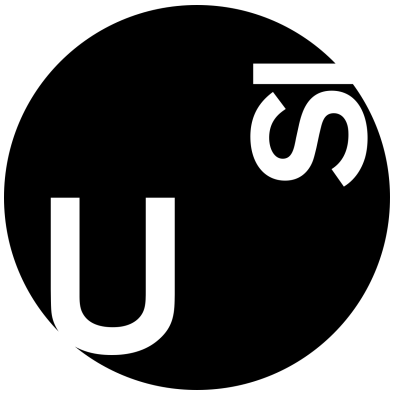


High-Performance Rotations of Multidimensional Signals using Spectral Schemes

Master Thesis

Biruk Amare

Monday 23rd September, 2024

Advisors: Prof. Dr. Rolf Krause

Co-Advisors: Prof. Dr. Diego Rossinelli, Dr. Patrick Zulian

Faculty of Informatics, USI Lugano

Abstract

Decoding Big Data offers significant potential to advance scientific inquiry. Extracting insights buried within data, however, requires exceptionally high data processing capabilities. The semiconductor industry has transitioned to progressively complex computing systems that are more and more difficult to utilize efficiently. In turn, data processing capabilities are often compromised. This thesis addresses these challenges by proposing a Low-Level Virtual Machine (LLVM)-based framework that automates kernels synthesis. By applying compiler design techniques, we explore computing patterns across various levels of abstraction aiming at efficient and portable implementation of finite impulse response (FIR) filters. These filters are fundamental to data processing, as they appear in applications ranging from convolutional layers in machine learning to numerical solutions of the Navier-Stokes equations in computational fluid dynamics. Results show performance improvements ranging from 1.45 to 11.93 times over a well-optimized C-based kernel, with measured performance reaching between 15% and 93% of the theoretical peak, holding the promise towards high-performance computing (HPC) software automation.

Contents

Chapter 1

Introduction

Decoding Big Data may formidably assist scientific inquiry. In AI, for example, advancements in Computer Vision and Large Language Models are driven by ever-increasing amounts of training data and the deepening of neural networks. The observation that qualitative improvements scale with data size, however, is not limited to machine learning and applies to Computational Science as well. The ability to extract high-quality insights from data is ultimately limited by our processing capabilities.

For decades, the continuous increase in transistor density has been a key driver of hardware advancements. This trend continues despite the "Power Wall", a barrier that marks the plateau in processor frequency and single-thread performance due to power delivery and heat dissipation limits [13]. Figure 1.1 highlights the steady increase in transistor density over time even as traditional gains from frequency scaling diminish. The semiconductor industry's response to the "Power Wall" is presented in Figure 1.2, on the left. It showcases an Apple Silicon M2 Pro processor die, which integrates billions of transistors into a single die. Rather than relying on raw speed increases, industry started replicating microprocessors (i.e., increasing core counts) and making uncore components wider, delegating the task of extracting performance to software. This has allowed Moore's law to persist, not through frequency scaling, but by increasing the computational resources available for parallel processing, offsetting the heavy burden of harnessing these enhanced processing capabilities to software.

In turn, paradigm shifts are required to transform how software will be developed to achieve superior data processing capabilities. For example, a C code snippet demonstrating single-core vectorized filtering using ARM Neon intrinsics is presented on the right of Figure 1.2. While these intrinsics are essential for managing complex tasks, they are far from sufficient; performance benchmark reveals that it only achieves about 1.75% of the theoretical peak performance on an Apple Silicon M2 Pro CPU.

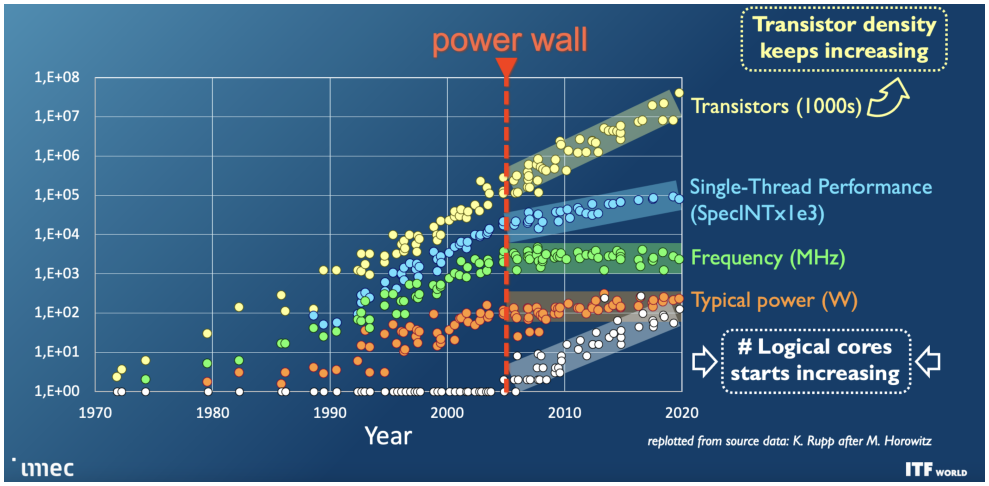
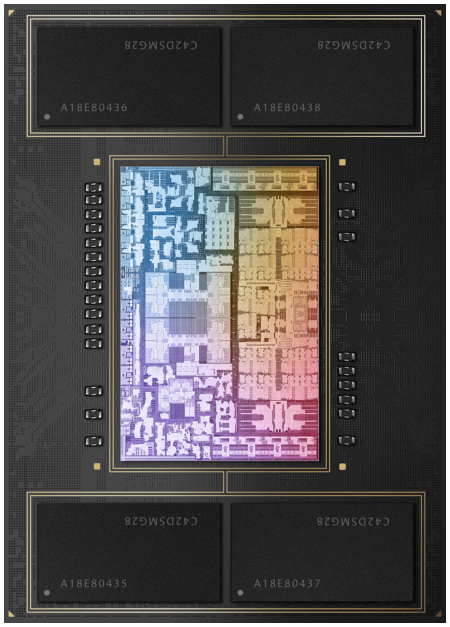


Figure 1.1: Demonstration of the increasing density of transistors and logical cores in modern processors [imec [8]]. Despite the plateau in processor frequency and typical power consumption, a phenomenon known as the "Power Wall", the expansion in available transistors and cores, as well as the moderate increase in single-thread performance, highlights the growing computational resources available. While transistor count has been steadily increasing, the number of logical cores began to rise significantly following the onset of the "Power Wall", driving advancements in parallel processing capabilities. In contrast, single-thread performance, measured using the SpecINT benchmark, is improving at a slower pace. This variation underscores the growing reliance on multi-core and parallel processing to handle complex tasks.

To improve processing quality while keeping run times acceptable, software must be able to leverage large fractions of the nominal performance offered by these systems. This involves finding and fine-tuning various algorithm implementations that execute exceptionally well, even when their performance is not easily explained from a microarchitectural standpoint. This is exemplified in computational fluid dynamics, where higher-order discretizations significantly expand computational workloads but also yield qualitative improvements in the results.

Scientific visualization, often perceived as a primitive form of data analysis, remains an essential tool for revealing insights from multidimensional data sets, as demonstrated in Figure 1.3. In the first abstraction layer of the visualization process, significant calculations are required to understand how light is transported within a data volume. The data involved often span hundreds of gigabytes or even terabytes. Advanced computational analysis takes place across the entire spectrum—from low-level signal processing to high-level computational modeling that relates experimental data against simulations, as exemplified in Figures 1.4 and 1.5. To excel in analysis quality within acceptable turnaround times, it is imperative that the associated computational workload is carried out at a high fraction of the nominal peak




```c
for (int i = 0; i < isc; ++i)
{

  float32x4_t acc =
    vdupq_n_f32(0.0f);

  for (int j = 0; j < fcc; j += 4)
  {
    float32x4_t invec =
      vld1q_f32(&in[i + j]);

    float32x4_t wvec =
      vld1q_f32(&w[j]);

    acc =
      vmlaq_f32(acc, invec, wvec);
  }

  out[i] = vaddvq_f32(acc);

}
```

Figure 1.2: An Apple Silicon M2 Pro (left) features 40 billion transistors and contains dense layout of cores and regions designed for optimized processing efficiency [Apple Inc. [2]]. On the right, a C code example demonstrates vectorized filtering, utilizing operations such as vectorized load, multiply accumulate, and addition. In this example, data is processed in chunks using the ARM Neon intrinsics to perform parallel computations on a single core efficiently. However, despite the use of advanced vectorized instructions, the performance achieved is only 1.75% of the theoretical peak. This demonstrates that while vectorization can improve computational efficiency, it is not sufficient on its own.

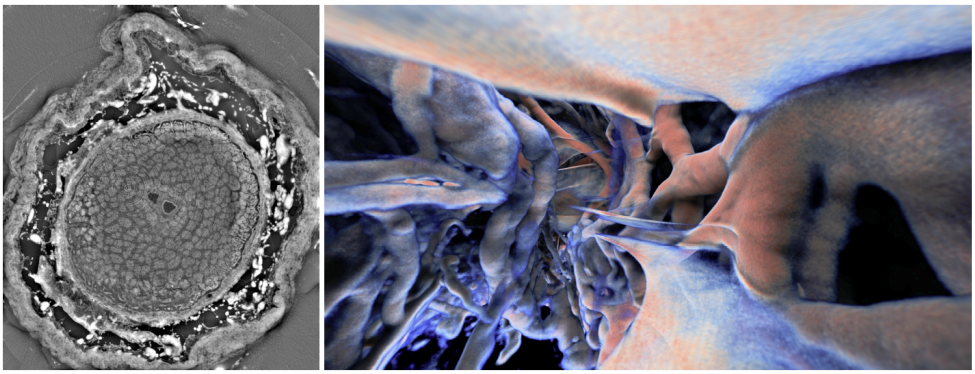


Figure 1.3: Two different types of data analysis, on the same data set. Gray-scale slice of the subarachnoid space of the optic nerve (left) and close-up view of the subarachnoid space of the optic nerve (right). The signal is the same, but the two data processing approaches lead to qualitatively different insights [Rossinelli et al. [18]].

Figure 1.4: Large-scale in-silico investigation of homeostasis in the subarachnoid space of the human optic nerve [Rossinelli et al. [18]]. Homeostasis is directly related with the interaction between the cerebrospinal fluid flow (see Figure 1.5) and the meningeal surface. Blue denotes poor levels of homeostasis, whereas red denotes sufficient material exchange between fluid and structure.

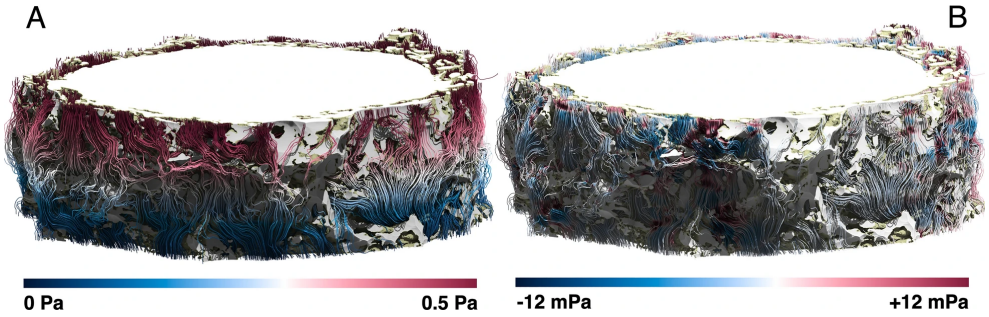


Figure 1.5: Streamlines of the CSF flow coloured linearly according to the total pressure A and dynamic pressure B [Rossinelli et al. [18]].

performance of the underlying hardware.

A focus on the computational modeling of light transport is also essential, specifically the attenuation of light. By applying a linear mapping from grayscale signal intensities to light absorption values, it is possible to maintain high accuracy while effectively managing the computational demands associated with processing large data volumes. High-performance, high-accuracy rotations of volumetric images are critical in this process, as they allow for the precise alignment of data with arbitrary directions of rays that are cast. In this work, we explore a novel approach to improve the computation of rotations within HPC environments.

1.1 Prior Works and Motivation

Our research builds upon previous work focused on improving the computation of image or signal rotations. A significant contribution in this area comes from Unser et al., who proposed and investigated different implementations in [23]. Some produced noticeable artifacts, while others produced satisfactory results. The underlying computations involved interpolations such as nearest neighbor and bilinear interpolation. The former is the quickest and simplest algorithm, but yields the poorest image quality. The latter produces a smoothing artifact, which is also apparent in cubic convolution techniques (picewise cubic model). These nonseparable methods become impractical when considering higher-order models because of the increase of size of the local neighbourhood that has to be considered and the complexity of the corresponding 2-D interpolation formulas.

The paper explored an alternative method to enable interpolation for higher-order models. It used decomposition of the rotation into a sequence of one-dimensional (1-D) transformations along the x and y axes. This *two-pass* approach simplified the computational process. However, the algorithms mentioned in the paper required signal contraction, which is very difficult to implement and is prone to errors. Even if aliasing can be prevented by an appropriate spectral shaping, there is nonetheless a loss of high spatial frequencies that becomes more pronounced for larger angles [23].

In the paper, a factorization that does not require scaling (i.e., no manipulation after rotation) is presented and is given by:

$$R(\theta) = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \tag{1.1}$$

$$= \begin{bmatrix} 1 & -tan\frac{\theta}{2} \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ sin\theta & 1 \end{bmatrix} \times \begin{bmatrix} 1 & -tan\frac{\theta}{2} \\ 0 & 1 \end{bmatrix}. \tag{1.2}$$

The first matrix shears the image in the x direction by $\Delta_x = -y \cdot tan(\theta/2)$, which then the second matrix shears in the y direction by $\Delta_y = x \cdot sin(\theta)$ and the last matrix shears it again in the x direction by Δ_x.

This *three-pass* implementation allows the computation of signal translation—shifting the entire image in space—to be decomposed into 1-D, which can be implemented using simple convolutions. The intermediate results within the factorization are shown in Figure 1.6. A remarkable characteristic of the matrices is that they involve no scaling. The transformation preserves all areas in the image, since the determinants of the matrices are all one. No frequency (high or low) components in the image are created or lost, avoiding the need for upsampling and downsampling. It is because of this reason that the paper used this approach to design algorithms that are more accurate than previous ones.

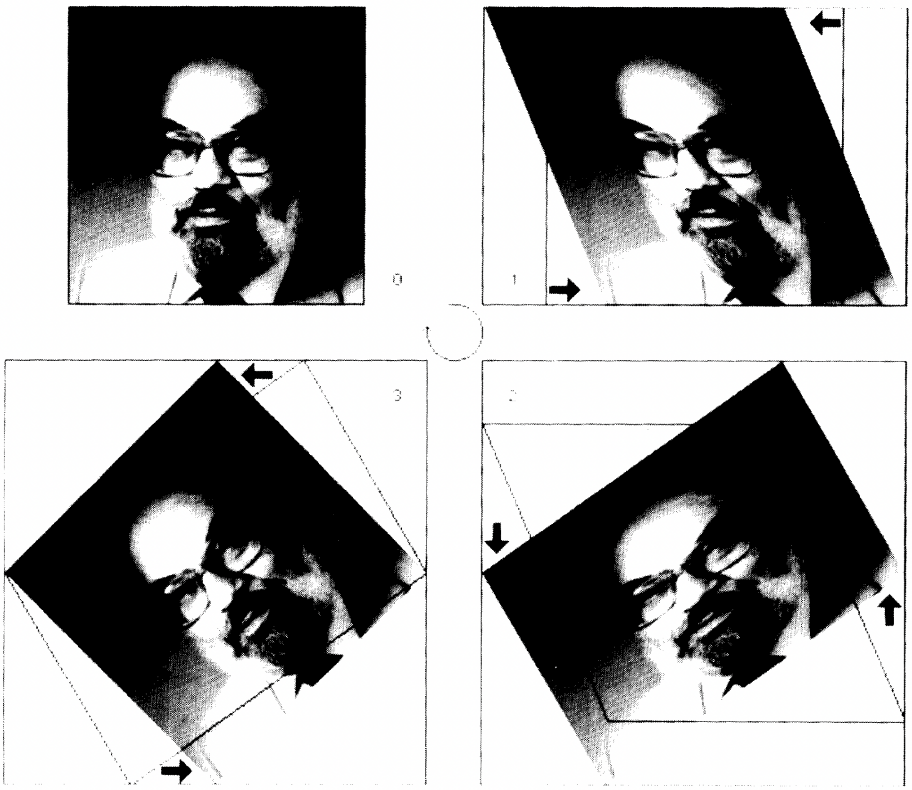


Figure 1.6: Rotation of an image is carried out by cascading translation operations [Unser et al. [23]]. Sequence follows the image indices i.e. top-left, top-right, bottom-right, bottom-left.

The paper introduced a general theoretical framework for the design and analysis of convolution-based interpolators. It discussed two approaches for the design of translation operators. One uses direct resampling of the signal, while the other minimizes the error between the ideal signal translation and its approximation within the given functional space. The paper showed that the latter can be implemented using the former, because it is an interpolation in the same approximation space. Instead of simply resampling, it computes the orthogonal projection of the translated function in the function space in L_2. For a given interpolation model, this is the optimal way to discretize the translation operator because it minimizes the L_2-error between the exact translation and its approximation in the function space.

Finite Impulse Response (FIR) filters

Filters are mathematical algorithms designed to alter the frequency content of a signal or an image. FIR filters are essential components in digital signal processing (DSP) systems, utilized for a broad range of applications,

from audio and image processing to telecommunications and biomedical engineering. Unlike Infinite Impulse Response (IIR) filters, FIR filters have a finite duration response, making them attractive for applications where linear phase and stability are critical. Linear phase ensures that all frequency components of an input signal are delayed by the same amount, preserving the original wave shape. Stability is achieved because the output of FIR filters depends only on a finite number of input samples, preventing oscillations or unbounded outputs regardless of the input.

The primary function of an FIR filter is to alter the frequency content of a signal by using convolution or correlation. This process effectively attenuates or enhances certain frequency components of the input signal according to the filter's design parameters. We will see more about the properties of FIR filters in Chapter 3.

Resampling Translation with FIR Filters

In [23], computational workload of translating a signal or image is separated into computing interpolating coefficients and a reconstruction kernel. Translation of each row and each column of an image is carried out by evaluating an interpolation scheme:

$$f(x-\Delta) \approx \sum_{l} c(l)\varphi(x-\Delta-l) = b_\Delta \circledast c(x), \tag{1.3}$$

where Δ is the translation or shift parameter, f is the input signal, φ is the reconstruction kernel, and c are the interpolating coefficients. Here, the digital filtering kernel b_Δ is the resampled version of $\varphi(x-\Delta)$ and $\circledast$ is the convolution operator. Translated signals are resampled by evaluating the interpolation scheme at sample points.

In the paper, B-splines of orders 1, 3, 7 were considered as reconstruction kernels. Except for the linear case, B-splines are non-interpolating, and thus one has to solve a system of interpolating equations. The system size is the size of the input signal, and it can be in the order of ten thousand samples. Inverting a linear system of equations $Ax = b$ of such sizes may incur non-negligible computational costs.

When B-splines are used as reconstruction kernels, the system can be inverted very efficiently by cascading a few IIR filters [22]. Despite such algorithmic advantages, the linear system remains global. This means that while the resampled translation of the signal at any given point will only consider a limited set of adjacent points, the system of equations used to perform this operation is proportional to the size of the input signal. The kernels have compact support, but the system of equations is global.

This costly method of solving a global system motivated the research on alternative computational schemes that are local. In [16], resampling trans-

lated 1-D signals by carrying out direct convolution with filters of compact support is proposed and implemented. This avoided the need for solving global system of equations. The type of filters sought were *FIR*.

Direct *convolution* is generally considered inefficient, but if the filter size is reasonably small, it can be carried out at outstanding performance. In [16], it is explained that it exposes *instruction-level parallelism* and its regular access pattern exposes *data-level parallelism* in the form of *single-instruction multiple-data* operations. These topics will be covered in detail in Chapter 4.

The objective in [16] was then to design an FIR filter of relatively compact support (e.g., 20 - 40 coefficients) that is used to shift a signal by a real (non-integer) value using direct convolution.

Translation Filter

Translation filters, also known as shift-invariant filters, are a class of linear filters that exhibit shift-invariance properties. In [16], the design is presented for a digital signal of infinite length. An arbitrary shift δ applied on the signal in the frequency domain can be exactly represented with the transfer function:

$$H(\omega) = e^{-i\delta\omega}. \tag{1.4}$$

In the discrete case, the transfer function of the filter becomes:

$$H[\omega_m] = e^{-i\delta\omega_m}, \tag{1.5}$$

where m is the wave number and $\omega_m = \frac{2\pi}{N}m$ and N is the number of samples of the data on which the transfer function is applied.

This works well for integer sized shifts, but is cause of error when implementing shifts in fractions. In order to maintain the Hermitian symmetry of the frequency spectrum of real-valued signals, a solution is proposed that takes into consideration the negative frequency indices. If the shift does not have a fractional component, the convolution with the filter is skipped entirely. Shifting the signal with an integer value is a trivial task.

Smoothing Filter

When dealing with finite signals of N samples, the frequency spectrum has to be sampled into N frequency bins. In such settings, any shift that is non-integer would lead to spurious oscillation due to the Gibbs phenomenon. Figure 1.7 shows an example of Gibbs phenomenon, which is a property of the Fourier series around jumps.

In [16], it is shown that the result from the translation filter had the strongest spurious oscillations for a shift of $\delta = 0.5$. The goal of the smoothing filter

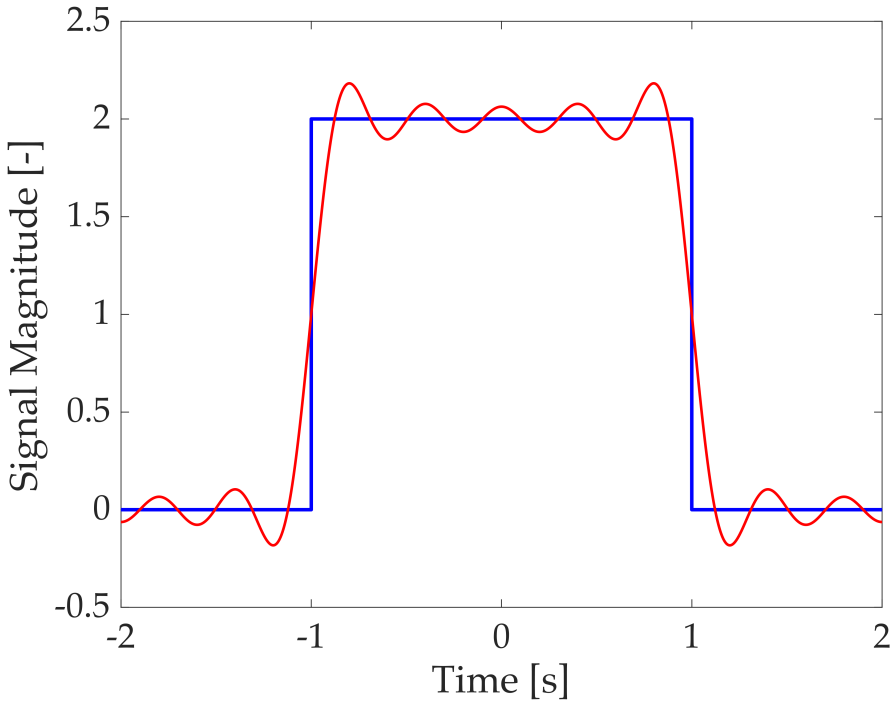


Figure 1.7: Gibbs phenomenon illustrated for a rectangular signal. The signal has a rising edge and a falling edge that cause the oscillations. The higher the number of Fourier series coefficients, the better the approximation of the series to the signal will be. Here, the series is shown in red to the signal in blue.

is to dampen these oscillations. An approach that utilized the Lanczos smoothing kernel is implemented. The smoothing filter is represented in the time domain as follows:

$$g(t) = \frac{1}{2} + \frac{1}{2}cos(\frac{\pi}{M}t), \tag{1.6}$$

where M is chosen between 1.5 and 3.

Similar to the Lanczos smoothing, the approach performed a Continuous Fourier Transform (CFT) of the filter in (1.6) and sampled it to obtain the Discrete Fourier Transform (DFT) representation. The CFT is given by:

$$F(\omega) = \int_{-M}^{M} \Big(\frac{1}{2} + \frac{1}{2}cos(\frac{\pi}{M}t)\Big)e^{-i\omega t}dt \tag{1.7}$$

$$= \frac{2M^2\omega sin(M\omega)}{M^2\omega^2 - \pi^2}. \tag{1.8}$$

Using (1.8), different filters for different M sizes could be built. Best size was found to be for $M = 3/2$.

The result from the CFT is then discretized by sampling and multiplied with the transfer function of the translation filter to get the final Fourier domain representation of the combined filter, which does translation *and* smoothing. Subsequently, the Inverse Discrete Fourier Transform (IDFT) of the combined filter is taken to obtain the coefficients of the target FIR filter in the time (signal) domain.

The report has elaborated the design and application of relatively small filters that translate large volumetric datasets to accurately rotate them in the Euclidean space. The approach implemented 3-D rotations by shifting the "1-D lines" of the dataset by fractional amounts. It is found to be executing very well (i.e., at high fractions of the nominal peak performance) on contemporary CPU (Central Processing Unit) micro-architectures.

This design outperformed the approach proposed by Unser et al., leading to investigate further improvements in the rotation of volumetric signals. The most obvious area of exploration is found to be the FIR implementation itself, which requires synthesis of filtering kernels.

Macro Processing and Hand-written Kernels

In this context, a kernel refers to a segment of code used for performing operations such as filtering, convolution, or smoothing on signal or image data. Powerful *macro processing* languages play an important role when it comes to developing kernels in an automated manner. These languages, such as M4, are used for text manipulation and code generation. Their flexibility allows the creation of highly customizable and efficient code templates that can adapt to different specifications. M4 excels in situations where the compiler might otherwise use suboptimal code.

