LibraryX: A Framework for Cross-Library-Call Optimization

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Sanil A Rao

B.S., Computer Science, University of Virginia
M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May 2025

Acknowledgments

I truly believe that it takes a village to succeed in completing the Ph.D. As such, I would like to thank those who helped me get to the finish line. First, I would like to thank my thesis committee members - Dr. Scott McMillan, Dr. Jeffrey Vetter, Dr. Tze Meng Low, and my thesis advisor Dr. Franz Franchetti. Your guidance and questions were critical in shaping the story and the quality of the work. Franz, I cannot thank you enough for selecting me to join your research group. The experiences you provided I will truly cherish. Tze Meng, I also want to thank you for challenging my assumptions coming into the program about what I thought I knew about computing. I will be sure to take the principles that we discussed over the years and apply them throughout my career.

I also would like to thank all the friends I made throughout this process. This includes members of the SPIRAL group and members of A level in general past and present. I learned so much from each of you and I know that I could not have been here without you. I also want to thank the members of the FFTX project and my Oak Ridge National Laboratory collaborators on the Bluestone project. I learned so much about what a research career looks like that will be invaluable to me as I progress in my own. I specifically want to thank Dr. Het Mankad for all the help she has provided me over the years. The work we have done together allowed us to travel the world and that would not have been possible without your guidance and feedback. In addition, I would like to thank Upasana Sridhar for always listening to my questions, comments, and concerns about my work and the program. Without you I don't know how I would have ever gotten a thesis. Finally, I want to thank my friends and colleagues Ruben Purdy, Neil Mehta, and

Mckenzie van der Hagen for keeping me sane and making Pittsburgh a fun place to be.

Of course, it goes without saying, but lastly I would like to thank my family. No matter how I felt or what was going on, I know that I could always count on you to help me through it. I am grateful to have been able to spend a portion of my time in the program back home with you, something that will become harder and harder as I get older.

The work in this thesis would not have been possible without the following funding sources listed chronologically during my doctoral tenure:

- Defense Advanced Research Projects Agency: Contract No. HR0011-20-9-0018
- Department of Defense: Contract No. FA8702-15-D-0002
- Joint Intel/VMware Center: Nexus FPGA for Datacenter Servers
- Department of Energy Exascale Computing Project (FFTX): Contract No. 7421006
- Department of Energy XStack (BLUESTONE): Contract No. DE-SC0022275
- Department of Energy SCIDAC: Contract No. DE-SC0023523
- Defense Advanced Research Projects Agency/DUALITY: Agreement No. HR0011-21-9-0003
- Semiconductor Research Corporation JUMP (PRISM): Contract No. 706048
- Department of Energy AI4SCIENCE (DURBAN): Contract No. DE-SC0025645

Abstract

Developing scientific computing applications that are both maintainable and achieve good performance is a challenging task. At the software level, software design principles increase productivity but obfuscate the ability to easily discover performance opportunities. This is exacerbated by the complexity of modern hardware systems, which require deep hardware knowledge to achieve good performance. This leaves application developers with two main options for performance critical operations, use domain specific software libraries to write their applications, or ask a performance expert to optimize their application. The library approach has the benefit of providing usability with good performance, but leaves performance on the table, as there are opportunities to optimize across the library call boundary. Unfortunately, a compiler cannot easily find these because library calls are treated as black boxes. A performance expert can provide the best performance, but has to write specialized code removing the library calls and any usability.

To address the gap between writing productive software and achieving optimized performance, this thesis introduces LibraryX, a framework for cross-library-call optimization. Using LibraryX scientific applications can be written using standard domain specific libraries which will be replaced with an optimized implementation during execution without source code modification. This is done through a combination of library call semantic capture, optimized code generation, and runtime compilation. LibraryX is able to recognize the semantics of library calls or what operation the library call is performing. The computation semantics is then sent to the SPIRAL code generation system for analysis and optimization, producing an optimized implementation. This implementation is then

executed in place of the original library call sequence.

We showcase the high level design of the LibraryX framework, specifically showing how it can be used for a few key domains within scientific computing. These domains include spectral methods, graph analytics/sparse linear algebra, and structured grids. We demonstrate how LibraryX uses various library capture mechanisms for each domain and how the SPIRAL code generation system can optimize specific library call sequences for each domain. This enables LibraryX to cross not only the library call boundary, but also the library domain boundary, allowing developers to use different domain libraries simultaneously. We then showcase how LibraryX can be extended to support multi-accelerator systems by plugging into a runtime system called IRIS. We finally show how LibraryX can be extended to support legacy applications written in Fortran and act as a hardware accelerator offloading system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The thesis of this dissertation is that scientific applications can express performance-critical computations using various domain-specific software libraries, while benefiting from domain expert performance optimizations without source code modification. Traditionally, performance experts are required to identify and implement optimizations that exist across library calls but come at the cost of removing the calls themselves to incorporate those optimizations. This suggests that an automated approach can be developed to automatically detect these library call sequences, optimize them using performance experts' techniques, and execute the optimized implementation in place of the original library call sequence. This enables application developers to leverage well-defined software libraries while abstracting away the complexity of low-level performance software.

1.1 Motivation

Complex high-performance computing systems have made writing performance portable and productive software a significant challenge for scientific application developers. These systems have had complicated cache hierarchies and microarchitectural features such as vector instructions, which require deep knowledge to obtain optimal performance. Furthermore, complexity at the software level through programming language constructs such as objects and portability layers inhibits the ability to automatically discover opti-

mizations. This has led to two options for scientific software developers writing performance-critical computations: write against domain-specific software libraries that are optimized for various hardware platforms, or write a highly optimized implementation of the computation in conjunction with a performance expert. Given the highly specialized nature of the latter approach, the library approach is utilized more often, as it balances software productivity with performance.

In recent years, the library approach has not provided enough benefit for performance-critical computations. This is due in part to the scale of the computation as well as the move to heterogeneous hardware platforms using accelerators. Sequences of library calls require significant memory space, as the input and output buffers must be allocated in full for each library call to execute properly. This incurs significant memory traffic for larger computations. This is exacerbated by accelerators, such as GPUs, which require creating explicit memory copies and performing data transfers along with the idea of a host platform initialized kernel launch to perform the computation on the accelerator. However, if these library calls were not implementations, but instead specifications, optimizations such as algorithm modification, kernel fusion, and memory footprint reduction can be introduced, enabling significant performance improvements. Unfortunately, this requires removing the library calls and writing a manually optimized implementation.

Current techniques cannot adequately address both components of this problem, the expressability of computations (possible through a library abstraction), while providing low-level performance optimizations that break the abstraction. Library developers could provide optimized implementations that take advantage of the optimizations mentioned above. This unfortunately is intractable for every combination of library operator as there are too many combinations. Furthermore, it is still ineffective for computations that require the use of multiple libraries. Optimizing compilers, both general-purpose and domain-specific, are also unable to adequately address this problem. General purpose compilers generally treat library calls as third-party black boxes, not allowing any inter-procedural optimizations. Domain-specific compilers can provide optimizations like library developers and can even do it for multi-domains depending on how much they en-

capsulate. However, utilizing these tools requires domain expert knowledge and replaces the library call implementation similar to a performance expert.

Ideally, there should exist a solution that allows application developers to write single-threaded, standard address space, and single instruction stream programs using domain-specific libraries, and provide optimizations done by performance experts without source code modification. To address this, this thesis introduces the LibraryX framework. LibraryX is able to capture library call sequences, optimize the sequence, and execute the optimized implementation without user modification of the library call sequence. LibraryX achieves this by transforming library calls from implementations to mathematical specifications. This allows a mathematical code generation system, such as the SPIRAL code generation system, to understand library calls and produce optimized variants. Using mathematical software libraries, SPIRAL can optimize library calls from different domains. LibraryX is demonstrated through a combination compile time and runtime approach for libraries taught to they system by domain experts. This thesis shows the LibraryX design, various capture mechanisms, performance optimizations, execution on various hardware platforms, and multi-device execution.

1.2 Background

1.2.1 Programming Models

There are a few programming models which describe how to implement libraries to achieve productivity and performance.

Active Libraries. There have been efforts to automate the optimization of libraries through the use of generative programming [25]. In generative programming, software components are generic specifications and heavily parameterized. Then during program execution, these components are instantiated with a custom, highly optimized implementation leveraging the specific set of parameters passed to that component. *Active Libraries* [91] are software libraries that leverage generative programming for all of the libraries' components. This enables flexibility in optimization through low-level opti-

mizations such as scheduling, and high level optimizations such a loop transformations. Many modern software systems and core language libraries can be classified as *Active Libraries*.

Telescoping Languages. Programming productivity is an important metric when creating and maintaining new software systems. As computing systems become more complicated, software systems become more challenging to program due to new syntax and programming models. Telescoping Languages [49] is an idea to reduce this complexity by allowing developers to write their application in high level domain-specific systems. Underneath this high-level system is an intelligent compiler that exhaustively searches all optimization opportunities, providing good performance. This technique bridges the gap between productivity and performance by allowing simple expression by the developer along with great performance during execution.

LibraryX builds upon the ideas expressed in Active Libraries and Telescoping Languages. LibraryX utilizes the semantics of domain specific libraries as a specification akin to the domain-specific systems in telescoping languages. This transforms any application written against static libraries into generative or active applications. Underneath, LibraryX uses code generation and run-time compilation to replace these library calls with a high-performance variant. This allows for separation of concerns when developing and optimizing applications.

1.2.2 Domain-Specific Libraries and Compilers

Providing optimized code for a specific domain is usually hidden away by a domain-specific library or behind a compiler. Instead, these software systems expose building blocks that make application development much simpler.

Domain-specific Libraries. Domain specific libraries are critical for developing large scale applications. These libraries provide good performance for their operators along with reuse opportunities across the applications' components. Broadly, domain-specific libraries are implemented as a collection of operations and implementations, but new libraries enable developers to interact with library features. The former tend to be C-

style libraries that use the traditional approach due to limitations of the implementation language. Meanwhile, modern libraries such as those written in C++ leverage language features to interoperate with user-provided data structures and user-provided functions, which live outside the library itself.

Compilers. Compilers have long been seen as the mechanism for automated performance of applications. Classical compilers such as LLVM [57] can use a common infrastructure to provide optimization and execution on a wide array of target platforms. The LLVM intermediate representation (IR) is critical to enabling architecture independent code for analysis and optimization. Recent advances in IR such as MLIR [58] build on this representation to provide domain-specific optimizations.

If a general-purpose compiler cannot provide enough optimization, domain-specific compilers can be utilized for the best performance. These compilers have specialized languages to express operations within that domain, similar to domain-specific libraries. These compilers then take that expression and generate optimized code that targets a variety of hardware platforms. They do this by separating the algorithm, how the computation is expressed, from the schedule, the optimizations that can be performed. There are many examples of domain-specific compilers such as [82] [100] [41] [54] [32] [45] [93]. In the domain of machine learning and deep learning, compilers are able to recognize and optimize computational dags at compile time [64]. These compilers [85] [23] can be used as a backend to many popular machine learning frameworks [8] [79], automatically transforming these library sequences into optimized implementations that can be executed on different hardware platforms. Some compilers are even able to translate C programs to specialized accelerators like the FACC [93] (Fourier ACcelerator Compiler) which can map legacy FFT programs to FFT accelerators.

LibraryX explores a wide variety of libraries for optimization in the domains of sparse linear algebra, FFTs, and structured grids. This includes traditional libraries such as [37] [1] [4], as well as modern libraries such as [3] [6]. LibraryX provides optimizations for applications using these libraries regardless of their implementation, and in the case of modern libraries, can utilize their language features. Furthermore, LibraryX is able to op-

timize across the library call boundary a limitation of current compilers. To provide optimization, LibraryX uses the SPIRAL [32] code generation system. SPIRAL uses Operator Language (OL) [31] to express and optimize general linear transforms, enabling multi-domain optimization, rivaling the performance of domain-specific compilers. This optimization process is performed without modification to any of the libraries or the source application. LibraryX takes inspiration from previous work to automate library optimization such as the Broadway [40] compiler. Using domain-expert annotations, Broadway is able to find opportunities to optimize library call sequences through dataflow analysis. LibraryX goes beyond the scope of Broadway by expanding beyond a single library domain.

1.2.3 Programming Paradigms

A number of different approaches exist to capture the computation of a given program at runtime. Once this computation is captured, it can be optimized and replaced by a high-performance variant.

Inspector/Executor. The Inspector/Executor paradigm enables the optimization of a computation by first determining its characteristics and then creating an optimized version that best leverages those characteristics. In the *Inspector* phase, the program is executed as written by the user, but in tracing mode. Tracing mode allows important metadata of the application to be collected. After collection, the relevant computation is updated based on the collected data to improve performance. The program is then reexecuted using the updated computation, which is the *Executor* phase. The Inspector/Executor approach has been used to great effect in the areas of irregular applications [87].

Preprocessor Directives. Preprocessor directives are a way to modify source programs before invoking the compiler, depending on certain information provided by the programmer. Using these directives, programmers can perform textual replacement of operations, provide compile time branches, and turn on or off sections of code. These directives offer increased flexibility for programmers to support portability to different systems and environments. Preprocessor directives can be used at any point in the program, but only

affect lines of code following the directive. The process of using preprocessor directives to redefine library calls is called compile time library interpositioning.

Operator Overloading. Operator overloading is a technique that modifies the meaning of a built-in operator in a given programming language. This kind of polymorphism enables the programmer to support custom operations with common keywords. Take, as an example, the expression $a + b$ where a and b are integer variables. With operator overloading, programmers can change the meaning of add with the $+$ operator to concatenation, creating a new integer with high digits a and low digits b. Operator overloading can be used not only for operations but also for primitive data types such as *int* and *double* as well as classes and objects.

Lazy Evaluation. Lazy or delayed evaluation [44] [59] is a programming language idea of deferring the execution of operations until they are required. This approach has the benefit of not performing unnecessary or duplicate work by only computing on access of the result. This idea is in direct contrast to strict or eager evaluation, which executes an operation as it is invoked. Generally, lazy evaluation is implemented through tags or decorations to certain operators or data holders, and can experience significant performance overhead if not implemented properly.

Runtime Compilation. Runtime compilation is a technique to compile and link external code into an already running program. It is an extension of Just-In-Time (JIT) compilation with the key difference that JIT compilation is seen in interpreted languages rather than in compiled languages. Runtime compilation is used to improve the performance of applications using runtime information and reduce the binary sizes of large libraries. Parallel programming frameworks such as OpenCL [86] rely strictly on runtime compilation to be portable across different hardware devices.

LibraryX utilizes all of these programming paradigms to optimize existing applications. Like the HotSpot JIT compiler [77], LibraryX focuses optimization on critical pieces of a given computation rather than whole program optimization. LibraryX supports Inspector/Executor for applications that are in constrained environments and cannot introduce extra runtime overhead. Otherwise, LibraryX's default model is lazy evaluation,

using this paradigm to capture and translate the computation to the SPIRAL internal language, OL.

1.2.4 Runtime Systems

There have been many runtime systems and compilers for optimizing scientific applications for multi-device heterogeneity and performance portability. Runtime systems are key components for providing support for multi-accelerator heterogeneity because they need to orchestrate different devices and their runtimes. There are runtime systems that support in-node heterogeneity, such as StarPU [10], OpenACC [75], OpenMP [76], and OmpSs [28], while systems such as HPX [47], Charm++ [48], Legion [12], and ParSEC [17] focus on distributed execution. These systems expose abstraction to hide the detail underneath execution and have adopted heterogeneity in their stack. However, IRIS [52] runtime provides heterogeneity support for extreme cases, such as multi-vendor heterogeneity in the same node. Moreover, some of the above mentioned runtime systems also use concepts similar to DAG and task fusion optimization, like ParSEC [17] using DAG fusion for the DPLASMA [16] math library while HPX [47] merges tasks at fine granularity by using task inlining. Examples of other such math libraries built on top of a runtime system are Chameleon [55] and MatRIS [72, 73], however, these libraries rely on other vendor libraries that provide optimized kernels and do not generate the kernels from the same abstractions. Portable abstraction Kokkos [29] exposes high-level constructs that utilize heterogeneity. Recently, there has been work on tightly integrating compilers and runtime systems to get optimized kernels that efficiently map to large distributed systems [95] [96]. These systems have special languages to express not only computations to be performed in their domains, dense and sparse linear algebra, but also constructs for how these computations should be mapped efficiently to distributed nodes from various hardware platforms.

IRISX (the integration of LibraryX and the IRIS runtime system) is different from these systems for a few key reasons. Once compiled, no source code modification is required to make it portable to different multi-accelerator heterogeneous systems, providing pro-

gramming productivity. Some effort in the literature [67, 84] showed the efficacy of such an approach. With the combination of systems like LibraryX and IRIS, IRISX is able not only to obtain architecture optimized kernels, but also it is able to step into the avenue of adapting the kernel, DAG and task representation of that application that works best for the given architecture and domain size, enabling performance portability.

1.2.5 Analysis and Code Generation

SPIRAL [32] is a code generation system that produces optimized C/C++ code for the specific domain of signal processing. SPIRAL is composed of a series of transformation stages that take compositions of mathematical expressions and produce optimized source code for various target platforms. These layers include algorithmic breakdowns, loops and indexing patterns, and abstract code for compiler transformations.

The SPIRAL internal language is called the Signal Processing Language (SPL). SPL follows a point-free matrix vector formulation in which every operator is a matrix with an implicit input vector x and an implicit output vector y. Operators can be composed to create entire algorithmic expressions for a given computation. We describe some of the core operators in SPL starting with the Discrete Fourier Transform (DFT) which is represented in matrix form as

$$y = \mathrm{DFT}_n x, \quad \mathrm{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell < n}, \quad \omega_n = e^{-2\pi i/n}. \tag{1.1}$$

Along with the DFT we represent the $n \times n$ identity matrix as I_n and the stride permutation matrix as L_n^{mn}. The stride permutation matrix rearranges the input elements as $in + j \to jm + i, 0 \le i < m, 0 \le j < n$. If the input vector x is a linearized $n \times m$ matrix stored in row-major order, then L_n^{mn} will perform a matrix transposition on that input.

SPIRAL uses the Kronecker product to build larger expressions with these operators. The Kronecker product of two matrices A and B can be expressed as

$$A \otimes B = [a_{i,j}B], \quad \text{for } A = [a_{i,j}], \tag{1.2}$$

where all entries $a_{i,j}$ of A are replaced by the matrix $a_{i,j}B$. Using the Kronecker product with the identity matrix and DFT yields two expressions, $I_n \otimes \mathrm{DFT}_n$ and $\mathrm{DFT}_n \otimes I_n$. In

the former case, a nxn block diagonal matrix is created with DFT as each block, while in the latter case a higher dimensional matrix is created with each entry having a block diagonal structure using DFT. We can express the popular Cooley-Tukey FFT algorithm using these primitives,

$$\mathrm{DFT}_n = \left(\mathrm{DFT}_m \otimes \mathrm{I}_k\right) T_k^n \left(\mathrm{I}_m \otimes \mathrm{DFT}_k\right) L_m^n, \quad n = mk, \tag{1.3}$$

where T is the twiddle matrix. The Kronecker product can also be used to build higher-dimensional DFTs such as 2D and 3D DFTs. They take the form

$$\mathrm{DFT}_{m\times n} = \mathrm{DFT}_m \otimes \mathrm{DFT}_n, \tag{1.4}$$

$$\mathrm{DFT}_{k\times m\times n} = \mathrm{DFT}_k \otimes \mathrm{DFT}_m \otimes \mathrm{DFT}_n. \tag{1.5}$$

In addition to DFTs, SPL can define other operations like zero-padding, taking an input and embedding it inside a larger input of zeros in all dimensions, and extraction/copy-out, taking a smaller input out of a larger input. We first define the rectangular identity matrix as

$$\mathrm{I}_{n\times N} = \left[\mathrm{I}_n \,|\mathbf{0}\right], \mathbf{0} \in \mathbb{R}^{n\times(N-n)} \quad \text{and} \quad \mathrm{I}_{N\times n} = \mathrm{I}_{n\times N}^{\mathsf{T}}. \tag{1.6}$$

Using the rectangular identity, zero padding and extraction take the form

$$y = \mathrm{I}_{N\times n}\, x \quad \text{and} \quad y = \mathrm{I}_{N\times n}^{\mathsf{T}}\, x, \tag{1.7}$$

where the rectangular identity copies elements from a vector x to the output vector y with zeros everywhere else. Similarly, copy-out/extraction uses the transposed rectangular identity matrix in zero-padding. These transforms can be performed on higher-dimensional inputs using the Kronecker product. We take advantage of these mathematical properties to make implementation decisions that yield better performance on various hardware platforms.

Σ-SPL. Once an SPL expression has been optimized it can be lowered to a Σ-SPL expression. Σ-SPL introduces the concepts of abstract loops and indexing functions. This enables loop-based optimizations previously unavailable in SPL alone. We discuss some of the key operators in Σ-SPL starting with the row and column basis vectors.

$$e_i^{1\times N} = [0, ..., 0, 1, 0, ..., 0] \in \mathbb{R}^{1\times N} \qquad e_i^{N\times 1} = \left[e_i^{1\times N}\right]^{\mathsf{T}} \tag{1.8}$$

These basis vectors can be used to describe the basis Gather and Scatter matricies,

$$\mathrm{G}_f = \sum_{i=0}^{n-1} e_i^n \left(e_{f(i)}^N\right)^\intercal \tag{1.9}$$

$$\mathrm{S}_f = \sum_{i=0}^{n-1} \left(e_{f(i)}^N\right)^\intercal e_i^n \tag{1.10}$$

which are parameterized by an index mapping function

$$f : \mathbb{I}_n \to \mathbb{I}_N \tag{1.11}$$

where $\mathbb{I}_k = \{0, 1, ..., k-1\}$ is the integer interval from 0 to $k-1$. We now define a few basic index mapping functions

$$\imath_n : \mathbb{I}_n \to \mathbb{I}_n,\ i \mapsto i \tag{1.12}$$

$$(j)_n : \mathbb{I}_1 \to \mathbb{I}_n,\ i \mapsto j \tag{1.13}$$

$$\ell_n^{mn} : \mathbb{I}_{mn} \to \mathbb{I}_{mn},\ i \mapsto \left\lfloor \frac{i}{n} \right\rfloor + m(i \bmod m). \tag{1.14}$$

These index mapping functions provide the following properties

$$\mathrm{G}_{\imath_n} = \mathrm{I}_n \qquad \mathrm{S}_{\imath_n} = \mathrm{I}_n \tag{1.15}$$

$$\mathrm{G}_{(j)_m} = e_j^{1\times m} \qquad \mathrm{S}_{(j)_m} = e_j^{m\times 1}. \tag{1.16}$$

The Gather matrix indexes an input vector based on a provided linear mapping function. Similarly, a Scatter matrix writes the output based on a provided linear mapping function. As these functions themselves are linear transforms, we can apply the tensor product to index mapping functions $f : \mathbb{I}_m \to \mathbb{I}_M$ and $g : \mathbb{I}_n \to \mathbb{I}_N$ as

$$f \otimes g : \mathbb{I}_{mn} \to \mathbb{I}_{MN} : i \mapsto Nf\Big(\left\lfloor \frac{i}{n} \right\rfloor\Big) + g\big(i \bmod n\big) \tag{1.17}$$

The tensor product definition gives the following compatibility conditions

$$\mathrm{G}_{f\otimes g} = \mathrm{G}_f \otimes \mathrm{G}_g \tag{1.18}$$

$$\mathrm{G}_{f\otimes g\otimes h} = \mathrm{G}_f \otimes \mathrm{G}_g \otimes \mathrm{G}_h \tag{1.19}$$

$$\mathrm{G}_{i_n\otimes (j)_n} = \mathrm{G}_{i_n} \otimes \mathrm{G}_{(j)_n} \tag{1.20}$$

$$\mathrm{G}_{(j)_n\otimes i_n} = \mathrm{G}_{(j)_n} \otimes \mathrm{G}_{i_n} \tag{1.21}$$

$$\mathrm{G}_{f\circ g} = \mathrm{G}_g \mathrm{G}_f \tag{1.22}$$

and the identity $\mathrm{S}_f = \mathrm{G}_f^{\mathsf{T}}$ for $f, g, h \in \imath_n, (j)_m$. Using these primitives, we can translate an SPL expression into a Σ-SPL expression using the following basic translation rules.

$$\mathrm{I}_m \otimes A_n \rightarrow \sum_{j=0}^{m-1} \mathrm{S}_{(j)_m\otimes \imath_n} A_n \mathrm{G}_{(j)_m\otimes \imath_n} \tag{1.23}$$

$$A_m \otimes \mathrm{I}_n \rightarrow \sum_{j=0}^{n-1} \mathrm{S}_{\imath_m\otimes (j)_n} A_m \mathrm{G}_{\imath_m\otimes (j)_n} \tag{1.24}$$

Additionally, there are some optimization identities that can be applied to Σ -SPL expressions that are listed below.

$$\Big(\sum_{j=0}^{m-1} A_j\Big) B = \Big(\sum_{j=0}^{m-1} A_j B\Big) \tag{1.25}$$

$$B\Big(\sum_{j=0}^{m-1} A_j\Big) = \Big(\sum_{j=0}^{m-1} B A_j\Big) \tag{1.26}$$

$$\mathrm{G}_{(j)_m\otimes \imath_n} \mathrm{L}_m^{mn} \;=\; \mathrm{G}_{\imath_n\otimes (j)_m} \tag{1.27}$$

Beyond Linear Transforms. In recent years, SPIRAL has been expanded to domains outside of FFTs and linear transforms, using another internal language called Operator Language (OL) [31], a superset of SPL. OL introduces multi-linear operators into the SPIRAL system. These operations include scalar/dot product, convolution, filters/stencils, and matrix multiplication. The OL representation of scalar product of two vectors of length n with scalars α and β is,

$$y = ([\alpha, \beta] \otimes \mathrm{I}_n) x, \quad x = \left[\frac{x_1}{x_2}\right], \tag{1.28}$$

where x is a vertical concatenation of the two input vectors x_1 and x_2. Using OL, SPIRAL can generate optimized code for other scientific domains, including graph analytics [83], stencils and structured grids [56] [42] [68], and cryptography [98].

1.3 Contributions

In this dissertation, the goal is to demonstrate an approach that provides a separation of concerns where scientific application developers can express computations using composable operations (library calls) without low-level performance optimizations. By describing computations in this manner, an automated system can reason about performance optimizations, providing an optimized implementation in place of the original library implementation. This work provides the following contributions towards that goal.

- Recognizes that scientific library primitives have semantics that can provide a high-level description of a computation independent of the primitive implementation.
- Develops an approach, LibraryX, that leverages the semantics of library calls by treating them as specifications rather than implementations. Using techniques such as preprocessor library interpositioning, Inspector/Executor, and operator overloading, library calls can produce their semantic meaning for analysis and optimization.
- Library semantics can be taught to the SPIRAL code generation system to discover optimizations through high-level analysis and automatically generate an implementation that has significant performance benefits. This enables whole computation (solver) optimization as opposed to single kernel/primitive optimization.
- Optimizes both single-library and multi-library call sequences, addressing the combinatorial explosion problem of providing optimized primitives for all combinations of library calls.
- Showcases how the interplay between code generation and an intelligent runtime system can finetune application performance on diverse multi-accelerator hardware platforms.

- Demonstrates the efficacy of this approach by applying it to multiple domains within scientific computing including spectral methods, graph analytics/sparse linear algebra, and structured grids, and showed that it can be expanded to support additional front-end languages and accelerators.

1.4 Thesis Outline

Chapter 2 presents the design details of the LibraryX framework. This design shows how library calls are captured, how the code generation system SPIRAL optimizes and produces an equivalent implementation, and finally how the optimized implementation can be executed on various hardware platforms. In addition, a system walkthrough is provided showing each step in detail for the specific example Hockney Freespace Convolution.

Chapter 3 shows a specific instantiation of LibraryX for the graph analytics space, called GBTLX. We introduce the system design for GBLTX and walk through a specific example of how a graph analytics problem, triangle counting, can be optimized using this system. Unlike the general LibraryX framework, GBTLX showcases a different strategy for library capture and recognition, called Inspector/Executor. This showcases how the main components of LibraryX can be specialized for different use cases.

Chapter 4 shows another instantiation of LibraryX, but for structured grid or stencil applications, called ProtoX. The focus in this chapter is to showcase how the LibraryX framework can be utilized by modern libraries that leverage modern software features. Specifically, it discusses the capture mechanisms for user-defined lambda expressions using two different strategies, operation capturing, and callbacks.

Chapter 5 moves beyond LibraryX by introducing IRISX, a tight coupling LibraryX's code generation system SPIRAL with a heterogenous runtime system called IRIS. We show the system design of IRISX and highlight the unique benefits of having a code generator and runtime system communication with each other. Using a structured grid application, we show how IRISX can achieve automatic performance portability from a single source

through the tuned selection of computation kernels and task scheduling.

Chapter 6 discusses expanding the scope of LibraryX by modernizing legacy applications in Fortran and offloading to custom build accelerators. In the case of legacy applications, FortranX takes unmodified Fortran source code for cyclic convolution and replaces it with an IRISX implemented version. For custom built accelerators, LibraryX_ASIC demonstrates how an unmodified FFTW program can execute through LibraryX on a custom built FFT accelerator. Together, these two instantiations demonstrate the adaptability of LibraryX to new frontend languages as well as new hardware platforms.

Chapter 7 presents detailed performance results of LibraryX and its various instantiations. The first set of experiments talk about LibraryX, GBTLX, and ProtoX. It showcases optimization of library call sequences within and across multiple domain specific libraries. The next set of results focuses on IRISX, showing performance portability and scalability for a structured gird application on large supercomputers. Finally, the last set of results shows the expansion of LibraryX to new languages like Fortran and new accelerators.

Chapter 8 presents conclution remarks and future directions.

Chapter 2

LibraryX Design and Example Walkthrough

Domain-specific software libraries have been the de facto standard for achieving performance productivity in scientific computing. These libraries provide key operators that make efficient use of available hardware resources. Using software libraries, application developers can focus on developing new algorithms efficiently, while library developers can focus on the design and performance of library operators. However, this model has opportunities for optimization both at the software and hardware level. At the software level, there may be opportunities to fuse library calls, increasing arithmetic intensity and data reuse, while removing unnecessary temporaries. At the hardware level, there may be more efficient hardware to perform a specific computation, which is inaccessible due to the library against which the application was written. The only way to introduce these optimizations is to manually rewrite the library implementation by hand, removing the library calls and productivity.

