DSP-MLIR: A Domain-Specific Language and MLIR Dialect for Digital Signal Processing

Abhinav Kumar[*]
Arizona State University
Tempe, USA
akuma294@asu.edu

Atharva Khedkar[*]
Arizona State University
Tempe, USA
apkhedka@asu.edu

Hwisoo So
Yonsei University
Seoul, Republic of Korea
Arizona State University
Tempe, USA
shs7719@yonsei.ac.kr

Megan Kuo
Arizona State University
Tempe, USA
megankuo@asu.edu

Ameya Gurjar
Arizona State University
Tempe, USA
agurjar2@asu.edu

Partha Biswas
MathWorks®
Natick, USA
pbiswas@mathworks.com

Aviral Shrivastava
Arizona State University
Tempe, USA
aviral.shrivastava@asu.eu

Abstract

High-quality compilation of Digital Signal Processing (DSP) algorithms is crucial for achieving real-time performance and optimizing resource utilization. Traditional compilers often struggle to effectively optimize DSP applications since their optimization passes mainly deal with low-level intermediate representations. This paper introduces DSP-MLIR – a comprehensive framework for DSP application development and optimization. DSP-MLIR comprises i) a Python-like domain-specific language (DSL) (named DSP-DSL) for intuitive and easier programming of DSP applications, ii) a dedicated MLIR dialect (named DSP-dialect) with 90+ operations and 16 optimizations at the level of DSP operations, and iii) lowerings to the Affine and standard MLIR dialects for high-quality compilation flow for DSP applications. The effectiveness of the proposed DSP-MLIR is evaluated by comparing the runtimes of the binaries generated by the various compilation flows, including GCC, Clang, Hexagon-Clang, and existing MLIR passes. Experiments on 20 DSP applications collected from various sources demonstrate an average performance improvement of 12% over state-of-the-art compilation flows with a 10% reduction in the generated binary size and no significant variation in compilation time. Further, expressing DSP applications in the proposed DSP-DSL reduces the code complexity and development time of DSP applications (as measured in lines of code (LOC)) by an average of 5x over their specification in the programming language, "C".

The DSP-MLIR framework is open-source and available at: https://github.com/MPSLab-ASU/DSP_MLIR

CCS Concepts: • **Software and its engineering** → **Compilers**; *Domain specific languages*; • **Hardware** → *Emerging languages and compilers*; **Digital signal processing**.

Keywords: Compiler, MLIR, DSP, LLVM

ACM Reference Format:
Abhinav Kumar, Atharva Khedkar, Hwisoo So, Megan Kuo, Ameya Gurjar, Partha Biswas, and Aviral Shrivastava. 2025. DSP-MLIR: A Domain-Specific Language and MLIR Dialect for Digital Signal Processing. In *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '25), June 16–17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3735452.3735527

[*]Both authors contributed equally to this research.

1 Introduction

Digital Signal Processing (DSP) applications leverage computational algorithms to manipulate real-world signals such as audio, video, and sensor data. These techniques enable improvements in quality, noise reduction, and efficient transmission across domains like telecommunications, medical imaging, and consumer electronics [33, 48]. DSP is widely deployed in systems like microphones, audio amplifiers, and speech assistants (e.g., Google Assistant, Amazon Echo).

Because many DSP algorithms operate under real-time constraints and on resource-limited embedded devices (e.g., microphones, mobile SoCs), compilation efficiency is critical. Dedicated DSP processors such as the *TI C6000* series [17], *ADI Blackfin/SHARC* [1], and *Qualcomm Hexagon DSPs* are optimized for such tasks. These vendors provide specialized compilers and libraries—e.g., TI's *TI_CGT* compiler with *DSPLIB* [3, 13], and Qualcomm's *QHL* (Qualcomm Hexagon Libraries) [5]—to exploit hardware capabilities and accelerate DSP development. These libraries contain C-callable, pre-optimized kernels for common DSP operations and can significantly reduce development time.

However, most traditional compilers—including GCC [4] and Clang [14, 15]—rely on a single, low-level intermediate representation (IR), which makes it difficult to detect and optimize high-level computational patterns. For instance, a common DSP technique, the *window method* for FIR filter design, multiplies the impulse response of an ideal filter with a Hamming window to yield a realizable symmetric FIR filter. This is used in applications such as speech enhancement, seismic data filtering, and audio equalization. As shown in entry #1 of Table 2, this pattern is defined by:

$$h[n] = h_{\text{ideal}}[n] \cdot w[n]$$

Since both $h_{\text{ideal}}[n]$ and $w[n]$ are symmetric, $h[n]$ is also symmetric, enabling optimizations that compute only half the values and mirror the result. Traditional DSP libraries, however, operate at the level of individual kernels and lower to IR before optimizations, making such *cross-operation* optimizations infeasible. For example, QHL may treat the ideal filter and window operations independently, missing opportunities to eliminate redundant computation by leveraging their shared symmetry.

DSP-MLIR, the compiler framework proposed in this work, overcomes these limitations by identifying and applying such high-level domain-specific optimizations. This optimization, among others, is detailed in Section 3.

The primary contributions of DSP-MLIR are:

1) DSP-dialect in MLIR: We introduce the **DSP-dialect** with 90+ operations covering commonly used DSP constructs. These operations are progressively lowered to the `affine` and `standard` MLIR dialects to enable optimization and code generation via LLVM.

2) DSP-specific Optimizations: We implement 16 optimizations based on DSP theorems and operation fusion. These are expressed at the domain level and often cannot be captured by general-purpose compiler passes.

3) DSP-DSL for Rapid Development: A Python-like domain-specific language enables intuitive specification of DSP algorithms. Code written in DSP-DSL is automatically lowered to DSP-dialect, optimized, and compiled to executable binaries via MLIR and Clang.

To evaluate DSP-MLIR, we use 20 DSP applications from benchmarks such as EEMBC [27], MiBench [32], and DSP-Stone [51]. Without DSP-dialect optimizations, DSP-MLIR achieves a 43% speedup over GCC and 27% over Clang on CPUs. With DSP-dialect optimizations, an additional 10% gain is observed. On the Hexagon v68 DSP processor, DSP-MLIR initially lags Clang by 8%, but surpasses it by 5% after applying optimizations, while also reducing binary size and keeping compile time low. These gains come exclusively from DSP-dialect-level transformations (Section 3.2), without altering the backend code generation.

In terms of developer productivity, DSP-DSL reduces code size significantly—requiring 5× fewer lines of code on average compared to C. The DSP-MLIR framework is portable across targets via MLIR and LLVM, and is featured on the official MLIR website [7, 44].

2 Background on MLIR (Multi-Level Intermediate Representation)

Traditional compiler infrastructures such as GCC and LLVM operate on a monolithic, single-level intermediate representation (IR), limiting their ability to express and optimize programs across diverse levels of abstraction. To address this, **MLIR** (Multi-Level Intermediate Representation) framework [8] was introduced to provide a flexible and extensible compilation infrastructure that supports multiple IRs across different abstraction levels through the use of *dialects*.

Each dialect in MLIR defines a namespace encapsulating a set of operations, types, and attributes that represent computations at an abstraction level. Compilation in MLIR proceeds via *progressive lowering*, wherein a program is transformed from high-level dialects (close to the source language) to low-level dialects (closer to the hardware), enabling targeted optimizations at each stage of the transformation pipeline.

