Use the Apple M1 GPU to accelerate weather/climate/health simulations

Richard Yuan

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2023

Abstract

This thesis delves into the intricacies of a compiler process that encompasses both translation and interpretation, focusing on the transition from high-level MLIR code to executable operations on the GPU. The study introduces a comprehensive workflow for the interpreter component of the compiler, detailing its preparation and execution stages. The interpreter is designed to work seamlessly with the translator, ensuring a smooth transition from MLIR code to GPU execution. The compilation process is structured, starting with initialization, moving through translation and interpretation, and concluding with a finalization stage. The finalization ensures the syntactic correctness of the output WGSL code and the correct retrieval of GPU computation results. The study also touches upon the performance evaluation of Apple's M1 GPU, especially in the context of WebGPU, a web-based graphics and compute API.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

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

(*Richard Yuan*)

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Tobias Grosser, for his unwavering guidance, support, and expertise throughout this research journey. His insights and mentorship have been invaluable, and I am truly fortunate to have had the opportunity to work under his supervision.

I would also like to extend my heartfelt thanks to Emilien Bauer. Collaborating with him has been both enlightening and rewarding. His contributions and dedication have played a pivotal role in the progress and achievements of this project.

Furthermore, I cannot express enough thanks to my family for their continuous love, encouragement, and unwavering support throughout this year. Their belief in me and my endeavors has been a driving force behind my determination and perseverance.

To all of you, thank you for being instrumental in my academic and research journey.

Table of Contents

Chapter 1

Introduction

1.1 Research Overview

Over the past few years, a great deal of interest has gone through approving the algorithms and data analytics capabilities of the machines [1][2]. The internet age has given rise to immense connectivity and brought about the era of big data. However, while big data's possibilities and effectiveness remain for developing more effective and focused solutions, their output remains limited to the hardware and software performance. Over the past few years, Apple has been the talk of the town not only for its impressive hardware products but also for the design and ingenuity it has shown with its products. One such thing was in the form of Apple's M1 chips [3]. These chips were a detachment from the Intel-made ones and allowed Apple to have greater control over the research and development of the products and its supply chain.

The introduction of Apple's M1 chips has been met with significant acclaim. Comparative studies have consistently showcased the M1's superior performance and energy management over its competitors [4]. These attributes are pivotal for hardware projects, where efficiency and performance are paramount. The potential of Apple's M1 chips extends beyond just consumer applications; they hold promise for advancing hardware research, especially in exploring the capabilities and roles of GPUs in various computational tasks.

This research project is situated within this context, focusing on developing a compilation flow within `xDSL` from `MLIR` (a powerful intermediate representation language used in many compiler frameworks) to WebGPU Shading Language (`WGSL`). The target platform, WebGPU, is a modern graphics and compute API that can leverage the power of hardware like the M1 chips. The project employs the `xDSL` framework, a Python

native compiler framework built around SSA-based intermediate representations (IRs). `xDSL` uses multi-level IRs, which allow the implementation of abstraction-specific optimization passes, similar to the structure of common DSL compilers.

In the context of our research, the cornerstone of the compilation flow we are developing is the translator. This translator is specifically designed to convert `MLIR` code into `WGSL`, making it primed for execution on `WebGPU`. Alongside this, the interpreter in our system provides the necessary runtime support for the translated code, ensuring seamless integration and execution.

The compilation flow is integrated as a module within the `xDSL` framework. This integration augments the framework's capabilities, enabling it to compile `MLIR` to `WGSL` and execute on WebGPU, making it compatible with advanced hardware like the M1 chips. While this project primarily focuses on enhancing the `xDSL` framework, it also lays the foundation for future exploration of modern hardware's potential in tasks such as simulations. In doing so, it sets the stage for potential advancements in big data processing and analysis.

1.2 Research Motivation

The primary motivation behind this study is to explore the capabilities of Apple's M1 Chips in scientific simulations, especially in areas such as climate, weather, and health. Historically, GPUs have been extensively utilized for both simulations and data analytics, with their ability to accelerate complex computations making them indispensable in various scientific domains [5]. While GPUs have long been at the forefront of these simulations [6][7][8], the M1 represents a new breed of GPU. Its unique architecture and design pose challenges in adapting existing simulation frameworks. This study aims to bridge the gap, ensuring that the M1 can effectively support the demands of these scientific simulations.

The speed of obtaining results is a critical consideration across computational platforms. While questions related to processing methodologies and the number of iterations required are primarily addressed at the numerical analysis level, the underlying hardware can influence the overall performance and efficiency. In the context of big data [9], these considerations are paramount, and they are equally significant in the realm of physical simulations. Evaluating the performance of the M1 Chip in these scenarios can offer insights into its potential to enhance scientific simulations, especially in areas like climate change modeling, health, and weather forecasting.

It is needless to say that these areas of scientific research are of any great importance. Their importance and relevance for the scientific analysis cannot be disputed. Moreover, this area of research has also great deal of relevance for the algorithms and machine learning. As the name implies machine learning is about making machines learn and work on their own [9]. Algorithms have a great role and influence to play in this process [10][11]. However, the application of the algorithms or their success rate can also be impacted by the processing powers of the machines and their capabilities. For instance, the ability of the machine to handle single and multiple iteration efficiently can not only impacts on the accuracy of the algorithm's but also how it they will be developed and impacted. Hence one can suggest that the capabilities of the processors and machines are very relevant to the computing capabilities of these machines.

This research aims to explore the potential of Apple's M1 chips in scientific computation and analysis. By enabling computational flows on the M1 chip, this study contributes to the existing literature by shedding light on the practical applications of this advanced hardware in scientific contexts. While there has been significant emphasis on enhancing the power profile and energy efficiency of chips, it's essential to understand how these advancements translate to real-world computational tasks. Preliminary observations suggest that Apple's M1 chips offer competitive energy efficiency and performance. However, this research primarily focuses on the feasibility and practicality of running specific computational flows on the M1, rather than an exhaustive performance or energy analysis.

1.3 Research Gap and Significance of the Study

The research gap addressed in this study is twofold. Firstly, there's a need to bridge the `xDSL` framework to the Apple M1 chip. While NVIDIA's CUDA platform offers tools like profiling and harnesses the full power of NVIDIA GPUs, it's specific to NVIDIA hardware and is not compatible with Apple's M1. The mention of CUDA serves to highlight the importance of having specialized tools and frameworks that can maximize the potential of specific hardware architectures. In the case of Apple's M1, frameworks like `xDSL` currently lack support, and the intricacies of `MLIR` dialects on this architecture remain largely uncharted. This study aims to address these gaps by implementing the `xDSL` and compiling `MLIR` dialects to `WGSL` for execution on the M1 GPU. Such efforts are crucial for gauging the computational prowess of Apple M1 chips and ensuring that software frameworks can fully leverage their capabilities.

1.4 Contribution

This research project aims to fill this gap by developing a compilation flow using the xDSL framework that translates MLIR code to WebGPU Shading Language (WGSL), ready for execution on WebGPU, potentially leveraging the power of Apple's M1 chips. By doing so, it will shed light on the possibilities and performance of using M1 chips for scientific computation and analysis.

This research will not only contribute to the ongoing development of the xDSL framework, but also explore the potential of modern hardware in simulations, thereby pushing the boundaries of big data processing and analysis. The project's findings could lead to significant improvements in data processing speed, algorithm performance, and energy efficiency, with potential applications in fields such as climate modeling, weather simulation.

1.4.1 Implementation Contributions to the xDSL Project

During the progression of this research, I contributed to the xDSL project, with a focus on the interface between MLIR and the WebGPU Shading Language (WGSL) and the WebGPU API. The specifics of my contributions are as follows:

1. **WGSL Printer Module and its Tests**:
    - *Module* (GitHub Link): I was responsible for the 'wgsl_printer' module, which translates MLIR code to WGSL. This module provides a mechanism to convert MLIR representations into a format suitable for the WebGPU API.
    - *Tests* (GitHub Link): To validate the 'wgsl_printer' module, I implemented tests that ensure the module's translations are accurate and consistent.
2. **WGPU Interpreter Module** (GitHub Link):
    - In collaboration with Emilien Bauer, we worked on the 'wgpu' interpreter module. This module utilizes the 'wgsl_printer' for MLIR to WGSL translation and subsequently executes the WGSL using the WebGPU API.

Chapter 2

Background

The objective of this chapter is to provide historical and contextual background for the research. It delves into the evolution of the ARM architecture, highlighting its distinctions from other prevalent silicon architectures in the industry. The chapter further explores the computational capabilities of GPUs, detailing how they can be interfaced with and evaluated through various benchmarks. A significant portion of this chapter is dedicated to reviewing studies on the performance of Apple Silicon, offering insights into its potential in the realm of scientific computing.

2.1 Apple Arm Architecture

The ARM architecture, which originated from Acorn Computers in the 1980s, has established a strong presence in mobile devices due to its efficiency and power management capabilities [3][4]. However, its adoption in personal computers has been more restrained compared to Intel-based systems. Several inherent features of ARM architectures make them appealing for a range of computational tasks [12][13]:

- Load/Store Architecture
- Integrated Security for enhanced data storage and management
- Orthogonal Instruction Set for optimized memory and processing
- Single-Cycle Execution for efficient instruction processing
- Superior Energy Efficiency due to an efficient transistor network
- Capability to switch between 32-bit and 64-bit modes [14]

- Enhanced Hardware Visualization

These features, combined, holds significant potential for data analysis and scientific exploration. While its dominance in mobile devices is a testament to its efficiency and power management, its application in broader computational domains, especially data analysis and simulation, remains an area of active research.

However, ARM processors have their limitations. Historically, they've faced challenges in supporting certain data analysis programs due to a lack of native programming support, leading some to argue their better suitability for simpler tasks. This preference for performance over efficiency is a key reason ARM chips initially found more traction in mobile devices than in personal computers. Nevertheless, recent advancements, spearheaded by companies like Apple, have expanded ARM's reach into the computing domain. Modern ARM chips, as seen in Apple's lineup, are lauded not just for their computational prowess but also for their energy efficiency.