The focus of library developers is on providing a set of key operators for a specific domain that provide good performance on hardware platforms [60] [27] [19]. These operators need to support a wide range of inputs and implementations, thereby providing both portability and efficiency. By expanding the scope to include optimized sets of

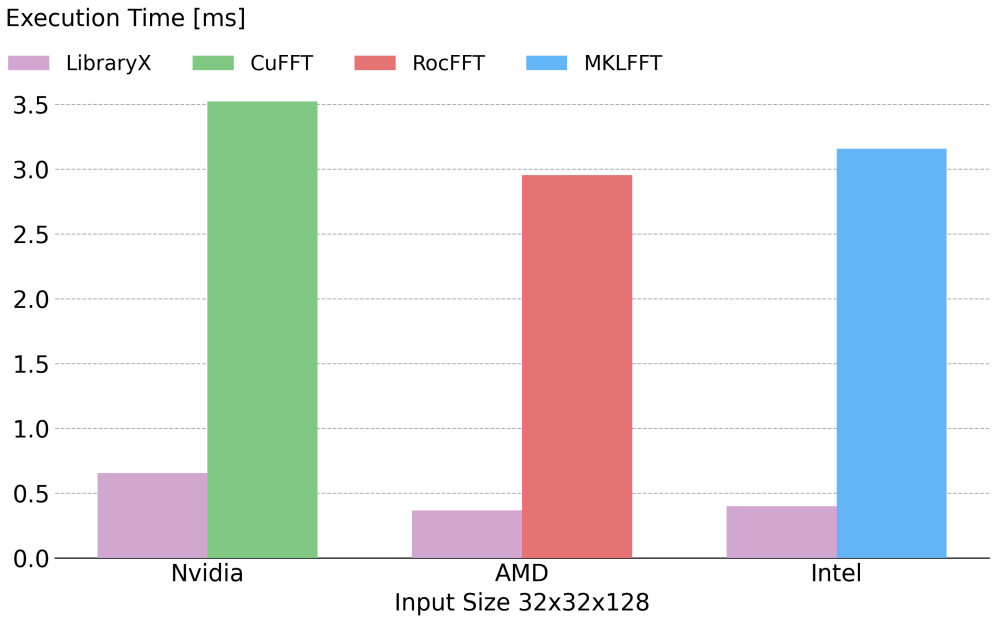
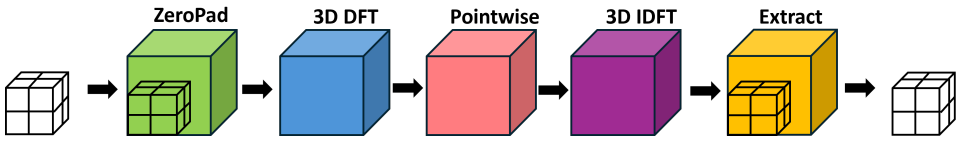


Figure 2.1: Performance comparison between various vendors and LibraryX implementation of Hockney Freespace convolution (lower is better). LibraryX outperforms all vendor implementations with speedups of 5x, 9x, and 8x respectively.

operators, the combinatorial explosion of new operators becomes difficult to maintain. Additionally, executing on various hardware devices requires different implementations for each operator with potentially non-uniform definitions, further burdening the library developer.

Automated approaches such as general purpose [57] [58] and domain-specific compilers [82] [100] [54] [45] [95] [96] could be used to resolve this abstraction breakdown. Unfortunately, each of these falls short in keeping the library implementation view for the application developer while providing the performance of a domain expert. General-purpose compilers are unable to easily cross the library call boundary, often treating them as black boxes. This limits the compilers ability to easily discover opportunities for optimization across library calls. Furthermore, while domain-specific compilers have the necessary performance, they require either learning a new language to extract the correct implementation or lack the interface to be readily utilized like a library. As a result, their implementation needs to be manually introduced in a similar way as a domain expert,



$$y = \left(\mathrm{I}_{k\times K} \otimes \mathrm{I}_{m\times M} \otimes \mathrm{I}_{n\times N}\right)\left(\mathrm{iDFT}_{K\times M\times N}\right)\left(\mathrm{Diag}_{f}\right)\left(\mathrm{DFT}_{K\times M\times N}\right)\left(\mathrm{I}_{K\times k} \otimes \mathrm{I}_{M\times m} \otimes \mathrm{I}_{N\times n}\right)x \tag{2.1}$$

Figure 2.2: Hockney Freespace Convolution illustration and SPL derivation from library calls.

breaking the library productivity model.

To address this challenge of providing a standard interface with opportunities for advanced optimization, this paper presents LibraryX, a framework for optimizing algorithms written with standard library operators. Rather than rewriting algorithms, LibraryX treats library calls as specifications describing the computation to be performed, instead of implementations. LibraryX uses these specifications to create an equivalent representation of the computation. This representation is passed to the SPIRAL code generation system to understand the computation and produce an optimized implementation for various hardware backends. LibraryX then transparently executes the generated implementation in place of the library-based implementation, writing the result to the user's output buffer.

By treating library calls as specifications, LibraryX can optimize library-based algorithms without modification of the source code implementation. With LibraryX, application developers do not have to worry about interoperability of libraries for different domains or different hardware targets. LibraryX can dynamically generate optimized kernels for all recognized operators and redirect execution to any supported hardware platform. This results in significant performance improvements as shown in Fig. 2.1, allowing for a renewed focus on application features rather than performance engineering.

2.1 Background

Large scientific applications rely heavily on software libraries as both a productivity tool and a performance tool. Generally, scientific applications are written as a collection of solvers performing a specific computation with glue logic to communicate information between solvers. Scientific libraries abstract common computation patterns into functions that application developers can leverage to implement their solvers. As solvers can span a wide variety of domains, there are many scientific libraries that tackle a specific set of operations within the space. Thanks to libraries, application developers can focus on library interoperability within a solver as opposed to operator performance.

2.1.1 Main Example: Hockney Freespace Convolution

An important kernel within the scientific community is circular convolution, as it has a wide application in spectral method calculations, PDE calculations, and other numerical solvers. There are two high-level implementations of convolution, direct convolution and FFT-based convolution. Direct convolution has a complexity of $O(n^2)$ when each of the two inputs is of size n. For large inputs, this time complexity is not efficient, so application developers use the FFT-based convolution with complexity $O(n \log n)$. This generally involves three operations that can be easily implemented with software libraries. These operations are a forward DFT on the two inputs, a pointwise multiplication of the forward DFT intermediates, and an inverse DFT to get the convolved output.

The general form of circular convolution can be further specified depending on the application. In the case of PDE solvers, specifically Poisson solvers in Freespace, the Hockney Freespace convolution [43] is used. Unlike circular convolution with periodic boundaries, Hockney Freespace convolution extends circular convolution for unbounded domains. This is done through zero-padding the inputs to account for the infinite domain before the convolution operation and extracting the specific resulting region of interest afterwards. We illustrate the Hockney Freespace Convolution calculation in Fig. 2.2 along with its mathematical specification, discussed in detail in subsequent sections.

2.2 LibraryX Design

We discuss in detail the key stages of LibraryX shown in Fig. 2.3. In the first stage, a library application's computational directed acyclic graph (DAG) is captured. The DAG is then unified into a single high-level description of the computation. This description is then taken through the stages of the SPIRAL code generation system, generating optimized source code. The generated code is then executed on a hardware target provided at build time.

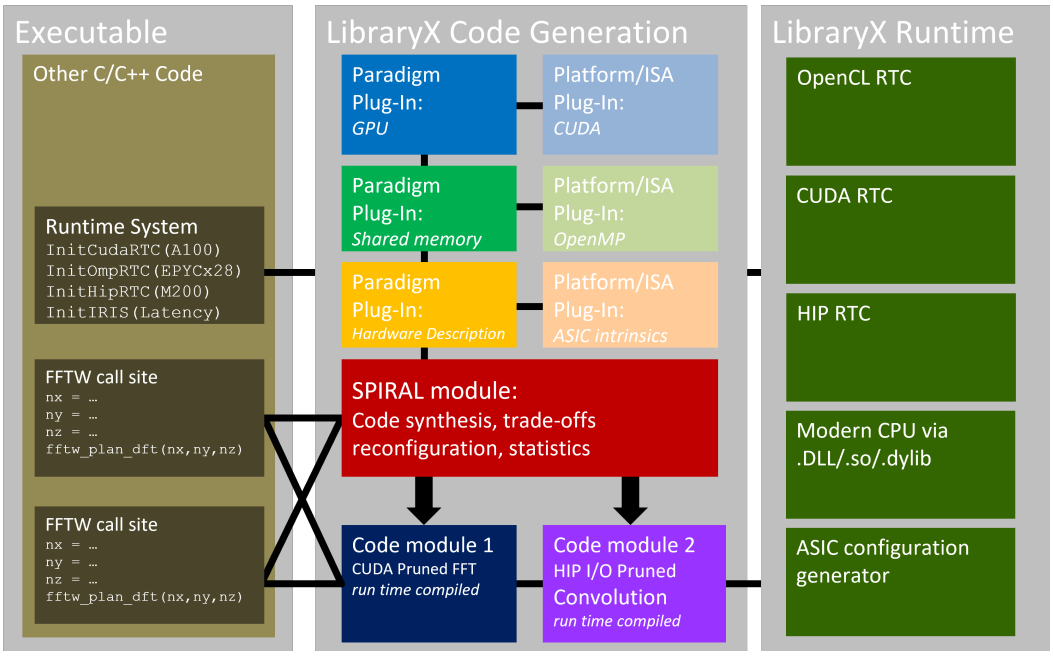


Figure 2.3: The flow of LibraryX from library call capture, to code generation, and finally backend hardware execution. The LibraryX Runtime can be expanded to support any new hardware platform or other runtime systems.

2.2.1 Capturing the Library Frontend

To capture a computation written using libraries without modifying the source implementation, there needs to be a mechanism to recognize and store library call information. LibraryX's top-level header file includes the original libraries header file along with a section of key operators for that library in the `shim`. The `shim` contains LibraryX imple-

mentations of these operators, rather than the base library implementation. At the end of the `shim` are the preprocessor directives to replace the library operator calls with the shimmed calls. This allows LibraryX to support any library-specific structs and utility operations.

At runtime, the LibraryX optimized program calls the shimmed operations instead of the library operations. By controlling the implementation, LibraryX can transform these operators from performing operations to gathering information about the operator. This information includes the function and its parameters, along with the SPL expression for that operation, the SPIRAL equivalent representations of the given operation. After all the operators have been collected LibraryX has created a computation DAG of the program where the inputs and outputs are the edges and the operators are the nodes.

The LibraryX backend is invoked as part of the last library call in a multi-library-call sequence. Here the LibraryX backend object is instantiated and a function pointer for the generated code is declared. If that function pointer has not been populated, LibraryX will generate and compile the function and pass it to the function pointer. This function pointer will then be used with the captured input and outputs to perform the optimized computation. LibraryX can also intercept an output buffer's memory access function through operator overloading, placing the LibraryX backend in the access function directly rather than the final library call.

2.2.2 DAG Unification and Abstraction Lifting

SPIRAL needs to transform the DAG representation, consisting of nodes of the SPL expressions with edges that show how inputs and outputs flow, into a unified SPL expression. During this process, SPIRAL uses its internal database to determine if the incoming DAG matches any known abstractions. If a match is found, SPIRAL will replace the DAG representation with a single abstract expression of the computation. This moves from a DAG representation from the library calls to a dataflow representation in SPL. This SPL expression can then be manipulated with algebraic rules to optimize the expression. Some of these rules are shown in Table 2.1. As part of the unification process, SPIRAL will con-

figure which of these rules should be applied to the unified abstraction. For highly tuned implementations, SPIRAL will also determine the application order of these rules. The final optimized expression will go through SPIRAL's code generation process providing implementations for various hardware platforms. This process is what enables SPIRAL to move from equation 2.1 to 2.2. This process will be shown in detail in Section 2.4.

2.2.3 SPIRAL Optimized Code Generation

After SPL unification and rule configuration, SPIRAL goes through its transformation stages, producing optimized code for a specific target hardware platform provided by the user. Within the SPL layer, SPIRAL has the flexibility to determine the best algorithmic implementation for an SPL expression. Examples include various FFT algorithms like Cooley-Tukey and Rader, various NTT algorithms, and various graph analytics algorithms. These decisions are a derived property of the configured rules, input, and target device captured through the front-end of LibraryX.

After the top-level implementation decisions have been made in SPL, SPIRAL lowers the SPL expression into a Σ-SPL expression. The Σ-SPL layer introduces loops and indexing functions for SPL operators. SPIRAL can perform optimizations such as loop merging and index simplification through an extensive term-rewriting system.

The optimized Σ-SPL expression is then sent through SPIRAL's basic block compiler where traditional compiler operations can be performed. This layer, named internal code, has an abstract representation of traditional source code similar to other compiler intermediate representations. Optimizations performed at this level include dead code elimination, copy propagation, and memory pooling. The final optimized internal code is then taken through a source code parser, which generates source code for the target platform.

In addition to the generated code, SPIRAL provides metadata about the generated code and its parameters. This metadata includes any intermediate memory names, with their types and sizes, kernel names, with their thread geometry and signature, as well as expected number of inputs and outputs with their associated data types. LibraryX's kernel execution backend utilizes this metadata to perform proper setup for hardware

execution.

2.2.4 Runtime Execution Environment (REE)

LibraryX abstracts hardware execution using an object-oriented design, instantiating the appropriate hardware back-end based on flags passed at configure time. These backend implementations vary slightly for CPUs, GPUs, and ASICs. Generally, all backends follow three distinct phases: metadata parsing and setup, runtime compilation, and kernel execution with the user captured input and output. The generated code and fully initialized backend are stored in memory for reuse within the program, with the generated code being cached to disk for future program execution.

The simplest backend is the CPU backend. As the CPU is the host device on modern systems, it is the default memory region of most programs and requires little in terms of setup and execution. The metadata for the CPU backend are the kernel names for initialization, kernel computation, and destruction. The initialization functions handle any temporary memory space creation and constant values, while destruction frees the memory. LibraryX uses dynamic library generation and linking for runtime compilation. Invoking a standard compiler, LibraryX builds a dynamic library file with the generated code and links to it, invoking the functions provided by the metadata.

The GPU backend leverages the runtime compilation APIs provided by the various GPU hardware vendors. As GPU's have distinct memory spaces, LibraryX has to do some additional setup for GPU kernel execution. SPIRAL metadata is initially parsed to gather the number of temporary memory objects to allocate. In addition, the kernel names, their launch parameters, and invocation order are collected. The generated code is then compiled and invoked with the collected parameters, linking against the user's memory objects, and LibraryX created temporaries.

The ASIC backend draws from the techniques utilized in the CPU and GPU backends. As ASICs can have a wide range of execution semantics, we discuss two techniques for ASIC offloading. Some ASICs utilize an intrinsic or programming language extension model similar to that of GPUs for execution. In this case, LibraryX can utilize the same

dynamic library building strategy as the CPU backend. LibraryX queries the system to find the intrinsic header files and bundles them into the runtime generated dynamic library. Other ASICs only support running bytecode or machine-specific code. In this case, LibraryX needs to invoke a separate toolchain to take the generated code, convert it to the desired ASIC representation, execute the code, and then retrieve its results to give back to the LibraryX program. This is achieved through invocation of toolchains via subprocesses.

Hardware Target Redirection. REE enables LibraryX to execute a computation on a target platform different from the system for which the source implementation was written. By converting the user computation into a specification, LibraryX has the ability to use any backend configured by the user. LibraryX only needs to keep track of the memory space in which the user-provided inputs reside. Using the same library call capture idea, LibraryX can capture hardware-specific memory creation calls for a given program. In the captured calls, LibraryX sets flags for the memory space from which the user parameters came from, enabling the movement of memory between distinct spaces transparently. The preprocessor directives used for capturing memory creation calls come after LibraryX's definition. Therefore, LibraryX can perform memory creation and memory copies using standard APIs even though the calls have been changed for the user program.

```
#include <iostream>
#include <complex>
#include <vector>
#include "fftw3.h"
#include "Proto.H"
#include "libraryX.hpp"
using namespace Proto;
BoxData<double,1> zeroPad(BoxData<double 1>& input, std::vector<int> dim ={1,2,3}) {
    BoxData output(Box(Point::Zeros(), Point({dim[0], dim[1], dim[2]})));
    input.copyTo(output);
    return output;
}
BoxData<double,1> extract(BoxData<double 1> large_out, std::vector<int> dim ={1,2,3}) {
    BoxData small(Box(Point::Zeros(), Point({dim[0], dim[1], dim[2]})));
    large_out.copyTo(small);
    return small;
}

int main() {
    int n = 32; int m = 32; int k = 128;
    int N = 64; int M = 64; int K = 256;
    u_int f = FFTW_ESTIMATE;

    BoxData<double,1> input(Box(Point::Zeros(),  Point({n-1,m-1,k-1})));
    BoxData<std::complex<double>,1> input2(Box(Point::Zeros(), Point({N-1,M-1,(K/2+1)-1})));
    BoxData<double, 1> output;

    // will not be materialized
    BoxData<double,1> linput;
    BoxData<double,1> loutput(Box(Point::Zeros(), Point({N-1,M-1, K-1})));
    std::vector<std::complex<double>> temp(N*M*(K/2+1));
    std::vector<std::complex<double>> fout(N*M*(K/2+1));

    buildInput(input);
    buildInput(input2);

    // no-op, just collecting parameters
    linput = zeroPad(input, {N-1,M-1,K-1});

    // no-op, just collecting parameters
    fftw_plan p = fftw_plan_dft_r2c_3d(N, M, K, linput.data(), (fftw_complex*)fout.data(), f);
    fftw_execute(p);

    // no-op, just collecting parameters
    auto complex_multiply = std::multiplies<std::complex<double>>{};
    std::transform(out.begin(), //start location
                   fout.begin(), //end location
                   input2.data(), //2nd input
                   temp.begin(), //output
                   complex_multiply); //operator

    // no-op, just collecting parameters
    fftw_plan p2 = fftw_plan_dft_c2r_3d(N, M, K, (fftw_complex*)temp.data(), loutput.data(), f);
    fftw_execute(p2);

    // no-op, just collecting parameters
    output = extract(loutput, {n-1,m-1,k-1});

    // output now contains the correct result,
    // but temporaries were never materialized
    checkOutput(output);
}
```

Figure 2.4: Source code for Hockney Freespace Convolution. This sequential C++ code is transparently executed on a GPU after it is dynamically translated to CUDA/HIP/OpenCL.

```
//libraryX.hpp
#ifndef LIBRARY_HPP
#define LIBRARY_HPP
BoxData<double,1> captured_in;
double *captured_in2;
BoxData<double,1> captured_out;
std::vector<std::string> script;
libraryx_zeroPad(BoxData<double 1>& input, std::vector<int> dim ={1,2,3}) {
    captured_in = input;
    std::string dims = to_string(dim);
    script.push_back("I("+dims+ "),input");
}
libraryx_fft_r2c_3d(int x, int y, int z, double *input, fftw_complex *output, u_int flag) {
    std::string dims = to_string(x) + "," + to_string(y) + "," + to_string(z);
    script.push_back("DFT(" + dims +"), input, output");
}

libraryx_transform(InputIt first1, InputIt last1, OutputIt d_first, UnaryOp unary_op ) {
    captured_in2 = (double*)get_pointer(last1);
    script.push_back("Diag, last");
}

libraryx_fft_c2r_3d(int x, int y, int z, double *input, fftw_complex *output, u_int flag) {
    std::string dims = to_string(x) + "," + to_string(y) + "," + to_string(z);
    script.push_back("DFT(" + dims +"), input, output");
}

libraryx_extract(BoxData<double 1>& input, std::vector<int> dim ={1,2,3}) {
    captured_out = output;
    std::string dims = to_string(dim);
    script.push_back("I("+ dims + "), output");

    //create LibraryX Obj and generate code
    LibraryXObj lb;
    void (*funcPtr)(double*, double*, double*) = nullptr;
    if(funcPtr == nullptr) {
        funcPtr = lb.compile(script);
    }
    funcPtr(captured_out.data(), captured_in.data(),
            captured_in2);
}

#define zeroPad libraryx_zeroPad
#define fftw_plan_dft_r2c_3d libraryx_fft_r2c_3d
#define fftw_plan_dft_c2r_3d libraryx_fft_c2r_3d
#define fftw_execute libraryx_execute
#define transform libraryx_transform
#define extract libraryx_extract
#endif
```

Figure 2.5: LibraryX header file showing LibraryX shimmed library calls using the C preprocessor and library interpositioning.

```
//spiral_generated_code.cu
void generated_code(double *Y, double *X, double *symbl) {
    ker_hockney0<<<g1, b626>>>(X);
    ker_hockney1<<<g2, b627>>>();
    ker_hockney2<<<g3, b628>>>(symbl);
    ker_hockney3<<<g4, b629>>>();
    ker_hockney4<<<g6, b631>>>(Y);
}
```

Figure 2.6: SPIRAL generated function which contains the GPU kernels to compute the optimized Hockney Freespace Convolution.

2.3 End-to-End Example: Hockney Freespace Convolution

We walk through how LibraryX optimizes Hockney Freespace convolution. We show the following steps:

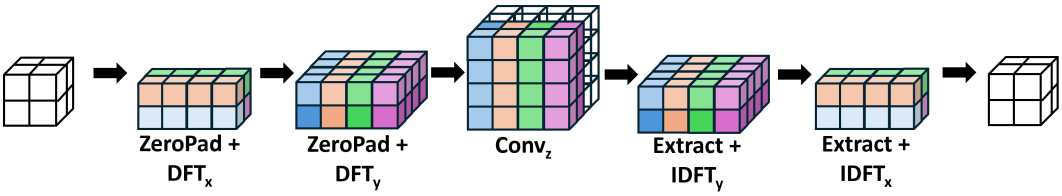
- Capturing library call semantics as a DAG specification.
- Recognizing and lifting DAG specification to high-level abstraction.
- Generating optimized implementations for the abstraction.
- Compiling and executing the generated code at runtime.

This process is done transparently to the user after compiling with a standard toolchain, and can be thought of conceptually as moving from Fig. 2.4 to Fig. 2.5 during program execution using code in Fig. 2.6.

Delayed Execution. To understand the Hockney computation, its semantics must be captured. This is done through LibraryX's lazy evaluation mechanism. Instead of executing a library call, LibraryX captures and transforms that call into a no-op. This allows LibraryX to side-effect the library call to expose its semantics details as SPL and generate a computation DAG of the operation. For Hockney Freespace Convolution this means that a sequence of SPL operators are generated consisting of the zero padding, the FFT, pointwise multiply, inverse FFT, and output extraction. In extraction, LibraryX gets invoked to call the rest of the system. This translation is shown in Table 2.2.

Abstraction Lifting and Code Generation. After a sequence of operations is captured as an SPL DAG its needs to be recognized. We leverage SPIRAL's extensive pattern matching engine to discover if the input SPL DAG is a known pattern. If successful, the DAG will be lifted into a single SPL expression that encapsulates the computation. This expression can then be optimized using algebraic manipulation rules as discussed in Section 2.4. This optimized expression then goes through the SPIRAL code generation system, consisting of algorithm selection, Σ-SPL, internal code, and finally source code for various hardware targets.

Runtime Compilation. After code generation is complete, the generated code needs to be compiled and linked to the running application executing in place of the delayed im-



$$y = \left(\mathrm{I}_{KM} \otimes \mathrm{iPrunedDFT}_N^u\right)\left(\mathrm{I}_K \otimes \mathrm{iPrunedDFT}_M^m \otimes \mathrm{I}_N\right)\left(\mathrm{PrunedCConv}_k^{f_i} \otimes_i \mathrm{I}_{MN}\right)\left(\mathrm{I}_K \otimes \mathrm{PrunedDFT}_M^m \otimes \mathrm{I}_N\right)\left(\mathrm{I}_{KM} \otimes \mathrm{PrunedDFT}_N^u\right)x \tag{2.2}$$

Figure 2.7: Optimized Hockney Freespace Convolution illustration and derivation. LibraryX automates what was done manually as part of a PhD thesis [90].

plementation. This is done through the LibraryX Runtime Execution Environment (REE). The REE parses the generated metadata of the SPIRAL generated code to compile and execute on a given target platform such as CPUs or GPUs. This runs the generated kernels shown in Fig. 2.5 by populating and executing a static function pointer instantiated by LibraryX. When the user then accesses the data from the output buffer, it is populated with the output from the LibraryX kernels.

2.4 Hockney Freespace Convolution Optimization Derivation

Having shown the end-to-end transformation of Hockney Freespace convolution, we now show how SPIRAL automates the process of DAG unification and abstraction lifting.

Given an input $x = \mathbb{R}^{N_x \times N_y \times N_z}$ where $n = N_x$, $m = N_y$ and $k = N_z$, and $N = 2n$, $M = 2m$, and $K = 2k$, Hockney Freespace convolution can be expressed as the point-free composition of five distinct operations shown in SPL as equation 2.1. This equation was semantically captured by LibraryX using Table 2.2. The colors for each operation represent a unique function call in the source application, and the expression is applied from right to left. This SPL composition can be optimized by applying mathematical transformation rules and modifying each expression to provide significant computational benefits. In equation 2.1, the composition of the expressions green and blue shows the zero expansion of the input with the 3D forward DFT. Using identity expansion rules 2.1 and 2.1 in Table

Table 2.1: FFT manipulation formulas using the Kronecker Product. Let A be $n_1 \times m_1$, B be $n_2 \times m_2$, C be $n_3 \times m_3$ and D be $n_4 \times m_4$ matrices. For rules 2.1, 2.1 and 2.1, $m_1 = n_3$ and $m_2 = n_4$.

Property	**Operation**	
Identity Expansion	$\mathrm{I}_{mn} = \mathrm{I}_m \otimes \mathrm{I}_n$	(2.4)
Identity Associativity	$A \otimes B = (A \otimes \mathrm{I}_{n_2})(\mathrm{I}_{m_1} \otimes B)$	(2.5)
Mixed-Product1	$(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$	(2.6)
Mixed-Product2	$(A \otimes \mathrm{I}_{n_2})(\mathrm{I}_{m_1} \otimes B) = (\mathrm{I}_{n_1} \otimes B)(A \otimes \mathrm{I}_{m_2})$	(2.7)
Decomposition	$A \otimes B = (A\,\mathrm{I}_{m_1} \otimes \mathrm{I}_{n_2}\,B) = (\mathrm{I}_{n_1}\,A \otimes B\,\mathrm{I}_{m_2})$	(2.8)
Left Identity Distribution	$\mathrm{I}_n \otimes (AC) = (\mathrm{I}_n \otimes A)(\mathrm{I}_n \otimes C)$	(2.9)
Right Identity Distribution	$(AC) \otimes \mathrm{I}_n = (A \otimes \mathrm{I}_n)(C \otimes \mathrm{I}_n)$	(2.10)
Permuted Commutativity	$A \otimes B = L^{mn}_n (B \otimes A) L^{mn}_n$	(2.11)
Structured Kronecker Product	$A^{m\times m}_i \otimes_i \mathrm{I}_n = L^{mn}_m \left(\bigoplus_{i=0}^{n-1} A^{m\times m}_i\right) L^{mn}_n$	(2.12)
Pruned DFT	$\mathrm{PrunedDFT}^n_N = \mathrm{DFT}_{N\times n}\,\mathrm{I}_{N\times n}$	(2.13)

Table 2.2: SPL formulation of standard library calls used to calculate Hockney Freespace Convolution. Each expression has an input vector x multiplied by a matrix operation and stored in an output vector y.

Library Call	**SPL Formulation**
`y = zeroPad1D(x,n,N)`	$y = \mathrm{I}_{N\times n}\,x$
`y = zeroPad2D(x,{k,m},{K,M})`	$y = \left(\mathrm{I}_{K\times k} \otimes \mathrm{I}_{M\times m}\right)x$
`y = zeroPad3D(x,{k,m,n},{K,M,N})`	$y = \left(\mathrm{I}_{K\times k} \otimes \mathrm{I}_{M\times m} \otimes \mathrm{I}_{N\times n}\right)x$
Forward 3D FFT	$y = \mathrm{DFT}_{k\times m\times n}x$
Element-wise Multiply	$y = \mathrm{Diag}_f x$
Inverse 3D FFT	$y = \mathrm{iDFT}_{k\times m\times n}x$
`y = extract1D(x,N,n)`	$y = \mathrm{I}_{n\times N}\,x$
`y = extract2D(x,{K,M},{k,m})`	$y = \left(\mathrm{I}_{k\times K} \otimes \mathrm{I}_{m\times M}\right)x$
`y = extract3D(x,{K,M,N},{k,m,n})`	$y = \left(\mathrm{I}_{k\times K} \otimes \mathrm{I}_{m\times M} \otimes \mathrm{I}_{n\times N}\right)x$

2.1 this expression can be rewritten as

$$\left(\mathrm{DFT}_K \otimes \mathrm{I}_{MN}\right)\left(\mathrm{I}_K \otimes \mathrm{DFT}_{M\times N}\right)\left(\mathrm{I}_{K\times k} \otimes \mathrm{I}_{MN}\right)\left(\mathrm{I}_k \otimes \mathrm{I}_{M\times m} \otimes \mathrm{I}_{N\times n}\right). \tag{2.3}$$

This expansion breaks the 3D zero expansion and the 3D DFT into a 2D expansion and 2D DFT both in dimensions x and y, and a 1D DFT and a 1D zero expansion in the z dimension. As these expressions are Kronecker products with identity matrices the mixed-product2 rule 2.1 is applied given a reordered expression

$$\left(\mathrm{DFT}_K \otimes \mathrm{I}_{MN}\right)\left(\mathrm{I}_{K\times k} \otimes \mathrm{I}_{MN}\right)\left(\mathrm{I}_K \otimes \mathrm{DFT}_{M\times N}\right)\left(\mathrm{I}_k \otimes \mathrm{I}_{M\times m} \otimes \mathrm{I}_{N\times n}\right). \tag{2.14}$$

This reordering exposes another key optimization through the application of the right

```
fftw_plan planx, plany, planz;

input = zeroPad(in, {N,N,N});

plan_x = fftw_plan_many_dft_r2c(
1, &N, N * N, input, NULL, 1,
N, output, NULL, 1, N/2 + 1,
FFTW_MEASURE);

plan_y = fftw_plan_many_dft(
1, &N, N * (N/2 + 1), output,
NULL, N/2 + 1, 1,
output, NULL, N/2 + 1,
1, FFTW_FORWARD, FFTW_MEASURE);

plan_z = fftw_plan_many_dft(
1, &N, N/2 + 1, output, NULL,
N * (N/2 + 1), 1, output, NULL,
N * (N/2 + 1), 1, FFTW_FORWARD,
FFTW_MEASURE);

fftw_execute(planx);
fftw_execute(plany);
fftw_execute(planz);
```

```
DFTI_DESCRIPTOR_HANDLE desc_x,
                desc_y, desc_z;
input = zeroPad(in, {N,N,N});

DftiCreateDescriptor(&desc_x,
    DFTI_DOUBLE, DFTI_REAL,
    1, N);
DftiCreateDescriptor(&desc_y,
    DFTI_DOUBLE, DFTI_COMPLEX,
    1, N);
DftiCreateDescriptor(&desc_z,
    DFTI_DOUBLE, DFTI_COMPLEX,
    1, N);

DftiSetValue(desc_x,
    DFTI_NUMBER_OF_TRANSFORMS,
    N * N);
DftiSetValue(desc_x,
    DFTI_OUTPUT_DISTANCE,
    N/2 + 1);
DftiSetValue(desc_x,
    DFTI_PLACEMENT,
    DFTI_NOT_INPLACE);
DftiCommitDescriptor(desc_x);
DftiSetValue(desc_x,
    DFTI_INPUT_DISTANCE, N);
 /* ... 20 more lines ...*/
DftiComputeForward(desc_x,
        input, output);
DftiComputeForward(desc_y,
        output, output);
DftiComputeForward(desc_z,
        output, output);
```

```
rocfft_plan plan_x, plan_y, plan_z;

input = zeroPad(in, {N,N,N});

size_t lengths_x[1] = {N};
rocfft_plan_description desc_x,
    desc_y, desc_z;
rocfft_plan_description_create(
    &desc_x);
rocfft_plan_description_create(
    &desc_y);
rocfft_plan_description_create(
    &desc_z);

rocfft_plan_description_set_data_layout(
    desc_x,rocfft_array_type_real,
rocfft_array_type_hermitian_interleaved,
    NULL, NULL, NULL, 1, NULL, 1);

rocfft_plan_create(&plan_x,
    rocfft_placement_notinplace,
    rocfft_transform_type_real_forward,
    rocfft_precision_double,
    1, lengths_x, N * N, desc_x);
/*...20 more lines...*/
rocfft_execute(plan_x,(void**)&input,
    (void**)&output,info);
rocfft_execute(plan_y, (void**)&output,
    nullptr, info);
rocfft_execute(plan_z,(void**)&output,
    nullptr, info);
```

Figure 2.8: Library implementations to perform a zeroPad and 3D FFT as batches of 1D FFTs in each dimension across FFTW/CuFFT, MKL, and RocFFT. This shows the complexity of attempting to do the LibraryX transformations by hand, without opportunities for library-call fusion.

identity rule 2.1 along with the associativity of matrix multiplication, allowing simplification of the four terms into two terms shown below

$$\left(\text{DFT}_K \, \text{I}_{K\times k} \otimes \text{I}_{MN}\right)\left(\text{I}_k \otimes \text{DFT}_{M\times N}(\text{I}_{M\times m} \otimes \text{I}_{N\times n})\right). \tag{2.15}$$

This term shows that the zero expansion and 3D FFT can actually be performed as a batch in the z-dimension of 2D DFTs in the xy-dimensions, called slabs, along with a 1D DFT in the z-dimension, called pencils. We can recursively expand the multiplication of the 2D DFT with identity again using the same rules 2.1, 2.1, 2.1 and 2.1. This gives us

$$\left(\text{DFT}_K \, \text{I}_{K\times k} \otimes \text{I}_{MN}\right)\left(\text{I}_k \otimes (\text{DFT}_M \, \text{I}_{M\times m} \otimes \text{DFT}_N \, \text{I}_{N\times n})\right), \tag{2.16}$$

$$\left(\text{DFT}_K \, \text{I}_{K\times k} \otimes \text{I}_{MN}\right)\left((\text{I}_k \otimes \text{DFT}_M \, \text{I}_{M\times m} \otimes \text{I}_N)(\text{I}_k \otimes \text{DFT}_N \, \text{I}_{N\times n} \otimes \text{I}_M)\right), \tag{2.17}$$

and,

$$\left(\text{DFT}_K \, \text{I}_{K\times k} \otimes \text{I}_{MN}\right)\left((\text{I}_k \otimes \text{DFT}_M \, \text{I}_{M\times m} \otimes \text{I}_N)(\text{I}_{KM} \otimes \text{DFT}_N \, \text{I}_{N\times n})\right). \tag{2.18}$$

The final simplification shows that the expansion of the input for both terms only needs to happen in the x and y dimensions, meaning only the 1D DFTs in x and y need to be computed in expansion while the 1D DFT in z can be computed only on the initial input.