MLIR includes a wide range of dialects tailored to specific domains and hardware targets. For example, `tf`[1] [43] and `onnx-mlir` [11] dialects support TensorFlow and ONNX frontends for machine learning models; `affine` [2] dialect supports polyhedral optimizations for loop nests; `nvgpu` [9] and `nvvm` [10] enable targeting NVIDIA GPUs; `scaleHLS` [49, 50] and `SODA` [20–22] are designed for high-level synthesis; `FHE` [35] and `HECO` [47] support compilation for fully homomorphic encryption; and `mlir-cgra` [42] targets coarse-grain reconfigurable architectures (CGRAs).

A major strength of MLIR lies in its ability to perform powerful optimizations in intermediate dialects. For instance, the `LinAlg` [6] and `affine` [2] dialects provide rich semantics for tensor algebra and loop transformations, respectively. This allows the compiler to detect and optimize patterns such as redundant tensor transpositions (e.g., eliminating

[1]Any text in `this format` refers to an MLIR keyword

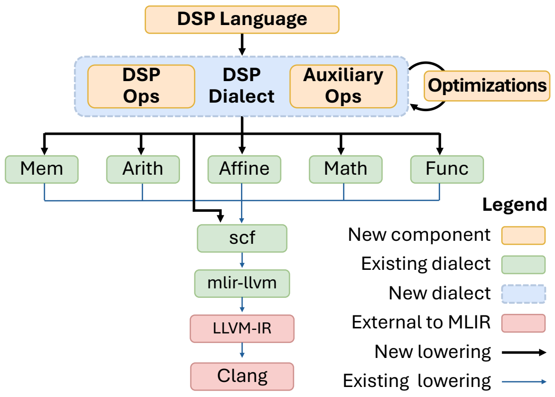


Figure 1. The compilation pipeline of the proposed DSP-MLIR framework. DSP-DSL is compiled into the DSP-dialect and then lowered to `affine` (whenever possible) or `structured control flow` (SCF) and then to *LLVM IR*. Finally, code is generated through Clang.

double transposes), enabling both high-level and low-level performance improvements.

MLIR's modular and reusable infrastructure has led to widespread adoption in modern compilation pipelines, including those for PyTorch (`torch-mlir` [39]) and TensorFlow (`tf` dialect [43]). In this work, we introduce a new MLIR dialect for easier expression and efficient compilation of DSP applications.

3 DSP-MLIR Framework

Figure 1 illustrates the compilation flow of the proposed DSP-MLIR framework. Application developers begin by writing their programs in a domain-specific language (DSP-DSL) tailored for DSP workloads. This high-level specification is then translated into the `DSP` dialect—a high-level MLIR dialect designed to closely reflect the semantics of the DSL.

This translation involves representing the application in terms of `DSP` dialect operations and data structures. Once in this form, dialect-specific optimizations are applied to improve the program's structure and performance at the high level. After these initial optimizations, the IR is progressively lowered to more concrete representations.

Where possible, the intermediate representation (IR) is transformed into the `affine` dialect to leverage MLIR's powerful polyhedral analysis and loop optimization capabilities. However, due to the static nature of the `affine` dialect—which only supports loop nests with constant bounds and strides—portions of the application that cannot be expressed in this form are lowered into other standard MLIR dialects such as `memref`, `arith`, `math`, `func`, and `scf`.

Through this staged lowering process, the IR is eventually converted into the `LLVM` dialect (`MLIR-LLVM`), which provides a direct mapping to the LLVM IR. Finally, the LLVM backend, typically invoked via `clang`, is used to generate the target machine binary. This structured and modular compilation flow enables both high-level expressiveness and low-level performance tuning for DSP applications.

Table 1. Some of the DSP-dialect operations. A full list of **90+** operations is available in [26].

Operation name	Equation
SlidingWindowAvg	$y[n] = \frac{1}{N} \sum_{i=0}^{N-1} x[n-i]$
FIRFilterResponse	$y[n] = \sum_{i=0}^{M} h[i] \cdot x[n-i]$
MedianFilter	$y[n] = median(x[n-1], x[n], x[n-1])$
FFT1DReal	$X_{\text{real}}[k] = \sum_{n=0}^{N-1} x[n] \cos\left(\frac{2\pi}{N} kn\right)$
FFT1DImg	$X_{\text{imag}}[k] = -\sum_{n=0}^{N-1} x[n] \sin\left(\frac{2\pi}{N} kn\right)$

3.1 The DSP-dialect

The DSP-dialect offers a comprehensive set of operations (90+) tailored for expressing digital signal processing (DSP) applications. A few of the supported DSP-dialect operations and their mathematical representations are presented in Table 1. The full list of the DSP-dialect operations is available in [26]. DSP-dialect operations are categorized into two primary groups: *Core DSP operations* and *Auxiliary operations*.

Core DSP operations form the backbone of many DSP algorithms. They include signal processing primitives like delay, transforms such as DFT, IDFT, and DCT, and various filter designs, including low-pass, high-pass, band-pass, and band-stop filters. These operations are fundamental building blocks for constructing sophisticated DSP systems.

Auxiliary operations complement the *Core DSP operations*, providing essential functionalities for efficient implementation. These include helper functions like trigonometric functions, vector generation, and printing for debugging purposes. Additionally, auxiliary operations facilitate data type conversion between different representations, enabling seamless integration of various DSP components.

The DSP-dialect leverages tensor data type to represent DSP signals effectively. The tensor data type can accommodate a wide range of signals, from one-dimensional signals to multi-dimensional signals and complex-valued signals. By employing the tensor data type, the dialect offers a unified and efficient representation for diverse DSP applications.

3.2 DSP-dialect Optimizations

MLIR's hierarchical structure provides a unique opportunity to identify and apply high-level optimizations that are very hard to implement in single-IR compilers. We broadly categorize our optimizations into 2 categories: 1) DSP theorem-based - which contains optimizations based on well-known DSP theorems and mathematical identities. 2) DSP operation fusion-based - which contains optimizations that fuse operations/loops that other MLIR passes (and/or compilers)

Table 2. A complete list of DSP-dialect optimizations. Some of them are based on well-known DSP theorems and mathematical identities, and others fuse inter-kernel/operation loops that other compiler passes are unable to optimize.

#	Canonical Pattern	Optimized Pattern	Category
1	Mul(IdealFilter(wc, n), Hamming(n))	FilterHammOpt(wc, n)	
2	FilterResponse(FilterHammOpt(wc, n))	FilterResSymmOpt(wc, n)	
3	Div(Sum(Add(Sq(DFTReal(ins)), sq(DFTImg(ins)))), len(ins))	Div(Sum(Sq(ins)), len(ins))	DSP Theorem
4	IFFT1D(Mul(FFT1D(Padding(ins1)), FFT1D(Padding(ins2))))	FIRFilterResponse(ins2, ins1)	
5	FilterResponse(ins, ReverseInput(ins))	FilterResponseYSymmOpt(ins)	
6	DFT1DReal(FilterResponseYSymmOpt(ins))	DFT1DRealSymmOpt(ins)	
7	SlidingWindowAvg(Median(ins))	Median2SlidingOpt(ins)	
8	Threshold(FilterResSymmOpt(wc, n), ths)	FilterResSymmThresholdOpt(wc, n, ths)	
9	Normalize(LMSFilterResponse(ins))	NormalizedLMSFilterResponseOpt(ins)	
10	FindPeaks(LMSFilterResponse(ins), h, dist)	LMS2FindPeaks(ins, h, dist)	
11	Mean(Diff(FindPeaks(in, h, dist), len(peaks)), len(peaks)-1)	FindPeaks2DiffMeanOpt(in, h, dist)	Operation Fusion
12	Gain(LMSFilter(ins))	LMSFilterGainOpt(ins)	
13	Max(Correlation(ins))	Corr2MaxOpt(ins)	
14	Sq(Sum(Sq(DFTReal(ins)), sq(DFTImg(ins))))	DFTAbsOpt(ins)	
15	Threshold(DFTAbsOpt(ins))	DFTAbsThresholdOpt(ins)	
16	DFTReal(ins), DFTImg(ins)	DFT1D(ins)	

fail to optimize. Overall, DSP-dialect provides 16 optimizations, listed in Table 2. The column Canonical Pattern represents the pattern of operations and dependency identified by our compiler and Optimized Pattern showcases the converted/optimized operation(s).

