DSP-MLIR: A Compiler for Digital Signal Processing in MLIR

by

Abhinav Kumar

A Thesis
Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved September 2024 by the
Graduate Supervisory Committee

Aviral Shrivastava, Chair
Chaitali Chakrabarti
Hwisoo SO

ARIZONA STATE UNIVERSITY
December 2024

ABSTRACT

Traditional Digital Signal Processing (DSP) compilers operate at a low level, such as C and assembly. Lowering from a high-level representation to low-level ones often results in a loss of high-level information, which is critical for optimizations based on DSP-specific theorems and properties. Consequently, traditional DSP compilers miss many optimization opportunities available at a high level. The MLIR (Multi-level Intermediate Representation) framework, an emerging multi-level compiler infrastructure, supports specifying optimizations at a higher level. This paper introduces a DSP Dialect within the MLIR framework to perform domain-specific optimizations based on DSP theorems and properties at a high level. Additionally, a domain-specific language (DSL) is proposed to facilitate the development of DSP applications. Experimental results demonstrate that domain-specific optimizations of the DSP dialect can improve the execution time of DSP applications by up to 10x compared to implementations in C and affine levels.

DEDICATION

To my brother, who always reminded me not to shy away from difficult challenges, teaching me that facing them head-on is what makes us stronger.

To my father, whose constant encouragement and belief in my abilities inspired me to always strive for my best.

To my mother, who, despite her illness, never once put her own struggles before mine, always prioritizing my work and well-being with endless love and care.

And to my sisters and their families, who have been my pillars of support, making this journey smoother with their kindness and warmth.

I am deeply grateful to each of you, for without your unwavering support, this would not have been possible.

ACKNOWLEDGMENTS

I would like to express my most sincere gratitude to my advisor, Dr. Aviral Shrivastava, for his active and patient support throughout my Thesis. I am also profoundly grateful to Atharva and Hwisoo, whose assistance and contributions have been immeasurable. Additionally, I extend my heartfelt thanks to Kaustubh, Sai, Adam, Rishabh, Shail, Omkar, Kaustubh Mhatre(Advant lab), Akarsh, Shubhvarata and Brownie for their invaluable help and constructive feedback, which have greatly enriched my work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Digital Signal Processing (DSP) is a crucial technology for signal processing that is useful in many areas including audio processing, speech processing, image processing, digital communication, and medical applications Wikipedia contributors (2024); Holton (2021). Many modern applications combine DSP with other computational tasks, such as deep learning, i.e., different domains. For example, speech recognition systems like Google Assistant, Siri, and Amazon Echo require audio preprocessing and deep learning. In medical imaging, filtering MRI scans and using deep learning for tumor detection Rasool and Bhat (2024) is crucial.

However, traditional compilers struggle to optimize DSP applications and diverse tasks combined with DSP, since they are hard to utilize domain-specific information at a high-level. The lowering process of the compiler from high-level to low-level programming languages, i.e., object code or machine code, loses the high-level domain-specific abstractions in the domains of DSP and other tasks, which are useful for optimizations. Existing compilers based on traditional compiler frameworks such as GCC (GNU Compiler Collection) GNU (2023) and LLVM (Low-level virtual machine) Lattner and Adve (2004) are more closely related to low-level languages. Hence, they cannot utilize such abstractions for optimizations.

MLIR (Multi-Level Intermediate Representation) Lattner *et al.* (2020), an emerging compiler infrastructure under the LLVM umbrella, overcomes the limitation of traditional compilers by supporting representations of multiple levels of instruction representation (IR) from close to the source language down to machine-level IR. This abstraction enables optimizations at the desired level of abstraction by utiliz-

ing domain-specific information in various domains such as quantum computing McCaskey and Nguyen (2021), machine learning models Jin *et al.* (2020); LLVM (2024); Hu *et al.* (2022), and hardware description languagesCIRCT (2024). In addition, MLIR supports representations of different IRs in the same infrastructure, which enables unprecedented cross-domain optimizations as well Martínez *et al.* (2022).

In this paper, we propose DSP-MLIR framework, which fully supports high-level representation and optimizations for DSP applications under MLIR infrastructure. This paper delivers a comprehensive compiler for DSP in MLIR, with a frontend language to develop DSP apps using supported DSP operations and a set of dialect-specific optimizations which would automatically optimize the code if one of the patterns matches giving better performing code. This approach not only enhances the performance of DSP applications but lays down a foundation for future cross-domain integrations. The main features of DSP-MLIR framework are as follows:

- This paper presents a novel MLIR Dialect for DSP and implements lowering routines to existing built-in MLIR dialects. The dialect includes operations covering a wide range of functionalities commonly used in DSP applications. Whenever possible, in order to make use of the powerful optimizations present in the Affine dialect, we lower the DSP operations to the Affine dialect. When not possible, we lower to structured control flow (SCF) dialect.

- This paper introduces DSP-Dialect-specific optimizations. All of the optimizations are based on the well-known theorems and properties in the DSP domain, which are straightforward to implement but unavailable in traditional compilers.

- This paper also provides a DSL (Domain-Specific Language) for writing DSP applications. Even though simple, this allows programmers to express their DSP applications in a programming language style (rather than writing in MLIR IR

format). The programs written in this DSL are converted to our MLIR DSP dialect, optimized, and lowered to Affine or SCF dialects.

By utilizing DSP-Domain Specific properties/theorems at our dialect level, we were able to achieve up to 10x performance improvement which would have been difficult for traditional compilers to achieve, i.e., at C-level.

The rest of this paper is organized as follows. Section 2 presents the background about MLIR and previous work for DSP compiler and MLIR dialects. Section 3 introduces DSP-MLIR framework consists of DSP dialect, DSP-specific optimizations, lowering for DSP dialect to Affine or SCF dialects, and domain-specific language (DSL) to develop DSP applications. Section 4 presents our experimental setup and evaluation strategy . Section 5 presents our experimental results for various DSP applications. Finally, we conclude our paper by discussing future work in sec 6.