A similar expansion and simplification strategy can be applied to the composition of the expressions in gold and purple, the extraction, and inverse DFT. By applying all the rules shown previously we get the expression,

$$(\mathrm{I}_{KM} \otimes \mathrm{I}_{n\times N}\,\mathrm{iDFT}_N)(\mathrm{I}_K \otimes \mathrm{I}_{m\times M}\,\mathrm{iDFT}_M \otimes \mathrm{I}_N)(\,\mathrm{I}_{k\times K}\,\mathrm{iDFT}_K \otimes \mathrm{I}_{MN}\,) \tag{2.19}$$

In this form, the 3D inverse DFT is decomposed into batches of 1D DFTS in each dimension, and writing the result is done directly in the DFT rather than as a unique step after the DFT operation.

In these new equations for the colors gold, purple, blue, and green every DFT is composed with an identity expansion in its given dimension. This expression can be optimized using rule 2.1, which introduces the pruned DFT [33], consuming the identity matrices. This transforms equations 2.18 and 2.19 as follows

$$(\mathrm{PrunedDFT}^k_K \otimes \mathrm{I}_{MN}\,)(\,\mathrm{I}_K \otimes \mathrm{PrunedDFT}^m_M \otimes \mathrm{I}_N\,)(\,\mathrm{I}_{MN} \otimes \mathrm{PrunedDFT}^n_N), \tag{2.20}$$

$$(\,\mathrm{I}_{KM} \otimes \mathrm{iPrunedDFT}^n_N)(\,\mathrm{I}_K \otimes \mathrm{iPrunedDFT}^m_M \otimes \mathrm{I}_N\,)(\mathrm{iPrunedDFT}^k_K \otimes \mathrm{I}_{MN}\,). \tag{2.21}$$

The new pruned equations 2.20 and 2.21 now express another transformation that involves the red Diag in equation 2.1. When written completely the middle three expressions are

$$(\mathrm{iPrunedDFT}^k_K \otimes \mathrm{I}_{MN}\,)(\mathrm{Diag}_f)(\mathrm{PrunedDFT}^k_K \otimes \mathrm{I}_{MN}\,). \tag{2.22}$$

Through the application of the right identity rule 2.1 and the structured Kronecker Product rule 2.1 the expression can be simplified to be

$$(\mathrm{iPrunedDFT}^k_K \mathrm{Diag}_{f_i} \mathrm{PrunedDFT}^k_K) \otimes_i \mathrm{I}_{MN}\,. \tag{2.23}$$

This exposes the key operation that a DFT followed by a pointwise multiplication followed by an inverse DFT is actually a circular convolution, pruned in this case. Therefore, we can finally simplify this term to be

$$(\mathrm{PrunedCConv}^{f_i}_k \otimes_i \mathrm{I}_{MN}\,). \tag{2.24}$$

Having completed all the transformations, the final expression for the Hockney Freespace convolution is shown in equation 2.2.

Automation of Optimization. Unlike its specification, the LibraryX implementation does not share kernels (colors) with the library-based implementation. This optimized expression breaks the library call abstraction, moving from a 3D forward and inverse DFT into a decomposition in each dimension of batched 1D DFTs. The expression also introduces the pruned DFT due to the composition with the identity matrices. During the transformation process, pruned circular convolution can be introduced, replacing the 1D DFTs in the z-dimension. This results in significant memory savings by only expanding in two of the three dimensions and reducing the number of operations by performing a convolution. These optimizations are the result of extensive study [90] on this computation, which requires significant manual implementation to achieve good performance.

Introducing some of the LibraryX optimizations can be done through the libraries themselves but requires a deep knowledge of each vendor library. Figure 2.8 shows different library implementations of a zeroPad followed by a 3D DFT broken down as 1D DFTs batched in each dimension. For each library, this DFT breakdown can significantly increase the number of lines of code in the implementation and removes any portability between libraries. Even still, these implementations have to work on fully expanded inputs as most libraries do not support pruning.

2.5 Summary

LibraryX is a framework for optimizing scientific applications written against multiple domain-specific performance libraries that implement mathematical functionality. LibraryX demonstrates runtime full-program optimization at the solver level that optimizes across library calls from multiple libraries, and utilizes both semantic knowledge of the library calls as well as performance characteristics and requirements of the targeted hardware. LibraryX is able to recognize and replace sequences (DAGs) of library calls within a solver with an internal representation that captures the higher-level semantics of the

computation. At run-time these DAGs are analyzed by the SPIRAL code generation system, and an optimized implementation for the target hardware is created transparently via run-time compilation and specialization.

The generated highly optimized and specialized implementation replaces the original sequence of library calls behind their API calls, and through lazy evaluation (futures) returns the final result to the user program as the return value of the last library call in the sequence.

Chapter 3

LibraryX for Graph Analytics: GBTLX

Graph algorithms have seen increased interest in recent years for a variety of reasons. Whether this be for biology, cybersecurity, or social network analysis, researching graph algorithms is a very important task in today's computing landscape [13]. This involves understanding, in detail, how graph algorithms perform on a variety of different types of graphs. As a result, many groups are researching ways to improve graph algorithms and processing across the entire system stack from algorithms and frameworks to hardware accelerators for graph applications.

Graph algorithms expressed using a linear algebra formalism [51], as seen through specifications such as the GraphBLAS Application Programming Interface (API) [20, 50] or implementations like the GraphBLAS Template Library (GBTL) [6], provide the benefit that the global behavior of the algorithm is easily understood and allows for optimizations inspired by linear algebra. However, writing graph algorithms with matrices often results in temporaries that are huge but normally would not need to be materialized, as they will, for example, be reduced in a subsequent algorithmic step. Expressing this in such a C/C++ library is challenging, as this leads to a combinatorial explosion in the API and a large, repetitive code base to capture all cases where optimizations are necessary, across all data formats, etc.

To address this issue, we are proposing GBTLX, a system that—to the user—looks like

a C++ class library based on GTBL, but under the hood is a code generation system based on SPIRAL [32, 80, 81]. GBTLX solves the combinatorial explosion problem by analyzing sequences of multiple GBTL calls to find temporaries that need not be materialized, and specializes code for various data formats and instruction sets and other target platform properties. In this paper, we present a first look at GBTLX where the applications are restricted to triangle counting and k-truss enumeration, and we only target multicore CPUs without targeting special instruction sets. We added algorithmic knowledge regarding triangle counting and k-truss to SPIRAL based on previous HPEC Challenge submissions [14, 65]. The resulting performance is on par with the performance reported in these submissions, which shows that GBTLX retains the software abstraction and maintainability of GBTL while providing performance on par with hand-tuned implementations.

3.1 System Overview

We show an end-to-end example of our system. This example highlights the template that will be used for any problem relating to graph processing.

A very common and easily expressible graph algorithm is to count the exact number of triangles present in a given undirected input graph $\mathcal{G}$. Through the language of linear algebra, triangle counting can be formulated as

$$\Delta = ||L \,.\otimes\, (L \,\oplus.\otimes\, L)||$$

where L is the lower triangular portion of the adjacency matrix representation of $\mathcal{G}$, $\oplus.\otimes$ is the semiring used for matrix multiplication, $.\otimes$ is the point-wise multiplication operator, and Δ is the exact number of triangles [78]. This formulation makes triangle counting a great target application for a linear-algebra based library like GBTL. However, while easy to write, the resulting GraphBLAS operations are quite expensive, resulting in poor performance when executed.

We demonstrate the use of GBTLX for the triangle counting problem, showing how to get better performance while writing a linear algebra-based application. We begin with the structure of a GBTLX triangle counting application.

```
1  //tcclass.hpp
2  #include <graphblas/graphblas.hpp>
3  #include "gbtlx.hpp"
4  template <Typename T>
5  void generateGraph(grb::Matrix<T> out, T row, T col) {
6      //create test graph for semantic capture
7  }
8
9  class TCProblem: public GBTLXProblem {
10 public:
11   TCProblem() : GBTLXProblem() {}
12   TCProblem(Signature &sig) : GBTLXProblem(sig) {}
13
14   virtual void randomProblemInstance() {
15     uint64_t *val = new uint64_t;
16     *val = 0;
17
18     // E.g., call external graph generator
19     const unsigned int N(10);
20     auto *L = new grb::Matrix<uint64_t>(N, N);
21     generateGraph<uint64_t>(L, N, N);
22
23     Signature s;
24     s.in.push_back(L);
25     s.out.push_back(val);
26     this->sig = s;
27   }
28 };
29
30 class TCSolver: public GBTLXSolver {
31   public:
32   virtual void semantics(GBTLXProblem &p) {
33     typedef grb::Matrix<uint64_t> MatrixT ;
34
35     MatrixT *inp = any_cast<MatrixT *>(p.sig.in[0]);
36
37     MatrixT B(inp->nrows(), inp->ncols());
38
39     //MatMul with mask
40     // B = L .* (L +.* L)
41     mxm(B, *inp , grb::NoAccumulate(), grb::ArithmeticSemiring<uint64_t>(), *inp, *inp);
42
43     //Perform reduction
44     uint64_t *out = any_cast<uint64_t *>(p.sig.out[0]);
45     reduce(*out, grb::NoAccumulate(), grb::PlusMonoid<uint64_t>(), B);
46   }
47 };
```

Figure 3.1: Structure of the Triangle Counting Problem Specification. In this file are the user created derived classes of the GBTLXProblem and GBTLXSolver shown in Fig. 3.8.

```
1  //tcdriver.cpp
2  #include "tcclass.hpp"
3  grb::Matrix<uint64_t> generateAndFill(std::string const &pathname) {
4     /*assign input data to input objects*/
5  }
6
7  int main(int argc, char **argv) {
8      //create GBTL initial objects
9      //load matrix from file argv[1]
10     grb::Matrix<uint64_t> L(generateAndFill(argv[1]));
11
12     uint64_t val = 0;
13
14     //Pass I/O for the Problem
15     Signature sig;
16     sig.in.push_back(&L);
17     sig.out.push_back(&val);
18
19     //create a Problem
20     TCProblem td(sig);
21
22     //create a Solver
23     TCSolver t;
24
25     //run the Solver on the Problem
26     t.solve(td);
27
28     std::cout << "Number of triangles " << val << std::endl;
29 }
```

Figure 3.2: Structure of the Triangle Counting Application.

3.1.1 User Code

Figures 3.1 and 3.2 illustrate a modified GBTL reference triangle counting application for GBTLX. The first file is the problem specification file. This file includes the header, `gbtlx.hpp`, which contains all the types, macros, and functions necessary to use GBTLX. This file also consists of two derived objects, `TCProblem` and `TCSolver`. In `TCProblem`, the user defines a method `randomProblemInstance`, creating a representative input for their application through a graph generator. This method is called when generating the program's trace file. In addition, `TCProblem` captures the initial input and final output data structures for the application, encapsulated in the `Signature` class. `TCSolver`, contains GBTL operations to count the number of triangles in the given adjacency matrix, defined in the `semantics` method. This method uses the input and output defined in `TCProblem` as parameters to GBTL operations. In this case, the operations are based on the mathematical

formulation described above. Finally, the `HIGHPERFORMANCE` macro, shown in Fig 3.8, allows the make system to link and run the GBTLX generated algorithm.

The second file is the driver application, utilizing the derived objects. The user declares and instantiates the initial input and final output variables, in the `Signature`, as well as `TCProblem` and `TCSolver`. `TCProblem` takes the `Signature` object, binding it internally. `TCSolver` then applies itself on `TCProblem`, using those bound member objects as parameters to the operations in the `semantics` function, thereby executing the application. The method `generateAndFill`, is responsible for instantiating the associated input matrix along the lower triangle.

3.1.2 Interface

The system header file `gbtlx.hpp`, wraps all GBTL operations and defines the base GBTLX classes and abstract member functions. When the `solve` function is called, a computational trace file is generated from the GBTL functions wrapped in `gbtlx.hpp`. This trace file contains a list of input/output data structures and operations performed by the application. In this case, it includes the input matrix and the result, as well as the set of operations performed on that input matrix to calculate the number of triangles. By definition, the operations are a matrix multiplication followed by a reduction. It is important to note that during matrix multiplication, there is a mask of the input matrix L. This allows encapsulation of both the point-wise multiply and the matrix multiplication in a single step. This trace file, seen in Fig. 3.3, will be read as input into our SPIRAL backend for analysis before producing the final binary.

```
1  //trace.txt
2  spiral_session := [
3      rec(op := "triangle_count"), //function name
4      rec(op := "MatrixCreation",row:= 90,col:= 90, ptr := 0x7fffff45bb30),
5      rec(op := "Matrix Multiplication",
6          output = IntHexString("0x7fffff45bb60"),
7          inputA = IntHexString("0x7fffff45bb30"),
8          inputB = IntHexString("0x7fffff45bb30"),
9          mask = IntHexString("0x7fffff45ba30")),
10     rec(op := "reduce(matrix->scalar)",
11         /*many more arguments*/),
12 ];
```

Figure 3.3: Generated Trace file for Triangle Counting.

```
1  //spiral_script.g
2  //load SPIRAL graph package
3  Load(graph);
4  Import(graph);
5
6  //parse trace for operations
7  //perform constraint analysis
8  t := parse("spiral_session");
9
10 //If all constraints met generate code
11 //load Triangle Counting Options
12 opts := TCDefaults;
13 //t is now TriangleCount(param(TInt, "n"));
14 /*www.spiral.net for RuleTree Overview*/
15 rt := RandomRuleTree(t, opts);
16 srt := SumsRuleTree(rt, opts);
17 cs := CodeSums(srt, opts);
18
19 //create output file with generated code
20 //and attaches "spiral_solve" function through opts
21 PrintTo("solve.hpp", PrintCode("generatedFunction", cs, opts));
```

Figure 3.4: Example SPIRAL script for High-Performance Code Generation.

3.1.3 Code Generation

The code generation backend, SPIRAL, utilizes a script file, seen in Fig. 3.4, to generate the high-performance equivalent of the operations in `semantics`. The script reads the trace file and performs constraint analysis on the set of operations performed. For this example, the system needs to determine that the set of operations includes a matrix multiplication masked by the input matrix L, followed by a reduction. It also has to know whether or not the input graph is undirected (i.e. the matrix is symmetric) in order to generate the correct triangle counting algorithm. Finally, the system has to know that the output is a scalar integer. After these constraints have been checked, a high-performance algorithm is generated and the build system will link `solve.hpp` during the compilation of the final high-performance binary, effectively replacing the GBTL operations. Figure 3.5 shows an example of the generated triangle counting algorithm. The generated algorithm takes advantage of an insight where the reduction can be fused with the inner loop of the matrix multiplication and the results of the matrix multiplication do not need to be materialized (the B matrix in line 41 of Fig. 3.1), reducing computation time [65].

```
//solve.hpp

void spiral_solve(GBTLXProblem &p) {
 //logic to parse the problem and call generatedFunction
}

void generatedFunction(uint64_t *res, uint64_t *IJ, uint64_t n) {
  uint64_t t1;
  t1 = 0;
  for (int i1 = 1; i1 < n; i1++) {
    int t2;
    int  *j1, *jm1;
    t2 = 0;
    j1 = (1 + IJ + n + IJ[i1]);
    jm1 = (1 + IJ + n + IJ[(i1 + 1)]);
    while (((((j1 < jm1))) && (((*(j1) < 0))))) {
      j1 = (j1 + 1);
    }
    while (((((j1 < jm1))) && (((*(j1) < i1))))) {
      int i2, t3;
      i2 = *(j1);
      int  *j11, *j1m1, *j21, *j2m1;
      t3 = 0;
      j11 = (1 + IJ + n + IJ[i2]);
      j1m1 = (1 + IJ + n + IJ[(i2 + 1)]);
      j21 = (1 + IJ + n + IJ[i1]);
      j2m1 = (1 + IJ + n + IJ[(i1 + 1)]);
      while (((((j11 < j1m1))) &&
              (((*(j11) < 0))))) {
        j11 = (j11 + 1);
      }
      while (((((j21 < j2m1))) &&
              (((*(j21) < 0))))) {
        j21 = (j21 + 1);
      }
      while (((((((j11 < j1m1))) &&
                 (((j21 < j2m1)))))
                && ((((((*(j11) < i1))) &&
                     (((*(j21) < i1)))))))) {
        if (((*(j11) < *(j21)))) {
          j11 = (j11 + 1);
        } else if (((*(j21) < *(j11)))) {
          j21 = (j21 + 1);
        } else {
          t3 = (t3 + 1);
          j11 = (j11 + 1);
          j21 = (j21 + 1);
        }
      }
      t2 = (t2 + t3);
      j1 = (j1 + 1);
    }
    t1 = (t1 + t2);
  }
  *(res) = t1;
}
```

Figure 3.5: Triangle Counting Algorithm generated by GBTLX. The algorithm is based off prior work [65].

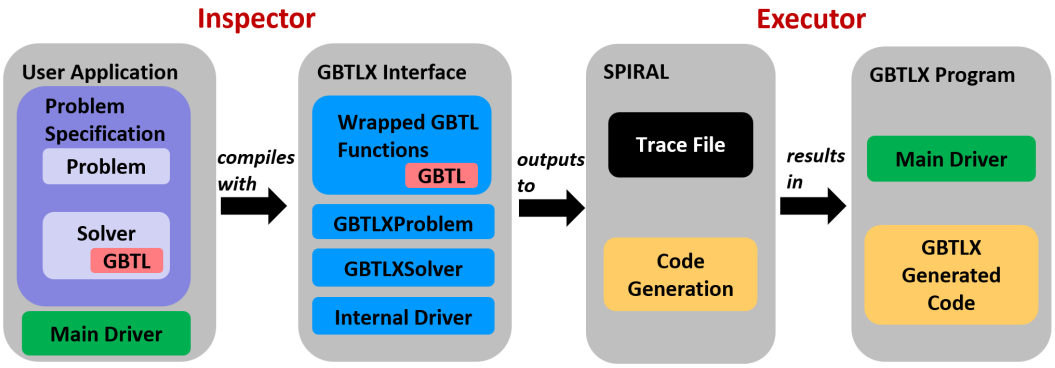


Figure 3.6: System overview of GBTLX from source C++ application to generated high-performance application. An original GBTL program is modified into a GBTLX program. That program is inspected through an interface, generating a trace file for the SPIRAL backend to generate a high-performance algorithm. The first portion of the diagram is the Inspector phase, where a computation is discovered, and the second portion of the diagram is the Executor phase, where an optimized implementation is executed.

3.2 System Walkthrough

GBTLX is designed as a user-triggered inspector/code generator, in which user input is given via Makefile targets. In this system, the user specifically decides what type of output they desire. This could be reference, high-performance, or debug output, with the final binary being created off of this decision. In addition, all GBTLX applications conform to a delayed execution model. In this model, the set of operations that comprise an application is captured and executed such that after the first input is given only the final output is received. There is no inspection of temporaries between operations. This model allows the SPIRAL backend to accurately generate high-performance code. Figure 3.6 illustrates the system overview of GBTLX from user code to generated code.

3.2.1 User Application

The user written application has the same general format. First, the user defines two derived classes, which are referred to as the problem specification. These classes embody the graph problem that is written as seen in the previous example through the `TCProblem` and `TCSolver` objects. These classes are derived from the base classes `GBTLXProblem` and

GBTLXSolver, which are described in a later section.

The user then creates a separate main application file. In the main application, the user declares the derived GBTLXProblem and GBTLXSolver objects. In addition, the user creates an object Signature to encapsulate the initial input and the final output data structures. This is necessary because the system creates mirrored data structures for use in any backend generated functions. As an example, the system would convert an adjacency matrix into a flattened one-dimensional array using a compressed sparse row format. The user then places these data structures in a class called Signature, which is passed into the constructor of GBTLXProblem. Finally, the user applies the GBTLXSolver to the GBTLXProblem, using the member function solve. The main application is written separately from the problem specification because of the trace file discussed in the next section.

3.2.2 GBTLX Interface

The interface, gbtlx.hpp, acts as the translator between GBTL and the SPIRAL backend. All GBTL functions are blocking or synchronous functions; they must return before the application can continue. In order to get GBTL to work within the delayed execution paradigm, the system wraps all of the user facing operations using C macros as seen in Fig 3.8 through OBSERVE. These macros allow the system to intercept GBTL functions without modifying the GBTL library keeping usage the same. The system utilizes these macros to trigger additional functionality depending on the given compile-time flag. As a result, the system has transformed each of the GBTL operations into either blocking or non-blocking functions, depending on the compile-time flag. Concretely, the GBTLX header file, gbtlx.hpp, defines wrapped_<op> functions for every <op> function in GBTL and defines macros like the ones at the bottom of Fig. 3.8 to intercept the GBTL function and replace it with a GBTLX function providing different functionality. This is shown explicitly for the operations mxm and reduce in Figure 3.8.

3.2.3 GBTLXProblem/Solver

In addition to the wrapped functions, the GBTLX base classes, `GBTLXProblem` and `GBTLXSolver`, are implemented in the interface. The `GBLTXProblem` class is the specific instance of a problem the user is trying to solve. Its abstract member function, `randomProblemInstance` is responsible for creating a smaller representative problem used during trace generation. This function's written representation should match characteristics of the original input dataset, like types and shape, and can be an external call to a graph generator. In addition, `GBTLXProblem's` implicit `Signature` captures the initial input and final output for the user application. `Signature` is responsible for holding not only the input and output of the application but also any additional data structures unique to the problem.

`GBTLXProblem's` complement, `GBTLXSolver`, contains the set of operations needed to solve a problem generally. This is captured through `GBTLXSolver's` abstract member function, `semantics`. In `semantics`, the user uses library-defined functions from a framework like GBTL, placing in data structures from `GBTLXProblem` as necessary. `GBTLXSolver`'s `solve` function either executes `semantics`, or is overridden, executing the SPIRAL generated function. Passing `GBTLXProblem` into `solve` allows for reuse of the `GBTLXSolver` on a variety of `GBTLXProblem` classes with different properties. Figure 3.8 shows the the base classes of GBTLX.

3.2.4 High-Performance

There are a few different targets that are available through GBTLX's build system. The most meaningful target is the high-performance target.

The high-performance target leverages the SPIRAL backend to generate a high-performance equivalent of the `GBTLXSolvers`' operations, replacing those operations. To do this, the build system first links the problem specification file together with an internal driver application, used for computational trace generation. The internal driver uses user-modified targets in the Makefile to replace derived GBTLX object names with

generalized object names `USER_PROBLEM` and `USER_SOLVER`. Then the internal driver creates a `randomProblemInstance`, and executes the `GBTLXSolver's semantics` function on that instance. `Semantics` calls the wrapped GBTL operations, which not only execute but also print out information about that function to a trace file. This trace file would contain the operations and the operations' inputs and outputs, including operators and masks. These pieces are important for the SPIRAL backend to accurately determine if optimization is applicable.

The internal driver, seen in Fig. 3.7, is called in place of the user written driver because of potential complexity in user applications. These applications could use large datasets, causing extended computation times or have user unknown exceptions. The internal driver instead creates a representative `GBTLXProblem` instance via the user implemented `randomProblemInstance`, to save execution time. Trace generation does not complete if there are application exceptions at run-time. Once the trace file is generated, the SPIRAL backend is launched, generating the high-performance algorithm and linking it to the final binary, by overriding `solve` with the function defined in `solve.hpp`. All of this is done without the need for the user to delineate which region of their application could be optimized.

```
1  //internal_driver.hpp
2  int main(int argc, char **argv) {
3     //create a Problem, randomInstance, and Solver
4     USER_PROBLEM p;
5     p.randomProblemInstance();
6     USER_SOLVER s;
7     //run Solver semantics
8     s.solve(p);
9  }
```

Figure 3.7: Structure of the GBTLX Internal Driver.

3.2.5 Reference/Debug

The other targets are the reference target and the debug target. In reference, instead of creating an output file and launching the SPIRAL backend, the interface will call GBTL directly to execute the original functions. This path is used for correctness verification and is the default build option. Furthermore, the debug target will avoid delayed execution

by allowing inspection of temporary objects used during computation. This is useful for developers to understand how exactly their application is getting the final output.

```
1  //gbtlx.hpp
2  #ifdef HIGHPERFORMANCE
3  #include "solve.hpp"
4  #endif
5  //GBTLX objects for capture and execution
6  struct Signature {
7      vector<any> in;
8      vector<any> out;
9      vector<any> in_out;
10 };
11
12 class GBTLXProblem {
13 public:
14     GBTLXProblem() {}
15     GBTLXProblem(Signature &Sig) : sig(Sig) {}
16     virtual void randomProblemInstance() = 0;
17     Signature sig;
18 };
19
20 class GBTLXSolver {
21 public:
22     virtual void semantics(GBTLXProblem &p) = 0;
23     void solve(GBTLXProblem &p){
24         #ifdef HIGHPERFORMANCE
25         spiral_solve(p);
26         #else
27         semantics(p);
28         #endif
29     }
30 };
31
32 // mxm operation wrapper
33 template</*many more arguments*/>
34 void wrapped_mxm(CMatrixT       &C,
35                  MaskT const    &Mask,
36                  AccumT          accum,
37                  SemiringT       op,
38                  AMatrixT const &A,
39                  BMatrixT const &B) {
40 #ifdef OBSERVE
41     fprintf(stderr,
42         "rec(op := \"Matrix Multiplication...",
43         &C, &A, &B, &Mask);
44     mxm(C,Mask,accum,op,A,B);
45 #endif
46 #ifdef REFERENCE
47     mxm(C,Mask,accum,op,A,B);
48 #endif
49 }
50
51 // Do something similar for reduce operation
52 template</*many more arguments*/>
53 void wrapped_reduce(/*many more arguments*/){
54     ...
55 }
56
57 //macro to intercept GBTL operations
58 #define mxm wrapped_mxm
59 #define reduce wrapped_reduce
```

Figure 3.8: Abbreviated GBTLX Interface between GBTL and SPIRAL.

3.2.6 SPIRAL Extensions

The open-source SPIRAL backend has been extensively applied to the area of FFTs, and its scope has been broadened to include new application domains. Within the SPIRAL system is a mathematical descriptor language, Operator Language (OL). OL describes the set of mathematical operations that are performed for a computation [35]. This set of operations is then placed in a rewrite system that works in a similar fashion to an optimization problem, resulting in generated code. We add on to this system cursory mathematical formulations for graph algorithms utilizing the existing infrastructure available in SPIRAL. This specifically includes OL objects for triangle counting and ktruss enumeration, which can target different hardware platforms. The details of the SPIRAL system are discussed in the next section.

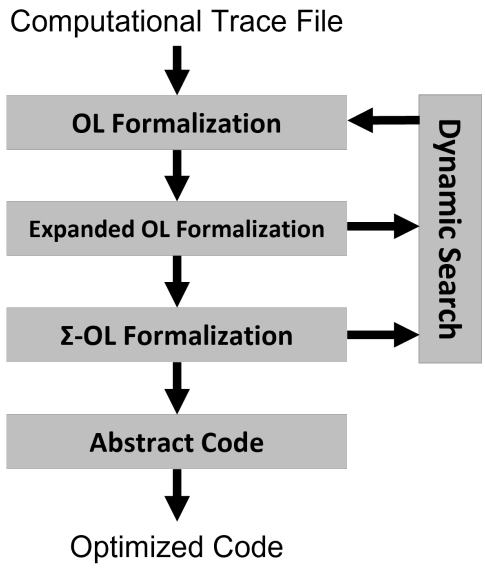


Figure 3.9: SPIRAL architecture, from library description to generated code.

3.3 Code Generation Explored: Triangle Counting

We highlight the process that the SPIRAL code generation system goes through to transform a sequence of GraphBLAS library calls into a domain expert variant for the applica-

tion triangle counting. We show the process for a much lower performing mathematical expression of triangle counting to highlight the benefits of GBTLX. The general outline is shown in Figure 3.9. SPIRAL takes as input the computation trace file, analyzing it and creating a high level description of the algorithms' mathematical representation. SPIRAL then performs a series of search's across a wide optimization space before returning domain expert C code. For a mapping of SPIRAL internal language to mathematics please see Tables 3.1 and 3.2.

An easily expressible graph application is counting the exact number of triangles in a given undirected graph $\mathcal{G}$. A mathematical specification for counting these triangles is given as

$$\Delta = \frac{1}{6}\Gamma(A^3). \tag{3.1}$$

where A is a symmetric undirected adjacency matrix representing the input graph $\mathcal{G}$ [21] and Γ is the trace operation, or the sum of the elements along the main diagonal. While Equation 3.1 is a precise definition it does not achieve great performance due to the extra computations being performed in the matrix multiplication.

Previous work [65, 66], recognizes that there are more efficient ways to compute the number of triangles without the need to perform multiple matrix multiplications. SPIRAL understands how to translate an application written using Equation 3.1 in GraphBLAS to a domain expert variant.

OL

In the first stage, SPIRAL reads as input a computational trace file highlighting all major operations and data structures used in the original application. This would be an input matrix and a output variable, followed by two matrix multiplication calls and a reduction call (trace) divided by six. SPIRAL then has to move these expressions into a directed acyclic graph (DAG) with each operation being a node in the graph and each edge having the input or output of that node. Additionally, each node contains carrier information, or properties, about that operation such as matrix symmetry, density/sparsity, as well as data masks and semirings.

DAGs in SPIRAL are useful for pattern recognition and matching. Many times, there is no one-size-fits-all algorithm for a specific input. DAGs allow SPIRAL to accurately determine which algorithm best fits the input graphs' constraints. In this instance, SPIRAL first recognizes that the operations being performed correspond to a triangle counting program. This done via a table lookup into known triangle counting mathematical formalization's.

