The rewrite engine uses an internal eDSL in Scala, simply called Pattern DSL, to express the rewrite rules in Table 4.1. The internal DSL overloads Scala operators <, > and -> and uses Scala companion object to reference \(\Sigma\)-LL constructs for transformation. Abstract symbols, are represented as strings. To illustrate this process, consider loop merging rule 4.21. The pattern will be expressed as:

1. \((\text{Scatter}, \text{"outer"}, (\text{Sigma}, \text{"from"}, \text{"to"}, \text{"i"}, \text{"body"}))) \rightarrow \)
2. \((\text{Sigma}, \text{"from"}, \text{"to"}, \text{"i"}, (\text{Scatter}, \text{"outer"}, \text{"body"})))\)

The rewrite engine, will take the pattern and a given \(\Sigma\)-LL DAG, and convert both into a Scalaz Trees, maintaining references to original DAG. Then, it will start traversing the tree obtained from the DAG to match the pattern tree. Once a match is found, the references to the original DAG are used as arguments to build the transformations defined in the pattern. As
the rewrite engine has the reference from the companion objects of the DSL constructs, it will use Scala reflection to invoke the routines that create new definitions in LMS and automatically convert the result into SSA form.

4.3 ABSTRACTING DATA PARALLELISM

Once an optimized Σ-LL program is reached it is converted to the I-IR / C-IR DSL. While Σ-LL deals with the representation of mathematical vectors, C-IR and I-IR facilitate the encoding of data representations of these vectors. For each data representation this includes all array operations composed by memory access functions and numerical computations in their scalar and vectorized versions, respectively. We abstract these different possibilities into a single data abstraction, which provides an interface that is very similar to its mathematical and conceptual equivalent in Σ-LL. These abstractions are implemented using staging, type classes and higher kinded types. All DSLs used in FGen are implemented through LMS. Translation from one DSL to another is performed via a staged interpreter approach as described in [11, 82]. While the implementation details are not relevant for Σ-LL, they are a key enabler in efficient abstraction at the I-IR and C-IR level. We give a quick, not entirely self-contained, overview over the main concepts in the next few paragraphs.

4.3.1 Parametric Stage Polymorphism

Utilizing the fact that staging within LMS is controlled through types allows us to use them in the context of type polymorphic functions and classes. The notion of switching between staged and regular expression by type parameters is called parametric staged polymorphism [70].

To illustrate this process, consider a regular addition function:

```scala
1 def add (a: Int, b: Int) = {
2   a + b
3 }
```

To create a staged function we only need to replace `Int` with `Rep[Int]`:

```scala
1 def add (a: Rep[Int], b: Rep[Int]) = {
2   a + b
3 }
```
Now, consider the identity type `Idt` (using Scala type aliases):

```scala
type Idt[T] = T
```

We can use it to rewrite the regular addition function into:

```scala
def add (a: Idt[Int], b: Idt[Int]) = {
  a + b
}
```

Looking at the two functions (regular addition and staged addition), we can observe that both functions have isomorphic bodies, and consequently can be merged into a polymorphic function that takes a single compile-time higher kinded type parameter `R[_]`:

```scala
def add[R[_]] (a: R[Int], b: R[Int]) = {
  a + b
}
```

This allows us to have a single implementation of the addition function, that we can switch between staged and regular expressions with a single parameter `R[_]`.

The same datatype applies to Scala for comprehensions which Scala treats as regular functions, with a parameter of type `Range` that can be overloaded in a similar fashion:

```scala
for (range: Idt[Range]) { body } // Regular loop
for (range: Rep[Range]) { body } // AST loop node
```

The first version executes the body expression, and the second version creates an AST node representing a loop that includes the body expression.

**Representing scalar and vector primitive types.** Scala natively supports signed scalar primitive types. Unsigned scalar primitives types are implemented using Scala Unsigned [35] and vector primitive types are implemented in LMS as abstract classes depicted at Figure 2.2 in Section 2.3.

However both scalar and vector types represent a continuous memory block of a given primitive of a different length. Thus we can represent both by a single generic type called `Packed[T]` such that `T` corresponds to a given scalar primitive. To explicitly distinguish between the two, we can use another identity type `Single[T]`:
Similarly as before, we can abstract `Single[_]` and `Packed[_]` using higher kinded type parameters.

**Abstracting Primitive Operations** Once we have polymorphic representations of scalar and vector instructions, we also need to define the operations that involve them. We use the notion of type classes, inspired by `Numeric[T]` in Scala, and define:

```scala
abstract class NumericOps[T:Manifest] {
  def plus (x: T, y: T) : T
  def minus (x: T, y: T) : T
  def times (x: T, y: T) : T
  def interleave (x: T, y: T) : (T, T)
  def unravel (x: T, y: T) : (T, T)
  // Syntactic sugar for convenience
  class Ops(lhs: T) {
    def +(rhs: T) = plus (lhs, rhs)
    def -(rhs: T) = minus(lhs, rhs)
    def *(rhs: T) = times(lhs, rhs)
  }
  implicit def mkNumericOps(lhs: T): Ops = new Ops(lhs)
}
```

This allows us to easily define extensions to the `NumericOps` class that deal with non-staged code involving scalar primitives:

```scala
class NumericIdtOps[T:Numeric:Manifest] extends NumericOps[Idt[Single[T]]] {
  def plus (x : T, y : T) = implicitly[Numeric[T]].plus (x, y)
  def minus (x : T, y : T) = implicitly[Numeric[T]].minus(x, y)
  def times (x : T, y : T) = implicitly[Numeric[T]].times(x, y)
  def interleave (x : T, y : T) = (x, y)
  def unravel (x : T, y : T) = (x, y)
}
```

or staged scalars primitives, as shown below:

```scala
  def plus (x : Rep[T], y : Rep[T]) = numeric_plus [T](x, y)
  def minus (x : Rep[T], y : Rep[T]) = numeric_minus[T](x, y)
  def times (x : Rep[T], y : Rep[T]) = numeric_times[T](x, y)
  def interleave (x : Rep[T], y : Rep[T]) = (x, y)
  def unravel (x : Rep[T], y : Rep[T]) = (x, y)
}
```
or even vector operations, as shown in Figure 4.8.

class PackedNumericOps[T:Numeric:Manifest] extends NumericOps[Rep[Packed[T]]] {
  private val vsize = codegen.getInstructionSetVectorSize[T]
  val m = implicitly[Manifest[T]]
  def plus (x: Rep[Packed[T]], y: Rep[Packed[T]]) = infix_vadd[T](x, y)
  def minus (x: Rep[Packed[T]], y: Rep[Packed[T]]) = infix_vsub[T](x, y)
  def times (x: Rep[Packed[T]], y: Rep[Packed[T]]) = infix_vmul[T](x, y)
  def interleave(x: Rep[Packed[T]], y: Rep[Packed[T]]) = m.toString() match {
    case "Int" | "Float" => codegen.instructionSet match {
      case SSE | SSE2 | SSE3 | SSE4 | SSE42 | AVX => {
        val inm1 = (2 << 6) + (0 << 4) + (2 << 2) + (0 << 0)
        val inm2 = (3 << 6) + (1 << 4) + (3 << 2) + (1 << 0)
        (infix_shuffle[T](x, y, inm1), infix_shuffle[T](x, y, inm2))
      }
    }
    case "Double" => codegen.instructionSet match {
      case SSE2 | SSE3 | SSE4 | SSE42 | AVX =>
        (infix_unpack_low[T](x, y), infix_unpack_high[T](x, y))
    }
  }
  def unravel(x : Rep[Packed[T]], y : Rep[Packed[T]]) = m.toString() match {
    case "Int" | "Float" => codegen.instructionSet match {
      case SSE | SSE2 | SSE3 | SSE4 | SSE42 | AVX =>
        (infix_unpack_low[T](x, y), infix_unpack_high[T](x, y))
    }
    case "Double" => codegen.instructionSet match {
      case SSE2 | SSE3 | SSE4 | SSE42 | AVX =>
        (infix_unpack_low[T](x, y), infix_unpack_high[T](x, y))
    }
  }
}

Figure 4.8: Abstracting operation on vector primitive types

Having abstracted scalar and vector primitives allows us to write code that is vector length agnostic, in terms of its underlying ISA. For example, consider the addition function. We extend it such that it works with the abstraction above:

def add[R[_], P[_]] (a: R[P[Int]], b: R[P[Int]])
  (implicit val numeric: NumericOps[R[T]]) = { numeric.plus(a, b) }
In this particular case, R[\_] will represent a type parameter that is either a \texttt{Rep[\_] or Idt[\_]}, and P[\_] will represent another parameter that is either a \texttt{Single[\_] or Packed[\_]. In this scenario, the function can be initialized with either vector or scalar types, while it can execute in the generator runtime, or generated runtime, according to R[\_].}

While the type parameters control the process of selecting the corresponding implementation, we still need to incorporate the implementation in order to be able to generate code.

The implementation required for \texttt{numeric.plus} will be encoded in the type class \texttt{numeric}, that is implicitly passed into the function.

\textbf{Abstracting complex/real elements}  
\texttt{FIR} domain deals with both complex and real numbers. Therefore, to be able to handle both cases, we need to be able to abstract both representations:

\begin{codesmall}
abstract class Element [+T] {}

case class Real [+T] (_re: T) extends Element[T] {}

case class Complex [+T] (_re: T, _im: T) extends Element[T] {}
\end{codesmall}

And similarly to the abstraction defined for \texttt{NumericOps}, we can abstract on top of the \texttt{Element} class:

\begin{codesmall}
abstract class ElementOps[ElementClass[_\_], T: NumericOps]
  (implicit m: Manifest[ElementClass[T]]) { self =>
  def plus (x: ElementClass[T], y: ElementClass[T]): ElementClass[T]
  def minus (x: ElementClass[T], y: ElementClass[T]): ElementClass[T]
  def times (x: ElementClass[T], y: ElementClass[T]): ElementClass[T]

class Ops(lhs: ElementClass[T]) {
  def +(rhs: ElementClass[T]) = plus (lhs, rhs)
  def -(rhs: ElementClass[T]) = minus(lhs, rhs)
  def *(rhs: ElementClass[T]) = times(lhs, rhs)
}

implicit def mkElementOps(lhs: ElementClass[T]): Ops = new Ops(lhs)
\end{codesmall}

\texttt{ElementOps} abstraction expects the \texttt{ElementClass} to have \texttt{T} as an underlying type, and the \texttt{NumericOps} as an implicit case class parameter. Similarly to \texttt{Numeric} in Scala, we can use this class to generalize the computations:

\begin{codesmall}
class ComplexOps[T:Manifest](implicit nops: NumericOps[T])
  extends ElementOps[Complex, T]
\end{codesmall}
The complex abstraction does not have to reason whether the underlying primitives are of a particular type, or whether are vectorized or not. This information is already provided by the NumericOps class. The abstraction that represents the real numbers is implemented in a similar fashion.

**Abstracting Memory Layout** Finally, once we have the building blocks for representing elements, and primitives, we can put everything together to abstract the memory layout. We achieve this by the highly polymorphic classes that take the shape:

```
class CVector[V[_], E[_], R[_], P[_], T](...) {
  type Element = E[R[P[T]]]
  def alloc (s: V[Int]): V[Array[P[T]]]
  def apply (i: V[Int]) : Element
  def update (i: V[Int], v: Element)
}
```

For simplicity the code above omits the type classes that are implicitly passed together with each type parameter. Within each of the type parameters provided to the CVector class we encode part of the abstraction, by providing abstract composable interfaces that are implemented through those type classes.

- **T** describes the underlying array primitive (double, float, etc).
- **P[_]** describes whether we deal with SIMD or SISD instructions. It is accompanied by a type class that abstracts SIMD specific operators
such as shuffle, hadd, vadd etc., and SISD specific operations such as addition, multiplication etc.

• R[\_] describes whether we stage the elements of the array. The accompanying type class abstracts numerical operations for both staged and non-staged version.

• E[\_] describes the the structure of one array element. E.g. it can be complex numbers consisting of two primitives or directly primitives. E[\_]. Abstractions for numerical operations relative to the element such as addition, multiplications, complex number interleaving are implemented in the accompanied type class.

• V[\_] encodes whether array accesses will be visible in the target code element operations.

A concrete choice of a data layout, code style and the vectorization is done through instantiating the CVector class accordingly. For example, in order to initialize staged scalar arrays, we can use:

```scala
1 class StagedSingleArrayOps[T:Manifest] extends CVector[Rep, Rep, Single, Idt, T] {
2  def alloc (s: Rep[Int]): Rep[Array[T]] = array_obj_new[T](s)
3  def apply (x: Rep[Array[T]], i: Rep[Int]): Rep[T] = array_apply[T](x,i)
4  def update(x: Rep[Array[T]], i: Rep[Int], y: Rep[T]) = array_update[T](x,i,y)
5 }
```

We can as well define an array of staged scalar elements:

```scala
1 class ScalarSingleArrayOps[T:Manifest] extends CVector[Idt, Rep, Single, Rep, T] {
2  def alloc (s: Idt[Int]): Array[Rep[T]] = new Array[Rep[T]](s)
3  def apply (x: Array[Rep[T]], i: Idt[Int]): Rep[T] = x(i)
4  def update(x: Array[Rep[T]], i: Idt[Int], y: Rep[T]) = x.set(i, y)
5 }
```

or a staged vector array (i.e., each load results in a vector type, and each store is done with a vector type):

```scala
1 class StagedPackedArrayOps[T:Manifest] extends CVector[Rep, Rep, Packed, Idt, T] { val vecLength = codegen.getInstructionSetVectorSize[T]
2  def alloc (s: Rep[Int]): Rep[Array[T]] = array_obj_new(s)
3 ```
4.3 Abstracting Data Parallelism

In the translation process from Σ-LL to I-IR, this enables us to define the target translations in terms of the type polymorphic base class CVector as, e.g.,

```python
def apply(x: Rep[Array[T]], i: Rep[Int]): Rep[Packed[T]] =
    infix_vload[T](x, i / vecLength)
def update(x: Rep[Array[T]], i: Rep[Int], y: Rep[Packed[T]]) =
    infix_vstore(x, i / vecLength, y)
```

Concrete implementations can be picked by passing corresponding instantiations of the base class to the function. Figure 4.9 illustrates this for four variants that could be passed to the add functions.

Staging decisions are of crucial importance to FGen. When tiling is performed to fit the SIMD vector length, SIMD instantiations are performed to generate the vectorized part of the code; the leftover computation is instantiated as scalar SISD code. When tiling for registers, data structures are properly instantiated to generate the unrolled version of the code. Data abstraction and staging decisions enable a single codebase that fits all generated code versions.

4.3.2 ISA Abstraction

Once data types are fixed, the next step is to replace each Σ-LL expression with one or several I-IR expressions. The Intrinsics IR is ISA independent and it only specializes to a particular ISA, once this argument is fixed in the generator. The specialization to a particular ISA is inter-dependent on the data abstraction. We observe this in the case of Interleaved Complex Array. Real and imaginary parts must be interleaved when data is loaded or stored. To achieve this, the ISA abstraction calls the corresponding shuffle, unpackhi and unpacklo instructions (showed in PackedNumericOps in Figure 4.8), to perform the desired interleaving.

**SIMD Intrinsics** Once the ISA is fixed, we perform the conversion of Σ-LL expressions to I-IR. FGen implements translation of Σ-LL to I-IR in
a single codebase and uses staging decisions to generate different code versions. The actual SIMD conversion of the PFIR filter is given below:

```scala
class SigmaLL2CIRTranslator[E[-], R[-], P[-], T] {
  type Element = E[R[P[T]]]
  def sum (in: List[Element]) =
    if (in.length == 1) in(0) else {
      val (m, e) = (in.length / 2, in.length)
      sum(in.slice(0, m)) + sum(in.slice(m, e))
    }
  def translate(stm: Stm) = stm match {
    case TP(y, PFIR(x, h)) =>
      val xV = List.tabulate(k)(i => x.apply(i))
      val hV = List.tabulate(k)(i => h.vset1(h.apply(k-i-1)))
      val tV = (xV, hV).zipped map (_*_)
      y.update(y(0), sum(tV))
  }
}
```

Note that if ISA is not specified, the code snippet will result with a construction of SISD C-IR code.

**Code level optimizations**  Code level optimizations are done on the C-IR DSL. Most of these are already provided by LMS. Those include common sub-expression elimination, dead code removal and code motion.

4.4 **Experimental results**

In this section we show performance benchmarks with FGen generated code against current commercial libraries.

**Experimental setup**  We execute our benchmarks on two machines, an Intel(R) Xeon(R) CPU E5-2643 3.3 GHz supporting AVX, running Ubuntu 13.10, kernel v3.11.0-12-generic, and an Intel(R) Core(TM)2 Duo CPU L7500 1.6GHz supporting SSSE3, running Debian 7, kernel v3.2.0-4-686-pae. Intel’s Hyper-Threading, Turbo Boost (Xeon) and Intel Dynamic Acceleration (Core2) were disabled on both machines during the tests. We compare against convolutions from Intel IPP v8.0.1 and Intel MKL v11.1.1. Note that in both, the vector lengths are parameters in contrast to our generated specialized code. As base line we also include a straightforward implementation of convolution: a double loop corresponding to (4.1) with fixed array
\[\text{Figure 4.9: Different Data Type Instantiations result with different code style (assuming arrays of size 4 and AVX as an ISA)}\]
sizes. All code is compiled using the Intel C++ Composer 2013.SP1.1.106, with flags `-std=c99 -O3 -xHost`.

We only consider double precision code (4-way on AVX and 2-way on SSSE3). The input sizes, related to the input vector of the convolution expression, are powers of two in the form of \( n = 512 \cdot 2^i \) for \( i = 1, \ldots, 16 \) to ensure a sampling of all cache levels for both machines. For each machine we perform two types of tests:

1. All vectors are arrays of real numbers, and the filter size is 8 or 20;

2. All vectors are arrays of interleaved complex numbers, and the filter size is 8 or 20 (complex numbers).

Time is measured under warm-cache conditions, using a two loops measuring strategy. The inner loop measures time as the mean of sufficiently many iteration; the outer loop returns the median of several such runs.

Figure 4.11 gives an overview of the results. All plots show the size of the input vector on the x-axis and the performance in flops per cycle (f/c) on the y-axis. The theoretical peak performance of the platform is represented with a horizontal line in each plot. We discuss real (left four plots) and complex (right four plots) convolutions separately.

**Real Convolution.** FGen-generated code outperforms the other implementations, except IPP for small sizes and 20 taps. The reason is not clear as the code is distributed as binary, which prevents inspection. In some cases MKL performs worse than the base implementation. Apparently, `icc` can efficiently optimize and vectorize the simple double loop with fixed bounds.

**Complex Convolution.** For large sizes FGen-generated code is faster (AVX) or roughly competitive (SSSE3) with the next best IPP. Again, MKL performs worse than the straightline code with a similar possible explanation as above. We note that in FGen there is further room for improvement in the shuffling needed to work on interleaved data. We believe that the gains for larger sizes on AVX are due to a more thorough exploration of the possible tiling strategies in FGen.

**Remarks.** We note for longer sizes of the filter \( h \), both IPP and MKL outperform FGen due to the use of FFTs, which reduces the asymptotic runtime from \( O(nk) \) to \( O(n \log k) \). FGen does not support FFT-based convolution at this time. Further, note that our generated code is specific to the
filter size in contrast to IPP. However, the generator could be extended to support general size, using the techniques described prior work [86].

4.5 RELATED WORK

The array programming paradigm favors computing on collections of data as a whole over element-at-a-time processing. Besides a higher-level programming style, the key benefit of going from scalar values to vectors of data as core computational units is that array computation is implicitly parallel, and easy to map to SIMD instructions by a compiler. Starting with a stylized mathematical array notation that lead to APL [87] and its successors, J [88] or K [89], array languages have focused on user-facing constructs that enable programmers to abstract over the rank, dimension or in general shape of the data. The same holds for Sisal [90] or, more recently, SAC [91], which has also inspired embedded DSLs [92, 93]. Compilers for all these languages attempt to generate SIMD intrinsics to varying degrees, e.g., by using type inference in the case of SAC [94], but in general do not expose this fact to the programmer in any way. Our focus in this paper has been on using an array-style language, Σ-LL, as an intermediate language in a program generator stack. As opposed to other array languages, Σ-LL models only single-dimensional arrays, i.e., does not provide shape polymorphism in the usual sense. The translation from Σ-LL to I-IR/C-IR and optimized C code, however, is highly parametric in data layout, vector ISA, and other parameters. This is again in contrast to user-facing languages, where the low-level part of the compilation is hidden from the programmers. We believe that our Σ-LL / I-IR combination offers a sweet spot in the design space between fully opaque array languages (which do not offer fine grained control about vectorization) and ISA specific intrinsics as provided by C compilers (which provide full control but are too low-level and cumbersome to use).

4.6 SUMMARY

The work presented in this chapter demonstrates that we can use staging and meta-programming to derive high-level abstraction to represent data parallelism. To achieve that, we use type classes to encapsulate staging decisions and drive the staging process though type parameters. Consequently, the work demonstrates that parametric stage polymorphism can be extended to facilitate data parallelism abstraction.
Figure 4.10: Performance comparison of FGen, IPP, MKL and a base implementation running on Intel(R) Xeon(R) CPU E5-2643 3.3 GHz, using Ubuntu 13.10. FGen produces AVX code.
Figure 4.11: Performance comparison of FGen, IPP, MKL and a base implementation running on Intel(R) Core(TM) 2 Duo CPU L7500 1.6 GHz, Debian 7. FGen produces SSSE3 code.
Furthermore, we show how we can apply this abstraction in building modular DSLs and demonstrate the design of Σ-LL language. Our implementation can successfully decompose convolution expressions into smaller convolutional filters, perform high level optimizations and map the mathematical representation into SIMD instructions. Results presented suggest that we can provide implementation that is competitive with commercial libraries, in some cases outperforming the Intel IPP hand-tuned implementation.
Object-Oriented Stage Polymorphism

In the previous chapter, we demonstrated the use of parametric stage polymorphism in designing high-performance DSLs. The ability to control staging decision with types, and encapsulate abstractions in type classes provides a convenient solution for designing statically typed DSLs. On the other hand, high-level dynamically typed mathematical DSLs such as MATLAB, Wolfram, R have a wide adoption worldwide by scientists, engineers and students. The high-level representation and lack of types make these DSLs easier to use in general mathematical computations and suitable for rapid prototyping. Consequently, in this chapter we explore the following research question:

• Can we use the LMS approach of “staging by types” and stage polymorphism for staging a dynamically typed DSL?

To explore this question, we use LMS to build a prototypical generator that takes a MATLAB program and produces equivalent C code. In order to produce statically typed low-level C code and ensure ahead-of-time compilation, the generator would need to be capable of performing a wide range of analyses and transformations. These include type inference, high-level computation lowering, analysis on dimensions on matrices and vectors, memory allocation and many others. Typically, these analyses and transformations are executed in phases, requiring the generator to store, query and propagate the information across phases. As long as these informations are stored in the IR of the DSL, they are effectively increasing its complexity. This is why analyzing high-level dynamic programming languages is challenging, while analyzing DSLs with less complex intermediate representations is simpler. This leads us to the second research question in this chapter:
• Can we use stage polymorphism to reduce the complexity in the intermediate representation of a given DSL and thus simplify the required analysis and transformations?

The chapter is organized as follows. Section 5.1 provides the motivation for building a MATLAB generator and introduces the concept of object oriented stage polymorphism. Section 5.2 provides the necessary background to define the language specification of MATLAB. Section 5.3 introduces the notion of metacontainers, the base abstraction used to build our generator. Section 5.4 gives an overview of the generator, the metacontainers involved in staging MATLAB and how they interoperate with the analysis and transformations for lowering to low-level C code. Section 5.5 demonstrates how our approach compares to recent MATLAB code generators. Section 5.6 discusses related work and 5.7 provides the summary.

The work in this chapter was published in the 2019 International Conference on Generative Programming: Concepts & Experiences [95].

5.1 Motivation

MATLAB is the de facto standard high-level programming language and interactive numerical computing environment in many domains in engineering and science. Scientists, engineers and students have widely adopted MATLAB due to its flexible and high-level language expressiveness, which makes it suitable for fast modeling and prototyping. The prototypes being built are often testbeds for new applications that might operate on a variety of architectures ranging from mobile and embedded devices to data centers. Once complete, these prototypes are typically reimplemented using low-level high-performance languages for efficiency and deployment.

To automate the transition from prototype to low-level code, there has been extensive research in automatically optimizing, interpreting, translating and compiling MATLAB [96–115]. The rich set of 3520 MATLAB functions [116] (as of version 2019a), of which 473 are considered as language fundamental functions and 536 as mathematical functions, is the reason why all prior work focused on implementing a subset of MATLAB language.

However, even in a reduced subset, compiling or translating MATLAB to a low-level language is challenging due to lack of type / shape information, dynamic aspects of the language, and many overloaded functions that include even basic arithmetic operators such as +, −, ×, /. Consequently,
not a single prior effort has ever focused on implementing the complete specification, even on a reduced subset of the language.

LMS, combined with parametric stage polymorphism approach, as described in Chapter 4, offers high-level abstractions to represent the rich set of functions and overloaded operators, and simplify the construction of a MATLAB generator. However, parametric stage polymorphism, deployed in FGen and SpiralS [70], has only been shown to work with statically typed DSLs. Porting it to a dynamic language is challenging, as this approach, as well as LMS, rely entirely on the use of static type information which is unavailable in a dynamic language.

To alleviate this challenge, we can design a multi-phase compiler that performs type inference first and then stages the rest of computations with the inferred types. But in that case, types only become available at runtime of the compiler. Even though we can override staging behaviours in type classes at runtime, type parameters (responsible for resolution of type classes) can only be instantiated at compile-time in Scala.

One way to work around these issues is to simply drop all type parameters and pass types as values. As variables can hold multiple types in a dynamic language, type values can be symbolic variables as well. In the absence of type parameters, type classes, responsible for abstraction encapsulation, are effectively reduced to regular instances. Therefore, we no longer deal with parametric stage polymorphism, but simply object oriented stage polymorphism. Staging decisions are no longer controlled by type parameters, but subtyping and overriding regular object instances. Without type classes, the staged variables passed as arguments in staged functions, must also pass their type values, as well as the encapsulated abstractions. To avoid the overhead, we can combine all three in a single object wrapper instance. In fact, this method of wrapping staged variables is not limited to type values only, but can be extended to other stage variable properties. For MATLAB, we can also include as information the dimension and structure of matrices. We refer to such object instances as metacontainers as they carry metadata.

Metacontainers inherit all properties from the object-oriented paradigm, but also introduce additional properties relevant to a staging environment. Namely, metacontainers, when present, define the relationship between the encapsulated staged variables. Combined with the sea of node representation in LMS (which explicitly represents control dependencies) this

\[ \footnote{Note that type classes can be overridden as well, and different instances can be explicitly passed in functions. Therefore, parametric stage polymorphism is not different from object oriented polymorphism, but rather a specific form of it, based on type parameters.} \]
metacontainer property can be used to reduce IR complexity, by storing information away from the IR.

Metacontainers can encapsulate abstractions in the same manner as type classes. Therefore, they can provide powerful abstractions to build a MATLAB generator, particularly when there is need to support many overloaded operators and when data-parallelism is a requirement. Therefore, we build our generator around the idea of metacontainers, and provide the following contribution:

1. We demonstrate the use of object oriented stage polymorphism and metacontainer abstraction as a tool for building compilers. Specifically we show the following properties of metacontainers:
   a) Metacontainers can reduce the complexity of IRs.
   b) Metacontainers can simplify analysis.
   c) Metacontainers can simplify transformations.
   d) Metacontainers can be used to specialize IRs.

2. A prototypical MATLAB to C code generator that supports a subset of the MATLAB language. Unlike prior work, our prototype provides systematic type inference involving all numerical primitives in the supported subset. It handles many dynamic aspects of MATLAB, and generates correct code with explicit vectorization.

In the subsequent sections, we provide examples of each of the metacontainers properties, and how they apply in the design of our generator. The work in this chapter does not focus on generating high-performant code for MATLAB. This would require implementation of a variety of performance-related optimisations, which are outside the scope of this project.

5.2 MATLAB

In this section we give an overview of the MATLAB language specifications.

MATLAB types and variables Variables in MATLAB are mutable and untyped, and can be either global or local. We classify them into three groups:

1. Multidimensional matrices, which we refer to as tensors, consist of single or double precision floats, 8-, 16-, 32- or 64-bit signed or unsigned integers, representing real or complex values. Tensors can
haven unbounded number of dimensions and each dimension can have unbounded value. We use the term shape to refer to the dimension of a given tensor. As long as the imaginary component is not present, tensors can be character arrays or boolean arrays. In total there are 12 primitive types, with double precision floating point as the default type.

2. Cells, which represent structures and objects. Each cell can consist of tensors, cells, or functions, which correspond to the fields and methods in an object oriented model.

3. Functions, which can be nested or anonymous, representing closures. Each function takes multiple tensors, cells or functions as input arguments and produces one or multiple outputs. Functions can be passed as function handles, and both input and output of function can have variable length.

Apart from the three base groups above, recent versions of MATLAB also provide additional datatypes, including:

- Sparse matrices to provide efficient storage of IEEE754 data that has a large percentage of zero entries.
- Strings as an alternative data-type to character arrays.
- table which represents arrays with named variables and timetables which represent time-stamped data in tabular form.
- Categorical arrays for storing data with values from a finite set of discrete categories.
- datetime arrays that represent points in time using the proleptic ISO calendar up to nanosecond precision, duration arrays that represent elapsed times in units of fixed length, such as hours, minutes, and seconds and calendarDuration that represents lengths of times in variable length calendar units.
- Persistent variables, which are local function variables, such that their values are retained in memory between calls to the function.
MATLAB stores elements in column-major-order. Multidimensional arrays are stored in pages, such that each page corresponds to a 2-dimensional matrix. Pages are then linearly stored in memory, one after the other. An illustration of a 3-dimensional tensor is shown in Figure 5.1.

**Indexing** MATLAB employs a versatile mechanism for indexing tensors. The indexing is done using one or several subscripts. These can be scalar literals, scalar variables, vector expressions, vector variables and colon notation (:) The behaviour depends on the shape of the variable at a given point in the program. The indexing can be applied to obtain values from a tensor, or to set values in a tensor. We refer to the former as indexed read and to the latter as indexed assignment. We group the indexing into several categories:

- **Linear indexing.** When a tensor is indexed using only one subscript, MATLAB treats the tensor as if its elements were strung out in a long column vector. Once an indexed read is performed, the index must be within the boundary of the tensor. When an indexed assignment is performed, accesses outside the boundaries extend the dimensions of the vector. The subscript could be a scalar or a tensor. If a scalar element is provided, only a single element is returned or assigned. Once a tensor is provided, the returned tensor has the shape of the subscript tensor, such that each element, corresponds to the linear
index of the subscript tensor. If indexed assignment is used, the elements referenced by the subscript tensor are assigned. Assignments are done with a scalar, or another tensor with identical shape as the subscript tensor. To illustrate some of the different variants, consider the examples below:

```matlab
>> M = reshape((1:12) * 2, 3, 4)
M =
         2         8        14        20
         4        10        16        22
         6        12        18        24

% linear indexed read within bounds
>> M(5)
ans = 10

% linear indexed read outside bounds
>> M(17)
Index exceeds matrix dimensions.

% linear indexed read with a tensor
>> idx = [2 3; 4 5];
>> M(idx)
ans =
         4         6
         8        10

% linear indexed assignment with a scalar
>> M(idx) = 0
M =
         2         0        14        20
         0         0        16        22
         0        12        18        24

% linear indexed assignment with a tensor
>> M(idx) = [10 11; 12 13]
M =
         2        12        14        20
        10        13        16        22
        11        12        18        24

% dimension expansion using indexed assignment
>> M = 1:5;
>> M(8) = 4
M =
      1         2         3         4         5         0         0         4
```
• **Indexing with multiple subscripts.** When multiple subscripts are given, they extract or assign parts of the vector according to its dimensions. A single : in a subscript position is a shorthand notation for 1:end and is often used to select entire rows or columns. Each subscript could be a tensor or a scalar. We describe the indexing in three steps:

1. Each subscript tensor is first linearized into a vector (a scalar is considered as a vector of size 1).
2. The values of each subscript vector index along the axis of the dimension that corresponds to the subscript.
3. The intersection of all values in all dimensions is the tensor being indexed.

To illustrate, consider the example below:

```matlab
% M is a 3-dimensional matrix
>> M = reshape(1:(2 * 3 * 4), 4, 3, 2);
>> M
M(:,:,1) =
    1   5   9
    2   6  10
    3   7  11
    4   8  12
M(:,:,2) =
   13  17  21
   14  18  22
   15  19  23
   16  20  24
% Extract a portion of it having size 2x2x2
>> M([3 4], [2 3], [1 2])
ans(:,:,1) =
    7   11
    8   12
ans(:,:,2) =
   19  23
   20  24
```

Similarly to the case of linear indexing, the intersection must be within bounds when indexed reads are performed. For an indexed assignment, length in each dimension can be extended. Also the number of dimension can be extended. For example, extending dimensions of a matrix and extending a 3-dimensional tensor into a 4-dimensional one can be done by:
• **Logical indexing.** In logical indexing, a single, logical array is provided as subscript. MATLAB extracts the nonzero elements corresponding to the nonzero values of the logical array for indexed reads, or overwrites them for indexed assignment. For indexed read, the output is always in the form of a column vector. Both indexed assignments and indexed reads can not exceed bounds in this case.
**MATLAB built-in operators**  Built-in MATLAB operators provide arithmetic, orderings, and boolean logic that can be applied to tensors of any dimensions. Some of these operators are exceptions, such as a matrix multiplication and matrix transposition, which are designed to work with 2-dimensional matrices only. Unlike arithmetic operators in C/C++, MATLAB uses *saturated arithmetic*. When an arithmetic operator is applied to two integer values, or a floating point value and an integer value, the obtained result is saturated to the range of the integer, effectively avoiding overflows and underflows. This does not apply when the two operands are integers from different classes (say 32-bit and 16-bit integer) and thus not all type combination are overloaded. MathWorks [117] provides regular updates to functions, providing implementation to new type combinations in subsequent MATLAB versions. We refer to the different type combination in operators as *type interaction*.

Operators are not only overloaded by types, but also by shape. Namely, all pointwise operations such as addition, subtraction, multiplication, division, are overloaded such that they operate on two tensors having equal dimensions, but also produce results if one operand is a tensor of any size, while the other is a scalar.

**MATLAB loops**  The language supports for and while loops. The for loops are specified with a start and an end value, as well as an iterator, each represented by a scalar having one of the 12 primitive types. If a non-scalar is provided for either value, MATLAB will extract the non-imaginary part of the first element. Alternatively, the for loops can iterate over all values of a tensor, similarly as a foreach construct. MATLAB loops support early exit and continuation of loops with the break and continue constructs.

**MATLAB conditionals**  Conditionals are implemented using switch and if-then-else constructs. The if and while conditionals are formed with a conditional expression that evaluates to an arbitrary tensor having one of the 12 primitives. MATLAB implicitly converts each conditional expression into a boolean value, expecting non-null values over all elements in the non-imaginary component of the tensor. switch is comprised of switch-expressions and case-expressions which can be of any type. Similar to C/C++ and other languages, the case-expressions must be constant expressions.
MATLAB exceptions Exception handling is supported in MATLAB through throw-try-catch constructs. The exception terminates the running function and returns control either to the keyboard or to an enclosing catch-block. When exception is thrown, it is based the information contained in the MException object.

5.3 Metacontainers

In this section we provide a detailed overview of the notion of metacontainers. We also demonstrate how the encapsulated abstractions in type classes, described in Section 4.3, can be adjusted for usage through metacontainers.

LMS stages functions through type parametrization, using the higher kinded type Rep[T]. For each type, a type class instance of Typ[T] is implicitly provided in staged functions, to maintain runtime type information and avoid type erasure. However, when staging a dynamically typed language, types are not available ahead of time, and thus type parameters are no longer a viable option to propagate type information. Instead we must resort to type values. Type values could be provided explicitly in the form of Typ[T] instance during runtime, however, as a dynamic language can hold multiple types for a variable at the same point in the program, type values can as well be symbolic values. Consequently, staged functions can no longer be overloaded by types, and type values must be passed to staged functions.

To simplify usage, staged variables can be “boxed” in wrapper classes, with their type values. To illustrate, consider a staged variable value and its type value typ:

```plaintext
val typ : Rep[Typ] // Representation of a type for variable "value"
```

In order to implement a NumericRepOps type class like functionality, defined in Section 4.3, we drop the type parameters and include type values as arguments. As we are unable to overload staged functions on different types, each operator must reason about type combinations, in order to dispatch the proper computation:

\[\text{Assuming presence of partial evaluation, the dispatch will happen either during staging or in the runtime of the generated code, depending on whether the type combination is comprised of constant type values or not.}\]
class NumericRepOps {

def plus(a: Rep[Any], aTyp: Rep[Typ], b: Rep[Any], bTyp: Rep[Any]) = {
    (aTyp, bTyp) match {
        case (DoubleTyp, DoubleTyp) => numeric_plus[Double](lhs, rhs)
        case (DoubleTyp, FloatTyp ) => plus(lhs, cast(rhs, DoubleTyp))
        case (DoubleTyp, IntTyp) => {...} // Implement other type combinations
    }
}

def minus(a: Rep[Any], aTyp: Rep[Typ], b: Rep[Any], bTyp: Rep[Any]) = {...}

def times(a: Rep[Any], aTyp: Rep[Typ], b: Rep[Any], bTyp: Rep[Any]) = {...}

def cast(a: Rep[Any], aTyp: Rep[Typ]) = {...}
}

However, this is rather inconvenient, as the number of arguments for every operator is doubled. To work around it, we can wrap the staged variable into a class wrapper, and including the type value inside:

class Number {
    // class wrapper i.e. "metacontainer"
    public val value: Rep[Any] // metacontainer reference
    public val typ : Rep[Typ] // additional meta information
}

We call this wrapper class a metacontainer. As the metacontainer is defined for the staged variable value, we refer to it as a _metacontainer reference_. The staged variable typ represents property of the metacontainer reference or in other words it carries its _metadata_.

Operator signatures in NumericRepOps are now simplified, such that they use metacontainers as arguments. Each computation creates a new metacontainer that corresponds to the result of the expression.

class NumericRepOps {
    def plus (a: Number, b: Number) = (a.typ, b.typ) match {
        case (Const(DoubleTyp), Const(DoubleTyp)) =>
            new Number(DoubleTyp, numeric_plus[Double](a.value, b.value))
        case (DoubleTyp, FloatTyp ) => plus(a, cast(b, DoubleTyp))
        case (DoubleTyp, IntTyp) => {...} // Implement other type combinations
    }
    def minus(a: Number, b: Number) = { ... }
    def times(a: Number, b: Number) = { ... }
    def cast (a: Number, tp: Rep[Typ]) = { ... }
}

With the changes above, we can replicate the abstraction of a type class for dynamically typed variables. However, the issues caused by the lack of type parameters do not end here. Without a type parameter, instances of
NumericRepOps can no longer be implicitly passed to type classes that build on top of it. In particular, implementing operators for ComplexOps like functionality, defined in Section 4.3, would require instances of NumericRepOps, in addition to the operands. We can avoid this by placing the operators of NumericRepOps as routines in the metacontainer:

```scala
class Number(val typ: Rep[Typ], val value: Rep[Any]) {
  def plus (rhs: Number): Number = (typ, rhs.typ) match {
    case (DoubleTyp, DoubleTyp) =>
      new Number(DoubleTyp, numeric_plus[Double](a.value, b.value))
    case (DoubleTyp, FloatTyp) => plus(rhs.cast(DoubleTyp))
    case (DoubleTyp, IntTyp) => // Implement other type combinations
  }
  def minus (rhs: Number) : Number = { ... }
  def times (rhs: Number) : Number = { ... }
  def cast (tp: Typ[Any]) : Number = { ... }
  class Ops(lhs: Number) {
    def +(rhs: Number) = plus (lhs, rhs)
    def -(rhs: Number) = minus(lhs, rhs)
    def *(rhs: Number) = times(lhs, rhs)
  }
  implicit def mkElementOps(lhs: Number): Ops = new Ops(lhs)
}
```

Finally, we can replicate ComplexOps using:

```scala
abstract class Element {}

case class Real ( _re: Number) extends Element {}

case class Complex (_re: Number, _im: Number) extends Element {

abstract class ElementOps { self =>
  def plus (x: Element, y: Element): Element
  def minus (x: Element, y: Element): Element
  def times (x: Element, y: Element): Element
}

class ComplexOps extends ElementOps {
  def plus (x: Complex, y: Complex) = Complex(x._re + y._re, x._im + y._im)
  def minus (x: Complex, y: Complex) = Complex(x._re - y._re, x._im - y._im)
  def times (x: Complex, y: Complex) = {
    val m1 = x._re * y._re
    val m2 = x._im * y._re
    val m3 = x._re * y._im
    val m4 = x._im * y._im
    Complex(m1 - m2, m3 + m4)
  }
}
```
The example above illustrates the use of metacontainers, and shows how we can encapsulate a given staged variable with its metadata, as well as the abstractions associated with it. To define metacontainers formally, we use the following definition:

**Definition 5.3.1** A Metacontainer is a runtime object wrapper for a given metacontainer reference that includes its metadata and a set of routines associated with it. In every metacontainer, the metacontainer reference is a set of at least one staged variable and the metadata is a set of runtime objects, staged variables or other metacontainers.

Metacontainers leverage all properties in the object-oriented paradigm. As they are objects, they can be nested, can be polymorphic or their routines can be overridden. In addition, they exhibit interesting properties once used in a staging environment:

- **Metacontainers can be specialized.** When either of the staged variables is known to be constant, routines in the metacontainer can be simplified through specialization. Consider the implementation of the `Number` metacontainer, shown above. If a type value is known to be a constant, then pattern matching is only required for the right-hand-side operand.

- **Metacontainers can store information outside an IR.** Metacontainers can be used to define relationships between staged variables contained inside the metacontainer. This allows us to define high-level dependencies, that otherwise would have to be encoded in the IR representation. For example, without metacontainers, we would have to create a new definition in LMS to maintain the relationship between a variable and its type:

```scala
1  case class ValAndType(value: Rep[Any], typ: Rep[Typ]) extends Def[Number]
```

Consequently, even standard numeric operations, such as addition or subtraction, will have to be redefined to operate with the `Exp[Number]` type. With metacontainers, however, we are able to keep the representation of `Number` “unboxed” inside the IR. Assuming that value is of primitive type, this allows us to use the already existing definitions and optimizations on primitives arithmetic operations in LMS.

- **Metacontainers are disposable.** One staged variable, or a metacontainer reference can have multiple containers, each storing different
types of metadata used for various analysis and transformation phases. When metadata changes, new metacontainers can be created on the fly to represent the new information, or to specialize their routines. Once a phase is completed, the created metacontainers can be disposed, and the subsequent phase can use other metacontainers for a new round of analysis and transformations.

These three design properties of metacontainers are used heavily in the next section, as we build most of the analysis and transformations in our generator, using these concepts.

5.4 MGEN

In this section we describe the details of the implementation of MGen. First we define the supported subset of MATLAB and provide a high-level overview of the design and generator phases. Then, we describe in detail the methodology used to infer types and shapes, and how we lower the computations to low-level C representation.

5.4.1 Supported Subset

MGen handles user-defined global functions, provided as .m files. However, it does not handle nested functions, anonymous functions or function handles. Each provided function in MGen must have a fixed number of inputs and outputs, and a variable length of arguments is not supported.

MGen does not support global variables, structures, cell arrays, or MATLAB classes. It also does not support recently included MATLAB types such as strings, sparse matrices, tables, timetables, categorical arrays, date and duration arrays, and persistent variables.

MGen supports all numeric classes including double, single, int8, int16, int32, int64, uint8, uint16, uint32 and uint64. It also supports boolean types and characters. We can use the 12 primitive types to create multi-dimensional arrays with variable-size complex or real data.

MGen supports double-precision, single-precision, and integer math via saturated arithmetic. Supported built-in operators include arithmetic, relational, and logical operators. Trigonometric functions are also included as are matrix operators such as transpose, matrix-matrix multiplication and others. The complete list of supported functions is given in Table 5.1.

Initialization of tensors is supported thorough zeros, ones and rand, which initialize the data with values of 0, 1 or random values, respec-
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+ - ./ .* .^</code></td>
<td>Pointwise arithmetics</td>
</tr>
<tr>
<td><code>&lt; &lt;= &gt; &gt;= == ~=</code></td>
<td>Relational operators</td>
</tr>
<tr>
<td>`&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td><code>mtimes</code></td>
<td>Matrix-matrix multiplication</td>
</tr>
<tr>
<td><code>A' / transpose</code></td>
<td>Matrix transpose</td>
</tr>
<tr>
<td><code>a/sin a/cos a/tan</code></td>
<td>Trigonometry functions</td>
</tr>
<tr>
<td><code>abs sqrt</code></td>
<td>Absolute value or square root of a tensor</td>
</tr>
<tr>
<td><code>exp log log10</code></td>
<td>Power and logarithmic functions</td>
</tr>
<tr>
<td><code>ceil floor round fix</code></td>
<td>Rounding</td>
</tr>
<tr>
<td><code>mod rem</code></td>
<td>Modulo and reminder</td>
</tr>
<tr>
<td><code>conj</code></td>
<td>Complex conjugate</td>
</tr>
<tr>
<td><code>length size numel</code></td>
<td>Tensor dimension inspection.</td>
</tr>
<tr>
<td><code>min max sum mean</code></td>
<td>Min, max, sum or mean of the elements of a tensor along the first array dimension whose size does not equal 1.</td>
</tr>
<tr>
<td><code>zeros ones rand</code></td>
<td>Multi-dimensional tensor initializations with zeros, ones or random numbers</td>
</tr>
<tr>
<td><code>: colon</code></td>
<td>Unit-spaced vector initialization, also used for array subscripting, and for-loop iteration</td>
</tr>
<tr>
<td><code>horzcat vertcat</code></td>
<td>Horizontal and vertical tensor concatenation</td>
</tr>
</tbody>
</table>

Table 5.1: Subset of MATLAB supported in MGen
Tensor concatenation is supported through `horzcat` and `vertcat`. Dimension inspection is supported by `length`, `size` and `numel`.

The program control statements `if`, `switch`, `for`, `while` are fully supported, while `continue` and `break` are not. Handling exceptions is not supported in our generator. Indexing is supported for multi-dimensional arrays of any size, including linear and logical indexing, as well as indexing with multiple subscripts.

### 5.4.2 High-level Overview

The high-level overview of our prototypical generator MGen is illustrated at Figure 5.2. The input of MGen is a MATLAB or a Scala program, and output is a standalone C program. For example, a Fibonacci function can be given in the following form:

```matlab
function [ result ] = fib (n
    prev = 1;
    next = 1;
    for i = 1:n
        tmp2 = next;
        next = next + prev;
        prev = tmp2;
    end
    result = next;
end
```

```scala
// Fibonacci in eDSL in Scala
def fib (n: Rep[Shape]): Unit = {
    var prev = 1
    var next = 1
    loop(n, (i: Rep[Shape]) => {
        val tmp2 = next;
        next = next + prev
        prev = tmp2
    })
    next
}
```

The interface allows several MATLAB functions to be provided as input, and expects one of them to be the entry point, assuming that the other provided functions will be called from the entry function. For the Scala front end, several function can be provided as well, as long as they are inlined into the entry function.

Along with the provided functions as inputs, MGen also takes type and shape annotations that describe the input. It also include switches that can specify whether the generated code should perform runtime checks to validate the input against the type and shape annotations, and whether to validate dimension constraints for each operator. It also includes switches that control whether the generated code conforms to MATLAB type interaction (e.g., saturated arithmetic), or C-like type interaction. Furthermore, it also takes arguments that describe ISA for code generation.
The first step of the program is building the initial Math IR or M-IR, the high-level representation of a MATLAB program in LMS. Using the Scala-front end, the M-IR is build directly, while using the MATLAB frontend, we use McLab Core [118] that provides the Tame-IR representation, which is then translated into LMS.

Upon creation of the M-IR, LMS converts the program into SSA form, and then performs dead-code elimination and common subexpression elimination. Once complete, we initiate a set of analysis and transformation phases to infer program properties:

1. The first compiler phase is type inference.
2. Then we infer whether variables represent complex or real data.
3. For each operator we perform analysis and transformations to track the changes of number of dimension of tensors.
4. Once we know whether the number of dimensions is fixed or variable, we perform analysis and transformations on the dimension values to infer whether tensors grow or shrink within dimensions.

Once these phases are complete, we obtain an M-IR that provides constant or symbolic values describing the type and the shape of each variable. With these informations, we can lower the M-IR representation into a C-like intermediate representation. We perform this in several steps:

1. For each program variable, we create memory references that will be allocated in subsequent phases.
2. For each computation, we take the inferred types from the M-IR, and generate code that performs type specialization on all the type combinations of the variables involved in the computations. It is during this step that once type is known, we know the length of the primitive to complete memory allocations.
3. Once types are specialized for each tensor, we perform operator lowering to low-level C-IR.
4. If a vector ISAs is specified, each operator is vectorized, and an ISA-specialized code is generated.

The result of the generator is a self-contained C-code bundle that can include one or several functions.
Figure 5.2: MGen Design Overview
5.4.3 Building the M-IR

M-IR Overview The M-IR provides a high-level representation of MATLAB operations and includes LMS definitions for all supported operations in Table 5.1. Each definition corresponds to the name of the MATLAB function that it represents, and extends a ShapeDef interface:

```scala
abstract class ShapeDef extends Def[Shape] {
  val typedFunction : TypedFunction
  val complexFunction : ComplexFunction
  val dimLenFunction : LengthFunction
}
```

MATLAB is typeless and thus Exp[Shape] is used as the default type to represent all expressions in the M-IR. For the definition of each MATLAB function, we provide three functions:

```scala
abstract class TypedFunction {
  def apply(t: Seq[Type]): Option[Type]
}
abstract class ComplexFunction {
  def apply (operands: Seq[Boolean]): Option[Boolean]
}
abstract class LengthFunction {
  def apply (operands: Seq[Exp[Int]]): Option[Exp[Int]]
}
```

TypedFunction describes the type rules for each built-in function. For example, for an addition operator, the type rules are given as:

- \(\text{double} \times \text{double} \rightarrow \text{double}\)
- \(\text{float} \times \text{double} \rightarrow \text{float}\)
- \(\text{double} \times \text{int16} \rightarrow \text{int16}\)
- \(\ldots \times \ldots \rightarrow \ldots\)

When a given type combination is not handled, the function returns None. ComplexFunction has a similar responsibility, describing rules that determine whether we deal with real or complex data, and also returns None for an unsatisfiable combination.

LengthFunction is used to describe the change of the number of dimension in each supported functions. To illustrate, consider the case of a point-wise operations, applicable to addition, multiplications, subtraction,
division, etc. We either perform point-wise operations using one scalar and a tensor, or two tensors. An \( n \)-dimensional tensor can always be treated as an \( n' \)-dimensional tensor \((n' > n)\), assuming that all dimensions values from \( n \) to \( n' \) are ones. Thus, the resulting tensor of a point-wise operation can have up to the maximum dimension of the two tensors:

```java
object PointwiseDimensionLength extends LengthFunction {
  def apply (operands: Seq[Exp[Int]]): Option[Exp[Int]] = 
    Some(math.max(operands(0), operands(1)))
}
```

For all built-in operators, M-IR includes not only definition for the generic version of the operators, but also specialized definition, that correspond to overridden operators. For example, consider the addition operator:

```scala
// The generic pointwise addition
case class ShapePlus (a: Exp[Shape], b: Exp[Shape]) extends ShapeDef { ... }

// Scalar + Tensor
case class ShapePlus1(a: Exp[Shape], b: Exp[Shape]) extends ShapeDef { ... }

// Tensor + Tensor
case class ShapePlusN(a: Exp[Shape], b: Exp[Shape]) extends ShapeDef { ... }
```

ShapePlus represents a generic addition, while ShapePlus1 represents tensor addition with a scalar and ShapePlusN represents addition of two tensors with identical dimensions.

In addition to the functions in the MATLAB subset, M-IR provides definitions for several implicit functions. For example, an implicit function that extracts the first primitive from a tensor (used to build `for`-constructs), or another one to check whether a tensor is comprised of non-null values (used to build `if` and `while`-constructs). Literals are also represented with implicit definitions, allowing us to distinguish between integer, decimal, boolean and string literals.

MATLAB FRONTEND The MATLAB front-end, parses the MATLAB functions and then builds the M-IR for each function. MGen does not implement a MATLAB parser; instead, it uses McLab Core [118] as a parser only. It parses the MATLAB files using MetaLexer and JFlex as scanners, and Beaver as a parser. It builds an AST tree, and uses JustAdd for computations of AST attributes. The AST undertakes several analysis and transformation steps, where the last one is conversion to the so-called Tame-IR [98]. Then, Tame-IR is converted into M-IR. The conversion is done by traversing the Tamer-IR AST, converting each AST node into a corresponding LMS
definition. In particular, if, for and while-constructs are translated into corresponding LMS constructs, while all other constructs are part of the M-IR. Tensor reads and assignments are represented as LMS variable reads and assignments, while indexed read and index assignment are treated as special variable assignments and reads in the M-IR. The conversion in LMS, results in an automatic SSA conversion of the Tame-IR code, separating the immutable expression and variables, from the mutable variables.

It is important to note that McLab Core is not only a MATLAB parser, but a MATLAB language toolkit that provides an extensible intra-procedural and inter-procedural value analysis for estimating MATLAB types, shapes and call graphs. While the Tame-IR provides type and shape inference, the results of these analyses are not used by MGen. There are two main reason for this decision:

• The Tame-IR does not provide guarantees that the type inference provided is correct or complete, or whether it corresponds to any particular MATLAB version. For example, we observe that the analysis of Tame-IR \cite{98} reports that power function is not defined on operands which are double and float, respectively, which is not the case for MATLAB 2016a or later. Therefore, MGen must be capable of performing type inference.

• Shape propagation in the Tame-IR either results in constants, or abstract values to represent dimension of a tensor at a given point in the program. While constants can be used directly in LMS, the Tame-IR does not provide the shape propagation equations for the abstract values, nor their dependencies. Consequently, MGen must be able to perform shape propagation by staging the propagation equations.

**Scala Frontend** The Scala front-end uses the M-IR as an embedded DSL, and using it is in many ways similar to writing MATLAB code, as it does not require any explicit type or shape declaration. However, as a Scala eDSL, it applies the type of \texttt{Exp[Shape]} to all tensor variables and function arguments.

5.4.4 Type Inference

**Profiling Type Rules** MathWorks continuously extends the MATLAB language, adding new features, and extending already existing func-
tions [117]. When certain functions are extended, changes in the type interactions are possible, as well as increased coverage on dimension and input types. Creating a MATLAB generator by manually inspecting overridden operators is a tedious process, usually limited to a single MATLAB version. To avoid this, we developed a profiler in MATLAB, that probes built-in functions with different types and dimensions.

The profiler works such that for a given function, and its arity, it “brute-forces” all possible combination of types, considering both real and complex data. Then, it uses similar approach by trying different combination of matrix dimensions. If a built-in operator is not defined for a given type / shape combination, a MATLAB exception is raised. Therefore, each combination is wrapped in try-catch statements, and invalid combinations are ignored. Once the function executes properly, a profile is being generated. The profile is a Scala object that extends TypedFunction and includes all valid type combinations.

We generate such profiles for each supported function in MGen. The profiles are then added to the corresponding function definition in the M-IR. To illustrate, consider the profile for the addition operator:

```scala
object MTI_plus extends TypedFunction {
  def apply(t: Seq[Type]): Option[Type] = (t.head, t(1)) match {
    case (DoubleType, DoubleType) => Some(DoubleType)
    case (DoubleType, FloatType) => Some(FloatType)
    case (DoubleType, CharType) => Some(DoubleType)
    case (DoubleType, BooleanType) => Some(DoubleType)
    case (DoubleType, ByteType) => Some(ByteType)
    case (DoubleType, ShortType) => Some(ShortType)
    case (DoubleType, IntType) => Some(IntType)
    case (DoubleType, LongType) => Some(LongType)
    case (DoubleType, UByteType) => Some(UByteType)
    case (DoubleType, UShortType) => Some(UShortType)
    case (DoubleType, UIntType) => Some(UIntType)
    case (DoubleType, ULongType) => Some(ULongType)
    case (FloatType, DoubleType) => Some(FloatType)
    case (FloatType, BooleanType) => Some(FloatType)
    // ... all other type combinations are defined here
    case _ => None
  }
}
```

```scala
case class ShapePlus (a: Exp[Shape], b: Exp[Shape]) extends ShapeDef {
  val typedFunction = MTI_plus
  // ... the other functions get defined here
}
```
The profiler allows us to update the profiles when new versions of MATLAB are introduced or to retrospectively generate profiles for older MATLAB versions. The systematic approach allows us to guarantee that a type inference in MGen will indeed conform to the type interactions specification of a particular MATLAB version.

**Type Inference** MATLAB variables obtain their types once they are defined, and they can only change upon assignment. All elements of a tensor have identical types and an indexed assignment does not change the type of the variable being assigned. MATLAB does not support variable declarations and each variable is the result of a MATLAB expression that is either a built-in or a user defined function. Therefore, variables types are obtained through the return type of functions and can change upon assignment only. Consequently, in order to propagate and infer types, we can use data-flow analysis.

The type profiles described above, allow us to specify types rules, and thus we can create a *type definition* for each variable. This allows us to propagate types using analysis based on reaching definitions [119]. However, we need to adjust the classical literature approach to work with our system of type definitions.

LMS distinguishes between mutable and immutable variables. Each immutable variables has type of $\text{Exp[Shape]}$, while each mutable variable has $\text{Var[Shape]}$. Due to SSA, each $\text{Exp[Shape]}$ variable has only one definition, while $\text{Var[Shape]}$ can be assigned and read multiple times. This allows us to create a type definition for each immutable variable, and to propagate these definitions when a mutable variable is read or assigned.

LMS provides a structured graph representation of IRs, such that it constructs a list of topologically sorted statements. Constructs such as `if`, `for` and `while` contain blocks that are also lists of topologically sorted statements. Consequently, we can perform our analysis by traversing statements, such that each has a predecessor defined by the topological sort. For `for` and `while` loops, each statement at the beginning of the block will also include the last statement in the block as a predecessor.

Now, we define the notations and equations required for reaching definitions analysis:

- $\text{REACH}_{in}(stm)$: set of definitions reaching a given statement $stm$.
- $\text{REACH}_{out}(stm)$: set of definitions reaching after a given statement $stm$ is executed.
• \textit{GEN}(stm): set of definitions which appear in \textit{stm} and may reach statements after the execution of \textit{stm}.

• \textit{KILL}(stm): set of definitions (in the whole program scope being analyzed) whose variables are definitely redefined in \textit{stm}.

The basic equations to perform the analysis are then given as:

\[
\text{REACH}_{\text{out}}(stm) \leftarrow \text{GEN}(stm) \cup (\text{REACH}_{\text{in}}(stm) - \text{KILL}(stm))
\]

\[
\text{REACH}_{\text{in}}(stm) \leftarrow \bigcup_{s \in \text{predecessor}(stm)} \text{REACH}_{\text{out}}(s)
\]

We maintain these sets for each statement in the M-IR. To calculate \text{GEN}(stm), where \textit{stm} corresponds to a MATLAB numerical function, we need to keep track of possible type which result of executing the same function. To achieve that, for each expression of type \text{Exp}[\text{Shape}] in the M-IR, a metacontainer is created:

```scala
// map each metacontainer reference to its metacontainer
var shapeExpMap = Map.empty[Exp[Shape], Shape]

class Shape (val exp: Exp[Shape] /* metacontainer reference */) {
    shapeExpMap += (exp -> this) // add the metacontainer in the map
    var pType = Set.empty[Type] // possible types
    var pComplex = Set.empty[Boolean] // possible complex / real data
}

object Shape {
    def apply(exp: Exp[Shape]) = shapeExpMap(exp)
}
```

The meta-container includes a parameter \text{pType} that keeps track of all possible types that a particular tensor can take at the given point in the program. Initially, for each variable in the M-IR, \text{pType} is set to be empty, except for the variables that correspond to the input arguments of the program. These variables obtain \text{pType} values that are specified in the type and shape annotations as input in MGen.

To execute our type propagation analysis, we perform a forward pass on the M-IR. A new type can be defined for an immutable variable if a new combination of types is applied to the definition of the variable. Therefore, we generate all possible combination, and we apply the type rules defined as \text{TypedFunction} in each ShapeDef. To illustrate consider:

```scala
trait ShapeInferType extends NestedBlockTraversal {
    val IR: MIR; import IR._
}
```
protected var isFixedPoint = true
var REACH_in : Map[Stm, Set[Tuple[Exp[Any], Type]]] = ...
var REACH_out: Map[Stm, Set[Tuple[Exp[Any], Type]]] = ...
var KILL : Map[Stm, Set[Tuple[Exp[Any], Type]]] = ...
var GEN : Map[Stm, Set[Tuple[Exp[Any], Type]]] = ...
private def combine[T](x: List[List[T]]): List[List[T]] = x
  match {
    case Nil => List(Nil)
    case h :: _ => h.flatMap(i => combine(x.tail).map(i :: _))
  }
def applyTypeRules(operandTypes: List[List[Type]], f: TypedFunction) = {
  val combinations = combine(operandTypes)
  val returnTypes = combinations.flatMap(f.apply).toSet
  returnTypes // ... also raise an exception in case returnTypes.isEmpty
}
override def traverseStm(stm: Stm): Unit = stm
  match {
    case TP(s, d: ShapeDef) => {
      val operandTypes = syms(d).map(o => Shape(o).pType.toList)
      val returnTypes = applyTypeRules(operandTypes, d.typedFunction)
      val union = Shape(s).pType union returnTypes
      isFixedPoint = Shape(s).pType == union
      Shape(s).pType = union
      // Update GEN(stm) and REACH_out(stm)
    }
    case TP(s, Reflect(Assign(Variable(a), v), _, _)) => { ... }
    case TP(s, Reflect(ReadVar(Variable(a)), _, _)) => { ... }
  }

Consequently, $GEN(stm)$ and $KILL(stm)$ are defined as:

$$KILL(stm) = \{(s\_type, t) \mid t \in \text{set of all 12 primitives}\},$$
$$GEN(stm) = \{(s\_type, t) \mid t \in p\_Type\},$$

where $s\_type$ is the type of an immutable variable $s$. For each statement that performs a mutable variable assignment $Assign(Variable(v), s)$, we set:

$$KILL(stm) = \{(v\_type, t) \mid t \in \text{set of all 12 primitives}\},$$
$$GEN(stm) = \{(v\_type, t) \mid t \in \{u \mid (s\_type, u) \in REACH\_in(stm)\}\},$$

where $v\_type$ is the type of the mutable variable $v$ and $s\_type$ is the type of immutable variable $s$. For each statement $s = ReadVar(Variable(v))$, that denotes read on a mutable variable $v$ we set:

$$KILL(stm) = \{(s\_type, t) \mid t \in \text{set of all 12 primitives}\},$$
$$GEN(stm) = \{(s\_type, t) \mid t \in \{u \mid (v\_type, u) \in REACH\_in(stm)\}\},$$
where \( s_{type} \) is the type of the immutable variable \( s \), and \( v_{type} \) is the type of the mutable variable \( v \). Note that every time we read from a mutable variable \( v \), using \( s = \text{ReadVar}(\text{Variable}(v)) \), we produce an immutable variable \( s \) and therefore we need to update the \( pType \) parameter in the metacontainer of \( s \). Therefore we have:

\[
\text{Shape}(s).pType = \{ t \mid (s_{type}, t) \in \text{REACH}_{out}(stm). \}
\]

Our analysis is iterative and upon each iteration it propagates types in the \( pType \) sets for each variables. We keep track of the changes, of the \( \text{REACH}_{in}, \text{REACH}_{out}, \text{GEN}, \text{KILL} \) and \( pType \) sets, and repeat the analysis until a fixed point is reached. Once complete, the metacontainers contain all possible types for a given variable.

**INFERRING REAL OR COMPLEX DATA**

To infer whether we are dealing with real or complex data, which we refer to as the complex type of a tensor, we use exactly the same approach for type inference. Instead of \( \text{typedFunction} \), we use \( \text{complexFunction} \) to encode complex type rules. Instead of \( pType \), we use \( pComplex \) to keep track of possible complex types for a given variable. We perform the same type of analysis based on reaching definitions to propagate the complex types in each metacontainer and iterate until fixed point is reached. Once complete, the metacontainer will contain all possible complex types for a given variable. The inference of complex types runs in a separate phase, after type inference completes.

To illustrate the two analyses, consider the Fibonacci program given in Section 5.4.2. After converting into M-IR, it will have the following form:

```plaintext
fib (m_x6) {
 m_x8 = Const(1);
 var131 = m_x8;
 var132 = m_x8;
 i14 = getFirstPrimitive(m_x6);
 i15 = i14 + 1;
 for (i13 = 1; i13 < i15; i13 = i13 + (1)) {
   m_x133 = var132;
   m_x134 = var131;
   m_x135 = m_x133 + m_x134;
   var132 = m_x135;
   var131 = m_x133;
 }
 m_x140 = var132;
 return m_x140;
}
```
We set-up the input argument to the program to have an integer type. Once both type inference and complex type inference phases are complete, the resulting program will infer the type and the complex type of each variable:

```java
fib (m_x6) {
    m_x8[Double] [R] = Const(1);
    var131[Double] [R] = m_x8[Double] [R];
    var132[Double] [R] = m_x8[Double] [R];
    i14[Int] [R] = getFirstPrimitive(m_x6[Int] [R]);
    i15[Int] [R] = i14 + 1;
    for (i13[Int] [R] = 1; i13 < i15; i13 = i13 + (1)) {
        m_x133[Double] [R] = var132[Double] [R];
        m_x134[Double] [R] = var131[Double] [R];
        m_x135[Double] [R] = m_x133[Double] [R] + m_x134[Double] [R];
        var132[Double] [R] = m_x135[Double] [R];
        var131[Double] [R] = m_x133[Double] [R];
    }
    m_x140[Double] [R] = var132[Double] [R];
    return m_x140[Double] [R];
}
```

Our analysis successfully infers the return type of the program and the types of all local variables.

5.4.5 Shape Inference

MATLAB variables obtain their shape as the result of a MATLAB function, and can only change upon assignment or indexed assignment. In each case, dimension values can change or the number of dimensions can grow or both. Therefore, we perform shape inference in two steps. First, we infer the number of dimensions for each tensor variable. Then, we infer the dimension values.

Infer number of dimensions To infer the number of dimensions, we start by extending the metacontainer with another parameter: pLen, that will represent the number of dimensions for a given tensor.

```java
class Shape (val exp: Exp[Shape] /* metacointainer reference */) {
    shapeExpMap += (exp -> this) // add the metacontainer in the map
    var pType = Set.empty[Type] // possible types
    var pComplex = Set.empty[Boolean] // possible complex / real data
```
For this part, we also create a metacontainer for each mutable tensor variable \( \text{Var}[\text{Shape}] \), having \( \text{pLen} \) as a mutable variable.