The ARM infrastructure, as implemented by Apple in their chip design, offers not only a wide range of personal computing solutions but also potential applications in data science and scientific research. Data science is an interdisciplinary field that uses various techniques, algorithms, processes, and systems to extract knowledge and insights from structured and unstructured data. It encompasses a range of activities, from data analysis and visualization to machine learning and advanced statistical modeling. Given ARM's efficiency, particularly in reducing data transfer time and overall system load, many experts are speculating about its potential benefits for data science tasks and broader research and development initiatives.

While the M1 architecture offers significant advancements, it also presents certain challenges for software compatibility. Notably, many native or platform-specific apps and programs have yet to be optimized for the M1. The M1 isn't directly targeted by the `MLIR` GPU dialect. The `xDSL` framework serves as a bridge, enabling the execution of WebGPU on the M1. Our goal is to utilize the `xDSL` framework to run specific computational tasks on the M1 chip, thereby harnessing its capabilities for a broader range of applications.

2.2 Relationship between GPU's and Compliers

GPU acceleration plays a crucial role in enhancing the performance of compiled code, especially for computationally intensive tasks [14][15]. It's not merely about increasing

the speed of operations; GPU acceleration broadens the scope of addressing intricate computational challenges. This is particularly evident in domains like data analysis and simulation. For instance, a study by Fuqiu [16] highlighted that a hybrid optimization approach, utilizing both GPU and CPU resources, can markedly enhance data simulation and analysis. This optimization is particularly beneficial for stencil computations, which are known to thrive with GPU acceleration. Such advancements primarily boost computational efficiency and the ability to process data concurrently.

The crux of this performance enhancement lies in the compilers. They bridge the gap between high-level programming languages and the hardware infrastructure, translating code in a manner that optimally utilizes the capabilities of GPUs.

Compilers are instrumental in translating high-level programming into instructions that hardware can execute. The `xDSL`, a burgeoning python-native framework, holds potential for applications in simulation, graphical processing, and analysis [15]. While domain-specific languages (DSLs) like `xDSL` have been recognized for their value, a recurring challenge is the tendency to reimplement the entire stack for each new DSL. This can lead to redundancy and inefficiencies. However, the primary advantage of DSLs is their ability to address specific optimization and communication challenges between software and hardware, offering tailored solutions that general-purpose languages might not provide [2][6].

Historically, high-performance simulation codes have been notoriously challenging to optimize on supercomputers[4][3]. Even with powerful GPUs or CPUs, the real potential of the hardware is often contingent on the optimization capabilities of the compilers and their ability to effectively translate and transform information. A key advantage of DSLs is their potential to improve the dialogue between software and hardware, requiring less bespoke expertise.

The challenge today isn't just about having advanced hardware like Apple's M1 chips. It's about making optimization and integration with such hardware more accessible. Currently, there's a dependence on a handful of experts who excel at optimizing code for supercomputers. The vision is to reduce the reliance on this limited expert-time, making the optimization process more straightforward and less expert-dependent. By doing so, the broader scientific community can navigate the myriad of compilers and the challenges they present in harmonizing with varying hardware.

2.3 LLVM and MLIR

Two compiler frameworks that stand out as potential solutions for a unified ecosystem are LLVM and MLIR. These frameworks aim to address the incoherence between various hardware and compilers. Multi-Level Intermediate Representation (MLIR) was developed as an evolution beyond the Low-Level Virtual Machine (LLVM). However, using LLVM has its challenges. A notable issue is the significant number of transformations required between the high-level representation of user code and the LLVM intermediate representation [17][18]. This efficiency concern has somewhat limited the broader adoption of LLVM. To tackle this efficiency problem, Google introduced MLIR. MLIR offers greater openness and flexibility in its architectural design and its interaction with hardware and software. A significant advantage of MLIR is its extensibility, allowing developers to integrate their own codes and frameworks while maintaining a level of flexibility comparable to LLVM. This positions MLIR as a promising compiler or intermediate representation for a unified ecosystem.

There are many other benefits that these two compliers offer as an intermediate representation. Firstly, they are supported by the large sets of developer and programmer communicates, which makes their application and implementation highly plausible. This also add into second key advantage of theirs is that they have wealth of third-party tooling, such as the debuggers and profilers, exists. This is considered to be highly crucial as the as in the case of high-performance computer optimisation, it can have a substantial influence and impact [18][19].

xDSL compliers seems to offer great deal intermediating source between the MLIR and LLVM frameworks [15]. The xDSL framework allows the translation and manipulation of the MLIR dialects and fundamentals. The open-source nature of the xDSL also expands its application capabilities. The xDSL also has scientific application, which is noted in the form of PSyclone[20] a Fortran-based DSL developed by the STFC. It is currently being used by the Met office for the weather and climate models. By integrating the xDSL with the PScylone, plenty of its shortcomings and limitations can be addressed. Some of these being that lack of uniformity and in-efficiency that requires the developers to make system specific codes. However the work of Brown [15] how xDSL integration can improve the overall functioning of the PScyclone. This not only instrumental from expanding the application of the new and improved technologies such as seen in the form of the Apple M1 chips, but at the same time, it also has great deal of benefit for the scientific research and development.

As argued earlier that modern scientific analysis and data computation are not only limited by the algorithmic powers and applications but also the how these systems operate with the hardware. This increasing the role and application of common ecosystem, that increases the application and implementations of compliers such as the MLIR.

2.4 Capturing the computational capabilities of GPU

As highlighted earlier that using GPU for data analysis and computational purposes, is less straightforward than the CPU. However the framework such as the WebGPU, Shading languages and CUDA frameworks have been developed for these purposes. The purpose of the WebGPU to work as a bridge (API) between the web browsers and GPU[21](see figure 2.1).

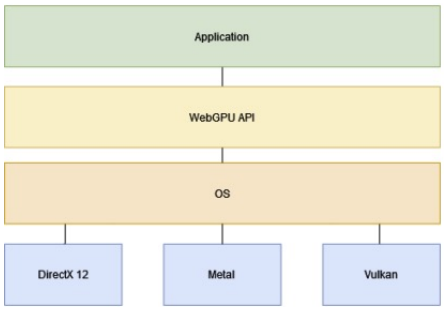


Figure 2.1: The architecture of WebGPU[22]

WebGPU Shading Language (WGSL), commonly referred to as shaders or kernels, serves as the primary language for WebGPU. While WGSL is the default communicator, other languages like the OpenGL Shading Language (GLSL) can also harness the power of GPUs [23][24]. WebGPU is pivotal for advanced 3D graphics and executing intricate computations such as Generic Matrix Multiplication (GEMM), Sparse Matrix Vector Multiplication (SPMV), and Fast Fourier Transform (FFT) [25]. Research, like the study by Dyken [26], suggests that WebGPU can significantly enhance data visualization and analysis. However, a challenge arises when interfacing with MLIR's GPU infrastructure, which currently lacks seamless compatibility with the Apple M1 architecture. This means users might face hurdles in terms of compilers and benchmarks, and achieving optimal compatibility remains a challenge.

Another pivotal framework in harnessing the computational capabilities of GPUs is CUDA, developed by NVIDIA. Unlike general-purpose programming languages, CUDA is specifically designed to facilitate GPU-accelerated applications. By providing

an extended C++ programming environment, CUDA allows developers to write parallel code for NVIDIA GPUs directly, bypassing traditional graphics APIs.

CUDA's strength is not just limited to graphics; it has found extensive application in the realm of data analysis and scientific computing. Its architecture is optimized for machine learning and deep learning frameworks, with popular libraries like TensorFlow, PyTorch, and Keras being CUDA-enabled for GPU acceleration [27].

However, it's worth noting that while CUDA is a powerful tool for NVIDIA GPUs, its application is limited when it comes to other architectures, such as Apple's M1 chip. This study aims to explore the potential of the `xDSL` framework on the M1, and how alternative GPU programming approaches can be integrated to harness the full capabilities of this chip.

2.5 Performance of Apple M1 Chips

In recent years, there has been a surge in research interest regarding the computational capabilities of Apple's M1 chips. Numerous studies have delved into the potential of the M1 chip in areas such as deep learning, machine learning, and scientific computing [3][6][25]. However, many of these investigations offer preliminary insights, suggesting there's more to uncover about the full potential of the M1 chip.

For example, Kenyon and Campano's study [3] explored the scientific computational capabilities of the M1 chip by comparing it to NVIDIA chips, which are renowned for their prowess in scientific computations. The benchmarking methodologies differed: the M1 chip was evaluated using OpenCL, a widely-accepted open standard for parallel programming, while NVIDIA chips were assessed via CUDA, a proprietary platform tailored for NVIDIA GPUs. Given CUDA's optimization for NVIDIA hardware, it's essential to approach such comparisons with caution. While the M1 demonstrated commendable performance in tasks like Generic Matrix Multiplication (GEMM), Sparse Matrix Vector Multiplication (SPMV), and Fast Fourier Transform (FFT), it's possible that the full extent of its capabilities hasn't been tapped into yet.

This underscores the need for further research, especially in exploring how frameworks like `xDSL` can be integrated with the M1 chip to maximize its potential.

Kasperek's study [2] offers a deeper dive into the scientific computational potential of Apple's M1 chip, particularly focusing on its machine learning capabilities. Unlike many previous studies that primarily assessed the graphical processing capabilities of the M1 chip, Kasperek's research emphasized its prowess in handling and processing

large datasets for machine learning tasks.

Two primary aspects set Kasperek's research apart. First, it centered on both training and analysis of data, offering a more holistic view of the M1 chip's capabilities in the realm of machine learning. Second, the study employed Apple's 'Create ML' tool, emphasizing the importance of hardware-software synergy. By using a tool designed specifically by Apple, the research aimed to minimize compatibility issues, which often act as bottlenecks in harnessing a chip's full potential.

Kasperek's findings indicate that the M1 chip outperforms older models based on non-Apple silicon. This underscores the significance of ensuring harmony between hardware and software. For instance, earlier Apple laptop models, which relied on Intel chips, didn't fare as well in similar tasks as their M1-based counterparts [2]. However, it's worth noting that while Kasperek's study sheds light on the M1 chip's capabilities in scientific computing, there's still much to explore in this domain.