DSP-MLIR leverages DSP theorems to optimize various operations, such as combining ideal filters and cosine windows, exploiting properties of symmetric filters and noisy signals, optimizing energy calculations in time and frequency domains, reducing computations for symmetric inputs, optimizing filter response for input patterns, and accelerating convolution operations using the convolution theorem.

In addition to theorem-based optimizations, our compiler focuses on fusing operations to reduce computational overhead. This includes combining filtering and thresholding, merging sliding window and median filtering, combining filtering and normalization, fusing filter response calculation and peak detection, optimizing peak finding algorithms, combining LMS filter and gain computation, optimizing spectrogram calculation, combining spectrogram calculation and thresholding, optimizing correlation calculations, and fusing real and imaginary parts of DFT calculations.

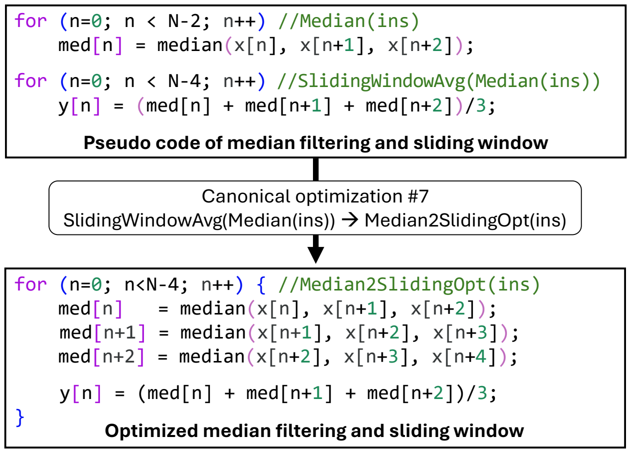
To implement these optimizations, we utilize the canonicalization approach in MLIR [12]. Canonicalization allows dialect developers to express program transformations in terms of pattern matching. MLIR uses a powerful pattern-matching system to identify specified patterns within the IR. Once a pattern is recognized, it is replaced with a more simplified or optimized equivalent. This allows for the elimination of redundant operations, simplification of complex expressions, and algebraic transformations. Next, we explain one optimization of each of the DSP theorem-based and DSP operation fusion-based optimizations.

Example Optimization Based on DSP Theorem - Filter Response for Symmetric FIR Filter. A typical filter response operation can be defined by Equation (1). Considering a symmetric FIR filter $h[n]$, where $h[i] = h[L-1-i]$, and assuming an odd number of filter coefficients, i.e., ($L \bmod 2 = 1$) we can exploit the symmetry property in Equation (1) to combine the symmetric inputs (equidistant from the center) and obtain final Equation (3). By using Equation (3), we can reduce the number of load instructions for the filter by almost half [33].

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

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

$$\begin{aligned} &= \sum_{i=0}^{\frac{L-3}{2}} h[i] \cdot \{x[n-i] + x[n-(L-1-i)]\} \\ &\quad + h[\frac{L-1}{2}] \cdot x[n - \frac{L-1}{2}] \end{aligned} \tag{3}$$



```
for (n=0; n < N-2; n++) //Median(ins)
    med[n] = median(x[n], x[n+1], x[n+2]);

for (n=0; n < N-4; n++) //SlidingWindowAvg(Median(ins))
    y[n] = (med[n] + med[n+1] + med[n+2])/3;
```

Pseudo code of median filtering and sliding window

Canonical optimization #7
SlidingWindowAvg(Median(ins)) → Median2SlidingOpt(ins)

```
for (n=0; n<N-4; n++) { //Median2SlidingOpt(ins)
    med[n]   = median(x[n], x[n+1], x[n+2]);
    med[n+1] = median(x[n+1], x[n+2], x[n+3]);
    med[n+2] = median(x[n+2], x[n+3], x[n+4]);

    y[n] = (med[n] + med[n+1] + med[n+2])/3;
}
```

Optimized median filtering and sliding window

Figure 2. Operation Fusion optimization of median filtering and sliding window average

Example Optimization Based on DSP Operation Fusion - Combining Median Filtering and Sliding Window Operation. Sliding window and median filtering are generally used together for applications like signal smoothing. For the same window sizes, this computation can be combined into a single filter and significantly reduce the computation overhead. For example, for a window size of 3, we load 5 elements to calculate the sliding window average as given in Figure 2. In this figure, $y[n]$ is the output signal, $x[n]$ is the input signal and n is the current sample index. This combined filter can be effective for noise reduction while preserving edges, as the average filter smooths out high-frequency noise, and the median filter helps to remove outliers and preserve edges. By itself, the loop fusion optimizations are unable to achieve this since the loop bounds for the following filter changes to $N - l + 1$ where N is the input length, l is the filter length, which affine loop fusion optimization is unable to recognize and fuse.

3.3 The DSP-DSL

DSP applications have traditionally been developed using C/C++ or library-based DSLs such as MATLAB® [34]. However, these approaches are often verbose, hard to maintain, and not well-suited for modern compiler infrastructures. For example, Figure 3 shows how one might implement a power spectrum computation in C.

In the absence of a high-level DSL, developers targeting our DSP dialect in MLIR would need to write code directly in MLIR. Figure 4 (right) illustrates this by showing the MLIR representation of a function that computes the power spectrum of a signal.

To address these limitations and improve programmability, we introduce **DSP-DSL** — a Python-like domain-specific language tailored for DSP workloads. DSP-DSL provides a rich set of built-in signal processing operations and offers an