Handwritten kernels are manually optimized code segments designed to fully exploit the underlying hardware. The utilization of handwritten kernels presents a significant challenge due to their inherent specificity to particular hardware architectures. Their tailored nature restricts their versatility and portability across different platforms. Moreover, crafting these kernels demands a meticulous approach and a deep understanding of hardware intricacies, making the process laborious and error-prone. Achieving optimal performance often entails intricate optimization techniques and fine-tuning, further complicating the development process. This requires a significant amount of care, consideration and expertise in their utilization.

Differently sized filtering convolutions require different filter coefficients and operations of corresponding sizes. Implementing these using macroprocessing techniques, such as M4, represents the state-of-the-art approach. M4 is particularly useful because it allows for the generation of code templates that can automatically adjust to various filter sizes. This eliminates the significant

burden and impracticality of manually writing and optimizing each specific instance. However, M4 itself is not without its challenges. It can be complex to use effectively, and debugging can be quite difficult.

Low-Level Virtual Machine (LLVM) Compiler Infrastructure

In this section, we briefly describe the LLVM project, which is an infrastructure that provides a collection of modular and reusable compiler and toolchain technologies [9]. At its core, LLVM is designed to optimize the compilation process, offering a suite of tools and libraries for building compilers, static analyzers, debuggers, and other software development tools. Unlike traditional compiler architectures, LLVM employs a unique intermediate representation (IR) known as *LLVM IR*, which serves as a platform-independent assembly language.

LLVM provides a set of well-defined interfaces and Application Programming Interfaces (APIs) that allow creation of custom passes, optimizations, and analyses into the compilation pipeline. Furthermore, LLVM's compilation support enables dynamic optimizations. This makes it suitable for applications requiring runtime code generation and optimization.

The parent project, LLVM Project, also contains other sub-projects. One of the prominent subprojects is Clang, a C, C++, and Objective-C frontend for LLVM. Clang is designed to offer fast compilation, expressive diagnostics, and a modular architecture, which facilitates easy integration and extension.

Multi-Level Intermediate Representation (MLIR) is a relatively recent addition to LLVM's suite of subprojects. Although it is not used in its entirety for this work, it is an important mention. MLIR aims to provide a common infrastructure for building high-level compilers and domain-specific languages (DSLs). It is a flexible and extensible compiler infrastructure, designed to represent and transform programs at multiple levels of abstraction. Unlike traditional compiler frameworks, MLIR adopts a layered approach, where each layer of the compiler stack is represented using a specialized *intermediate representation* tailored to its specific domain and optimization requirements.

While MLIR provides valuable strengths and advantages for custom Domain-Specific Languages (DSLs), it must ultimately be reduced to a low-level IR, such as LLVM IR, via a dialect during compilation. Since optimization with LLVM IR provides the last abstraction layer before machine code and our project does not involve a DSL, MLIR is not suitable for our needs. Therefore, for this thesis, *we focus on leveraging LLVM IR and API to generate well-optimized machine code for FIR filtering*.

1.2 This Thesis

This thesis presents the development of high-performance filtering kernels within an LLVM-based framework, instead of relying on macro-processing techniques, which become exceedingly cumbersome for higher-order filters and for performance analysis and improvement. We adopt a bottom-up approach to developing filtering algorithms that demonstrate high hardware utilization.

This thesis is organized as follows:

Chapter 2. We focus on the LLVM infrastructure. We present what it offers for the development and optimization of kernels. We delve into the IR, examining its foundational concepts, instruction set, and optimization techniques. Finally, we delve into the LLVM application program interface which enables interaction with LLVM programmatically. This grants the creation of the LLVM IR, custom tools, analysis passes, and optmizations tailored to specific needs. Through this comprehensive exploration, we aim to provide insights into the capabilities and applications of LLVM and its associated technologies in the realm of compiler construction and kernel synthesis.

Chapter 3. We focus on a thorough exploration of FIR filters, convolution, cross-correlation, and related concepts essential in digital filtering. We delve into the principles of convolution and cross-correlation, highlighting their significance in filtering. We distinguish between linear and circular convolution/cross-correlation, elaborating their discrepancies and differences. We explore a specific type of FIR filter: the low-pass filter. We examine their design principles and applications. Through this discussion, we aim to provide a nuanced understanding of FIR filters and related concepts that are necessary for the generation of efficient filtering schemes.

Chapter 4. We focus on an in-depth exploration of HPC and its fundamental principles, techniques and challenges. We delve into concepts such as Instruction-Level Parallelism and Data-Level Parallelism, which are pivotal in harnessing the computational power of modern processors and tackling the challenge of performance optimization. Furthermore, we investigate memory architecture considerations, including cache hierarchies, memory access patterns, and optimization strategies such as loop blocking. We also delve into parallel computing paradigms such as Message Passing Interface (MPI), which facilitate distributed computing across multiple cores. Finally, we present the Roofline model as a tool for understanding and optimizing performance in HPC systems.

Chapter 5. Here, we describe the implementation and setup of our project, which integrates the topics discussed beforehand. We first show implementations for simple kernels and build our way up to the heart of the thesis, which is implementation of LLVM supported high-performance low-level FIR filtering that is well optimized for all available microarchitectures. After describing implementations of the small kernels, we present our structure for generating LLVM IR code that is computationally efficient and can be used for any desired filter size.

Chapter 6. Here, we evaluate the accuracy, hardware utilization and performance of the proposed implementation. We present the result of using our implementation for a low-pass filtering scheme on a sinusoidal wave that contains additional noise. We report the performance of the baseline that is the initial point for comparison, the handwritten code and the synthesized implementations. We use the roofline to make performance comparisons and understand where our computation lies compared to the capability of the computing hardware. This is done for different filter sizes.

Chapter 7. High quality visualization relies on approximating how light interacts with volumetric structures, and such calculations can be simplified by leveraging 3-D rotations. In this chapter, we address the challenges up ahead to further optimize the performance of the FIR filtering involved in this process. Here, we discuss what is in the outlook for more improvement.

Chapter 2

Low-Level Virtual Machine (LLVM) Compiler Infrastructure

This chapter discusses the different features of LLVM. We will delve into its intricate components, starting with an overview of LLVM's IR, its syntax, and semantics. We will then explore the LLVM instruction set. Furthermore, we will examine the various optimizations that LLVM performs to enhance the performance of generated code. Finally, we will cover the LLVM API, showcasing how it can be leveraged to programmatically generate the IR.

The prevalent architecture for traditional static compilers is characterized by a three-phase design, comprising the front end, the middle end (with optimizer), and the back end. In this model, the front end parses the source code, checks for error, and constructs a language-specific Abstract Syntax Tree (AST) to represent the input code [9]. Optionally, the AST undergoes conversion into a representation for subsequent optimization processes. Following this, the optimizer executes a series of passes on the representation. Finally, the back end tailors the output to suit the target architecture.

The LLVM middle end serves as a foundational element for representing program code at a low level. It provides platform-independent representation that facilitates the implementation of analyses, optimizations and transformations. When a compiler adopts a *universal* code representation in its middle component, it allows for the development of front ends tailored to a diverse set of programming languages, each capable of compiling to the common intermediate representation. Likewise, back ends can be crafted to generate machine-specific instructions for a wide array of target architectures, as depicted in Figure 2.1. The optimizer in LLVM has a pivotal role in the compilation process. It makes transformations aimed at enhancing the runtime performance of the code. These transformations typically include tasks such as eliminating redundant computations and streamlining code execution. Subsequently, the back end then maps the code onto the target instruction

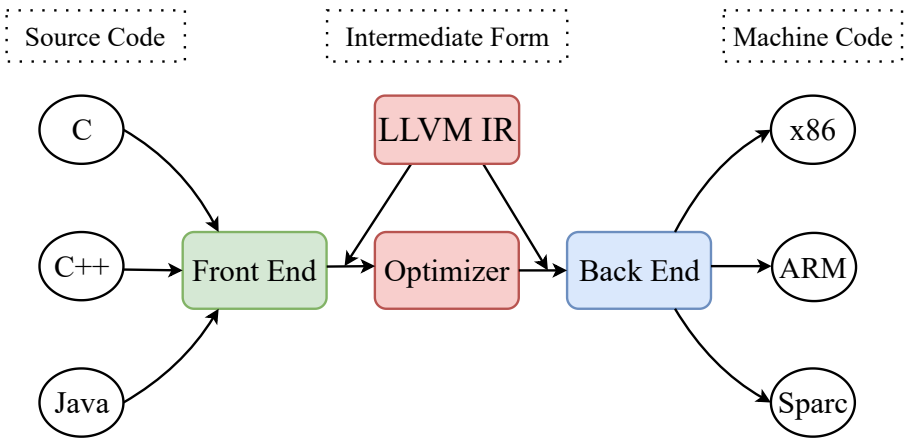


Figure 2.1: Major components of a three-phase compiler.

set. It performs instruction selection, register allocation, and instruction scheduling, all of which are crucial for generating efficient and effective machine code that fully leverages the capabilities of the target architecture.

2.1 Intermediate Representation

At the heart of LLVM's design lies its IR, which is used to represent code in a format optimized for hosting a series of analyses and transformations. The LLVM IR is itself defined as a first class language with well-defined semantics. Listing 2.1 shows a simple example that implements doubling of a 32-bit integer in LLVM IR. Its corresponding C based implementation is given in Listing 2.2. The LLVM IR exhibits characteristics akin to a low-level

```
1 define i32 @doublenumber(i32 %num) {
2   %result = alloca i32, align 4
3   store i32 %num, i32* %result, align 4
4   %0 = load i32, i32* %result, align 4
5   %1 = mul nsw i32 %0, 2
6   ret i32 %1
7 }
```

Listing 2.1: Doubling function written in LLVM IR.

```
1 int doublenumber(int num)
2     return num * 2;
```

Listing 2.2: Doubling function written in C.

Reduced Instruction Set Computer (RISC)-like virtual instruction set, as exemplified in Listing 2.1. Much like real RISC instruction sets, LLVM IR operates on linear sequences of instructions. It supports operations such as addition, subtraction, comparison, and branching. It incorporates support for labels and the properties of RISC architectures. However, LLVM IR diverges from conventional RISC instruction sets in significant ways. Notably, it has a strong typing system, wherein data types like `i32` denote 32-bit integers, and `i32**` signifies a pointer to a pointer to a 32-bit integer. This emphasis on strong typing enhances code clarity and facilitates robustness in program semantics. Additionally, LLVM abstracts away certain machine-specific details, such as the calling convention, through dedicated instructions like `call` and `ret` along with explicit handling of function arguments. A fundamental departure from traditional machine code lies in LLVM IR's approach to register usage. While conventional instruction sets rely on a fixed set of named registers, LLVM IR employs an arbitrary set of virtual registers denoted by the % character.

2.2 Virtual Instruction Set

The LLVM instruction set is designed to reflect the operations of typical processors while deliberately avoiding hardware-specific constraints such as physical registers, pipeline configurations, complex calling conventions, or system-level traps. In the LLVM framework, there are unlimited typed virtual registers that can store various primitive data types, including integers, floating-point numbers, and memory addresses. These virtual registers follow the Static Single Assignment (SSA) form, a standard format used in compiler representations.

In LLVM programs, the loading and storing of values between virtual registers and memory occur exclusively through designated `load` and `store` operations utilizing typed pointers.[1] Object allocation is implemented using `alloca` and `malloc` instructions, which are responsible for stack and heap allocations, respectively. Listing 2.3 shows an example. Stack objects are accessed through pointer values returned by the `alloca` and `malloc` instructions. They reside within the stack frame of the current function and are automatically deallocated upon function exit, while heap objects necessitate explicit freeing via a designated `free` instruction.

```
1 %stack = alloca i32
```

Listing 2.3: Allocation of a 32-bit integer variable on the stack.

[1] The nature of the types depend on the LLVM version. They can be opaque or explicit types

LLVM abstains from defining runtime and operating system functionalities like Input/Output (I/O) operations or memory management. Instead, these essential functionalities are provided by separate runtime libraries and APIs, which programs link against as needed. Despite this delineation, LLVM's virtual instruction set retains a prominent position as a primary language, supported by textual, binary, and in-memory representations [9].

Three-Address Code

Three-address code has long been the preferred representation for RISC architectures and language-independent compiler optimizations. Its similarity to machine code, with a concise set of simple operations, has made it a popular choice. Additionally, it can be easily compressed, enabling the creation of densely packed LLVM files.

Most LLVM operations follow the three-address format, typically involving one or two operands and producing a single result stored in a separate virtual register [1]. LLVM provides a wide range of arithmetic and logical instructions, such as `add`, `sub`, `mul`, `div`, `rem`, `not`, `and`, `or`, `xor`, `shl`, `shr`, among others. Some instructions, however, may take zero, three, or a variable number of operands. Examples include `call` instructions and the `phi` instruction, which is used to represent code in SSA form. These topics are explored in greater detail in the next sub-section.

A notable feature of LLVM is its polymorphic nature, allowing instructions to operate on various types of operands. This design reduces the number of distinct opcodes by eliminating the need for separate opcodes for operations on signed and unsigned integers, single- and double-precision floating-point values, or arithmetic and logical shifts [9]. Instead, the operand types automatically determine the semantics of the operation and the type of the result. For example, Listing 2.4 demonstrates how the operand types dictate the operation type and its resulting value. In the listing, the type information

```
1     %X = div float 4.0, 9.0         ; Float division
2     %Y = div double 12.0, 4.0       ; Double division
3     %cond = fcmp olt float %X, 8.0 ; Produces a boolean value
4     br bool %cond, label %True, label %False
5 True:
6     ...
```

Listing 2.4: LLVM code snippet of a single- and double-precision floating-point divisions that illustrates typed operations.

specifies whether to perform a `float` or `double` division and determines if the comparison operation should be for `float` or `double` values.

Static Single Assignment Form

LLVM uses the SSA form as its primary code representation. A program is in SSA form when each variable is defined exactly once, and each usage of a variable is *dominated* by its definition [9]. SSA form simplifies many dataflow optimizations and analyses because each variable has only one definition, making it easy to locate that definition.

An instruction that computes a value implicitly creates a new virtual register to hold that value. The value can be given an explicit name (e.g., `%s = add ...`) or be automatically assigned a sequentially numbered signature by the LLVM system, allowing direct reference to the operation that computes the value.

Transforming straight-line code into SSA form is straightforward. However, in the presence of control flow, simple variable renaming is insufficient for the IR to be in valid SSA form. Nodes with multiple predecessors may bring different versions of a variable. To manage control flow merges, SSA form uses the Φ function. This function selects an incoming value based on the originating block. LLVM provides a `phi` instruction with the syntax given in Listing 2.5. All `phi` instructions in a block must appear at the beginning

```
1 <result> = phi <type> [ <val0>, <label0> ],...,[ <valN>, <↩
    labelN> ]
```

Listing 2.5: LLVM's `phi` instruction. The `result` is assigned the value `val0` if control reaches this instruction from the block labeled `label0`, and `val1` if it comes from block `label1`, and so on [14].

of the block. Figure 2.2 illustrates the control flow, demonstrating the `phi` node as a "merger" of expressions. Listing 2.7 illustrates a `phi` instruction in

```
1 float sum(const float M, unsigned N)
2 {
3     float result = 0.0;
4     for (int i = 0; i < N; ++i)
5         result = result + M;
6     return result;
7 }
```

Listing 2.6: C-based function that sums a floating-point number `M` iteratively `N` times, demonstrating basic loop and arithmetic operations.

action. It is taken from the LLVM IR implementation of the C code in Listing 2.6. There, depending on the origin of the operands, `%6` is assigned the value `0` or the value in `%10`. If label `5` is reached from label `3`, `%7` is assigned the

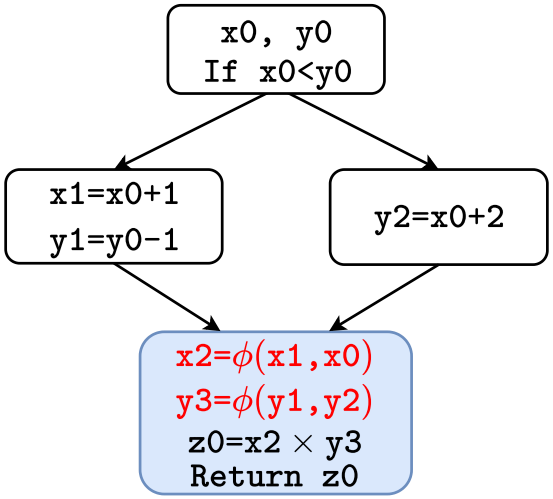


Figure 2.2: A Φ node is a "phony" use of a variable. It chooses to set x2 to either x0 or x1 based on which control flow edge was used to get to its location.

```
5:
  %6 = phi i32 [ 0, %3 ], [ %10, %5 ]
  %7 = phi float [ 0.000000e+00, %3 ], [ %9, %5 ]
  %8 = fadd float %7, %0
  %9 = fadd float %8, %0
  %10 = add nuw i32 %6, 2
  %11 = icmp eq i32 %10, %4
  br i1 %11, label %12, label %5
```

Listing 2.7: LLVM loop body snippet illustrating SSA and Φ-nodes.

value `0.000000e+00`. Otherwise, it gets the value contained in the register %9. The latter case implies a loop from label 5 back to itself.

High-Level Type Information

As described earlier, LLVM adheres to a strictly typed representation, where every SSA value and memory location is associated with a specific type, and all operations follow strict type rules. The LLVM type system includes primitive types that are independent of the source language, such as void, boolean, signed and unsigned integers ranging from 8 to 64 bits, and both single- and double-precision floating-point numbers [9]. Additionally, it supports composite types like pointers, arrays, structures, and functions. These types offer language-agnostic data representations, enabling a mapping

from higher-level language constructs.

In LLVM, all instructions are strictly typed, which imposes restrictions on their operands to facilitate transformations and maintain type correctness. For instance, the `add` instruction mandates that both operands be of the same arithmetic type, and it outputs a result of that same type. Likewise, the `load` instruction requires a pointer operand from which it will load, while the `store` instruction needs both a value and a pointer operand to store into, with the types of the pointer and value needing to correspond.

The `getelementptr` instruction is used to calculate the address of a specific element within a composite data structure. It allows LLVM to access elements within arrays, fields within structures, or components within nested data structures. When provided with a pointer to an array and an index, this instruction returns a pointer to the element at the specified element. It supports both single-level and multi-level indexing, which enables the specification of multiple indices within a single instruction.

LLVM Intrinsics

LLVM intrinsics are special functions recognized by the LLVM compiler that represent low-level operations or processor-specific instructions. Unlike regular functions, which are typically written in high-level programming languages and compiled to machine code, intrinsics are implemented directly within LLVM and closely map to specific hardware instructions or processor features. The primary purpose of LLVM intrinsics is to provide a means of expressing low-level operations in a platform-independent manner. For example, the `llvm.fmuladd.*` intrinsic functions represent multiply-add expressions that can be fused if the code generator identifies that the target instruction set supports a fused operation and that this fused operation is more efficient than performing the equivalent separate `fmul` and `fadd` instructions [9].

Conversion operators

Conversion operators in LLVM facilitate the transformation of data between different integer and floating-point types while preserving the semantics of the original values. These operators operate at the bit level, manipulating the binary representation of values to achieve the desired conversion. The zero extension operator `zext` extends an integer value by adding zero bits to its most significant end. This operator is typically used when converting smaller integer types to larger ones, such as extending an 8-bit integer to a 32-bit integer without changing the value's sign. In contrast to zero extension, the sign extension operator `sext` extends an integer value by replicating its sign bit to fill the additional bits. This operator is commonly used when

converting signed integers to larger types, preserving the sign of the original value. The truncation operator `trunc` discards the least significant bits of an integer value, effectively reducing its width. This operator is often used to convert larger integer types to smaller ones, such as truncating a 32-bit integer to an 8-bit integer.

Vectors

In LLVM, vectors represent a collection of elements of the same data type that can be processed in parallel using vector instructions. This is similar to the concept of vector processing, which we will see in Chapter 4. These vectors are typically used to exploit parallelism in operations such as arithmetic, logical, and memory operations, leading to significant performance improvements on single instruction multiple data enabled hardware. An important

```
1 define ptr @simd_add(ptr %A.in, ptr %B.in, ptr %C.out) {
2     %A = load <5 x i32>, ptr %A.in
3     %B = load <5 x i32>, ptr %B.in
4     %sum = add <5 x i32> %A, %B
5     store <5 x i32> %sum, ptr %C.out
6     ret ptr %C.out
7 }
```

Listing 2.8: Vectors of size 5 can be used for simultaneous element-wise addition of two 5-sized arrays. Implementation utilizes opaque pointers.

point to note in Listing 2.8 is that LLVM allows any positive integer, such as 5, as the element count for creating the virtual vector register. It is later converted during the compilation process to fit the vector width available in the target architecture. Similar to scalar (non-vectorized) operations, vector operations include arithmetic, logical and memory operations, with the key difference that the operand types are vectorized. Additionally, LLVM provides various intrinsics that are explicitly for vector operations.

VPlan Vectorization Plan (VPlan) is designed to facilitate high-level vectorization analysis and transformation. Its primary purpose is to enable scalable and efficient vectorization of loops and code regions. By representing vectorization decisions in a structured and composable form, it aids in optimization and transformation passes. The vectorization workflow, as described in [15], is as follows:

1. Legal Step: Check if a loop can be legally vectorized; encode constraints and artifacts if so.
2. Plan Step: Build initial VPlans based on the constraints and decisions from the Legal Step, and compute their cost. Apply optimizations to

the VPlans, possibly creating additional VPlans. Prune sub-optimal VPlans with relatively high costs.

3. Execute Step: Materialize the best VPlan. Note that this is the only step that modifies the IR.

2.3 Optimizations with LLVM

At the core of LLVM's optimization infrastructure lies a suite of optimization passes that analyze and transform code at different levels of abstraction. These passes encompass a wide range of techniques, including simple constant propagation, simple dead code elimination, loop invariant code motion, global common subexpression elimination, and many others. By applying these optimizations iteratively and strategically, LLVM can significantly enhance the performance and efficiency of written/compiled code.

One of the fundamental optimization techniques employed by LLVM is constant propagation and folding. This technique involves replacing symbolic constants with their computed values and eliminating redundant expressions. By propagating constants through the program's control flow graph and folding them into arithmetic and logical operations, LLVM can reduce the number of instructions executed at runtime, leading to improved performance and reduced code size. Another crucial optimization technique in LLVM is dead code elimination (DCE). This technique involves identifying and removing unreachable or redundant code segments that do not contribute to the program's output or behavior. LLVM can identify dead code regions and safely eliminate them, thereby reducing code bloat and improving runtime efficiency.

Listings 2.9 and 2.10 show an example of unoptimized code and the same code after applying constant propagation and DCE, respectively. The latter demonstrates how significantly the code is simplified as a result of these two optimization passes.

Loop optimization is another area where LLVM employs a suite of techniques. These techniques include loop unrolling, loop fusion, and loop vectorization, among others. By analyzing loop structures and data dependencies, LLVM can transform and parallelize loops to exploit the underlying hardware architecture's parallel execution capabilities, leading to significant performance gains for compute-intensive applications. Other traditional SSA based optimizations include Aggressive Dead Code Elimination (ADCE), Global Common Subexpression Elimination (CSE), Induction Variable Simplification, Loop Invariant Code Motion (LICM), Expression Reassociation, Simple Constant Propagation (SCP), Sparse Conditional Constant Propagation, Value Numbering (GVN), and others. Control flow graph based optimizations and

```
define i32 @doublenumber(i32 %num) {
entry:
    %num.addr = alloca i32, align 4
    %result = alloca i32, align 4
    store i32 %num, i32* %num.addr, align 4
    store i32 0, i32* %result, align 4
    %0 = load i32, i32* %num.addr, align 4
    store i32 %0, i32* %result, align 4
    %1 = load i32, i32* %result, align 4
    %mul = mul nsw i32 %1, 2
    store i32 %mul, i32* %result, align 4
    %2 = load i32, i32* %result, align 4
    ret i32 %2
}
```

Listing 2.9: Unoptimized doubling function written in llvm.

```
define i32 @doublenumber(i32 %num)  {
entry:
  %mul = shl nsw i32 %num, 1
  ret i32 %mul
}
```

Listing 2.10: Optimized version of the doubling function shown in Listing 2.9.

analyses include Critical Edge Elimination, various forms of Dominator Information, Interval Construction, Natural Loop Construction, Loop Pre-header Insertion, CFG Simplification, and others.

LLVM's optimization infrastructure also encompasses advanced techniques such as profile-guided optimization (PGO), interprocedural optimization (IPO), and whole-program optimization (WPO). PGO involves using runtime profiling information to guide optimization decisions, enabling LLVM to optimize code based on its actual runtime behavior. IPO extends optimization across function boundaries, enabling LLVM to perform optimizations that require global analysis and transformation. In addition, there are target-specific optimization passes and heuristics, which enables tailoring of optimizations to the characteristics of the target hardware architecture.

2.4 LLVM Application Programming Interface

The LLVM API offers a wide range of functionalities for compiler development, optimization, and code generation tasks.

IR Building and Manipulation

The API provides a set of classes and functions for constructing and manipulating LLVM IR. Construction of the IR using the API typically follow a sequence of steps:

1. Creating a context: An LLVM context acts as a central repository for all the global data that is used by the LLVM infrastructure during the compilation process. This includes types, constants, and other entities that need to be unique within a certain scope. When using LLVM for any compilation task, an LLVM context must first be created. This context is then passed around to various LLVM components to ensure they all operate within the same context. The context manages types and constants. This ensures that identical types and constants are not redundantly recreated. For example, all integer types of a given bit width are managed within the context, and the same type object is reused. An example is given in Listing 2.11. The context is

```
1 llvm::Type* int32Type = llvm::Type::getInt32Ty(context);
```

Listing 2.11: A 32-bit integer type is determined from a context.

needed for creating a module. Each module is associated with a context. This ensures all elements within the module use the same context for consistency. Contexts also manage diagnostic information and error handling. When errors occur, the context can be queried to provide detailed diagnostic messages, which can be helpful for debugging and improving the reliability of the compilation process.