```scala
5  var pLen = Option.empty[Exp[Int]] // number of dimensions
6 }
```

Initially, only the input arguments of the program have values set for \( \text{pLen} \), while others are set to \text{None}. The first step is lowering the dimensions length, by inserting code into the M-IR that reasons about the number of dimensions. LMS uses a sea-of-nodes representation; thus, inserting code into the M-IR means creating new nodes. We perform the transformations by traversing forward through the M-IR, creating the new nodes along the way. For this purpose, we use the \text{LengthFunction}:

```scala
1 trait LowerDimensionsLength extends ForwardTransformer {
2  val IR: MIR; import IR._
3  override def transformStm(stm: Stm): Exp[Any] = stm match {
4   case TP(_, Reflect(Assign(Variable(v), s), _, _)) => {
5     val (v_len, s_len) = (ShapeVar(v).pLen.get, Shape(s).pLen.get)
6     var_assign(apply(v_len), apply(s_len))
7   }
8   case TP(s, Reflect(ReadVar(Variable(v)), _, _)) => {
9     val v_len = ShapeVar(v).pLen.get
10    Shape(s).pLen = Some(readVar(apply(v_len)))
11   }
12   case TP(s, d: ShapeDef) => {
13     val args = syms(d).flatMap(o => apply(Shape(o).pLen.get))
14     d.dimLenFunction(args) map {
15       case len@Some(_) => Shape(s).pLen = len
16       case _ => // raise an exception
17     }
18   } // rest of the cases implemented here ...
Figure 5.3: Transformation of the initial M-IR Fibonacci program using metacontainers. In this transformation we add definitions for the number of dimensions in the metacontainer.
As we traverse forward, we start from the input parameters, and propagate the definitions for number of dimension across all M-IR definitions. The propagation is done as follows:

- Each definition of a MATLAB function, will also generate definitions that represents the number of dimensions of the resulting tensor. These definitions are a result of applying the `LengthFunction` on the number of dimensions of the function arguments. This generates an immutable variable, stored in the `pLen` parameter of the metacontainer of the resulting tensor.

- Each assignment of a mutable tensor generates an assignment on the number of dimension of the mutable tensor.

- Each time a mutable tensor variable is read, a read on the number of dimensions of the mutable tensor is performed.

Once this process completes, we have successfully lowered the number of dimensions. To illustrate, consider the Fibonacci program in Section 5.4.2. The lowering is illustrated in Figure 5.3 such that for every node in the initial M-IR (above), we create a metacontainer that contains the parameters for the number of dimensions (below). Effectively, this process creates new definitions that are overlaid on top the initial M-IR.

Unfortunately, adding new nodes into an LMS IR does not make them automatically part of the IR itself. Namely, there are no dependencies between the nodes that represent the number of dimensions and the existing sea-of-nodes that represents the rest of program. While the new nodes can be accessible through the metacontainers, LMS will discard them during IR traversals. However, we can work around this issue. LMS allows us to override behaviour for *anti-dependency*. This relationship establishes a “must not after” scheduling of nodes, which fits in our use case, as we want to schedule the reasoning for number of dimensions of a given computation, before the computation occurs. To achieve that, we add another method in the metacontainer `deps` and define the virtual dependencies:

```java
class Shape (val exp: Exp[Shape] /* metacontainer reference */) {
    shapeExpMap += (exp -> this) // add the metacontainer in the map
    var pType = Set.empty[Type] // possible types
    var pComplex = Set.empty[Boolean] // possible complex / real data
    var pLen = Option.empty[Exp[Int]] // number of dimensions
    def deps () = pLen.toList // define metacontainer dependencies
}
// override LMS methods for "soft symbols" i.e. anti-dependencies
```
Figure 5.4: Inferring number of dimensions effectively means analyzing integer operations which are overlaid on top of the M-IR

All functions that implement LengthFunction are built using the available primitive integer operations, which are natively supported by LMS. This gives us extensibility, in supporting any MATLAB functions, and at the same time, due to its low-level representation, allows us to perform analysis and transformations using the optimizations already available in LMS. In this respect, we extend LMS with support of data-flow analysis based on constant propagation [120]. Our analysis does not have to be adapted to reason about the existence of M-IR specific definitions and operators, and can only focus on the overlaid nodes which deal with integer operations only, as illustrated in Figure 5.4.

The result of the analysis is a mapping of mutable variables to immutable variables or constants. Then, for each immutable variable, we perform substitution with the respective immutable variable or constant, effectively eliminating the mutable variable from the M-IR. When substitutions of variables are executed, particularly with constants, all expressions that involve them will start a new round of partial evaluation and dead-code elimination, which are already provided by LMS.

However, substituting variables in the M-IR does not automatically substitute them in the respective metacontainer. Nevertheless, we can work around this issue. Transformations in LMS are done through iterated stag-
ing, described in Section 1.3.2, and each transformer maintains a mapping from old symbols to new ones. We can benefit from it and apply substitution in each metacontainer. For that purpose, we extend the metacontainer with a transformation method, that takes a transformer as an argument, and applies substitutions within the metacontainer. To illustrate this process, consider:

```scala
class Shape (val exp: Exp[Shape] /* metacontainer reference */) {
    shapeExpMap += (exp -> this) // add the metacontainer in the map
    var pType = Set.empty[Type] // possible types
    var pComplex = Set.empty[Boolean] // possible complex / real data
    var pLen = Option.empty[Exp[Int]] // number of dimensions
    def deps () = pLen.toList // define metacontainer dependencies
    def transform (f: Transformer) = pLen
        match {
        case Some(v) => pLen = Some(f(v)) // apply substitutions
        case _ =>
        }
}
```

After transform is executed on all metacontainers, pLen is substituted with another symbol or a constant. Consequently, dependencies in deps are automatically restored. Finally, we obtain an M-IR that holds all possible types and complex types of a variable, as well as the number of dimensions at each point in the program. We can now proceed do the next step, i.e., inferring dimension values.

**Infer Dimension Values** To infer dimension values, we need an abstraction to reason about structures that can represent dimensions. Dimensions can grow in length and can change their respective values. Therefore, the simplest form to abstract dimensions is to build an array-like structure. In MGen, we use metacontainers to implement it. The abstract interface of the metacontainer is given as:

```scala
abstract class DArray {
    def length (): Exp[Int]
    def apply (i: Exp[Int]) : Exp[Int]
    def update (i: Exp[Int], v: Exp[Int]): Unit
    def deps () : List[Sym[Any]]
    def transform (f: Transformer): DArray
}
```

Similarly to the Shape metacontainer used to represent tensor variables, we provide a method to return all dependencies of a DArray metacontainer.
We also provide a transformation method, but unlike the Shape metacontainers, transformations in DArray return a new instances of DArray. As metacontainers support nested objects, we can include DArray in the Shape metacontainer of each tensor. To ensure compatibility with the existing metacontainers, we adjust the deps and transform methods to interface with DArray:

```scala
class Shape (val exp: Exp[Shape] /* metacontainer reference */) {
  shapeExpMap += (exp -> this) // add the metacontainer in the map
  var pType = Set.empty[Type] // possible types
  var pComplex = Set.empty[Boolean] // possible complex / real data
  var pLen = Option.empty[Exp[Int]] // number of dimensions
  def deps () =
    pLen.toList ::: (dims.toList.flatMap(_.deps())) // deps from both
  def transform (f: Transformer) = {
    if (pLen.nonEmpty) pLen = Some(f(pLen.get))
    if (dims.nonEmpty) dims = Some(pLen.get.transform(f))
  }
}

abstract class DArrayVar {
  def read () : DArray
  def assign (a: DArray) : Unit
  def deps (): List[Sym[Any]]
  def transform (f: Transformer): DArray
}

The abstraction above, allows us to specialize the dimension array, particularly if we know that the number of dimensions is fixed. This can be particularly useful, as we already infer the number of dimensions. Therefore, similarly to the abstractions shown in Section 4.3, we can create a metacontainer that represents an array of staged dimension values:

```scala
case class DArrayScalar(val dims: Array[Exp[Int]]) extends DArray {
  def length () : Exp[Int] = Const(dims.length)
  def apply (idx: Exp[Int]) : Exp[Int] = idx match {
    case Const(c) => if (c < dims.length) dims(c) else Const(1)
  }
  def update (idx: Exp[Int], v: Exp[Int]) : Unit = idx match {
    case Const(c) => dims(c) = v
  }
  protected def transform(f: Transformer) : DArray = {
    new DArrayScalar(dims.map(v => f(v))
  }
```
However, once we have a symbolic length of dimensions, the dimension array must be built using staged arrays:

```scala
case class DArrayStaged(n: Exp[Int], arr: Exp[Array[Int]]) extends DArray {
  def length (): Exp[Int] = n
  def apply (i: Exp[Int]): Exp[Int] = arr(i)
  def update (i: Exp[Int], v: Exp[Int]) = { arr(i) = v }
  def deps (): List[Sym[Any]] = List(n, arr) flatMap {
    case x: Sym[_] => Some(x)
    case _ => None
  }
  protected def transform (f: Transformer) : DArray = {
    createDArray(f(n), f(arr))
  }
}
```

The abstraction provided by DArray allows us to create method for dimension inspection, in particular, to check whether a given DArray is a matrix, vector or scalar. Depending on the type of the instance of the DArray, these checks will either be resolved during staging, or the code that inspects the dimensions will propagate in the runtime of the generated code.

Having defined the abstraction to stage the dimensions of tensors, we can proceed to lower dimension values. The lowering works in the same way as the lowering of the number of dimensions. Initially, each tensor variable has an empty dims parameter, with the exception of the input arguments. We traverse the M-IR forward, and create new DArray instances for each supported operator, essentially creating new nodes in the M-IR. The propagation works as follows:

- For each definition of MATLAB function, we extract the DArray instances of the input arguments. We create definitions that inspect their values, and determine whether they satisfy the shape constraint for the given MATLAB function (for example matrix-transpose is only defined on two-dimensional tensors). Then we generate a new DArray instance that represents the dimension of the resulting tensor. For each of these operations, we opportunistically use the previously inferred
dimensions length, specializing each newly created DArray instance. The new instance is stored in the dims parameter of the metacontainer of the resulting tensor.

- Each assignment of a mutable tensor, generates an assignment on its DArrayVar parameter. If the pLen argument is inferred to be a constant, DArrayVar is specialized as an array of mutable variables. The assignment results in copying the dimensions values from DArray to DArrayVar.

- Each time a mutable tensor variable is read, a read on the DArrayVar parameter of the mutable tensor is performed, producing a DArray instance. If the pLen argument is inferred to be a constant, then the resulting DArray is specialized. The procedure results in copying the dimensions values from DArrayVar to DArray.

Apart from the fact that a built-in function has dimension constraints for its arguments, some built-in operators have multiple implementations depending on the shape of the arguments. For example the addition operator is valid for adding scalar and tensor of any dimension, or two tensors with identical values. Therefore, while lowering the dimensions, we can also separate the two overridden implementations.

```scala
trait LowerDimensions extends Forw1dTransformer {
  val IR: MIR; import IR._
  val f: Transformer = this
  override def transformStm(stm: Stm): Exp[1n] = stm match {
    case TP(s, dShapePlus(a, b)) => {
      val (fa, fb) = (apply(a), apply(b))
      val (sa, sb) = (Shape(fa), Shape(fb), Shape(s))
      val (da, db) = (sa.dims.get.transform(f), sb.dims.get.transform(f))
      val pLen = Shape(s).pLen.get.transform(f)
      var shape: Shape = null
      (da.isScalar(), db.isScalar()) match {
        case (Const(true), _) =>
          shape = Shape(shape _plus_1(fb, fa))
          shape.dims = Some(db)
        case (_, Const(true)) =>
          val shape = Shape(shape _plus_1(fa, fb))
          shape.dims = Some(da)
        case _ =>
          assert(check_for_equal_dimensions(da, db, pLen))
      }
      val dims = create_darray_for_equal_dimensions(da, db, pLen)
      val shape = Shape(shape _plus_n(fa, fb))
      shape.dims = Some(dims)
    }
  }
}
During the lowering process, we check whether a given tensor is a scalar. If that is the case for at least one operand, we replace the ShapePlus definition with a ShapePlus1, that has a scalar as a second operand. If both tensors are not scalar, then we generate code that checks the equality of dimensions of the two arrays. If the dimensions are not equal, then the operation results in reporting an error. For each case we create a new metacontainer, disposing the old one, and we propagate the dimension in the resulting metacontainer.

Note that the code above is a simplification of the lowering phase. MGen will attempt to statically prove during staging whether a tensor is a scalar or not. This involves opportunistically selecting the inferred information from both dimension arrays, as well as the inferred number of dimensions to reduce iterations over the dimension arrays. Only when this process fails, it will generate nested if-constructs that perform these checks during the runtime of the generated code.

When the lowering phase completes, we perform another round of analysis based on constant propagation. We use the analysis to eliminate mutable variables through substitution. The transformer is then applied to all metacontainers involved, which performs substitutions in the Shape metacontainer and in the DArray metacontainer as well. When substitutions in DArray instances occur, a staged DArray can also be specialized into a scalar DArray. In that particular case, the old DArray metacontainer is discarded, and a new one is generated.

To illustrate the process, consider the Fibonacci program in Section 5.4.4. After the type inference phase completes, we infer the number of dimensions, and then we lower the dimension values. The state of the M-IR, after the completion of these phases, is illustrated in Figure 5.5. We can observe that there are no staged arrays in the code, which means that the phase of inferring the number of dimensions has resolved all tensors to have a constant number of dimensions (i.e., no growth of dimensions is detected). We can also observe how the different overridden versions for the addition operator are already separated (lines 33 – 43) into individual ones. But at the same time we can also observe that code is generated for reasoning about

```c
shape.pType = Shape(s).pType
shape.pComplex = Shape(s).pComplex
shape.exp
}
// all other supported operators
}
```
dimension values, even though all variables involved in this computation are in fact scalars.

Once constant propagation is executed, the analysis suggests that variables `var155`, `var153`, `var163` can be replaced by a constant value of 1. Performing the substitution will replace `i157` with constant of 1 (line 8). Then LMS provided partial evaluation will evaluate `x161` to true (line 11). This will eventually result in LMS taking only the “then” branch at line 12 and 32, performing dead-code elimination to the rest of the code. Finally, after all optimizations are completed, the resulting M-IR will have the following form:\(^3\):

```
1 fib (m_x6) {
2     m_x8[Double][R][1x1] = Const(1);
3     var203[Double][R][1x1] = m_x8[Double][R][1x1];
4     var204[Double][R][1x1] = m_x8[Double][R][1x1];
5     int32_t i14 = getFirstPrimitive(m_x6[Int][R][2]);
6     int32_t i15 = i14 + 1;
7     for (int32_t i13 = 1; i13 < i15; i13 = i13 + (1)) {
8         m_x205[Double][R][1x1] = var204[Double][R][1x1];
9         m_x206[Double][R][1x1] = var203[Double][R][1x1];
10        m_x212[Double][R][1x1] = ShapePlus1(m_x206[Double][R][1x1], m_x205[Double][R][1x1]);
11        var204[Double][R][1x1] = m_x212[Double][R][1x1];
12        var203[Double][R][1x1] = m_x205[Double][R][1x1];
13    }
14    m_x221[Double][R][1x1] = var204[Double][R][1x1];
15    return m_x221[Double][R][1x1];
16 }
```

We can conclude that once the last phase is completed, MGen will infer the shapes of all variables in the Fibonacci program.

The description of the shape inference phase shows us the properties of the powerful abstraction provided by metacontainers:

- The use of metacontainers does indeed simplify the M-IR. We can easily “inject” code into the M-IR to reason about dimension values and number of dimensions, without changing the initial M-IR definitions. Consequently, the analysis becomes simpler, allowing us to perform constant propagation on integer operations without considering the relationship between a tensor and its number of dimensions.

\(^3\) Note that as consequence of using transformations based on iterative staging in LMS, new symbols are generated in each transformation step. Consequently, variable names change through transformation phases, which is also observed in the generated code.
fib (m.x6) {
    m_x8[Double] [R][1] = Const(1);
    int32_t var153 = 1; var154[Double] [R][1] = m_x8[Double] [R][1];
    int32_t var155 = 1; var156[Double] [R][1] = m_x8[Double] [R][1];
    int32_t i14 = getFirstPrimitive(m_x6[Int] [R][2]);
    int32_t i15 = i14 + 1;
    for (int32_t i13 = 1; i13 < i15; i13 = i13 + (1)) {
        int32_t i157 = var155; m_x158[Double] [R][1] = var156[Double] [R][1];
        int32_t i159 = var153; m_x160[Double] [R][1] = var154[Double] [R][1];
        int32_t var163 = 1;
        if (i157 == 1) {
            var163 = i159; x175 = false;
        } else {
            bool x161, x162 = i159 == 1;
            if (x162) {
                var163 = i157; x173 = true;
            } else {
                bool x160 = i157 == i159;
                bool x171;
                if (x160) {
                    var163 = i157; x171 = true;
                } else {
                    x171 = false;
                }
                x173 = x171;
            }
            x175 = x173;
        }
        int32_t i176 = var163;
        var177[Double] [R][1] = m_null;
        if (x161) {
            m_x180[Double] [R][1] = ShapePlus1(m_x160[Double] [R][1], m_x158[Double] [R][1]);
            var177[Double] [R][1] = m_x180[Double] [R][1];
        } else {
            bool x162 = i159 == 1;
            if (x162) {
                m_x183[Double] [R][1] = ShapePlus1(m_x158[Double] [R][1], m_x160[Double] [R][1]);
                var177[Double] [R][1] = m_x183[Double] [R][1];
            } else {
                m_x186[Double] [R][1] = ShapePlusN(m_x158[Double] [R][1], m_x160[Double] [R][1]);
                var177[Double] [R][1] = m_x186[Double] [R][1];
            }
        }
        m_x192[Double] [R][1] = var177[Double] [R][1];
        var155 = i176; var156[Double] [R][1] = m_x192[Double] [R][1];
        var153 = i157; var154[Double] [R][1] = m_x158[Double] [R][1];
    }
    m_x200[Double] [R][1] = var156[Double] [R][1];
    return m_x200[Double] [R][1];
}
• Transformations become simpler. Unlike using a dedicated rewrite engine, as shown in Section 4.2.2, rewrites in MGen use transformers based on substitution, that will first modify the M-IR, and then apply the substitutions in each metacontainer, restoring the relationship with the substituted symbols.