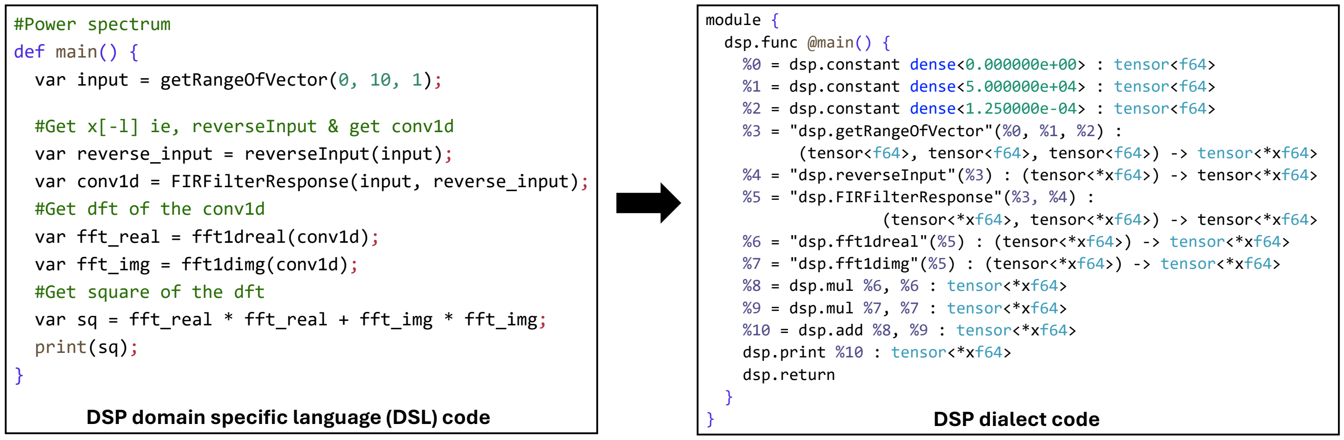
```
#include<stdio.h>
#include<math.h>
#define PI 3.14159265358

void main() {
    double input[10], reverse_input[10];
    for (int i = 0; i < 10; i++) //getRangeOfVector
        input[i] = 0.0 + i * 1.0;
    for (int i = 0; i < 10; i++) //reverseInput
        reverse_input[i] = input[10 - 1 - i];
    int conv_length = 2 * 10 - 1;
    double conv1d[conv_length];
    for (int n = 0; n < conv_length; n++) { //FIRFilterResponse
        conv1d[n] = 0;
        for (int k = 0; k < 10; k++)
            if (n - k >= 0 && n - k < 10)
                conv1d[n] += input[n - k] * reverse_input[k];
    }

    double fft_real[conv_length], fft_img[conv_length];
    for (int k = 0; k < conv_length; k++) { //dftReal
        fft_real[k] = 0;
        for (int n = 0; n < conv_length; n++) {
            double angle = 2.0 * PI * k * n / conv_length;
            fft_real[k] += conv1d[n] * cos(angle);
        }
    }

     for (int k = 0; k < conv_length; k++) { //dftImag
         fft_img[k] = 0;
         for (int n = 0; n < conv_length; n++) {
             double angle = 2.0 * PI * k * n / conv_length;
             fft_img[k] -= conv1d[n] * sin(angle);
         }
     }

    double sq[conv_length];
    for (int i = 0; i < conv_length; i++)
        sq[i] = fft_real[i]*fft_real[i] + fft_img[i]*fft_img[i];

    for (int i = 0; i < conv_length; i++) //print
        printf("%f ", sq[i]);
}
```

Figure 3. Function to calculate power spectrum of a signal in C language

intuitive syntax that enables developers to express complex DSP computations in just a few lines of code. The left side of Figure 4 demonstrates the same power spectrum function written in DSP-DSL. The grammar of the DSP-DSL is an extension of the language used in the MLIR tutorial, and its formal specification is presented in Extended Backus–Naur Form (EBNF) in Figure 5.

The DSP-DSL compiler performs a multi-stage translation to produce MLIR code in the DSP dialect. First, a lexer converts the DSL source into a stream of tokens. These tokens are then consumed by a recursive descent parser to construct an Abstract Syntax Tree (AST), where each module is parsed into a `moduleAST` comprising functions and statements. Next, a second parser, implemented in the `mlirGen` class, takes the `moduleAST` and generates corresponding MLIR operations based on the structure and semantics of each DSL construct.

This compiler pipeline results in MLIR code that integrates seamlessly into the broader MLIR ecosystem for analysis, optimization, and code generation.



```
#Power spectrum
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);
}
```

```
module {
  dsp.func @main() {
    %0 = dsp.constant dense<0.000000e+00> : tensor<f64>
    %1 = dsp.constant dense<5.000000e+04> : tensor<f64>
    %2 = dsp.constant dense<1.250000e-04> : 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
  }
}
```

Figure 4. A function to calculate the power spectrum of a signal represented in DSP-DSL (left) and DSP-dialect in MLIR (right). DSP-DSL allows DSP application developers to write their applications in intuitive syntax and high-level abstractions of DSP-DSL, rather than in the MLIR in DSP-dialect (right).

```
program = { function-def } ;
function-def = "def" id "(" ")" "{" { statement } "}" ;
statement = variable-declaration | assignment | function-call |
            print-statement | comment ;
variable-declaration = "var" id [ dim-specifier ] "=" expression ";" ;
assignment = id "=" expression ";" ;
print-statement = "print" "(" expression ")" ";" ;
comment = "#" { any-character-except-newline } newline ;
dim-specifier = "<" integer-literal { "," integer-literal } ">" ;
expression = term { ( "+" | "-" ) term } | array-literal ;
term = factor { ( "*" | "/" ) factor } ;
factor = id | number-literal | "(" expression ")" | function-call ;
function-call = id "(" [ argument-list ] ")" ;
argument-list = expression { "," expression } ;
array-literal = "[" [ expression { "," expression } ] "]" ;
number-literal = [ "-" ] integer-literal | [ "-" ] float-literal ;
integer-literal = digit { digit } ;
float-literal = digit { digit } "." digit { digit } ;
id = letter { letter | digit | "_" } ;
operator = "=" | "*" | "/" | "+" | "-" ;
digit = "0" .. "9" ;
letter = "a" .. "z" | "A" .. "Z" ;
```

Figure 5. EBNF Form of DSP-DSL.

4 Evaluation Setup

Benchmarks: To evaluate the performance enhancement of DSP-dialect and the ease of programming with DSP-DSL, we conducted experiments on a set of 20 real-world DSP applications from various sources, including AudioMark™ [28] of Embedded Microprocessor Benchmark Consortium (EEMBC) suite [27], MiBench [32], and DSPStone [51]. These benchmarks cover a broad spectrum of DSP applications, from classical filter design and spectral analysis to audio compression, biomedical signal processing, and modern communication.

Evaluation Platforms: To rigorously evaluate the performance of DSP-MLIR against state-of-the-art DSP compilers, we benchmarked DSP-MLIR against Qualcomm's Hexagon-Clang compiler, specifically developed for Hexagon DSP processors (now known as NPUs). Hexagon-Clang is built upon the widely adopted LLVM open-source compiler framework, featuring an LLVM backend specifically designed for code generation on Hexagon processors.

Among available compilers for prevalent DSP architectures such as Texas Instruments (TI) and Analog Devices, even TI's latest c7000 compiler continues to rely on a proprietary compilation infrastructure maintained since the 1980s, and it does not utilize modern compiler frameworks like GCC or LLVM [16]. Consequently, Qualcomm's Hexagon compiler (hexagon-clang), leveraging modern compiler research and its open-source nature, was selected as the most suitable baseline. The Hexagon-Clang compiler is accessible as part of the Hexagon NPU SDK [5].

Application binaries were generated to match the intended execution platforms. DSP applications typically run on DSP processors for real-time scenarios or on CPUs for general-purpose use cases such as WebRTC. To demonstrate DSP-MLIR's versatility across both specialized DSP hardware and general-purpose computing platforms, the generated benchmark binaries were deployed and tested on the Qualcomm Hexagon v68 processor using Qualcomm's Hexagon Simulator (hexagon-sim version 8.8.06 [5]) and an AMD Ryzen Threadripper PRO 7955WX CPU.

CPU Compilation Flows: We compare the compilation flow of the proposed DSP-MLIR against several baselines, including DSP applications expressed in DSP-DSL and C.

- **GCC:** DSP applications expressed in C, and compiled with GCC 11.4.0 using -O3 optimizations.
- **Clang:** DSP applications expressed in C, and compiled with Clang-19 using -O3 optimizations.

- **Affine:** DSP applications expressed in our proposed DSP-DSL, compiled to the DSP-dialect, then lowered to the `affine` and other standard dialects, apply `affine` and other optimizations, and then compile to LLVM-IR, and then finally binaries are generated using Clang. (Most of the important optimizations in MLIR are in `LinAlg` and the `Affine` dialect.) This represents the state-of-the-art in the compilation flow.
- **DSP-MLIR (ours):** This is the same flow as Affine, except that after converting the representation to DSP-dialect, we perform DSP-dialect optimizations defined in Section 3.2.