2. Creating a module: An LLVM module acts as a top-level container for all the global variables, functions, type definitions, and other data used in a program. Like mentioned before, an LLVM context is required to create a module. This ensures that all the elements within the module share the same context, providing consistency and efficient resource management. Modules contain functions and global variables, which

```
1 llvm::LLVMContext context;
2 llvm::Module module = llvm::Module("module", context);
```

Listing 2.12: Creation of a module from a context.

are added to the module during the code generation phase. Each function and global variable has a unique name within the module. Modules can include metadata, which provides additional information

about the code. Metadata is often used for optimization, debugging, and analysis purposes. Modules can be serialized to disk as LLVM bitcode files, which can later be deserialized back into LLVM modules. This feature is useful for saving the IR of a program for further analysis or compilation. Modules can be linked together. This is useful for modular compilation and linking of large programs. Modules are often the primary unit of optimization in LLVM. Various optimization passes can be applied to a module to improve the performance and reduce the size of the generated code.

3. Creating functions: An LLVM function is an entity that contains a series of basic blocks, which in turn contain a series of instructions. It is similar to a function in a high-level programming language, but represented in LLVM's IR. Each function has a type, name, linkage type, and attributes that define its properties and behavior. Functions can be defined or declared in an LLVM module. A function definition includes the body (i.e., the sequence of basic blocks), while a declaration only specifies the function's type and linkage without providing the body. The linkage type of a function determines its visibility and linkage to other modules. Common linkage types include `ExternalLinkage`, `InternalLinkage`, and `PrivateLinkage`. Functions can have various attributes that provide additional information to the compiler about the function's behavior, such as `inline`, `noinline`, `readonly`, `readnone`, etc. Functions can call other functions, and this is represented in LLVM IR using the `call` instruction. Function calls are a central mechanism for composing more complex behavior from simpler functions and intrinsincs. Functions can take parameters, and these parameters are used within the function body. By analyzing function calls and their interactions, LLVM can perform optimizations across function boundaries. This leads to more efficient programs. Functions in LLVM are strongly typed. An example of creation of a function is given in Listing 2.13.

4. Creating 'Basic Blocks': An LLVM basic block is a container for a sequence of instructions. Each basic block has a list of instructions and a terminator instruction, which determines the control flow to other basic blocks. Basic blocks are linked together to form the control flow of a function. This dictates the execution order of the blocks. Basic blocks are created within the context of a function. Each function consists of one or more basic blocks. Typically, a function starts with an entry block. Creation of a basic block is given in Listing 2.13. Instructions are added to a basic block. The order of instructions within a block is important as they execute sequentially. Sequential execution is described in the next step. Basic blocks are connected to form the control flow graph (CFG) of a function. The CFG defines the possible paths through the code

```
llvm::FunctionType *funcType = llvm::FunctionType::get(
  llvm::Type::getVoidTy(context), false
);
llvm::Function *func = llvm::Function::Create(
  funcType, llvm::Function::ExternalLinkage,
  "function", module
);
llvm::BasicBlock *entry =
  llvm::BasicBlock::Create(context, "entry", func);
```

Listing 2.13: Creation of a function and a basic block. Creating a basic block requires a function and the context.

based on conditional and unconditional branches. Basic blocks have predecessors (blocks that can branch to them) and successors (blocks they can branch to). These relationships are crucial for control flow analysis and optimization. For example, the Φ function relies on these to decide on its values.

5. Adding Instructions: An LLVM instruction is an operation that produces a value or modifies the control flow within a function. Instructions can perform a wide range of tasks, from arithmetic and logical operations to memory access, control flow management, and function calls. Instructions are created and added to basic blocks using the `IRBuilder` utility, which simplifies the process of generating IR code. An example is provided in Listing 2.14.

```
llvm::IRBuilder<> builder(entry);
llvm::Value *val = builder.CreateAdd(
  llvm::ConstantInt::get(
    llvm::Type::getInt32Ty(context), 1
  ), llvm::ConstantInt::get(
    llvm::Type::getInt32Ty(context), 2
  )
);
```

Listing 2.14: Creation and addition of an instruction to a basic block using the `IRBuilder` utility.

Each instruction has strict type requirements as described in Section 2.2. Instructions within a basic block can be iterated over for analysis or transformation, as given in Listing 2.15.

6. Verifying the function: This verification step is crucial because it helps catch errors early in the development process. It ensures that the generated IR is valid and can be successfully translated into machine code by the LLVM backend. LLVM provides a built-in verifier pass that

```
19 for (llvm::Instruction &I : *entry)
20 {
21   // Do something with I
22 }
```

Listing 2.15: Iteration over instructions in a basic block.

can be used to verify functions, modules, or even individual instructions. This verifier pass performs a series of checks and reports any errors it finds. It implements checks like SSA form, the proper use of Φ nodes and ensures that the types of operands match the expected types for each instruction, and checks the function arguments and return types are consistent with the function's declaration. The verifier also ensures that each basic block ends with a terminator instruction (e.g., `ret`, `br`, `switch`). It verifies that branches and jumps target valid basic blocks within the function. It checks that all reachable blocks are properly connected.

Intrinsic instruction verification is an important step taken by the verifier to ensure they are used correctly. Here, it validates intrinsic functions and their arguments. It also checks that they only use operands that are in scope and of the correct type. Function call verification checks for non matching function signatures and calls. It also verifies that indirect calls use function pointers of the correct type.

Memory access verification ensures that pointers used in memory instructions are properly typed and that load and store instructions operate on valid memory locations. However, it is important to note that the verifier does not handle the check for aligned loads and stores. Depending on the implementation, non-aligned loads and stores may cause errors like segmentation faults during execution, even if the compilation is successful, particularly on architectures like `x86_64`.

7. Printing LLVM IR: To print the IR in LLVM, one typically uses the `print` method provided by the IR objects, such as Module or Function. These methods output the textual representation of the IR to a specified stream.

```
23 module->print(llvm::outs(), nullptr);
```

Listing 2.16: Printing IR in a module to an output stream.

Modifying constructed LLVM IR involves manipulating existing IR constructs through a series of methodical steps:

1. Accessing IR constructs: References to existing IR constructs (e.g., functions, basic blocks, instructions) are obtained using the LLVM API. This typically involves traversing the IR hierarchy or querying the module for specific constructs.
2. Creating an IR Builder: IR builder for a specific basic block is created and used to construct new instructions as described in step 5 of IR building and manipulation. The created IR builder does not disconnect the basic block from the existing IR.
3. Removing Instructions: To remove instructions from the IR, target instructions within the basic block are located and erased from the parent container. The traversal method described in step 5 can also be used for this purpose. This operation ensures that the IR remains valid and well-formed after modification.

Machine Code Generation with the API

Central to LLVM's code generation capabilities is its target machine interface. It abstracts the low-level details and provides a uniform interface for generating optimized machine code. It specifies the target architecture, CPU and features of the targetmachine. The target triple is a string that encodes information about the target platform for which the machine code is being generated. The name 'tripe' was given, because it originally had three parts in its signature. Today, it can have between two to five parts. The three main parts are the architecture, vendor and operating system. The architecture specifies the CPU architecture, like `x86_64` or `arm`. The vendor indicates the CPU, such as `pc` or `apple`. The operating system (OS), as its name suggests, could be `linux`, `darwin` (macOS), `windows`, or another OS. The syntax is given in Listing 2.20. For example, `arm64-apple-macosx14.0.0` indicates the target

```
1 <arch><sub_arch>-<vendor>-<sys>-<env>
```

Listing 2.17: Target triple syntax.

machine is arm architecture of 64-bit system. The vendor is `apple` and the OS is macOS version 14.0.0. The target triple helps LLVM understand the specific features and constraints of the target platform. It is essential for the correct configuration of the toolchain, including the selection of the appropriate libraries, calling conventions, and binary formats. All the available values for the target triple are listed in the `llvm::Triple` class.

The features mentioned earlier specify the capabilities or extensions of the CPU. In some cases, these can be queried from the host itself using `lvm::TargetMachine::getTargetFeatureString()` method. In other cases,

they can be queried using `llvm::sys::getHostCPUFeatures( StringMap< bool, MallocAllocator > & Features)`. `Features` is a string mapping that maps feature names to either true (if enabled) or false (if disabled). This routine makes no guarantees about exactly which features may appear in this map, except that they are all valid LLVM feature names [14]. One can use the mapping to iterate over the features with their support status. The features could include support for vector instructions, atomics, hardware acceleration features, or other specialized capabilities.

Target options is a required field for generating a `TargetMachine` in the framework. The `llvm::TargetOptions` provides a way to customize the behavior of the target machine and can be used to fine-tune the generated machine code for specific requirements. `UnsafeFPMath` is a target option set to `false` by default. If set to true, it enables unsafe floating-point math optimizations. `NoInfsFPMath` assumes floating-point math will not produce infinities. This is also set to `false` by default. `NoTrappingFPMath` specifies whether floating-point operations should be allowed to 'trap' (i.e., cause exceptions or interrupts) on errors such as overflow, underflow, or division by zero. This is set to `true` by default, which implies that the target machine will not be 'trapping'. Other options are also available and have default values.

The data layout is another component of the target machine that describes the memory layout conventions used by the target, such as byte order, alignment requirements, and pointer sizes. It ensures that the generated code adheres to the platform's application binary interface (ABI). Endianness (byte order) of the target can be `e` for little-endian, where least significant byte has the lowest address or `E` for big-endian, where most significant byte has the lowest address. `m:e` or `m:o` are parts of the layout that specify the mangling mode or how symbol names are mangled. `m:e` uses Executable and Linkable Format (ELF) mangling, which are typically used in Unix-like systems. `m:o` uses Objective-C mangling, used for macOS and `m:m` for Microsoft mangling on Windows platforms. Type alignment and size define the amount for pointers for each address space. They specify the bit size and ABI alignment. Aggregate alignment for structures and array, vector alignment for vector types and stack alignment for stack variables also exist. In Listing 2.18, an example data layout for an arm64 machine is

```
1 e-m:o-i64:64-i128:128-n32:64-S128
```

Listing 2.18: Example of a target data layout.

presented. There, `e` is for little-endian, `m:o` is the mangling mode and `i64:64` specifies that 64-bit integers are used with an alignment of 64 bits. Similarly,

`i128:128` specifies that 128-bit integers have a 128-bit alignment. `n32:64` indicates that native integer width of 32 and 64 bits are supported by the target machine. This information is used by the compiler to decide on the optimal integer size for various operations. The `S128` specifies that the stack should be aligned to 128 bits (16 bytes). Stack alignment is important for ensuring efficient access to stack-allocated variables and maintaining ABI compatibility. The API allows manual configuration of the data layout. This can be done using `llvm::Module::setDataLayout( StringRef Desc )`. A string formatted similar to Listing 2.18 can be passed in place of `Desc` to set the target machine data layout.

The relocation model is another component of the target machine. A relocation model specifies how the linker and loader handle the addresses of functions and data within a program. It defines the rules and mechanisms for adjusting address references when the program is loaded into memory. The choice of relocation model affects the performance, memory usage, and compatibility of the generated code with the architecture. Static relocation determines the code and data addresses at link time. The addresses are resolved once, and the program is loaded at a specific location in memory. This lacks flexibility, but is simple and efficient, because there is no need for address adjustment at runtime. In dynamic relocation, addresses are not fixed at link time. Instead, they are adjusted at runtime by the loader. This model is more flexible and allows programs to be loaded at different memory addresses. This comes with additional overhead for address adjustment at runtime. Position-independent code (PIC) allows code to be executed regardless of its absolute address. The code uses relative addressing or indirection through tables to access functions and data. It is essential for shared libraries and certain security measures, as it allows multiple instances of the code to share the same memory. But, it is slightly less efficient due to the use of indirect addressing and additional indirection.

When setting up a target machine in LLVM, the relocation model is specified. There are `Static`, `PIC_`, `DynamicNoPIC` and some other options. Static relocation is typically the fastest, however, PIC is crucial for shared libraries and enhances security by enabling address space layout randomization (ASLR). PIC can lead to more efficient memory usage in systems where multiple instances of the same code are loaded. The target architecture's addressing modes, memory layout, instruction set complexity, and platform ABI conventions all play critical roles in determining the most suitable relocation model.

Optimization Passes

Optimizations can be applied at various stages of the compilation process to improve the performance and efficiency of the generated code as described

in Section 2.3. Optimization passes are added to a `PassManager`, which then runs these passes on the module or function. In Listing 2.19, the instruction

```
1 llvm::legacy::FunctionPassManger funcPassManager;
2 funcPassManager.add(llvm::InstCombinePass());
3 funcPassManager.run(*func);
```

Listing 2.19: Optimization passes can be added to a pass manager, which can then be run.

combine pass is added to the function pass manager, which is then run on the function. Other analyses and transformation passes can also be added in a similar way. Creating custom passes is also possible by interacting with the API and registering the pass.

Emitting Machine Code

Once the module is optimized, the code generation process begins. Instruction selection is the first step, where LLVM IR instructions are translated into target-specific machine instructions. Instruction selection is a complex process that maps high-level operations to the corresponding low-level instructions supported by the target architecture. LLVM uses pattern matching and heuristics to choose the most efficient instructions. The next step is to allocate physical registers to hold the values used by the instructions. LLVM's register allocator assigns the virtual registers used in the IR to physical registers, considering constraints such as register pressure and the target architecture's register file. Then, the instructions are scheduled to minimize pipeline stalls and maximize instruction-level parallelism. Instruction scheduling takes into account the target architecture's pipeline structure and execution units to arrange instructions in an optimal sequence. Finally, localized optimizations are done that further refine the machine instructions. Peephole optimizations look at small windows of instructions and apply transformations to improve performance.

After the code generation passes are completed, the machine instructions are ready to be emitted as machine code. The Machine Code (MC) layer is responsible for converting the machine instructions into a binary format that can be executed by the target hardware. The MC layer includes components such as the assembler, linker, and object file writer. The assembler translates the assembly-like machine instructions into binary machine code. It handles tasks such as encoding instructions, resolving symbols, and generating relocation information. The object file writer generates the final object file, which contains the machine code with metadata such as symbol tables and relocation entries. The object file can then be linked with other object files to produce an executable. The linker combines multiple object files into a

single executable or library. It resolves symbol references, performs address binding, and generates the final binary. The code in Listing 2.20 sets up a

```
1 std::error_code EC;
2 llvm::raw_fd_ostream dest("output.o", EC, llvm::sys::fs::OF_None);
3 llvm::legacy::PassManager codeGenPass;
4 TargetMachine->addPassesToEmitFile(
5   codeGenPass, dest, nullptr,
6     llvm::CodeGenFileType::CGFT_ObjectFile
7 );
8 codeGenPass.run(*module);
9 dest.flush();
```

Listing 2.20: Generating an object file from LLVM IR.

`raw_fd_ostream` to write the output to a file and adds the necessary passes to the PassManager to emit an object file. The PassManager's run method generates the machine code and writes it to the specified file. In addition to object files, there is support for `llvm::CodeGenFileType::CGFT_AssemblyFile` and in latest versions `llvm::CodeGenFileType::AssemblyFile`. Using the LLVM API, configuration of modules that are created in the IR builing phase, setting of parameters such as target triple, data layout, and optimization level to tailor code generation for specific platforms and optimization goals can be enabled. By emitting machine code, LLVM completes the transformation from abstract program representation to executable object or assembly ready for deployment and execution.

Chapter 3

Finite Impulse Response (FIR) Filters

This chapter presents the essential concepts of FIR filters, starting with two fundamental principles of filtering: convolution and cross-correlation. We will then compare linear and circular convolution, highlighting their respective implications. Finally, we will delve into the design and implementation of low-pass FIR filters, discussing their significance, implementation and practical applications.

The FIR filter is a non-recursive filter that possesses a finite impulse response, which makes it suitable for applications requiring linear phase response and precise control over frequency characteristics. It operates by convolving an input signal with a finite-length impulse response, which is a sequence of coefficients that defines how the filter responds to an impulse input. The impulse response of a filter describes how it reacts over time to a single, idealized impulse signal (a signal consisting of a single spike or "impulse" followed by zeros). This sequence determines the filter's output for any arbitrary input signal. These coefficients determine the filter's frequency response and shape its behaviour in the frequency domain. They are carefully designed based on desired filter specifications, such as cutoff frequency, passband ripple, and stopband attenuation.

The cutoff frequency is an important parameter in FIR filter design, which defines the boundary between the filter's passband and stopband. The *passband* is the range of frequencies that the filter allows to pass through with minimal attenuation, meaning these frequencies experience only slight reduction in amplitude. Conversely, the *stopband* is the range of frequencies that the filter significantly attenuates or blocks, thereby minimizing the impact of unwanted frequencies. The *cutoff frequency* sets the point at which the filter transitions from allowing signals to pass through to significantly attenuating them. *Passband ripple* refers to any variations in gain within the passband, while *stopband attenuation* measures the extent of attenuation applied to frequencies in the stopband. By adjusting these parameters, the

FIR filter achieves the desired balance between passing and attenuating different frequency components of the signal.

A defining characteristic of an FIR filter is its linear phase response, which ensures that all frequency components of the input signal experience uniform delay, preserving the signal's integrity and avoiding phase distortion. This property makes FIR filters particularly well-suited for applications where phase linearity is critical, such as in image and audio processing, and digital communications. Its stability is also a critical aspect that impacts its performance and reliability in signal processing applications. Stability refers to the boundedness of the filter's impulse response. Unlike an IIR filter, which requires careful consideration of pole locations inside the unit circle in the z-plane, an FIR filter inherently possesses a stable impulse response. This simplifies its design and implementation.

FIR filters can be implemented using various techniques, including windowing, frequency sampling, and optimization algorithms. These techniques aim to design filter coefficients that meet desired frequency response specifications while minimizing filter order and computational complexity.

3.1 Convolution

Convolution is a fundamental operation in the realm of of digital filtering. The convolution operation represents the mathematical operation of combining two signals to produce a third signal that represents the "overlap" of the two input signals. In the context of FIR filters, convolution is used to process input signals through the filter's impulse response to generate output signals. By convolving the input signal with the filter's impulse response, the filter effectively applies its frequency response characteristics to the input signal, thereby shaping its spectral content according to the filter's design specifications.

Mathematically, the convolution operation is represented as the integral of the product of the input signal $x(t)$ and the impulse response $h(t)$ of a filter, where t is time:

$$y(t) = (x \circledast h)(t) = \int_{-\infty}^{\infty} x(t-\tau)h(\tau)d\tau. \tag{3.1}$$

In discrete-time systems, convolution is represented using a sum:

$$y[n] = (x \circledast h)[n] = \sum_{m=-\infty}^{\infty} x[n-m]h[m]. \tag{3.2}$$

In the context of FIR filters, the convolution is finite:

$$y[n] = (x \circledast h)[n] = \sum_{m=0}^{M-1} x[n-m]h[m]. \tag{3.3}$$

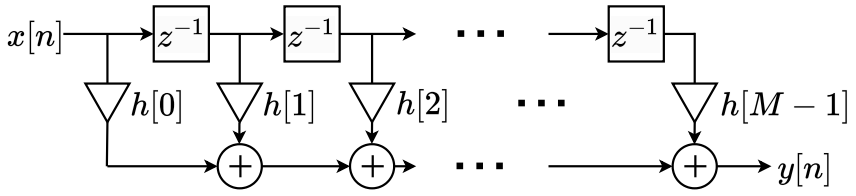


Figure 3.1: General, causal, length M, FIR digital filter [20].

Figure 3.1 shows the discretized FIR convolution, where x is the input signal, h represents the filter coefficients, z^{-1} is a unit time delay and y is the output signal. The process begins by multiplying the first input sample by the first filter coefficient. Each subsequent input sample is delayed by one unit and multiplied by the next filter coefficient. This continues until all filter coefficients are used, after which the results are summed, producing the output. The figure corresponds exactly to the finite sum given in (3.3).

3.2 Cross-correlation

Cross-correlation is a fundamental operation in signal processing that measures the similarity between two signals or a signal and a filter as a function of the displacement of one relative to the other. In the context of FIR filters, cross-correlation is utilized to provide insights into how well the filter matches certain characteristics or features of the input signal.

Mathematically, the cross-correlation is represented as the integral of the product of the input signal $x(t)$ and the impulse response $h(t)$ of a filter:

$$y(t) = (x \star h)(t) = \int_{-\infty}^{\infty} x(t+\tau)h(\tau)d\tau. \tag{3.4}$$

For discrete-time systems:

$$y[n] = (x \star h)[n] = \sum_{m=-\infty}^{\infty} x[n+m]h[m]. \tag{3.5}$$

In FIR filters, the cross-correlation is finite:

$$y[n] = (x \star h)[n] = \sum_{m=0}^{M-1} x[n+m]h[m]. \tag{3.6}$$

The cross-correlation operation provides valuable information about the similarity between the input signal and the filter's impulse response at different time offsets. A high cross-correlation value at a particular displacement indicates a strong similarity between the input signal and the portion of the

filter's impulse response corresponding to that displacement. This can be interpreted as the filter responding strongly to features or characteristics in the input signal that align with the corresponding portion of its impulse response.

Linear Convolution/Correlation vs Circular Convolution/-Correlation

Linear convolution is the most common form of convolution encountered in signal and image processing. Linear convolution is typically performed using techniques such as the convolution sum or a fast convolution algorithm (e.g., Fast Fourier Transform - FFT). It operates on signals of finite length, with the output signal being of length $N - M$, where N and M are the lengths of the input signal and the impulse response, respectively. This is achieved by starting at the end sample of the input signal and truncating the iteration at the last sample that contains the entire filter windowed.

Circular convolution, also known as cyclic convolution, differs from linear convolution in that it operates cyclically. Instead of extending the signals with zero-padding, circular convolution wraps the signals around in a circular manner, assuming that the signal repeats infinitely in both directions. This is implemented by wrapping the signal around. To do this in computer CPU's, the memory needs to be accessed in a linear fashion. The circular buffer needs to be flattened. Efficient circular convolution often necessitates adjusting sequence sizes to powers of 2 and introducing extra zero-padding, which can complicate implementation and resource allocation, particularly in scenarios where such adjustments are not feasible or efficient. Therefore, this project opts for linear *cross-correlation* due to its greater flexibility in handling sequences of arbitrary lengths without the need for size manipulation or padding. The filtering process is designed to stop if there are insufficient input samples remaining, ensuring that only complete sequences larger than the filter size are processed.

Low-pass Filters

Lowpass FIR filters are fundamental components in digital signal processing, serving to attenuate high-frequency components while preserving low-frequency components within a signal. The impulse response of a lowpass FIR filter typically exhibits a symmetric, finite-duration characteristic.

The window method is a popular technique for designing lowpass FIR filters. It involves multiplying an ideal impulse response (e.g., sinc function) with a window function (e.g., Hamming, Blackman) to obtain the desired filter response. The length of the filter and the choice of window function

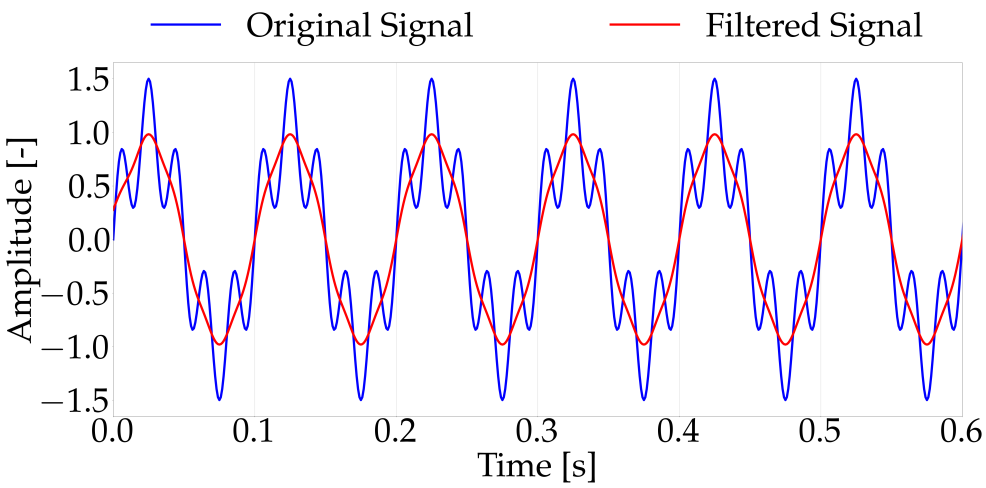


Figure 3.2: The original signal contains both desired frequency components and unwanted noise. After applying a lowpass filter, only the desired low-frequency components remain, while the high-frequency noise is removed.

dictate the frequency transition bandwidth—range of frequencies over which the filter transitions from the passband to the stopband—and the stopband attenuation of the filter. The frequency sampling method is an alternative that directly specifies the desired frequency response of the filter at discrete frequency points. These frequency points are sampled from the desired frequency response curve, and an IDFT is used to obtain the filter coefficients. Optimization-based approaches, such as least squares optimization or constrained optimization, can also be employed to design lowpass FIR filters with specific performance criteria, such as minimum ripple in the passband or maximum attenuation in the stopband.