Chapter 2

BACKGROUND AND RELATED WORK

MLIR, an emerging compiler infrastructure and a project under LLVM umbrella introduces multi-level Intermediate Representation and its IR structure is powerful enough to represent Graph IR (Tensorflow) , Affine (C-like) to lower levels IRs like LLVM IR. The mechanisms to engage with MLIR infrastructure are Dialects and there are dialects for different levels like tf for Tensorflow, onnx-mlir for onnx , affine for polyhedral optimizations, nvgpu for Nvidia GPUs. Dialects act as namespace for defining operations (to define functionality), types (to define datatype) and attributes (to get compile time information ex-constant data about operation) and in this work, we will introduce a new dialect for DSP domain and define set of operations required for signal processing applications.

There has been previous work in designing DSL for DSP like Fieldspar Axelsson *et al.* (2010) (which provides dataflow style of algorithm description based on Haskell and the backend compiler produces C code and the optimizations (variable elimination, loop unrolling) are done at C-level) and FAUST Letz *et al.* (2018) , another functional programming based language developed for audio processing for the Web. There has been work targetting backend DSP hardware and C/assembly-level software like DSP Processors co-design with compiler Zivojnovic *et al.* (1996) and for generating optimized code for DSP Processors using SIMD Lorenz *et al.* (2004) , Parallelization of C programs Franke and O'Boyle (2003), assembly-level optimizer de Dinechin *et al.* (2000), DSP processors address optimization Leventhal *et al.* (2005), enabling auto vectorized code generation Thomas and Bornholt (2024), etc while our work targets the frontend for DSP. Matlab , a popular framework for DSP Applica-

tions also has embedded coder Elrajoubi *et al.* (2017) which automatically generates fast C code for embedded processors but is closed-source.

There have been recent works to utilize MLIR for various domains like Quantum Computing (Quantum MLIR) McCaskey and Nguyen (2021), machine learning models (Onnx MLIR Jin *et al.* (2020), torch-mlir LLVM (2024)), and hardware description languages (Circt CIRCT (2024)) through specific dialects. Our work showcases another domain-specific (DSP) computations utilizing the MLIR framework.

Chapter 3

PROPOSED DSP MLIR FRAMEWORK

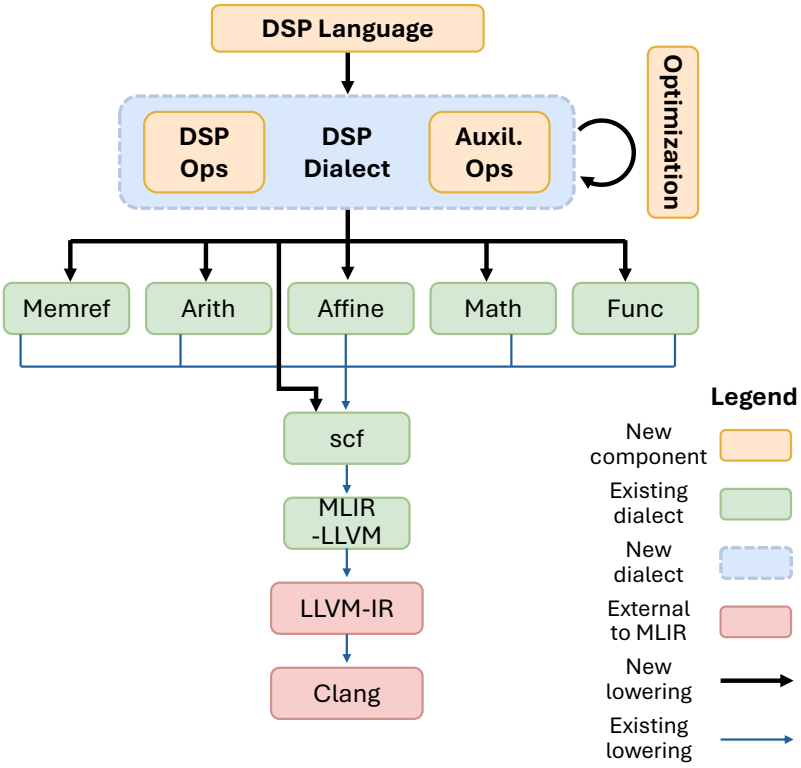

Figure 3.1: The compilation pipeline of the proposed DSP-MLIR framework. The DSP language is first compiled into the DSP dialect, and then lowered to Affine (whenever possible) or structured control flow (SCF), and then to LLVM IR and Clang.

DSP Framework consists of following components - A Dialect , a DSL , Lowering and Optimizations. The compilation pipeline for our framework can be seen as Fig 3.1.

The DSP Dialect

DSP Dialect provides two set of operations - one for DSP specific blocks [like signal operations (like delay), transforms (like dft , dct , idft), filter operations (like low-pass filter design, response)] and other for auxiliary operations (like sin, cos, vector of given size generation and print) for development of DSP applications as shown in Fig 3.1 . DSP applications here represent a set of various DSP operations combined to realize applications like audio compression, low-pass noise filter etc. Sample list of Operations and Applications are provided in Table 3.1. MLIR already supports tensor types which is sufficient to represent DSP datatypes which are mostly one-dimensional (we will demonstrate our work on one-dimensional applications although we support multi-dimensional inherently because of tensor type which may be required for image-processing etc.) hence we use tensor type directly.

```
module {
  dsp.func @main() {
    %0 = dsp.constant dense<0.000000e+00> : tensor<f64>
    %1 = dsp.constant dense<1.000000e+01> : tensor<f64>
    %2 = dsp.constant dense<1.000000e+00> : tensor<f64>
    %3 = "dsp.getRangeOfVector"(%0, %1, %2) : (tensor<f64>,
    tensor<f64>, tensor<f64>) -> tensor<*xf64>
    %4 = "dsp.reverseInput"(%3) : (tensor<*xf64>) ->
    tensor<*xf64>
    %5 = "dsp.FIRFilterResponse"(%3, %4) : (tensor<*xf64>,
    tensor<*xf64>) -> tensor<*xf64>
    %6 = "dsp.fft1dreal"(%5) : (tensor<*xf64>) ->
    tensor<*xf64>
    %7 = "dsp.fft1dimg"(%5) : (tensor<*xf64>) ->
    tensor<*xf64>
    %8 = dsp.mul %6, %6 : tensor<*xf64>
    %9 = dsp.mul %7, %7 : tensor<*xf64>
    %10 = dsp.add %8, %9 : tensor<*xf64>
    dsp.print %10 : tensor<*xf64>
    dsp.return
  }
}
```

```
#Power spectral density
def main() {
  var input = getRangeOfVector(0, 10, 1);

  #Get x[-l] ie, reverseInput & get conv1d
  var reverse_input = reverseInput(input);
  var conv1d = FIRFilterResponse(input, reverse_input);
  # Get dft of the conv1d
  var fft_real = fft1dreal(conv1d);
  var fft_img = fft1dimg(conv1d);
  # Get square of the dft
  var sq = fft_real * fft_real + fft_img * fft_img;
  print(sq);
}
```