More in-depth research on the application of Apple's M series chip was done by the work of Gebraad and Fichtner[6]. The study just like the previous studies on the topic tends to only examine the phenomena from only one direction. For instance, the study by Gebraad and Fichtner[6] look at the utilisation of CUDA and MSL (Metal Shading Language), how they can be used to improve the scientific computation on the Apple's M series devices. Compared to this the study done by Ali[4] examined the application of Apple's M1 chip on the data centres, found that Apple's M1 chips outperform the x86 in terms of energy management and data processing. This further highlights the potential that these chips can have when it comes to the energy management and data analysis.

The exploration of Apple's M1 chip in the context of scientific computing is still in its early stages, with limited literature available. While initial findings suggest that the M1 chip offers promising advancements, particularly in terms of energy efficiency and integrated architecture, it's essential to approach these claims with caution.

Several studies have touched upon the potential of the M1 chip in scientific computing, machine learning, and data science. However, many of these studies have only scratched the surface, focusing primarily on the chip's basic capabilities without delving deep into its full potential. A significant gap in the literature pertains to the exploration of compilers and frameworks that can effectively harness the M1 chip's GPU capabilities. While there is mention of a study leveraging the Metal Shading Language (MSL) to tap into the M1's GPU power, a comprehensive understanding of its performance remains elusive.

The current literature lacks a detailed examination of the interplay between compilers, interfaces, and the M1 architecture. While theoretical possibilities exist, as highlighted in some studies, practical implementation presents challenges. It's crucial to recognize that without rigorous empirical analysis, we cannot fully grasp the true potential of Apple's M series chips. In essence, while the M1 chip presents an exciting development in the realm of computing, it's essential to temper expectations and focus on its specific advantages, such as energy efficiency and integrated design, rather than broad revolutionary claims.

Chapter 3

Methodology

The methodology section for this research delineates the explicit procedures and techniques employed in our approach to compiling MLIR[28] to the WGSL [29] using xDSL [30]. This part of the thesis is structured into the following four subsections:

- Compiler Architecture and Design: In this section, we offer an initial overview of our compiler strategy for the MLIR to WebGPU translation via xDSL. We shed light on the two key components of our compiler: the translator and the interpreter. We delve deeper into the compiler's architecture, illustrating the intricate interplay between the translator and the interpreter, both of which are crucial to the operation of the compiler.

- Design and Functionality of the Translator: This section delves into the straightforward and efficient design of the translator, a core component of the compiler. Using a single dispatch mechanism, the translator ensures simplicity and effectiveness in its operations. Its primary role is to convert MLIR representations into the WGSL, facilitating further processing and execution.

- Design and Functionality of the Interpreter: This section provides an in-depth explanation of the interpreter, the second integral component of our compiler. The interpreter works in concert with the translator to execute the translated xDSL code in the WebGPU environment. We discuss the interpreter's design elements, its operational characteristics, and its role in the overall compilation process.

- Compilation Workflow: This section describes the systematic procedure of our compilation approach, transitioning from source (MLIR) to target (WGSL) via xDSL.

- Interpreter Workflow: This section outlines the stages of the interpreter, from its preparation through the execution of the translated `xDSL` code.

This arrangement provides a clear overview of the primary processes, methods, and strategies employed in our research. We emphasize the roles of the translator and interpreter within the compiler, ensuring readers gain a straightforward understanding of our approach. Subsequent sections of the thesis will delve into the experimental setup, testing procedures, results, validation, and a discussion on potential challenges and considerations.

3.1 Compiler Architecture and Design

At the core of our methodology is Python's single dispatch mechanism, a powerful tool for organizing various functions based on the operation type present in the `MLIR` code. In combination with the `xDSL` framework we've created an efficient and dynamic system for translating `MLIR` code into `WGSL`. Our compilation flow comprises two primary components: the translator and the interpreter, which work in tandem to translate `MLIR` into `WGSL` and then execute the `WGSL` code via `WGPU`.

The translator begins by identifying and translating the `gpu.func` operation within the `MLIR` code. This operation encompasses a variety of arguments and operations, each necessitating a specific translation into `WGSL`. Proper translation requires a comprehensive grasp of the semantics and implications of each operation within the `MLIR` code. To cater to the diverse operations within `MLIR`, our design employs Python's single dispatch method. Each distinct operation type in the `MLIR` is linked to a specific translation method. The dispatcher function then assigns the operation to its respective translation function. If an unsupported operation is detected, a `NotImplementedError` is raised, ensuring the translation's integrity is maintained.

Subsequent to the translation process, the interpreter comes into play. Working in collaboration with the translator, the interpreter is responsible for executing the translated `WGSL` code via `WGPU`. It takes the translated `WGSL` constructs and transforms them into corresponding `WGPU` API calls, enabling the operations to be executed on the GPU. The design of this component serves as an essential check on the translation process's effectiveness and acts as a crucial link to the overall success of the compiler.

Our compiler's architecture, undergirded by the synergy between Python's single dispatch mechanism, the `xDSL` framework, and the `WGPU`, offers the necessary flexibility

and scalability to translate MLIR into WGSL and subsequently execute it via WGPU. This dual-component structure of the compiler ensures a thorough and efficient translation process, navigating the intricacies of MLIR and ensuring accurate execution in WGPU.

3.2 Design and Functionality of the Translator

Our translator's design is informed by the hierarchical structure of MLIR. While at first glance, the operations nested within Regions & Blocks might suggest a tree-like structure, the MLIR framework is more intricate. Operations can be interconnected through SSAValues, making the structure more akin to a graph. Recognizing this graph-like nature is essential to our approach. In this context, we treat the `gpu.launch_func` operation as a central point of reference, though not in the traditional sense of a "root node" in tree structures.

The translation process involves sequentially processing the list of operations in a block, ensuring each is accurately converted to its corresponding WGSL structures.

A cornerstone of our translator is its integration with the xDSL framework. As a domain-specific language tailored for MLIR, xDSL provides us with a suite of tools for efficiently handling and transforming MLIR code. It serves as the foundation of our translator, facilitating the conversion from MLIR to WGSL.

The translation process commences within the `gpu.module`, where the translator dissects the encapsulated `gpu.func`. Within this function, the translator encounters an array of arguments and operations, each requiring a specific translation approach into WGSL.

Following the argument translations, the translator proceeds to handle the individual operations housed within the `gpu.func`. This is facilitated by the single dispatch method, which delegates each operation to its dedicated translation function based on its type. This flexible design not only ensures comprehensive coverage of all operation types but also allows for easy expansion or modification of the translator's capabilities by simply adding or adjusting the dedicated methods for different operations.

Our translator incrementally writes the WGSL constructs into a output stream, rather than accumulating the translated code in memory. Once the translation process is complete, the content of the output stream is extracted as a string, representing the fully translated WGSL code.

Informed by the intricacies of MLIR, our translator is designed to effectively convert it into WGSL. By utilizing the capabilities of xDSL and Python's single dispatch mechanism,

we aim for our translator to offer flexibility, scalability, and efficiency. While the design decisions were made with these goals in mind, comprehensive evaluations would be needed to fully demonstrate these attributes.

3.3 Design and Functionality of the Interpreter

The interpreter, the second critical component of our compiler, works in tandem with the translator to execute the translated `xDSL` code in the WebGPU environment. Its design is focused on interpreting GPU operations using the WebGPU API, allowing for the execution of operations on the GPU.

Our interpreter extends the functionality of a class from the `xDSL` framework. This class, known as InterpreterFunctions, serves as a foundational structure for our interpreter. The InterpreterFunctions class in the `xDSL` framework is a fundamental building block that provides a set of functionalities for interpreting operations. Our interpreter builds upon this by incorporating operations that are specific to the WebGPU environment.

Within our work, the principal class, `WGPUFunctions`, encompasses several methods tailored to facilitate and execute WebGPU operations. For instance, the `buffer_from_operand` method is designed to prepare a GPUBuffer from an SSA operand. Similarly, the `prepare_bindings` method is dedicated to preparing argument bindings, a crucial component of the WebGPU API. The `compile_func` method ensures a GPU function is compiled if it hasn't been previously, leveraging the `WGSLPrinter` to produce the `WGSL` source code for the function and then compiling it into a GPU shader module.

The interpreter also implements various GPU operations. The `run_alloc` function manages the GPU allocation operation (`gpu.alloc`), allocating a GPUBuffer and returning it as the memref value. The `run_memcpy` function handles the GPU memory copy operation (`gpu.memcpy`), supporting both device-to-host and host-to-device copies. The `run_launch_func` function implements the GPU kernel launch operation (`gpu.launch_func`), setting up and launching the GPU kernel using the WebGPU API.

Each of these methods has a crucial role in the interpretation process, converting the `xDSL` code into `WGSL` that can be executed by `WGPU` on the GPU. By harnessing the power of the `WGPU` API, our interpreter can directly execute GPU operations, providing an essential link in the overall compilation process.

In summary, the design of the interpreter ensures a seamless transition from the translated `xDSL` code to executable `WGPU` operations, forming an efficient and effective

bridge between the translation and execution stages of the compilation process.

3.4 Compilation Workflow

The compilation process, which encompasses both translation and interpretation, is a multi-stage task that begins with reading the input `MLIR` code. The translator, leveraging Python's single dispatch method, navigates through the `MLIR` code structure, interpreting it as a tree with the `gpu.launch_func` serving as the root.

Once the translator completes its task, generating the intermediate representation (`xDSL`), the interpreter takes over. The interpreter, designed to work with the WebGPU API, reads the translated `xDSL` code and executes it on the GPU.

Thus, our compilation process constitutes an intricate dance between the translator and the interpreter, seamlessly moving from the high-level `MLIR` code to executable operations on the GPU. The process ensures not just the accurate translation of `MLIR` into `WGSL`, but also its successful execution in the WebGPU environment, providing a comprehensive solution for compiling `MLIR` to the `WGSL` using `xDSL`.

3.4.1 Initialization

The compilation process starts with the initialization of key components for the translator and the interpreter. However, this wasn't straightforward due to a few issues we encountered.

One of the first issues was ensuring the uniqueness of variable names during the translation process. To tackle this, we initialized the `WGSLPrinter` class for the translator, which forms the backbone of the translation process. During its initialization, the `WGSLPrinter` class creates an empty dictionary (`name_dict`) and a counter variable (count) set to zero. The `name_dict` dictionary was our solution to quickly look up unique names corresponding to each SSAValue, thus streamlining the translation process.