Lowpass FIR filters are commonly used for smoothing noisy signals, removing high-frequency noise while preserving the underlying signal. As an example, let us consider Figure 3.2. There, a signal that includes both a low-frequency component and a high-frequency component is presented. The low-frequency component is at 10 Hz, and the high-frequency component is at 50 Hz. The plot illustrates how the signal varies over time, showing the superposition of these two sine waves. The resulting waveform demonstrates how the amplitudes of the desired signal and unwanted noise combine. The purpose of this figure is to show the raw signal before and after filtering is applied, highlighting the presence of both the intended low-frequency signal and the high-frequency noise. The figure also presents the signal after it has been processed through a lowpass FIR filter. The filter is designed to attenuate frequencies above a certain cutoff, which in this case is 30 Hz. The resulting filtered signal primarily retains the low-frequency component while significantly reducing the high-frequency noise. This plot

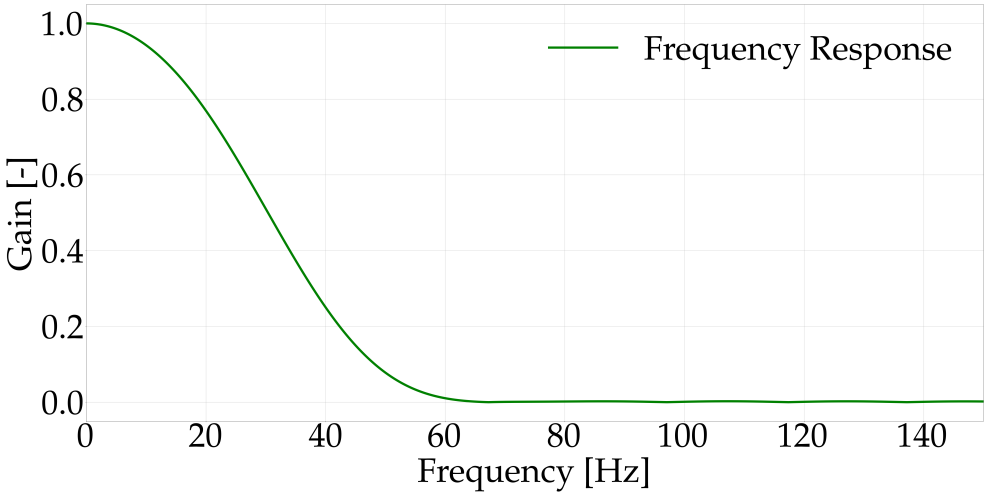


Figure 3.3: Frequency response of a lowpass FIR filter with an order of 50 and a cutoff frequency of 30 Hz, showing positive frequencies.

demonstrates the effectiveness of the lowpass filter in removing unwanted frequency components from the original signal.

Figure 3.3 shows the frequency response of the lowpass FIR filter, focusing on positive frequencies. The plot illustrates how the filter attenuates frequencies above the cutoff frequency while allowing lower frequencies to pass through with minimal attenuation. The frequency response confirms the filter's design to effectively separate the low-frequency component from the unwanted high-frequency noise.

For the purpose of this thesis, lowpass FIR filters will specifically be similarly employed to achieve noise reduction without compromising the original signal.

Chapter 4

High-Performance Computing

This chapter provides an overview of HPC techniques, focusing on optimizing computational efficiency. We begin with instruction-level parallelism, exploring methods to enhance instruction throughput within a single processor. Next, we delve into data-level parallelism, including vectorization. We also examine loop tiling, a technique to improve cache utilization and reduce memory access latency. Thread-level parallelism and the use of distributed computing will be discussed to illustrate how tasks can be parallelized across multiple processors. Finally, we will introduce the roofline model, a performance analysis plot that helps in understanding and optimizing the balance between computational and memory-bound operations.

4.1 Instruction-Level Parallelism

Since the mid-1980s, processors have incorporated pipelining to execute multiple instructions simultaneously, thereby enhancing overall performance. This technique, known as instruction-level parallelism (ILP), allows instructions to be processed in parallel [7]. There are two main approaches to harnessing ILP: (1) dynamically identifying and exploiting parallelism through hardware, and (2) statically detecting parallelism at compile time using software.

Hardware ILP is implemented using multiple-issue processors, which can issue multiple instructions per clock cycle. These processors fall into two categories: superscalar processors and VLIW (very long instruction word) processors. Superscalar processors can issue a varying number of instructions per clock cycle and may use in-order execution if statically scheduled or out-of-order (OoO) execution if dynamically scheduled. Conversely, VLIW processors issue a fixed number of instructions, either as one large instruction or as a fixed packet, with parallelism explicitly indicated by the instruction [7]. VLIW processors rely on static scheduling by the compiler. Statically scheduled superscalar processors are less advantageous compared

to their dynamically scheduled counterparts when dealing with more than two instructions. As a result, ILP is typically implemented with dynamic scheduling (OoO) utilizing replicated resources, particularly execution ports, or using VLIW in most designs.

On the software side, the potential for parallelism within a single *basic block*—a linear sequence of code without branches, akin to LLVM's basic block—is limited. Significant performance gains require exploiting ILP across multiple basic blocks. The most common method to increase ILP is by exploiting parallelism among loop iterations, known as *loop-level parallelism*. This can be achieved by unrolling loops, either statically by the compiler or dynamically by the hardware.

Basic Compiler Techniques for Exposing ILP

Basic compiler techniques play a critical role in processors that utilize static scheduling or issue. To maintain an efficient pipeline, it is essential to leverage parallelism among instructions by identifying sequences of unrelated instructions that can be executed concurrently. To prevent pipeline stalls, it is necessary to ensure that the execution of a dependent instruction is spaced apart by a number of clock cycles equivalent to the pipeline latency of the preceding instruction. The effectiveness of a compiler in scheduling depends on the degree of ILP present in the program and the latencies associated with the pipeline's functional units. Listing 4.1 demonstrates a basic C operation to illustrate methods for enhancing ILP through transformations. Observing this loop reveals that it exhibits parallelism, as each iteration's operations are independent of one another. The Microprocessor without Interlocked Pipeline

```
1 double s = 0;
2 for (int i = 100; i > 0; --i)
3   s += x[i];
```

Listing 4.1: Reduction operation that sums elements of an array.

Stages (MIPS) assembly language translation without loop initializations is presented in Listing 4.2. In the beginning, `$5` holds the address of an element in `x`. `s` is represented by register `$f2`, and the counter by `$7`. To reduce the number of costly additional clock cycles, it's essential to increase the ratio of operations to overhead instructions. One effective method for achieving this is *loop unrolling*, which involves duplicating the loop body multiple times and modifying the loop termination logic accordingly. This technique not only reduces the overhead from branch instructions but also enhances scheduling by allowing instructions from different iterations to be executed concurrently. By removing the branch, loop unrolling can help

```
1 loop:
2   l.d     $f0, 0($5)        ; $f0 = x[i]
3   add.d   $f2, $f2, $f0     ; $f8 = $f2 + x[i]
4   daddui  $5, $5, #-8       ; decrement pointer for x[i]
5   addi    $7, $7, -1        ; decrement loop counter
6   bgtz    $7, loop          ; Continue if loop counter > 0
```

Listing 4.2: MIPS Assembly equivalent snippet of Listing 4.1.

avoid data stalls by creating more independent instructions within the loop body. However, simply replicating instructions during unrolling can lead to conflicts if the same registers are reused. Therefore, to effectively schedule the loop, different registers must be used for each iteration, which necessitates an increased number of registers.

The unrolled version of Listing 4.2 by a factor of 4 is presented in Listing 4.3. The unrolling factor dictates the stride size and the number of times the loop gets repeated.

```
1 loop:
2   l.d     $f0, 0($5)      ; iteration with displacement 0
3   add.d   $f2, $f2, $f0   ; drop daddui, addi & bne
4
5   l.d     $f4, -8($5)     ; iteration with displacement 8
6   add.d   $f2, $f2, $f4   ; drop daddui, addi & bne
7
8   l.d     $f6, -16($5)    ; iteration with displacement 16
9   add.d   $f2, $f2, $f6   ; drop daddui, addi & bne
10
11  l.d     $f8, -24($5)    ; iteration with displacement 24
12  add.d   $f2, $f2, $f8
13  daddui  $5, $5, #-32    ; displacement at -32
14  addi    $7, $7, -4      ; decrement by 4
15  bgtz    $7, loop        ; Continue loop if $7 > 0
```

Listing 4.3: Unrolled version of Listing 4.2 with an unroll factor of 4.

Here, three branches and decrements of `$5` and `$7` are removed. The addresses used for loads are adjusted to enable the merging of `daddui` instructions involving `$5`. Although this might seem straightforward, it involves complex operations such as symbolic substitution and simplification. These techniques rearrange expressions to consolidate constants, transforming an expression like `((i + 1) + 1)` into `(i + (1 + 1))` and subsequently simplifying it to `(i + 2)` [7]. Typically, loop unrolling is performed early in the compilation process to reveal and remove redundant computations through optimization.

As demonstrated, *full* unrolling removes loop control overhead but increases code size and register pressure. In contrast, *interleaved* unrolling involves interleaving multiple iterations of the loop body, a technique often used in Single Instruction Multiple Data (SIMD) architectures. An example of interleaved unrolling in a scalar (non-SIMD) context is shown in Listing 4.4, which implements the unrolled and interleaved version of the loop code from Listing 4.2. This implementation avoids stalls and yields even greater performance improvements compared to the fully unrolled version. Effective scheduling in this way requires that the loads be independent and capable of being reordered.

```
1  loop:
2    l.d      $f0, 0($5)       ; iteration with displacement 0
3    l.d      $f4, -8($5)      ; iteration with displacement 8
4    l.d      $f6, -16($5)     ; iteration with displacement 16
5    l.d      $f8, -24($5)     ; iteration with displacement 24
6
7    add.d    $f2, $f2, $f0
8    add.d    $f2, $f2, $f4
9    add.d    $f2, $f2, $f6
10   add.d    $f2, $f2, $f8
11
12   daddui   $5, $5, #-32     ; displacement at -32
13   addi     $7, $7, -4       ; decrement by 4
14   bgtz     $7, loop         ; Continue loop if $7 > 0
```

Listing 4.4: Interleaved unrolling of the reduction operation in Listing 4.2.

By reducing the number of loop iterations, loop unrolling *and* interleaving effectively decreases the overhead associated with loop control instructions, such as loop counters, loop bounds checks, and loop branch instructions. This reduction in loop overhead results in a more streamlined execution flow, enabling faster computation. Moreover, loop unrolling can improve memory access patterns by reducing loop trip counts. This enables more effective prefetching and caching of loop data, which enhances cache utilization and reduces memory latency. Lastly, unrolled loops facilitate compiler optimizations (LLVM) by providing more opportunities for optimizations such as LICM, strength reduction, and CSE. These optimizations further enhance performance by reducing redundant computations and improving code efficiency.

4.2 Data-Level Parallelism

Another method for leveraging loop-level parallelism is the use of SIMD in both vector processors and GPUs (Graphics Processing Units). SIMD

instructions boost data-level parallelism by executing the same operation on multiple data elements simultaneously. Vector instructions further enhance data-level parallelism by processing a large number of data elements concurrently using parallel execution units and deep pipelines. For instance, let's look at the code sequence in Listing 4.5. In its straightforward form, this sequence requires seven instructions per iteration: two loads, an addition, a store, two address updates, and a branch, totaling 7000 instructions for 1000 iterations [7]. With a SIMD architecture that can process four data items per instruction, this sequence could be executed with only a quarter of the instructions. On some vector processors, the sequence might need only four instructions: two to load vectors `x` and `y` from memory, one to add these vectors, and one to store the resulting vector. Even though these instructions are pipelined and have long latencies, they can take advantage of overlapping latencies for improved performance [7].

```
1 for (int i = 0; i <= 999; ++i)
2   x[i] = x[i] + y[i];
```

Listing 4.5: A simple example of a loop that adds two 1000-element arrays [Hennessy et al. [7]].

Data-Level Parallelism (DLP) focuses on executing identical operations simultaneously across multiple data elements, unlike ILP. A key optimization technique for DLP is vectorization, which involves performing multiple arithmetic or logical operations concurrently on *vectors* of data elements using vector instructions. Vectorization is especially useful for applications with regular, data-parallel computations, such as matrix operations, signal processing, and numerical simulations. By processing multiple data elements in a single instruction, vectorized operations can effectively hide memory access latencies and improve overall efficiency. They also reduce overhead by performing multiple operations with a single instruction.

To achieve effective code vectorization, algorithms and code structures must be modified to fully utilize vector instructions. This includes restructuring loops, optimizing memory access patterns, and reducing dependencies between data elements. Tools like intrinsics (LLVM) and pragmas can assist with this process. Intrinsics allow for direct insertion of vector instructions into the code, while pragmas provide hints to the compiler about how to vectorize the code.

The SAXPY or DAXPY (Single- or double-precision AX+Y) problem that forms the inner loop of the Linpack benchmark, which is a collection of linear algebra routines for performing Gaussian elimination [7], is given by:

$$D = a \times X + Y. \tag{4.1}$$

The code for MIPS and VMIPS (vectorized) versions for the DAXPY loop are presented in Listings 4.6 and 4.7, respectively, for comparison.

```
1     l.d     $f0, a           ; load scalar a
2     daddiu  $r4, $rx, #512   ; last address to load
3 loop:
4     l.d     $f2, 0(rx)       ; load X[i]
5     mul.d   $f2, $f2, $f0    ; a * X[i]
6     l.d     $f4, 0(ry)       ; load Y[i]
7     add.d   $f4, $f4, $f2    ; a * X[i] + Y[i]
8     s.d     $f4, 9($ry)      ; store into Y[i]
9     daddiu  $rx, $rx, #8     ; increment index to X
10    daddiu  $rx, $ry, #8     ; increment index to Y
11    dsubu   $r20, $r4, $rx    ; compute bound
12    bnez    $r20, loop
```

Listing 4.6: MIPS Assembly Code for DAXPY Loop [Hennessy et al. [7]].

```
1 l.d      $f0, a           ; load scalar a
2 lv       $v1, rx          ; load vector x
3 mulvs.d  $v2, $v1, $f0    ; vector-scalar multiplication
4 lv       $v3, ry          ; load vector Y
5 addvv.d  $v4, $v2, $v3    ; add
6 sv       $v4, $ry         ; store vector result
```

Listing 4.7: Vectorized MIPS Assembly Code for DAXPY Loop [Hennessy et al. [7]].

The most significant contrast lies in the substantial reduction in dynamic instruction bandwidth when using a vector processor compared to MIPS. The vector processor executes only 6 instructions, whereas MIPS executes nearly 600 instructions [7]. This reduction is due to vector operations handling 64 elements at once, whereas MIPS must process many overhead instructions, which make up nearly half of the loop's instructions. Loops can be vectorized as long as there are no dependencies between iterations. If dependencies do exist, special registers are used to manage them.

Another key difference between MIPS and VMIPS is the frequency of pipeline interlocks. In straightforward MIPS code, each `add.d` instruction must wait for a `mul.d` instruction, and each `s.d` instruction must wait for the preceding `add.d`. In contrast, on the vector processor, each vector instruction only experiences a stall for the first element of the vector. Consequently, pipeline stalls occur once per vector instruction, rather than once per vector element. This method, known as *chaining*, significantly reduces the frequency of pipeline stalls. In this scenario, MIPS will experience approximately 64 times more pipeline stalls than VMIPS [7]. While techniques such as software pipelining or loop unrolling can help mitigate pipeline stalls in MIPS, they

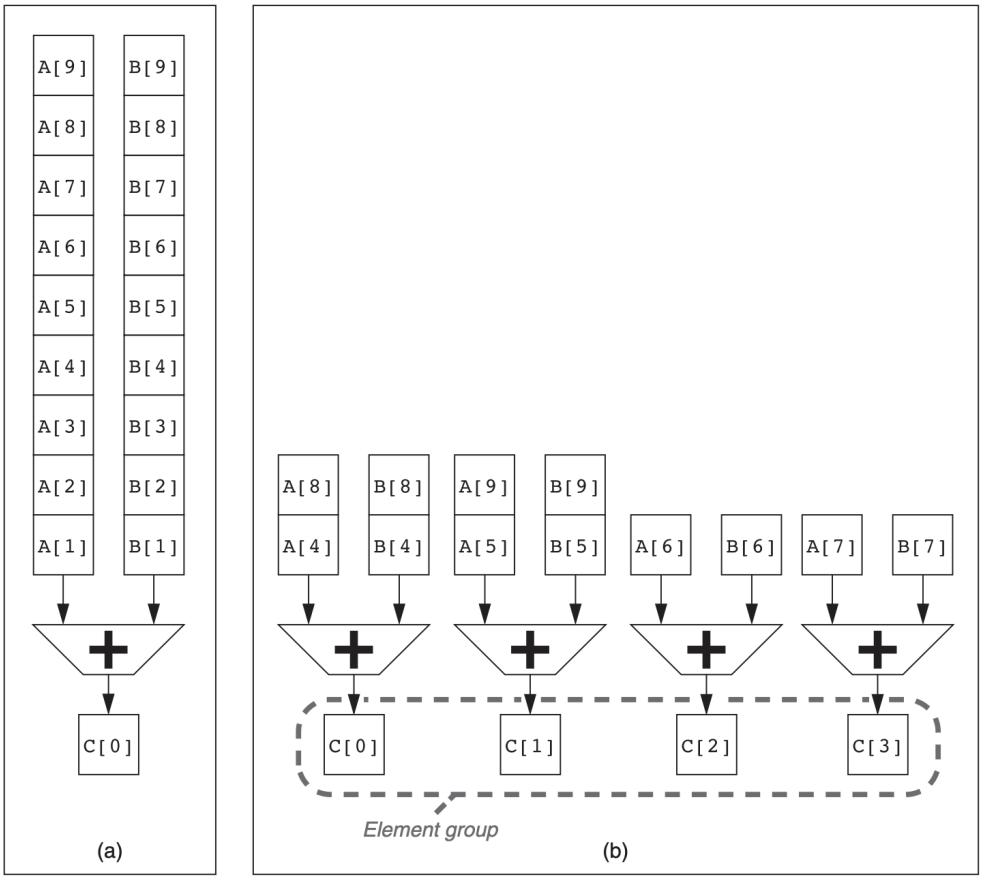


Figure 4.1: Using multiple functional units can significantly enhance the performance of a single vector addition instruction, $A + B = C$. Consider two vector processors: the first (a) on the left has a single addition pipeline and can complete one addition per cycle. In contrast, the second (b) on the right has four addition pipelines and can complete four additions per cycle. In the latter, the elements within a single vector add instruction are interleaved across the four pipelines. The group of elements processed concurrently through these pipelines is referred to as an element group. This interleaving allows for more efficient parallel processing, improving overall computational throughput [Hennessy et al. [7]].

cannot substantially close the gap in instruction bandwidth. Performance can be further enhanced through parallel execution, depending on the number of available SIMD arithmetic units. Figure 4.1 demonstrates how vector performance can be improved by using parallel pipelines to execute a simple vector addition instruction.

Handling Loops Not Equal to the Maximum Vector Length

A vector register processor has an inherent vector length defined by the number of elements in each SIMD register. In practice, the length of a vector operation is often not known until runtime, and a single code segment may

need to handle varying vector lengths. Listing 4.8 illustrates the C translation of the DAXPY operation from (4.1). In this example, the size of all vector

```
1 for (int i = 0; i < n; ++i)
2   y[i] = a * x[i] + y[i];
```

Listing 4.8: DAXPY C code.

operations depends on `n`, which is only known at runtime and may vary. A special register called the maximum vector length (MVL) controls the length of any vector operation. To maximize efficiency, the MVL should be fully utilized. For the code in Listing 4.8, *strip mining* can be used. Strip mining generates code such that each vector operation is performed on a size less than or equal to the MVL. The strip-mined version of Listing 4.8 is shown in Listing 4.9. The effect is that the vector is divided into segments, which are

```
1 low = 0;
2 VL = (n % MVL); /*find odd-size piece using modulo op %*/
3 for (j = 0; j <= (n/MVL); ++j) { /*outer loop*/
4   for (i = low; i < (low+VL); ++i) /*runs for length VL*/
5     Y[i] = a * X[i] + Y[i] ; /*main operation*/
6   low = low + VL; /*start of next vector*/
7   VL = MVL; /*reset the length to maximum vector length*/
8 }
```

Listing 4.9: Strip-mined version of DAXPY C code [Hennessy et al. [7]].

then processed by the inner loop. The length of the first segment is `n % MVL`, while all subsequent segments have a length equal to the MVL.

It is important to point out that in SIMD, the number of elements processed in parallel is determined by the instruction's opcode, which corresponds to a fixed register width. This means SIMD instructions cannot dynamically adjust the number of elements processed based on runtime conditions. As a result, SIMD cannot directly handle the code in Listing 4.9, which adjusts the vector length dynamically. Vectorization, however, can handle it, and is implemented in RISC-V architectures.

SIMD Instruction Set Extensions for Multimedia

In contrast to vector machines like the VMIPS, which feature large register files capable of holding sixty-four 64-bit elements in each of its 8 vector registers, SIMD instructions generally specify fewer operands and therefore use smaller register files. Extensions for multimedia SIMD, such as those found in the x86 architecture (MMX (Multimedia Extensions), SSE (Streaming

SIMD Extensions), and AVX (Advanced Vector Extensions)), fix the number of data operands in the opcode, leading to hundreds of additional instructions [7]. Unlike vector architectures, multimedia SIMD typically does not include mask registers for conditional execution of elements.

The x86 architecture's Streaming SIMD Extensions (SSE) introduced 128-bit wide registers and enabled parallel processing of single-precision floating-point arithmetic. Intel expanded these capabilities to double-precision floating-point data types with SSE2 in 2001, SSE3 in 2004, and SSE4 in 2007 [7]. AVX increased the register width to 256 bits, doubling the number of operations on narrower data types. AVX512 further extended the register width to 512 bits, doubling the capacity again.

Although multimedia SIMD extensions have limitations compared to vector processors, they are prevalent in many applications due to their low cost and ease of implementation. They require minimal additional state compared to vector architectures and do not need as much bandwidth as vector processors, nor do they deal with virtual memory.

Listing 4.10 provides MIPS SIMD code for the DAXPY loop on a MIPS architecture with a 256-bit SIMD multimedia instruction extension. The new `mov` operations copy data, and `l.4d` and `l.4s` represent vectorized double-precision loads and stores. The `add.4d` and `mul.4d` operations perform vectorized arithmetic. While SIMD MIPS achieves a 4× reduction in dynamic instruction bandwidth compared to standard MIPS (149 instructions executed versus 578), it is not as significant as the 100× reduction achieved by VMIPS [7].

```
1   l.d F0,a ;load scalar a
2   mov F1, F0 ;copy a into F1 for SIMD MUL
3   mov F2, F0 ;copy a into F2 for SIMD MUL
4   mov F3, F0 ;copy a into F3 for SIMD MUL
5   daddiu R4,Rx,#512 ;last address to load
6
7 Loop:
8   l.4d F4,0(Rx) ;load X[i], X[i+1], X[i+2], X[i+3]
9   mul.4d F4,F4,F0 ;a*X[i],a*X[i+1],a*X[i+2],a*X[i+3]
10  l.4d F8,0(Ry) ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
11  add.4d F8,F8,F4 ;a*X[i]+Y[i], ..., a*X[i+3]+Y[i+3]
12  s.4d F8,0(Rx) ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
13  daddiu Rx,Rx,#32 ;increment index to X
14  daddiu Ry,Ry,#32 ;increment index to Y
15  dsubu R20,R4,Rx ;compute bound
16  bnez R20,Loop ;check if done
```

Listing 4.10: MIPS SIMD code for DAXPY, assuming that the starting addresses of `x` and `y` are in `Rx` and `Ry`, respectively [Hennessy et al. [7]].

4.3 Memory Architecture

Memory architecture refers to the organization and structure of a computer's memory subsystems, encompassing primary memory (RAM), cache memory, and secondary storage (such as disk drives or SSDs). A typical memory architecture consists of multiple tiers of memory with differing access speeds, capacities, and costs. This memory hierarchy includes registers, cache memory, main memory (RAM), and secondary storage.

Cache memory is a small, high-speed memory subsystem situated between the CPU and main memory. Its primary function is to store frequently accessed data and instructions, thereby reducing memory access latency and enhancing system performance. Cache memory is typically organized into three levels: L1, L2, and L3 caches, each offering progressively larger capacities and longer access latencies. The organization of cache includes cache lines, cache sets, cache associativity, and cache replacement policies, all of which determine how data is stored, accessed, and evicted from the cache. The efficiency and performance of the cache are greatly influenced by memory access patterns, including spatial locality and temporal locality. Spatial locality is the tendency of programs to access memory locations that are near each other, while temporal locality refers to the repeated access of previously accessed memory locations over time. Cache optimization techniques aim to enhance spatial and temporal locality by optimizing data layout, memory access patterns, and overall cache utilization. Cache lines, which are the smallest units of data transferred between main memory and cache, typically range in size from 64 to 128 bytes. These lines are loaded into the cache as units, and cache operations occur at this granularity. When the cache capacity is exceeded, frequent cache misses and evictions can occur, leading to cache thrashing, which degrades performance. Cache thrashing can be alleviated by optimizing cache organization, reducing cache contention, and improving spatio-temporal locality. Prefetching is a hardware technique used to anticipate future memory accesses by loading data into the cache in advance, thus reducing access latency and improving cache hit rates. Prefetching can be implemented at various levels of the memory hierarchy, including hardware prefetching in CPU caches and software prefetching in application code. However, it is largely the responsibility of the software to manage memory access patterns efficiently to avoid cache misses.

Loop Tiling/Blocking

Loop blocking, or loop tiling, is a powerful optimization technique used to improve spatio-temporal locality, thereby reducing memory access latency, and enhancing data reuse in loop-based computations. It involves partitioning loop iterations into smaller, contiguous blocks of work, known as *tiles* or

blocks, which are then processed sequentially. By dividing the computation into smaller tiles, loop blocking improves spatio-temporal locality by reducing cache thrashing. This leads to significant performance improvements, particularly in memory-bound applications. Loop blocking exploits temporal locality by ensuring that data accessed within each tile remains in cache for reuse across iterations. It allows computations to be performed on subsets of data multiple times within the cache, rather than fetching data repeatedly from main memory. Loop blocking aligns tile sizes with cache sizes to optimize cache utilization. Spatial locality is implicitly exploited using cache lines. When data is loaded into the cache, it is organized in cache lines, which store contiguous blocks of data. Accessing one piece of data in a cache line often brings adjacent data into the cache. Here, all the data within that block is used before moving on to the next block.