Figure 3.2: DSP Dialect[left] and Corresponding DSL[right]

Domain Specific Patterns -Dialect Optimizations

In this section, we introduce the optimizations/patterns that we do at our dialect level. MLIR provides pass infrastructure to perform transformation or optimization, and the pass infrastructure includes the pass manager, PassManager, which is respon-

SN	OpName	Syntax	Equation	Description
1	Delay	`delay(input, n) -> output`	$y[n] = x[n-k]$	Delays the input signal by k samples.
2	FIRFilterResponse	`FIRFilterResponse(input, coeffs) -> output`	$y[n] = \sum_{i=0}^{M} h[i] \cdot x[n-i]$	Applies a Finite Impulse Response filter to the input signal.
3	SlidingWindowAvg	`slidingWindowAvg(input, window) -> output`	$y[n] = \frac{1}{N} \sum_{i=0}^{N-1} x[n-i]$	Computes the average of the input signal over a sliding window of size N.
4	FFT1DReal	`fft1dreal(input) -> real`	$X_{\text{real}}[k] = \sum_{n=0}^{N-1} x[n] \cos\left(\frac{2\pi}{N} kn\right)$	Computes the real part of the 1D Fast Fourier Transform of the input signal.
5	FFT1DImg	`fft1dimg(input) -> imag`	$X_{\text{imag}}[k] = -\sum_{n=0}^{N-1} x[n] \sin\left(\frac{2\pi}{N} kn\right)$	Computes the imaginary part of the 1D Fast Fourier Transform of the input signal.
6	LowPassFIRFilter	`lowPassFIRFilter(input, coeffs) -> output`	$y_{\text{lpf}}[n] = \frac{w_c}{\pi} \cdot \text{sinc}\left(w_c\left(n - \frac{N-1}{2}\right)\right)$, for $n \neq \frac{N-1}{2}$ $y_{\text{lpf}}[n] = \frac{w_c}{\pi}$, for $n = \frac{N-1}{2}$	Applies a low-pass FIR filter to the input signal using the given coefficients.
7	LMSFilter	`lmsFilter(input, desired, mu) -> output`	$e[n] = d[n] - y[n]$ $y[n] = \sum_{i=0}^{M-1} w_i[n] x[n-i]$ $w_i[n+1] = w_i[n] + \mu e[n] x[n-i]$	Adaptive filtering using the Least Mean Squares algorithm.

Table 3.1: Sample List of DSP Dialect Operations

sible for organizing and executing passes at the different levels like dialect, module and function. There is also a specific kind of transformation aimed at simplifying operations (operation being the main unit of abstraction and transformation) called Operation Canonicalization in MLIR, which derives from the base pass manager. In this work, we utilize operation canonicalization to write our patterns/optimizations.

There are two mechanisms for defining canonicalizations on operations - Rewrite Patterns and Fold. Fold mechanism is powerful mechanism and powerful at simplying operations at constant input or known properties but don't allow new operation creation. Rewrite pattern in MLIR allows for different types of canonicalizations as well as new operation creation so we use this for defining our dialect specific patterns.

Patterns (Optimizations) in DSP dialect can be defined as replacing a group of

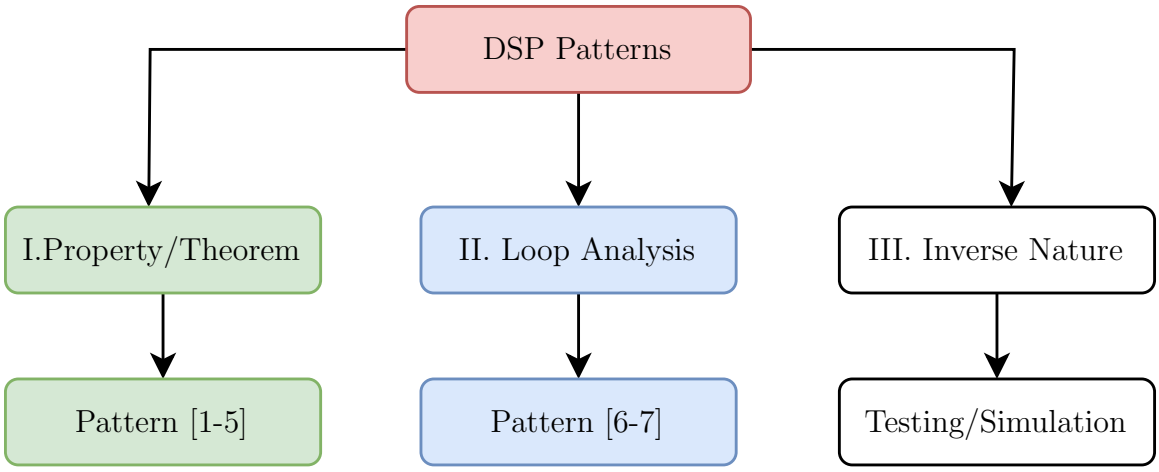

Figure 3.3: 3 Categorizations of DSP Patterns: Category I are the ones that are useful properties used in real world apps. Category II are loop analysis ones which aren't currently available with MLIR framework. Category III are the ones not useful in real world apps but useful for simulation purposes.