• Metacontainers can be specialized or disposed. The DArray metacontainer reflects this property directly, as we can specialize the dimension representation for a fixed or variable number of dimensions, and we can dispose old DArray instances after transformation steps.

At the end of the shape inference phase, we have inferred the shape and the type of each tensor. With that information, the next phase is lowering the computation into a C-like representation.

5.4.6 Conversion to C-IR

C INTERFACE To be able to generate C code, we need a way to represent tensors in the language. For that purpose we define an enumeration to represent types, and a struct to represent tensors. Complex type is represented as a boolean value, with complex and real numbers represented as true and false, respectively. Dimensions are represented as an array, including a number to represent the length of the array.

```c
typedef enum {
    BooleanType = 0, ByteType = 1, CharType = 2, DoubleType = 3,
    FloatType = 4, IntType = 5, LongType = 6, ShortType = 7,
    UByteType = 8, UIntType = 9, ULongType = 10, UShortType = 11
} type_t;

typedef struct {
    void* data; // values stored in the tensor
    type_t td; // type descriptor of the tensor
    bool cd; // complex type descriptor
    int32_t* ld; // dimension values as dimension array
    int32_t ld_len; // length of the dimension array
} shape_t;
```

We use the representation as an interface for the generated code, applied to the input and output argument of the generated C-functions. To illustrate, consider the Fibonacci program in Section 5.4.2. Translation into a C-function will have the following form:
Both input and output of the functions are arguments to the C function, allowing us to define multiple inputs and outputs. The only difference between the two is that MGen expects the outputs to be defined as pointers.

**Mapping M-IR Metacontainers to C-IR**

To start the conversion, we traverse the M-IR forward and map each Shape metacontainer into a CShape metacontainer. CShape resembles Shape, but differs in important ways. Instead of maintaining a set of possible types for a tensor, it includes a type descriptor td and a complex type descriptor cd that keep track of the current type and complex type of the tensor at a particular location in the C-IR. It also includes a staged variable that represents the array responsible for the tensor data. As MATLAB matrices can take arbitrary size, we represent the tensor data as a HeapArray of any type, which (as the name suggests) represents an array allocated on the heap.

The interface of CShape is defined as follows:

```java
abstract class CShape(shapeMIR: IR.Shape) {
    protected val id: C.Exp[C.HeapArray[Any]]  // tensor data
    protected val td: C.Var>Type]            // tensor type
    protected val cd: C.Var[Boolean]          // tensor complex type
    protected val ld: DArray                  // tensor dimensions
    def allocate (): C.Rep[Unit]              // allocate memory
    def terminate (): C.Rep[Unit]             // free memory
    def setComplex (c: C.Exp[Boolean]): C.Rep[Unit]
    def getComplex (): C.Rep[Boolean]
    def setType (t: C.Exp>Type]: C.Rep[Unit]
    def getType (): C.Exp>Type]
    def getCArray (tp: Type, isComplex: Boolean): CArray
}
```

The ld parameter represents a DArray instance, which is a 1-to-1 translation of M-IR Shape.dims metacontainer. The metadata in this metacontainer provides us with all necessary information to allocate tensor data. For this purpose we include a method allocate to govern heap allocations, and terminate to relinquish them.

CShape can be specialized. We perform specialization for the input and output CShape to interface with the C-struct of shape_t. However, for all other local tensors, we use the CShapeIntermediate metacontainer, which keeps all internal data “unboxed” to the C-IR.
In the specialized versions, `CShapeInput` does not perform any allocations, and assumes that the data provided is already allocated. Also does not set types, or complex types or dimensions, as they are already supplied. On the other hand, `CShapeOutput` does not free memory, but it allocates a `shape_t` structure for the output. `CShapeIntermediate` can also be further specialized into `CShapeScalar` if the tensor that it represents is a scalar.

For every `ShapeVar` metacontainer, we create a `CShapeVar` metacontainer:

```java
abstract class CShapeVar(shapeMIR: IR.ShapeVar) {
  protected val id: C.Exp[C.HeapArray[Any]] // tensor data
  protected val td: C.Var[Type] // tensor type
  protected val cd: C.Var[Boolean] // tensor complex type
  protected val ld: DArrayVar // tensor dimensions
  def assign (x: CShape): C.Rep[Unit] // copy data CShape -> CShapeVar
  def read (x: CShape): C.Rep[Unit] // copy data CShapeVar -> CShape
  def terminate (): C.Rep[Unit] // free memory
}
```

This metacontainer can only be assigned and read by a `CShape` instance. The two methods will generate definitions in the C-IR that will perform data copy from the `CShape` memory region to the `CShapeVar` memory region and vice versa. During assignment, the memory region in `CShapeVar` is freed, and new memory region is allocated using `malloc`, before the data is copied.

Similarly to `CShape`, `CShapeVar` can also be further specialized into a `CShapeVarScalar` assuming the mutable tensor gets assigned with a scalar tensor across the whole program.

Finally, the mapping between M-IR metacontainers and C-IR metacontainers is governed by simple Scala `HashMap` instances:

```java
var mapCShape = HashMap.empty[IR.Shape , CShape]
var mapCShapeVar = HashMap.empty[IR.ShapeVar, CShapeVar]
```

**Memory allocations** When lowering a MATLAB computation to C-IR, we need to allocate memory regions to store the results of the computation. When assigning and reading tensor variables in the M-IR, we also need to allocate and copy tensor data in the C-IR. We distinguish
four separate cases in MGen, when dealing with memory allocations and memory copies and describe them next.

**Mutable tensor in the M-IR is either assigned or read.** This is relatively straightforward, as we use the CShape and CShapeVar metacontainers for this purpose. Upon assignment and read, the type and the complex type of a CShape metacontainer will be assigned to the CShapeVar metacontainer or vice versa. The translation in C-IR is given with:

```scala
trait MIR2CIR extends NestedBlockTraversal {

  val IR: MIR
  val C: MCIR

  override def traverseStm (stm: IR.Stm): Unit = stm match {
    case IR.TP(s, IR.Reflect(IR.ReadVar(IR.Variable(v)), _, _)) => {
      val shapeVar = mapCShapeVar(v)
      val shapeExp = mapCShape(s)
      shapeVar.read(shapeExp)
    }
    case IR.TP(_, IR.Reflect(IR.Assign(IR.Variable(v), exp), _, _)) => {
      val shapeVar = mapCShapeVar(v)
      val shapeExp = matCShape(exp)
      shapeVar.assign(shapeExp)
    }
    // all other cases are implemented here
  }
}
```

Both read and assign pattern match on the CShape to distinguish specialized metacontainers and generate code accordingly. For example, if an assignment is performed with a CShapeScalar, a single variable is updated in the memory region. If CShapeIntermediate is used, a copy of the region is performed using memcpy.

**Indexed read.** When an indexed read is performed the type remains the same for both tensors, the source and the destination. The description of the implementation of three different groups is given below.

- Logical indexed reads do not require bound checks. We perform the lowering in two steps. First, we create an intermediate array of the same size as the source array that performs the boolean expression specified by the logical index. Then we create the destination array that represents a vector of size equal to the number of elements in the intermediate array that evaluate to true. Then we iterate the source array linearly and copy each element into the destination array, if the corresponding element from the intermediate array is set to true.
• Indexing with a single subscript requires bound checks. We iterate through all elements specified in the tensor subscript, maintaining min and max values, and for each we evaluate whether the value specified is an integer or not. If the min value is less than 1 or the max value is greater than the linear length of the source tensor, or one element is not an integer, we generate code that raises an exception. Otherwise, we generate a destination tensor with the same shape as the subscript. Then for each element of the tensor subscript, we perform a copy that corresponds to the linear index of the source tensor.

• Indexing with multiple subscript requires bound checks. The first step checks whether the number of subscripts corresponds to the number of dimensions of the source tensor. If this is the case, we linearize each tensor subscript. Similarly as indexing with a single subscript, we check for integer values, and scan for min and max values. We perform this operation for all subscripts. Then we allocate the destination tensor. It will have the same number of dimensions as the source tensor, however the dimension values for each dimension will correspond to the linear size of the corresponding tensor subscript. Finally for all subscripts we create nested blocks of for-loops, that iterate the each linearized subscript tensor, performing copies in the destination tensor.

Index assignment. In an indexed assignment, destination tensors and the source tensors must have equal types. In general, indexed assignments follow the same patterns of indexing as indexed reads, but the main difference is that the copies flow in the other direction, and that they can change the dimensions of the array.

Logical indexed assignment does not change the dimensions of the destination tensor. Indexing with a single subscript can only change dimension to a one dimensional tensor. And index with multiple subscripts can change the number of dimensions and the dimension values of the destination tensor. In both cases, we adopt the following principle:

• First we perform checks to detect dimension growth.

• If dimension do not grow, we proceed with copies, following the same principle of indexed read, but in the reverse direction.

• When dimension growth is detected, we allocate memory to fit the extended destination tensor.
• Then we iterate along each dimension, copying the destination tensor values into the extended destination tensor.

• Once the data is copied, we proceed with copying the values from the source tensor into the existing tensor.

• After the end of the procedure, the initial destination tensor memory region is freed.

**Built-in functions.** For each built-in function, we need to allocate memory before the computation is executed. CShape carries the logic to do so in the allocate method. It calculates the amount of memory required, based on the dimensions and type, and initializes the array on the heap.

**Type specialization** To lower M-IR computations into C-IR, we need to know the exact type of every variable at a given point in the program. Type inference helps, however, variables can have multiple types at the same point in the program. When several variables with multiple types become inputs to a function, the computation must consider all possible type combination in order to provide the correct implementation.

To handle this case, we generate code that will inspect the type and complex type descriptors and create nested switch statements for each type and complex type combination of the arguments of a given MATLAB function. Then, for each combination, we can cast the memory regions of the variables into the corresponding types, and perform the computation, specializing on the type combination. To illustrate, consider:

```scala
1 class CShapeLift(d: IR.ShapeDef, output: CShape, inputs: List[CShape]) {
2   // obtain all possible type / complex type combinations
3   val possibleTypes = inputs.map(c => { ... })
4   val possibleComplex = inputs.map(c => { ... })
5   // generate nested switch statements for all combinations
6   def liftTypes ( ... ) { ... }
7   def liftComplex ( ... ) { ... }
8   // specialize CShape into CArray and call the specialized function f
9   def applyList(f: List[CArray] => C.Exp[Unit]) = {
10      liftTypes(possibleTypes, (argsType, returnType) => {
11         liftComplex(possibleComplex, (argsComplex, returnComplex) => {
12            val cArrayInputs = inputs.zip(argsType.zip(argsComplex)) map {
13               case (shape, (tpe, isComplex)) => shape.getCArray(tpe, isComplex) }
14            f(cArrayInputs ++ output.getCArray(returnType, returnComplex))
15         }rapped)
16      })
17   }
18   def apply(f: (CArray) => Unit) = applyList(l => f(l(0)))
```
The `lift` function takes an M-IR definition node, a sequence of `CShape` instances, treating them as inputs, and another `CShape` instance treating it as an output. For each input, it will inspect the possible types, and use the type rules provided in the M-IR node, to recursively generate switch statements that correspond to one combination. Then, in every innermost nest of type / complex combinations, it will call the `getCArray` method for all inputs and the output, using constant types and constant complex types. The result is a list of `CArray` instances that specialize each `CShape` with a particular type combination. Then we can apply the instances as input and output to a function specified through the `applyList` method.

`CArray` is a metacontainer that has a fixed type and a fixed complex type. It uses the memory region of the corresponding `CShape`, casting it to the specified type. As a metacontainer, `CArray` can be specialized into `CArrayReal` or `CArrayComplex` to represent real and complex arrays respectively. Furthermore, they can further specialize into `CScalarReal` or `CScalarComplex` to represent scalar arrays. An overview of all metacontainers used in MGen is given in Figure 5.6.

Each `CArray` operates with `Element`, which can be `Real` or `Complex`. All these are built as metacontainers, and they follow the principles explained in Section 5.3. Combined with the `lift` convenience function, we can provide a powerful abstraction that specializes on any type combination in lowering M-IR computations into C-IR. To illustrate, consider addition:
id: Rep[Shape]

```
class Shape {
  val id: Rep[Shape]
  val ld: DArray
  val pt: Set[Type]
  val pc: Set[Boolean]
}
```

class CShape {
  val ld: DArray
  val data: Rep[Array[Void]]
  val td: Rep[Type]
  val cd: Rep[Boolean]
}

class CShapeInput extends CShape
class CShapeIntermediate extends CShape
class CShapeOutput extends CShape

```
class DArray {
  def length (): Rep[Int]
  def apply (i: Rep[Int]): Rep[Int]
  def update (i: Rep[Int], v: Rep[Int])
  def equals (o: DArray): Rep[Boolean]
}
```

class DArrayScalar extends DArray
class DArrayFixed extends DArray
class DArrayStaged extends DArray

Specialized dimensions representations

```
DArray exists in both M-IR and C-IR
```

Specialized input / output arguments and intermediate variables

```
class CArray {
  def copy (): CArray
  def length (): Rep[Int]
  def apply (i: Rep[Int]): Element
  def update (i: Rep[Int], v: Element)
}
```

class CArrayReal extends CArray
class CArrayComplex extends CArray

Specialized scalars or arrays

```
class VArray {
  def apply (i: Rep[Int]): VectorElement
  def update (i: Rep[Int], v: VectorElement)
  def copy (): VArray
  def length (): Rep[Int]
}
```

class Real extends Element
class Complex extends Element

Real / Complex elements

Operations on scalar and / or vector elements result in emitting C code

ISA abstraction

VectorElements
VectorOperations
PackedOperations
mistrinsics
lms-intrinsics

C Code

**Figure 5.6:** Metacontainers used in MGen
The lowering of the addition operator, shown above, will iterate linearly through all elements of the tensor and perform addition. The abstraction allows us to handle any type combination, and any complex type combination, in a single codebase.

**Numerical Operations**  In order to have a powerful abstraction as shown above, we need strong support on primitive operations in LMS. LMS supports primitive operation, but does not provide support for saturated arithmetic. Therefore we extended LMS, providing the standard operations including addition, subtraction, multiplication, division and power with saturation. Some of these functions are straightforward to implement:

```c
void saturated_subtraction_uint32_t(uint32_t a, uint32_t b) {
    if (a < b) return 0; else return a - b;
}
```

For others, we used the MATLAB Coder [121] to generate different type combinations, and then we stage the generated code back in LMS. For example, a saturated power function of two unsigned 32-bit integers will have the following form:

```c
void saturated_power_uint32_t(uint32_t a, uint32_t b) {
    uint32_t x, ak, bku; int32_t exitg1; uint64_t u0;
    ak = a; x = 1U; bku = b;
    do {
        exitg1 = 0;
        if (((bku & 1U) != 0U) {
            u0 = (uint64_t)ak * x;
            if (u0 > 4294967295UL) u0 = 4294967295UL;
            x = (uint32_t)u0;
        }
        bku >>= 1U;
        if (((int)bku == 0) exitg1 = 1; else {
            u0 = (uint64_t)ak * ak;
            if (u0 > 4294967295UL) u0 = 4294967295UL;
            ak = (uint32_t)u0;
        }
    } while (exitg1 == 0);
    return x;
}
```
For all arithmetic and relational operations, we implemented all type combinations that MATLAB supports. To illustrate how MGen generated code looks like, consider the computation as \( C = A \times B \), applied to two matrices of size \( 10 \times 10 \), such that the first matrix has 16-bit unsigned complex elements, while the second contains double-precision real elements. MGen-generated code is given in Figure 5.7. For simplicity, we omit the initialization code that reads from shape_t structure in the beginning, and also omit the code that populates the return structure.

### 5.4.7 Vectorization

In MGen we vectorize each operator individually. The vectorization is done through layers of abstractions, illustrated in Figure 5.6, that build on top of the \texttt{lms-intrinsics} package, introduced in Chapter 2. We provide a short description for each layer, starting from low-level abstractions and working our way upwards.