Hexagon DSP Compilation Flows: While compiling and executing on hexagon device, we use the same **Clang**, **Affine** and **DSP-MLIR** compilation flows as mentioned above and execute then using hexagon-sim. To evaluate against hexagon compiler (`hexagon-clang`) we also compare with the following compilation flows:

- **Affine-hexagon:** This is the same **Affine** compilation flow except after generating llvm for hexagon v68 processor, we use *hexagon-clang* using -O3 optimzations and *hexagon-link* linker to generate binary.
- **Clang-hexagon:** DSP application expressed in C and compiled with *hexagon-clang* using -O3 optimizations.
- **DSP-MLIR-hexagon (ours):** Same flow as **Affine-hexagon** except that we enable the DSP-dialect optimizations defined in Section 3.2.

For each compilation flow, we measure the runtime, compilation time, and binary code size.

Execution: We execute the generated binaries of each application with input signal sizes ranging from 10 to 100 million elements in steps of order.

For the AMD Ryzen, each version (compilation flow, input signal size) is executed 30 times, and we measure the average execution time to mitigate system-level variations. We flushed the cache after each run to ensure consistent initial conditions for subsequent executions. Each application was limited to single-core execution on the device using the command *taskset -c 0* to set the CPU affinity, ensuring equal comparison across different compilation methods. However, the compilers do use the SIMD capabilities of the core using SSE [45] and AVX [40] instructions.

For the Hexagon v68 processor, we generate llvm instructions for hexagon v68 using its opensource LLVM backend and then use *hexagon-link* linker provided with the Hexagon SDK to generate an executable binary. Finally, to get the execution cycle count, we use *hexagon-sim* for each variation of compilation flow.

It is worth noting any performance improvement achieved by DSP-MLIR is strictly due to the optimizations discusses in Section 3.2 as the results are normalized with respect to the *Affine* compilation flow.

5 Empirical Claims

5.1 The DSP-dialect Optimizations Improve Performance of DSP Applications by an Average of 12% Across CPU and DSP Devices

Figures 6 and 7 present the runtime performance of various DSP applications compiled using different compilation flows (lower is better). The applications are arranged from left to right in order of increasing runtime, reflecting growing computational complexity. For each application, the runtime achieved by each compilation flow is normalized to that of the *Affine* flow, which serves as the baseline due to its status as the state-of-the-art in compiler optimizations.

5.1.1 AMD Ryzen Processor: In Figure 6, the first bar for each application is the runtime from the *Affine* flow and therefore, has a height of 1. The second and the third bars are the normalized runtime by the *GCC* and *Clang* compilation flows respectively. The rightmost bar for each application is the normalized runtime of the proposed **DSP-MLIR** compilation flow.

GCC and Clang: *GCC* and *Clang* demonstrated comparable execution times for most applications, except for Low-pass Filtering and Speaker Identification applications. For these applications, Clang outperformed *GCC* due to its superior utilization of SSE and AVX vector instructions, enabling simultaneous processing of multiple elements.

Clang and Affine: *Clang* and *Affine* compilation flows yield comparable performance in 9 applications. This can be explained by the fact that after optimizing applications within MLIR, the *Affine* compilation flow emits optimized LLVM IR and uses Clang-19 to generate the executable binary. *Clang*, being an earlier development and an integral part of the LLVM project, already incorporates many of the optimizations available in the `affine` dialect in MLIR.

However, for FIR Filter Design, Audio Compression and Target Detection applications *Clang* performed better than *Affine* because *Affine* uses 8-byte alignment tailored for scalar double-precision floating-point constants while *Clang* generates 16-byte alignment optimized for vectorized instructions. Moreover, *Clang* uses simplified scalar arithmetic instructions and fewer expensive math-function calls. *Affine*'s repeated packed operations (mulpd, addpd) and frequent sin/cos calls create high overhead and slower execution.

Effectiveness of DSP-MLIR: As shown in Figure 6, the binaries generated by our proposed *DSP-MLIR* compilation flow outperform the binaries generated by the *Affine* compilation flow across all applications and surpass both Clang and GCC in 19 out of 20 applications, yielding an average performance gain of 10% across the entire application suite.

The substantial performance enhancement can be primarily attributed to the ease of utilizing DSP domain-specific properties and theorems at the DSP-dialect level, resulting in more efficient code generation. The multi-level optimization framework provided by MLIR enables the exploitation

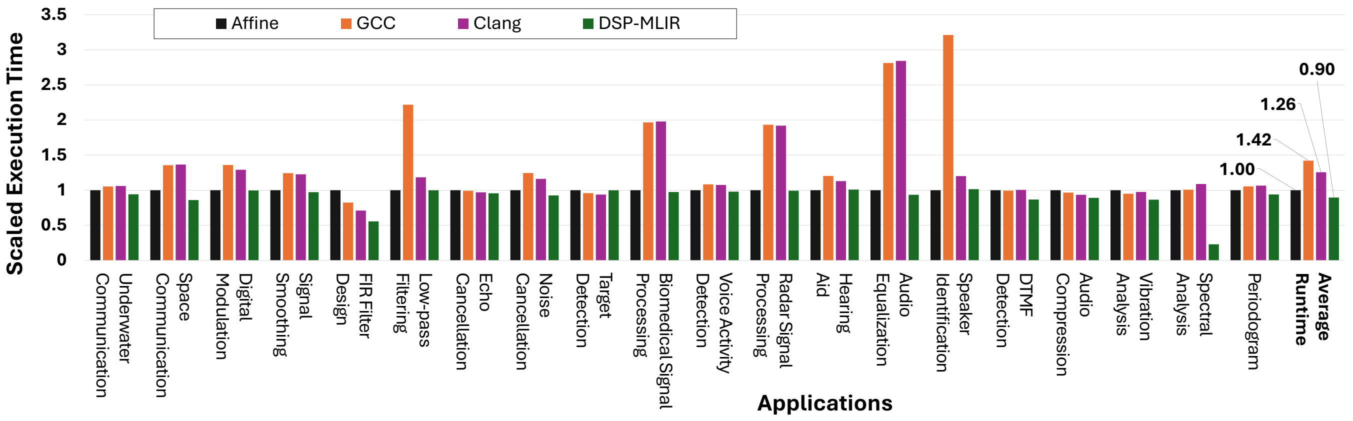


Figure 6. Runtime of the binaries generated by the various compilation flows, normalized to *Affine* and executed on AMD Ryzen Processor. *Clang* outperforms *GCC* by about 16%. *Affine* outperforms *Clang* by about 27%. *DSP-MLIR* outperforms *Affine* by about 10%.