computationally expensive dsp operations with cheaper operations. These patterns can be categorized into three categorizes as shown in Fig. 3.3. The first two categorizes are useful in real world applications while the third category is only useful for simulation/testing purposes and we use them to test our operations as well. The first category directly comes from DSP-Domain theorems and properties and we will discuss it in details in the next section. The second category patterns are those which are currently not supported by affine optimizations and can be done by affine transformations in the future by complex analysis, but we do these quite easily at our dialect level. We will now dive in details of each of these optimizations :

Optimization 1 - Ideal filter and Cosine-Window Multiplication:

When designing a window-based filter, a common method is to multiply ideal low pass filter (3.1) with cosine-window (3.2) to obtain desired filter (3.3). As both of them are symmetric about mid-point, the multiplication will also be symmetric and hence we have the opportunity to reduce the number of calculations to half by just

Opt	Pattern Name	Base OP	Old Pattern	New Pattern
1	Symmetric Filter	MulOp	[IdealFilter, Hamming, Mul]	[FilterHammOpt]
2	Symmetric Filter Response	FilterRespOp	[FilterHammOpt, FilterResponse]	[FilterResSymmOpt]
3	Filter Response at Input and Reverse	FilterRespOp	[ReverseInput, FilterResponse]	[FilterYSymmOpt]
4	DFT Response at Symmetric Input	DFT1DRealOp	[FilterResponse, DFT1DReal]	[DFT1DRealSymmOp]
5	Parsevaal's Theorem	DivOp	[DFT1D, Square, Sum , Div]	[Square , Sum]
6	DFTReal and DFTImg Fusion	DFTImgOp	[DFTReal, DFTImg]	[DFT1D]
7	LMSFilter and Gain Fusion	GainOp	[LMSFilter, Gain]	[LMSFilterGainOpt]

Table 3.2: Pattern corresponding to DSP Optimizations: Here, we provide canonicalization pattern on *BaseOp* and match *OldPattern* and replace it with *NewPattern* . Example: For row 1 , when the operands for Mul is IdealFilter and HammingWindow , we replace the 3 operations (shown in col - *OldPattern*) with single operation FilterHammOpt (shown in col - *NewPattern*).

calculating the first as shown in the below equation (3.4). Rest half is just same as first half i.e., $h[n] = h[L-1-n]$.

$$y_{\text{lpf}}[n] = \begin{cases} \frac{\omega_c}{\pi} \cdot \text{sinc}\left(\omega_c \left(n - \frac{L-1}{2}\right)\right), & \text{if } n \neq \frac{L-1}{2} \\ \frac{\omega_c}{\pi}, & \text{if } n = \frac{L-1}{2} \end{cases} \tag{3.1}$$

$$ham[n] = 0.54 - 0.46 \cdot cos(\frac{2\pi n}{L-1}), 0 \leq n < L \tag{3.2}$$

$$h[n] = y_{lpf}[n] * ham[n], 0 \leq n < L \tag{3.3}$$

$$h[n] = h[L-1-n] = y_{lpf}[n] * ham[n], 0 \leq n < \frac{L+1}{2} \tag{3.4}$$

Optimization 2 - FilterResponse at Noisy signal and Symmetric filter:

Filter response operation can be defined as (3.5). When the filter is symmetric, ie, $h[i] = h[L-1-i]$ then we use the symmetry property on (3.7) to combine the beginning and end terms to obtain final (3.8). By using (3.8) , we reduce the number of load instructions for the filter and we get performance benefits.

$$y[n] = \sum_{i=0}^{L-1} h[i] \cdot x[n-i], 0 \leq n < N \tag{3.5}$$

$$\begin{aligned} &= h[0] \cdot x[n] + h[1] \cdot x[n-1] + \ldots + \\ &\quad h[L-2] \cdot x[n-(L-2)] + h[L-1] \cdot x[n-(L-1)] \end{aligned} \tag{3.6}$$

$$\begin{aligned} &= h[0]\{x[n] + x[n-(L-1)]\} + \\ &\quad h[1]\{x[n-1] + x[n-(L-2)]\} + \ldots \\ &\quad + h[\frac{L-1}{2}] \cdot x[n - \frac{L-1}{2}] \end{aligned} \tag{3.7}$$

$$\begin{aligned} &= \sum_{i=0}^{\frac{L-1}{2}} h[i] \cdot \{x[n-i] + x[n-(L-1-i)]\} \\ &\quad + h[\frac{L-1}{2}] \cdot x[n - \frac{L-1}{2}] \end{aligned} \tag{3.8}$$

Optimization 3 - FilterResponse/Conv1D Property at input and reverse input:

According to filter response property, we know that when the inputs are vector and reverse vector ie, $h[l] = x[-l] or, h[l] = x[L-1-l]$ then the output of filter response will be symmetric about its mid-point ie, $\frac{L+1}{2}$ so we check the pattern for the operands of filter response and if the inputs are reverse of each other, we calculate the output for first half only as shown in the below equation (3.10) and for the second half, we use $y[n] = y[N-1-n]$. Hence we achieve performance benefit by reducing the number of loop iterations.

$$y[n] = \sum_{i=0}^{L-1} h[i] \cdot x[n-i], 0 \leq n < N \tag{3.9}$$

$$y[n] = y[N-1-n] = \sum_{i=0}^{L-1} h[i] \cdot x[n-i], 0 \leq n < \frac{N+1}{2} \tag{3.10}$$

Optimization 4 - DFT Response at Symmetric Input:

According to DFT Property , when the input is real and symmetric , then the real part of DFT will be symmetric and the imaginary part will be conjugate symmetric as shown in (3.13) and (3.14) so this also gives an opportunity to reduce the outer loops to half. Here, the symmetry happens after the first element.