**mistrinsics** “Mistrinsics” stands for missing intrinsics. Namely, Intel ISAs provide instructions for various vector type combination, however, not the complete set of type combinations. Consider SSE and Table 5.2. We can observe that while comparison is provided for signed integers, it is not available for unsigned integers. Also, saturated addition is available for 16-bit unsigned integers, but not for 32-bit unsigned integers. We can use the principles for low-level abstractions, discussed in Chapter 3, and implement the comparison on top of existing ISA instructions. For example a 32-bit unsigned comparison can be implemented in SSE2 using:

```python
1  def _mm_cmpgt_epu32 (a: Exp[\_\_m128i], b: Exp[\_\_m128i]): Exp[\_\_m128i] = {
2     val min_int = _mm_set1_epi32(Const(Int.MinValue))
3     val xor1 = _mm_xor_si128(a, min_int)
4     val xor2 = _mm_xor_si128(b, min_int)
5     _mm_cmpgt_epi32(xor1, xor2)
6  }
```

Consequently, we can also provide a vectorized implementation for saturated addition on unsigned 32-bit integers, in SSE2:

```python
1  def _mm_adds_epu32 (a: Exp[\_\_m128i], b: Exp[\_\_m128i]): Exp[\_\_m128i] = {
2     val max_int = _mm_set1_epi32(Const(UInt.MaxValue.intValue))
```
static inline uint16_t ssub_uint16_t(uint16_t a, uint16_t b) {
    if (a < b) return 0;
    else return a - b;
}

static inline uint16_t sadd_uint16_t(uint16_t a, uint16_t b) {
    if (a > UINT16_MAX - b) return UINT16_MAX;
    else return a + b;
}

void addition (shape_t x0, shape_t x1, shape_t* x2) {

    // Initializaton code to read from shape_t
    for (int32_t i28 = 0; i28 < 10; i28 = i28 + (1)) {
        int32_t i30 = i28 * 10;
        for (int32_t i29 = 0; i29 < 10; i29 = i29 + (1)) {
            int32_t i31 = i30 + i29;
            int32_t i32 = 2 * i31;
            int32_t i34 = i32 + 1;
            x27[i32] = 0;
            x27[i34] = 0;
            for (int32_t i36 = 0; i36 < 10; i36 = i36 + (1)) {
                uint16_t x57 = x27[i32];
                uint16_t x58 = x27[i34];
                int32_t i37 = i30 + i36;
                int32_t i38 = 2 * i37;
                uint16_t x39 = x25[i38];
                double x45 = (double) x39;
                int32_t i42 = i36 + 10;
                int32_t i43 = i42 + i29;
                double x44 = x26[i43];
                double x46 = x45 + x44;
                double x47 = round(x46);
                double x48 = (x47 > UINT16_MAX) ? UINT16_MAX : (x47 < 0) ? 0 : (uint16_t) x47;
                x27[i32] = x48;
                x27[i34] = x49;
                x27[i32] = x50;
                x27[i34] = x51;
            }
        }
    }

    // Code to populate the return instance of shape_t
    for (int32_t i28 = 0; i28 < 10; i28 = i28 + (1)) {
        int32_t i30 = i28 * 10;
        for (int32_t i29 = 0; i29 < 10; i29 = i29 + (1)) {
            int32_t i31 = i30 + i29;
            int32_t i32 = 2 * i31;
            int32_t i34 = i32 + 1;
            x27[i32] = 0;
            x27[i34] = 0;
            for (int32_t i36 = 0; i36 < 10; i36 = i36 + (1)) {
                uint16_t x57 = x27[i32];
                uint16_t x58 = x27[i34];
                int32_t i37 = i30 + i36;
                int32_t i38 = 2 * i37;
                uint16_t x39 = x25[i38];
                double x45 = (double) x39;
                int32_t i42 = i36 + 10;
                int32_t i43 = i42 + i29;
                double x44 = x26[i43];
                double x46 = x45 + x44;
                double x47 = round(x46);
                double x48 = (x47 > UINT16_MAX) ? UINT16_MAX : (x47 < 0) ? 0 : (uint16_t) x47;
                x27[i32] = x48;
                x27[i34] = x49;
                x27[i32] = x50;
                x27[i34] = x51;
            }
        }
    }

    // Code to populate the return instance of shape_t

}

Figure 5.7: MGen-generated for matrix-matrix multiplication of two 10x10 matrices. The first matrix has 16-bit unsigned complex elements, while the second contains double-precision real elements
In MGen, we build these low-level abstractions to complete the ISA set for numerical and relational operators, focusing on vectors of 8-, 16-, 32- and 64-bit integers, signed and unsigned, as well as single and double precision floating point numbers. We also complete the sets for permute and blend operations and provide functions that perform fast integer division. All functionality is implemented as a standalone package, called mistrinsics package, providing implementations for SSE and AVX.

**Packed Operations** On top of the mistrinsics, we build abstractions using metacontainers. We use the Packed metacontainer that contains a staged variable representing a vector primitive, and its base type. Then we provide operators that also include a parameter that describes the set of available ISAs. This parameter allows us to opportunistically select the best ISA, and dispatch the corresponding implementation:
<table>
<thead>
<tr>
<th>Name</th>
<th>SSE</th>
<th>SSE2</th>
<th>SSE3</th>
<th>SSSE3</th>
<th>SSE4.1</th>
<th>SSE4.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm_add_ps</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_add_pd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_add_epi8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_add_epi16</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_add_epi32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_add_epi64</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_adds_epi8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_adds_epi16</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_adds_epi32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_adds_epi64</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_adds_epl8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_adds_epl16</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_adds_epl32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_adds_epl64</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_mul_ps</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_mul_pd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_mullo_epi8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_mullo_epi16</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_mullo_epi32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_mullo_epi64</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_cmpgt_ps</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_cmpgt_pd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_cmpgt_epi8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_cmpgt_epi16</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_cmpgt_epi32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_cmpgt_epi64</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_cmpgt_epl8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_cmpgt_epl16</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_cmpgt_epl32</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_cmpgt_epl64</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_blendv_ps</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_blendv_pd</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>_mm_blendv_epi8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 5.2:** Available intrinsics in SSE
Packed operations are analogous to PackedNumericOps, introduced in Section 4.3.1, but also allow the implementation of operations with operands of different type. In MGen, we do not provide all possible type combination for packed operators. Instead, we only focus on vector operations that involve equal types.

**Vector operations and vector elements** From this point forward, we use the principles from Chapter 4 to build abstractions for data-level parallelism, however, we use metacontainers instead of type classes. VectorOps or vector operations build on top of packed operations. They are responsible for dispatching the correct implementation to satisfy MATLAB type interaction. For example, if saturated arithmetic is expected, they dispatch the implementation that uses saturation, and if not, they proceed with packed operators that do not reason about overflows and underflows.

VectorElement builds on top of VectorOps, representing complex or real vector elements. They are analogous to ElementOps introduced in Section 4.3.1. However, as they are not type classes, but instances, they can reason about the layout of the vector primitive they represent, providing specialized implementations for split complex representation or an interleaved complex representation.

**VArray** Finally, to reach full circle and enable vectorization, we specialize CArray into VArray. This metacontainer is analogous to CVector, introduced in Section 4.3.1, providing vectorized loads and stores that apply or update an instance of VectorElement in the array.

```scala
abstract class VArray { import C._
  val tpe: Type
  def apply (i: Rep[Int]): VectorElement
  def update (i: Rep[Int], s: VectorElement): Unit
}
```

VArray can also be further specialized, representing arrays of complex or real elements. Furthermore, if it specializes in representing a single scalar element, apply becomes a broadcast, and update extract elements from the vector primitive.

Every time an operator is lowered to the C-IR, we use the available set of ISAs provided as input in MGen. If vectorization is available, the operator will generate vectorized code. If not, MGen will switch to the scalar version that is available for all type combinations.
5.5 RESULTS

In this section we evaluate MGen’s capability to generate correct C code for given set of MATLAB functions. First, we give an overview on the infrastructure built to validate MGen-generated code against MATLAB virtual machine execution for a given MATLAB function. Then we use the infrastructure to derive thorough tests to validate type and shape inference.

We compare MGen capabilities to MATLAB Coder [121], a MathWorks commercial solution for MATLAB to C/C++ compilation. We also compare it to the recent open source research alternatives MATISSE [106], Mc2For [99] and MiX10 [97]. The high level overview of this comparison is shown in Table 5.3.

Finally, we evaluate MGen by generating correct C code for common MATLAB computations and discuss limitations.

5.5.1 Evaluation

TESTING INFRASTRUCTURE Our evaluation is based on MATLAB version 2016A. MGen requires Java 8 and Scala SBT to compile, and is compatible with Mac OS X, Linux and Windows. It generates C11 compliant code, which is supported by popular C compilers such as GCC, clang and Intel.
Figure 5.8: Validation setup: MGen interfaces with MATLAB through inter-process communication, passing MXArray instances. To communicate with the generated C code, the instances are inlined in the code, and the result is read from stdout.

ICC. Our testing environment uses Mac OS X 10.11.6, Java JDK version 1.8.0 144-b01, Intel ICC compiler version 17.0.4, and LLVM using clang version 800.0.42.1.

All tests are implemented using ScalaTest [122]. For validation, MGen assumes a pre-installed MATLAB version. It is able to spawn a MATLAB VM process from within the JVM, and execute arbitrary MATLAB code. The MATLAB VM is controlled through inter-process communication as illustrated in Figure 5.8, based on the MATLAB Engine API (libeng) [123]. The API provides the routine engEvalString to execute any MATLAB code and two routines engPutVariable and engGetVariable, to pass instances of an mxArray data-structure. The mxArray structure is defined in the MATLAB C Matrix API (libmx) [123] and represents a multi-dimensional complex or real tensor, having one of the 12 MATLAB primitives as a base type. As libeng and libmx APIs are available as C libraries, MGen uses BridJ [124] to access mxArray structures and implements MXArray as a Scala wrapper on top of it.

Both MATLAB VM and JVM use IEEE754 standard for floating point representation and the same set of primitives to represent 8-, 16-, 32- or 64-bit integers. Consequently the mxArray structure is binary interoperable between the two runtimes. This allows us to define any number of mxArray instances and use them as input to execute any MATLAB function in the MATLAB VM through the JVM, and obtain the result back to the JVM.

We can do a similar process for the code generated by MGen. For each generated function, MGen can generate a main function that invokes the generated code and returns the result. The input arguments are passed to
the generated function, such that a list of \texttt{MXArray} instances are serialized and inlined in the \texttt{main} function. The whole code bundle is then compiled and executed as a standalone program. Once the program terminates, MGen converts the output to \texttt{MXArray} instances. The execution of the standalone program is monitored using Valgrind [125] to detect any possible memory leakages.

The testing infrastructure allows us to execute any MATLAB function, in MATLAB VM, or as a standalone MGen-generated C program, and compare the results in the JVM runtime.

\textbf{Validation of operators} With the validation infrastructure in place, we can easily invoke any function of any arity, and perform extensive testing. To illustrate this process, consider a binary operator \((\texttt{op})\), or a binary function \texttt{foo}. We generate a MATLAB function in either of the two forms:

\begin{verbatim}
function \[x\] = validate (A, B)
    x = A (op) B;
end

function \[x\] = validate (A, B)
    x = foo(A, B);
end
\end{verbatim}

For the input we create two \texttt{MXArray} instances \(A\) and \(B\), that are initialized with random values. These structures, along with the MATLAB function are then transferred to the MATLAB runtime and the function \texttt{validate} is executed. The result is then transferred back to the JVM. If the execution has no errors, the function is then staged in MGen, and a standalone C program is generated with \(A\) and \(B\) inlined in the \texttt{main} function. MGen then anticipates successful compilation and execution of the code, resulting with an \texttt{MXArray} instance with identical type, dimensions and values as the one obtained from the MATLAB runtime. Any deviation of this process is considered as a test failure.

The generated code must be able to handle computations with different assumptions, i.e. tensors with different input types, complex or real data, fixed or variable dimensions, etc. To test the validity of all these different assumptions, we generate different code variants, such that for each input argument:

1. The type can be one of the 12 primitive types.
2. The tensor can represent complex or real data.
3. The dimension length can be either be fixed, having 1, 2 or 3 dimension, or be passed as a variable.

4. If the dimension length is variable, then the values for each dimensions are passed as an array. If the dimension length is fixed, then we pass a 1, 2 or 3 dimensional tensor with fixed or variable dimensions.

5. We execute the tests several times, each case using either scalar, SSE or AVX code.

This creates more than 144 different instantiations per single `MXArray` structure. For unary operators, we execute all possibilities, while for functions with arity larger than 1 we group the functions in different groups, depending on the specifications of the built-in function. For example, for a single pointwise addition operator, we perform type validation with two scalar instances, one scalar and one tensor, and two tensors. This is done because the `+` operator is defined in two ways: for two tensors with identical dimensions, or one tensor and a scalar. In each of the 3 groups, we try all possible type combinations and complex combinations, which leads to about 5184 combinations. We execute 3 times, having vectorization disabled, SSE enabled, and AVX enabled. In total this amounts to 15552 tests.

The process for testing is relatively efficient. The original MATLAB function is loaded only once in the MATLAB runtime, and can be reused several times with the different instantiations of the `MXArray` arguments. The majority of time is spent compiling the standalone MGen-generated C code and takes about one hour to validate a single operator on an Intel Core i7-4980HQ machine. Our observations suggest that:

- **MGen** produces results which are binary compatible with MATLAB in most cases. All operations resulting in an integral type are bitwise identical. The same applies to IEEE754 computations with standalone implementation in MGen. However, once external libraries are used, such as trigonometric functions, logarithms, power functions etc, binary compatibility is no longer guaranteed. The reason for this behaviour comes from the fact that these functions are typically part of the C standard library (`libc`), and even though MATLAB is linked to `libc`, it is very likely that it uses a proprietary implementation of the same set of functions. The binary inconsistency between the different implementation also applies when using vector instructions such as SVML for the same set of trigonometry functions, logarithms and the power function.
MATLAB Coder handles all numerical types, with few exception. For example, the plus operator fails once characters are added to 8-bit integers:

```matlab
function [x] = addition(A)
    x = A + 'a';
end
```

On the other hand, executing the same code in MATLAB, we obtain:

```matlab
>> a = int8(10);
>> a + 'a'
ans = 107
```

MATISSE shares the same problem as the MATLAB Coder, and fails adding a character to any other type. Furthermore, handling any integer types does not seem to use saturated arithmetic, and the generator does not handle any complex types.

Mc2For / MiX were able to generate code for all type combinations of the basic arithmetic operators, and also other supported operators. However, none of the two produced correct results, in terms of integer operations, as they lack support for saturated arithmetic.

**TYPE INFERENCE VALIDATION**

A MATLAB variable can change its type at any point in the program. Changes happen upon assignment. Therefore, to test different variants, we generate tests in the following manner:

1. We use if-constructs, such that we assign a variable with a different type in one of the conditional branches, in both, or in none. The instances of different types are obtained either by computations or initialization.

2. We use for and while-constructs and assign a variable with a different type inside the body of the loops.

3. We combine the conditional constructs with the loop constructs to create an arbitrary level of nesting up to a depth of 3.

Unlike the process of validating the operators, these tests are not automated, but are handwritten. Our results are:
• **MGen** passes all our tests, inferring all types at each point in the code. Due to the simplicity of the data-flow analysis used for type inference, MGen may overapproximate the possible types for a given variable. While this results in generating inefficient code to handle the overapproximated types, it is nevertheless correct.

• **Mc2For / MiX10** use MATLAB Tamer [98] as a front-end to perform type inference and analysis. All our tests suggest that types are correctly inferred at each location in the code. For this part, we disregard the generated code from Mc2For and MiX10, and only focus on the intermediate representation generated in MATLAB Tamer.

• **MATLAB Coder** and **MATISSE** do not pass all our tests. A concise example for type inference failure can be illustrated with the following function:

```plaintext
function [result] = addition(a)
    while a < 10
        a = a + int32(1);
    end;
    result = a;
end
```

This is no surprise as MATISSE can handle variables that change types in a function, provided that they only have a single type at each point of the program. MATLAB Coder on the other hand also does not support changing types through assignment.

**Shape inference validation**  Similarly to the type of the variable, the shape of a tensor can change at any point in the program. Changes happen upon assignment, as well as upon indexed assignment. To test the behaviour we perform testing using the same nesting variants as described in the type inference validation. However, we add two more rules that will be combined with the rest of the variants:

1. Perform indexed assignment within bounds of a given tensor and also outside bounds of a given tensor. We focus on MATLAB code such that a scalar tensor gets extended into a vector, or a matrix into a 3-dimensional tensor.

2. Use built-in operators that are overloaded on the shape of the tensor. For example a pointwise addition operator which is overloaded for adding a scalar and a tensor and two tensors.
Our observations show that MGen and MATLAB Tamer pass all our tests. This is not true for MATLAB Coder, as well as MATISSE. Consider the following function that tries to extend a 2-dimensional matrix into a 3-dimensional tensor:

```matlab
function result = foo(a);
a(:,:,2) = a(:,:,1)
result = a;
end
```

MATLAB Coder reports an error stating that the index expression is out of bounds, while MATISSE, without providing any further explanation, returns a java.lang.RuntimeException.

Apart from index assignment, both generator have troubles handling overloaded operators. Consider the function below, such that `a` is setup as a matrix, and `b` is setup as a scalar:

```matlab
function [result] = foo(a, b)
if (rand > 0.5)
c = a .* a;
else
    c = b .* b;
end;
result = a + c;
end
```

In this function, `c` can be either a matrix or a scalar, with equal probability (due to the `rand` function at line 2). Thus the addition at line 7, is either a matrix-to-matrix addition, or a matrix-to-scalar addition. MATLAB Coder generates invalid code, such that addition at line 7 (line 9 in the generated code) is treated as a matrix-to-matrix addition:

```c
void foo(const double a[100], double b, double result[100]) {
    double c_data[100];
    int i0;
    if (b_rand() > 0.5) {
        for (i0 = 0; i0 < 100; i0++) { c_data[i0] = a[i0] * a[i0]; }
    } else {
        c_data[0] = b * b;
    }
    for (i0 = 0; i0 < 100; i0++) { result[i0] = a[i0] + c_data[i0]; }
}
```
The code will attempt to access memory space that is invalid at line 9, and cause a memory fault. MATISSE on the other hand does not even generate code, but throws an `UnsupportedOperationException`.