Techniques for Loop Blocking

Techniques that can be used to implement loop blocking include loop interchange, loop fusion and fission, loop peeling, and padding and handling boundary conditions. Loop interchange reorders nested loops to facilitate loop blocking by bringing the innermost loop, which accesses data elements in the outer dimensions of the data arrays, to the outer positions. Loop fusion combines multiple loops into a single loop, while loop fission splits a single loop into multiple loops, enabling more flexible control over loop blocking granularity and tile sizes. Loop peeling handles loop iterations, at the boundaries of the iteration space, separately to accommodate partial tiles. Padding techniques are used to handle boundary conditions and ensure that tiles are properly aligned with the original data array boundaries, preventing out-of-bounds memory accesses.

A typical way of implementing blocking is using additional loops that fit the cache sizes, as described earlier. Listing 4.11 shows a non-tiled matrix-vector multiplication implementation and Listing 4.12 shows a blocked version of the same multiplication using the inner loop as a blocking loop and a tile size, `tile`, which fits in the cache. For optimal performance, cache sizes should be fully utilized. The blocked version listed can also be tiled again over `i`. Listing 4.13 presents a version that does blocking over both loop indices.

```
1 for (int i = 0; i < n; ++i)
2     for (int j = 0; j < m; ++j)
3         y[i] += A[i][j] * x[j];
```

Listing 4.11: Matrix-Vector multiplication code in C.

```
for (int jj = 0; jj < m; jj += tile)
{
    int je = min((jj+tile), m);
    for (int i = 0; i < n; ++i)
        for (int j = jj; j < je; ++j)
            y[i] += A[i][j] * x[j];
}
```

Listing 4.12: Matrix-vector multiplication after inner loop blocking.

```
for (int ii = 0; ii < n; ii += tile)
{
    int ie = min((ii+tile), n);
    for (int jj = 0; jj < m; jj += tile)
    {
        int je = min((jj+tile), m);
        for (int i = ii; i < ie; ++i)
            for (int j = jj; j < je; ++j)
                y[i] += A[i][j] * x[j];
    }
}
```

Listing 4.13: Matrix-vector multiplication after loop tiling over both indices.

4.4 Thread Level Parallelism

Thread-level parallelism (TLP) involves executing multiple threads on cores. By dividing a process into multiple threads, each performing a subset of the tasks, TLP can lead to significant performance improvements, especially in applications that are designed to run concurrently. A software implementation of TLP involves multiprogramming parallelism, i.e. multiple input multiple data (MIMD). To take advantage of a MIMD multiprocessor with n processors, there need to be at least n threads or processes to execute [7]. Managing synchronization between threads is crucial to avoid issues such as race conditions and deadlocks. Proper mechanisms are needed to ensure safe data sharing between threads.

OpenMP

OpenMP (Open Multi-Processing) is an API designed to facilitate multi-platform shared-memory parallel programming in C, C++, and Fortran. By offering a collection of compiler directives, library routines, and environment variables, OpenMP simplifies the creation of parallel code. Multithreading, a common parallelization technique supported by OpenMP, involves a master thread that spawns multiple slave threads to distribute a task among them. These threads execute concurrently, with the runtime environment managing

their assignment to different processors. Listing 4.14 illustrates a basic OpenMP utilization.

```
#include < stdio.h >

int main(void)
{
  #pragma omp parallel
  {
    printf("Hello, world.\n");
  }

  return 0;
}
```

Listing 4.14: OpenMP example that displays "Hello World" using multiple threads.

There are different constructs in OpenMP. The work-sharing construct is prominent. It distributes the execution of code across threads without creating additional threads. In the construct, the `for` loop directive splits the iterations of a loop among threads. An example is shown in Listing 4.15.

```
#include <omp.h>
#include <stdio.h>

int main()
{
  #pragma omp parallel for
  for (int i = 0; i < 10; i++)
    printf("Thread %d: i = %d\n", omp_get_thread_num(), i);
  return 0;
}
```

Listing 4.15: OpenMP example that splits iterations among threads.

There are also other directives, such as Sections, and constructs, which we do not cover here.

4.5 Message Passing Interface

Message Passing Interface (MPI) is a standardized communication protocol used for parallel programming and distributed computing. MPI enables processes running on different compute nodes to exchange data and coordinate their execution through message passing.

MPI enables processes to communicate directly with each other using point-

to-point communication operations such as `MPI_Send` and `MPI_Recv`. MPI supports collective communication operations, such as `MPI_Bcast`, `MPI_Reduce`, and `MPI_Allreduce`, which allow processes to exchange data collectively and synchronize their execution. MPI also provides functions for creating, managing, and terminating parallel processes, enabling dynamic process creation and control. It includes mechanisms for error detection, error reporting, and fault tolerance.

4.6 The Roofline Model

The Roofline Model is an intuitive and utilitarian model to bound attainable performance [24]. It provides a representation of the performance achievable by a computational kernel as a function of operational intensity and memory bandwidth limitations. The model visualizes performance ceilings (rooflines) imposed by hardware constraints and identifies performance bottlenecks to guide optimization efforts effectively.

The Roofline Model consists of two primary components: rooflines and performance data points. Rooflines represent theoretical performance ceilings imposed by hardware constraints, such as computing capability (peak Floating Point Operations (FLOP)) and memory bandwidth limitations. Different rooflines correspond to different levels of computational workload, with steeper rooflines indicating higher computational efficiency. Performance data points represent actual performance measurements of computational kernels plotted on the Roofline graph. These data points are typically characterized by their operational intensity (FLOP/Byte) and achieved performance (FLOP/s).

Interpreting the Roofline Model

Interpreting the Roofline Model involves analyzing the relative position of performance data points with respect to the rooflines to identify performance bottlenecks and optimization opportunities [6].

Computational intensity, typically expressed in terms of FLOP per byte (FLOP/Byte), occupies the lower spectrum of the Roofline Model. It encapsulates the relationship between computational workload and memory access overhead within computational kernels. For example, in Listing 4.8, there are 2 FLOP per iteration and 3 load operations and 1 store operation per iteration. Assuming that all of them are floating point numbers, the computational intensity is equal to $2/(4 \cdot (3+1)) = 0.125$ FLOP/Byte. Higher computational intensity values indicate a higher ratio of computation to memory traffic, suggesting that the algorithm is compute-bound and may benefit from optimizations targeting computational efficiency and parallelism.

There are two types of computational intensities: operational intensity and arithmetic intensity. Operational intensity is defined as the ratio of FLOP to the amount of data transferred (in bytes) between the main memory and the processor. Arithmetic intensity is similarly defined as the ratio of computational operations to the amount of data movement, but it can be more broadly applied to different levels of the memory hierarchy, including cache. It focuses on the efficiency of data usage within the processor's internal memory hierarchy (e.g., L1, L2, and L3 caches).

On the upper segment, performance metrics illustrate the achieved performance of computational tasks. The Roofline Model delineates performance limitations and optimization opportunities based on the relative positioning of these data points with respect to the rooflines :

Memory-bound: Data points located before the plateau, indicate that performance is limited by memory bandwidth constraints. To improve performance, optimizations should focus on increasing data reuse, increasing computational intensity reducing memory accesses, and optimizing memory access patterns to better utilize available memory bandwidth.

Compute-bound: Data points lying close to the roofline peak represent performance achievable by the computational kernel given the hardware constraints. To maximize performance, optimizations should focus on, exploiting parallelism, and optimizing algorithmic efficiency to move data points closer to the core bottleneck.

Steps to go from memory to compute bound and reach peak performance as portrayed in Figure 4.2 follow:

1. Attain maximum achievable bandwidth by adopting Non-Uniform Memory Access (NUMA) aware strategies (process placement memory and placement policies libraries such as `libnuma` and `memkind`), by using wide registers for data transfer (burst mode), as well as relying on write combining or non-temporal stores and software prefetch to overlap calculations with load instructions [17].
2. Increase the operational intensity by adopting a proper memory layout that decreases the off-chip memory traffic. For example, using temporal blocking.
3. Improve the performance by increasing the DLP and ILP and mitigating register and port pressure.

Plotting the Roofline

To plot the roofline, the peak performance P_{peak}, the memory bandwidth of the underlying architecture denoted as b_s, and the operational intensity OI

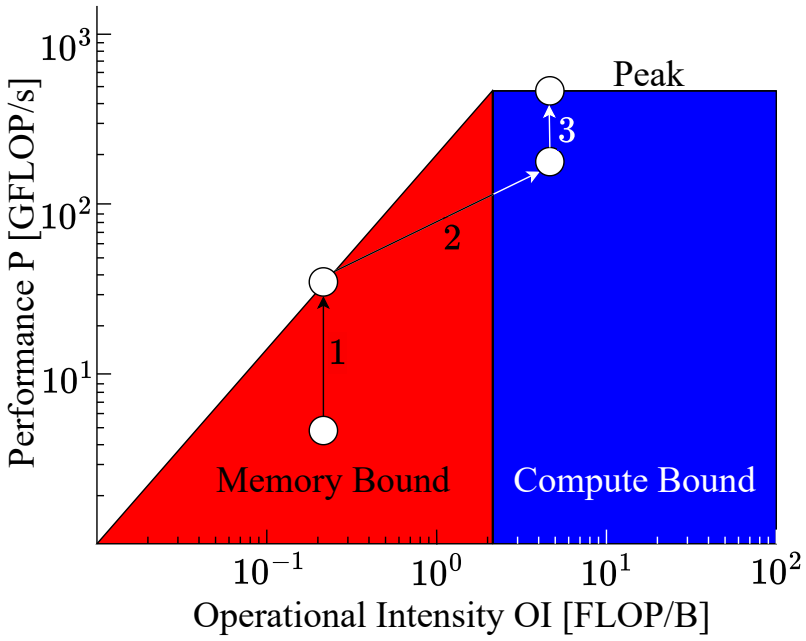


Figure 4.2: Performance optimization steps based on roofline.

are required.

Peak performance of a computing architecture is given by:

$$P_{\text{peak}} = P_{\text{chip}} = n_{\text{core}} \times n_{\text{super}}^{\text{FP}} \times n_{\text{FMA}} \times n_{\text{SIMD}} \times f. \tag{4.2}$$

Here, n_{core} is the total number of processing cores. n_{super}^{FP} is the n-way replication of the execution port (superscalarity). n_{FMA} denotes the number of Fused Multiply-Add (FMA) ports available. n_{SIMD} is the SIMD register width. And f is the clock speed measured in Hertz. For example, for a 2P Xeon Gold 6240, the peak performance is:

$$(2 \times 18)(\text{cores}) \times 2 \times 2(\text{fma}) \times 16(\text{SIMD width}) \times 2.4\text{GHz} = 5.5\text{TFLOP/s}.$$

The peak bandwidth can be calculated using the following:

$$\begin{aligned}\text{Peak Memory Bandwidth} = b_s = \text{Memory Clock Speed} \times \text{Bus Width} \\ \times \text{Number of Channels} \times \text{DDR}.\end{aligned} \tag{4.3}$$

Here, the Clock Speed is the effective clock speed of the memory module, the Bus Width is the number of bytes of width, the Number of Channels is as its name suggests and DDR is the factor for the data rate [5]. For example, the Intel Core X-Series Processors DDR4 2933 has a maximum

clock speed of 1466.67MHz. Its bus width is 64 bits (8 bytes) per channel and has 4 channels. The DDR factor is 2, for double data rate. That means, $b_s = 1466.67 \times 8 \times 2 \times 2 = 93,866.68$ MB/s, or 94 GB/s.

Then, the roofline plot can be drawn using the following:

$$P = \min(P_{\text{peak}}, OI \cdot b_s),$$

where OI is the operational intensity.

Chapter 5

High-Performance FIR Filtering using LLVM

This chapter introduces the implementation of FIR filtering using the LLVM API, integrating concepts from previous discussions on HPC techniques and the LLVM framework. Building on our understanding of LLVM IR and optimization capabilities, we explore how these principles can be applied to optimize FIR filter design. We'll demonstrate how to leverage ILP and DLP through interleaved unrolling, vectorization, and loop tiling to enhance computational efficiency. Additionally, we'll incorporate MPI to distribute workload across multiple cores. Finally, we will discuss the specifications of the computing architectures used for running our benchmarks and schemes.

5.1 Project Setup

Setting up the project involved several important steps. Firstly, the LLVM framework was installed. Detailed instructions for installing LLVM on various platforms were followed according to the official documentation to ensure a standardized development environment.

A significant part of the project, involved familiarization with the LLVM IR by directly implementing various operations, thus bypassing the frontend parsing stage described in Chapter 2. These operations encompassed tasks like reduction, element-wise addition, vectorized element-wise addition, and matrix-matrix multiplication. Each operation was carefully crafted, leading to a thorough grasp of the LLVM IR and its functionalities. This hands-on approach facilitated a deeper understanding of the IR's capabilities and its practical applications within the project context.

Programmatically Writing LLVM IR

In order to streamline the creation of efficient LLVM IR code for implementing FIR filters across various filter sizes, the LLVM API, elaborated in Section 2.4, was employed. This involved the *generation* of LLVM IR through calls to the C++ API. A C++ based project utilizing the Make build system was constructed for each component of the manually crafted kernels mentioned.

In this section, we present an example of *synthesis* of LLVM IR for a "Hello World" program using the API that was an early part of the project. To begin, context and module initializations are performed in a shared location accessible by all methods. In the `hello` method, references to the context and modules are utilized. Listing 5.1 showcases the types necessary for the `hello` method to initialize the parent function within the IR. In this segment, the `Int8PtrType` is utilized to initialize the `puts` function responsible for printing the string. Listing 5.2 illustrates the function return and parameter types,

```
1 // Types
2 llvm::Type *VoidType = llvm::Type::getVoidTy(context);
3 llvm::Type *Int32Type = llvm::Type::getInt32Ty(context);
4 llvm::PointerType *Int8PtrType = llvm::PointerType::getUnqual(
5   llvm::PointerType::getInt8Ty(context)
6 );
```

Listing 5.1: Type initializations for function return and parameter types in a "hello world" LLVM IR code generation.

which leverage the types presented in Listing 5.1. The `VoidType` designation

```
7 // Function Return and Param Types
8 llvm::FunctionType *funcType = llvm::FunctionType::get(
9     VoidType, false
10 );
11 llvm::FunctionType *printfType = llvm::FunctionType::get(
12     Int32Type,
13     Int8PtrType,
14     true
15 );
```

Listing 5.2: The functions return and parameter types are initialized using integer and void types.

indicates that the function returns a void type, while `false` indicates that it does not accept variable arguments. Similarly, the type definition for the printing function, `printfType`, denotes a return type of 32 bits and expects a parameter represented by a pointer to an 8-bit integer. This pointer typically refers to constant arrays within LLVM. The inclusion of `true` specifies that

the function accepts variable arguments in addition to the pointer to the 8-bit integer. Listing 5.3 presents the creation of the two functions, whose types we just described. There, as described in Section 2.4, the function type,

```
16 // Create Functions
17 llvm::Function *helloFunc = llvm::Function::Create(
18     funcType, llvm::Function::ExternalLinkage, "hello", module
19 );
20 llvm::Function *printfFunc = llvm::Function::Create(
21     printfType, llvm::Function::ExternalLinkage, "puts", module
22 );
```

Listing 5.3: Function creations for the parent and for the printing functions.

linkage type, name of the function and module are provided as parameters to facilitate the creation of functions. Listing 5.4 demonstrates the basic block creation that is a child of the parent function. There, creating the basic block requires passing in `helloFunc`. The name of the block is not visible

```
23 // Entry Block - Name is not visible in the IR
24 llvm::BasicBlock *entry = llvm::BasicBlock::Create(
25     context, "", helloFunc
26 );
```

Listing 5.4: Creation of a `BasicBlock` that is contained in `hellofunc`.

in the synthesized IR if an empty string is passed as a label name. Listing 5.5 illustrates the initialization of the IR builder, followed by the setting of insertion point within the basic block. After this, the string is created using the `CreateGlobalStringPtr` command available in the IR builder. The identifier "`str`" serves as the name of the virtual register and is of type `llvm::Twine`. The command returns a `llvm::Constant` type, which is then used to call the `puts` function, alongside the `llvm::Twine` name.The use of a common name, "`puts`", for the printing function and the register that calls the function does not cause any issues. At the end of the `hello` method, the

```
27 llvm::IRBuilder<> builder(context);
28 // Write IR
29 builder.SetInsertPoint(entry);
30 llvm::Constant *str =
31   builder.CreateGlobalStringPtr("Hello, World!\n", "str");
32 builder.CreateCall(printfFunc, {str}, "puts");
33 builder.CreateRetVoid();
```

Listing 5.5: Initialization and use of the LLVM `IRBuilder` to create instructions.

return type is set to void. In this context, there is no need to explicitly use the `VoidType` pointer, as it is managed internally during the instruction creation process.

The module verification is performed using `llvm::verifyModule(*module, &llvm::errs())`. This function returns a boolean value indicating whether there is an issue with the module creation, as detailed in Section 2.4. If a problem is detected, the error is printed to a valid stream such as `llvm::errs()` or `llvm::outs()`. These streams are part of LLVM's infrastructure, directing output to the standard error stream and standard output stream, respectively.

The corresponding LLVM IR generated from the previous listings is presented in Listing 5.6.

```
34 @str = private unnamed_addr constant [15 x i8] c"Hello, World↩
       !\0A\00", align 1
35
36 define void @hello() {
37   %puts = call i32 (i8*, ...) @puts(i8* getelementptr inbounds ↩
         ([15 x i8], [15 x i8]* @str, i32 0, i32 0))
38   ret void
39 }
40
41 declare i32 @puts(i8*, ...)
```

Listing 5.6: Generated LLVM IR from C++ API calls for a "Hello World" example.

In all cases, a common implementation for creating the target machine was utilized. Object generation, as described in Section 2.4, was then employed to create object files. This universal implementation applied several passes, such as setting the optimization level and enabling vectorization. Listing 5.7 presents the IR code just before compilation for the "Hello World" example. Here, the pass manager adds additional attributes and metadata about different members of the IR. This aids in the machine code generation or object file emission, which are then used to be linked or run on the machine.

5.2 Kernel Synthesis using the API

This section outlines the generation of FIR filtering kernels using the LLVM API, following synthesis similar to the earlier "Hello World" example. This forms the core focus of this thesis. Throughout the project, all computations are performed using 32-bit floating-point precision (FP32).

To implement FIR filtering, a signal was generated using FFTW's C libraries, which offer the necessary tools for DSP tasks. The signal used is a 2π-periodic cosine wave. To test the filtering scheme, noise generated from a normal

```
42 @str = private unnamed_addr constant [15 x i8] c"Hello, World↩
       !\0A\00", align 1
43
44 ; Function Attrs: nofree nounwind
45 define void @hello() local_unnamed_addr #0 {
46   %puts = tail call i32 (i8*, ...) @puts(
47     i8* nonnull dereferenceable(1) getelementptr inbounds (
48       [15 x i8], [15 x i8]* @str, i64 0, i64 0
49     )
50   )
51   ret void
52 }
53
54 ; Function Attrs: nofree nounwind
55 declare noundef i32 @puts(i8* nocapture noundef readonly, ...) ↩
       local_unnamed_addr #0
56
57 attributes #0 = { nofree nounwind }
```

Listing 5.7: Optimized LLVM IR of a "Hello World" example, prepared for compilation and object file generation after applying optimization passes.

distribution with a mean of 0 and a standard deviation of 0.1 was added to the signal. A low-pass filter with the impulse response given by the following equation was used:

$$H[k] = \begin{cases} 1, & \text{if } k = 0, \\ \dfrac{sin(\omega_k)}{\omega_k - \dfrac{\omega_k^3}{\pi^2} + \varepsilon}, & \text{if } k \neq 1, \\ 0.5, & \text{if } m\%L == 0, \end{cases} \tag{5.1}$$

where m is the filter size, $\omega_k = \dfrac{k\pi}{m}$ are the frequency bins, $\varepsilon = 10^{-9}$ is used to prevent division by 0, i.e, for numerical stabilty, and L is a parameter to control the behavior of the low-pass filter—larger L results in smoother filtered output. The filter was implemented in C to avoid file I/O and to keep operations in memory, utilizing FFTW's `fftwf_plan_dft_c2r_1d`. This function performs a DFT similar to Python's `numpy.fft`, but requires normalization. Using the given equation, filter coefficients are determined based on the values of m and L.

Subsequently, LLVM IR was generated programmatically using the LLVM C++ API. This involved a `switch-case` statement at the root of the IR. The default case of the statement directs execution to a basic block that performs FIR cross-correlation by applying element-wise scalar FMA calls based on the specified filter coefficient size. For the other cases, implementations specific to different filter sizes were generated. This involved creating code

that uses vectorized FMAs. Each filter element is broadcast into a vector and used as operands in the vectorized multiplications or FMAs. The first element vector is used for the `fmul` operation, while the subsequent vectors are used as operands in the vectorized FMAs. For instance, for a filter size of 8 with an unroll factor of 2, the IR performs two vectorized `fmuls` and 14 vectorized FMAs. The input is loaded into vectors using an interleaved unrolling technique, which greatly boosts performance as detailed in Section 4.1. Both the vector size and unroll factor that correspond to the computing hardware are determined at compile time.

To implement all options, various helper functions and classes were utilized. One such function is `mkPHI`, which is essential for managing control flow within loops. As described in Section 2.2, this function generates Φ nodes based on the predecessors of the given basic block. The `LoopWorkload` class, which serves as a base for handling loop workloads, manages basic blocks for loop checks and bodies, providing a structure for specific loop-based computations. The `DotP` class is used for handling delegated dot products from the default case of the switch statement. The `FIR` class manages the default scalar FIR filtering, as described earlier, while the `FIR_UV` class implements the unrolled and vectorized cases. Figure 5.1 presents the UML diagram for the described setup.

The scalar version, implemented by the `FIR` class, performs one-by-one filtering in LLVM, similar to the C code in Listing 5.8. This implementation does not utilize vectorization and relies solely on *scalar* FMAs. Although *vector reduction* will be applied by the compiler frontend if the C code in the listing is compiled with optimization enabled, this listing is mentioned at this point solely for illustration. The vectorized version of this C code is used as the *baseline* vanilla FIR cross-correlation. The non-vectorized and non-unrolled version of this baseline is illustrated in Figure 5.2.

The vectorized and interleaved-unrolled version of our LLVM scheme, implemented by the `FIR_UV` class, employs an unrolled and vectorized filtering scheme that leverages ILP and DLP, which are topics covered in Sections 4.1 and 4.2. An equivalent pseudocode for a filter size of 20, with unroll and vectorization factors of 4, is provided in Listing 5.9. This listing is included solely for illustration. It assumes that the output size is at least the product of the unroll factor, `UF`, and the vector factor, `VF`. Each line processes $\texttt{UF} = 4$ elements at a time, resulting in a total of $\texttt{UF} \times \texttt{VF} = 16$ elements per iteration. An automated M4 based pseudocode similar to this can be generated, as shown in [16], although only small filter sizes could be synthesized accurately and effectively.

At the end of this initial, but promising implementation, IR verification was performed to ensure correctness, and the IR was written to a file rather than using the object generation described in Section 2.4 and in the previous

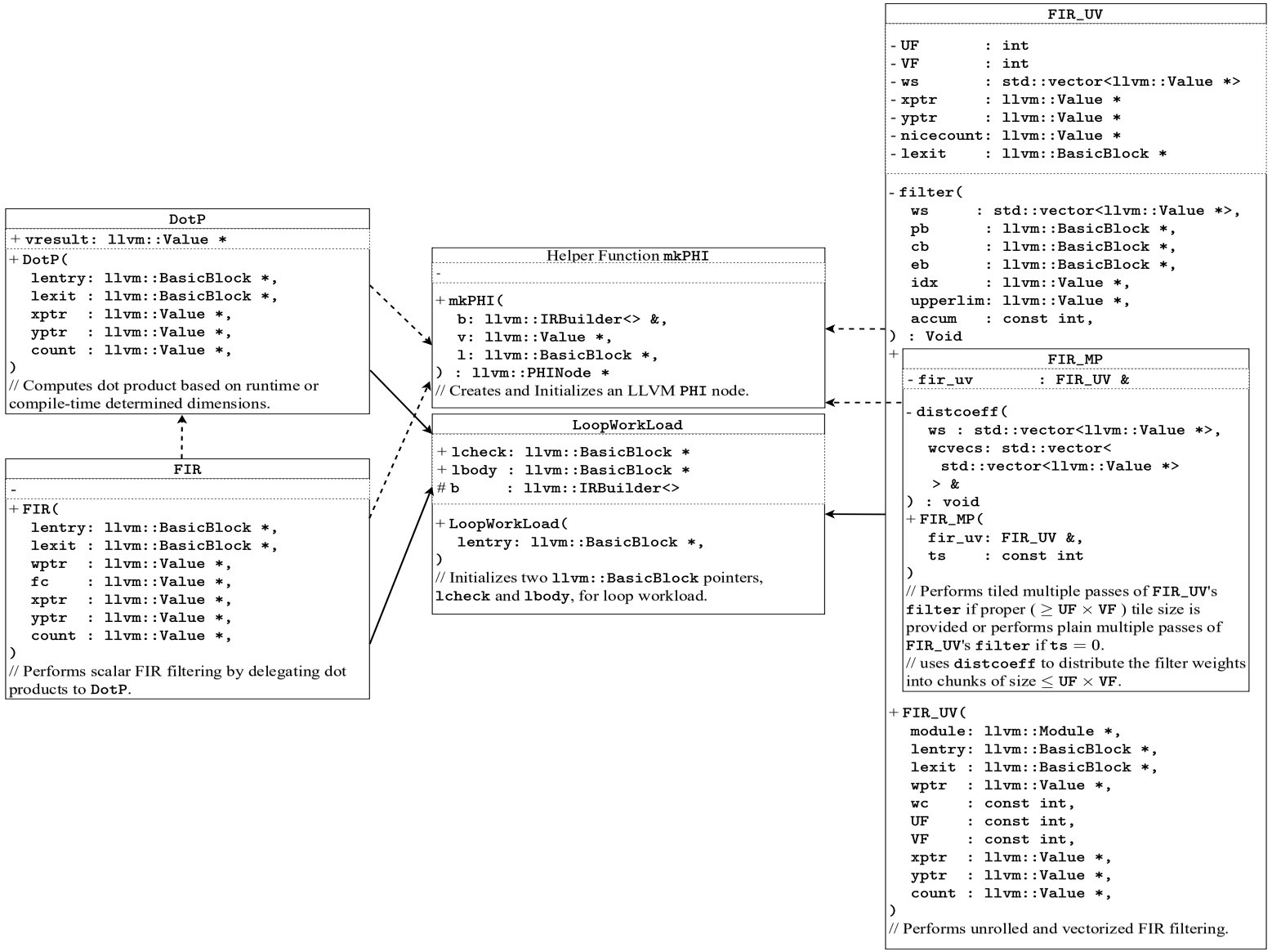


Figure 5.1: Unified Model Language (UML) class diagram of the classes and functions used in the project. Dashed lines of a class indicate the use of the class or method it points to, and solid lines imply inheritance from the class it points to.

```
1  void fir
2  (
3  /* input sample count */
4  const int isc,
5  /* input samples */
6  const float * in,
7  /* filter coefficient count */
8  const int fcc,
9  /* filter coefficients */
10 const float * w,
11 /* output samples */
12 float * out
13 )
14 {
15   const int osc = isc - fcc;
16
17   for (int i = 0; i < osc; ++i)
18   {
19     float s = 0;
20
21     for (int j = 0; j < fcc; ++j)
22       s += w[j] * in[i + j];
23
24     out[i] = s;
25   }
26
27   for (int i = osc; i < isc; ++i)
28     out[i] = 0;
29 }
```

Listing 5.8: The C baseline implementation. Its strictly scalar version is equivalent to `FIR` class implementation.

```
1 for(int i = 0; i < outputsize; i+=16)
2 {
3   out[i:i+3] = in[i:i+3]*w[0]+...+in[i+19:i+22]*w[19];
4   out[i+4:i+7] = in[i+4:i+7]*w[0]+...+in[i+23:+26]*w[19];
5   out[i+8:i+11] = in[i+8:i+11]*w[0]+...+in[i+27:i+30]*w[19];
6   out[i+12:i+15] = in[i+12:i+15]*w[0]+...+in[i+31:i+34]*w[19];
7 }
```

Listing 5.9: Pseudocode snippet of `FIR_UV` class implementation with the same function arguments as Listing 5.8. Here, ':' denotes a range, while the remaining syntax is similar to that of C.

section. This approach leaves the optimization process to the compiler backend, which is optimal for our use-case.