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}nk}, \quad k = 0, 1, 2, \ldots, N-1 \tag{3.11}$$

$$= X_{real}[k] + jX_{img}[k] \tag{3.12}$$

$$X_{real}[k] = X_{real}[N-k], \quad k = 1, 2, \ldots, \frac{N-1}{2} \tag{3.13}$$

$$X_{img}[k] = -X_{img}[N-k], \quad k = 1, 2, \ldots, \frac{N-1}{2} \tag{3.14}$$

Optimization 5 - Parsevaal's Theorem:

According to Parseeval's theorem as shown in (3.15), energy of a signal is equal in time as well as frequency domain. So, whenever there is a pattern in which energy is calculated in frequency domain, we can replace the same with that in time domain which means if we find a code in which there is sum of square of real and imag part of DFT is calculated (as shown in Table 3.2 (Opt 5)) , we replace it with sum of square of the input.

$$\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|^2 \tag{3.15}$$

Optimization 6 - Loop fusion for DFTReal and DFTImg Part:

Currently, with our DSL just supports returning a single result so we calculate DFTReal and DFTImg part separately according to (3.16) and (3.17) respectively. When the inputs to both the operations are same, affine loop fusion should fuse the operations into one but we observe it is unable to do so. Hence, we write our own pattern to fuse these operations into one , overall saving loop iterations.

$$X_{\text{real}}[k] = \sum_{n=0}^{N-1} x[n] \cos\left(\frac{2\pi}{N} kn\right), 0 \leq n < N \tag{3.16}$$

$$X_{\text{img}}[k] = -\sum_{n=0}^{N-1} x[n] \sin\left(\frac{2\pi}{N} kn\right), 0 \leq n < N \tag{3.17}$$

$$X[k] = X_{\text{real}}[k] + jX_{\text{img}}[k] \tag{3.18}$$

Optimization 7 - LMSFilter and Gain:

LMS Filter is an adaptive filter that aims to minimize the mean square error between the desired signal $d(n)$ and the output signal $y(n)$. The filter is widely used for applications such as hearing aid, noise cancelling and echo cancelling among others. LMS Filter can be represented as the following equations:

1. **Output Signal:**

$$y(n) = \mathbf{w}^T(n)\mathbf{x}(n) \tag{3.19}$$

where

- $\mathbf{w}(n) = [w_0(n), w_1(n), \ldots, w_{M-1}(n)]^T$ is the weight vector at time n.
- $\mathbf{x}(n) = [x(n), x(n-1), \ldots, x(n-M+1)]^T$ is the input vector at time n.
- M is the number of filter taps.

2. **Error Signal:**

$$e(n) = d(n) - y(n) \tag{3.20}$$

where

- $d(n)$ is the desired signal at time n.

3. **Weight Update:**

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu e(n)\mathbf{x}(n) \tag{3.21}$$

where

- μ is the step size (learning rate) parameter.

As seen from the above equations, the calculation of LMSFilter weights is a two-dimensional nested loop while gain operation is a one-dimensional loop computation. LMS and Gain blocks are used together for hearing aid application where the gain can be fused with LMSFilter as follows:

Weight Update with Gain:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu G e(n)\mathbf{x}(n) \tag{3.22}$$

where μ is the step size (learning rate) parameter and G is the gain applied to the error signal.

However traditional compilers such as affine optimization fails to see this opportunity which we exploit in our dialect.

Table 3.2 shows the patterns with dsp operations corresponding to each of these optimizations and the corresponding new pattern.

Here, first 5 patterns are the category one patterns directly coming from DSP Domain property/theorem. Pattern 6 and 7 are category two patterns. The category three patterns are the patterns like upsampling followed by downsampling , transforms followed by its inverse(like fft, ifft , dct, idct) which are mostly used for

testing and simulation of dsp operations but would have really difficult to do at C or lower-levels.

The DSP Dialect Lowering

Once the MLIR compatible IR is obtained, the next step is to define the lowering of operations defined in DSP-Dialect, which means the actual implementation of operations. For DSP operations, the implementation will require loop-based computations hence the choices in MLIR Framework are either affine dialect or scf dialect. Affine dialect in MLIR promises powerful polyhedral optimizations like loop fusion, ScalarReplacement hence we choose this dialect for lowering. But this dialect has certain restrictions like it requires constant value for loop-indices and symbols which is not applicable in certain scenarios like dynamic index value calculation, then the lowering is done to scf loop directly. For implementation, we need to have understanding of affine IR structure and syntax and the corresponding affine C++ API for our framework development. MLIR framework is still in its early phase and is emerging and the C++ APIs are evolving and hence the development and testing can be quite slow for the developers. For example, there are multiple ways to implement same functionality through C++ API for ex- there are api's for implementing for loop in affine like *affine::buildAffineLoopNest* and *affine::AffineForOp*. The second one *affine::AffineForOp* is more powerful in expressing affine loops and which one to choose needs well-documented programmer's guide which is currently lacking from MLIR side. Similarly, for binding affine references, one may either use *bindDims()* or the references can be obtained directly by *PatternRewriter.getAffineDimExpr()* and which one to use requires proper documentation from framework developers. Mlir is also evolving and certain feature usage like returning multiple value types from con-

trol block needs more complex analysis ie, *scf::IfOp* block using *TypeRange*{*floatType* , *indexType*} and we need to reiterate affine-IR for another simpler code and this can lead to longer development time. Overall, the development of mlir C++ code is harder, hence we need an approach for our lowering so that the development is smoother.

We developed an approach using multiple stages to make our development smoother as shown in Fig 3.4 where the stages are C-level, Affine-level IR and affine C++ code. The complexity level increases as we go down the stages hence we need to validate each stage so that our implementation is functionally correct. For each stage, we develop the code and compare the output with standard libraries like numpy/Matlab. The validation at each stage also makes our debugging easier which gets harder as we go to lower stages. The first stage is C-level code which is quite easy to develop and can be easily compiled & tested with corresponding dsp libraries. The second stage is Affine-level IR and it is C-like but with slight differences and restrictions so if our C-code can't be expressed as affine loop iterations , we again move to stage first and try another C-code resolving those restrictions like SSA restrictions, constant index etc. We try C-code iteration till we can express our c-code in terms of Affine-IR then this Affine-level IR can be tested with mlir-opt which is available from MLIR framework and validated against desired output. Once the output of this stage matches with that of standard libraries, we can go for our final C++ development which is hardest (as C++ API is very dynamic and can change a lot) but this is needed for our framework development. In this stage, we check the existing api (through corresponding C++ definition) and get the corresponding usage (through MLIR source code APIs and other existing lowering) and develop the proper C++ code and use `mlir->dump()` for matching against second-stage output. This final stage output can be obtained by emitting affine-IR from the our framework and we match against the output of

std library (or, the second stage Affine-IR not shown in the fig 3.4 for simplicity) .