**Validation of Programs** Once we have successfully validated type and shape inference, and correctness of individual operators, we evaluate the correctness of MGen to generate valid code on commonly used numerical computations. We collect a set of benchmarks, acquiring MATLAB code from a variety of sources and related projects, including Mc2For [99], FALCON [101], OTTER [111] and The MathWorks Central File Exchange [126]. The benchmark corresponds to the subset of MATLAB supported by MGen, including standard constructs such as `if-else`, `for` and `while` loop statements; dynamic growth of tensor dimensions, array growth by out-of-bound array indexing; multiple type values per variable; and built-in function overloading. A brief description of each benchmark is given below:

1. `fib` calculates the $n$-th Fibonacci number. The benchmark includes a simple `for`-loop.

2. `gd` calculates 1000 iterations of gradient descent represented by equation 3.7 in Section 3.3. The benchmark includes a `for`-loop with matrix transposition, matrix-vector multiplication, and pointwise addition and multiplication of vectors.

3. `adpt` finds the adaptive quadrature using Simpson’s rule. This benchmark features an array whose size cannot be predicted before compilation.

4. `bbai` uses Babai’s algorithm computed on fixed-sized arrays.

5. `bubl` is the standard bubble sort algorithm. This benchmark contains nested loops and consists of many array read and write operations.

6. `capr` computes the capacitance of a transmission line using the finite difference and Gauss-Seidel method. It is loop-based and involves scalar operations on two small-sized arrays.

7. `clos` calculates the transitive closure of a directed graph. It contains matrix multiplication operations between two 450-by-450 arrays.

8. `crni` computes the Crank-Nicholson solution to the heat equation. This benchmark involves some elementary scalar operations on a 2300-by-2300 array.
9. *dich* computes the Dirichlet solution to Laplace’s Equation. It’s also a loop-based program which involves basic scalar operation on a small-sized array.

10. *diff* calculates the diffraction pattern of monochromatic light through a transmission grating for two slits. This benchmark also features an array whose size is increased dynamically like as in the benchmark *adpt*.

11. *fiff* computes the finite-difference solution to the wave equation. It is a loop-based program which involves basic scalar operation on a 2-dimensional array.

12. *mbrt* computes a Mandelbrot set with a specified number of elements and number of iterations. This benchmark contains elementary scalar operations on complex data.

13. *nb1d* simulates the gravitational movement of a set of objects. It computes on vectors inside nested loops.

For each of the 13 experiments, we setup a separate ScalaTest wrapper to include the MATLAB code as well as the setup of the input arguments to fit the requirements of each function. Similarly as before, we run the code through the testing infrastructure. Our experiments confirm that MGen generated code produces the same functionality as the MATLAB runtime.

**Performance Evaluation**  In the current state, MGen does not offer competitive performance. The biggest impediment in this aspect is the lack of memory optimizations. Namely for each intermediate computation, memory is allocated to facilitate the temporary values. To allocate, we use the *malloc* system function that imposes significant overheads. The overheads are particularly crucial if a given computation is part of a hot-path of the code.

To illustrate the diminishing effects due to lack of memory optimizations on performance, we consider the the gradient descent benchmark. We evaluate 3 versions: MATLAB VM, MGen, and MATLAB Coder generated code. We benchmark on an Intel Core i7-4980HQ machine with 16GB of RAM and 25.6 GB/s bandwidth to main memory. For MGen and MATLAB Coder generated code, we use *RDTSC* to measure cycle count, and for MATLAB VM we use *tic* and *toc* to measure elapsed time. We perform 15 repetitions with warm cache, and report the median result. To avoid the effects of
frequency scaling and resource sharing on the measurements, Turbo Boost and Hyper-Threading are disabled.

We set up the benchmark to run for 10 iterations, operating on an $n \times n$ double precision matrix and a double precision vector of size $n$. For all 3 versions, the flop count is given as $4n^2 + n$ flops, and we report the results in flops / cycle.

Figure 5.9 shows us the performance profile. MGen generated code is up to 37 times slower than MATLAB Coder and up to 8 times slower than MATLAB VM execution time. The low variance of the measurements for different matrix sizes suggest that the computation is dominated by the memory allocations.

5.5.2 Limitations

The work presented in this chapter focused on the abstractions required to stage MATLAB code and to produce correct code. However, many performance optimizations are still missing and the consequence is a huge impact on the performance delivered by MGen. These limitations are not due to the proposed abstraction model of metacontainers, but opportunities that this work has not yet explored.

**High level optimizations** MGen is capable of performing some high level optimization such as dead code elimination and common subexpression elimination. However, many performance-relevant optimizations such as multi-level tiling, loop merging and loop exchanging are not im-
implemented in this work. Even before applying loop optimization, semantic properties of matrix operations can be used to perform high-level optimizations [127]. Prior work [128–130] shows that MATLAB programs can be partially evaluated and optimized by automatically transforming loops to equivalent computations already available in the built-in operations. This suggests that optimization opportunities are available even before staging the initial MATLAB code. As shown in the previous chapters, these optimizations can certainly be implemented in LMS, and we believe that the metacontainer abstraction can provide convenience and productivity in integrating these optimizations.

**Type Inference Overapproximation** Types inference in MGen results in overapproximation. This is a consequence of the simplistic usage of data-flow analysis in MGen, based on reaching definitions. While the implementation used in LMS is flow-sensitive, it is not path-sensitive. Integrating additional static analysis techniques, combined with more sophisticated means for data-flow analysis can significantly improve the type inference precision. The same applies also for shape inference, as it also uses data-flow analysis to reason about dimension length and values.

**Memory Optimizations** As indicated in the evaluation subsection, MGen does not employ any optimizations relevant to memory. Intermediate computations result with memory allocations that increase the overall memory footprint of the program, and thus introduce overheads. These problems can be alleviated by performing in-place computations when possible, reusing memory regions in subsequent computations and coalescing arrays as suggested in prior work [131].

**Range Analysis** Range analysis is not implemented in MGen. As a result, tensor indexing and indexed assignment operations, which are quite common in MATLAB programs, must undergo bound checks to ensure valid memory accesses. Many of these run-time checks can be removed. Recent work of MATISSE [132] demonstrates integration of Z3 theorem prover [133] to perform algebraic analysis in order to remove runtime checks. Mc2For implements a range value analysis such that statically estimates the minimum and maximum values for each scalar variable. Analysis on the generated code from MATLAB Coder also suggests that some form of range analysis is implemented in the closed source generator.
5.6 RELATED WORK

The Sable Lab at McGill University has done extensive work around the MATLAB language [118], developing an open-source MATLAB virtual machine, toolkits for static compilation and source-to-source MATLAB translation. We provide an overview of these projects:

• **McVM [96]** is a virtual machine that performs function specialization based on the runtime knowledge of the types and shapes in function calls. While McVM has an extensive support for hundreds of built-in functions, support is not provided for integer matrices, or floating-point matrices with single precision. McVM uses an LLVM-based JIT compiler and does not perform vectorization. Instead it uses LLVM autovectorization and efficient SIMD implementation of the included BLAS / LAPACK libraries to benefit from data-parallelism.

• **MATLAB Tamer [98]** is a compiler toolkit that can take MATLAB programs as input and produce output which is suitable for static compilation. It uses static analysis techniques to infer shape, class, and complex information, supporting more than 300 built-in MATLAB functions. To achieve that, the authors built a domain-specific language to categorize type behaviour of each built-in function, and to provide flow equations required for shape inference. Separate DSLs are created for shape inference, class propagation, as well as complex information propagation. All three DSLs are then processed by ANTLR [134], resulting in 3 separate AST trees that will be interpreted by the MATLAB Tamer tool to perform analysis. The models of all built-in functions are stored in a single .ser file, where flow equations, class and complex propagation rules are defined.

Designing code generators following the principles of MATLAB Tamer is a tedious process, as it requires the development and maintenance of three separate DSLs, to describe type rules, dimension constraints, and dimension changes. Updating a built-in function might result in either an update of the .ser file, an update to the DSL (to support new math routines / constructs), or the interpreter that does the analysis, or any combination of the three. While built-in functions do not change often, and maintain backward compatibility, MATLAB Tamer does not offer any diligence in generating MATLAB version specific built-in models.
• Mc2For [99] and MiX10 [97] are backends to MATLAB Tamer, generating Fortran and X10 code respectively. Both projects support operations on integer types, however do not conform to the specifications of MATLAB in providing saturated arithmetic. Implementation of BLAS routines, in particular BLAS 3 routines, depends on external BLAS libraries or Intel MKL, suggesting that the scope of supported data types in these operations is limited to the specifications of the libraries used. Similarly to McVM, both approaches rely on the compiler for optimizations such as automatic vectorization and parallelization.

Apart from the work of the Sable group, the translation of MATLAB to other programming languages is not recent. De Rose and Padua proposed the FALCON environment [100–102] to translate MATLAB to FORTRAN90 code. They leverage an aggressive use of static analysis and type inference for base types (doubles and complex) as well as shape of the matrices. Joisha et al. [103, 104] and Olmos et al. [105] present additional type and shape inference techniques. All of these techniques were capable to provide a partial support of the MATLAB type system, and were based on classical AST rewrites to perform shape and type inference.

MATLAB Coder [121] is a MathWorks proprietary solution for MATLAB to C/C++ compilation. It supports a large subset of MATLAB and allows code customizations through directives and options. The GUI interface provides capabilities to limit stack usage, to enable or disable usage of saturation arithmetic, to include external libraries such as OpenMP for parallelization, or to invoke custom LAPACK callbacks. MATLAB Coder generates efficient code that is also readable. However, in many cases it is incapable of supporting basic arithmetic operations, or type inference for specific cases. Until version R2016b, MATLAB Coder did not support changing types through assignment and was not able to generate standalone ISA specific vector code.

MATISSE [106] also translates MATLAB to C with focus on embedded systems. It also focuses on heterogeneous architectures and translates MATLAB to OpenCL [107]. It supports generation of highly customized C code by employing user-provided hints. The work explores the use of Aspect-Oriented Programming (AOP) concepts, using the LARA [135] domain specific language. While the use of AOP provides convenience in building compilers and code generators, MATISSE does not aim at generating a single correct / efficient implementation and does not support complex numbers or variables that can hold multiple types at the same point in
the program. Consequentially the mechanisms for type inference, handling multidimensional tensors and overloaded operators are limited.

MATLAB is a proprietary software, however the MATLAB language has several free open-source alternatives, including Octave [136], Scilab [137] and FreeMat [138] that are mostly compatible with MATLAB. In that aspect, another related work is Sci2C [139], that translates Scilab to C. Unfortunately, Sci2C is completely dependent on annotations embedded in the Scilab code to specify data sizes and types, and requires that the sizes of all arrays are fixed and known in advance.

Orthogonally to our work, there have been approaches to translate MATLAB / Octave to languages suitable to multicore and/or GPU architectures [109–113]. MEGHA [108] is a compiler that processes MATLAB / Octave scripts and generates CUDA and C/C++ code. MEGHA uses heuristics to decide which portions of the code should be executed on the CPU and which should be offloaded to the graphics processing unit (GPU). Chun-Yu Shei et al. [114] presents another MATLAB / Octave to CUDA compiler, which generates both C++ and CUDA code. However, unlike MEGHA, portions of the resulting code remain in MATLAB. Sharma et al. [115] explore the parallel capabilities MATLAB currently offers out-of-the-box (e.g., parfor). They focus on using MATLAB on a cluster environment and on the use of MPI.

5.7 SUMMARY

The work presented in this chapter uses object oriented stage polymorphism as an abstraction tool for building code generators. In particular, it demonstrates that through the use of the metacontainer abstractions, we can simplify the analysis and transformation of staged DSLs, and handle many aspects in generating code for a dynamically typed numerical DSL, such as MATLAB.

We show that metacontainers can take full advantage of IRs with a sea-of-node representation, allowing us to simplify lowering of high-level computation, by gradually adding new definitions in the IR. Furthermore, we show how metacontainers maintain relationships between IR nodes, that no longer require to be explicitly encoded in the IR and thus reduce its complexity. Consequently, analysis become simpler, as demonstrated with the use of constant propagation in the M-IR. The best property of metacontainers is the tight integration with transformers based on iterated staging, where symbol substitutions can be directly applied in the metacontainer.
This allows us to create simpler transformation phases and rewrite rules, that are agnostic towards the structure of the metacontainer. And once these rewrite rules are applied, the resulting substitutions can specialize metacontainers “on-the-fly”, dynamically swapping one metacontainer with a more specialized one, specializing the IR even further.

In this work, we demonstrate the use metacontainer abstraction to lower a dynamically typed language as MATLAB. Our prototypical generator, provides a systematic approach in inferring MATLAB types and shapes, providing a complete support of all numerical primitives in the subset that it supports. Furthermore, as metacontainers can provide type-class like functionality, we also demonstrate how the same set of data parallelism can be applied to vectorize MATLAB code.
Conclusion and Future Work

We restate the goal of this dissertation from the introduction:

Provide low-level and high-level abstractions that simplify the construction, transformation, and optimization of staged DSLs. The provided abstractions will incorporate knowledge of a wide variety of instruction set architectures (ISA), take advantage of data parallelism, integrate tightly in a staging environment and offer high performance.

The research presented in this work satisfies the goal of the dissertation. In the pursuit of low-level abstractions, we showed that we can complete an ISA by adding missing instructions. Specifically we demonstrated support for variable-precision computations on x86, including efficient implementation of 4-bit computations on state-of-the-art CPUs.

Once it comes to high-level abstractions, we demonstrated support for data parallelism though parametric stage polymorphism. Through the FGen prototype, we showed how we can apply this abstraction to perform performance oriented optimizations such as loop merging, loop unrolling with scalar replacement and vectorization to provide high-performant code for convolution operations.

Finally, with MGen, we introduced the metacontainer abstraction, and applied it in order to perform high-level analysis and transformations, such as type and shape inference, but also to provide low-level abstractions as a replacement for type-classes and parametric stage polymorphism. Furthermore, we showed that this methodology can be used to stage a general purposes, numerical DSLs such as MATLAB.

The modular approach of these methods can provide building blocks for future program generators, where analysis, transformations, data representation, instruction set architecture, and different code styles, can be reused, amortising the cost of building a generator from scratch.
FUTURE WORK The work presented in this thesis, can be improved and extended in many different ways. We provide a list of some possible directions.

- **MGen as an environment for other generators.** Once MGen completes the type and shape inference phases, the result is an IR that is expressive and contains informations for each tensor operation. One or several expressions that reside in one specific domain can be delegated to other generators, to perform lowering, or further optimizations. A good example for that would be MATLAB convolution operators that could be delegated to FGen, or linear algebra operations that could be delegated to LGen [8, 140]. The output of these generators can then be used either in lowering the M-IR to C-IR, or used to perform rewrites in the M-IR. A possible research question would be how to systematically extract expressions or subgraphs from the M-IR and automatically delegate them to other generators. In such a setting, MGen can become an environment for other generators in providing high-performant code.

- **SIMD code generation based on profile information.** The SIMD intrinsics DSLs are automatically generated from instruction specifications provided by Intel. It is interesting to observe that newer versions of the specification come with micro-architecture performance profiles. These profiles are based on throughput and latency, and are available for large subset of instructions. One possible benefit of these profiles is to use them to navigate optimizations and scheduling of instructions to maximize CPU utilization.

- **Support for other ISAs.** The automatic generation of DSLs for Intel ISAs can be easily extended for other architectures and their ISAs, for example - ARM Scalable Vector Extension (SVE). This will result in a support of multiple backends for LMS in producing optimized code. But even further, their existence in a staging environment with the already available ISA support can be used to build cross-architecture abstractions supporting data-parallelism or other features.

- **Optimizations in MGen.** Developing a MATLAB compiler is a challenging task by itself. Even more when using a research prototype as an underlaying compiler framework. We would like the research on MGen to continue, but in order to provide competitive performance, few optimizations are a priority, including:
1. **A more precise data-flow analysis.** MGen heavily depends on the quality of data-flow analysis required for type and shape inference. Improving it would greatly affect the transformation of the high-level M-IR, and in turn improve the generated C-IR code.

2. **Memory optimizations in MGen.** Optimizing memory copies and reducing memory allocation in MGen is crucial for performance. On top of that, optimizing indexed reads and index assignments to provide in-place operation would not only further improve performance, but also reduce memory footprint of the runtime of the generated code.

3. **Tiling and loop optimization.** Optimization such as loop merging and loop interchange become important when consecutive linear algebra tensor computations are executed within the same context. As a result, they can provide substantial improvements of the performance of the generated code. They can be done either at high level, as described in the work of FGen, or at a low level in LMS as shown before [141].

- **Multiple front-ends for MGen.** Currently MGen stages mathematical computation from a MATLAB file, or through an embedded DSL in Scala. Recent work [142] presents virtualization rules in Python to perform staging by types. Integrating this work, can provide yet another fronted for MGen, and enable machine learning frameworks already built in Python to be processed by our generator.


Publications