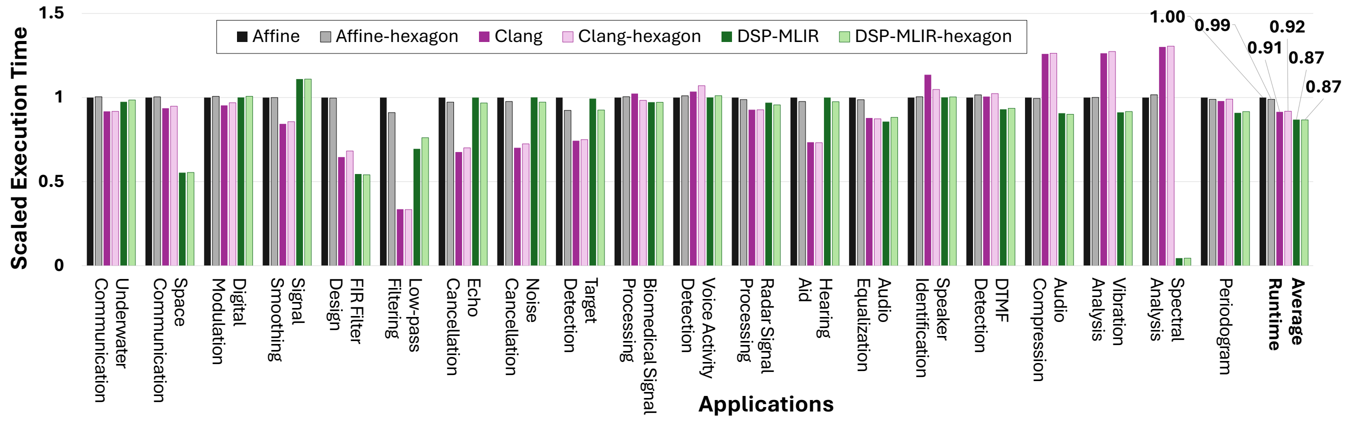


Figure 7. Runtime of the binaries generated by the various compilation flows, normalized to *Affine* and executed on Hexagon v68 Processor. *Clang* outperforms *Affine* by about 8%. However, *DSP-MLIR* outperforms *Affine* by about 13% and *Clang* by about 5%.

of DSP theorem-level optimizations and operation fusion optimizations that are challenging to implement at other lowering stages.

For the Target Detection application, DSP-MLIR performs worse than Clang. This is because the Clang-compiled binary has many more SIMD instructions but our MLIR-generated binary uses a lot of memory and branch instructions, reducing performance.

DSP-MLIR outperforms other benchmarks in various applications, e.g., Spectral Analysis, which uses Parseval's theorem (#3 in Table 2) to eliminate the FFT computation and results in almost 5x reduction in runtime over Affine. For Audio Equalization, DSP-MLIR uses optimization (#1 in Table 2) to optimize the filter design calculation for various filters and result in 2.75x reduction in runtime over GCC and Clang. For Low-pass Filtering, DSP-MLIR uses optimization #1 and #2 from Table 2 to reduce the number of computations and memory load and store operations, which results in an almost 2.25x and 1.25x reduction in runtime over GCC and Clang respectively.

5.1.2 Hexagon DSP Processor: Figure 7 represents the normalized execution cycles for respective compilation flows on the hexagon processor.[2] For each application mentioned on the x-axis, the first bar represents *Affine* compilation flow and hence has a height of 1, followed by *Affine-hexagon*, *Clang*, *Clang-hexagon*, *DSP-MLIR*, and *DSP-MLIR-hexagon* compilation flows as explained in Section 4.

[2]For the Signal Smoothing benchmark, hexagon-clang failed to compile the generated LLVM code from the DSP-MLIR framework due to version mismatches between the LLVM used by hexagon-clang and the one used to build DSP-MLIR. As a result, we report the same performance numbers for both Affine-hexagon and DSP-MLIR-hexagon as those obtained with clang, to maintain consistency and enable comparison.

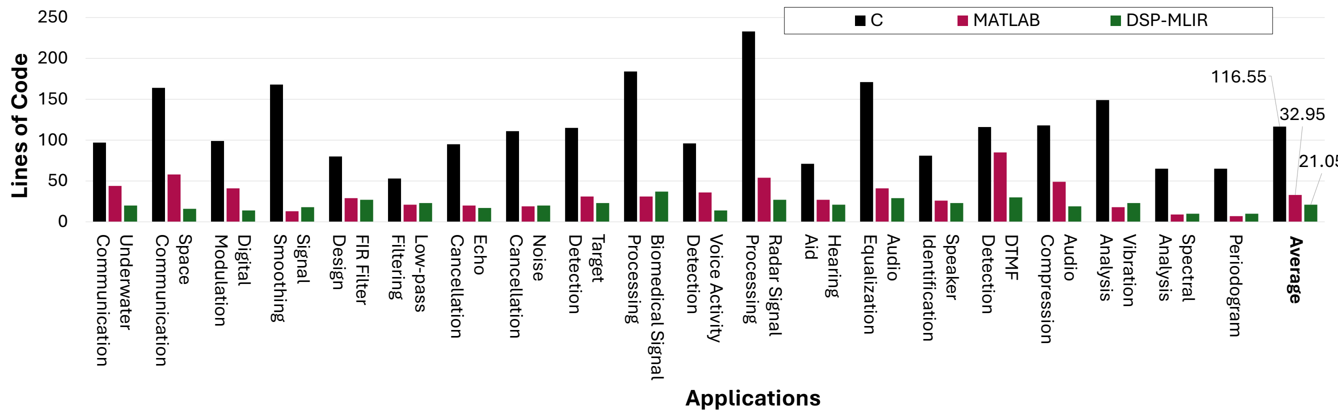


Figure 8. Lines of code in C, MATLAB®, and DSP-DSL for the various DSP applications. The DSP-DSL representation is about 5x smaller than in "C".

Affine, *Clang*, *DSP-MLIR* and their respective hexagon-clang compilation flows (e.g. Affine and Affine-hexagon / Clang and Clang-hexagon) exhibit similar performance with minor variation in performance. For Speaker Identification application, *Clang-hexagon* performs slightly better than *Clang* as it generates optimized DSP-specific instructions (dfmpyfix), efficient memory alignment (.falign), and specialized stack allocations for hexagon processors. These optimizations enhance arithmetic throughput and memory efficiency for hexagon DSP architectures.

Clang and Affine: *Clang* and *Clang-hexagon* compilation flows perform significantly better in 12 out of 20 applications than *Affine* and *Affine-hexagon* compilation flows and performs 8% better on average, as the C code compiled version uses simpler, fewer arithmetic operations and efficient loops. On the contrary, *Affine* and *Affine-hexagon* versions perform repeated heavy DSP instructions (dfmpyfix), creating pipeline stalls and higher computational overhead.

The *Clang* and *Clang-hexagon* compilation flows perform worse for Audio Compression, Vibration Analysis and Spectral Analysis applications due to similar reasons i.e. *Clang* and *Clang-hexagon* compilation flows generate complex nested loops, and excessive intermediate computations resulting in slower runtime.

Affine and DSP-MLIR: *DSP-MLIR* and *DSP-MLIR-hexagon* performs around 13% better on average as compared to *Affine* and *Affine-hexagon* compilation flow and around 5% better as compared to *Clang* and *Clang-hexagon* compilation flows. While from the results, it is worth noting that the Spectral Analysis application benefits the most from DSP optimization, excluding the application still shows 9% improvement over *Affine*, *Affine-hexagon* compilation flows and comparable results to the *Clang* and *Clang-hexagon* compilation flows. Notably, as we move right along the x-axis—corresponding to applications with longer runtimes—*DSP-MLIR* shows increasingly significant performance gains over *Clang*, highlighting its applicability for more complex workloads.

5.2 DSP-DSL Improves Programmer Productivity by 4x

To assess the productivity benefits of DSP-DSL, we measured the average number of lines of code (LOC) required to express DSP application functionality in C, MATLAB®, and our proposed DSP-DSL. Figure 8 presents the LOC comparison across these languages, with comments and empty lines excluded to ensure an accurate measurement.

As shown in Figure 8, C implementations require over 100 lines of code on average due to their low-level nature and verbosity. MATLAB® offers a more concise syntax, averaging 32.95 lines of code, which can accelerate DSP application development.