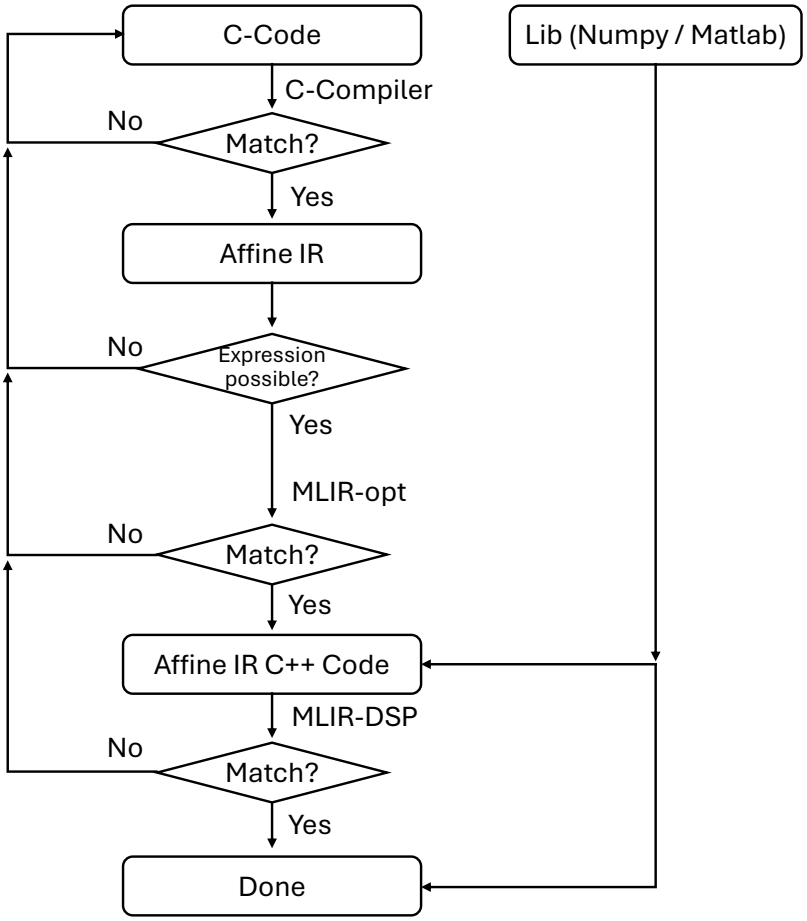

Figure 3.4: Approach for Developing Mlir C++ Lowering : There are 3 stages- C-code (easiest), Affine IR and finally C++ code (hardest). We develop C-code , match against std library , then develop Affine IR , and if IR can't be expressed with C-code, go back to developing C-code. Once IR output matches with std library, we move to final stage of C++ code development until output matches.

Once the lowering is done, the rest of the lowering is done using built-in MLIR lowering passes. MLIR framework provides lowering from affine/scf loops to mlir-llvm (a dialect in MLIR). We use this and then there is a conversion pass which translates the mlir-llvm IR to LLVM IR. Once this LLVM IR is obtained , clang tool is used to convert this LLVM IR into executable as shown in the fig 4. This executable is run and validated against desired output.

The DSP Domain-Specific Language

Using the lowering and clang-17, DSP applications can be developed and tested but in MLIR, dialects have to be written at MLIR IR level and is not user-friendly as shown in Fig 3.2 and hence affects productivity from a programming perspective. Hence, we introduce a simple user interface language, DSP DSL to ease the development. The grammar for DSP DSL Language consists of simple syntax rules [like statement ending with semicolon] and tokens [like keywords(def , main, var), arithmetic operators(+,-,*,/), braces , square bracket for tensors]. An example application for calculating power spectral density of a signal is shown in Fig 3.2 - right part. As demonstrated in the Fig 3.2, this code is simple but robust enough to ease the programming and testing of DSP dialect operations. The conversion steps are usual as any standard compiler stages :a) Lexer produces the token/symbols which are then used by recursive Parser to generate AST (Abstract Syntax Tree). Parser parses the module (source file) , which is made of functions and function is made of statements and produces moduleAST. Then, there is another Parser (mlirGen class) which takes the moduleAST as input and generates corresponding operation based on statement types from DSL AST which will generate the final MLIR IR as an output, which can be easily fed into MLIR system for further processing/analysis.

Chapter 4

EVALUATION SETUP

SN	AppName	DSP Ops Sequence	Optimizations	Opt
1	Filter Design	[Ideal LowPass Filter → Hamming → Mul]	[Symmetric Filter]	[1]
2	LowPass Filtering	[Input + Noise → Filter Design → Filter Response]	[Symmetric Filter and FilterResponse]	[1,2]
3	Energy of Signal	[Input → DFT → Square → Sum]	[Parsevaal's Theorem]	[5]
4	Spectral Analysis	[Input, Reverse → Conv1D → DFT → Square]	[Filter Response and DFT Symmetric]	[3,4]
5	Audio Compression	[Input → DFT → Threshold → Quantization → RunLenEncoding]	[DFT Loop Fusion]	[6]
6	Hearing Aid	[Input, Ideal LMS → LMSFilter → Gain → LMSFilter Response]	[LMSFilter and Gain Fusion]	[7]
7	Audio Equalizer	[Input, LowPass, BandPass, HighPass → Gain for Bands → FilterResponse → Sum]	[Symmetric Filter and FilterResponse]	[1,2]

Table 4.1: Sample DSP Apps and sequence of the DSP operations in the app and the optimizations applied to them.

We evaluate DSP-MLIR framework on sample DSP Apps (here, an application is input, set of DSP blocks/operations and output) which are representative of Audio signal processing but can be easily applied to other digital signal processing as well.