However, not all SSAValue instances have a `name_hint`, which created another problem. We addressed this by using the count variable to generate unique names for such SSAValue instances. Each time we encountered a new SSAValue without a `name_hint`, we incremented count to provide a unique name for that SSAValue.

The `wgsl_name` method was implemented to manage this unique name generation. It takes an SSAValue as input and checks its presence in `name_dict`. If it's already

present, it returns the associated name. If not, it uses the SSAValue's `name_hint` (if available) or generates a new unique name using count, updates `name_dict` with this new pairing, and then returns the unique name.

Next, we faced the challenge of executing the translated `WGSL` code using the WebGPU API. For this, the interpreter component needed to prepare the necessary environment and data structures. Our solution was to have the interpreter initialize the setup of GPU buffers, prepare argument bindings, and initialize GPU functions for compilation.

In summary, the initialization phase lays the groundwork for the rest of the compilation process by addressing these key issues. It ensures that each operation is processed accurately and efficiently, setting the stage for a seamless transition from `MLIR` to `WGSL` and, finally, to executable GPU operations.

3.4.2 Translation Process

Once the `WGSLPrinter` is initialized, the translation process moves on to the core of the operation: processing individual operations based on their type. This is achieved using the print function of the `WGSLPrinter`, which is designed as a single dispatch method. This method dispatches calls to different functions based on the type of the operation being processed. Here's a breakdown of how different operation types are handled:

3.4.2.1 Example `MLIR` to `WGSL`

Figures 3.1(a) and 3.1(b) show an example of `MLIR` dialects selected and translated into `WGSL`, where I have labelled the transformations of each operation from (1) - (9) in both figures.

```
"gpu.func"() ({
^0(%arg : memref<4x4xindex>①):
  %0 = "arith.constant"()② {"value" = 2 : index} : () -> index
  %1 = "gpu.global_id"()③ {"dimension" = #gpu<dim x>} : () -> index
  %2 = "gpu.global_id"() {"dimension" = #gpu<dim y>} : () -> index
  %3 = "arith.constant"() {"value" = 4 : index} : () -> index
  %4 = "memref.load"(%arg, %1, %2)④ {"nontemporal" = false} : \\
       (memref<4x4xindex>, index, index) -> (index)
  %5 = "arith.muli"(%1, %4)⑤ : (index, index) -> index
  %6 = "arith.addi"(%5, %2)⑥ : (index, index) -> index
  "memref.store"(%6, %arg, %1, %2)⑦ {"nontemporal" = false} : \\
       (index, memref<4x4xindex>, index, index) -> ()
  "gpu.return"() : () -> ()
}) {"function_type" = (memref<4x4xindex>) -> (),
    "gpu.kernel",
    "gpu.known_block_size"⑧ = array<i32: 128, 1, 1>,
    "sym_name"⑨ = "fill"
    } : () -> ()
```

(a) `MLIR` dialects

```
@group(0) @binding(0)
var<storage,read_write> varg: array<u32>①;

@compute
@workgroup_size(128,1,1)⑧
fn fill⑨(@builtin(global_invocation_id) global_invocation_id : vec3<u32>,
@builtin(workgroup_id) workgroup_id : vec3<u32>,
@builtin(local_invocation_id) local_invocation_id : vec3<u32>,
@builtin(num_workgroups) num_workgroups : vec3<u32>) {

  let v0 : u32 = 2u;②
  let v1: u32 = global_invocation_id.x;③
  let v2: u32 = global_invocation_id.y;
  let v3 : u32 = 4u;
  let v4 = varg[4u * v1 + 1u * v2];④
  let v5 = v1 * v4;⑤
  let v6 = v5 + v2;⑥
  varg[4u * v1 + 1u * v2] = v6;⑦
  }
```

(b) Translated `WGSL` code

Figure 3.1: Example `MLIR` operations translate to `WGSL` code

3.4.2.2 ModuleOp

While processing operations of the ModuleOp type, the translator was originally designed to iterate over the body of the operation and handle instances of FuncOp. The objective was to ensure that all function operations within a module were correctly managed by their respective handlers.

However, during implementation, we encountered a challenge. The current translation process failed to work correctly for multiple functions within one module. As a workaround for this issue, we adopted a strategy of compiling the functions separately for now.

While this approach ensures successful compilation, it's important to note that it represents a divergence from the initial design. Therefore, a potential area of future work could involve refining the translator to effectively handle multiple functions within a single module, as initially intended. This improvement would streamline the translation process and bring the implementation closer to the original design.

3.4.2.3 FuncOp

The processing of FuncOp operations begins by setting the workgroup size. If the operation has a known block size, that size is used as the workgroup size. If not, the workgroup size is set to one as a default.

The function then iterates over the arguments of the operation one by one. For each argument, it analyzes its type, which could be `f32`, IndexType, or MemRefType. Depending on the type, the corresponding `WGSL` type is determined. For `f32`, the `WGSL` type is `f32`. For IndexType, the `WGSL` type is `u32`. For MemRefType, the `WGSL` type is set as an array of the element type.

This process also considers that `WGSL`'s design prioritizes efficient GPU workload execution and hardware compatibility. As a result, storage class variables in `WGSL` expect structures or arrays, not scalars. Therefore, when scalar data is encountered, it is directed towards uniform buffers instead, which are designed to handle smaller, infrequently changing data.

After determining the type for each argument individually, the translator immediately writes the corresponding `WGSL` code to the output stream. This code defines a variable for the argument with the appropriate type. This step-by-step process of analyzing and printing ensures that each argument is handled accurately and efficiently.

Finally, once all arguments have been processed and their `WGSL` code written, the

translator writes the WGSL function definition to the output stream. This includes the set workgroup size and the processed arguments. This approach, where each argument is handled individually, makes the handling of FuncOp operations more elegant and streamlined.

3.4.2.4 ReturnOp, ModuleEndOp

These operation types represent the end of a function or a module, respectively. They do not require any specific processing or translation, so no WGSL code is written to the output stream for these operations.

3.4.2.5 Processing of Dimension-Related Operations

Certain operations in the MLIR code are related to the dimensions of the GPU execution model. These operations include BlockIdOp, ThreadIdOp, GridDimOp, and GlobalIdOp. For each of these operations, the translator extracts the dimension information, generates a unique name for the result, and writes the corresponding WGSL code to the output stream. This process involves defining a u32 variable with the extracted dimension value.

In the MLIR code, operations like GlobalIdOp, ThreadIdOp, GridDimOp, and BlockIdOp play significant roles. GlobalIdOp returns the unique global id, ThreadIdOp returns the thread id or the index of the current thread within the block, GridDimOp returns the number of thread blocks in the grid, and BlockIdOp returns the block id or the index of the current block within the grid.

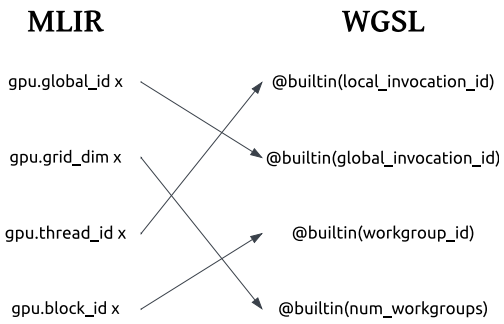
In WGSL, the corresponding operations are global_invocation_id, local_invocation_id, num_workgroups and workgroup_id. global_invocation_id represents the current invocation's global invocation ID, local_invocation_id represents the current invocation's local invocation ID, num_workgroups represents the dispatch size of the compute shader dispatched by the API, and workgroup_id represents the ID of the workgroup where all invocations have the same ID.

A challenge arises from the fact that these are built-in function arguments in WGSL, whereas in MLIR, they are more like functions. This difference required a strategy to accurately map MLIR operations to their WGSL counterparts.

Our solution was to match each MLIR operation with its corresponding WGSL function argument. We mapped GlobalIdOp with global_invocation_id, GridDimOp

with `num_workgroups`, `ThreadIdOp` with `local_invocation_id`, and `BlockIdOp` with `workgroup_id`. This approach ensured the accurate representation of the grid dimensions, block dimensions, thread and local invocation dimensions, and global invocation dimensions in the translated `WGSL` code.

In this way, despite the differences between `MLIR` and `WGSL`, we ensured that crucial grid and block dimension information was accurately translated, maintaining the correct semantics in the resulting GPU code.The graph on RHS provides a clear understanding of how each `MLIR` dimension-related operation corresponds to its equivalent in `WGSL`. This translation map is a crucial part of the translator's strategy to accurately and efficiently convert the dimension-related operations from `MLIR` to `WGSL`.



3.4.2.6 Load, Store

The translator's processing of GPU memory operations involves generating unique names for the memory references and indices, calculating the linearized index for storage and load operations, and writing the corresponding `WGSL` code to the output stream.

One specific challenge arises from the fact that the WebGPU Shading Language (`WGSL`) does not natively support multi-dimensional arrays. In fact, `WGSL` only allows 1D arrays, which are stored in a contiguous block of memory, making it easier for the GPU to access the data.

```
@print.register
def _(self, op: memref.Load, out_stream: IO[str]):
    load_ref = self.wgsl_name(op.memref)
    name_hint = self.wgsl_name(op.res)
    indices = [self.wgsl_name(i) for i in op.indices]
    index_value = self.calculate_index(op, indices)
    out_stream.write(
        f"""
    let {name_hint} = {load_ref}[{index_value}];"""
    )
```

```
@print.register
def _(self, op: memref.Store, out_stream: IO[str]):
    value = self.wgsl_name(op.value)
    store_ref = self.wgsl_name(op.memref)
    indices = [self.wgsl_name(i) for i in op.indices]
    index_value = self.calculate_index(op, indices)
    out_stream.write(
        f"""
    {store_ref}[{index_value}] = {value};"""
    )
```

Figure 3.2: `Load`, `Store` Translator

Given this limitation of `WGSL`, any multi-dimensional array that needs to be processed by the translator must first be converted into a 1D array. This is achieved through the `calculate_index` function, which generates a linearized index for accessing the elements of the multi-dimensional array in its flattened 1D form.