Once SPIRAL has understood the computation that is being performed, an internal representation needs to be generated. This representation describes the computation and synthesizes additional carrier information. The result is an initial SPIRAL OL formalization that describes the triangle counting computation. In this example, we will say that we have a sparse undirected symmetric graph of integers. The corresponding expression after recognition is shown below.

```
TriCount(TSparse_Mat(TSemiring_Arithmetic(TInt)))
```

OL expressions like this could not be generated previously as there was no concept of sparsity within SPIRAL. In many cases graph algorithms use sparse data formats for their inputs and outputs because the graph datasets have a limited number of non-zero elements. As such, compressed representations are used to decrease the memory footprint of the application, performing operations on only those non-zero elements. By introducing sparse data formats in SPIRAL, we now have a way to generate sparse algorithms properly.

Expanded OL

Having the internal representation, SPIRAL then performs a series of transformations to obtain the final domain expert kernel. The first transformation calls a `RuleTree` on the expression `TriCount`. The goal of the `RuleTree` is to determine which of the known algorithms best matches the carrier information expressed in `TriCount`. This then transforms TriCount into an expanded OL expression showing specific details of the selected algorithm. This is written both mathematically and in the code below.

$$\Delta = \Delta + a_n^T A_n b_n \tag{3.2}$$

```
ts := TSparseArray(TInt, TSemiring_Arithmetic(TInt));
tsm := TSparse_Matrix(ts, "col");
rv := RowVec(tsm);
```

Equation 3.2 describes a sequence of multiplications in which the first multiplication filters a column of matrix A and then multiplies that column by a row of matrix A, thus finding a triangle. This can be further optimized if the graph is symmetric using the lower triangular matrix L to avoid overcounting. In code we represent this by a sparse array that is promoted to a sparse matrix, marked with `col` to represent the orientation. This is then encapsulated in a `RowVec` to perform the multiplication against the rows. This and other such expressions are based on prior work [65].

Σ-OL

After creating the new expression, the next transformation stage, the Σ-OL formalization [36], is called via `SumsRuleTree`. This transformation stage inserts loop expressions and access pattern information for given inputs. SPIRAL organizes memory linearly, which determines the proper structuring of the loops and strides. The Σ-OL expression for triangle counting is shown below.

$$\Delta = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} S_{(j)_i \otimes (j)_j} \circ \left(I_n \otimes (.)^{1\times 1 \to 1} \right) \circ G_{(j)_i \otimes (j)_j} \tag{3.3}$$

```
ISum(i, n,
  ISum(j, m,
    Scat(fTensor(fBase(i), fBase(j)))
    * RowVec(tsm)
     * Gath(fTensor(fBase(i), fBase(j)))))
```

Equation 3.3 begins with a loop over all rows n by the outermost `ISum` followed by a loop over all elements in a row of length m. Within the second loop, we start with the `Gath` which holds the access pattern for gathering the nonzero elements of the row

and column vector to be compared (sets of edges per vertex), followed by a `RowVec` to intersect(multiply) them and determine if a triangle exists. This is completed with a `Scat`, to accumulate the intermediate result into a scalar variable. Furthermore, `fBase` is an access function in a given dimension, and `fTensor` tensors access functions together for multi-dimensional access.

One of the issues at the Σ-OL stage is that the generated expression is still a bit too generic for the problem we want to solve, triangle counting. The `Scat` is writing out the result based on the variables `i` and `j`, based on the matrix within the `RowVec`. However, triangle counting wants accumulation to be applied to a single result scalar for all row/-column multiplications. To solve this, SPIRAL uses rewrite rules, expression modifiers triggered by certain patterns of expressions. SPIRAL has an established rewrite rule that it can apply here to change the `Scat` to solve this issue. The new expression is shown in 3.4.

$$\Delta = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} S_{i_1} \circ \left(I_n \otimes (.)^{1\times 1 \to 1} \right) \circ G_{(j)_i \otimes (j)_j} \tag{3.4}$$

```
UnifyKernel(ISum(i, n,
  ISum(j, m,
    Scat(fId(1))
    * RowVec(tsm)
     * Gath(fTensor(fId(i), fBase(j))))))
```

In addition to changing the `Scat`, SPIRAL also added a decorater object called `UnifyKernel`. This decorator object is used during code generation to further assist the system in proper code layout. In this specific case, the decorator is used to unify the `Scat` and `Gath` within the actual multiplication operation defined by the `RowVec` object. The decorator will move the indexing into the variables that `RowVec` will generate. Decorators can also be used to add preamble/epilogue code to a kernel, such as algorithm-specific data structures and data cleanup. In most cases these decoraters are added within the rewrite framework, only becoming visible as SPIRAL traverses the ruletrees, but are shown directly for clarity.

Optimized code

The final transformation converts the previous expression to intermediary code called `icode`, which then becomes C code. In the `icode` stage, classical compiler optimizations are performed, such as dead code elimination, copy propagation, and array scalarization. This is then translated into C code, as seen in Fig. 3.5 which is linked back to the original application to produce the final output binary.

3.4 Hardware Backends and Parallelization: Triangle Counting

In addition to generating pure sequential code, GBTLX can also generate parallel implementations for some graph algorithms. GBTLX can target CPUs and GPUs for code generation and use a variety of different programming models on those platforms, such as OpenMP, CILK, pthreads on CPU and CUDA, HIP on GPU. This is enabled through different code generation backends available within SPIRAL [34]. The traditional `DefaultCodegen`, parses SPIRAL's intermediate representation into C Code. Other code generation objects like `OpenMPCodegen` and `CUDACodegen` parse for their own specific hardware backend.

These code generation objects are actually quite simple to implement within SPIRAL thanks to inheritance. Take for example a OpenMP pragma based parallel programming model. The SPIRAL `OpenMPCodegen` object will inherit from the `DefaultCodegen` object taking with it all the fields, some previously shown. Now, it can introduce new objects like parallel loops and barriers as new fields in its class. This will not affect the default object and allows reuse of core default OL expressions like the ones above. It can even go on to modify some of those objects without interfering with the default behavior. We go on to show how GBTLX can easily take current graph expressions and make them parallel.

CPU

We begin on the CPU side, showing how a parallel Triangle Counting implementation can be generated using the expressions seen previously.

```
tsm := TSparse_Mat(TSemiring_Arithmetic(TInt))
TriCount(tsm).withTags([AParSMP(4)])
```

The OL expression shown mirrors that of the previous expression, but with a new modifier, `withTags`. This modifier tells the SPIRAL system to break down the expression with a new set of OL objects based on the parameter provided. In this case, the parameter `AParSMP(4)` says to use the OpenMP code generator object with four threads. This will then trigger the options to point to the new code generator object. From here we get a similar expanded OL expression, but a new Σ-OL expression.

$$\Delta = \underbrace{\sum_{i=0}^{n-1}}_{\substack{smp(4),\\ reduction}} \sum_{j=0}^{m-1} S_{t_1} \circ \left(I_n \otimes (.)^{1\times 1\to 1}\right) \circ G_{(j)_i \otimes (j)_j} \tag{3.5}$$

The new Σ-OL expression seen in Equation 3.5 has many similarities with the previous expression except for the outermost loop. Instead of a traditional loop, there is now a parallel accumulation loop. During parsing, this loop will generate alongside a parallel for pragma with a reduction clause on the implicit output `Y` using four threads. Its also important to highlight that just like in the previous example this Σ-OL expression is not complete for triangle counting. SPIRAL again utilizes the rewrite rule system to get the optimized triangle counting Σ-OL expression, now with the new `SMP_Accum` object.

GPU

GPU parallelization within GBTLX shares many similarities with CPU parallelization from an expression standpoint. The main differences lie in how SPIRAL recognizes GPU architectural features.

```
tsm := TSparse_Mat(TSemiring_Arithmetic(TInt))
TriCount(tsm).withTags(
[ASIMTKernelFlag(ASIMTGridDimX(80)),
ASIMTBLockDimX(256)])
```

Our OL expression for Triangle Counting remains unchanged for code generation on GPUs, but now has a new GPU tag, `ASIMTKernelFlag`. SPIRAL supports generalized SIMT

allowing for code generation for any SIMT architecture; the generated kernel will be unparsed using the domain language such as CUDA [7]. Within this tag is a reference to `ASIMTGridDimX`, as GPUs have a three dimensional thread geometry in the form (x, y, z). For our triangle counting example, we only use the X dimension.

Algorithmically, the implementation of triangle count differs on the GPU when compared to the CPU due to the massively parallel architecture of the GPU. Instead of having an outermost loop on the number of rows n, the implementation now loops on all elements of the column, m. This better saturates the GPU compute units as there are more elements in all the rows than there are numbers of rows. The computation is partitioned so that each thread on the GPU gets a neighborhood based on m and another neighborhood from the row n that m belongs to [14]. These neighborhoods are intersected to discover triangles. For this example, the SPIRAL representation remains mostly intact, only changing in loop structure and indexing.

$$\Delta = \underbrace{\sum_{j=0}^{m-1}}_{\substack{grid(80),\\ block(256)}} \underbrace{S_{t_1}}_{\substack{reduce(grid)\\ reduce(block)}} \circ \left(I_n \otimes (.)^{1\times 1\to 1}\right) \circ G_{(j)_j \otimes (j)_i} \tag{3.6}$$

The `TriCount` expression is broken down into its Σ-OL expression, shown in Equation 3.6, placing the GPU tags in the appropriate locations. During the breakdown, the tag is passed to each of the individual components that make up the expression. The outermost loop of our computation holds the dimension tag of our SIMT architecture. This tag is broken down into the grid and block in order to allow SPIRAL to map threads to all cores of the GPU. The core of the computation, `RowVec`, remains unchanged while the `Gath` has an updated indexing function to match the loop interchange. The `UnifyKernel` will update the indexing for the input graph data format.

Unlike the OpenMP implementation, there is not a way to gather all the intermediate results across the GPU with just a loop-level expression. Therefore, in the breakdown, the `Scat` is tagged with additional reduction statements. The first reduction is at the local level, reducing intermediate results of a thread group into a single variable based on

Table 3.1: OL Object primitives with mathematical meaning. A complete formalization can be found in previous work [36] [32]. Table 3.2 shows examples of functions for f used in the Gath, Scat and Diag primitives.

Primitive	Math
Gath	$G_f^{n\to N} : \sum_{i=0}^{n-1} [e_i^{n\times 1} e_{f(i)}^{1\times n}]^\top$
Scat	$S_f^{n\to N} : \sum_{i=0}^{n-1} e_i^{n\times 1} e_{f(i)}^{1\times n}$
Diag	$D_f^{n\to\mathbb{C}} : \mathbb{I}_n \to \mathbb{C}$
RowVec	$\mathbb{1}_n^T \circ (I_n \otimes (.)^{1\times 1\to 1})$
COND	$g(x) \mapsto \begin{cases} OL_1, & \text{if } g(x) \\ OL_2, & otherwise \end{cases}$
IterVStack	$[-]_{i=0}^{n-1} A_i = \begin{bmatrix} A_0 \\ A_1 \\ \dots \\ A_{n-1} \end{bmatrix}$
IterHStack	$[\|]_{i=0}^{n-1} A_i = ([-]_{i=0}^{n-1} A_i)^\top$

the previous expression's output. The second reduction is at the group level, adding all the group level results to the final implicit output, Y. These reductions have synchronization/atomic points that will be included within their respective stages.

During code generation, a sequence of passes are performed in order to place in the final architectural features present on GPUs. This includes marking arrays as global arrays versus shared arrays, setting up specific memory properties such as cache sizes and partitions, and setting up host and device memory copies. During these passes, the variable `tsm` becomes a shared memory array with a unique location per thread on each core to prevent any data races. Finally, the generated code will contain both the computational kernel and a function that wraps the kernel launch. The kernel will be launched with the values present in `ASIMTGridDimX` and `ASIMTBlockDimX`.

3.5 Towards a generalized Graph Framework

Triangle counting serves as a simple example to illustrate the different functions and operators used by GBTLX to generate graph processing applications. Table 3.1 highlights the subset of functions and operators that needed to be updated with the SPIRAL code

Table 3.2: OL Function primitives with mathematical meaning. A complete formalization can be found in previous work [36] [32].

Primitive	Math
fId(i)	$\imath_n : \mathbb{I}_n \to \mathbb{I}_n; \quad i \mapsto i$
fBase(j)	$(j)_n : \mathbb{I}_1 \to \mathbb{I}_n; \quad i \mapsto j$
fStack	f [-] g: $\mathbb{I}_{m+n} \to \mathbb{I}_N; \quad i \mapsto \begin{cases} f(i), & \text{if } i < m \\ g(i-m), & \text{if } i > m \end{cases}$
fTensor	$f \otimes g : \mathbb{I}_{nm} \to \mathbb{I}_{NM}; \quad in + j \mapsto N * f(i) + g(j)$

generation system to support additional applications outside of triangle counting. Table 3.2 shows some of the indexing functions f used for the primitives in Table 3.1.

Broadly, SPIRAL has high level OL objects which apply mathematical expression to implicit input and output objects. The `Diag` operator for example performs the haddamard product, or element-wise multiplication, of the input, while the `RowVec` performs the scalar product of input elements. Both operators rely on functions such as the constant function, updating with a constant value, or a more complex tensor function, dynamically changing value based on the index. These mathematical operators can be composed together to obtain a concise, point-free representation of the computation.

The general flow of OL operators involves gathering a set of elements performing some computational updates and scattering the results back to the output. Taking this pattern, we could write a simple operation that gathers elements from a matrix, scales them by a value from a separate vector, and writes them to a temporary output vector. This expression would be the composition of the operators `Scat`, `Diag(fConst)`, and `Gath`, where the function `Gath` reads from a two-dimensional linear input, scales by the constant value by `Diag` and writes the result to the output vector. This computation once looped over all columns is the axpy-based sparse matrix vector multiplication (SpMV) the cornerstone for algorithms like push breadth-first search [11]. In fact, if `Diag` is replaced with a `RowVec`, the resulting mathematical expression is the SpMV based on the dot product which is used in the first search for the pull width [11]. These two expressions can then be wrapped in a `COND`, a switch statement for OL expressions, resulting in direction optimizing breadth-first search, expressed in a fully point free matter.

3.5.1 Capturing Algorithms with embedded Language Constructs

Unlike Triangle Counting, some algorithms written in GraphBLAS do not use GraphBLAS primitives entirely. For these algorithms, core language constructs, such as loops and conditionals, are required to completely express the algorithm. This proves to be a challenge for GBTLX, as it is not a general language parser, independent of the semantic capture mechanism employed. We discuss two options available for the system as currently designed to support algorithms using language constructs, using the examples of Direction Optimizing Breadth First Search (DO-BFS) and Betweeness Centrality shown in the evaluation section.

The first and simplest option is to ignore the language construct and optimize subsections of the code that rely on primitives. In the case of DO-BFS, the main language construct is the conditional that decides whether to use the push phase or the pull phase, and the loop within each phase to determine how many iterations to perform. As the core operations are written in the library, we can generally ignore these statements, matching the signature of the push/pull implementations to capture the required inputs from the user.

The previous approach comes with some performance shortcomings, as DO-BFS can use custom data structures that can significantly improve performance when deciding which phase to employ. The same can be said for Betweeness Centrality, which contains loop structures that can be merged to improve data reuse. For this reason, another approach is to mix GraphBLAS primitives with function call representations of core language constructs. In C++, for loops can be represented using `std::for_each` and conditionals can be represented using `conditional`, both members of the C++ Standard Library. These functions use lambda expressions to describe the loop/conditional body and have no restrictions on using external library calls. Therefore, using the C preprocessor, we can shim `std::for_each` and `std::conditional` to capture external loops and the loop body, as well as the conditional and if-then-else branch bodies. A similar approach is used for the pointwise multiplication in Hockney Freespace convolution using `std::transform`.

3.6 Summary

GBTLX is a LibraryX implementation that focuses on sparse linear algebra implementations of graph analytics. GBTLX, interprets GBTL as an embedded DSL and leverages code generation and automatic performance tuning to overcome the problem of combinatorial explosion in the GraphBLAS API and to avoid materialization of huge non-essential temporaries. Using the example of triangle counting, we showcase the design of the GBLTX system to discover a computation pattern and provide it as a specification for the SPIRAL code generation system for analysis and optimization. within SPIRAL are a series of transformation stages that introduce new algorithms and implementations for different hardware devices. SPIRAL's operators have been extended to support a variety of primitives in graph processing through the language of linear algebra.

Chapter 4

LibraryX for Structured Grids: ProtoX

There is a myriad of application areas in the field of scientific computing and engineering where numerical solutions to partial differential equations (PDEs) are required to be computed. Numerical methods like the finite difference method (FDM), finite element method (FEM), finite volume method (FVM) and multigrid method are used to approximate the solutions to these PDEs. The key components of these algorithms are the stencil and pointwise operations. Stencils are defined as a linear transformation,

$$S(x)_i = \Sigma_j \alpha_j x_{i+j}, \tag{4.1}$$

where $i, j \in \mathbb{Z}^D$ with D denoting the number of space dimensions, α denotes the weight and x is the multidimensional data array.

Typically these numerical algorithms are iterative in nature resulting in performing the stencil operations multiple times. Writing codes for these stencil based methods from scratch can be quite a cumbersome task for anyone. As such developers turn to libraries that provide stencil operations for them. One such library is Proto. It is a domain specific library written in C++ that provides a high level of abstraction for solving various PDEs using some of the aforementioned numerical methods.

Proto's abstraction enables ease of programmability, but has drawbacks when it comes to performance. Many of Protos' abstractions can be fused and optimized together, resulting in better performance. However, abstraction fusion is something no compiler can

easily perform. This results in additional burden on the library developers to manually introduce these optimizations. To enable abstraction fusion in Proto, we propose ProtoX, which is a C++ library based on Proto and runs a code generation system SPIRAL [36,81] in the backend. The concept of using SPIRAL in the backend and a C/C++ based library in the front has shown positive results in the past [35,83]. Some of the related works in the area of optimizing stencil computation with either automatic code generation or by optimizing data movement involved while performing stencil operation are discussed in the next section.

4.1 ProtoX

In this section we will describe the structure of ProtoX. The idea is to interpret Proto as a Domain Specific Language (DSL) with the help of SPIRAL. This is done by first interpreting an example from Proto as a mathematical specification and then map the Proto program specification to an OL expression. It is then broken down into a $\Sigma-$OL expression which introduces loop fusion. This will help generate a highly optimized C++ code. Here we will explain these ideas with respect to the 2D Poisson equation. The design layout for ProtoX is shown in Fig 4.1

4.1.1 2D Poisson equation Example

The Poisson equation is given as,

$$\Delta\phi(x,y) = \rho(x,y), \quad x,y \in \Omega := [0,1] \times [0,1], \tag{4.2}$$

where ρ is a given function and ϕ is what we are solving for. Δ is the Laplace operator. We use the 5-pt stencil as a second order finite difference approximation of the Laplacian. The Jacobi iteration method is implemented in Proto to find the solution of (4.2). Let h denote the mesh spacing for the discretized domain Ω, then the Jacobi formula for a single iteration n is given as

$$\phi_i^n := \phi_i^{n-1} + \lambda(S(\phi)_i^{n-1} - \rho), \tag{4.3}$$

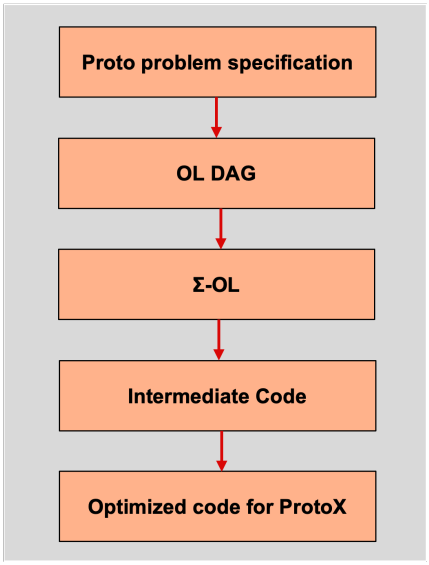


Figure 4.1: ProtoX design layout starting with a problem specification from Proto to the final optimized code generated using SPIRAL

where $\lambda = h^2/4D$ with D being the dimension of the input and $S(\phi)_i = \sum_{s \in \mathbb{Z}^2} a_s \phi_{i+s}$ is the 5-point stencil applied to the input data ϕ_i^{n-1}.

4.1.2 Algorithm Description

In Proto, the domain space Ω is divided into several boxes and (4.2) is solved for each point in the box and then information is exchanged between the boxes to update the corresponding box data. The three main steps involved in this algorithm are

1. Applying the 5-pt Laplacian stencil to the initial guess given for ϕ.
2. Approximate the new value for ϕ using the Jacobi iteration method shown in (4.3).
3. Check the latest approximation of ϕ against the convergence criterion. Max norm is used in Proto to check for convergence.

We would like to note that in Proto each of the steps described in the algorithm above correspond to separate C++ function calls. Figure 4.3 provides the code sample that is used in Proto to solve (4.2). We can observe that the code reads in the same way

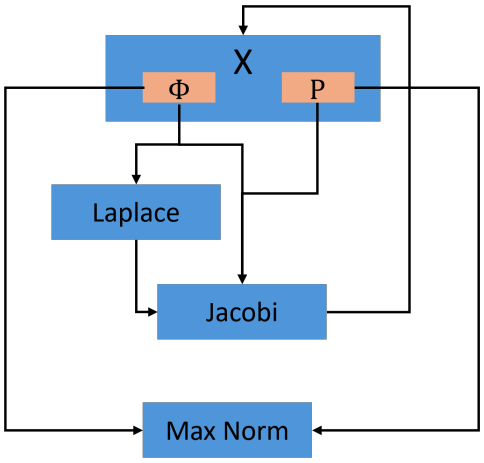


Figure 4.2: DAG for the Poisson problem in Proto.

as discussed in the algorithm above indicating an ease of use from a user perspective. However, this leads to intermediate data holders generating too much memory traffic. Hence, some optimization techniques need to be applied either at a compiler level or at an algorithmic level to overcome this issue.

```
1  // Defining the 5-pt Laplacian stencil
2  Stencil<double> laplace = Stencil<double>::Laplacian();
3  double lambda, wgt, dx; //parameters to function defined externally
4  for (int iter = 0; iter < maxiter; iter++){
5  ...
6    // Solve for all boxes with each Box of size 64x64
7    for (auto dit=phi.begin();*dit != dit.end();++dit){
8      BoxData<double>& phiPatch = phi[*dit]; //pointer to individual box in patch
9      BoxData<double>& rhoPatch = rho[*dit]; //pointer to individual box in patch
10
11     // Compute the Laplacian
12     BoxData<double> temp = laplace(phiPatch,wgt);
13
14     // Jacobi iteration
15     forallInPlace(jacobiUpdate, phiPatch, temp, rhoPatch,lambda);
16   }
17    // Computing || ||_{inf}
18    double resmax=computeMaxResidualAcrossProcs(phi, rho, dx); //norm across patches
19 }
```

Figure 4.3: Sample Proto code for the 2D Poisson problem

Operation	Description
$(A_{m\times n} \circ B_{p\times q})$	Operator composition: AB if $n = p$
$(A_{m\times n} \oplus B_{p\times q})$	Direct sum operation: $\begin{bmatrix} A & \\ & B \end{bmatrix}$
$(A_{m\times n} \otimes B_{p\times q})$	Tensor product: $\begin{bmatrix} A_{00}B & \cdots & A_{0,n-1}B \\ \vdots & \ddots & \vdots \\ A_{m-1,0}B & \cdots & A_{m-1,n-1}B \end{bmatrix}$
$\left[-\right]$	Vertical stacking $\left[\frac{A_{m\times n}}{B_{p\times q}}\right]$
$\left[-\right]_{i=0}^{k}$	Iterative vertical stack
I_n	Identity matrix $I_{n\times n}$
$pw_{x\mapsto f(x)}^{r\times s}$	Pointwise operation
(a, b, c)	Row vector with three entries a, b and c

Table 4.1: List of some operations used in SPIRAL for this work is shown here. The matrices $A_{m\times n}$ and $B_{p\times q}$ are considered as operators with $A : \mathbb{R}^n \to \mathbb{R}^m$ and $B : \mathbb{R}^q \to \mathbb{R}^p$.

4.1.3 SPIRAL Implmentation

Figure 4.2 provides the data flow for the algorithm discussed above. The first step in generating an optimized code using SPIRAL is to understand the data flow of the problem specification. Consequently, this data flow needs to be translated into OL with the help of the different operations shown in table 4.1.

We will consider the case where the size of each `Box` in Proto for this problem specification is $n \times n$ with $m \times m$ total elements in the box including ghost cells. The resulting OL expression corresponding to that data flow is shown below.

$$\text{Poisson}_{n,m,t}^{\ell,w,a} \to \begin{bmatrix} \text{Jacobi}_{n,m,w,l} \\ \|.\|_{\infty}^{n,m,a} \end{bmatrix} \circ \left(\begin{bmatrix} \text{I}_{n^2} \\ \text{Laplace} \end{bmatrix} \oplus \text{I}_{n^2} \right) \circ X, \tag{4.4}$$

$$\text{Laplace}_{n,m,t} \to \text{Scatter}_{n^2\times m^2} \circ [\text{Filt}(t)]_{i=0}^{m^2}, \tag{4.5}$$

$$\text{Jacobi}_{n,m,w,l} \to (1, w, -\lambda) \otimes \text{I}_{n^2}, \tag{4.6}$$

$$\|.\|_{\infty}^{n,m,a} \to \text{Max} \circ pw_{x\mapsto|x|}^{n\times n} \circ (0, 1/(a^2), -1) \otimes \text{I}_{n^2}. \tag{4.7}$$

Here X denotes the linearized input vector. For this problem specification it contains the initial data for ϕ and ρ with ϕ being of size $m \times m$ or m^2 and ρ is of size n^2. Hence,

X is of size $n^2 + m^2$. t denotes the filter taps corresponding to the 5-pt stencil. Its a 3×3 matrix with entries as $[0, 1, 0], [1, -4, 1]$ and $[0, 1, 0]$ for the first, second and third row respectively. This matrix is flattened out and is iteratively applied to the input vector with the proper shifts. Scatter$_{n^2 \times m^2}$ denotes the *scatter matrix* [15] in SPIRAL. ℓ, w and $, a$ are scalar parameters used for the Jacobi iteration and the Laplacian.

The next step is to rewrite the OL breakdown rules shown in (4.4)-(4.7), in terms of $\Sigma-$OL. This is where loop merging is introduced, which helps in fusing different Proto abstractions, resulting in a considerable amount of performance gain. At this stage, the computation broken down to a per point granularity rather than the entire input, which enables fine grained operation merging. A sample of the $\Sigma-$OL expression for this problem its specification is shown below.

$$\left(\sum_{j=0}^{N-1} S_{r_j \text{MaxNorm}}\, G_{s_j}\right) \circ \left(\sum_{j=0}^{N-1} S_{p_j \text{Jacobi}}\, G_{q_j}\right) \circ \left(\sum_{j=0}^{N-1} S_{u_j \text{Laplace}}\, G_{t_j}\right). \tag{4.8}$$

The G and S are the gatther and scatter functions that are used in SPIRAL to read and write data [36]. The subscripts t_j, u_j, q_j, p_j, s_j and r_j are the functions that indicate how many points should be read for each operation (like the Laplace, Jacobi and MaxNorm in this case) as well as how many points should be written after the computation is complete.

```
1  //spiral_generated.hpp
2  void Poisson_2D_fused(double  *Y, double  *X, double weight1, double lambda1, double *rhs,
3                        double a_h1, double  *retval1){
4      for(int i1 = 0; i1 <= 4095; i1++) {
5          double s20, s21, s22;
6          int a48, b15;
7          b15 = ((66*(i1 / 64)) + (i1 % 64));
8          a48 = (b15 + 67);
9          s20 = X[a48];
10         s21 = ((X[(b15 + 1)] - (4.0*s20)) + X[(b15 + 66)] + X[(b15 + 68)] + X[(b15 + 133)]);
11         s22 = rhs[i1];
12         Y[a48] = ((s20 + (weight1*s21)) - (lambda1*s22));
13         *(retval1) = ((((*(retval1) >= fabs((((1.0/(a_h1*a_h1))*s21)-s22))))) ? (*(retval1)) :
14                       (fabs((((1.0/(a_h1*a_h1))*s21) - s22))));
15     }
16 }
```

Figure 4.4: SPIRAL generated CPU code for the merged 2D Poisson equation for a `Box` of size 64×64. All the abstractions from Proto have been fused into one single function call with a single loop.

```
//spiral_generated.hpp
#include <omp.h>
#include <math.h>
const int NUM_THREADS = 4;

//SPIRAL code for OpenMP
void possion_2d(double  *Y, double  *X, double weight1, double lambda1, double a_h1,
                double  *rhs, double  *retval1) {
    #pragma omp parallel num_threads(4)
    #reduction (max : retval)
    {
        { /* begin parallel loop */
            int tid1 = omp_get_thread_num();
            for(int i1 = tid1; i1 <= 4095; i1 += 4) {
                int a51, b15;
                double s28, s29, s30, s31, s32;
                b15 = ((66*(i1 / 64)) + (i1 % 64));
                a51 = (b15 + 67);
                s28 = X[a51];
                s29 = ((X[(b15 + 1)] - (((4.0)*(s28)))) + X[(b15 + 66)] + X[(b15 + 68)] + X[(b15 + 133)]);
                s30 = rhs[i1];
                s31 = ((s28 + ((weight1)*(s29))) - (((lambda1)*(s30))));
                Y[a51] = s31;
                s32 = max(*(retval1), abs((((((1.0) / (((a_h1)*(a_h1))))) *(s29))- (s30))));
                *(retval1) = s32;
            }
        } /* end parallel loop */
    }
}
```

Figure 4.5: SPIRAL generated code for OpenMP using four threads.

```
//spiral_generated.hpp
#include "hip/hip_runtime.h"

__global__ void ker_code0(double  *X, double  *Y, double weight1, double lambda1,
                          double a_h1, double  *retval1) {
    if (((((256*blockIdx.x) + threadIdx.x) < 4096))) {
        double s21, s22;
        int a66, a67, b16;
        a66 = (threadIdx.x + (256*blockIdx.x));
        b16 = ((66*(a66 / 64)) + (threadIdx.x % 64));
        a67 = (b16 + 67);
        s21 = X[a67];
        s22 = ((X[(b16 + 1)] - (4.0*s21)) + X[(b16 + 66)] + X[(b16 + 68)] + X[(b16 + 133)]);
        Y[a67] = ((s21 + (weight1*s22)) - (lambda1*X[(a66 + 4356)]));
        *(retval1) = (((((*(retval1) >= fabs((((1.0 / (a_h1*a_h1))*s21) - s22))))) ?
                     (*(retval1)) :
                     (fabs((((1.0 / (a_h1*a_h1))*s21) - s22))));
    }
}
void possion_2d(double  *Y, double  *X, double weight1, double lambda1, double a_h1,
                double  *rhs, double  *retval1) {
    dim3 b17(256, 1, 1), g1(17, 1, 1);
    hipLaunchKernelGGL(ker_code0, dim3(g1), dim3(b17), 0, 0, X, Y, weight1, lambda1, a_h1, retval1);
}
```

Figure 4.6: SPIRAL generated code for AMD GPUs.

Once the optimization process in $\Sigma-$OL is complete, an intermediate representation of the expression is generated, which produces the final optimized code for various hardware platforms. Figures 4.4, 4.5, and 4.6 show the generated code for the 2D Poisson prob-

lem as captured and optimized using SPIRAL, targeting various hardware platforms. The code is generated for the `Box` size of 64×64 with the overall domain size being 256×256. We can see that in comparison to the original code in Proto (see Fig. 4.3), which has various library calls for the different computational kernels required, the SPIRAL generated code produces a single kernel for the entire computation. In this kernel, all three operations, Laplace, Jacobi, and MaxNorm are fused into one single loop, significantly improving data locality, arithmetic intensity, and kernel launch overhead.