The test applications were executed on a machine having AMD Ryzen 5700u , 8 physical cores, 16 Logical Cores, 16 GB RAM. Code version of LLVM base was 19.0.0git and version of clang was 17.0.6. We used two criterias for the evaluation - i) Functional Correctness, ii) Execution Time Measurement. For the first criteria ie, functional correctness, we developed similar application in numpy and since it had graphical support, we plotted the graph to make sure the desired output was

as expected (ie, not only the output numbers matching but the actual application producing the desired output For example - the output signal having no ripples for low-pass filtering application). Wherever the plots weren't applicable, we matched with the corresponding numpy output. This helped in verifying the correct sequence of the dsp operations (and the individual operations were also tested), and hence correctness of the application.

For the second criteria of measuring the execution time , we had to compare our optimizations against C/affine-level optimizations (C/affine-level is where traditional compilers provide most optimizations) for which we enabled affine-level optimizations. For measuring the execution time of our optimizations, we also used the following methods to obtain our time - a) Took average of 5 iterations to take out system effect, b) Flushed out cache after any run so that we see same effect for every run , c) we printed out single element at random index (not the whole vector which would be an expensive operation and would increase the total time of application dramatically).

Chapter 5

RESULTS

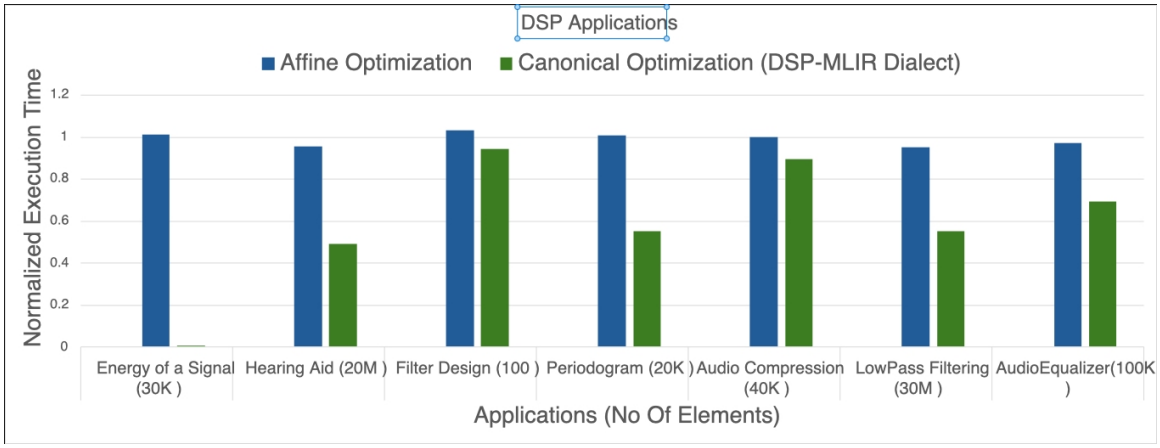

Figure 5.1: Normalized Performance with canonical optimizations in DSP-MLIR

In the evaluation, we ran three versions of the code - 1) without any optimization, 2) with affine optimization (AffineLoopFusion and AffineScalarReplacement) and 3) lastly with affine plus our dsp-specific optimizations for all the test apps. Table 4.1 lists all the test apps and the optimizations applied on them. Fig 5.1 shows the normalized execution time versus the input size for all the applications. Here, the baseline is No Optimization (execution time = 1) and the blue and green bars are affine and affine + dsp specific optimizations respectively.

Our DSP lowerings and Optimizations yield much better-performing code

As evident from the figure 5.1, utilizing the dsp domain specific properties/theorems are much easier at dsp dialect level which yields much better performing code. This is because we are able to exploit the dsp theorem level optimizations at our dialect level which is not possible at any other lowering, performing these canonical optimizations

yields around 2x performance improvement as compared to Affine optimization.

DSP DSL reduces programming complexity with respect to C-code

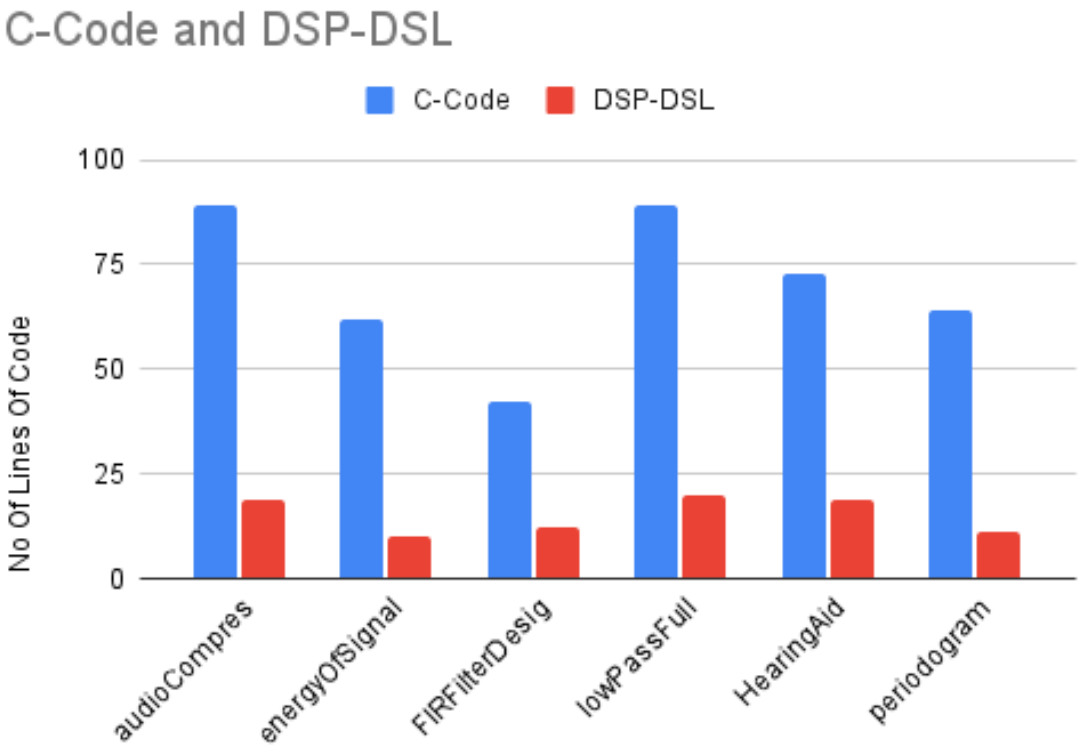

Figure 5.2: Lines of Code C-Code vs Our DSL for Sample Apps

By utilizing the DSP dialect, it is easier for a signal processing engineer or compiler developer to develop and utilize DSP operations either by directly using the DSP dialect or the DSL provided within this framework. As seen from 5.2, while it may take an average of 3.5x more lines of code to write such DSP applications in C language, it is much easier and faster to implement DSP applications in our DSL.