Therefore, the handling of multi-dimensional arrays in this manner is not an opti-

mization but a necessary step to meet the requirements of WGSL. This solution is critical for ensuring that the translator can handle any array, regardless of its dimensions, and translate it into a form that can be processed efficiently by the GPU. This solution to the multi-dimensional array challenge is a key aspect of the translator's functionality and its ability to generate valid and efficient GPU code.

3.4.2.7 Arithmetic Operations

Arithmetic operations involving integers and floating-point numbers, including addition (Addi, Addf), multiplication (Muli, Mulf), and subtraction (Subi, Subf), are processed by the WGSLPrinter. The process involves generating unique names for the operands and the result of the operation. Following these steps, the corresponding WGSL code is then written to the output stream.

3.4.2.8 Arithmetic Operations: Constant

The processing of Constant operation requires special handling when the constant type is an unsigned integer (u32). In WGSL, unsigned integers are not allowed to have a sign associated with them, meaning they cannot be negative. When a negative value is assigned to an unsigned integer, it is converted according to the rules for signed-to-unsigned conversion in WGSL. This conversion reduces the value modulo $UINT_{MAX} + 1$ (which is $2^{32} + 1$ for u32), resulting in a non-negative value.

Therefore, when processing a Constant operation where the constant type is u32, if the value is negative, it is converted to a non-negative value by adding $2^{32} + 1$ to it. This ensures that the constant value is valid in the context of WGSL and adheres to the language's rules for unsigned integers.

Through this selective processing of operations, the translator ensures that each operation is translated accurately and efficiently into WGSL code.

3.5 Interpreter Workflow

This section details the comprehensive workflow of the interpreter component of our compiler, from the initial preparation stage through the execution of the translated xDSL code in the WebGPU environment. The workflow is divided into two main subsections: Interpreter Preparation and Interpretation Process.

3.5.1 Interpreter Preparation

The interpreter preparation stage sets the stage for the execution of the translated `xDSL` code in the WebGPU environment. The interpreter's main class, WGPUFunctions, is initialized and readied for the upcoming interpretation process.

At this stage, the interpreter prepares the necessary environment and structures for the WebGPU API. This includes setting up GPU Buffer objects using the `buffer_from_operand` method for the SSA operands that will be involved in the GPU operations. This method reads the SSA operand, which is expected to be a GPU Buffer at this stage, and prepares it for the GPU operations.

Next, the `prepare_bindings` method sets up the argument bindings for the WebGPU API. These argument bindings are essential for correctly setting up the GPU operations in the WebGPU environment.

Finally, the `compile_func` method is prepared to compile the GPU functions that have been translated into `WGSL` by the translator. It uses the `WGSLPrinter` to generate the `WGSL` source code for the function, which is then compiled into a GPU shader module ready for execution.

These preparation steps ensure that the interpreter is ready to handle the translated `xDSL` code and execute it in the WebGPU environment. The interpreter is designed to work seamlessly with the translator, taking the output from the translator and executing it on the GPU, ensuring a smooth and efficient compilation process from the `MLIR` code to the GPU execution.

3.5.2 Interpretation Process

The Interpretation Process represents the core stage where the interpreter executes the translated `xDSL` code on the GPU. This process can be further divided into the following sub-stages:

- Buffer Preparation: In this sub-stage, the interpreter prepares a GPUBuffer from the SSA operands using the `buffer_from_operand` method. This method sets up the necessary data structures in the GPU memory for the upcoming operations.
- Argument Bindings Preparation: The interpreter prepares argument bindings for the WebGPU API using the `prepare_bindings` method. These argument bindings are crucial for correctly setting up and executing the GPU operations in the WebGPU environment.

- Function Compilation: The `compile_func` method compiles GPU functions into GPU shader modules. It uses the `WGSLPrinter` to generate the `WGSL` source code for the function, which is then compiled into a GPU shader module ready for execution.

- GPU Allocation Operation: The `run_alloc` function manages the GPU allocation operation (`gpu.alloc`). It allocates a GPUBuffer and returns it as the memref value, setting up the necessary space in the GPU memory for the execution of operations.

- GPU Memory Copy Operation: The `run_memcpy` function implements the GPU memory copy operation (`gpu.memcpy`). It currently supports device-to-host copies, copying the results of GPU computations back to the host memory.

- GPU Kernel Launch Operation: The `run_launch_func` function manages the GPU kernel launch operation (`gpu.launch_func`). It sets up the GPU kernel using the argument bindings and the compiled GPU shader module, and launches it for execution on the GPU.

This structured interpretation process ensures a systematic and efficient execution of the translated `xDSL` code on the GPU, successfully completing the compilation workflow from `MLIR` to executable GPU operations.

3.6 Finalization

As the compilation process approaches its conclusion, it enters the finalization stage. This stage is concerned with wrapping up the translation process and finalizing the interpretation results.

For the translator, finalization primarily involves completing the syntax of the output `WGSL` code to ensure it is well-formed and executable. It closes off any open syntax structures that were initiated during the translation of operations. For example, each function definition (`FuncOp`) opened during the operation processing stage is closed off with a closing bracket, ensuring that all function definitions in the `WGSL` output are syntactically complete. This final touch guarantees that the entire `WGSL` code structure is well-formed and ready for the interpreter.

For the interpreter, the finalization stage involves retrieving the results of the computations performed on the GPU and finalizing any ongoing GPU operations. If any data

was copied from the GPU to the host memory during the interpretation process (using the `run_memcpy` function), this data is collected and processed as needed. Any ongoing GPU operations are also completed, and all allocated GPU resources are properly released to avoid memory leaks.

This finalization stage is integral to the overall compilation process as it ensures the syntactic correctness of the output `WGSL` code and the correct retrieval of the GPU computation results. It represents the successful completion of the compilation process, marking the journey from `MLIR` to `WGSL` and finally to executable GPU operations.

Chapter 4

Evaluation

The introduction of Apple's M1 chip, boasting its integrated GPU, has ignited discussions and curiosity about its performance capabilities. This is especially true when juxtaposed against other renowned GPU technologies. Our evaluation aims to shed light on this topic by focusing on the performance of the M1 GPU when running specific benchmarks.

The benchmarks in question are centered around the use of WebGPU, a web-based graphics and compute API. The efficiency and throughput of applications using WebGPU on the M1 GPU become pivotal, especially when considering the growing demand for web-based graphics applications.

In this chapter, our exploration is twofold. Firstly, we will assess the performance metrics of the M1 GPU by running selected benchmarks using the `wgpu` framework. Secondly, we will evaluate the performance of the WebGPU API, comparing its throughput on the RTX 4080 with results obtained using other technologies. Specifically, we will contrast WebGPU (via xDSL's interpreter) against CUDA (via xDSL+MLIR) and OpenACC (via DeVito). It's essential to clarify that our objective is not to critique other GPUs but to establish a performance baseline for `wgpu` on the M1.

Through this evaluation, we aspire to offer a comprehensive understanding of the M1 GPU's capabilities and the potential of the WebGPU API, enriching the ongoing dialogue surrounding these innovative technologies.

4.1 Evaluation Objectives

Our evaluation is centered around gaining a deep understanding of the `MLIR` to `WGSL` translation process and the prowess of the M1 GPU. With this in mind, we've outlined

two primary objectives:

- Assessing Execution Time on the M1 GPU: Our first goal is to delve into the performance metrics of selected benchmarks when executed on the M1 GPU. This entails running these benchmarks using the `wgpu` framework and subsequently gauging the throughput of the stencil kernels.
- Analyzing WebGPU's Performance: The second objective shifts focus to the performance of the WebGPU API. Here, we'll run our chosen benchmarks on the RTX 4080 using WebGPU. However, the twist lies in the comparison: we'll juxtapose this throughput against results from `xDSL` and `devito`. It's essential to note that our intention isn't to critique the CUDA GPU's performance but rather to establish a performance baseline for `wgpu`.

.

Through these objectives, we aim to provide a comprehensive evaluation of the performance of the M1 GPU and the throughput of WebGpu.

4.2 Evaluation Environment

The systems used for the experiments were:

- 2021 MacBook Pro equipped with Apple's M1 chip, which features an 10-core CPU and 16-core GPU. The device was configured with 16GB of unified RAM.
- Razer Blade 16 powered by an Intel Core i9-13950HX processor. This laptop is equipped with the GeForce RTX 4080 graphics card. The device was configured with 32GB of RAM.

We use modified versions of `xDSL` and `devito` and `xDSL` with webgpu api module. For the performance evaluation of the stencil kernels we use $Gpts/s$(a.k.a $GCells/s$) for throughput.

In our experiments, we focus on two primary benchmarks derived from Computational Fluid Dynamics (CFD) [31][32] and seismic imaging [33]. The first benchmark is the heat diffusion problem, which utilizes a Jacobi-like stencil. This stencil is a mathematical representation used in the discretization of the heat diffusion equation. The second benchmark is based on the isotropic acoustic wave equation, a fundamental equation in seismic imaging that describes how acoustic waves propagate in a medium.

For both of these benchmarks, we conduct simulations in both two-dimensional (2d) and three-dimensional (3d) spaces. To achieve this, we vary the space discretization orders (SDO) [34], specifically using orders of 2, 4, and 8. In the context of 2d simulations, these orders correspond to stencils with 5, 9, and 12 points, respectively. On the other hand, for the 3d simulations, the stencils have 7, 13, and 19 points corresponding to the aforementioned orders.

The size of the problem or the computational domain for these benchmarks is defined by the number of grid points. For the 2d simulations, we use a grid size of 2048 by 2048 points. For the 3d simulations, the grid size is 512 by 512 by 512 points. It's important to note that these sizes are chosen for simulations run on a single computational node.

Lastly, in terms of numerical precision, all our simulations are performed using 32-bit single precision floating point numbers. This choice ensures a balance between computational efficiency and the accuracy of the results.

4.3 Execution Time Evaluation on M1 GPU

In this section, we will focus on the performance of the M1 GPU, specifically when running our selected benchmarks derived from Computational Fluid Dynamics (CFD) and seismic imaging.

1. Benchmark Selection: As previously mentioned, our primary benchmarks are the heat diffusion problem and the isotropic acoustic wave equation. These benchmarks will be simulated in both two-dimensional (2d) and three-dimensional (3d) spaces.