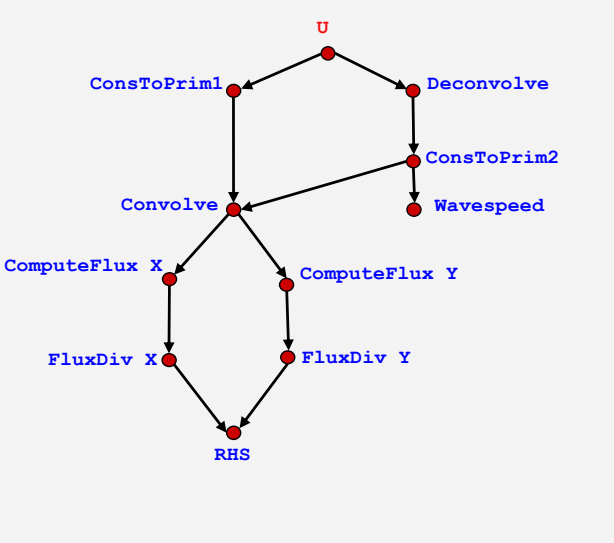


Figure 4.7: Dataflow for the spatial discretization for the 2D Euler equations implementation in Proto.

4.2 Σ-OL transformations in 2D Euler equations

We showcase some more Σ-OL optimizations for the 2D Euler equations example. Figure 4.7 shows the computational DAG of the 2D Euler equations, specifically for the spatial discretization portion. It takes, as input, a multi-dimensional vector, U, consisting of four matrices stacked on top of each other. Then it moves through a series of pointwise operations such as `ConstToPrim1` and `ConstToPrim2`, as well as some stencil operations such as `Deconvolve` and `Convolve`. Focusing on these operations, we show two key fusion opportunities available in this computation, namely fusion of a stencil followed by a

pointwise operation and fusion of a pointwise operation followed by a stencil. We first begin by showing the OL formulas for each of the operations discussed, `ConstToPrim1`, `ConstToPrim2`, `Deconvolve`, and `Convolve`. For each expression, we use an input box (matrix or tensor depending on the input dimension) of length n in each dimension.

The key benefits of performing these two fusions are an increase in arithmetic intensity and a decrease in the overall memory footprint. If each operation exists in isolation, they need to write out the entire output before the next operation can occur. However, if these operations are fused, only the working set within the kernel will change. This working set will be significantly smaller than writing out the entire intermediate memory to move between operations.

$$\begin{aligned}\text{ConsToPrim(1 and 2)}_{n,\gamma} \rightarrow & \left(\mathrm{I}_{3n^2} \oplus \text{pPrim}_{n,\gamma}\right) \\ & \circ \begin{bmatrix} \mathrm{I}_{3n^2} \\ (\mathrm{I}_{n^2} \oplus \text{StatePrimX2rho}_n \oplus \text{StatePrimY2rho}_n \oplus \mathrm{I}_{n^2}) \end{bmatrix} \\ & \circ (\mathrm{I}_{n^2} \oplus \text{StatePrimX}_n \oplus \text{StatePrimY}_n \oplus \mathrm{I}_{n^2})\end{aligned} \tag{4.9}$$

$$\text{StatePrimX}_n \rightarrow \mathrm{X}_{2\times n\times n} \odot \left(\text{Pointwise}^{n\times n}_{x\mapsto 1/x}\left(\mathrm{X}_{1\times n\times n}\right)\right) \tag{4.10}$$

$$\text{StatePrimY}_n \rightarrow \mathrm{X}_{3\times n\times n} \odot \left(\text{Pointwise}^{n\times n}_{x\mapsto 1/x}\left(\mathrm{X}_{1\times n\times n}\right)\right) \tag{4.11}$$

$$\text{StatePrimX2rho}_n \rightarrow \text{Pointwise}^{n\times n}_{x\mapsto x^2}\left(\text{StatePrimX}_n\right) \odot \left(\mathrm{X}_{1\times n\times n}\right) \tag{4.12}$$

$$\text{StatePrimY2rho}_n \rightarrow \text{Pointwise}^{n\times n}_{x\mapsto x^2}\left(\text{StatePrimY}_n\right) \odot \left(\mathrm{X}_{1\times n\times n}\right) \tag{4.13}$$

$$\text{pPrim}_{n,\gamma} \rightarrow \left(0, \frac{-(\gamma-1)}{2}, \frac{-(\gamma-1)}{2}, (\gamma-1)\right) \otimes \mathrm{I}_{n^2} \tag{4.14}$$

$$\text{Deconvolve}_n \rightarrow (-1/24, 1) \otimes \mathrm{I}_{4n^2} \circ \begin{bmatrix} \oplus_{i=1}^{4} \text{Filt} \cdot \mathrm{X}_{i\times n\times n} \\ \oplus_{i=1}^{4} \mathrm{I}_{n^2,i} \cdot \mathrm{X}_{i\times n\times n} \end{bmatrix} \tag{4.15}$$

$$\text{Convolve}_n \rightarrow (-1/24, 1) \otimes \mathrm{I}_{4n^2} \circ \begin{bmatrix} \oplus_{i=1}^{4} \text{Filt} \cdot \text{ConsToPrim1}_n \\ \oplus_{i=1}^{4} \mathrm{I}_{n^2,i} \cdot \text{ConsToPrim2}_n \end{bmatrix} \tag{4.16}$$

4.2.1 Fusing Stencil followed by Pointwise

The first optimization we discuss is fusing a stencil operation that is followed by a pointwise optimization. This is a relatively simple optimization as there are no complicated data dependency analyses that need to be performed. At a high level, this optimization requires verifying the iteration spaces of the stencil output and the pointwise match and that all output points needed for the pointwise for a given iteration are precalculated by the stencil. We show the Σ-OL transformations for `Deconvolve` and `ConsToPrim2` to highlight this type of fusion. For each of these expressions, we only show the Σ-OL for one component for simplicity, and add the subscript m to signify the total number of components.

$$\text{Deconvolve}_n \rightarrow \sum_{j=0}^{(n-2)(n-2)} \text{S}_{j_m} \left(-1/24, 1\right)_m \circ \begin{bmatrix} \text{Filt} \cdot \text{G}_k \\ \text{G}_f \end{bmatrix}_m \tag{4.17}$$

Equation 4.17 shows the Σ-OL for `Deconvolve` that introduces the gather and scatter functions. The first operation is a vertical stack that places a stencil on top and collects the center of the stencil underneath. For the stencil gather function, we define the indexing function f where $f = (\left\lfloor \frac{j}{n} \right\rfloor \times n + j \mod n) + n$. In this formula n is the number of columns in the input matrix. This function helps to capture the value of the input vector that corresponds to the center of the stencil. To capture each point of the stencil, we use the piecewise gather function called the stack $[-]$, expressed with the variable k. For the five-point stencil case, k captures the center f along with the other 4 points $f - n$, $f - 1$, $f + 1$, and $f + n$ storing them as a linearized array in row-wise order (points above the center, followed by points along the center, followed by points below the center). As the other gather function only requires the center, we use f. The two resulting points are used for the scalar product with the row vector containing the points $\frac{-1}{24}$ and 1, before the output is written.

$$\begin{aligned}\text{ConsToPrim2}_{n,\gamma} \rightarrow & \sum_{j=0}^{n*n} \text{S}_g \left(\text{pPrim}_{n,\gamma}\right) \\ & \circ \begin{bmatrix} \text{I}_3 \\ (\text{StatePrimX2rho}_n \oplus \text{StatePrimY2rho}_n) \end{bmatrix} \\ & \circ (\text{StatePrimX}_n \oplus \text{StatePrimY}_n)\, \text{G}_g \end{aligned} \tag{4.18}$$

Equation 4.18 shows the Σ-OL for `ConsToPrim2`. Here, the function g captures a point across all components as required to perform the pointwise function. It is another piecewise gather function collecting the same point at component stride, n^2. This will give us a collection of j, $j + n^2$, $j + 2 \times n^2$, and $j + 3 \times n^2$. Within SPIRAL, we verify that the iteration space of `Deconvolve` matches the iteration space of `ConsToPrim2`. If this is true, we can merge expressions by placing the operations in `ConsToPrim2` immediately after the expression in `Deconvolve`. The final expression is shown below.

$$\begin{aligned}\text{Deconvolve}_n + \text{ConsToPrim2}_{n,\gamma} \rightarrow & \sum_{j=0}^{(n-2)*(n-2)} \text{S}_g \left(\text{pPrim}_{n,\gamma}\right) \\ & \circ \begin{bmatrix} \text{I}_3 \\ (\text{StatePrimX2rho}_n \oplus \text{StatePrimY2rho}_n) \end{bmatrix} \\ & \circ (\text{StatePrimX}_n \oplus \text{StatePrimY}_n)\, \text{G}_g \text{S}_{j_m}\, (-1/24, 1)_m \\ & \circ \begin{bmatrix} \text{Filt} \cdot \text{G}_k \\ \text{G}_f \end{bmatrix}_m \end{aligned} \tag{4.19}$$

4.2.2 Fusing Pointwise followed by Stencil

The second optimization that we discuss is how to fuse a pointwise that is immediately followed by a stencil. Unlike the previous optimization, it is not as simple as performing an iteration space analysis and then merging. This is because the pointwise needs to be performed for every point of the stencil operation before the stencil operation can be performed successfully. The general process for this optimization requires merging

the indexing of the gathers for both the pointwise and stencil together at the beginning of the operation, then performing the pointwise on the updated set of input points followed by the stencil. Due to extra complexity in the case of `ConsToPrim1` and `Convolve`, we showcase the example of reordered `ConsToPrim2` and `Deconvolve` to illustrate this optimization.

To begin, we pull the gather expressions from each of the Σ-OL operators to the beginning of the expression as shown below.

$$\begin{bmatrix} G_k \\ G_f \end{bmatrix}_m \cdot G_g \tag{4.20}$$

We now want to unify the Gather expressions so that we can perform the pointwise on all the required points of the filter. To do this, we use the distributive property of gather functions, allowing us to compose the k, g, and f expressions defined previously. This produces

$$\begin{bmatrix} G_k \cdot G_g \\ G_f \cdot G_g \end{bmatrix}_m . \tag{4.21}$$

We recognize that the points needed for the stencil are defined in k and f, while the points in g describe the stride capture that point across the four input components to perform the pointwise. Therefore, we perform a function composition to pull the indexing mapping functions together along with a reordering to first gather the points for the stencil and then gather for each stencil point the required component points to perform the pointwise. We see this unified expression below.

$$\begin{bmatrix} G_{g \circ k_m} \\ G_{g \circ f_m} \end{bmatrix} . \tag{4.22}$$

We can further unify the stack of Gathers into a single Gather stacking the indexing functions which we unify with the new capture function u shown below.

$$G_u \tag{4.23}$$

After capturing all the points, we need to apply the pointwise operation to each set of points captured by the new gather. We show this in the expression below. Unlike equation 4.18 we have to change the domain of this expression to only compute over the set of points we gathered instead of the whole input. We show that u is based on the outer loop variable using the subscript j. The scatter function c is a strided write which writes each value at the number of component stride. This groups the points required for the stencil at unit stride, as each group of stencil points is stacked on top of each other for each component.

$$\begin{aligned}\text{ConsToPrim2}_{n,\gamma} \rightarrow \sum_{l=0}^{u} \text{S}_c \left(\text{pPrim}_{n,\gamma}\right) \\ \circ \begin{bmatrix} \text{I}_3 \\ (\text{StatePrimX2rho}_n \oplus \text{StatePrimY2rho}_n) \end{bmatrix} \\ \circ (\text{StatePrimX}_n \oplus \text{StatePrimY}_n)\, \text{G}_{u_j}\end{aligned} \tag{4.24}$$

Having updated `ConsToPrim2` we can now merge it inside of `Deconvolve` where we use new gather functions to index over the output of ConstToPrim2 rather than the entire input coming in. The first function h is a unit stride read of the points required for the stencil, as each group of stencil points is stacked on top of each other. The second function d is a strided read, finding the center of each stencil depending on which component its calculating. The final expression is shown below.

$$\begin{aligned}\text{ConstToPrim2 + Deconvolve}_n \rightarrow \sum_{j=0}^{(n-2)(n-2)} \text{S}_{j_m} \left(-1/24, 1\right)_m \circ \begin{bmatrix} \text{Filt} \cdot \text{G}_h \\ \text{G}_d \end{bmatrix}_m \\ \circ \sum_{l=0}^{u} \text{S}_c \left(\text{pPrim}_{n,\gamma}\right) \circ \begin{bmatrix} \text{I}_3 \\ (\text{StatePrimX2rho}_n \oplus \text{StatePrimY2rho}_n) \end{bmatrix} \\ \circ (\text{StatePrimX}_n \oplus \text{StatePrimY}_n)\, \text{G}_{u_j}\end{aligned} \tag{4.25}$$

In Equation 4.25 we have a Gather function that is based on the outer loop variable j, collecting a number of points required for the stencil and for each set of points performing the pointwise operation based on domain of u. This is written to a temporary vector that

is accessible by the gather functions h and d to perform the stencil and produce the result. Although the pointwise is shown as a loop in this expression, it is fully unrolled in the generated code as a performance optimization, which is why there is no dependence on the loop variable l.

4.2.3 Nesting Optimizations

Having shown how to fuse a pointwise followed by a stencil along with a stencil followed by a pointwise, we can recursively perform these optimizations to our entire 2D Euler equations computation. This will significantly reduce the memory footprint of the application as now the only boundary between kernels is the stencil boundary, a current hard boundary within ProtoX. By performing these optimizations recursively, there exist opportunities to find other optimizations. In the case of the 2D Euler equations example after fusing `Deconvolve` and `ConsToPrim2` and fusing `ConsToPrim1` with `Convolve`, we can notice that the iteration spaces for both kernels are the same, $(n-2)^2$ the iteration space of the stencil. This allows us to fuse those two kernels together as long as we do the instruction scheduling such that the final reduction takes the result of those two fused expressions.

4.3 Capturing Modern Language Features

```
void f_consToPrim_(
        double*&          a_W,
        const double*&    a_U,
        double a_gamma)
{
    double rho = a_U[0];
    double v2 = 0.0;
    a_W[0] = rho;

    double v_1, v_2;
    v_1 = a_U[1] / rho;
    v_2 = a_U[2] / rho;
    a_W[1] = v_1;
    a_W[2] = v_2;
    v2 = v_1*v_1+v_2*v_2;

    a_W[3] = (a_U[3] - .5 * rho * v2) * (a_gamma - 1.0);

}
```

Figure 4.8: User-defined Pointwise operation in Proto, which is passed as part of the general Pointwise infrastructure.

Proto is a modern library that contains modern features, including the ability to leverage user-defined pointwise functions expressed as lambdas as shown in Fig. 4.8. Given that these are outside the base library, it is difficult to provide OL expressions easily. ProtoX uses two approaches to capture these lambda expressions. The first approach involves overriding primitive data types in the base language. By overriding all primitive types, ProtoX is able to overload all assignment and arithmetic operators for these data types. During execution, ProtoX then prints each operation within the lambda expression in SPIRAL's IR. This translation follows the three-address code model seen in most compilers which breaks arithmetic expression chains into one output two input expressions, passing the intermediate outputs while unwinding the chain. Using traditional compiler optimizations in SPIRAL, like strength reduction, SPIRAL can optimize the captured IR to produce an optimized implementation shown in Fig. 4.9. Currently, ProtoX is able to capture these types of lambda expressions given the following criteria, only primitive types are used, basic if-then-else control flow, no loops, and no data dependent control flow.

```
func(TVoid, "f_consToPrim", [ Y, X, gamma1 ],
   loop(i1, [ 0 .. 1599 ],
      decl([ a36, a37, a38, a39, a40, s24, s25, s26 ],
         chain(
            assign(s24, nth(X, i1)),
            assign(nth(Y, i1), s24),
            assign(a36, add(i1, V(1600))),
            assign(s25, div(nth(X, a36), s24)),
            assign(nth(Y, a36), s25),
            assign(a37, add(i1, V(3200))),
            assign(s26, div(nth(X, a37), s24)),
            assign(nth(Y, a37), s26),
            assign(a38, add(i1, V(4800))),
            assign(a39, sub(gamma1, V(1.0))),
            assign(a40, mul(V(0.5), a39)),
            assign(nth(Y, a38), sub(mul(a39, nth(X, a38)), add(mul(a40,
            mul(mul(s25, s25), s24)), mul(a40, mul(mul(s26, s26), s24)))))
         )
      )
   )
)
```

Figure 4.9: SPIRAL optimized IR for a captured lambda function.

Given the significant number of constraints on direct lambda capture, ProtoX introduces another capture mechanism for lambdas as callback functions within SPIRAL generated code. ProtoX has demonstrated that operation fusion between stencil and pointwise operations is great for performance and callbacks provide that functionality without inspection. ProtoX wraps the lambda in an OL container, capturing its arguments and function name. SPIRAL then embeds these callbacks into stencil operations when possible. During runtime compilation it binds those lambda expression declarations to the generated code allowing the generated code to invoke those functions.

4.4 Summary

ProtoX is the LibraryX implementation for structure grid applications. ProtoX interprets the C++ domain specific library Proto, enabling optimization for key operators, stencil and pointwise. Using the SPIRAL code generation system, ProtoX can optimize structured grid applications by providing optimized implementations of stencils and pointwise operations, while also finding opportunities to perform operation fusion. This fusion includes a pointwise followed by a stencil, a stencil followed by a pointwise, and iteration space

merging. These optimizations enable a significant reduction in temporary memory and an increase in arithmetic intensity. As Proto is a modern library that uses modern C++ language features, ProtoX is designed to support capturing these features. This includes the ability to capture arbitrary pointwise functions as SPIRAL IR or reference pointwise functions directly through function callbacks.

Chapter 5

Extending the LibraryX Backend: IRISX

Modern high-performance computing (HPC), cloud, and embedded systems have become increasingly heterogeneous with multiple coexisting accelerators. These accelerators provide significant performance benefits, with the trend of accelerators being predicted to intensify in the future with various kinds of accelerators co-existing for various computations. Although these heterogeneous systems provide significant performance improvements, they introduce two key challenges [92]. The first challenge is performance portability, where a common abstraction is employed to ensure programming productivity, which contains architecture-optimized implementations for different hardware platforms. Abstractions such as Kokkos [29] or code generation systems such as SPIRAL [32] aim to address this challenge. The second challenge is efficient utilization of multiple accelerators housed in a single node, as data movement and orchestration of multiple accelerators increases the complexity of building portable code. Runtime systems such as StarPU [10] or IRIS [52] focused on providing solutions for orchestration and data movement challenges in diverse heterogeneity through task graphs. As these challenges will only become more complicated, a new portable abstraction is needed that addresses both.

There also exists a third connected challenge, tuning, based on factors from the application and specific hardware being utilized. These factors include kernel implementation, computation representation based on concurrency, number of devices, and efficient

scheduling to reduce unnecessary data movement. Although solutions involving tuning have been incorporated with prior challenges individually, they must all be combined to achieve the best performance. *Therefore, an application using an ideal portable abstraction should have the ability to automatically adapt itself based on the architectures and number of devices in a node to provide the best performance by dynamically employing optimized kernels with the right granularity and a graph representation of the computation that enables efficient orchestration.* To the best of our knowledge, no such solution exists.

This work presents IRISX, a dynamic trade-off system for harnessing multi-accelerator heterogeneity that strives towards providing the ideal solution mentioned above. IRISX exposes architecture-agnostic high-level interfaces to applications which provide functional portability, and at runtime, it establishes an active interaction between the SPIRAL code generation engine that generates architecture-optimized kernels and the heterogeneous runtime IRIS [71] to efficiently orchestrate computation to ensure performance. IRISX goes beyond the state-of-the-art efforts by employing dynamic adaptation through re-organizing the computation at the kernel and task graph levels to provide efficient execution using various underlying heterogeneous processors.

When invoked from the application, IRISX discovers the underlying heterogeneous architectures, generates the optimized kernels for those architectures, and automatically creates a directed acyclic graph (DAG) that is, task graph. IRISX then automatically performs data movement among heterogeneous processors, while leveraging scheduling strategies to ensure concurrent utilization. This process involves a model-guided search for the best configuration that takes advantage of various opportunities for kernel, task, and graph concurrency depending on hardware resources. Moreover, IRISX can dynamically discover a functional execution by employing a memory model if the size of a problem would exceed the memory of the available hardware and can reduce the memory footprint by regenerating fused kernels if needed. Using the class of structured grid problems, IRISX demonstrates the wide variety of execution scenarios based on accelerator computation capabilities and problem size, where it is capable of efficiently finding the optimal kernel and graph-level computation granularity for various heterogeneous systems. By consid-

ering state-of-the-art high-performance computing platforms (Frontier, Aurora, ExCL and Cades cloud), IRISX demonstrates its capability of providing a portable interface that dynamically selects the best configuration to provide the best performance.

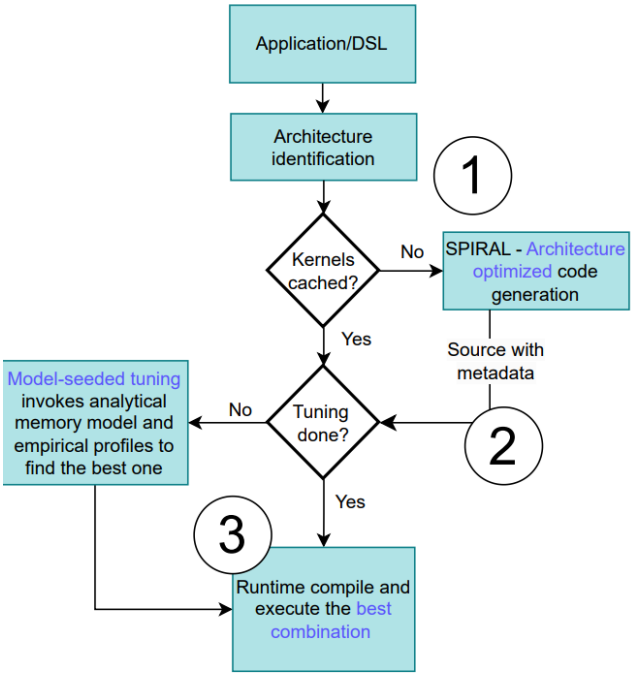
5.1 IRISX Design

We discuss the main components of IRISX, shown in Fig. 5.1, its design flow, and highlight novel functionality enabled by the dynamic interactions and tight coupling of these systems.

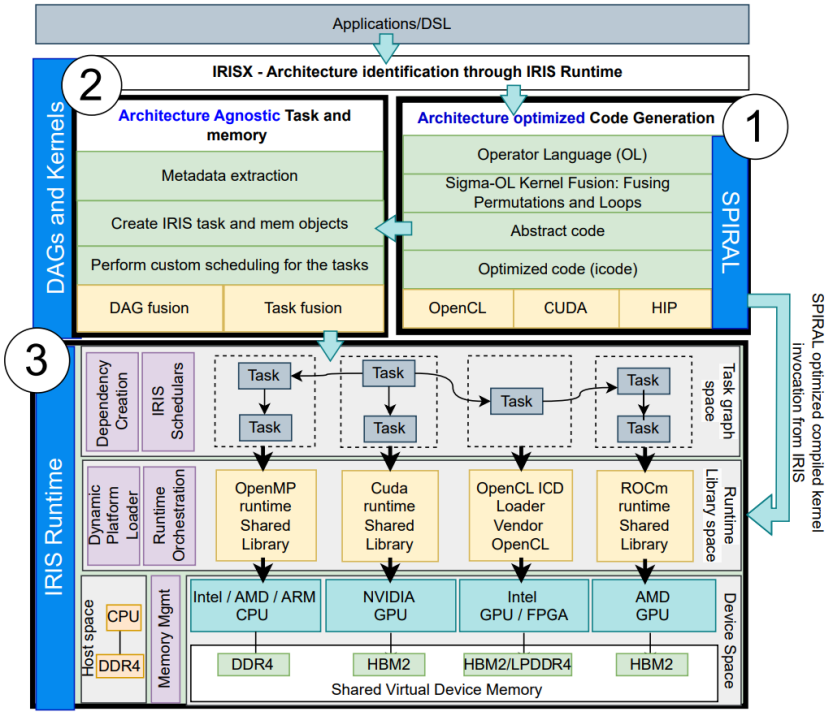
5.1.1 Components

SPIRAL. The code generation system, SPIRAL, has been in development for more than 25 years [5]. SPIRAL was originally developed to generate optimized kernels for linear transforms like the discrete Fourier transform (DFT), discrete sine, and cosine transforms. The internal language of SPIRAL, the Signal Processing Language (SPL) [94], was critical to enabling optimization in both the selection and the implementation spaces of the FFT algorithm. SPL has since been generalized to the mathematical language called Operator Language (OL) [31], which expands the scope of SPIRAL to application domains other than signal processing. Recent work in the areas of graph algorithms [83], partial differential equations (PDE) [68], and cryptography [98] showcases the use of SPIRAL in different scientific domains.

IRIS Runtime. 2024 R&D 100 award winner, IRIS [22,52] is an intelligent task-based runtime designed for diverse heterogeneity. IRIS is designed to work with heterogeneous computing systems that comprise multicore CPUs, GPUs (NVIDIA, AMD, Intel), DSPs (Qualcomm Hexagon), and FPGAs (Xilinx, Intel). It accommodates kernels written in various programming languages, including OpenMP, OpenCL, CUDA, HIP, XilinxCL, IntelCL, Hexagon C++, and OpenACC. IRIS provides fundamental abstractions, tasks to express a computation, and memory objects to express data needed by the task. Both task and memory objects are architecture-agnostic. Three levels of abstraction are there



Block diagram of IRISX



Detail software stack of IRISX

Figure 5.1: IRISX Design. Top: The three main components and the flow of the IRISX system are shown. Bottom: The detailed software stack of the IRISX system is shown.

for computation, 1) kernel is the collection of instructions, 2) task is a collection of kernels where a single task can point to one or multiple kernels, and 3) DAG is the directed acyclic graph of tasks where data dependencies are expressed using edges between different tasks. At runtime, tasks invoke kernels suitable for a particular architecture. Similarly, IRIS memory objects are orchestrated at run-time, including memory allocation and transfer among devices without user intervention. In addition, IRIS provides a rich set of schedulers to efficiently schedule a DAG of tasks.

```
#include <iostream>
#include <vector>
#include "fftx.hpp"
#include "irisx.hpp"

int main(int argc, char** argv) {
  int n,m,k;
  n = 8;
  m = 8;
  k = 8;
  int length = n*m*k*2;
  std::vector<int> sizes{length, length, length, n, m, k};
  double *Y, *X, *sym;
  X = new double[length];
  Y = new double[length];
  sym = new double[length];
  generateInputBuffer(X, sizes);
  std::vector<void*> args{Y,X,sym};

  IRISXProblem mdp(args,sizes,"mddft");
  mdp.readKernels();
  mdp.createGraph();
  mdp.transform();

  for(int i = 0; i < n*m*k; i++) {
    std::cout << Y[i] << std::endl;
  }
  return 0;
}
```

Figure 5.2: Example of the IRISX API shown here for a multidimensional discrete Fourier transform (MDDFT) kernel, which is part of the FFTX [35] library. This design allows functional portability to various hardware platforms, and performance portability by dynamically generating optimized kernels and scheduling them on all available hardware platforms.

5.1.2 Flow in IRISX

To provide performance portability, IRISX combines an architecture-agnostic front end, architecture-optimized code generation, and the dynamic capability of modifying the

computation representation at different granularity. To enable such a combination, IRISX provides a simple class-based API (Fig. 5.2) to access its functionality. The user specifies an IRISX object that takes in its constructor all the input and output objects of the computation and their sizes and instantiates the object's virtual semantics function with the Operator Language (OL) description of the problem to be solved. The user then calls the three core functions, *readKernels*, *createGraph*, and *transform*, to generate architecture-optimized code, create the architecture-agnostic task graph, and execute the task graph. At runtime, the *readKernels* function, shown in ①, causes SPIRAL to ask IRIS for the architectures of interest and the number of resources, using those values to generate variants of optimized code, such as code with varying levels of concurrency via kernel fusion. This is done for each provided architecture and includes additional metadata for the IRIS runtime system. In ②, the *createGraph* function parses the metadata to create the intermediate memory objects and the task graph in the IRIS runtime. Within this step is a profile-based autotuning system, which determines the specific concurrency variant of the generated kernels, and which kind of scheduling should be performed for this particular input. Finally, the *transform* function, shown in ③ runtime compiles and executes the code based on the best-tuned profile discovered. Here, the IRIS runtime does on-the-fly device memory creation and task orchestration. The generated code and profile are cached for future reuse based on problem size and hardware resources.

In summary, IRISX enables the interaction between code generation in SPIRAL and runtime orchestration in IRIS to deliver the targeted dynamic system. Through such interaction, the main objective is not only to enable portability, but also to find the right representation of the computation that provides the best performance on the underlying heterogeneous systems. The various extensions that enable IRISX are further elaborated in the rest of the section.

5.1.3 SPIRAL and IRIS Interaction through Metadata Capture and Runtime Compilation

SPIRAL needs to provide not only optimized kernels but also kernel metadata that the IRIS runtime can use to execute the computation and schedule tasks. This is done by generating metadata that describe the computation flow to the runtime system. SPIRAL provides information on all intermediate memory objects, including a number of parameters, types, and sizes. Additionally, SPIRAL provides the kernel launch semantics for each of the generated kernels so that IRIS can launch them appropriately. Finally, SPIRAL provides information on the number and types of input arguments that are required for the generated code. SPIRAL helps IRIS identify opportunities for kernel-level concurrency by writing intermediate results to different memory objects for concurrent kernels. In this way, dependency analysis can be performed on the computation DAG to provide opportunities for concurrent scheduling.

IRISX compiles the generated OpenMP, OpenCL, CUDA and HIP kernels to create and cache the individual binaries to be invoked and orchestrated by IRIS. The runtime compiler takes the generated code and produces the final kernel binaries. This is then ingested by IRIS during task graph execution to find the appropriate kernel to execute.

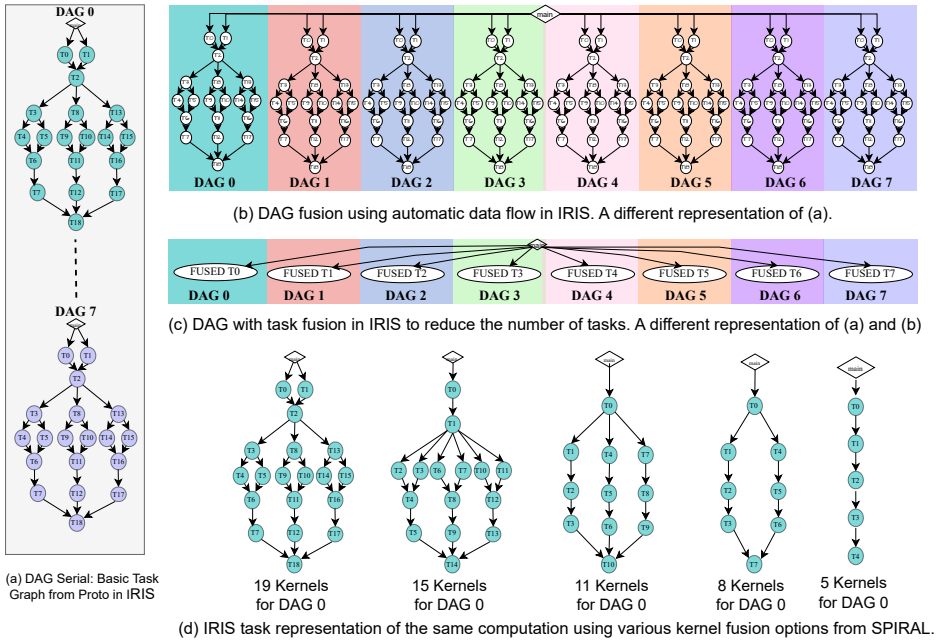


Figure 5.3: Pictorial depiction of the task graph generated in IRIS and the different fusion combinations available in IRIS and SPIRAL which are are combined in IRISX. Figure 5.3-a depicts the first possibility where the given DAG is executed in a serial manner. Figure 5.3-b increases the concurrency in the DAG by doing data flow informed DAG fusion. Figure 5.3-c depicts the combination of DAG fusion with task fusion capability in IRIS where all the kernels in a given single DAG (like in fig. 5.3-a) are fused in one task. Figure 5.3-d represents IRIS task graphs for different kernel fusion options generated using SPIRAL that are used by IRISX

5.1.4 IRIS DAG Generation and Optimization

Denoted by ② is phase-2 of IRISX shown in Fig. 5.1 where the computation is expressed using architecture-agnostic APIs from IRIS runtime. The metadata generated by SPIRAL is used in this phase. Each task points to the kernels generated by SPIRAL, where kernel arguments are expressed using IRIS memory objects. During this phase, no architectural details are expressed and no memory transfer between devices is specified; rather, existing features of IRIS runtime can facilitate data transfer among devices. Therefore, this phase is crucial to enabling functional portability. This phase also enables dynamic transformation

of the representation of the computation through DAG and Task fusion.