In contrast, DSP-DSL achieves the most concise expression of functionality, averaging only 21.05 lines of code. Beyond code brevity, it also delivers improved execution performance, as demonstrated in Figures 6 and 7.

5.3 DSP-DSL Doesn't Increase Compilation Time or Binary Size

There is no significant variation in the compilation times of the various compilation flows, bracketed by GCC at 0.136s and Clang at 0.152s, and DSP-MLIR at 0.139s. The size of the binaries generated by Affine is about 20% smaller than GCC or Clang. The binaries generated by DSP-MLIR are about 10% smaller than Affine. This is primarily because of the operation/loop fusion optimizations in DSP-dialect.

6 Related Work

6.1 Domain-specific Languages (DSLs) for DSP

Prior work on domain-specific languages (DSLs) for DSP applications has focused on functional programming approaches, like Feldspar [23] and FAUST [37]. Feldspar, based on Haskell, provides a dataflow-style language that describes DSP algorithms at a high level, with a backend compiler that translates them into optimized C code. The optimizations performed by Feldspar, such as variable elimination and loop unrolling, primarily occur at the C level, limiting opportunities for application level optimizations. Similarly, FAUST, a functional DSL tailored for real-time audio processing, has been widely adopted for web-based DSP applications but doesn't allow for the efficient implementation of algorithms requiring multi-rates such as the FFT, convolution and doesn't have an imperative programming model familiar to many DSP practitioners.

As compared to these works, our DSP-DSL closely resembles the popular programming language *Python*, resulting in a minimal learning curve for users.

6.2 DSP Compiler Backends

There has been work targeting backend DSP hardware and C/assembly-level software like DSP processors co-design with compiler [52] and for generating optimized code for DSP Processors using SIMD [41], parallelization of C programs [30], assembly-level optimizer [25], DSP Processors address optimization [38], enabling auto vectorized code generation [46], etc. MATLAB® [34], a popular framework for DSP applications, also has Embedded Coder® [29], which automatically generates fast C code for embedded processors but is closed-source. As opposed to these, our work targets the compiler front end and may be able to work with and benefit from these backend approaches.

6.3 Hardware-specific DSP Libraries

DSP processor vendors provide platform-optimized libraries. For example, Qualcomm provides Qualcomm Hexagon Library (QHL) and QHL Hexagon Vector eXtensions (HVX), which support mathematical computations, basic linear algebra operations, and DSP operations for Hexagon DSP processors. Texas Instrument provides DSP libraries (DSPLIBs) for their DSP processors such as TMS320C6000 [13] and microcontrollers such as MSP430 [19]. Common Microcontroller Software Interface Standard (CMSIS) DSP Software Library [18] supports common DSP functions for ARM Cortex-M and Cortex-A processors.

While existing DSP libraries such as QHL focus on hardware specific optimizations, the objective of the proposed DSP-MLIR framework is fundamentally different. DSP-MLIR introduces a hardware-agnostic domain-specific language (DSL) and a compiler infrastructure that enables high-level, inter-kernel optimizations for DSP applications. These high-level transformations are orthogonal and complementary to the low-level performance tuning performed by hardware-specific libraries. Importantly, DSP-MLIR has the potential to further enhance the performance of library-based DSP codes by enabling transformations that current library toolchains cannot express or apply.

For instance, manually applying Optimization #1 in Table 2 (Mul(IdealFilter(wc, n), Hamming(n)) to FilterHammOpt(wc, n)) to the FIR Filter Design application using QHL with HVX results in an additional performance improvement of approximately 80%. However, this optimization is not realized by *hexagon-clang* due to its inability to perform inter-kernel analysis and transformation. By lowering DSP-MLIR's high-level operations to invoke optimized CRL (Code Replacement Library) functions where appropriate, such performance gains can be automatically unlocked.

This work represents an essential step toward bridging the gap between high-level algorithmic intent and low-level hardware efficiency. Rather than replacing hardware-specific optimizations, DSP-MLIR complements them by enabling a new class of compiler transformations that are otherwise inaccessible. Future work can further extend this framework to automatically lower to existing DSP libraries like QHL, combining the benefits of domain-aware high-level compilation with mature hardware-specific backends.

7 Conclusion

This paper introduced DSP-MLIR – a framework for optimizing and simplifying the development of DSP applications. Experiments on a diverse set of 20 DSP applications demonstrate performance gains of 12% over state-of-the-art Affine compilation flows on CPU and Hexagon DSP processor. Additionally, DSP-MLIR enables programming simplification of about 5x over "C" representations.

8 Future Directions

This work lays the foundation for a unified compiler flow targeting both DSP and deep learning (DL) workloads. While current systems often deploy DSP and DL operations on separate processors, limiting cross-optimization, this work anticipates a shift toward unified compute architectures. Recent processor designs such as *MxCore* [31] and *Marsellus SoC* [24] demonstrate this trend by integrating vector, DSP, and neural processing units into a single, cohesive system. By enabling domain-specific compilation within a shared MLIR-based framework, our approach supports future hardware-software co-design efforts. In particular, it facilitates the development of hybrid processors with shared memory or tightly coupled compute units, where compiler-level fusion of DSP and DL operations can significantly reduce memory transfers, improve locality, and lower end-to-end latency. This integration can lead to more efficient deployment of real-time, compute-intensive applications across domains such as audio, radar, communications, and beyond.

Another future direction is developing hardware-specific lowering passes from DSP dialect to platform-optimized library functions. As we demonstrated in Section 6.3, the high-level optimizations in DSP-MLIR have the potential to further optimize codes that utilize such hardware-specific library functions. This future work will include lowering optimized DSP-MLIR workflows to library functions by developing robust compiler backend to maximize the benefits of the platform-optimized libraries.

9 Data-Availability Statement

The compiler code, scripts to reproduce the results are available on Zenodo [36].

Acknowledgments

This work was partially supported by funding from National Science Foundation grants CPS 1645578 and CCF 2436016, Semiconductor Research Corporation (SRC) project 3154, and National Research Foundation of Korea (NRF) funded by the Korean Government, Ministry of Science and ICT (MIST), South Korea, grant RS-2024-00341794.

References

[1] (Accessed on 03/21/2025). ADSP-21992 Datasheet and Product Info | Analog Devices. https://www.analog.com/en/products/adsp-21992.html#part-details

[2] (Accessed on 03/21/2025). affine Dialect - MLIR. https://mlir.llvm.org/docs/Dialects/Affine/

[3] (Accessed on 03/21/2025). C6000-CGT - C6000 code generation tools compiler. https://www.ti.com/tool/C6000-CGT.

[4] (Accessed on 03/21/2025). GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[5] (Accessed on 03/21/2025). Hexagon NPU SDK | Qualcomm Developer. https://www.qualcomm.com/developer/software/hexagon-npu-sdk

[6] (Accessed on 03/21/2025). linalg Dialect - MLIR. https://mlir.llvm.org/docs/Dialects/Linalg/

[7] (Accessed on 03/21/2025). MPSLab-ASU/DSP_MLIR: Digital Signal Processing Compiler in MLIR. https://github.com/MPSLab-ASU/DSP_MLIR

[8] (Accessed on 03/21/2025). Multi-Level Intermediate Representation Overview. https://mlir.llvm.org/.