This showed good results, but port pressure causes performance degradation when the filter size gets larger. This is shown in the performance plots in the next chapter. To mitigate this, a *multi-pass* approach similar to strip-mining, discussed in Section 4.2, was employed in `FIR_MP`. This approach

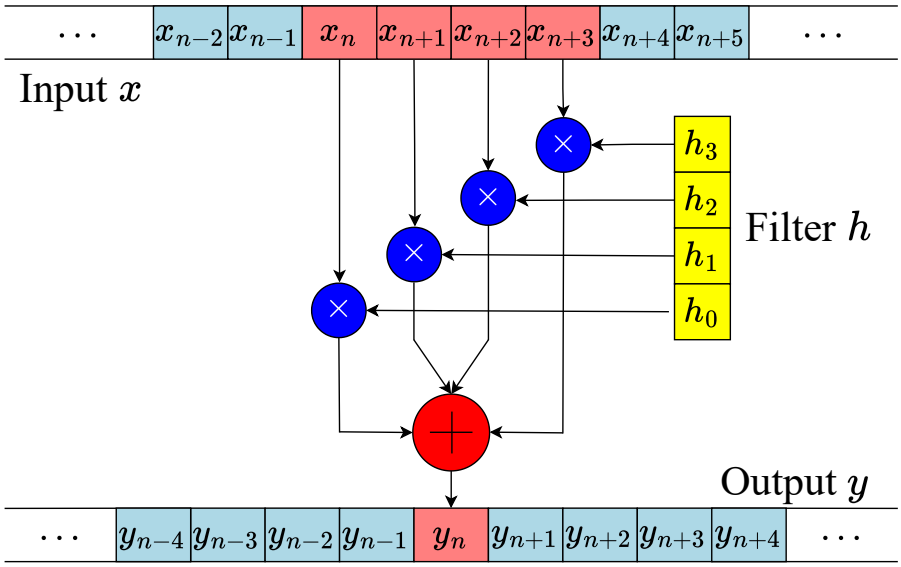


Figure 5.2: The baseline implementation, demonstrated for a filter size of 4, involves a reduction operation at the end of the filtering process. While this is the default compiler approach, it is costly in terms of performance. By avoiding this operation and employing an alternative scheme, as the one given in Listing 5.9, we achieve significant performance improvements. There, only vectorized *FMA*s are utilized.

uses multiple passes to do the filtering as opposed to the single pass method that is implemented in `FIR_UV`. An illustration is presented in Figure 5.3, where two passes are employed for a filter size of 20 and a pass size of $\texttt{UF} \times \texttt{VF} = 16$. It is important to note that the passes go through the entire input before going to the next pass and filtering with different weights. The last pass can only work with the 4 remaining filters: $w[16], \cdots, w[19]$.

This approach mitigates the performance degradation from increasing filter sizes. However, due to increased memory access in the multipass, performance can still worsen for larger filter sizes and higher cache levels. To address this issue, *loop blocking* was employed, as described in Chapter 4. The tile size is chosen to be close to or equal to the cache size of the target architecture and is a multiple of $\texttt{UF} \times \texttt{VF}$. An additional outer loop with a stride equal to the tile size ensures contiguous and efficient memory access within the cache. The first loop within handles new data from memory, while subsequent passes work with existing elements in the cache. The outer loop is implemented at the beginning of `FIR_MP`. The equivalent pseudocode implementation is shown in Listing 5.11.

In this filtering scheme, traditional, cumbersome, handwritten FIR filtering

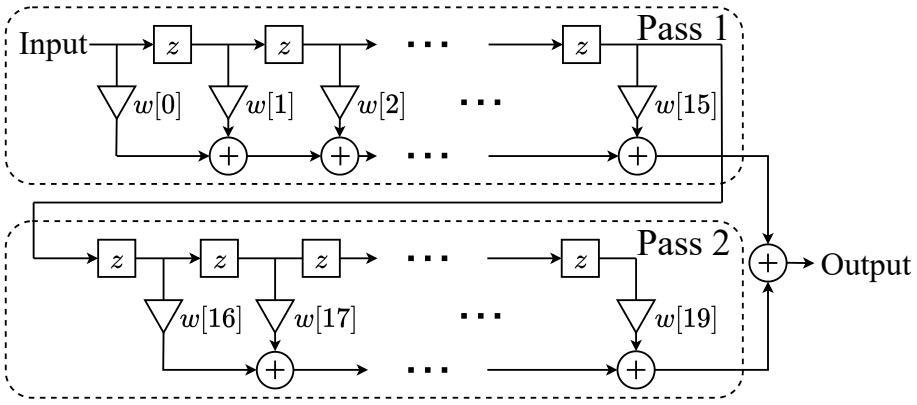


Figure 5.3: Multipass approach to implementing FIR filtering. Here, the entire input sample is processed in one pass before being filtered in subsequent passes. An unrolled and vectorized equivalent is provided in Listing 5.10.

```
for(int i = 0; i < outputsize; i+=16)
{
  out[i:i+3]=in[i:i+3]*w[0]+...+in[i+15:i+18]*w[15];
  out[i+4:i+7]=in[i+4:i+7]*w[0]+...+in[i+19:i+22]*w[15];
  out[i+8:i+11]=in[i+8:i+11]*w[0]+...+in[i+23:i+26]*w[15];
  out[i+12:i+15]=in[i+12:i+15]*w[0]+...+in[i+27:i+30]*w[15];
}
for(int i = 0; i < outputsize; i+=16)
{
  out[i:i+3]+=in[i+16:i+19]*w[16]+...+in[i+19:i+22]*w[19];
  out[i+4:i+7]+=in[i+20:i+23]*w[16]+...+in[i+23:i+26]*w[19];
  out[i+8:i+11]+=in[i+24:i+27]*w[16]+...+in[i+27:i+30]*w[19];
  out[i+12:i+15]+=in[i+28:i+31]*w[16]+...+in[i+31:i+34]*w[19];
}
```

Listing 5.10: Pseudocode equivalent of the multipass filtering implementation.

implementations are replaced by an object-oriented approach for generating LLVM IR. This new method was rigorously tested and benchmarked to evaluate its effectiveness in improving FIR filtering performance. Performance assessment involved a benchmark written to utilize MPI, as outlined in Section 4.5, to establish a barrier and measure time across different CPU processes, thereby avoiding unintended boosting effects. The number of floating-point operations per second is calculated using the measured time and the total number of FLOPs, which is $2 \times m \times (n - m)$, where n denotes the input size and m is the filter size. The measurements were repeated enough times to capture the effect of cache levels on memory accesses. To report and analyze performance, a roofline plot was employed. Computational intensity, crucial for this analysis, was calculated as detailed in Section 4.6, with the formula $\mathrm{OI} = \frac{2 \cdot m \cdot (n-m)}{4 \cdot m + 4 \cdot n + 4 \cdot (n-m)}$. The measured bandwidth was

```
for(int ii = 0; ii < outputsize; ii+=tilesize)
{
  for(int i = 0; i < min(ii, i+outputsize); i+=16)
  {
    out[i:i+3] = in[i:i+3]*w[0]+...+in[i+15:i+18]*w[15];
    out[i+4:i+7] = in[i+4:i+7]*w[0]+...+in[i+19:i+22]*w[15];
    out[i+8:i+11] = in[i+8:i+11]*w[0]+...+in[i+23:i+26]*w[15];
    out[i+12:i+15] = in[i+12:i+15]*w[0]+...+in[i+27:i+30]*w[15];
  }
  for(int i = 0; i < min(ii, i+outputsize); i+=16)
  {
    out[i:i+3]+=in[i+16:i+19]*w[16]+...+in[i+19:i+22]*w[19];
    out[i+4:i+7]+=in[i+20:i+23]*w[16]+...+in[i+23:i+26]*w[19];
    out[i+8:i+11]+=in[i+24:i+27]*w[16]+...+in[i+27:i+30]*w[19];
    out[i+12:i+15]+=in[i+28:i+31]*w[16]+...+in[i+31:i+34]*w[19];
  }
}
```

Listing 5.11: Pseudocode representation of the `FIR_MP` class implementation, which does tiled multiple passes.

obtained using the STREAM benchmark [11], which serves as an alternative to the peak theoretical bandwidth in the roofline plot.

Performance analysis was conducted on an Apple Silicon M2 Pro with 8 performance cores, an Intel Xeon E5-2690 CPU with 12 cores, an NVIDIA Grace CPU superchip with 72 cores, and an Intel Xeon Platinum 8276 CPU with 56 cores.

The theoretical peak bandwidth of the M2 Pro is 200GB/s and the clock frequency is 3.5 GHz. It is important to note that the internal architecture of this system is proprietary and not open source, with details such as the number of FMA ports remaining not publicly disclosed. Consequently, accurately calculating the peak performance is challenging. To address this, our multipass implementation was used to estimate the value. By running benchmarks with varying unroll factors (i.e., different numbers of FMA ports), we were able to make this estimation. This is further discussed in the next chapter. Additionally, the vector factor (i.e., the SIMD register width) was determined by examining the assembly machine code generated from the optimized compilation of the simple kernels at the start of the project. As described, the width is dictated by the underlying architecture. From the information we gathered, the peak performance could then be calculated using (4.2) from Section 4.6:

$$\begin{aligned} P_{\text{peak}} &= n_{\text{core}} \times n_{\text{super}}^{\text{FP}} \times n_{\text{FMA}} \times n_{\text{SIMD}} \times f \\ &= 8 \times 2(\text{Educated guess}) \times 4 \times 4 \times 3.49\text{GHz} \\ &= 893.44\text{GFLOP/s}. \end{aligned} \tag{5.2}$$

Table 5.1: Apple Silicon Spectifications.

CPU Model	Apple M2 Pro
Architecture	`arm64`
Populated Socket	1
Physical Core Count	12 (8 P-Cores & 4 E-Cores)
Hyper-Threading	-
RAM capacity	16GB
Nominal-FP32 Peak Performance	893.44 GFLOP/s
Peak Bandwidth	200 GB/s
STREAM[11] Bandwidth/Copy	178.576 GB/s

Table 5.2: Intel Xeon E5-2690 CPU Compute Node Spectifications.

CPU Model	Intel Xeon E5-2690 v3, 2.60GHz
μarch	*Haswell*
Populated Socket	1
Physical Core Count	12
Hyper-Threading	Disabled
RAM capacity	64GB
Nominal-FP32 Peak Performance	998.4 GFLOP/s
Peak Bandwidth	68 GB/s
STREAM[11] Bandwidth/Triad	44.4 GB/s

Here, the n-way replication of the execution port was inferred based on typical values observed in other architectures. With this, along with other available and benchmarked information, we now have a complete specification of the M2 Pro. It is presented in Table 5.1.

Unlike Apple Silicon, the specifications of the Intel Xeon E5-2690 CPU are publicly available and are given in Table 5.2. The peak double-precision floating-point performance of this CPU is listed as 499.2 GFLOP/s in [4]. Running at a base clock frequency of 2.3 GHz, it supports AVX2 instructions that leverage SIMD registers capable of executing 8 single-precision floating-point operations. The CPU also features 2 FMA ports. The peak memory bandwidth is specified at 68 GB/s, with the attainable bandwidth measured through the STREAM benchmark.

The specifications for the ARM-based NVIDIA Grace CPU superchip is given in Table 5.3. The peak double-precision floating-point performance of a 144-core NVIDIA Grace superchip is listed as 7.1 TFLOP/s in [19]. It supports Full SVE-2 (Scalable Vector Extensions) instruction set, utilizing SIMD registers capable of executing between 4 and 64 single-precision floating-point operations simultaneously. At the time of writing, a specialized and dedicated FMA LLVM intrinsic that supported SVE was not available. For this reason, fixed-length vector operations were used instead.

Table 5.3: NVIDIA Grace CPU Superchip Node Spectifications.

CPU Model	Neoverse V2
μarch	*arm64*
ISA	*ARM V9.0*
Populated Socket	4
Physical Core Count	72
Hyper-Threading	Disabled
RAM capacity	800GB
Nominal-FP32 Peak Performance	28.4 TFLOP/s
Peak Bandwidth	2 TB/s
STREAM[11] Bandwidth/Triad	1.54 TB/s

Table 5.4: Intel Xeon Platinum CPU Node Spectifications.

CPU Model	Intel Xeon Platinum 8276, 2.20 GHz
μarch	*Cascade Lake*
Populated Socket	2
Physical Core Count	28
Hyper-Threading	Disabled
RAM capacity	1 TB
Nominal-FP32 Peak Performance	7.9 TFLOP/s

Lastly, the details of the Intel Xeon Platinum CPU is given in Table 5.4. This architecture supports the AVX-512 extension, which employs 512-bit wide SIMD registers. This capability allows the architecture to execute 16 single-precision floating-point operations simultaneously.

Performance was evaluated on these systems for multiple implementations of the FIR filter. The following list summarizes the different FIR filtering implementations and assigns labels to each. These labels will be used for reference in the next chapter during performance comparison and analysis:

- *Baseline*: This version, as illustrated in Listing 5.8, uses a standard C cross-correlation. The compiler applies a reduction operation during optimization, which results in suboptimal performance due to the straightforward nature of VPlan, mentioned briefly in Section 2.2.
- *Handwritten*: This implementation is a manually written LLVM IR code. Though functional, it is not practical for large-scale optimization due to the rigidity and difficulty in implementing enhancements.
- *LLVM Baseline*: Our LLVM-based synthesized approach that implements unrolling and vectorization to leverage ILP and DLP. An equivalent pseudocode is given in Listing 5.9. While this version improves performance in many cases, it can suffer when the filter size increases.

- *Multipass Tiled*: Building on the LLVM Baseline, this implementation employs a multipass approach combined with tiling. This helps mitigate performance degradation for larger filter sizes by optimizing memory access and reducing register pressure. Again, the equivalent representation is given in Listing 5.11.

These implementations are compared against each other and visualized using a roofline plot. The impact of varying filter sizes is also assessed. Detailed results of these analyses are presented in Chapter 6.

Chapter 6

Results

This chapter presents the results of our FIR filtering implementation and focuses on performance metrics and comparisons. While the results shown in this chapter are limited to 1-D filtering, as mentioned in Section 1.1, it can be extended to multidimensional cases simply by employing 1-D factorizations to the translations. We start by showcasing the effectiveness of the FIR filter. We will delve into performance results to provide a comprehensive evaluation of our approaches. Finally, we will discuss the advantages of our LLVM-based approach over traditional handwritten kernels and generic implementations, emphasizing the improvements in overall performance. Roofline plots are used to compare the various performance results.

To illustrate the filter's frequency response, a filter size of 40 is employed with the scaling parameter $L = 30$ substituted into (5.1). Figure 6.1 demonstrates the filter's characteristic in attenuating frequencies beyond the passband. The ringing oscillations observed are a result of trade-offs in the filter design, particularly in the transition from the passband to the stopband. In the figure, a filter size of 40 is chosen for illustration, but the project investigates a range of filter sizes. The input signal for correctness evaluation consisted of a cosine wave, with noise added from a normal distribution, as described in the previous chapter. Figure 6.2 shows the resulting output before and after applying the filter to this noisy signal from two different implementations.

The performance results from different platforms are given in their respective sections:

Apple Silicon M2 Pro

Due to the proprietary nature of Apple Silicon's architectural details, we relied on estimation techniques to infer specifications. To determine the number of FMA ports, we employed a series of performance measurements

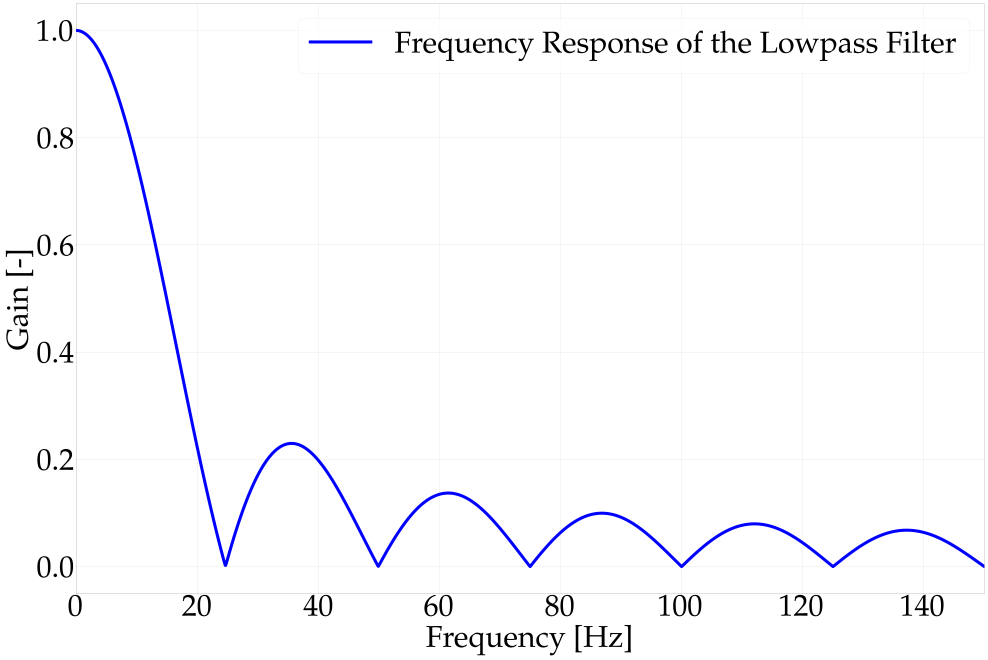


Figure 6.1: Frequency response of the lowpass filter used in our FIR scheme. The sampling frequency is chosen here to be 500 Hz. Decaying ringing oscillations is a result of the filter's attempt to attenuate frequencies while maintaining some level of smoothness.

using our LLVM Baseline approach, as detailed in the previous chapter. The results for a filter size of 80 are shown in Figure 6.3. The implementation used for this analysis, similar to Listing 5.9, offers insights into the unroll factor, which helps deduce the potential number of FMA ports. The analysis confirms that the highest performance is achieved with an unroll factor of 4, suggesting that the architecture includes 4 FMA ports. This finding justifies using 4 for n_{FMA} in (5.2).

We evaluated the four implementation strategies listed at the end of the previous chapter, across different filter sizes. Figures 6.4 and 6.5 display the performance results from these evaluations on the M2 Pro. The results reveal that our LLVM-based high-performance implementations, namely Multipass Tiled and LLVM Baseline, outperform the C Baseline and LLVM-based Handwritten implementations. However, as the filter size increases, as shown in Figure 6.5, the performance of the LLVM Baseline implementation deteriorates. In contrast, the multipass approach with tiling maintains strong performance. Tiling optimizes the multipass implementation by keeping data in the cache, which is particularly beneficial for larger input sizes.

This observation is further supported by the average performance results and the average fraction of peak performance, presented in Tables 6.1 and 6.2. The results for a small filter size is highlighted in light red in the tables

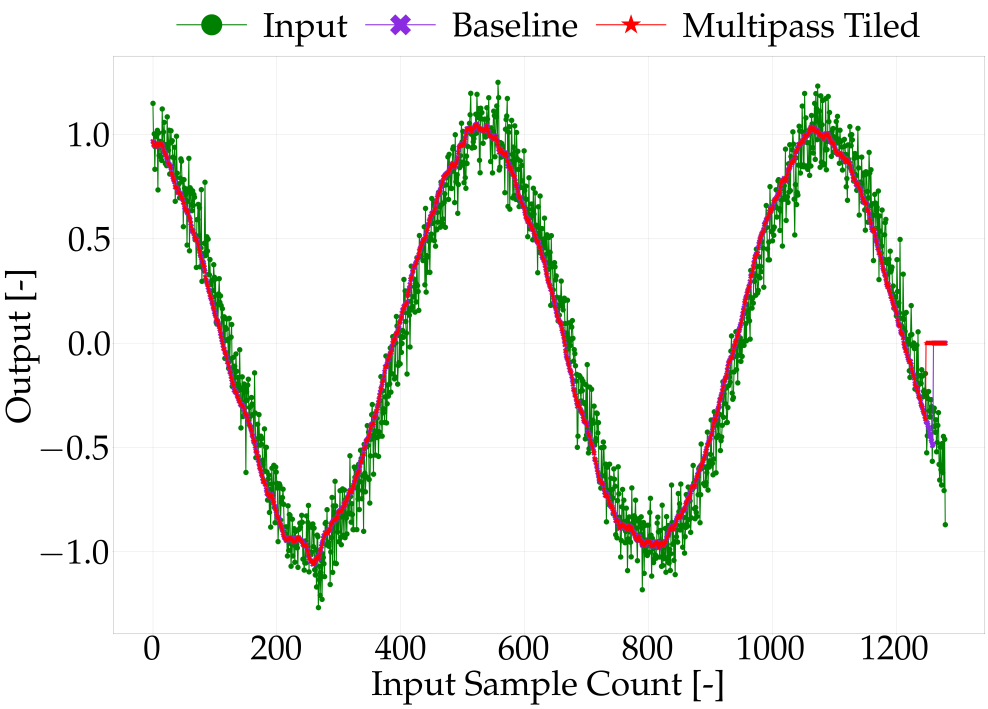


Figure 6.2: The result from our synthesized multipass implementation shows strong visual alignment with the baseline C implementation. Both methods use the same algorithm and produce filtered outputs with reduced noise. While the synthesized version terminates sooner, the early conclusion has negligible impact on the overall performance comparisons.

(corresponding to Figure 6.4), while a larger filter size is highlighted in light blue (corresponding to Figure 6.5). For small filter sizes that are not memory-bound (e.g., 16), the average performance of the LLVM Baseline and Multipass Tiled approaches is similar. However, as filter size increases, the tiled multipass approach maintains performance while the LLVM Baseline degrades. The C Baseline shows comparatively worse performance for smaller sizes and gets a modest boost with larger sizes, whereas Handwritten implementations generally perform worse as filter sizes increase.

The roofline plot for an input size of 9×10^6 is shown in Figure 6.6. It illustrates the transition from memory-bound to compute-bound regions as filter size increases, highlighting the efficient use of ILP and DLP by our synthesized implementations—LLVM Baseline and Multipass Tiled. The tiled multipass implementation, in particular, demonstrates consistent performance compared to the LLVM Baseiline as described earlier.