DAG Fusion. IRIS DAGs are represented as graph objects. Each DAG execution requires data transfer from the host to device and sends the final result back. Serial execution of DAGs are shown in Fig. 5.3-a, where such data transfers take place. IRIS has been enhanced with the capability of fusing multiple DAGs which can increase concurrency in the DAG thereby enabling the simultaneous usage of multiple devices (such as multiple GPUs) or the streams in capable devices (such as simultaneously multi-kernel execution in the same GPU using streams). IRISX utilizes these features by combining multiple graph objects, which in turn enables DAG fusion. The pictorial depiction of DAG fusion is shown in Fig. 5.3-b where it shows the increased concurrency. Moreover, DAG fusion reduces unnecessary data movement. Although DAG fusion capabilities have already been explored in other efforts by manual implementation, IRISX performs the fusion of DAGs completely transparently when the option is enabled.

Task Fusion. While DAG fusion increases the concurrency in the DAG, it also increases task management overhead because tasks are orchestrated simultaneously among heterogeneous devices. To address high task management overhead, IRISX can fuse multiple IRIS tasks together. This new fused task inherits all the kernels of previous tasks. Since all kernels are combined in a single task, task fusion reduces concurrency. Moreover, the IRIS runtime extension to support task fusion uses a single stream to serially execute the kernels. IRISX automatically merges multiple tasks and adds those merged tasks to the IRIS graph object, which then significantly reduces the total number of tasks in a graph object. The impact of task fusion on DAG fusion is shown in Fig. 5.3-c.

5.1.5 Data Flow, Scheduling and Runtime Orchestration in IRISX

After phase 2, tasks are submitted to IRIS runtime, shown by ③. During metadata generation, SPIRAL augments memory access information (such as read, write, or read/write), which is embedded into IRIS tasks in phase 2. Using this information, IRIS performs data flow analysis to create the necessary dependencies to build a DAG to determine which tasks can be executed concurrently. The upper box of ③ shows the creation of DAG

in Fig. 5.1. After creating the DAG, IRISX can employ different scheduling algorithms available in IRIS or a custom scheduling algorithm written for a specific problem or architecture. Being fully aware of the architectures, tasks, and DAG, IRISX creates a custom scheduler in phase 2 to enable concurrent execution in phase 3. Taking into account the custom scheduler, the dynamic platform loader for IRIS runtime (shown in the second layer from the top in ③) loads the binary created by runtime compilation in phase 1 and invokes the corresponding kernel from the IRIS tasks.

Scheduling can be done on a heterogeneous system by utilizing all or a subset (for example, scalability in terms of the number of GPUs) of the computing devices. IRISX enjoys this flexibility through the IRIS runtime, which can use environment variables to choose different architectures or schedulers without changing the implementation. Therefore, IRISX can dynamically change the number of devices without any change in the source implementation.

Memory orchestration is a challenging task that is required to support concurrent execution in diverse heterogeneity. IRIS runtime is equipped with automatic and efficient memory movement (such as device-to-device) based on the data flow defined DAG [71]. To take advantage of this capability, IRISX binds the IRIS memory objects to the host memory space that IRISX captures from the application. Once the mapping is established in phase 2, IRIS runtime performs necessary device memory creation and movement automatically during execution.

5.1.6 Code Generation and Kernel level Fusion

The SPIRAL code generation system provides the ability to control for various degrees of kernel fusion. This enables SPIRAL to generate different kernel variants of the computation as shown in Fig. 5.3-d. During code generation, SPIRAL transforms the provided OL expression into a Σ-OL expression which makes loops and indexing patterns for the input and output explicit for all operators. From here, Σ-OL's robust rewriting system [36] can be used to configure kernel level fusion in generated code. SPIRAL exposes fusion opportunities in the form of the stack ($[-]$) operator. At code generation time, each Σ-OL

operator within the stack can be a different kernel or can be fused together to form a larger kernel. This fusion is available only if the domains of each operator are the same and changes the amount of work per thread, not the thread geometry. Given a stack of N operators with the same domain, SPIRAL can choose to create up to N concurrent kernels each with T work per thread or one kernel with $N * T$ amount of work per thread. This can be further generalized to N/G concurrent kernels where G is a specific number of groups. Kernel fusion affects the amount of temporary memory a set of kernels requires, as temporary memory reuse can be employed as the concurrency width decreases. Generally, the smaller the number of kernels, the less temporary memory is required.

5.1.7 Model-seeded Tuning

IRISX's dynamic nature allows for a wide range of possibilities to represent the computation on different hardware architectures. On the kernel side this includes configuring the granularity of concurrency in the generated kernels (Kernel fusion). IRISX supports higher concurrency by splitting independent work into different kernels or bundling work together into fat kernels. When the graph of tasks (i.e. DAG) is created, there are different ways to organize it. This includes no fusion, which creates a task graph per operation, and dag fusion, which creates a large graph of all operations. Finally, task fusion, which bundles sets of tasks together into a single fused task within the task graph.

A model-seeded tuning approach that uses analytical memory models and empirical profiles to find which representation of computation provides a functional and efficient execution. When IRISX runs for the first time, it empirically collects the profiles for the possible combinations to determine the best-performing option for a given input (one time effort). To reduce the number of combinations on the kernel side, IRISX employs an analytical memory model to determine if certain variants will fit given the application memory requirements. IRISX queries the amount of available device memory and compares it with the total size of the memory objects used in a DAG. IRISX can eliminate execution of any variant that uses excess memory automatically and discover the functional variant of the kernels. The best option is cached for reuse in all subsequent iterations/executions

of the program so that the application gets optimal performance. Building a sophisticated model is not the focus of the work; however, this simplistic model shows the efficacy and opens the door for future research opportunties.

5.2 Summary

IRISX is a novel and dynamic system that provides high-level abstraction for performance portability for multi-device heterogeneity. By providing a hardware-agnostic API, IRISX enables the generation of architecture-optimized kernels for different heterogeneous accelerators, determines the best computation representation by employing run-time reshaping of the computation graphs, and efficiently schedules the computation graph to take advantage of concurrency. It enables dynamic interaction between the SPIRAL code generation engine and the IRIS runtime to make an application adapt for the underlying set of accelerators in a heterogeneous system. IRISX demonstrates that careful selection of kernel, task, and graph representation is critical to obtaining the best performance for varying sets of accelerators with various computation capabilities. Additionally, IRISX shows its efficacy for future heterogeneous systems with multi-kind accelerators by scaling beyond the vendor boundary.

Chapter 6

Expanding the Scope: LibraryX-ASIC & FortranX

LibraryX has been designed for C/C++ libraries and can run on hardware platforms that have defined software stacks. However, scientific software has a long history in languages like Fortran, with many legacy applications written in Fortran. Similarly, new hardware accelerators don't have the same software stack support as current hardware, making it difficult for application developers to port their applications easily to these accelerators. To address these two challenges, we extend LibraryX to support legacy Fortran applications, via FortranX, and to support new hardware accelerators with LibraryX-ASIC.

6.1 FortranX

Fortran has a long history in scientific computing due to its mathematical basis and strong compiler support. However, over the past few decades support for new features in Fortran has decreased dramatically in favor of more popular languages like C++ and Python. This poses a great challenge to legacy Fortran applications that have no modern language implementation. These applications do not benefit from new programming models or new hardware platforms without significant development effort.

To combat this development effort and leverage new features we propose FortranX.

FortranX is a compiler framework that recognizes and optimizes key algorithms in Fortran applications, without source code modification. FortranX uses a custom compiler pass to capture the semantics of the algorithm. This semantic description is passed to IRISX [84] to perform optimization and execution. IRISX is a tight coupling of the SPIRAL [32] code generation system and the IRIS runtime system. SPIRAL generates architecturally optimized code for various hardware platforms, while IRIS [53] dynamically executes those kernels on any available hardware platform, allowing automated portability.

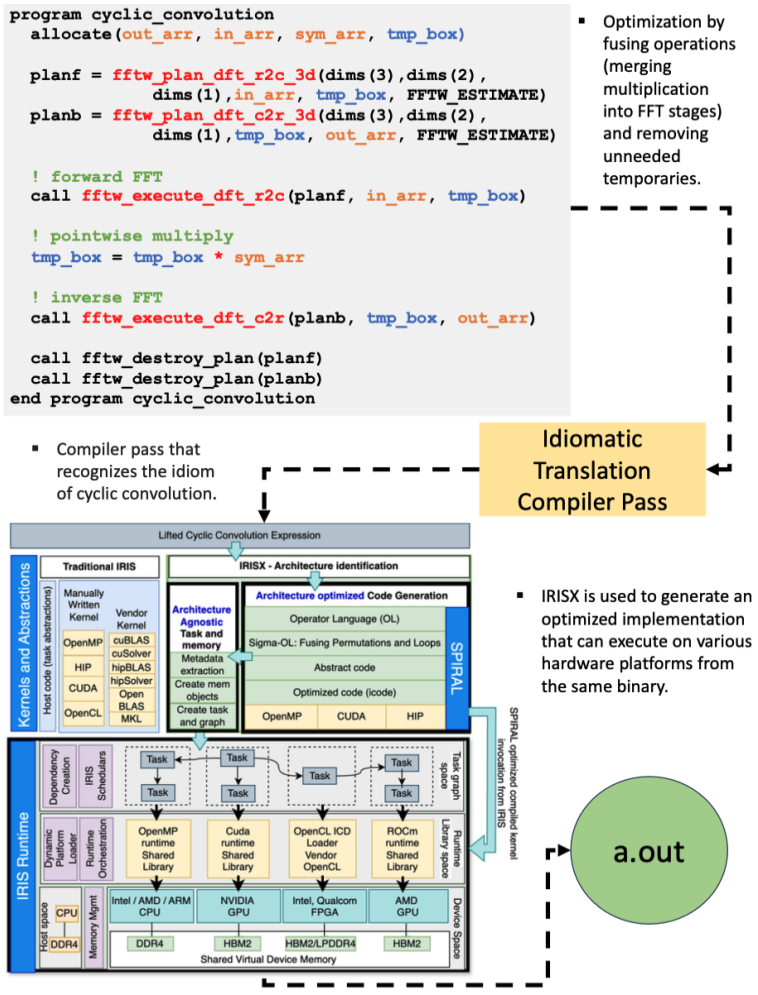


Figure 6.1: FortranX Toolflow

6.2 FortranX Design

FortranX has four distinct phases, computation capture, semantic analysis and lifting, code generation, and task scheduling/execution. Each phase uses a different internal framework to perform the specific operation. This entire process is done transparently to the user.

The first phase involves a custom compiler pass as part of the GCC [24] compiler toolchain. This compiler pass walks the GCC Intermediate Representation (IR) tree to discover the computation's implementation. Unlike general-purpose optimizations, this pass is specialized for known computation patterns. This includes library calls and perfectly nested loops. If successful, the semantic analysis phase is invoked using IRISX.

During semantic analysis, the computation pattern is tested against a known computation database. This is done using the SPIRAL code generation system within IRISX. If there is a computation match, the computation is lifted into a high-level computation abstraction expressed in Operator Language (OL) [31]. The OL abstraction is then broken down through SPIRAL's code generation system to provide an optimized implementation for various hardware platforms.

After code generation, IRISX's runtime system, IRIS, takes the generated kernels and creates a task graph for the computation. This task graph leverages metadata provided SPIRAL include task dependency information, kernel launch parameters, and temporary memory objects. This completed task graph is invoked via a function pointer that is passed back to the compiler pass. The compiler pass replaces the computation's initial IR with the provided function pointer, before creating the final binary. The end-to-end process flow for FortranX is shown in Fig. 6.1.

6.3 LibraryX-ASIC

With the end of Dennard scaling, hardware developers have begun to build domain-specific accelerators to increase computing performance. These accelerators provide two key benefits over general purpose hardware: performance and energy efficiency. Accel-

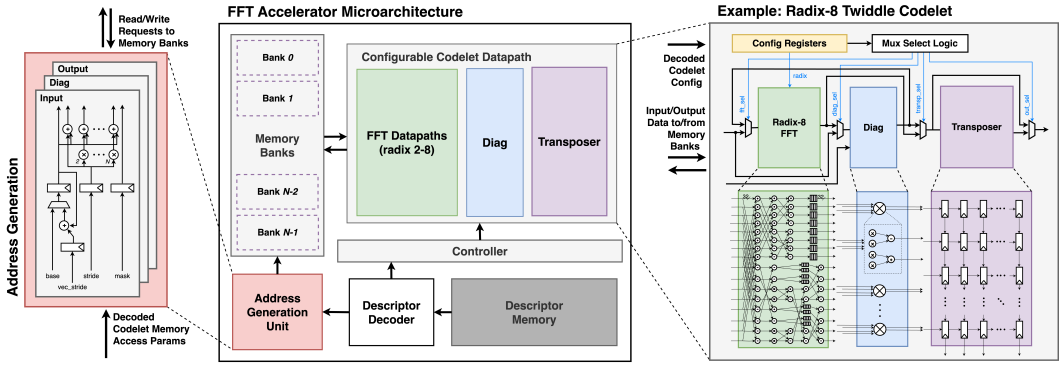


Figure 6.2: Overview of the FFT accelerator microarchitecture.

erators have custom circuits for critical operations in their domain, enabling increased energy efficiency for their workloads.

While accelerators have many benefits, they also have a few challenges specifically in programmability and adoption. Accelerator designers define their unique end-to-end software stack for developer use. This is challenging because it requires developers to potentially learn a new programming model to access the accelerator. This becomes increasingly more complicated for each new accelerator introduced, thereby limiting program portability.

To address these issues of programmability and portability of accelerator devices, we propose LibraryX-ASIC. LibraryX-ASIC is an automated framework designed to hide the complexity of accelerator offload behind domain-specific software libraries. Using the library's standard interface, LibraryX-ASIC recognizes the operation, generates an optimized accelerator implementation, and offloads it to the accelerator automatically, populating the output buffer upon completion. This relieves the programmer from having to worry about various accelerator offload paradigms. We show how LibraryX-ASIC can be utilized through an FFT benchmark and an FFT accelerator.

6.3.1 FFT Accelerator

The FFT accelerator is designed to address two primary challenges — flexibility and programmability, in existing FFT hardware implementations by following the design prin-

ciples outlined in [89]. The key observation is that software flexibility in libraries like FFTW [37] arises from recursion where the base cases are small sized FFTs known as *codelets*. Thus flexibility in hardware can be retained by designing highly configurable hardware codelets and a surrounding architecture that orchestrates their execution.

Similar to an FFTW plan, the FFT computation on the accelerator is defined by a sequence of descriptors containing the configuration parameters for the hardware codelet. Descriptors are fetched from a local instruction memory and then decoded to obtain configuration parameters including input/output base address and stride, batch size, FFT radix, and compute ordering. The codelet datapath can be reconfigured to compute different small sized FFTs and also reordered between element-wise multipliers and a transposer. Details of the accelerator microarchitecture are presented in Fig. 6.2. The codelet datapath has been silicon-verified in an FFT accelerator [88] consisting of a radix-8 FFT core and eight element-wise multipliers to accelerate a radix-8 *twiddle codelet* of FFTW.

```
#include <iostream>
#include <complex>
#include <vector>
#include "fftw3.h"
#include "LibraryX-ASIC.hpp"
using namespace std;
int main() {
  int N = 64;
  int sign = -1;
  u_int f = FFTW_ESTIMATE;
  vector<complex<float>> input(N);
  vector<complex<float>> output(N, 0.0);

  buildInput(input);

  //call is replaced with accelerator offload and executed
  fftwf_plan p = fftwf_plan_dft_1d(N, (fftwf_complex*)input.data(),
                                   (fftwf_complex*)output.data(), sign, f);
  fftwf_execute(p);

  //output buffer contains accelerator result
  checkOutput(output);

  fftwf_destroy_plan(p);
  return 0;
}
```

Figure 6.3: Example FFT program. This FFT application written against FFTW will be executed on an FFT acclerator without user modification using LibraryX-ASIC. The output buffer will be populated as if nothing changed.

6.4 End-To-End Example: FFT

We describe the LibraryX-ASIC system design using a simple FFT program shown in Fig. 6.3. We discuss how FFT library calls are recognized and captured. We then describe the high-level process to generate equivalent accelerator code using SPIRAL. Finally, we show how that code is compiled and executed on the accelerator.

```
#include "model.h"
#include "utils.h"
#include "accel.h"

Program dft_desc[7] = {
  {CONFIGI, 8, 0, 0, 1, 0, 1, 1, 0, 8},
  {MEMI, MEM_IN, 1, 8, 0xFF, LOCAL_MEM, 0},
  {MEMI, MEM_DIAG, 1, 8, 0xFF, LOCAL_MEM, LOCAL_MEM_REGION_SIZE};
  {MEMI, MEM_OUT, 1, 8, 0xFF, LOCAL_MEM, 0};
  {CONFIGI, 8, 0, 0, 0, 0, 1, 1, 1, 8};
  {MEMI, MEM_IN, 1, 8, 0xFF, LOCAL_MEM, 0};
  {MEMI, MEM_OUT, 1, 8, 0xFF, LOCAL_MEM, 0}
};

void dft64(float *Y, float *X) {
  enter();
  float *T23;
  T23 = initTwiddles64();
  dmaLoad(LOCAL_MEM, 0, 0, 8, 1, X, 8, 8, 8);
  dmaLoad(LOCAL_MEM, 1, 0, 8, 1, T23, 8, 8, 8);
  // Invoke Accelerator
  executePlan(0, dft_desc);
  dmaStore(LOCAL_MEM, 0, 0, 1, 8, Y, 8, 8, 8);
  exit();
}
```

Figure 6.4: SPIRAL generated code for the FFT accelerator. This code uses information from SPIRAL's FFT description include input ranges and input and output strides.

Capturing Library Calls. LibraryX-ASIC uses a function call capturing technique called delayed execution to intercept library calls at runtime. This transforms library calls from operations performed on inputs and outputs to specifications describing the computation to be performed. In the case of FFTs this includes the type of transform, its dimensionality, and the types for the call's input and output. LibraryX-ASIC implements its delayed execution mechanism through preprocessor directives, with the equivalent library call stored in the LibraryX-ASIC header file, which gets replaced at compile time. At runtime the LibraryX-ASIC captured call is invoked. Here, LibraryX-ASIC extracts

the library call information, building an SPL expression that describes the library call's semantics. This SPL expression is then passed to the SPIRAL system for code generation.

SPIRAL Code Generation. The captured SPL expression is sent to the SPIRAL code generation system for ASIC implementation. This SPL expression goes through a series of transformation stages within SPIRAL that implement and optimize the FFT calculation. In the first stage, a specific algorithm is selected to instantiate the FFT. As the FFT accelerator natively supports radix-8 codelets, SPIRAL specializes its algorithmic breakdown for radix-8.

After algorithm selection, SPIRAL lowers the SPL expression to a Σ-SPL expression. This expression introduces abstract loops, access patterns, and operations that will be performed in each step of the FFT calculation. SPIRAL converts these expressions into instructions for the FFT accelerator, walking the loops and access patterns to generate the load, store, and computation instructions.

These instructions are called internal code, an intermediate representation similar to other general-purpose compilers. The FFT accelerator exposes an intrinsic C library for computation offload, SPIRAL produces the intrinsic code for the FFT computation. Along with the actual operation, SPIRAL also produces the setup code to move the pointers from the host device to the accelerator, and performs memory cleanup once the operation is complete. An example generated FFT implementation is shown in Fig. 6.4.

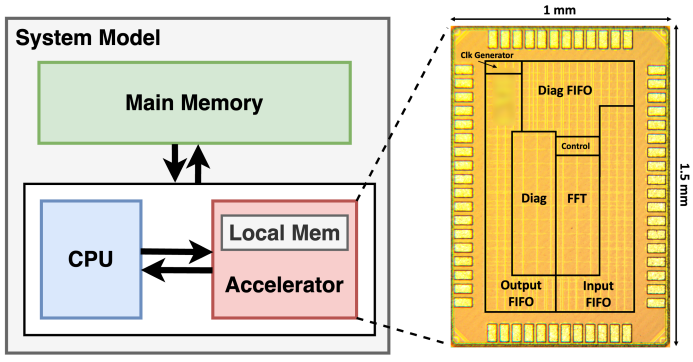


Figure 6.5: High level abstract system model of the CPU-accelerator system. The model consists of a CPU controller coupled to the accelerator, fast on-chip local memory, and main memory. The micrograph shows a silicon-verified FFT accelerator implementing a part of the architecture in Fig. 6.2.

Runtime Compilation and Execution. LibraryX-ASIC compiles the generated code into a dynamic library, which it then immediately links to. This enables access of the generated functions using the user's input and output buffers. The generated program in Fig. 6.4 consists of two parts: the host code defined on line 16 in function `dft64()` which runs on a controlling CPU and the accelerator code defined by the data structure, `dft_desc`, that is executed on the FFT hardware. To facilitate code generation targeting the accelerator, we have designed an API that enables data movement to/from on-chip local memory and main memory, accelerator invocation, and the FFT accelerator program itself. We outline the execution flow of the program on a high level abstract machine shown in Fig. 6.5. The three main components are (1) main memory with a mechanism for data transfer, (2) the accelerator which interfaces to fast on-chip local memory, and (3) a controller CPU coupled to the accelerator.

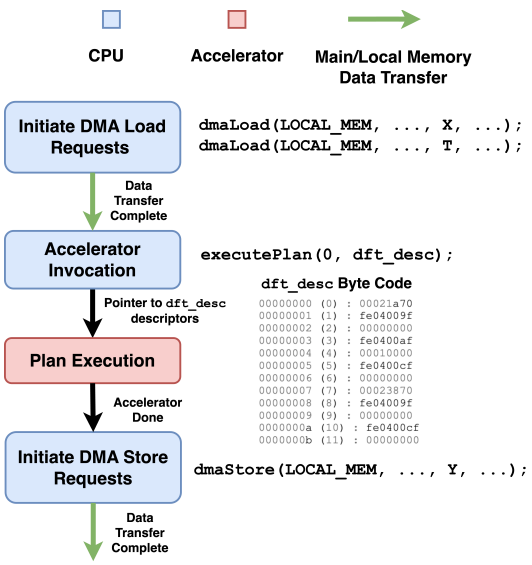


Figure 6.6: Flow chart demonstrating the execution of the generated accelerator program.

Fig. 6.6 now walks through the execution of the generated program. The program first initiates input data transfer from main memory to the accelerator's local memory with the function call to `dmaLoad()`. Once data transfer is complete, the accelerator is invoked from the CPU and passes a memory pointer to the base address of the descriptor array,

`dft_desc`. The accelerator program is defined by this array which constructs the byte code that programs the accelerator. The accelerator then fetches, decodes, and executes all descriptors in the program. Finally, the accelerator signals completion to the CPU and a DMA store request is issued to transfer output data from local memory back to main memory. At function call exit the accelerator output now exists in the output buffer.

6.5 Conclusion

We discuss the expansion of LibraryX in two key areas, new frontend programming languages, with FortranX, and as a framework for accelerator offload with LibraryX-ASIC. FortranX is designed to modernize legacy Fortran applications, interally leveraging performance portability and heterogeneity capabilities of IRISX. Using a custom compiler pass, FortranX can recognize computation patterns and pass the specification to IRISX for analysis and code generation. This optimized implementation can then execute on a variety of different hardware platforms automatically.

LibraryX-ASIC is an automated framework for software portability on custom accelerators. LibraryX-ASIC uses the semantics of library calls to recognize computations, generates an optimized implementation using the SPIRAL system, and executes the generated code on the accelerator device. Using an example of an FFT program and an FFT accelerator, LibraryX-ASIC demonstrates significant performance improvements compared to the original software library implementation without software modification.

Chapter 7

Evaluation

7.1 LibraryX Results

To evaluate LibraryX, we show LibraryX as the optimization backend for a few key domain libraries in scientific computing. These domains include FFTs/NTTs, stencils and structured grids, and sparse linear algebra. We describe our applications in each domain along with the hardware used for these experiments. We focus on results for GPU-based systems from all major vendors because of their popularity as the main computational unit on modern systems. We compare the runtime performance of our implementation against vendor libraries and state-of-the-art tools when applicable. The hardware devices we used are shown in Table 7.1. The libraries and compilers used are shown in Table 7.2.

Table 7.1: Heterogeneous systems used in this work.

GPU	**H100**	**Titan V**	**MI250X**	**Max 1100**
Vendor	Nvidia	Nvidia	AMD	Intel
#Cores	16896	5120	14080	56
Max Freq	1980 MHz	1530 MHz	1700 MHz	1550 MHz
RAM Size	80 GB	32 GB	128 GB	48 GB
Bus Type	HBM3	HBM2	HBM2e	HBM2e
Toolkit	CUDA-12.2	CUDA-11.3	ROCm-6.2.0	oneAPI-2024.02

Table 7.2: Libraries and Frameworks used in this work.

Library	Version	Compiler
cuFFT	11.7	cuda 11.4
rocFFT	1.0.28	rocm 6.2
MKLFFT	2024	oneAPI-2024.07
ICICLE	v2.8.0	cuda 12.2
Proto	main	cuda 11.4
GraphBLAST	v0.1.0	cuda 11.4
Galois	v3	cuda 11.4
LaGraph	4Jan2021	oneAPI-2022
SuiteSparse	v4.0.1	oneAPI-2022
GraphIT/G2	main	oneAPI-2022

7.1.1 Applications

PSATD. Pseudo-spectral-analytical time-domain method [101] is a computational solver in WarpX [30], an advanced electromagnetic and electrostatic Particle-In-Cell code. This algorithm approximates spatial derivatives with high-order discrete expressions to mitigate numerical dispersion in finite-difference algorithms. PSATD consists of three main operations, namely resample, FFT, and sparse-matrix vector multiplication (spmv). The Resample operation takes as input an irregular tensor, called a brick, and performs a copy to a fixed size in all dimensions, FFT of the well formed tensor to move to frequency space, half-point shift with the nth root of unity, and then inverse FFT to move back to real space with the new well-formed dimensions. This Resample is done for every brick in the input. After resampling, an FFT is performed for each well-formed brick. This FFT'd result is iterated in all dimensions, and a pencil, a vector of elements across bricks at each index point, is extracted and multiplied with a unique sparse matrix generated at each index point. An inverse FFT is applied to this result, and resampled for each brick of the output.

NTT. The Number Theoretic Transform is an algorithm for performing polynomial multiplication. Like the FFT the NTT moves between spaces, transforming a polynomial from its coefficient form to its evaluation form. This in turn reduces the time complexity of polynomial multiplication from $O(n^2)$ to $O(n \log n)$. Modern crypotgraphic

schemes, such as fully homomorphic encryption (FHE), heavily rely on polynomial arithmetic, which makes the NTT incredibly important for those computations.

3D Euler Equations. The 3D compressable Euler equations are used in the study of gas dynamics and take the form shown below,

$$\frac{\partial U}{\partial t} + \nabla \cdot \vec{F}(U) = 0. \tag{7.1}$$

A fourth-order finite volume method [69] can be used to solve equation 7.1 which involves two phases, a time integration step, and a spatial discretization step. The most computationally expensive component is the spatial discretization step and is implemented as a sequence of stencil and pointwise operations.

Triangle Counting. An easily expressible graph application is to count the exact number of triangles in a given graph $\mathcal{G}$. A mathematical specification for counting these triangles is given as

$$\Delta = \frac{1}{6}\Gamma(A^3). \tag{7.2}$$

where A is a symmetric adjacency matrix representing the input graph $\mathcal{G}$ [21] and Γ is the trace operation, the sum of the elements along the main diagonal. Libraries like GraphBLAS [51] [20] [6] implement high performance graph operations through the language of linear algebra. As A is a sparse matrix, we can use these libraries to implement triangle counting.

7.1.2 LibraryX Discussion

We show the results of the LibraryX implementation against vendor libraries or state-of-the-art tools for each application. We measure execution time using timing functions without warm-up and exclude any initial data transfers or setup operations. For applications outside of FFTs, we show results only on Nvidia platforms because they only provide Nvidia compatible implementations.

PSATD. The LibraryX implementation of PSATD is compared with two vendor library implementations in Fig. 7.1. We see that LibraryX outperforms the vendor implementation across both vendor platforms achieving speedups of 3.4x, and 2.3x respectively. There

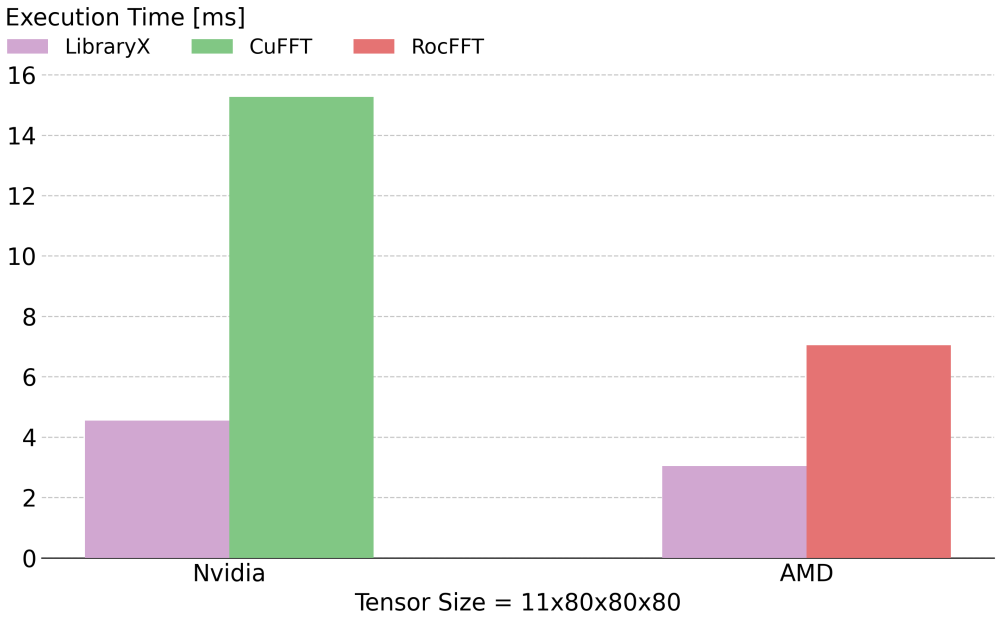


Figure 7.1: PSATD Performance Estimate between LibraryX and various vendors. LibraryX achieves estimated speedups of 3.4x and 2.3x across different vendor platforms.

are a few key reasons for this performance improvement, which stem from the ability of LibraryX to break the library call abstraction. In the resample, there is a copy followed by an FFT. This operation can be fused if the FFT library supports a guru interface for variable geometry which the vendor libraries do not support but LibraryX does. Additionally, in order to use a batched FFT call, the input bricks need to be the same dimension, as the vendors do not support variable geometry batched FFTs. The next optimization is to have a lookup table for the half-point shift omega value instead of calculating it on the fly, along with fully unrolling the spmv. Finally, when written against libraries, there are three passes through the data to perform the input half-shifts, spmv, and output half-shifts. By merging operations, LibraryX performs the computation with a single pass over the data.