[9] (Accessed on 03/21/2025). nvgpu Dialect - MLIR. https://mlir.llvm.org/docs/Dialects/NVGPU/

[10] (Accessed on 03/21/2025). nvvm Dialect - MLIR. https://mlir.llvm.org/docs/Dialects/NVVMDialect/

[11] (Accessed on 03/21/2025). onnx-mlir dialect. http://onnx.ai/onnx-mlir/Dialects/onnx.html

[12] (Accessed on 03/21/2025). Operation Canonicalization - MLIR. https://mlir.llvm.org/docs/Canonicalization/

[13] (Accessed on 03/21/2025). SPRC265 - TMS320C6000 DSP Library (DSPLIB). https://www.ti.com/tool/SPRC265.

[14] (Accessed on 03/21/2025). The Clang Compiler Frontend. https://clang.llvm.org/index.html.

[15] (Accessed on 03/21/2025). The LLVM Compiler Infrastructure. https://llvm.org/.

[16] (Accessed on 03/21/2025). TI C7000 compiler. https://www.ti.com/video/series/c7000-compiler.html

[17] (Accessed on 03/21/2025). TMS320C6452 - C64x+ fixed point DSP-up to 900MHz, 1Gbps Ethernet. https://www.ti.com/product/TMS320C6452.

[18] (Accessed on 05/05/2025). CMSIS DSP Software Library. https://arm-software.github.io/CMSIS_5/DSP/html/index.html.

[19] (Accessed on 05/05/2025). MSP-DSPLIB - Digital Signal Processing (DSP) Library for MSP430 Microcontrollers. https://www.ti.com/tool/MSP-DSPLIB.

[20] Nicolas Bohm Agostini, Serena Curzel, Ankur Limaye, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, Antonino Tumeo, and Fabrizio Ferrandi. 2022. The SODA approach: leveraging high-level synthesis for hardware/software co-design and hardware specialization. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1359–1362.

[21] Nicolas Bohm Agostini, Serena Curzel, Jeff Jun Zhang, Ankur Limaye, Cheng Tan, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Brooks, et al. 2022. Bridging python to silicon: The soda toolchain. *IEEE Micro* 42, 5 (2022), 78–88.

[22] Nicolas Bohm Agostini, Ankur Limaye, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, Antonino Tumeo, Serena Curzel, and Fabrizio Ferrandi. 2022. SODA Synthesizer: an Open-source, Multi-level, Modular, Extensible Compiler from High-level Frameworks to Silicon. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–7.

[23] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajdax. 2010. 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)*. 169–178. doi:10.1109/MEMCOD.2010.5558637

[24] Francesco Conti, Gianna Paulin, Angelo Garofalo, Davide Rossi, Alfio Di Mauro, Georg Rutishauser, Gianmarco Ottavi, Manuel Eggiman, Hayate Okuhara, and Luca Benini. 2023. Marsellus: A heterogeneous RISC-V AI-IoT end-node SoC with 2–8 b DNN acceleration and 30%-boost adaptive body biasing. *IEEE Journal of Solid-State Circuits* 59, 1 (2023), 128–142.

[25] B Dupont de Dinechin, F de Ferri, Christophe Guillon, and Artour Stoutchinin. 2000. Code generator optimizations for the ST120 DSP-MCU core. In *Proceedings of the 2000 International Conference on Compilers, architecture, and synthesis for embedded systems*. 93–102.

[26] DSP-MLIR. (Accessed on 03/21/2025),. https://bit.ly/dspmlir

[27] EEMBC. 1997. The Embedded Microprocessor Benchmark Consortium. https://www.eembc.org/

[28] EEMBC. 2022. AudioMark™. https://github.com/eembc/audiomark

[29] Akrem Elrajoubi, Simon S Ang, and Ali Abushaiba. 2017. TMS320F28335 DSP programming using MATLAB Simulink embedded coder: Techniques and advancements. In *2017 IEEE 18th Workshop on Control and Modeling for Power Electronics (COMPEL)*. IEEE, 1–7.

[30] Björn Franke and Michael FP O'Boyle. 2003. 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*. 219–224.

[31] Biji George, Om Ji Omer, Ziaul Choudhury, Anoop V, and Sreenivas Subramoney. 2022. A Unified Programmable Edge Matrix Processor for Deep Neural Networks and Matrix Algebra. *ACM Transactions on Embedded Computing Systems (TECS)* 21, 5 (2022), 1–30.

[32] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 3–14. doi:10.1109/WWC.2001.990739

[33] Thomas Holton. 2021. *Digital Signal Processing: Principles and applications*. Cambridge University Press.

[34] The MathWorks Inc. 2025. MATLAB. https://www.mathworks.com/help/matlab/

[35] Miaomiao Jiang, Yilan Zhu, Honghui You, Cheng Tan, Zhaoying Li, Jiming Xu, and Lei Ju. 2024. FHE-CGRA: Enable efficient acceleration of fully homomorphic encryption on CGRAs. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[36] Abhinav Kumar, Atharva Khedkar, Hwisoo So, Megan Kuo, Ameya Gurjar, Partha Biswas, and Aviral Shrivastava. 2025. *MPSLab-ASU/DSP_MLIR: DSP-MLIR artifact.* doi:10.5281/zenodo.15549101

[37] Stéphane Letz, Yann Orlarey, and Dominique Fober. 2018. FAUST domain specific audio DSP language compiled to WebAssembly. In *Companion Proceedings of the The Web Conference 2018*. 701–709.

[38] Sean Leventhal, Lin Yuan, Neal K Bambha, Shuvra S Bhattacharyya, and Gang Qu. 2005. DSP address optimization using evolutionary algorithms. In *Proceedings of the 2005 workshop on Software and compilers for embedded systems*. 91–98.

[39] LLVM. (Accessed on 03/21/2025),. Torch-MLIR. https://github.com/llvm/torch-mlir

[40] Chris Lomont. 2011. Introduction to intel advanced vector extensions. *Intel white paper* 23 (2011), 1–21.

[41] Markus Lorenz, Peter Marwedel, Thorsten Drager, Gerhard Fettweis, and Rainer Leupers. 2004. 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)*. IEEE, 839–842.

[42] Yixuan Luo, Cheng Tan, Nicolas Bohm Agostini, Ang Li, Antonino Tumeo, Nirav Dave, and Tong Geng. 2023. ML-CGRA: an integrated compilation framework to enable efficient machine learning acceleration on CGRAs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[43] TensorFlow MLIR. (Accessed on 03/21/2025),. https://www.tensorflow.org/mlir/tf_ops

[44] Users of MLIR. 2025. https://mlir.llvm.org/users/

[45] S Thakkur and Thomas Huff. 1999. Internet streaming SIMD extensions. *Computer* 32, 12 (1999), 26–34.

[46] Samuel Thomas and James Bornholt. 2024. 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*. 19–34.

[47] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. 2023. {HECO}: Fully Homomorphic Encryption Compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4715–4732.

[48] Wikipedia contributors. 2024. Digital signal processing — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Digital_signal_processing&oldid=1233991896.

[49] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *2022 IEEE international symposium on high-performance computer architecture (HPCA)*. IEEE, 741–755.

[50] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: a scalable high-level synthesis framework with multi-level transformations and optimizations. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1355–1358.

[51] Vojin Zivojnovic. 1994. DSPstone: A DSP-oriented benchmarking methodology. *Proc. Signal Processing Applications & Technology, Dallas, TX, 1994* (1994), 715–720.

[52] Vojin Zivojnovic, Stefan Pees, Christian Schlager, Markus Willems, Rainer Schoenen, and Heinrich Meyr. 1996. DSP processor/compiler co-design: a quantitative approach. In *Proceedings of 9th International Symposium on Systems Synthesis*. IEEE, 108–113.

Received 2025-03-21; accepted 2025-04-21