2. Problem Size and Configuration:

   - For the 2d simulations, we will use varying grid sizes of 256, 512, and 1024 points.
   - For the 3d simulations, the grid sizes will be 128, 256, and 512 points.

   All simulations will be run on a single computational node.

3. Execution on M1 GPU: The benchmarks will be executed on the 2021 MacBook Pro equipped with Apple's M1 chip. We will measure the time taken for each execution process, repeating the process multiple times to calculate an average time, ensuring consistency in the results.

4. Data Collection and Analysis: The average execution time for each benchmark will be recorded, along with details about the configuration. This data will be analyzed to determine patterns and trends in the execution time on the M1 GPU.

4.3.1 Findings from Execution Time Evaluation on M1 GPU

In our exploration of the computational performance of Apple's M1 GPU, we focused on two primary benchmarks: 2D and 3D heat diffusion kernels. The performance metric, measured in GPts/s, represents the ability of the GPU to process grid points per second.

Observations for 2D Benchmarks: Upon analysis of the 2D benchmarks, a distinct performance trend emerges, as illustrated in Figure 4.1. As the grid size escalates, the performance in GPts/s correspondingly increases. This amplification indicates a heightened utilization of the M1 GPU for larger problem sizes, which is likely attributable to enhanced parallelism. Delving deeper into the benchmarks, it is observed that among the three variations (5pt, 9pt, and 13pt), the performance discrepancies are relatively marginal, especially pronounced for larger grid sizes. This conveys that the variation in stencil points (spanning 5 to 13) imparts a minimal impact on the GPU's throughput capabilities.

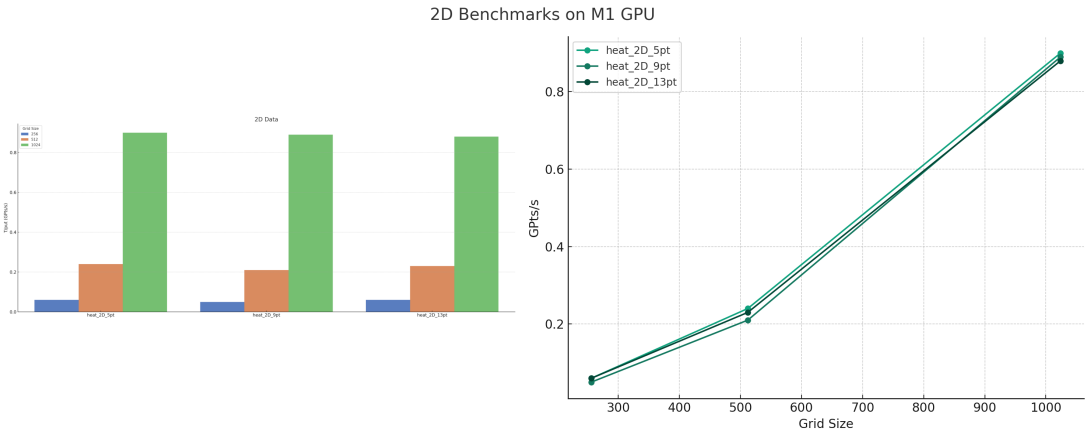


Figure 4.1: 2D Heat diffusion kernels

Observations for 3D Benchmarks: Transitioning to the 3D benchmarks, the narrative remains consistent with the trends observed in 2D, as depicted in Figure 4.2. The performance in GPts/s amplifies with the progression in grid size, underscoring the M1 GPU's adeptness in managing larger problem sizes. However, a divergence in the performance of the three 3D benchmarks (7pt, 13pt, and 19pt) is noticeable. The 7pt

stencil exhibits superior performance compared to its 13pt and 19pt counterparts, with the distinction being especially marked for a grid size of 256. This suggests that the computational intricacy inherent to the stencil calculations significantly influences the observed performance.

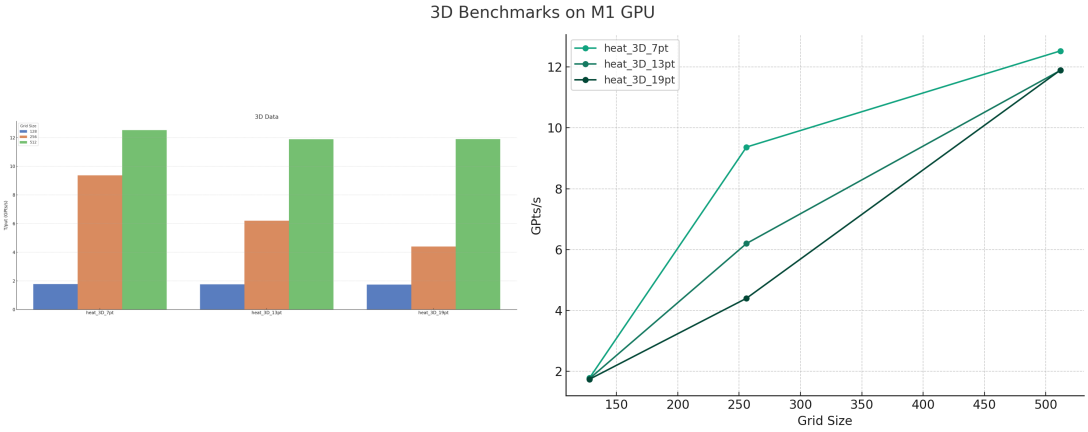


Figure 4.2: 3D Heat diffusion kernels

Overall Evaluation: In synthesizing the findings, several key takeaways emerge. Firstly, the M1 GPU consistently exhibits an escalation in throughput, measured in GPts/s, as the problem size burgeons, spanning both 2D and 3D benchmarks. This trend, evident in Figures 4.1 and 4.2, signals the GPU's prowess in handling intricate computations. Secondly, the choice of stencil, determined by the number of points, distinctly influences performance, with simpler stencils generally resulting in enhanced throughput. Conclusively, for practitioners seeking to harness the M1 GPU's capabilities in real-world applications, judicious selection of grid sizes, coupled with a deep understanding of the computational demands of the specific stencil, is pivotal to achieve optimal performance.

4.4 Execution Time Evaluation on Different GPUs

In this section, we will evaluate the performance of the WebGPU API on the RTX 4080, using the same benchmarks and configurations as described in the previous section.

1. Benchmark Selection: We will continue with the heat diffusion problem and the isotropic acoustic wave equation benchmarks, simulating them in both 2d and 3d spaces.

2. Problem Size and Configuration:

- For the 2d simulations: Heat diffusion problem will use a grid size of 1024 by 1024 points. Isotropic acoustic wave equation will use a grid size of 2048 by 2048 points.
- For the 3d simulations, the grid size will be 512 by 512 by 512 points for both benchmarks.

The timestamp (-n) is fixed at 512 for all cases. All simulations will be run on a single computational node.

3. Execution on RTX 4080: The benchmarks will be executed on the Razer Blade 16 equipped with the GeForce RTX 4080 graphics card. We will measure the time taken for each execution process, repeating the process multiple times to calculate an average time.

4. Data Collection and Analysis: The average execution time for each benchmark will be recorded. This data will be juxtaposed against results from `xDSL` and `devito` to provide a comparative performance analysis of the WebGPU API on the RTX 4080.

Through this methodology, we aim to provide a comprehensive evaluation of the execution time of the `MLIR` examples running on different GPUs.

4.4.1 Findings from Execution Time Evaluation on Different GPUs

In this section, we provide a comprehensive examination of the performance characteristics of the WebGPU API on the RTX 4080, benchmarking against the heat diffusion and isotropic acoustic wave equations. This evaluation juxtaposes the performance of the WebGPU API against that of `xDSL` and `devito` methodologies on the RTX 4080.

Heat Diffusion Benchmark: The combined bar and line charts for the heat diffusion benchmark offer a nuanced perspective on the performance dynamics across the methodologies:

- Both `xDSL` and `devito` manifest closely matched performance dynamics, with `devito` marginally surpassing `xDSL` in some configurations.

- `wgpu`, on the other hand, showcases a pronounced performance dip in the 2D space. However, a transition to the 3D realm sees `wgpu` registering a significant uplift in performance, drawing closer to the `devito` metrics, albeit not reaching the pinnacles set by `xDSL`.

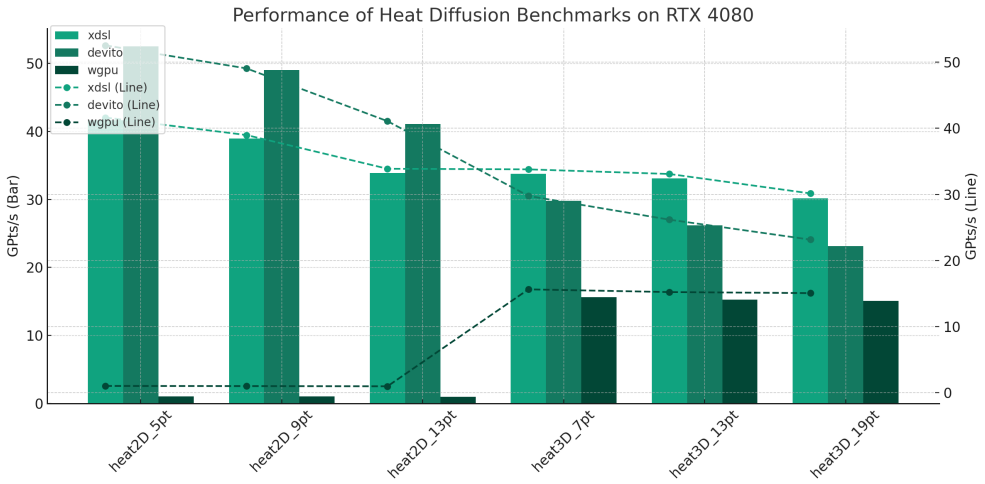


Figure 4.3: Combined performance metrics for heat diffusion benchmarks on RTX 4080

Isotropic Acoustic Wave Benchmark: Transitioning to the isotropic acoustic wave benchmark, the following trends emerge from the visual data:

- As with the previous benchmark, the tussle between `xDSL` and `devito` remains closely contested in the 2D spectrum.
- The `wgpu` continues to lag in 2D computations, but an intriguing shift unfolds in the 3D benchmarks. Here, `xDSL` retains its performance supremacy, but `wgpu` makes a remarkable leap, even eclipsing `devito` in certain configurations.