The performance gain of the best synthesized method (either Multipass Tiled or LLVM Baseline) was compared against both the Handwritten implementation and the Baseline method. The results are summarized in Table 6.3. As shown in the figures and tables before, the Multipass Tiled approach demon-

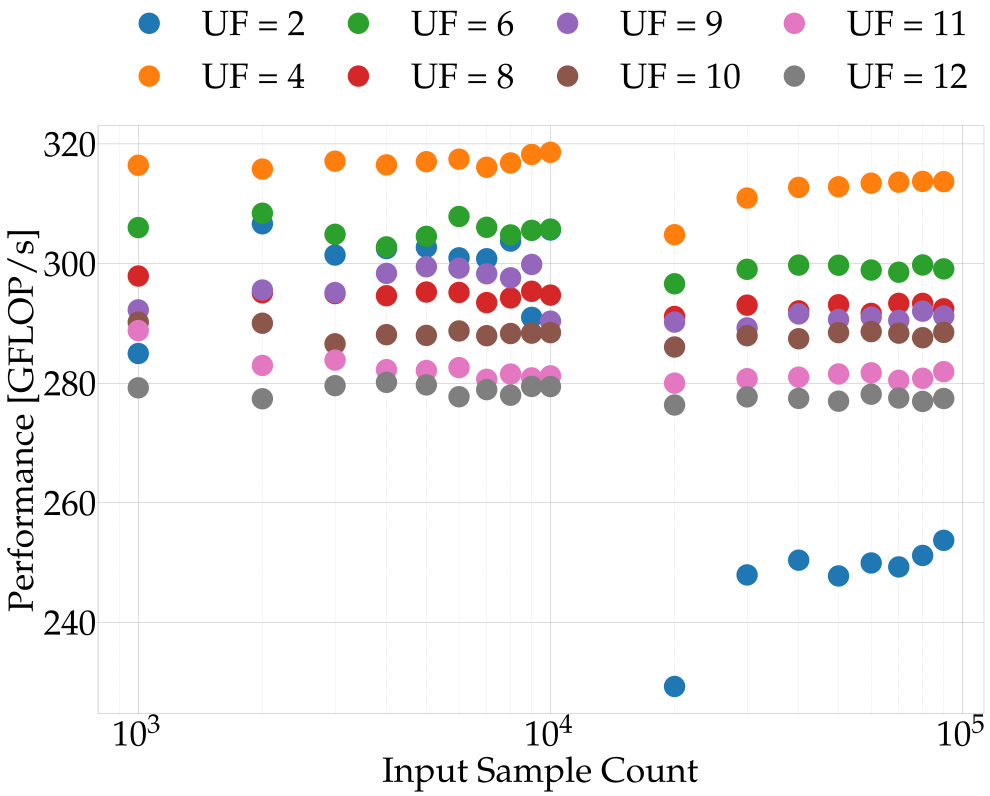


Figure 6.3: Performance results of a tiled multipass FIR filtering approach with a filter size of 80. The figure illustrates how different unroll factors impact performance. The data indicates that the Apple Silicon M2 Pro features 4 FMA ports, as evidenced by the peak performance achieved with an unroll factor of 4. The case when `UF` is 2 is particularly interesting, as the cache effect seems to be more pronounced. Because it under utilizes the available FMA ports, the CPU may not be able to take full advantage of spatio-temporal locality and might have to wait for data to be loaded from memory, which can result in cache misses.

Table 6.1: Average performance in GFLOP/s of the different implementations on the M2 Pro, calculated from all input sizes and categorized by filter size. This table compares the average performance of the different approaches. The results highlight how performance varies across different implementations and filter sizes.

Filter Size	Baseline	Handwritten	LLVM Baseline	Tiled Multipass
4	28.80	24.90	*319.25*	*312.69*
16	*164.06*	*65.16*	*525.03*	*521.70*
28	70.17	45.08	*414.15*	*513.08*
40	118.49	46.41	*365.65*	*475.97*
52	170.90	42.03	*319.00*	*461.48*
64	226.00	42.03	*294.92*	*476.06*
76	130.23	33.50	*285.03*	*468.68*
80	*223.40*	*35.24*	*289.90*	*462.71*

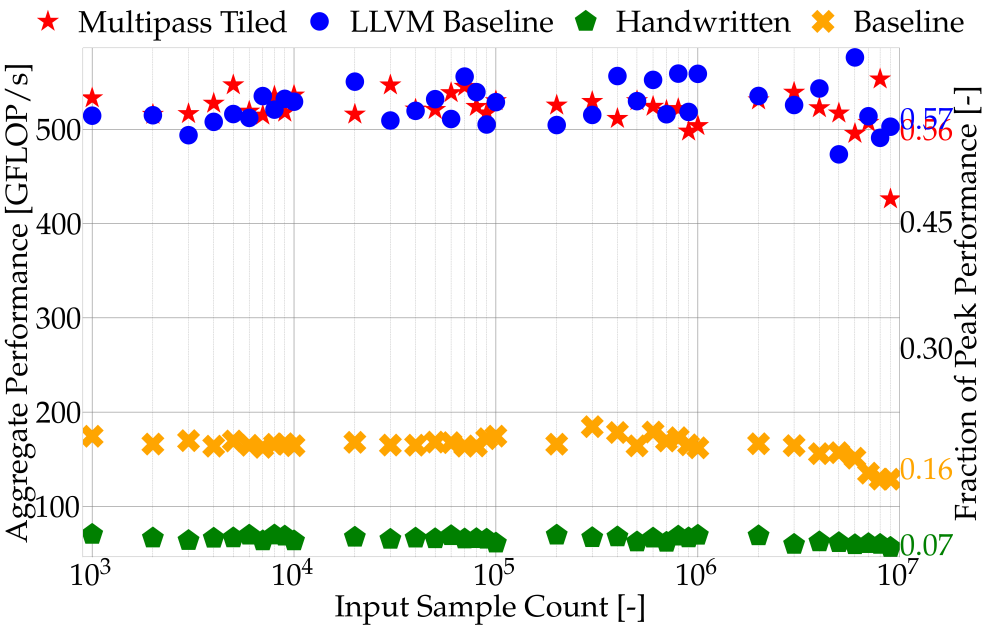


Figure 6.4: Performance results from 8 MPI processes on an 8 P-core M2 Pro machine, with a filter size of 16, an unroll factor of 4, and a vectorization factor of 4. The figure compares the performance of different implementations on an M2 Pro.

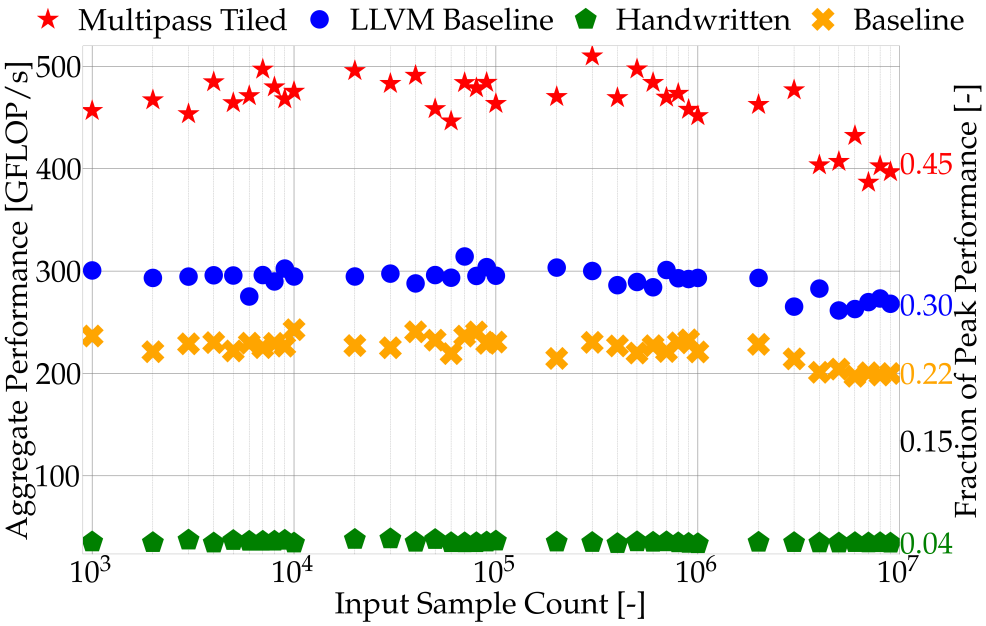


Figure 6.5: Performance results from 8 MPI processes on an 8 P-core M2 Pro machine, with a filter size of 80, an unroll factor of 4, and a vectorization factor of 4. Compared to Figure 6.4, this figure illustrates how performance evolves with increasing filter size, emphasizing that the LLVM Baseline suffers a decline in performance while the Multipass Tiled approach maintains consistent performance. Additionally, the C Baseline implementation sees a modest performance improvement, whereas the Handwritten implementation suffers a performance degradation.

Table 6.2: Average fraction of peak performance in GFLOP/s of the different implementations on the M2 Pro (P_{peak} = 893.44 GFLOP/s), calculated from all input sizes and categorized by filter size.

Filter Size	Baseline	Handwritten	LLVM Baseline	Tiled Multipass
4	0.03	0.03	*0.36*	*0.35*
16	*0.18*	*0.07*	*0.59*	*0.58*
28	0.08	0.05	*0.46*	*0.57*
40	0.13	0.05	*0.41*	*0.53*
52	0.19	0.05	*0.36*	*0.52*
64	0.25	0.05	*0.33*	*0.53*
76	0.15	0.04	*0.32*	*0.52*
80	*0.25*	*0.04*	*0.32*	*0.52*

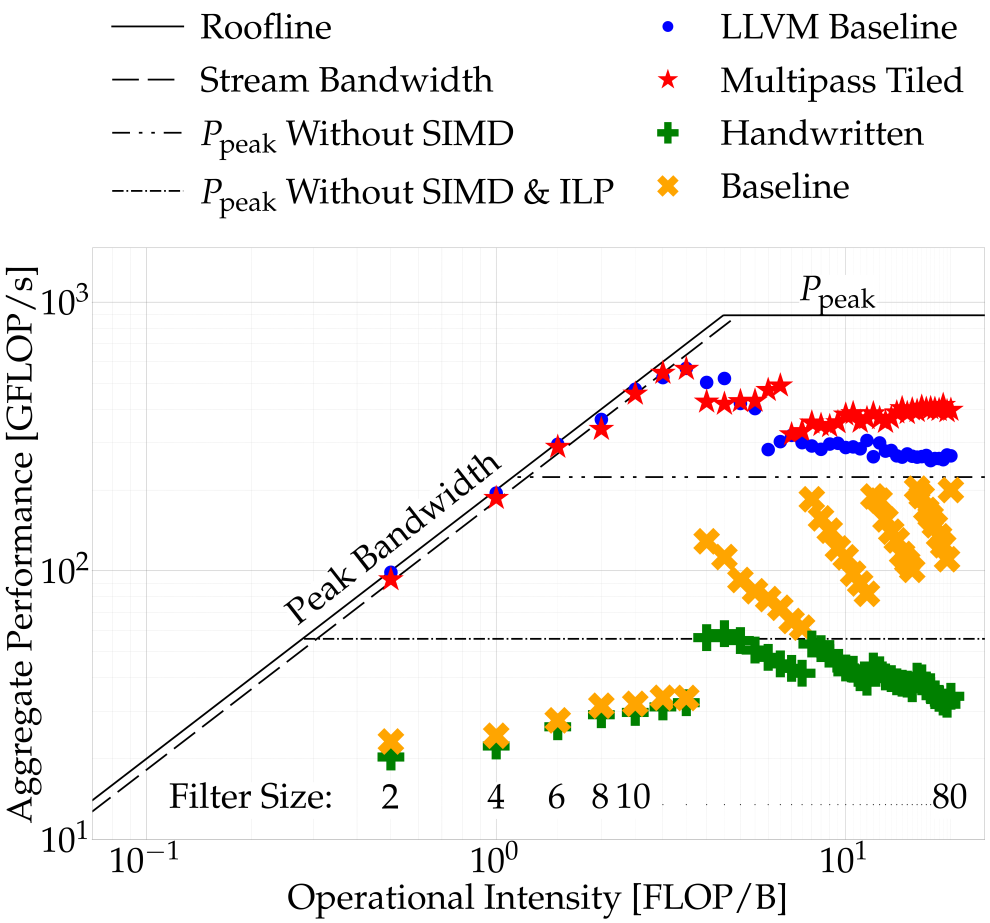


Figure 6.6: Roofline plot for different implementations on the Apple Silicon CPU for an input sample count of 9×10^6, showing the impact of filter size on memory bandwidth and performance. The plot reveals the efficient utilization of ILP and DLP by our implementations, with the tiled multipass approach maintaining strong performance across various filter sizes.

Table 6.3: Average performance across all filter and input sizes for each method on the Apple Silicon.

Method	Average Performance (overall) [GFLOP/s]
Multipass Tiled	471.39
LLVM Baseline	370.64
Baseline	122.92
Handwritten	42.20

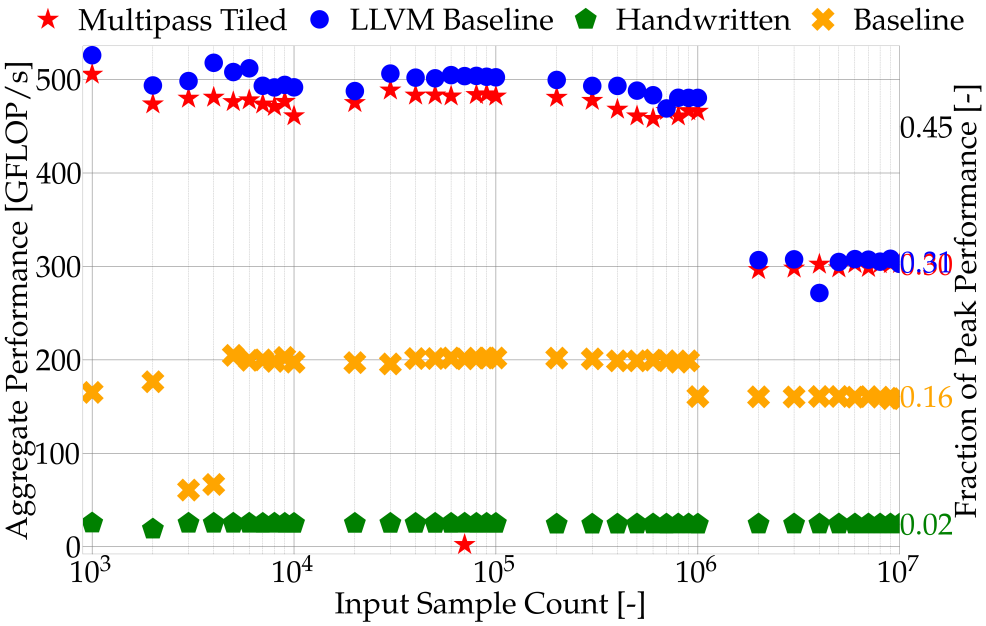


Figure 6.7: Performance results from 12 MPI processes on an Intel Xeon E5-2650 machine, with a filter size of 56, an unroll factor of 8, and a vectorization factor of 8.

strated the highest average performance, followed by the LLVM Baseline. The Handwritten implementation had a significantly lower average performance. Specifically, Multipass Tiled showed an impressive performance gain of over 1000% compared to the Handwritten kernel and a performance gain of 283.50% compared to the C Baseline. This demonstrates a substantial improvement in performance when using the synthesized approach over both the handwritten and baseline implementations.

Intel Xeon E5-2690 v3

Performance results from the Intel CPU measurements are detailed in Figures 6.7 and 6.8. We again evaluated the different approaches across different filter sizes.

The results show that our LLVM-based implementations, Multipass Tiled and

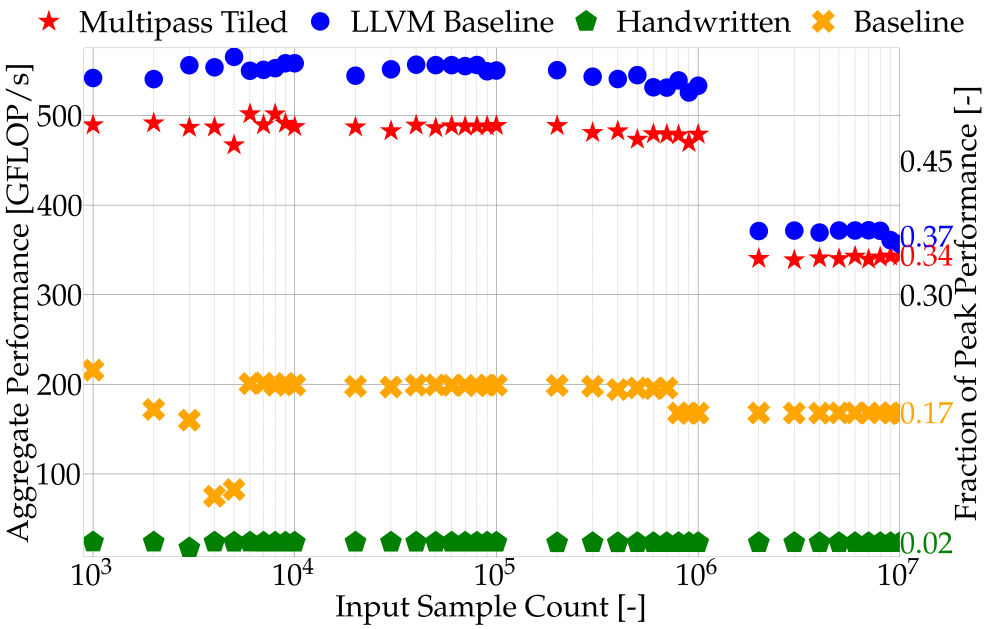


Figure 6.8: Performance results from 12 MPI processes on an Intel Xeon E5-2650 machine, with a filter size of 80, an unroll factor of 8, and a vectorization factor of 8.

LLVM Baseline, again outperform the C Baseline and LLVM-based Handwritten implementations. Notably, as filter size increases, the performance of the LLVM Baseline implementation continues to improve, contrasting with the performance trends observed on the M2 Pro. However, the multipass approach with tiling shows a smaller improvement in performance. Employing a multipass approach is beneficial only when the LLVM Baseline performs poorly, making it less advantageous in this context.

This observation is supported by the average performance results and the average fraction of peak performance, as presented in Tables 6.4 and 6.5. Again, results for a small filter size is highlighted in light red (corresponding to Figure 6.7), while larger filter sizes are highlighted in darker red (corresponding to Figure 6.8). For smaller filter sizes that are not memory-bound (e.g., 56), the average performance of the LLVM Baseline and Multipass Tiled approaches is similar. However, as the filter size increases, the tiled multipass approach shows a moderate increase in performance, while the LLVM Baseline exhibits a better performance improvement. The C Baseline shows relatively worse performance for smaller sizes but a modest boost for larger sizes, whereas the Handwritten implementation generally degrades in performance with increasing filter sizes.

The roofline plot for an input sample count of 90×10^6 is shown in Figure 6.9. It again illustrates the transition from memory-bound to compute-bound regions as filter size increases and emphasizes the effective use of ILP

Table 6.4: Average performance in GFLOP/s of the different implementations on the Intel Xeon, calculated from all input sizes and categorized by filter size.

Filter Size	Baseline	Handwritten	LLVM Baseline	Tiled Multipass
8	68.92	40.81	*261.53*	*256.80*
44	103.56	24.41	*374.26*	*377.18*
56	*176.88*	*24.44*	*423.76*	*399.42*
80	*178.48*	*23.31*	*479.76*	*431.29*
116	132.54	19.90	*536.52*	*433.49*
152	164.17	18.35	*538.21*	*450.77*
188	123.00	17.60	*562.74*	*453.62*
200	*157.59*	*17.36*	*586.70*	*449.02*

Table 6.5: Average fraction of peak performance in GFLOP/s of the different implementations on the Intel Xeon (P_{peak} = 998.4 GFLOP/s), calculated from all input sizes and categorized by filter size.

Filter Size	Baseline	Handwritten	LLVM Baseline	Tiled Multipass
8	0.07	0.04	*0.26*	*0.26*
44	0.10	0.02	*0.37*	*0.38*
56	*0.18*	*0.02*	*0.42*	*0.40*
80	*0.18*	*0.02*	*0.48*	*0.43*
116	0.13	0.02	*0.54*	*0.43*
152	0.16	0.02	*0.54*	*0.45*
188	0.12	0.02	*0.56*	*0.45*
200	*0.16*	*0.02*	*0.59*	*0.45*

Table 6.6: Average performance across all filter and input sizes for each method on the Intel compute node.

Method	Average Performance (overall) [GFLOP/s]
LLVM Baseline	465.88
Multipass Tiled	405.31
Baseline	125.33
Handwritten	22.42

and DLP by our synthesized implementations. The performance of these implementations continues to improve, suggesting that very large filter sizes are required to observe significant performance degradation from the LLVM Baseline.

The performance gain of the best synthesized method (either Multipass Tiled or LLVM Baseline) was compared against both the Handwritten implementation and the Baseline method, similarly to the Apple Silicon. The results are summarized in Table 6.6. As we've seen in the performance figures and tables, the LLVM Baseline implementation demonstrated the highest

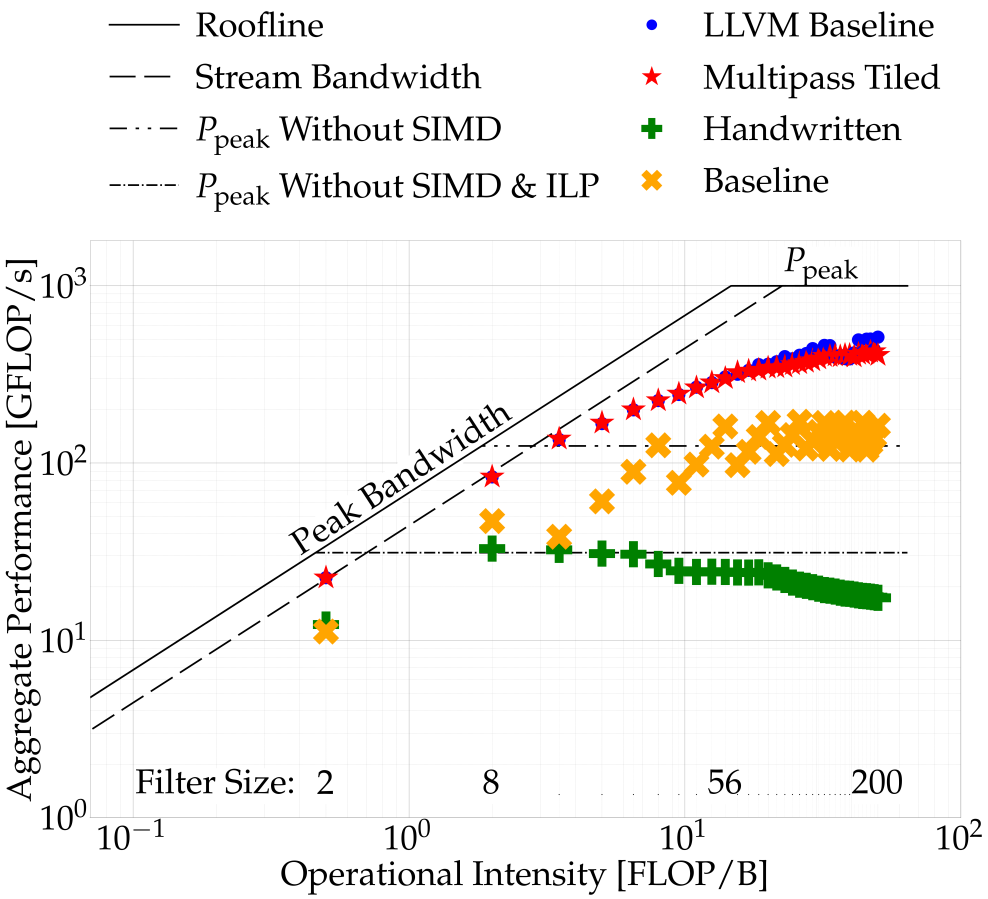


Figure 6.9: Roofline plot for different implementations on the Intel Haswell CPU for an input sample count of 90×10^6, depicting the impact of filter size on memory bandwidth and performance. The plot highlights the effective use of ILP and DLP by the synthesized implementations. The performance of these implementations continues to improve, which indicates that substantial degradation may only occur with very large filter sizes.

average performance, followed by the Multipass Tiled. The Handwritten kernel exhibited significantly lower performance compared to the synthesized approaches. Specifically, LLVM Baseline showed an outstanding performance gain of 1978.40% compared to the Handwritten implementation and a gain of 271.73% compared to the Baseline C implementation. This demonstrates a substantial improvement in performance when using the synthesized methods over both the handwritten and baseline implementations.

NVIDIA Grace CPU Superchip

Performance results from the NVIDIA Grace CPU measurements are detailed in Figures 6.10 and 6.11. We again evaluated the different approaches across

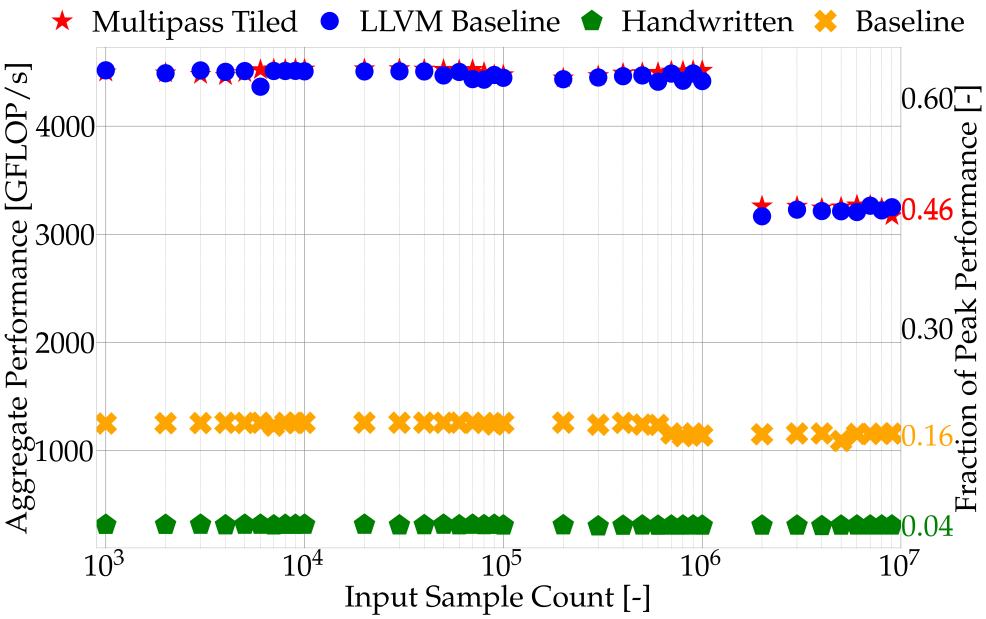


Figure 6.10: Performance results from 72 MPI processes on an NVIDIA Grace CPU superchip, with a filter size of 16, an unroll factor of 4, and a vectorization factor of 4.

different filter sizes.

The results show that our LLVM-based implementations, Multipass Tiled and LLVM Baseline, again outperform the C Baseline and LLVM-based Handwritten implementations. Notably, as filter size increases, the performance of the Multipass Tiled implementation continues to outperform the LLVM Baseline, similar to the performance trends observed on the M2 Pro. However, the gain is comparatively incremental.