NTT. The LibraryX implementation of NTTs builds on previous work to extend SPIRAL to support NTTs [98] [99]. This work enables SPIRAL to perform NTTs of arbitrary bit-width that transparently executes using the native machine word width, providing significant speedup. We compare the LibraryX implementation of the NTT to a state-of-

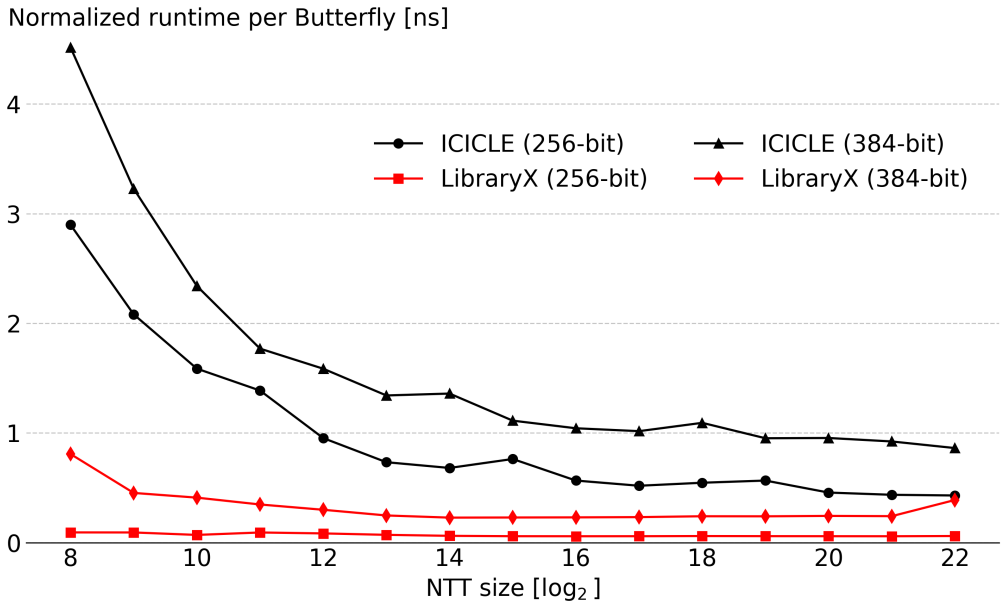


Figure 7.2: Performance comparison of NTTs between ICICLE and LibraryX on the Nvidia H100. For different bit-widths and NTT sizes LibraryX significantly outperforms ICICLE.

the-art library for high performance cryptographic acceleration called ICICLE [46]. ICICLE has a very good generalization of NTTs of varying bit widths. Figure 7.2 shows the performance of LibraryX and ICICLE for two bit widths 256 and 384 bit, for a number of different NTT sizes on an NVIDIA H100. For all sizes, LibraryX outperforms ICICLE with an average speedup of 13x for 256-bit and 4.8x for 384-bit. LibraryX enables developers to take advantage of the library interface of ICICLE while providing the performance of SPIRAL-generated NTT kernels.

3D Euler Equations. We compare the performance of LibraryX for the spatial discretization portion of the 3D Euler equations against the structured grid library Proto [3]. Proto is a domain-specific library for PDE solvers that easily expresses key operations. LibraryX leverages previous work that extends SPIRAL to support PDE solvers by optimizing the generation of their key computations, stencils, and pointwise operations [68]. Figure 7.3 shows the performance of LibraryX and Proto for Euler equations for two different input grids 32 cubed and 64 cubed on a Nvidia Titan V. LibraryX outperforms

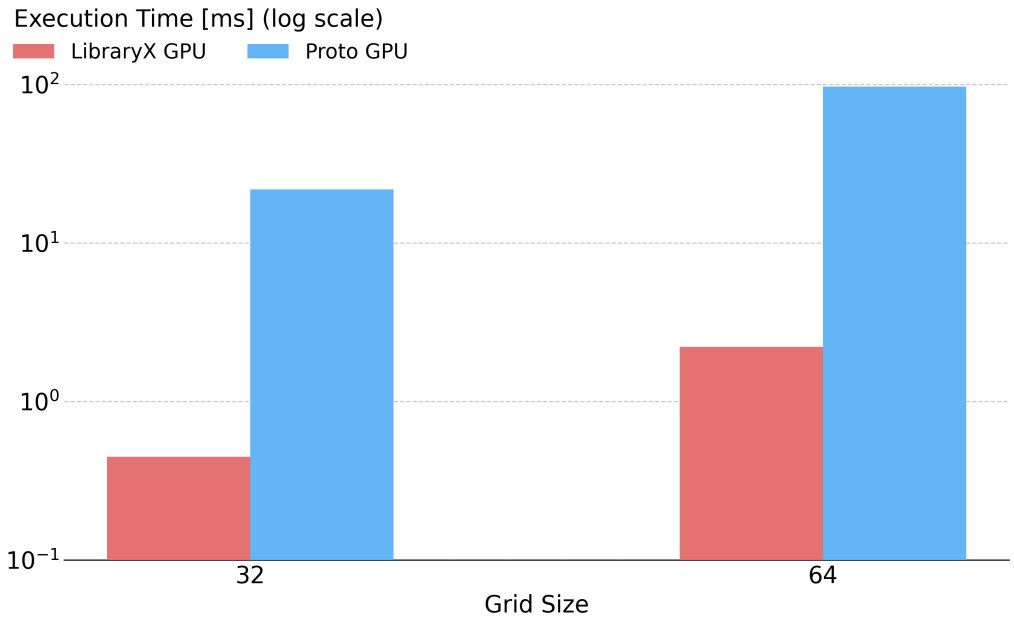


Figure 7.3: Performance comparison of the spatial discretization of the 3D Euler Equations between LibraryX and Proto on the Nvidia Titan V. LibraryX significantly outperforms the Proto implementation.

Proto by 48x and 43x for each size, respectively. The significant improvement is the result of SPIRAL being able to find opportunities for kernel fusion of pointwise kernels into stencil kernels. This reduces round-trips to global memory by removing temporaries and reduces kernel launches for different operations. We recognize that Proto is designed as a productivity library rather than a performance library, and we expect other stencil libraries to get similar performance improvements. The key impact of LibraryX is that Proto developers can take advantage of these optimizations without changing their Proto implementation.

Triangle Counting. We compare the LibraryX implementation of triangle counting against two graph processing frameworks GraphBLAST [97] and Galois [74]. The GraphBLAST framework is an extension of GraphBLAS [20] which expresses graph algorithms through the language of linear algebra. Galois is a data-parallel framework that can be applied to graph analytics applications. Figure 7.4 shows the performance across each framework on a variety of datasets. These datasets range from small graphs of $\sim$ 5 thou-

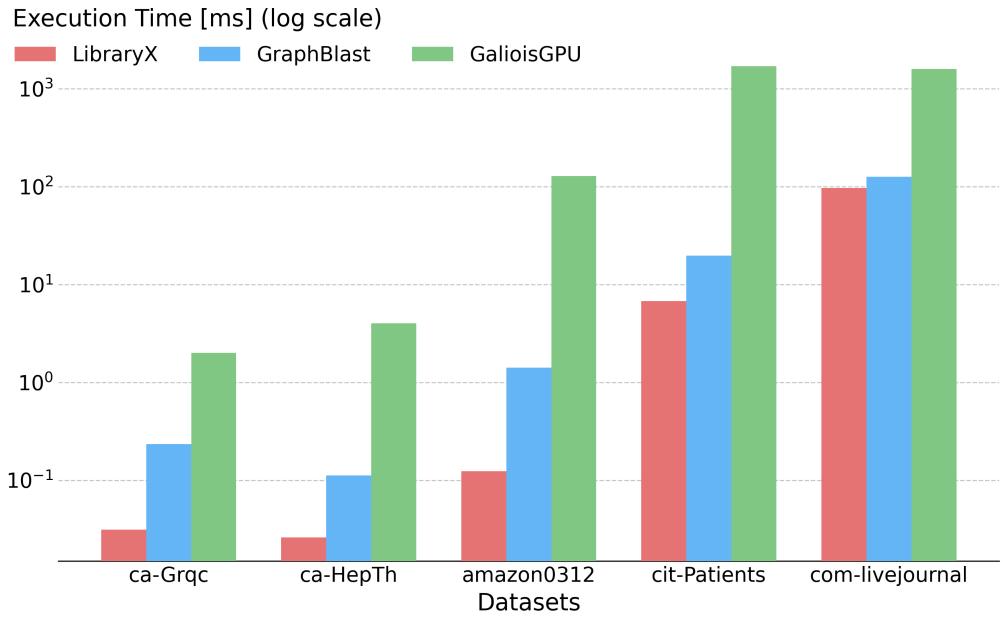


Figure 7.4: Performance comparison between state-of-the-art graph processing frameworks and LibraryX for Triangle Counting on an Nvidia Titan V. LibraryX outperforms both tools for a range of datasets.

sand nodes and $\sim$ 14 thousand edges to $\sim$ 3.9 million nodes and $\sim$ 34 million edges. We see that across the datasets LibraryX outperforms both GraphBlast and Galois. LibraryX is around 5x faster than GraphBLAST and around 200x faster than Galois. LibraryX uses SPIRAL's triangle counting implementation [83] which is based on optimizations found in previous work [61] [14]. The SPIRAL implementation parallelizes the graph over the edges rather than the vertices, performing a set intersection, the core operation, per thread rather than across threads. This results in significant performance improvements and better load balancing. LibraryX can be used by libraries like GraphBLAS to optimize algorithms such as triangle counting.

7.1.3 Explaining Performance: Hockney Freespace Convolution

We provide a detailed performance breakdown to understand why the optimizations provided by LibraryX in Hockney Freespace Convolution produce the performance benefits discussed previously. First, we show the scalability of the LibraryX implementation on

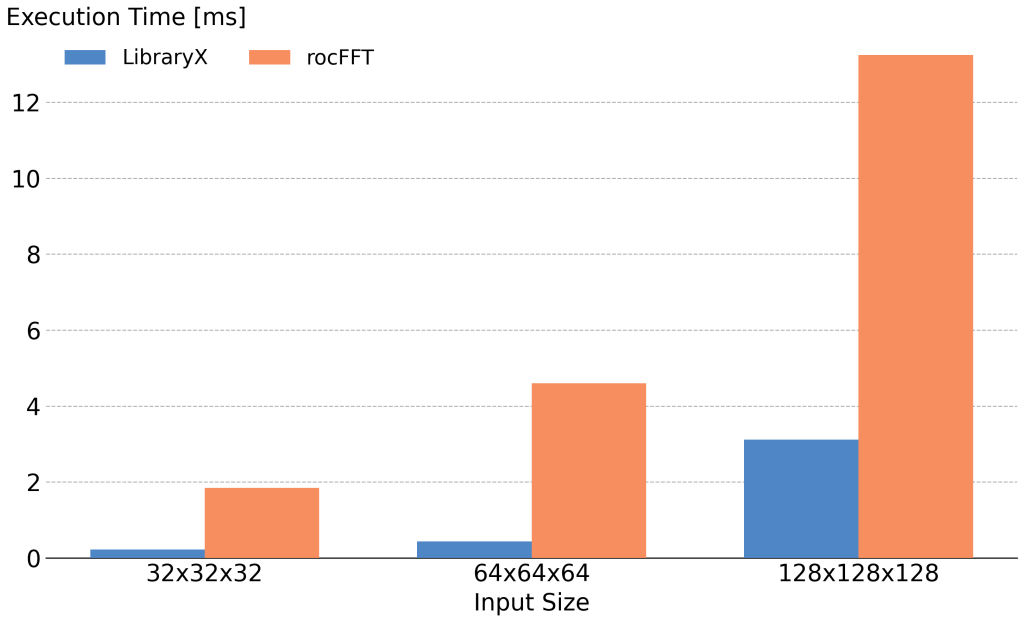


Figure 7.5: Scalability results of Hockney Convolution on an MI250X system. We see for a range of sizes that LibraryX provides significant performance improvements compared to the vendor library implementation of $\sim 8\times$, $\sim 10\times$, and $\sim 4\times$ from left to right.

an AMD system for various powers of two domain sizes compared to the vendor library rocFFT in Fig. 7.5. For smaller sizes, we see an order of magnitude improvement in performance, decreasing to $\sim 4\times$ for the largest supported size in LibraryX. Generally, this performance improvement is the result of significant memory savings through the implementation of our algorithm. The LibraryX implementation uses pruned FFTs to avoid computing on zero elements. As the input is domain doubled this results in a memory saving of $8\times$ the amount of data as the library implementation. To show this completely, we calculate the theoretical memory performance on the AMD MI250X system for the library and the vendor implementation for an input size of 128 cubed. The formula for FFT memory traffic is defined as $M = 5 * N^3 * d$ where d is the size of the data type. The library implementation must perform the complete cube expansion $N = 256$, while for the LibraryX implementation $N = 128$. This results in values of $N_{Library} = 671$MB and $N_{LibraryX} = 83$MB, an $8\times$ reduction. Dividing these values by the MI250X memory bandwidth of 3.2TB/s gives us memory times of 0.2 ms and 0.026 ms respectively, which

is the order of magnitude reduction we see for smaller sizes.

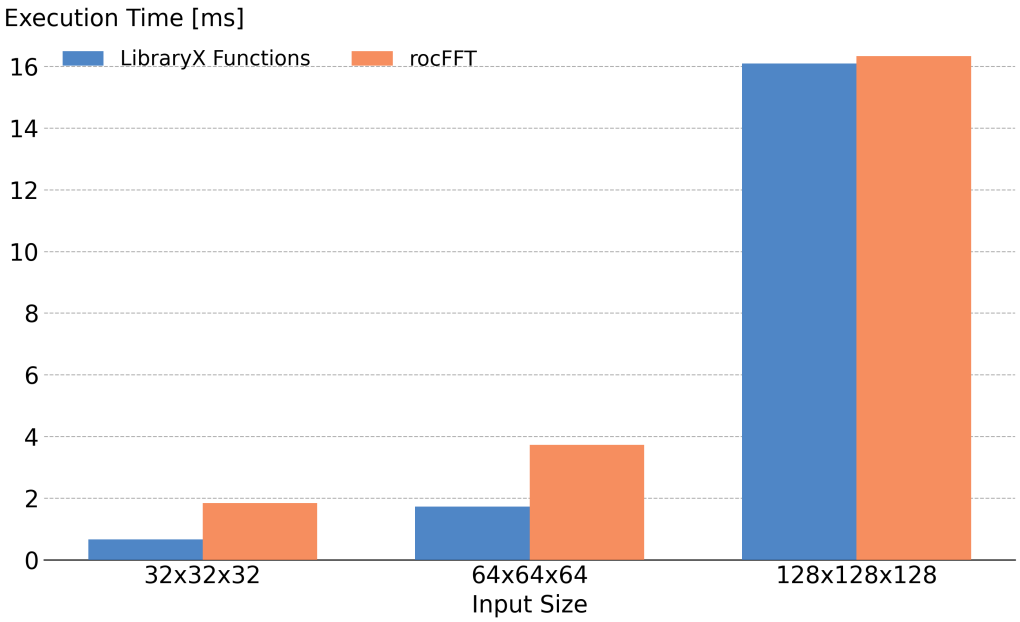


Figure 7.6: Hockney Convolution performance for 128 cubed if SPRIAL generated implementations of each of the library calls without any additional optimization. The performance is almost matching, indicating that cross-library-optimization is critical for performance.

Although the theoretical performance holds for smaller sizes, for the 128 cubed case our performance is only $\sim 4\times$ better. The reason we are not seeing $\sim 10\times$ performance improvements is because the vendor FFT implementation, the main bottleneck of this computation, performs better than the LibraryX implementation for larger FFTs. More work needs to be done to improve the compute performance of larger FFTs to get closer to theoretical performance improvements. Digging into this further, if we call the SPIRAL generated implementations of each of the library calls without cross-library optimization, as shown in Fig. 7.6, we see that the performance gap reduces as we increase the problem size, eventually becoming equal. This showcases the real benefits of cross-library-call optimization; just replacing library calls with better implementations does not give the same performance improvements. Each library can finetune their implementation at various sizes to outperform the others.

Table 7.3: Dataset Description Table [62] [26]

Dataset	Vertices	Edges	Diameter
ca-GrQc	5,242	14,496	17
ca-Hepth	9,877	25,998	17
amazon0302	262,111	1,234,877	32
Cit-Patents	3,774,768	16,518,948	22
com-livejournal	3,997,962	34,681,189	17

7.2 More GBTLX Results

We compare the GBTLX implementations of Triangle Counting, Direction Optimizing Breadth-First Search, and Betweeness Centrality on a variety of different input graphs ranging from small graphs with ∼5,000 nodes and ∼30,000 edges to large graphs with ∼4 million nodes and ∼40 million edges [62] [26]. A full description can be seen in Table 7.3. None of the graphs were pre-sorted and all graphs are made symmetric and undirected. All CPU experiments were run on an Intel Skylake architecture with 4 cores and 8 threads. We use the Intel icpx c++ compiler provided in the Intel oneApi BaseToolkit version 2022.1.2 with OpenMP for parallelization across all CPU experiments. All GPU experiments were run on an Nvidia Titan V GPU with 80 Streaming Multiprocessors. We use the cuda-11.3 toolkit for the nvcc compiler.

7.2.1 Parallel CPU Results

Figures 7.7, 7.8 , and 7.9 show the performance of GBTLX compared to a wide range of different graph libraries and frameworks. We begin with the Triangle Counting performance seen in Figure 7.7. GBTLX has significant performance improvements over the other graph frameworks as the graphs increase in size. For small graphs, GBTLX is 3x faster than Galois, while being 1.2x slower than GraphIT and LAGraph. Moving to larger graphs, GBTLX is at least 3x faster than every other framework. We again attribute this performance improvement to the algorithmic implementations for triangle counting dis-

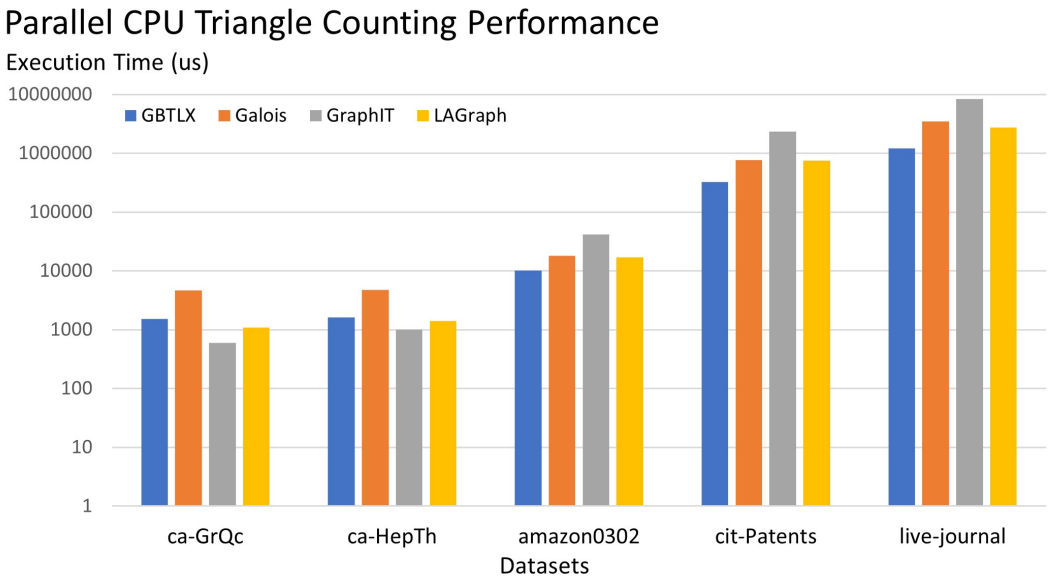


Figure 7.7: Parallel CPU performance of Triangle Counting across frameworks. The y-axis is log scale.

cussed earlier.

Direction Optimizing Breadth First Search performance is shown in Figure 7.8. This application experiences some performance slowdown when compared to all other frameworks. For small graphs, we see a 1.7x slowdown, while for larger graphs, it is closer to 1.2x. Unlike Triangle Counting, getting performance for Breadth First Search requires specific data structure support and deep knowledge of parallelization frameworks. We noticed in our testing that for small graphs, the overhead of the OpenMP parallelization library is double the actual execution time of the serial code for our implementation. In the event that this time could be ignored or a serial implementation could be run for small graphs, our implementation would be very competitive with both Galois and LAGraph.

As we move towards larger graphs the our implementation is within the same order of magnitude when compared to the other frameworks except for GraphIT. We attribute our slowdown for large graphs to a lack of specific thread-safe data structures such as a bitvector and a queue used in pull and push respectively. We expect our performance to rival the other frameworks with the introduction of customized data structures.

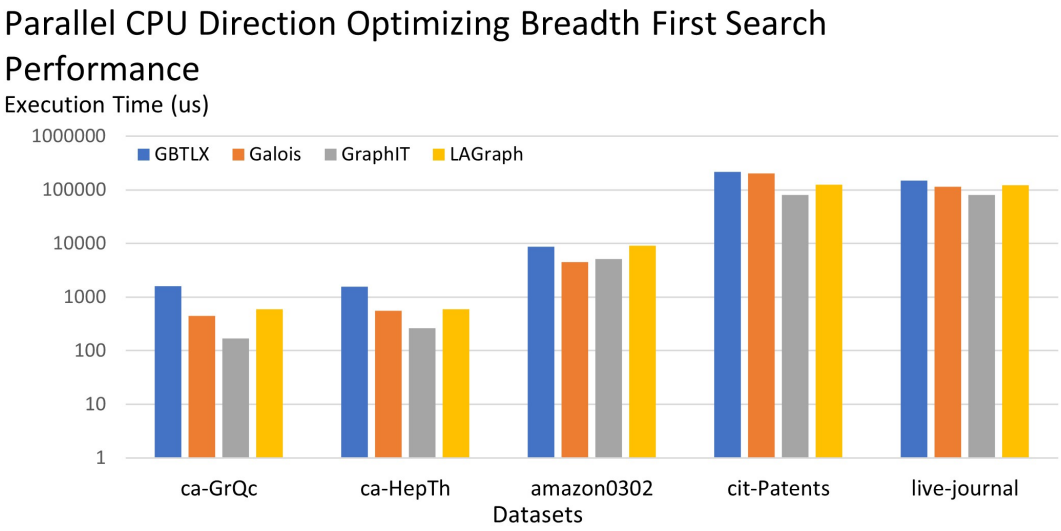


Figure 7.8: Parallel CPU performance of Direction Optimizing Breadth First Search across frameworks. The y-axis is log scale.

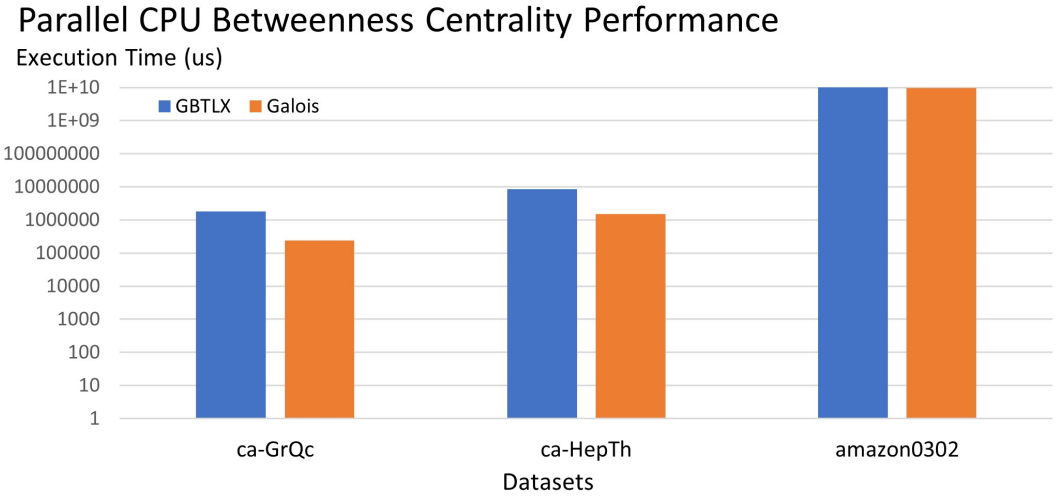


Figure 7.9: Parallel CPU performance of Betweenness Centrality between GBTL and Galois. The y-axis is log scale.

Figure 7.9 shows the comparison between our implementation of Betweenness Centrality and Galois. Unfortunately, both GraphIT and LAGraph only perform the algorithm on a subset of nodes, not the entire graph, so we were unable to include them in this comparison. Again, we see a slowdown of our algorithm compared to the Galois implementation. As a more complex algorithm, it is clear that performing purely library call

fusion is not enough to give performance improvements rivaling that of a more mature parallelization framework like Galois, and further research is required. One such area is to perform a batched version of Betweenness Centrality. In this implementation, the outer loop is unrolled so that multiple vertices path counts and betweenness scores are updated together. However, this incurs additional space overhead, so it must be carefully orchestrated. In addition, Betweenness Centrality can be implemented in such a way that it uses both push and pull implementations for the forward and backward stages. This is something we would like to explore, but at this time, we only use the push implementation for both phases. This will also affect our parallelization scheme when using both push and pull phases.

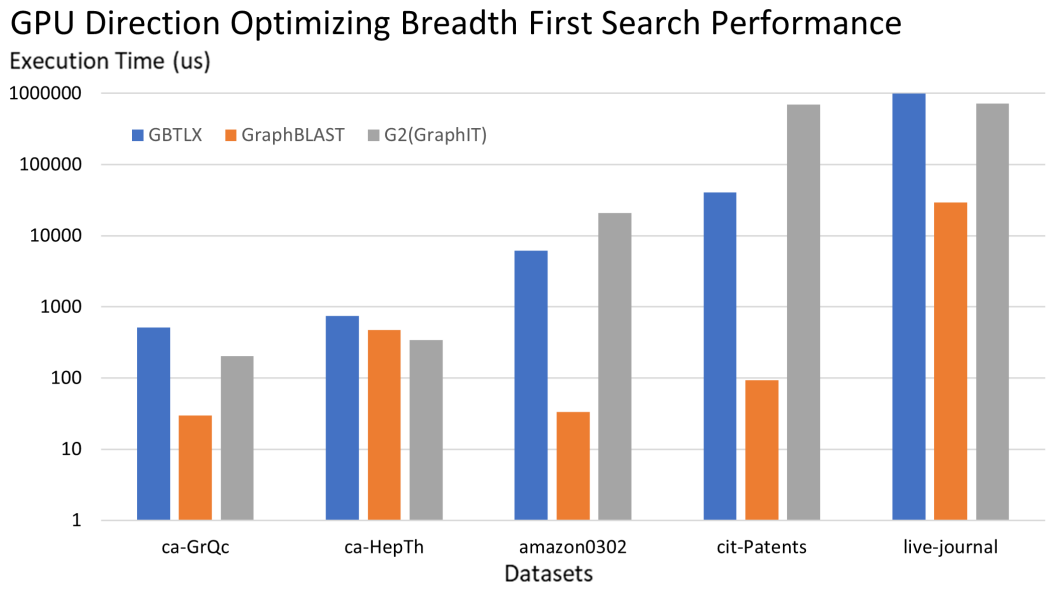


Figure 7.10: GPU performance of Direction Optimizing Breadth First Search across frameworks. The y-axis is log scale.

7.2.2 GPU Results

Figure 7.10 shows the performance of Direction Optimizing Breadth First Search. For small to medium-sized graphs, we are competitive or achieve speedup compared to G2 [18], the GPU extension to GraphIT, but both frameworks fall short of GraphBLAST.

Unlike both frameworks, GraphBLAST is a hand-optimized framework for graph processing and has some specific optimizations that neither of the automatic frameworks currently support. Furthermore, we see a significant slowdown of our code relative to G2 for the largest graph. We attribute this to a synchronization overhead and a lack of load balancing for very large graphs. Both G2 and GraphBLAST have very specific techniques to avoid using grid-level synchronization, a design decision made in GBTLX.

Finally, for Betweenness Centrality we were not able to compare to the other frameworks mentioned. Only G2 has a Betweenness Centrality implementation, but it again only goes over a subset of nodes in the graph rather than the entire graph, which our algorithm is tuned to do. Our implementation parallelizes vertices across the sets of cores, computing the Betweenness Centrality score on a per core basis, rather than using all cores for one vertex. This parallelization scheme was chosen to reduce the number of grid-wise synchronization calls. More research needs to be done to support per vertex Betweenness Centrality utilizing the full GPU architecture.

7.3 More ProtoX Results

In Fig. 7.11, we provide a comparison of the run time between Proto and ProtoX for the 2D Poisson equations. We compare three different box sizes - $64 \times 64, 128 \times 128$ and 256×256. We keep the number of boxes fixed to 4×4. This makes the corresponding domain sizes of the problem as $256 \times 256, 512 \times 512$, and 1024×1024, respectively. We ran both the Proto code and the ProtoX code for 100 iterations. We can observe from the graph that ProtoX performed up to $2\times$ faster than the base Proto code for the 2D Poisson problem. These results were obtained on a local CPU machine with 2.3GHz Quad-core Intel i7 processor.

In addition to the 2D Poisson equation we also compared the ProtoX implementation of the 2D Euler equations for the serial CPU case. In Fig. 7.12 we see that we gain up to $8\times$ speedup over the base Proto code. The reason for this performance improvement in both problems is due to ProtoX kernel fusion capabilities outlined previously. By fusing pointwise and stencil operations, ProtoX is able to significantly reduce the memory

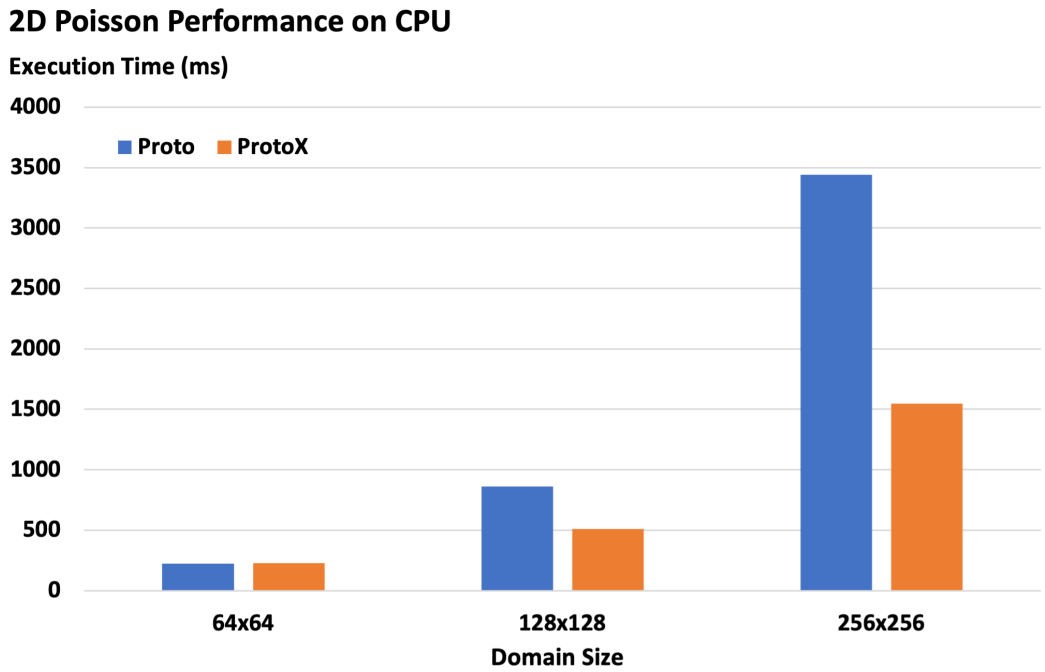


Figure 7.11: Run time comparison between the reference Proto code and the ProtoX code generated code for CPU. Here we are comparing different `Box` sizes ranging from 64×64 to 256×256 for a fixed 100 iterations. We can observe that the ProtoX is performing up to $2\times$ faster than the base Proto code.

footprint of its implementation. Combine this with traditional compiler optimizations and memory pooling results in such a large performance win.