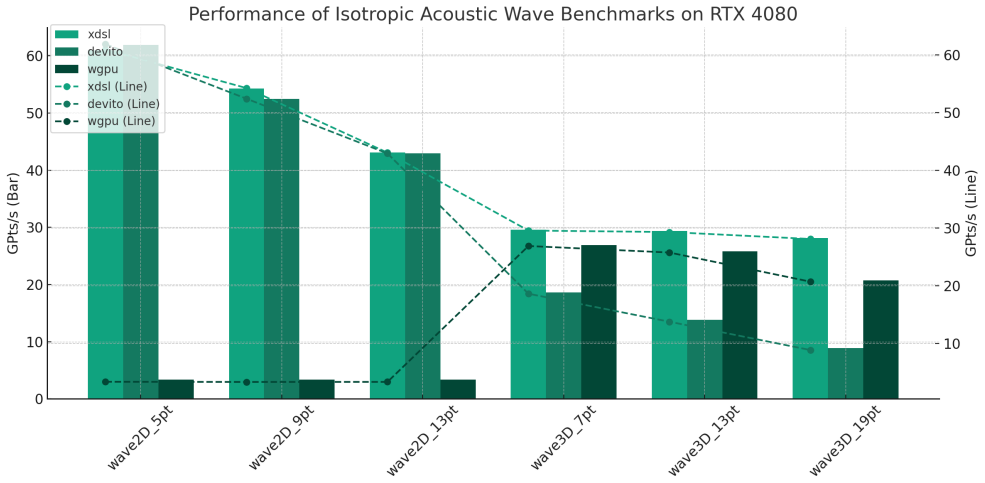


Figure 4.4: Combined performance metrics for isotropic acoustic wave benchmarks on RTX 4080

4.4.1.1 Compare with CPU baseline

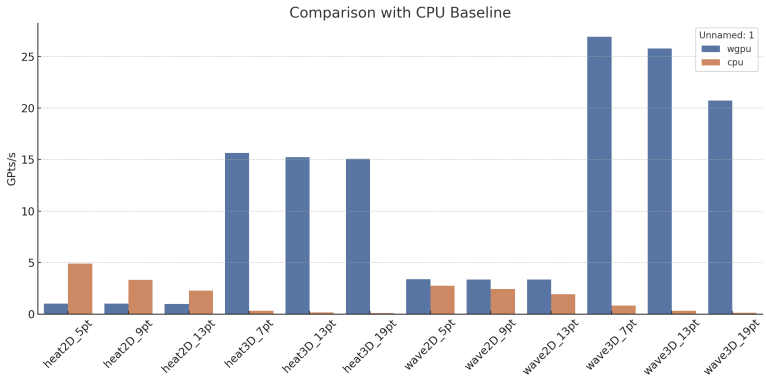


Figure 4.5: Perfermance metrics for wgpu and cpu baseline

The bar chart (Fig 4.5) represents the performance of two benchmarks—the heat diffusion problem and the isotropic acoustic wave equation—simulated in both 2D and 3D spaces on the RTX 4080 using the WebGPU API. Upon comparison with the CPU baseline, it's evident that the WebGPU API performance on the RTX 4080 is consistently lower across all scenarios, with the difference being more pronounced in 2D simulations. While the WebGPU API achieves close to 1 GPts/s in all cases, the CPU baseline demonstrates several times better performance, particularly in the 2D simulations. This suggests that for these specific benchmarks, the WebGPU API on the RTX 4080 may not be as optimized as the CPU baseline.

Overall Evaluation: The evaluation of the benchmarks reveals intriguing performance dynamics between the `xDSL` and `devito` methodologies. Across the varied benchmarks and dimensions, both methods consistently showcase formidable performance metrics. However, nuanced differences occasionally emerge, setting them apart, albeit slightly. On the other hand, the WebGPU API, represented by `wgpu`, presents a different performance trajectory. While it grapples with certain bottlenecks in the 2D computational space, its potential becomes increasingly evident in the 3D domain. This is particularly pronounced in the isotropic acoustic wave benchmark, emphasizing the potential applicability of `wgpu` in specific multi-dimensional computational scenarios. Such evaluations underscore the intricate dance of methodological selection, especially when tailored to distinct benchmarks and problem dimensions. While `xDSL` and `devito` consistently remain versatile contenders, `wgpu` hints at emerging as a compelling choice in certain 3D computational arenas, especially with further optimizations. Collectively, this meticulous evaluation paints a comprehensive picture of the RTX 4080's perfor-

mance dynamics across a gamut of computational paradigms. These insights not only inform but also pave the way for more judicious decisions in the realm of GPU-centric computations.

4.5 Discussion

In the evaluation chapter, Figures 4.1 and 4.2 offer an insightful comparison between 2D and 3D benchmarks. As depicted in Figure 4.1, `WGPU` faces performance challenges in 2D operations. This could be attributed to the fact that `WGPU`, being primarily designed for 3D graphics and compute workloads, might introduce overhead when setting up resources for simpler 2D tasks. This overhead, especially when compared to specialized 2D rendering contexts, might outweigh its benefits for 2D operations. The superior performance of `WGPU` in 3D benchmarks, as showcased in Figure 4.2, reinforces this understanding.

Figures 4.3 and 4.4, delve into specific computational tasks. `WGPU` demonstrates competitive performance in these benchmarks, particularly in wave calculations (Figure 4.4). This suggests that `WGPU`'s strengths are more pronounced in contexts that tap into the parallel processing capabilities of modern GPUs, such as 3D rendering and specific compute tasks.

The discrepancy in performance across different benchmarks further indicates that while `WGPU` is optimized for 3D and complex compute scenarios, there's potential room for improvement in optimizing its performance for traditional 2D operations.

Chapter 5

Conclusions

The realm of GPU technology, particularly in the context of Apple's M1 chip, has witnessed significant advancements and innovations in recent years. This research embarked on a comprehensive exploration of the M1 GPU's capabilities, its relationship with compilers, and the intricacies of translating `MLIR` to `WGSL`. As we conclude this thesis, it's imperative to reflect upon the journey, revisit the objectives set forth, and ponder upon the implications of our findings. This chapter aims to encapsulate the essence of the research, drawing conclusions from the methodologies employed, the challenges encountered, and the insights gained.

5.1 Recap of Research Objectives

In the journey of this research, we embarked on a mission to delve deep into the capabilities of Apple's M1 GPU, particularly its potential to accelerate weather/climate/health simulations. The primary objectives set at the outset were:

- Exploration of Apple's M1 Architecture: To understand the foundational architecture of Apple's M1 chip, its design intricacies, and the innovations that set it apart from other GPU technologies.
- Relationship between GPUs and Compilers: A thorough investigation into how GPUs, particularly the M1, interact with compilers, emphasizing the significance of this relationship in optimizing performance.
- Translation from `MLIR` to `WGSL`: One of the core objectives was to develop a compilation flow within `xDSL` from `MLIR`, a powerful intermediate representation language, to `WGSL` (WebGPU Shading Language). This translation aimed to

harness the computational capabilities of the M1 GPU, bridging the gap between high-level programming constructs and efficient GPU execution.

- Evaluation of M1 GPU's Performance: Through various methodologies and tests, the research aimed to evaluate the performance of the M1 GPU in the context of the translation from `MLIR` to `WGSL`, providing insights into its efficiency and potential areas of improvement.

5.2 Significance of the Study

The endeavor to construct a compilation flow using `xDSL` for translating `MLIR` to `WGSL` and subsequently evaluating the performance of the M1 GPU on executing `WGSL` holds multifaceted implications:

- Advancing Compilation Techniques with `xDSL`: The use of `xDSL` as the foundation for the compilation flow underscores the versatility and potential of this tool. This research not only showcases `xDSL`'s capabilities but also contributes to its broader recognition and adoption in the compiler community.

- Seamless Integration of `MLIR` to `WGSL`: The translation from `MLIR` to `WGSL` represents a significant step in bridging diverse programming paradigms. By achieving this translation, the research facilitates more streamlined GPU programming, potentially simplifying the development process for GPU-centric applications.

- Performance Evaluation of Apple's M1 GPU: At a time when the tech world is keenly observing the capabilities of Apple's M1 chips, this research provides empirical data on its performance, particularly in the context of executing `WGSL`. Such evaluations are crucial for developers, researchers, and industry stakeholders to gauge the real-world potential of the M1 GPU.

- Implications for GPU Programming and Development: The insights from this research can influence the strategies of developers aiming to optimize applications for the M1 architecture. Moreover, understanding the nuances of the M1 GPU's performance can guide future hardware and software innovations, ensuring more efficient and powerful GPU-centric solutions.

In summary, this research stands at the intersection of cutting-edge compiler technology and GPU performance evaluation, offering valuable insights that can shape the trajectory of future developments in both domains.

5.3 Comparison with Existing Literature

The domain of GPU programming, compiler optimizations, and performance evaluations has been a subject of extensive research over the years. However, the introduction of Apple's M1 chip and the evolution of intermediate representation languages like `MLIR` have ushered in new avenues of exploration. This research, centered around the use of `xDSL` to translate `MLIR` to `WGSL` and evaluate the M1 GPU's performance, stands in contrast and complement to existing literature in several ways:

- Novelty of the Compilation Flow: While there are numerous compiler frameworks and tools available, the use of `xDSL` for translating `MLIR` to `WGSL` is a pioneering effort. Existing literature primarily focuses on more established compiler tools, making this research a unique contribution to the field.

- Focus on Apple's M1 GPU: Much of the existing literature on GPU performance evaluation revolves around older or more mainstream GPU architectures. The emphasis on the M1 GPU, a relatively new entrant, provides fresh insights and fills a knowledge gap in the current body of research.

- Integration of `MLIR` and `WGSL`: The translation from `MLIR` to `WGSL` is a nuanced process, and there's limited literature that delves into this specific transformation. This research not only explores this translation but also sheds light on the challenges and intricacies involved.

- Empirical Performance Evaluation: While theoretical discussions and simulations are common in existing literature, this research offers empirical data on the M1 GPU's performance when executing `WGSL`. Such hands-on evaluations provide a more grounded understanding of the GPU's capabilities.