This observation is supported by the average performance results and the average fraction of peak performance, as presented in Tables 6.7 and 6.8. Again, results for a small filter size is highlighted in light red (corresponding to Figure 6.10), while larger filter sizes are highlighted in light blue (corresponding to Figure 6.11). For smaller filter sizes that are not memory-bound (e.g., 16), the average performance of the LLVM Baseline and Multipass Tiled approaches is similar. However, as the filter size increases, the Multipass Tiled approach shows a higher gain in performance over the LLVM Baseline. The C Baseline shows relatively worse performance for smaller sizes but an irregular boost for larger sizes, whereas the Handwritten implementation generally degrades in performance with increasing filter sizes.

The roofline plot for an input sample count of 9×10^6 is shown in Figure 6.12. It again illustrates the transition from memory-bound to compute-bound regions as filter size increases and emphasizes the effective use of ILP and DLP by our synthesized implementations. The Multipass Tiled

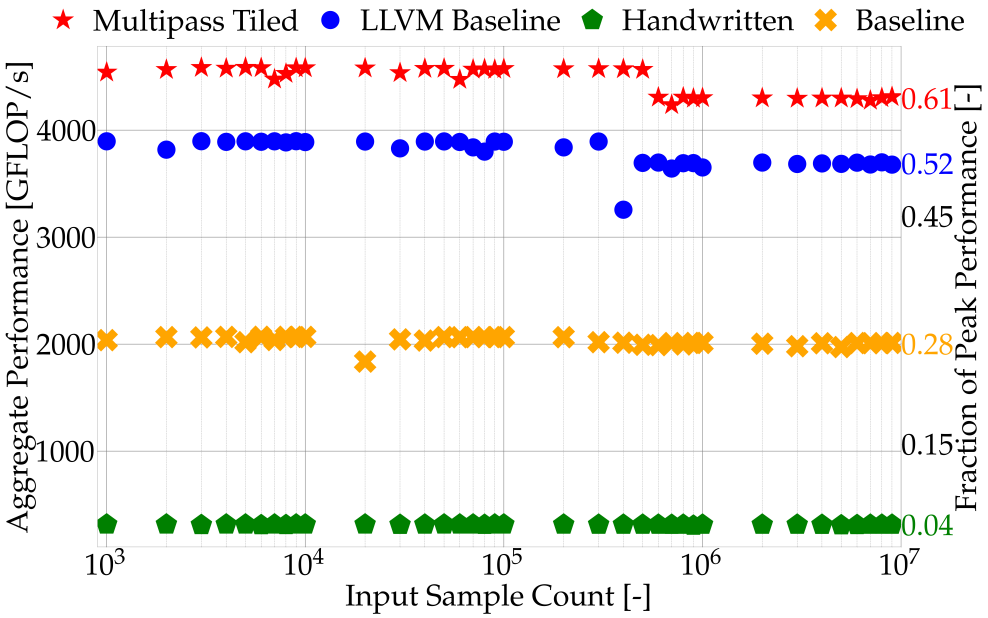


Figure 6.11: Performance results from 72 MPI processes on an NVIDIA Grace CPU superchip, with a filter size of 80, an unroll factor of 4, and a vectorization factor of 4.

Table 6.7: Average performance in GFLOP/s of the different implementations on the NVIDIA Grace CPU Superchip, calculated from all input sizes and categorized by filter size. The results highlight how performance varies across different implementations and filter sizes.

Filter Size	Baseline	Handwritten	LLVM Baseline	Tiled Multipass
4	297.45	299.20	*3113.75*	*3096.22*
16	*1218.32*	*311.43*	*4194.24*	*4223.43*
28	756.93	522.05	*4002.41*	*4409.62*
40	1146.58	390.94	*3887.05*	*4367.84*
52	1511.10	338.50	*3841.05*	*3724.51*
64	1980.89	313.46	*3796.36*	*4087.99*
76	1321.60	372.69	*3811.46*	*4446.53*
80	*2027.64*	*313.25*	*3784.94*	*4465.39*

Table 6.8: Average fraction of peak performance in GFLOP/s of the different implementations on the NVIDIA Grace CPU Superchip (P_{peak} = 7.1 TFLOP/s), calculated from all input sizes and categorized by filter size. It shows the results from Table 6.7 compared to P_{peak}. The data again underscores the performance consistency of the Multipass Tiled approach and the slight degradation observed in the LLVM Baseline implementation as filter size increases.

Filter Size	Baseline	Handwritten	LLVM Baseline	Tiled Multipass
4	0.04	0.04	*0.44*	*0.44*
16	*0.17*	*0.04*	*0.59*	*0.59*
28	0.11	0.07	*0.56*	*0.62*
40	0.16	0.06	*0.55*	*0.62*
52	0.21	0.05	*0.54*	*0.52*
64	0.28	0.04	*0.53*	*0.58*
76	0.19	0.05	*0.54*	*0.63*
80	*0.29*	*0.04*	*0.53*	*0.63*

Table 6.9: Average performance across all filter and input sizes for each method on the NVIDIA Grace CPU.

Method	Average Performance (overall) [GFLOP/s]
LLVM Baseline	3885.48
Multipass Tiled	4081.32
Baseline	1133.95
Handwritten	371.27

implementation, in particular, demonstrates the highest performance.

The performance gain of the best synthesized method (either Multipass Tiled or LLVM Baseline) was again compared against both the Handwritten implementation and the Baseline. The results are summarized in Table 6.9. The Multipass Tiled implementation showed an outstanding performance gain of 999.27% compared to the Handwritten implementation and a gain of 259.92% compared to the Baseline C implementation. This demonstrates a substantial improvement in performance when using the synthesized method over both the handwritten and baseline implementations.

Intel Xeon Platinum 8276 CPU

Lastly, performance results from the Intel Xeon Platinum 8276 CPU measurements are detailed in Figures 6.13 to 6.15. The different approaches were evaluated across different input and filter sizes.

The results show that our LLVM-based implementation, Multipass Tiled, outperform the C Baseline by a big margin. Specifically, they show an outstanding performance gain ranging from 1.45 to 11.93 times compared to the Baseline C implementation. This demonstrates a substantial improvement in

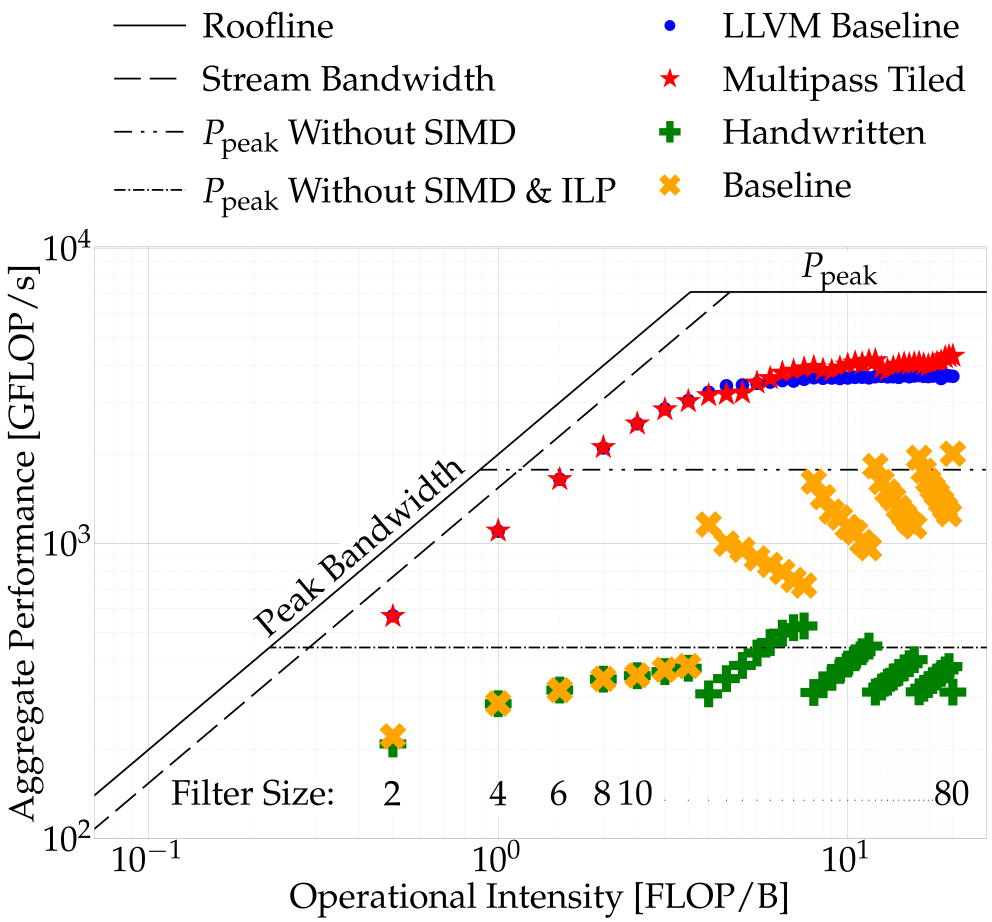


Figure 6.12: Roofline plot for different implementations on the NVIDIA Grace CPU for an input sample count of 9×10^6, depicting the impact of filter size on memory bandwidth and performance. The plot highlights the effective use of ILP and DLP by the synthesized implementations.

performance when using the synthesized method over the "vanilla" baseline implementation.

Summary

Noteworthy observations include the patterns observed in the roofline plots of the C Baseline implementation that exhibit a repetitive nature, possibly due to the vectorization and unroll factors utilized by the compilers. It seems that for certain filter sizes, these factors effectively exploit the available ports and SIMD register slots, while in other cases, they do not. Another observation is that in some architectures, significant increases in input sizes lead to a drop in performance. While one might be tempted to attribute this to varying levels of cache, this behavior is not easily discernible and warrants

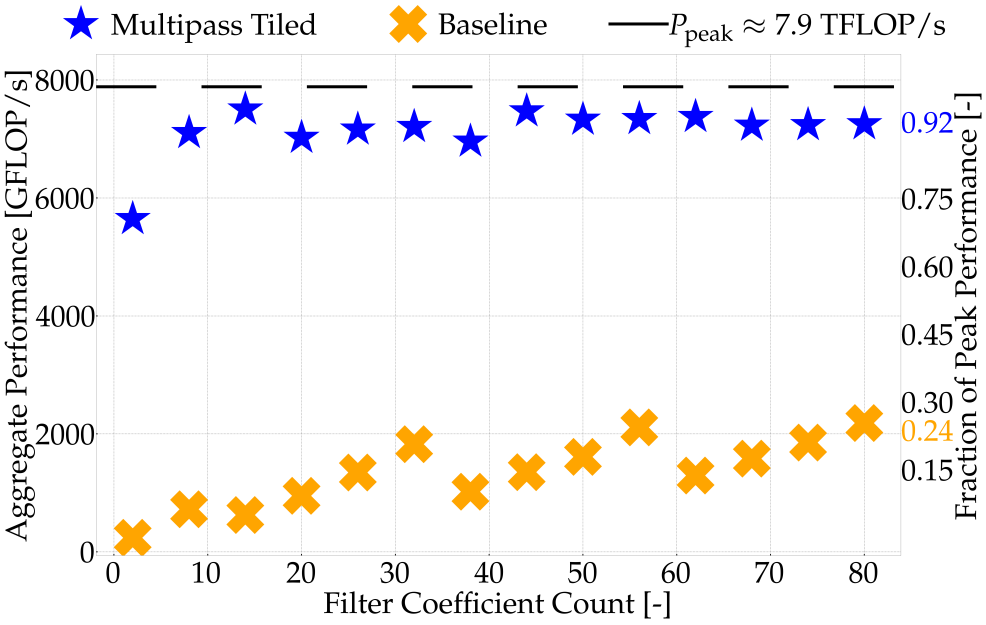


Figure 6.13: Performance results from 56 MPI processes on an Intel Xeon Platinum 8276 machine, with an input size of 1000, an unroll factor of 4, and a vectorization factor of 16. This architecture, as stated in the previous chapter, supports AVX-512 extension.

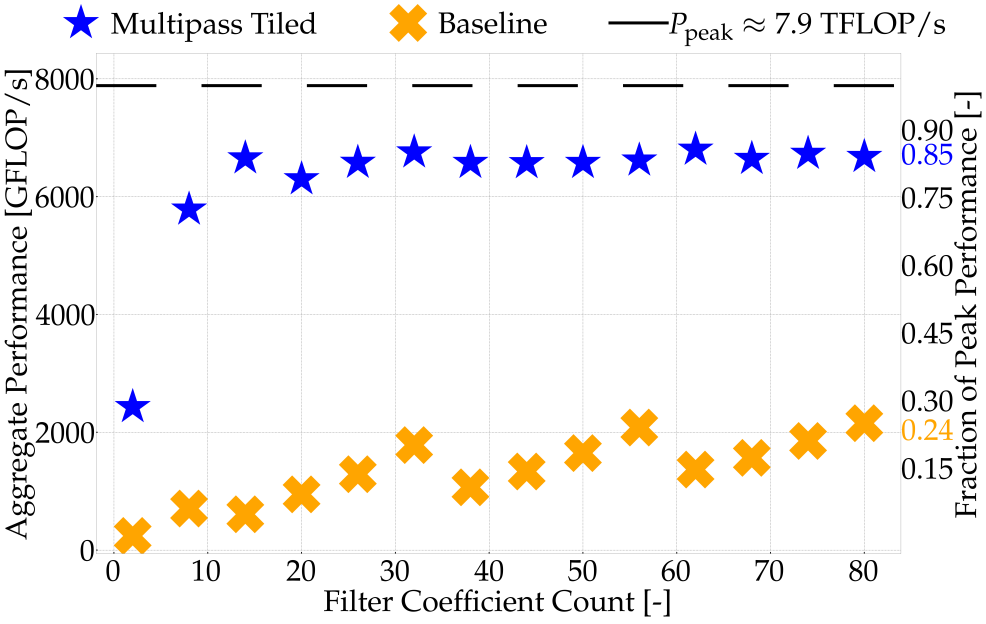


Figure 6.14: Performance results from 56 MPI processes on an Intel Xeon Platinum 8276 machine, with an input size of 100000, `UF` and `VF` unchanged from Figure 6.13.

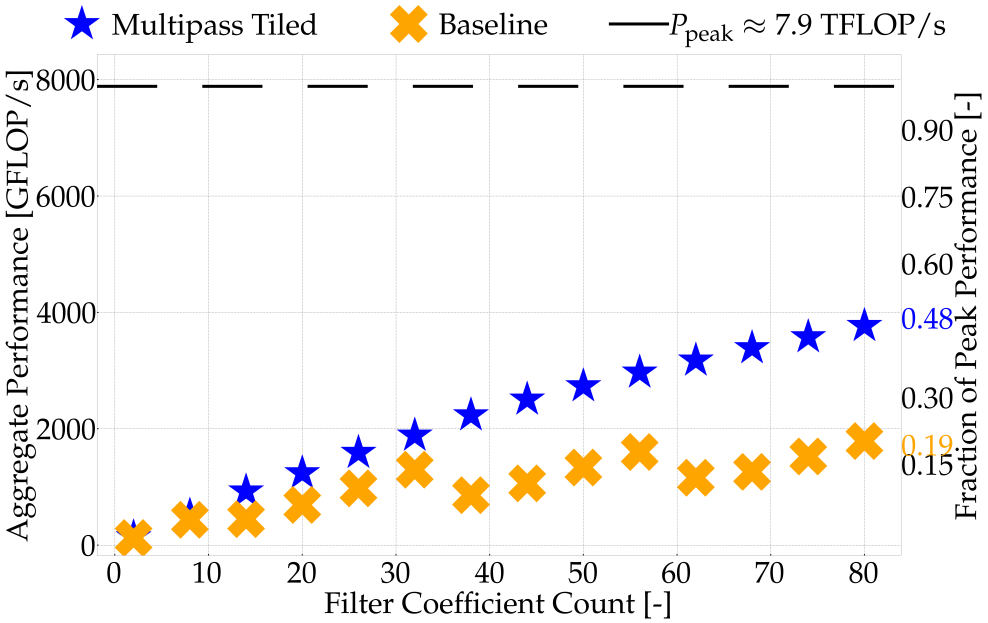


Figure 6.15: Performance results from 56 MPI processes on an Intel Xeon Platinum 8276 machine, with an input size of 1000000. Here, the unroll and vectorization factors are also 4 and 16, respectively.

further investigation, especially in relation to the roofline model seen on Intel's `x86_64` CPUs.

To summarize, the performance trends observed are consistent across various workstations and architectures, confirming that our synthesized approaches offer a significant performance boost and can achieve a meaningful fraction of peak performance. Our FIR filtering scheme demonstrates how generation using LLVM effectively handles a wide range of filter sizes simultaneously, revealing performance benefits that would otherwise remain hidden. As outlined in the abstract and Chapter 1, this was the primary objective of the thesis.

Chapter 7

Concluding Remarks

The focus of this work has been the direct implementation of FIR filtering, which is ubiquitous in data processing, by leveraging the LLVM ecosystem and bypassing conventional compiler frontends. Our work demonstrated the feasibility and effectiveness of integrating HPC techniques—interleaved unrolling, loop tiling, and vectorization—directly at the LLVM IR level.

The benchmarking results clearly illustrate the advantages of this approach. Compared to a highly optimized C baseline, our implementation achieved performance gains of up to 11.93-fold, reaching 93% of the nominal peak performance. The results also highlight the importance of automating kernel synthesis at the IR level, as it allows for direct and effective expression of optimization strategies that would otherwise be opaque.

However, certain limitations were encountered during the project. The proprietary nature of Apple Silicon hardware made it challenging to assess and optimize performance on that platform. Moreover, on Intel CPUs, computation reordering through loop tiling, which is typically used to improve spatiotemporal locality, did not consistently yield the expected performance improvements. This suggests that further investigation is needed to better understand interaction with `x86_64` CPU architectures.

This study serves as an exploratory study towards the development of a DSL for FIR-based calculations on multidimensional signals, relevant in many areas of science and engineering, such as turbulence modelling of compressible flow problems. This DSL could leverage MLIR to automate performance optimization. Another key objective involves completing the synthesis for the rotation of 2-D, single-channel signals, which will broaden its applicability in volumetric signal and image processing. Another milestone would be the deployment of optimized libraries, which would make the project flexible and reusable across various systems.

Exploring the potential of GPU acceleration presents an enormous oppor-

tunity for further performance gains. Investigating how best to map the LLVM-based optimizations to GPU architectures would open new possibilities in optimizing performance across diverse hardware platforms.

In addition, extending the framework to handle cascading FIR filters and exploring FIR factorization through the Fundamental Theorem of Algebra could lead to more efficient implementations. This option offers an opportunity for improvement by breaking down complex filtering operations into simpler components.

The completion of these future objectives would not only strengthen the practical utility of our current approach but also provide new avenues for complex signal processing and scientific computing tasks.

Appendix A

List of Abbreviations

Table A.1: Significance of various abbreviations and acronyms used throughout the thesis. The page on which each one is defined firstly is also given. Nonstandard acronyms are not included in the list.

Abbreviation	Meaning	Page
1/2/3-D	One/Two/Three-dimensional	5
ABI	Application Binary Interface	30
ADCE	Aggressive Dead Code Elimination	23
API	Application Programming Interface	11
ASLR	Address Space Layout Randomization	31
AST	Abstract Syntax Tree	15
AVX	Advanced Vector Extensions	49
CFG	Control Flow Graph	26
CFT	Continuous Fourier Transform	9
CPU	Central Processing Unit	10
CSE	Global Common Subexpression Elimination	23
DAXPY	Double-precision AX+Y	45
DCE	Dead Code Elimination	23
DFT	Discrete Fourier Transform	9
DLP	Data-Level Parallelism	45
DSL	Domain-Specific Language	11
DSP	Digital Signal Processing	6
FFT	Fast Fourier Transform	38
FIR	Finite Impulse Response	6
FLOP	Floating Point Operations	54
FMA	Fused Multiply-Add	2
GPU	Graphics Processing Unit	44
GVN	Global Value Numbering	23
HPC	High-Performance Computing	i
I/O	Input/Output	18

Abbreviation	Meaning	Page
IDFT	Inverse Discrete Fourier Transform	10
IIR	Infinite Impulse Response	7
ILP	Instruction-Level Parallelism	41
IPO	Interprocedural Optimization	24
IR	Intermediate Representation	11
LICM	Loop Invariant Code Motion	23
LLVM	Low-Level Virtual Machine	i
M4	Macro-Processing Programming Language	10
MC	Machine Code	32
MIMD	Multiple Input Multiple Data	52
MIPS	Microprocessor without Interlocked Pipeline Stages	42
MLIR	Multi-Level Intermediate Representation	11
MMX	MultiMedia eXtensions	48
MPI	Message Passing Interface	53
MVL	Maximum Vector Length	48
NUMA	Non-Uniform Memory Access	55
OMP/OpenMP	Open Multi-Processing	52
OS	Operating System	29
OoO	Out-of-Order	41
PGO	Profile-Guided Optimization	24
PIC	Position-Independent Code	31
RAM	Random Access Memory	50
RISC	Reduced Instruction Set Computer	17
SAXPY	Single-precision AX+Y	45
SCP	Simple Constant Propagation	23
SIMD	Single Instruction Multiple Data	44
SSA	Static Single Assignment	17
SSE	Streaming SIMD Extensions	49
SVE	Scalable Vector Extensions	70
TLP	Thread-Level Parallelism	52
VLIW	Very Long Instruction Word	41
VPlan	Vectorization Plan	22
WPO	Whole-Program Optimization	24

Appendix B

Acknowledgements

This thesis would not have been possible without the support of Prof. Diego Rossinelli, who proposed the research topic and provided invaluable guidance throughout. His trust, patience, and respect have been instrumental in shaping my work. By setting high standards while also allowing me the freedom to explore my interests, he has motivated and encouraged me to succeed. I am also deeply grateful to Dr. Patrick Zulian, whose support was vital to the completion of this thesis. His insightful feedback and assistance with administrative tasks significantly eased the process. I would also like to extend my sincere gratitude to Prof. Rolf Krause for his invaluable feedback and steadfast support throughout this journey. Additionally, I would like to express my heartfelt thanks to Dr. Marco Favino for his constructive feedback.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[2] Apple Inc. Apple unveils m2 pro and m2 max: next-generation chips for next-level workflows, 2023. Accessed: 2024-09-16.

[3] Jonathan Burket et al. Visualizing the llvm compiler system. https://www.cs.cmu.edu/afs/cs/academic/class/15745-s15/public/lectures/L6-LLVM2-1up.pdf, 2015. Accessed: 2024-05-28.

[4] Intel Corporation. Intel xeon processors: Application note. Technical report, Intel Corporation, April 2016. https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Xeon-Processors.pdf, Accessed: 2024-08-17.

[5] Intel Corporation. The theoretical maximum memory bandwidth for intel® core™ x-series processors. https://www.intel.com/content/www/us/en/support/articles/000056722/processors/intel-core-processors.html, 2021. Accessed: 2024-05-28.

[6] Georg Hager and Gerhard Wellein. Introduction to the roofline model. *Journal of Supercomputing*, 36(3):375–375, 2010.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[8] Imec. 20-year roadmap: Tearing down the walls. *Imec*, 2021. https://www.imec-int.com/en/articles/20-year-roadmap-tearing-down-walls, Accessed: 2024-08-26.

[9] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[10] A. Mallik et al. Cmos rfic design principles. *IEEE Circuits and Systems Magazine*, 17(2):6–16, 2017.

[11] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, 1995. https://www.cs.virginia.edu/stream/.

[12] Michael D. McCool, James Reinders, and Reinders Robison. Scalable parallel programming with kernels. *Communications of the ACM*, 47(9):30–36, 2004.

[13] David Padua. *Encyclopedia of Parallel Computing*. Springer Publishing Company, Incorporated, 2011.

[14] LLVM Project. The llvm compiler infrastracture. https://llvm.org/doxygen/namespacellvm_1_1sys.html. Accessed: 2024-05-01.

[15] LLVM Project. Vectorization plan. https://llvm.org/docs/VectorizationPlan.html. Accessed: 2024-05-01.

[16] Dylan Reid Ramelli. Rotation of multidimensional signals with spectral schemes. Bachelor thesis, Universitá della Svizzera italiana, Lugano, Switzerland, Oct 2023.

[17] Diego Rossinelli. *Multiresolution Flow Simulations on Multi/Many-Core Architectures*. Doctoral thesis, ETH Zurich, Zürich, 2011.

[18] Diego Rossinelli, Gilles Fourestey, Hanspeter Killer, Albert Neutzner, Gianluca Iaccarino, Luca Remonda, and Jatta Berberat. Large-scale in-silico analysis of csf dynamics within the subarachnoid space of the optic nerve. *Fluids and Barriers of the CNS*, 21, 02 2024.

[19] Dave Salvator. Nvidia grace cpu superchip architecture in-depth, 2023. Accessed: 2024-09-19.

[20] Julius O. Smith. *Spectral Audio Signal Processing*. http://ccrma.stanford.edu/~jos/sasp/, Accessed : 2024-05-01. online book, 2011 edition.

[21] Laura Toma. Introduction to parallel programming with openmp. https://tildesites.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall17/Lectures/openmp.html, 2017. Accessed: 2024-05-28.

[22] M. Unser, A. Aldroubi, and M. Eden. B-spline signal processing. i. theory. *IEEE Transactions on Signal Processing*, 41(2):821–833, 1993.

[23] M. Unser, P. Thevenaz, and L. Yaroslavsky. Convolution-based interpolation for fast, high-quality rotation of images. *IEEE Transactions on Image Processing*, 4(10):1371–1381, 1995.

[24] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

HIGH-PERFORMANCE ROTATIONS OF MULTIDIMENSIONAL SIGNALS USING SPECTRAL SCHEMES

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
AMARE	BIRUK

With my signature I confirm that

- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Lugano, 08/09/2024	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.