DSP Domain-Specific optimizations are easier to perform at domain (high-level)

One of the major contributions of this research is the high-level optimizations that exploit DSP theorems and patterns to compile efficiently. These optimizations

are best implemented at much higher abstraction, like a DSP dialect in MLIR. For example, for implementing the symmetric filter property at C-level in Table 3.2 (Opt 1), this would require complex polyhedral (mathematical abstractions) analysis which would lead to complex code implementations while we do this simply by utilizing the domain knowledge about the filters. Even for loop fusion Table 3.2 (Opt 6) , we were able to do this quite easily while affine dialect - *AffineLoopFusion* and *AffineScalarReplacement* optimizations were not able to fuse this double nested loops.

Chapter 6

CONCLUSION AND FUTURE WORK

In this work, we have developed a Domain specific compiler for DSP in MLIR and utilized the multi-level IR property to specify domain specific optimizations which was difficult to represent at lower-level IR (C/assembly). We ran these optimizations on some sample applications and obtained significant performance improvement using these optimizations with respect to affine-level optimizations.

Future Work: Existing DSP Compilers like Matlab, Scilab have graphical interfaces (like graph, 3D plots) which is quite useful in signal processing domain. Our compiler being modular can be easily integrated as the backend for these by just providing the corresponding language AST to MLIR conversion. Another interesting work will be integrating with Deep Learning blocks through available dialects like torch-mlir and look for inter-dialect optimizations and also apply in real-world applications.

REFERENCES

Axelsson, E., K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson and A. Vajdax, "Feldspar: A domain specific language for digital signal processing algorithms", in "Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)", pp. 169–178 (IEEE, 2010).

CIRCT, URL `https://circt.llvm.org/` (2024).

de Dinechin, B. D., F. de Ferri, C. Guillon and A. Stoutchinin, "Code generator optimizations for the st120 dsp-mcu core", in "Proceedings of the 2000 International Conference on Compilers, architecture, and synthesis for embedded systems", pp. 93–102 (2000).

Elrajoubi, A., S. S. Ang and A. Abushaiba, "Tms320f28335 dsp programming using matlab simulink embedded coder: Techniques and advancements", in "2017 IEEE 18th Workshop on Control and Modeling for Power Electronics (COMPEL)", pp. 1–7 (IEEE, 2017).

Franke, B. and M. F. O'Boyle, "Compiler parallelization of c programs for multi-core dsps with multiple address spaces", in "Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis", pp. 219–224 (2003).

GNU, URL `https://www.gnu.org/home.en.html` (2023).

Holton, T., *Digital Signal Processing: Principles and applications* (Cambridge University Press, 2021).

Hu, P., M. Lu, L. Wang and G. Jiang, "Tpu-mlir: A compiler for tpu using mlir", arXiv preprint arXiv:2210.15016 (2022).

Jin, T., G.-T. Bercea, T. D. Le, T. Chen, G. Su, H. Imai, Y. Negishi, A. Leu, K. O'Brien, K. Kawachiya *et al.*, "Compiling onnx neural network models using mlir", arXiv preprint arXiv:2008.08272 (2020).

Lattner, C. and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation", in "International symposium on code generation and optimization, 2004. CGO 2004.", pp. 75–86 (IEEE, 2004).

Lattner, C., M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law", arXiv preprint arXiv:2002.11054 (2020).

Letz, S., Y. Orlarey and D. Fober, "Faust domain specific audio dsp language compiled to webassembly", in "Companion Proceedings of the The Web Conference 2018", pp. 701–709 (2018).

Leventhal, S., L. Yuan, N. K. Bambha, S. S. Bhattacharyya and G. Qu, "Dsp address optimization using evolutionary algorithms", in "Proceedings of the 2005 workshop on Software and compilers for embedded systems", pp. 91–98 (2005).

LLVM, "Torch-mlir", URL `https://github.com/llvm/torch-mlir` (2024).

Lorenz, M., P. Marwedel, T. Drager, G. Fettweis and R. Leupers, "Compiler based exploration of dsp energy savings by simd operations", in "ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No. 04EX753)", pp. 839–842 (IEEE, 2004).

Martínez, P. A., G. Bernabé and J. M. García, "Hdnn: a cross-platform mlir dialect for deep neural networks", The Journal of Supercomputing **78**, 11, 13814–13830 (2022).

McCaskey, A. and T. Nguyen, "A mlir dialect for quantum assembly languages", in "2021 IEEE International Conference on Quantum Computing and Engineering (QCE)", pp. 255–264 (IEEE, 2021).

Rasool, N. and J. I. Bhat, "Brain tumour detection using machine and deep learning: a systematic review", Multimedia Tools and Applications pp. 1–54 (2024).

Thomas, S. and J. Bornholt, "Automatic generation of vectorizing compilers for customizable digital signal processors", in "Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1", pp. 19–34 (2024).

Wikipedia contributors, "Digital signal processing — Wikipedia, the free encyclopedia", `https://en.wikipedia.org/w/index.php?title=DigitalSignalProcessing&oldid=1233991896`, [Online; accessed 28-July-2024] (2024).

Zivojnovic, V., S. Pees, C. Schlager, M. Willems, R. Schoenen and H. Meyr, "Dsp processor/compiler co-design: a quantitative approach", in "Proceedings of 9th International Symposium on Systems Synthesis", pp. 108–113 (IEEE, 1996).