- Bridging Compiler Technology and GPU Execution: Many studies either focus on compiler optimizations or GPU performance in isolation. This research intertwines the two, offering a holistic view of the end-to-end process from code translation to GPU execution.

- Simplified Installation Process with `xDSL`: One of the standout features of this research is the ease of setup. While the existing `xDSL` project for executing code on CUDA demands a more intricate installation process, this project simplifies it dramatically. Users can get started by simply executing pip install `xDSL` and pip

install `wgpu`, making the adoption and utilization of the tool more accessible to a broader audience.

5.4 Limitations and Challenges

Every research journey is marked by its unique set of challenges, and understanding these hurdles is crucial for both the researcher and the audience. In the course of developing the compilation flow using `xDSL` to translate `MLIR` to `WGSL` and evaluating the M1 GPU's performance, several specific challenges emerged:

- Understanding `xDSL`'s Intricacies: Navigating the complexities of `xDSL` proved to be a significant challenge. Grasping its inner workings, especially for someone new to the tool, required a steep learning curve.
- Extraction of SSAValue and Operands from `MLIR`: Delving into the details of `MLIR`, particularly extracting SSAValue and understanding the role and functioning of operands, posed considerable challenges. These intricacies demanded a deep understanding and careful handling to ensure accurate translation.
- Integration into `xDSL` Framework: Initially conceived as a standalone project, integrating the research into the `xDSL` framework presented its own set of challenges. The transition required adherence to stricter coding standards, such as pyright checks, demanding a more meticulous approach to code writing and structure.
- Navigating `WGSL`'s Buffer and Binding: The buffer and binding mechanisms in `WGSL` introduced another layer of complexity. Gaining a comprehensive understanding of how these elements work in `WGSL` was crucial for the successful translation and execution of the code.
- Setting Up the Testing Environment: Establishing a conducive testing environment was not straightforward. The complexities associated with setting up CUDA added to the challenges, requiring careful configuration and troubleshooting to ensure a seamless testing process.

These challenges, while formidable, also provided valuable learning experiences. They highlighted areas of potential improvement and offered insights into the intricacies of compiler development, GPU programming, and performance evaluation.

5.5 Future Work

While this research has made significant strides in understanding the translation of MLIR to WGSL using xDSL and evaluating the performance of Apple's M1 GPU, there are several avenues that future research can explore to build upon this foundation:

- Enhanced xDSL Understanding: Given the complexities faced in understanding xDSL's intricacies, future work could focus on creating comprehensive documentation or tutorials. This would aid new researchers and developers in navigating xDSL more efficiently.
- Optimization Techniques: While the current research laid the groundwork for the translation process, there's potential to delve deeper into optimization techniques. Exploring ways to further optimize the translation from MLIR to WGSL could lead to even better performance on the M1 GPU.
- Advanced WGSL Features: Given the challenges faced in understanding WGSL's buffer and binding mechanisms, future research could delve deeper into other advanced features of WGSL. This would not only enhance the translation process but also provide insights into leveraging WGSL's full potential.

5.6 Final Thoughts

As we draw the curtains on this research journey, it's essential to reflect upon the broader implications and the path it has paved for future explorations. The endeavor to understand the translation of MLIR to WGSL using xDSL and evaluate the performance of Apple's M1 GPU was not just a technical exercise but a testament to the ever-evolving landscape of GPU programming and compiler technologies.

The challenges faced, from navigating the intricacies of xDSL to understanding the nuances of WGSL, underscore the complexities inherent in such research. Yet, they also highlight the immense potential that lies ahead. With every challenge overcome, new doors of opportunity opened, offering insights, learnings, and avenues for innovation.

The significance of this research extends beyond its immediate findings. It serves as a beacon for future researchers, developers, and enthusiasts in the field, illuminating the possibilities and guiding them towards new horizons. The confluence of compiler technologies like xDSL with cutting-edge GPU architectures like Apple's M1 offers a glimpse into the future – a future where programming becomes more streamlined, GPU

performance reaches new pinnacles, and technological innovations continue to reshape the world.

In closing, this research is but a chapter in the vast tome of GPU programming and compiler development. The journey ahead is long, filled with challenges and opportunities, and it beckons to all those curious minds eager to push the boundaries and chart new territories.

Bibliography

[1] Zixuan Zhang. Analysis of the advantages of the m1 cpu and its impact on the future development of apple, 09 2021.

[2] David Kasperek, Pawel Antonowicz, Marek Baranowski, Marta Sokolowska, and Michal Podpora. Comparison of the usability of apple m2 and m1 processors for various machine learning tasks. *Sensors*, 23:5424, 01 2023.

[3] Connor Kenyon and Collin Capano. Apple silicon performance in scientific computing.

[4] Zohaib Ali, Talha Tanveer, Samia Aziz, Muhammad Usman, and Awais Azam. Reassessing the performance of arm vs x86 with recent technological shift of apple, 10 2022.

[5] Giovanni Isotton, Carlo Janna, and Massimo Bernaschi. A gpu-accelerated adaptive fsai preconditioner for massively parallel simulations. *The International Journal of High Performance Computing Applications*, 36:153–166, 05 2021.

[6] Lars Gebraad and Andreas Fichtner. Seamless gpu acceleration for c++ based physics with the metal shading language on apple's m series unified chips, 06 2022.

[7] Giovani Bernardes Vitor, André Körbes, Roberto de Alencar Lotufo, and Janito Vaqueiro Ferreira. Analysis of a step-based watershed algorithm using cuda. *International Journal of Natural Computing Research*, 1:16–28, 10 2010.

[8] J. Delgado. A case study on porting scientific applications to gpu/cuda. *Journal of Computational Interdisciplinary Sciences*, 2, 2011.

[9] Dilpreet Singh and Chandan K Reddy. A survey on platforms for big data analytics. *Journal of Big Data*, 2, 10 2015.

[10] S. Angra and S. Ahuja. Machine learning and its applications: A review, 03 2017.

[11] Massimo Bertolini, Davide Mezzogori, Mattia Neroni, and Francesco Zammori. Machine learning for industrial applications: a comprehensive literature review. *Expert Systems with Applications*, page 114820, 03 2021.

[12] John Goodacre. The evolution of the arm architecture towards big data and the data-centre (abstract only). 11 2013.

[13] Khushi Gupta and Tushar Sharma. Changing trends in computer architecture : A comprehensive analysis of arm and x86 processors. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pages 619–631, 06 2021.

[14] Patrick Cronin, Xing Gao, Haining Wang, and Chase Cotton. An exploration of arm system-level cache and gpu side channels. *Annual Computer Security Applications Conference*, 12 2021.

[15] Nick Brown, Tobias Grosser, Mathieu Fehr, Michel Steuwer, and Paul Kelly. xdsl: A common compiler ecosystem for domain specific languages.

[16] Wang Fuqiu, Weiqiang Zhang, and Jia Liu. Gpu accelerated gmm supervectors for speaker and language recognition. 10 2012.

[17] Chris Lattner. Llvm and clang: Next generation compiler technology llvm: Low level virtual machine, 2008.

[18] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*, 2015.

[19] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. An empirical study of optimization bugs in gcc and llvm. *Journal of Systems and Software*, 174:110884, 04 2021.

[20] S.V. Adams, R.W. Ford, M. Hambley, J.M. Hobson, I. Kavčič, C.M. Maynard, T. Melvin, E.H. Müller, S. Mullerworth, A.R. Porter, M. Rezny, B.J. Shipway, and R. Wong. Lfric: Meeting the challenges of scalability and performance portability in weather and climate models. *Journal of Parallel and Distributed Computing*, 132:383–396, 10 2019.

[21] Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andrew Quinn, and Tyler Sorensen. Mc mutants: Evaluating and improving testing for memory consistency specifications. 01 2023.

[22] http://www.diva-portal.org/smash/record.jsf?pid=diva2

[23] Ben Kenwright. Introduction to the webgpu api. 08 2022.

[24] Abdul Dakkak, Carl M Pearson, and Wen-mei W Hwu. Webgpu: A scalable online development platform for gpu programming courses. 05 2016.

[25] Vaibhav Dalakoti and Diptamon Chakraborty. Apple m1 chip vs intel (x86). *EPRA International Journal of Research and Development (IJRD)*, 7:207–211, 05 2022.

[26] Landon Dyken and Pravin Poudel. Graphwagu: Gpu powered large scale graph layout computation and rendering for the web. *Eurographics Symposium on Parallel Graphics and Visualization*, 06 2022.

[27] Mikhail Khalilov and Alexey Timoveev. Performance analysis of cuda, openacc and openmp programming models on tesla v100 gpu. *Journal of Physics: Conference Series*, 1740:012056, 01 2021.

[28] Mlir. `https://mlir.llvm.org/`.

[29] Webgpu shading language. `https://www.w3.org/TR/WGSL/`.

[30] xDSL project. xdsl. `http://www.xdsl.dev`.

[31] Angela d'Esposito, Paul D Sweeney, Morium Ali, Magdy Saleh, Rajiv Ramasawmy, Thomas M Roberts, Giulia Agliardi, Adrien E Desjardins, Mark F Lythgoe, RB Pedley, Rebecca J Shipley, and Simon Walker-Samuel. Computational fluid dynamics with imaging of cleared tissue and of in vivo perfusion predicts drug uptake and treatment responses in tumours. *Nature Biomedical Engineering*, 2:773–787, 10 2018.

[32] Edward Ferdian, Avan Suinesiaputra, David J Dubowitz, Debbie Zhao, Alan Wang, Brett R Cowan, and Alistair A Young. 4dflownet: Super-resolution 4d flow mri using deep learning and computational fluid dynamics. *Frontiers in Physics*, 8, 05 2020.

[33] Johannes Töger, Matthew J Zahr, Nicolas Aristokleous, Karin Markenroth Bloch, Marcus Carlsson, and Per Olof Persson. Blood flow imaging by optimal matching of computational fluid dynamics to 4d-flow data. *Magnetic Resonance in Medicine*, 84:2231–2245, 04 2020.

[34] Carlile Lavor, Jon Lee, Audrey Lee-St. John, Leo Liberti, Antonio Mucherino, and Maxim Sviridenko. Discretization orders for distance geometry problems. *Optimization Letters*, 6:783–796, 03 2011.