As shown in the previous LibraryX results section, ProtoX also supports the 3D Euler equations on GPU systems. For this case we also compared the callback approach on GPU. This case is more general than the other tested cases as ProtoX doesn't have to ingest the pointwise functions. Instead, it calls the functions as attached kernels or standalone kernels depending on if fusion is available. As shown in Fig. 7.13, ProtoX still achieves good performance with callbacks compared to the base implementation of Proto on GPU, with up to $15\times$ speedup. This speedup is the result of being able to reduce the kernel launch overhead for kernels that are fused within ProtoX, a significant performance inhibitor for small box sizes.

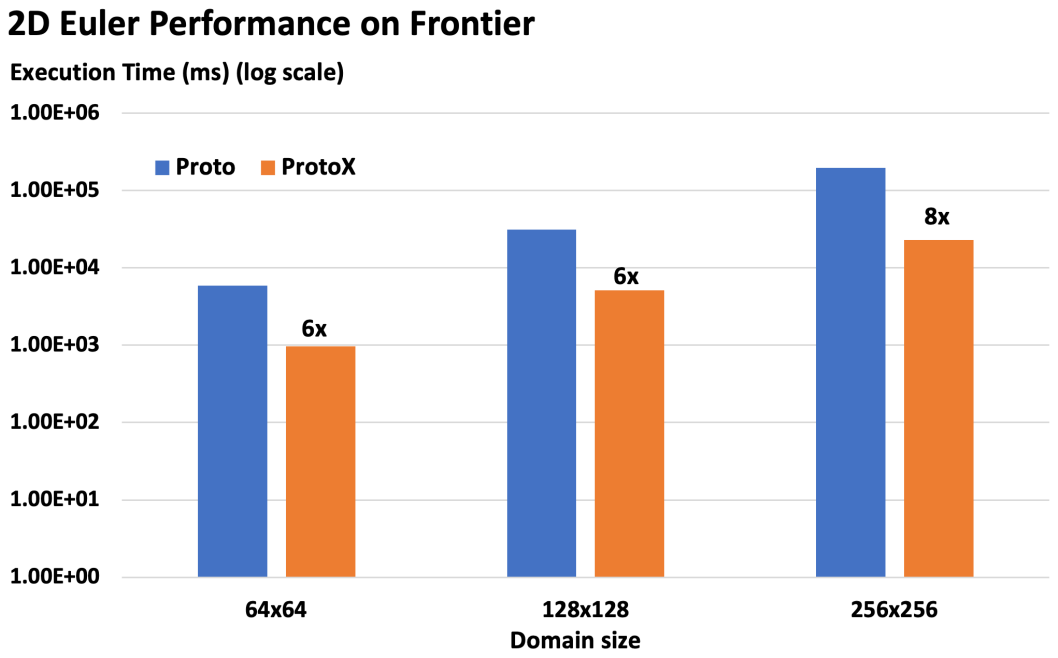


Figure 7.12: Performance comparison between the Proto and ProtoX implementations of the 2D Euler Equations on CPU.

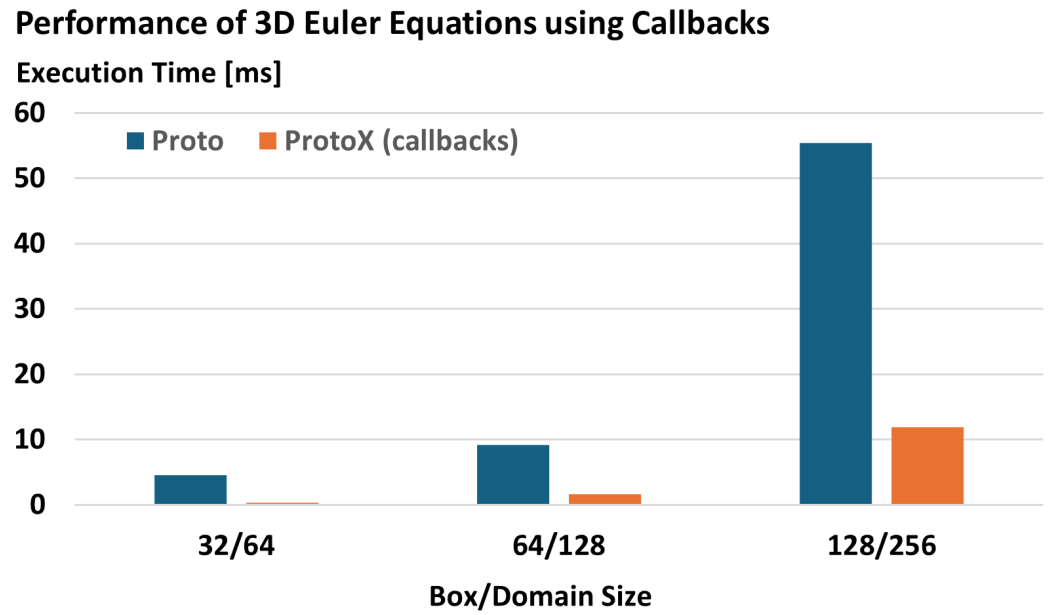


Figure 7.13: Performance comparison between the Proto and ProtoX implementations of the 3D Euler Equations on GPU using function call backs.

Table 7.4: Heterogeneous systems used in this research.

System	GPUs	CPU	Compiler	Vendor Runtime
Aurora	Total 6 GPUs 6× Intel Data Center GPU Max 1550	Intel Xeon CPU Max 9470C, 52 cores	ICPX-2025.0.0	oneAPI-2024.07
Frontier	Total 8 GPUs Total 4/8 AMD 250X GPUs/GCDs	AMD EPYC 7702, 128 cores	Clang-17.0.0	ROCm-6.2.0
Equinox	Total 4 GPUs 4× NVIDIA V100	Intel Xeon CPU E5-2698, 20 cores	GNU-11.4.0	NVHPC-24.3
Zenith	Total 2 GPUs 1 Nvidia GTX 3090 1 AMD Radeon RX 6800	AMD Ryzen Threadripper 3970X, 32 cores	GNU-11.4.0	NVHPC-24.3 ROCm-6.2.0
Cades Cloud	Total 8 GPUs 4× NVIDIA A100 4× AMD MI100	AMD EPYC 7763, 128 cores	GNU-8.5.0	CUDA-11.7 ROCm-5.1.2
Milan	Total 2 GPUs 2× NVIDIA A100	AMD EPYC 7513, 64 cores	GNU-11.4.0	NVHPC-24.3

7.4 IRISX

7.4.1 Heterogeneous Systems

To evaluate the portability of IRISX for performance in heterogeneous systems, we consider the heterogeneous systems in Table 7.4. To ensure a wide range of architectures, we considered Frontier, which has eight AMD MI250X GPUs, Aurora with 6 Intel Data Center GPUs, and Equinox node from ExCL [2], which has 4 NVIDIA V100 GPUs. To represent a case of diversity in node heterogeneity, we considered Cades, a cloud system with four Nvidia A100 GPUS and four AMD MI100 GPUs. Similarly, the Zenith node in ExCL is used to represent in-node heterogeneity with two different consumer-grade GPUs, Nvidia GTX 3090 and AMD Radeon RX 6800. The considered systems represent different architectures in multi-accelerator and multi-vendor environments used to test IRISX's performance portability for diverse heterogeneity.

7.4.2 Library and Application: Proto and the 3D Euler Equations

Proto. In this work we are able to target a class of motifs [9], namely the operations on a structured grid, that appear in multiple PDE-based applications, with the help of Proto [3]. It's a C++ library that focuses on providing optimized numerical approximations to various PDE based model problems. It provides high-level of abstractions for many popular numerical methods like finite difference (FDM), finite volume (FVM) and finite element (FEM) along with multigrid and adaptive mesh refinement (AMR) methods, that are used to solve PDE models numerically on structured grids. A broad range of operators involved with these methods can be represented as a composition of stencil and pointwise operations. For example, the Poisson equation, which is a fundamental equation in the field of fluid dynamics, is mathematically represented in the case of 2 dimensions (2D) as shown in equation 4.2, where one can use the Jacobi method to iteratively compute the solution of (4.2). Here, the 2D Laplacian $\Delta\phi$ is computed as a 5-point stencil obtained via FDM,

$$\Delta\phi_{i,j} = \frac{\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j}}{h^2}, \tag{7.3}$$

where h is the grid spacing used. The Jacobi iterative formula is given in equation 4.3.

One can represent this computation in Proto with approximately the same number of lines of code as the mathematical formulation for this problem. This is done using the `Stencil::Laplacian()` function within the `Stencil` class with appropriate shifts and weights associated with the 5 - point Laplacian stencil (weights are the coefficients (1, 1, 1, 1, -4) from (7.3)) and the Jacobi iteration can be done in a pointwise fashion with the help of the `forall(func, args, ...)` function in Proto.

This example shows that one can use multiple combinations of stencils (like the 3D 7 - point Laplacian stencil, 27- point Laplacian for higher orders, stencils for computing flux divergence and many more) and pointwise functions (different types of `forall()` functions) available in Proto to compute complex PDE applications that are based on the structured grid motif. Hence, by showcasing a capability to work with such a library provides us with an opportunity to target applications like computational fluid dynamics,

magnetohydrodynamics, particle methods, and many more. In this work, we have used an extension of Proto called ProtoX [68], which is a domain specific language with Proto as its front end and SPIRAL as its back end.

3D Euler Equations. The application in Proto that we are using is the 3D compressible Euler equations, which are used in the study of gas dynamics [63, 70]. It is a hyperbolic conservation equation which is expressed in Equation 7.1, where $U = (\rho, \rho\vec{u}, \rho E)$ represents the conserved quantities of mass, momentum and energy. Here ρ is the mass density, $\vec{u} = (u_x, u_y, u_z)$ is the velocity and E is the energy per unit mass. A fourth-order accurate FVM as derived by McCorquodale et al. in [70] is used to solve (7.1) in Proto. The algorithm is divided into two main parts-

1. Temporal discretization

   - For a given solution $\langle U \rangle (t^n)$ of (7.1) at time t^n, the solution for the next time step $\langle U \rangle (t^n + \Delta t) = \langle U \rangle (t^{n+1})$ is computed using the fourth order Runge-Kutta (RK4) scheme as

     $$\langle U \rangle (t^n) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O((\Delta t)^5), \tag{7.4}$$

     where for each $j \in [1,4], k_j = -\frac{d\langle U \rangle^{(j)}}{dt}\Delta t$ and $\langle U \rangle^{(j)} = \langle U \rangle^{(0)} + \frac{1}{2}k_j$ with $\langle U \rangle^{(0)} = \langle U \rangle (t^n)$.

2. Spatial discretization

   - Method-of-lines approach is used in [70], where fourth-order quadratures in space are used to compute the flux integrals.

The solution is computed in an iterative fashion where each k_j in the RK4 computation requires one to go through the spatial discretization step. Being a fourth order accurate solution, the algorithm to compute the spatial discretization is much more compute intensive compared to the time integration part of the overall algorithm. The DAG used to solve the spatial part in Proto is shown in Fig. 5.3-a. The same figure also indicates the sequential manner in which Proto computes the given problem. SPIRAL is used to

generate the architecture optimized code only for the spatial discretization part of this problem. Moreover, this application provides an opportunity to test the various fusion combinations available in the IRIS runtime system. We focus on 32, 64, 128, and 256 size `Boxes` or subdomains in three dimensions, with a range of domain decompositions.

Table 7.5: All DAG configurations in IRISX.

Configuration	**Concurrency**	**Task Overhead**	**Async**
DAG Serial	Least	Middle	Yes
DAG Fusion	Most	Most	Yes
Task Fusion + DAG Fusion	Middle	Least	No

7.4.3 DAG Configurations

Table 7.5 highlights all the configurations tested in the experimentation for the 3D Euler equations example. These configurations are divided into three groups, `DAG Serial`, `DAG Fusion`, and `Task Fusion + DAG Fusion`. Within each group is the option of configuring various levels of `Kernel Fusion`. Each group has three major characteristics: the amount of concurrency, the amount of task overhead, and the support for asynchronous execution. With asynchronous execution enabled, the IRIS runtime uses ten streams (CUDA and HIP) or device queues (OpenCL) per GPU to concurrently execute the kernels, with three streams reserved for data transfers. However, the `Task Fusion` case cannot take advantage of the asynchronous tasking features of IRIS since IRISX fuses all tasks in a single task, as shown in Fig. 5.3-c.

For each of the machines, Frontier, Aurora, and Equinox, we tested seven different combinations of box and domain sizes for the 3D Euler equations application. For each box and domain, we choose four GPU configurations for Frontier, Aurora, and Equinox, resulting in 84 different values for each of the three DAG configurations across the three systems for different box and domain sizes. There are a few missing box and domain sizes, for which the value was not generated due to system memory limits.

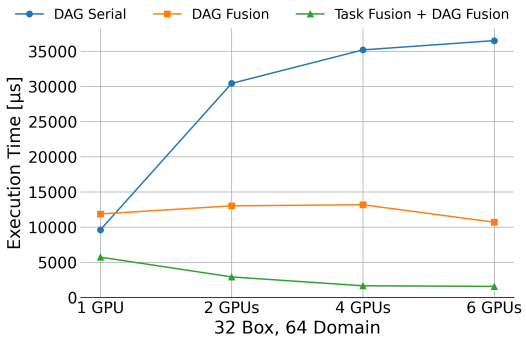
7.4.4 Custom Scheduling

To maximize parallelism and minimize data movement, IRISX schedules individual DAGs on different GPUs for the DAG configurations `DAG Fusion` and `Task Fusion + DAG Fusion`. Since the boxes do not share any dependencies as shown in Fig. 5.3-b, and -c, scheduling the DAGs/Boxes on different GPUs ensures minimal data transfers. There exists only an initial host to device transfer and another device to host transfer after the DAG completes. The `DAG Serial` does not exhibit such inter-DAG/Box concurrency, therefore, the default IRIS dynamic scheduler (first to finish) is used, which incurs device to device transfers.

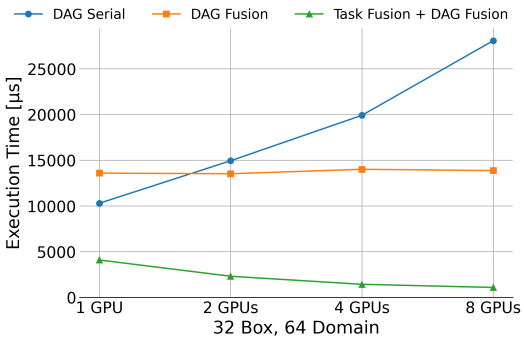
7.5 IRISX Results and Discussion

This section evaluates IRISX for different DAG and kernel configurations on various machines while scaling on various numbers/kinds of GPUs. All experiments were performed without the need to modify the source code. After compilation, the number of GPUs is changed using environment variables, showcasing IRISX's functional portability. The purpose of this section is to demonstrate two key properties of IRISX. The first is to demonstrate how different DAG representations perform when the input size, architecture, and number of devices are changed, demonstrating the need for a dynamic system like IRISX that can find the best-performing DAG configuration. The second is to show that varying the size and number of kernels further impacts the performance of the computation and can aid in discovering functionality. The rest of the section discusses these two areas in detail and includes 1) impact of DAG configurations for small- and large-sized kernels, 2) impact of kernel fusion and functionality discovery, 3) multi-vendor scalability, 4) comparison with Proto library, and 5) overhead and limitations.

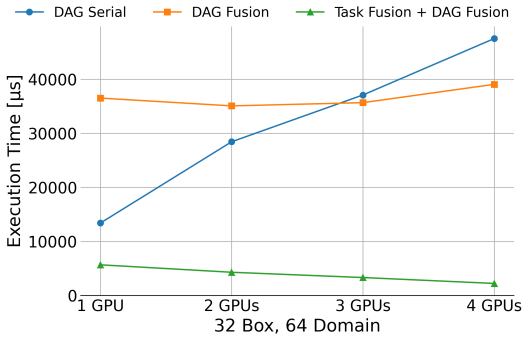
All results are presented as the execution time of the task graph. The 84 combinations (# of GPUs configurations * # of machines * # of Box and Domain combination) are run 10 times for each of the three DAG configurations. Even though we performed experiments using nine Box and Domain sizes, three of them are shown, as they showcase all performance variations.



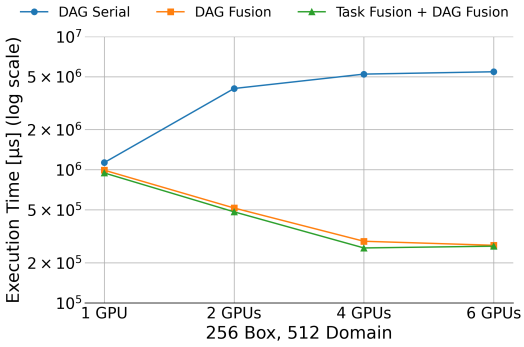
(a) Aurora-32



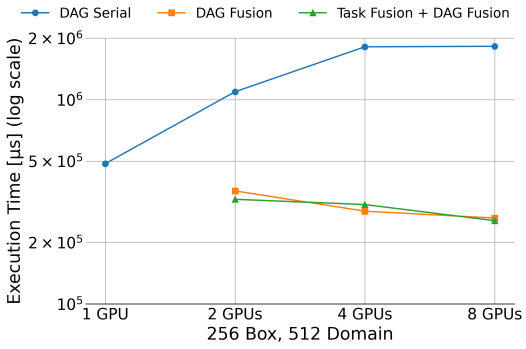
(b) Frontier-32



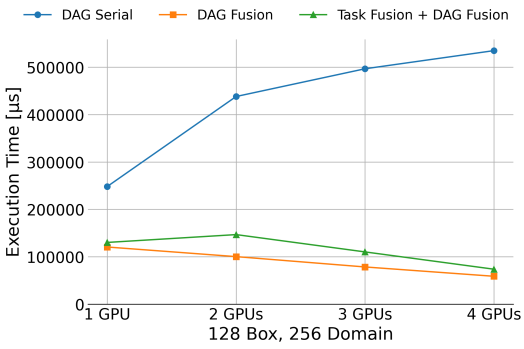
(c) Equinox-32



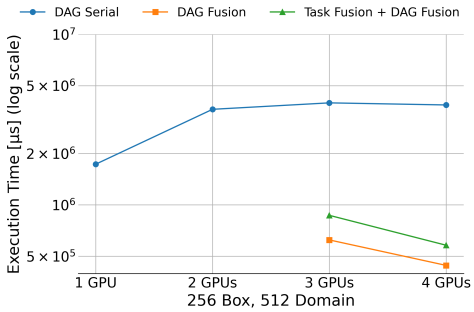
(d) Aurora-256



(e) Frontier-256



(f) Equinox-128



(g) Equinox-256

Figure 7.14: Results for various box and domain sizes across Aurora, Frontier and Equinox, for each DAG configuration. The first three graphs (read left to right then top down) shows the results for a small Box Size (32). The second three graphs shows the results for a large Box size (128 and 256) on modern supercomputers. The last graph shows large Box size on older hardware.

7.5.1 Task Graph Level Representation

Figure 7.14 shows a handful of key results when varying DAG configurations in IRISX. Figures 7.14a, 7.14b, 7.14c, show the performance for graphs with smaller kernels, while Figs. 7.14d, 7.14e, 7.14f, and 7.14g show the performance for graphs with larger kernels (kernels sizes vary due to box sizes). The results presented show $2 \times 2 \times 2$ decompositions. There are three groups of DAG configurations, `DAG Serial`, `DAG Fusion`, and `Task Fusion + DAG Fusion` (shown in Fig. 5.3-a, -b and -c). The `DAG Serial` group has the least concurrency, where IRISX creates a task graph for each box that runs serially. The `DAG Fusion` group has the highest concurrency, as it is able to use multiple GPUs for different boxes and CUDA and HIP streams (Frontier and Equinox) and OpnenCL device queues (Aurora) simultaneously for each box. Finally, the `Task Fusion + DAG Fusion` group trades off higher concurrency for less task overhead and can only leverage inter-GPU concurrency because of custom scheduling without supporting multi-stream execution. We discuss in more detail the variation of Figure 7.14 by Box size.

Impact of GPU computation Power for Small Boxes. Figures 7.14a, 7.14b, 7.14c, show the performance of a Box of size 32 on each of the machines — Aurora, Frontier, and Equinox. For all three figures, we see that `DAG Serial` provides worse performance as we increase the number of GPUs. This is because of the increased device to device data movement as DAGs for a single Box are scheduled on multiple GPUs. For `DAG Fusion` we see a small amount of scalability for larger numbers of GPUs as seen in Aurora, but mostly remains either flat or performs worse as we increase the number of GPUs as seen on Frontier and Equinox respectively. To understand why we are seeing this almost flat trend line, we used NVIDIA's Nsight system and AMD's Rocprof to quantify the task and kernel management overhead. Through profiling, we found that the summation of data movement and kernel execution time is insignificant compared to the total execution time. Most of the execution time is spent performing runtime orchestration for task and kernel management. Therefore, even with more GPUs there is little scalability because any increase in parallelism is overcome by the overhead. `Task Fusion + DAG Fusion` sig-

nificantly reduces this overhead, which is why there is scalability across the machines for this DAG configuration. For this small size, task and kernel management/orchestration overhead is significantly reduced by fusing all the tasks and their kernels for the Box into a single task.

It is important to note that the gap between `DAG Fusion` and `Task + DAG Fusion` widens significantly as we move from Fig. 7.14a to Fig. 7.14c. This is because of the relative computation power of each GPU for this type of computation. Generally, the newer GPUs in Aurora and Frontier have less kernel launch overhead; hence the gap is smaller than Equinox with V100 GPUs. Moreover, Aurora's GPU has the strongest double precision performance; therefore, it has the smallest gap between `DAG Fusion` and `Task Fusion + DAG Fusion`. Meanwhile, the opposite can be said for Equinox, as the GPU is much older.

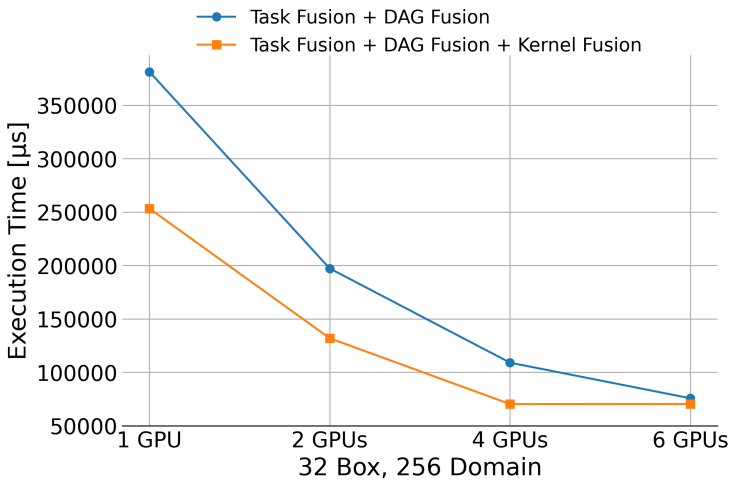
Impact of Multi-stream Execution for Large Boxes. Figures 7.14d, 7.14e, 7.14f, 7.14g show the scalability of larger boxes on each of the machines, Aurora, Frontier, and Equinox. Just like small Boxes, `DAG Serial` exhibits no scalability due to significant device to device memory transfers per Box. In Aurora and Frontier, we see that both `DAG Fusion` and `Task Fusion + DAG Fusion` provide scalability as we increase the number of the GPUs, with `Task Fusion + DAG Fusion` being better in Aurora and alternating slightly on Frontier before converging. In contrast, on Equinox for both Box sizes 128 and 256 `DAG Fusion` was considerably faster than `DAG Fusion + Task Fusion`. This is the result of two things: the ratio of the kernel size to the relative GPU computation power, and the effect of multi-stream execution. As these box sizes are significantly larger than the first three graphs, the execution time per kernel is significantly longer. This enables better usage of multi-stream execution, allowing better utilization of the GPU as different boxes can be executed concurrently on multiple streams. Figures 7.14d and 7.14e also use multi-stream execution but the GPUs on these machines are significantly more powerful than the ones in Equinox, especially for double precision calculations. For this reason, there is not as much opportunity to have overlapping streams because the kernels themselves complete so quickly.

General Statistics Across Configurations. Of the 84 configurations we tested, 60 used small Boxes, 32 and 64 Box sizes, while 24 used larger boxes, 128 and 256 Box sizes. For small Boxes, `Task Fusion + DAG Fusion` performed the best in 66 configurations, while `DAG Fusion` performed the best for 2 configurations across the three machines. For large Boxes, `Task Fusion + DAG Fusion` performed the best in 8 configurations, while `DAG Fusion` performed the best in 13 configurations across the three machines. Therefore, for different architectures with varying computation power, number of devices, and input sizes, statically selecting one task graph representation could result in a significant performance loss depending on problem size, demonstrating the necessity of IRISX which can automatically adapt the best performing graph representation.

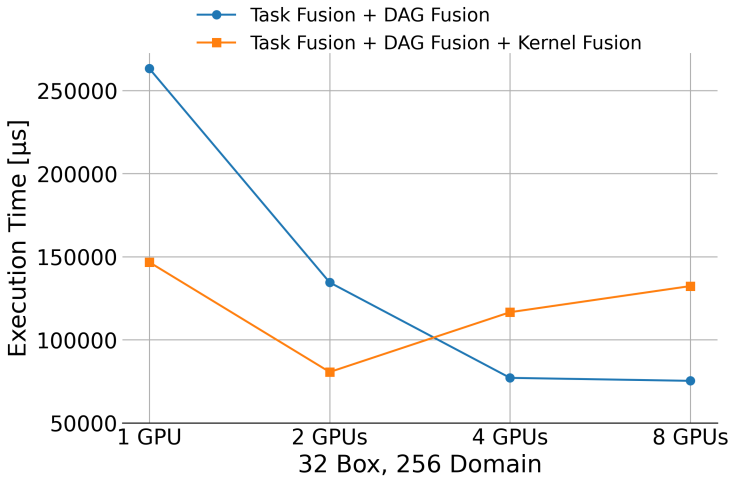
7.5.2 Kernel-Level Representation and Functionality Selection

Not only can IRISX dynamically select DAG configurations, IRISX can enable dynamic interaction between IRIS and SPIRAL to vary the degree of kernel fusion provided by SPIRAL. This changes the number of kernels and the relative computation per kernel, the amount of temporary memory required, and the overall concurrency for a given computation. Generally, a larger number of kernels indicates lighter kernels (fewer operations) with more concurrency and larger memory footprint, while a smaller number of kernels indicates heavier kernels (more operations) with less concurrency and smaller memory footprint. We discuss how introducing kernel fusion impacts the variability of the Euler equations applications.

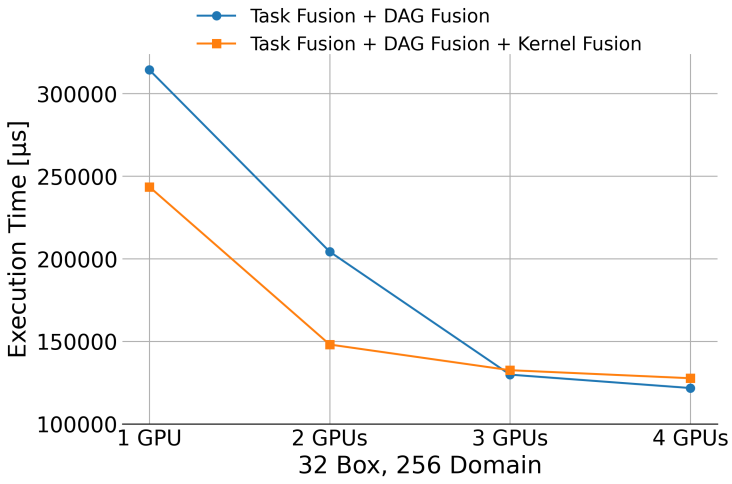
Varying Kernel Implementations. Figure 7.15 shows the performance comparison when adding the maximum kernel fusion available in IRISX. For each of the machines tested we compare the maximum kernel case of 19 kernels with the maximum fusion case of 5 kernels (the first and last DAG in Fig. 5.3-d), for the DAG configuration `Task Fusion + DAG Fusion` with a Box size of 32 and a Domain size of 256. We see an interesting trend on Frontier and Equinox, where initially the introduction of kernel fusion increases performance. This aligns with the general trend that introducing more fusion results in better performance for small box sizes. However, as the number of GPUs increases, there



(a) Aurora



(b) Frontier



(c) Equinox

Figure 7.15: Effects of Kernel Fusion on the performance variability for various machines for the 32 Box Size and 256 Domain size with DAG Configuration Task Fusion + DAG Fusion. As the number of devices increases lighter kernels perform better.

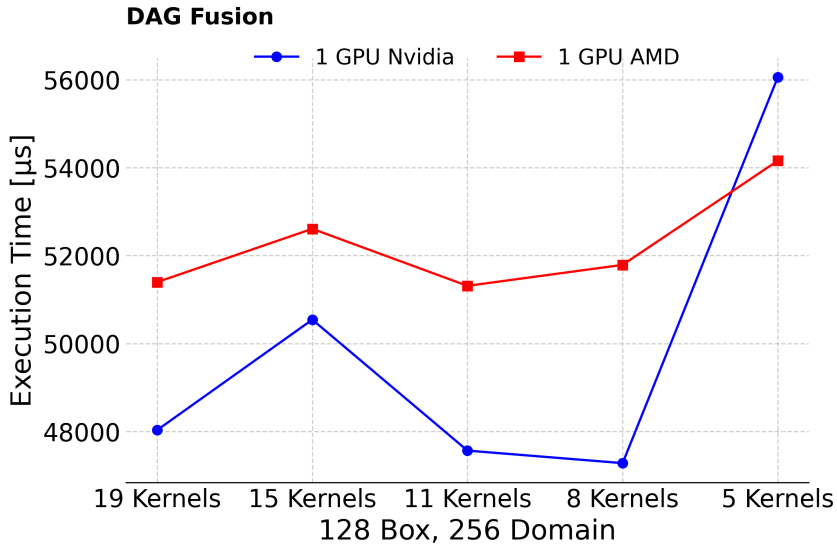


Figure 7.16: Different kernel variants for the Euler equations run on consumer-grade GPUs present in the Zenith node of ExCL. The 19 kernel case represents lighter more concurrent kernels while the 5 kernel case represents heavier less concurrent kernels.

is a cross-over point where kernel fusion provides worse performance due to the increased computation per individual kernel. This increased computation is good when there are less opportunities to schedule kernels as there are less devices to utilize, but more devices enable parallel execution that can amortize the overhead. On Aurora, we believe this trend will be visible if we were able to extend beyond six GPUs.

Our results so far indicate that there exists a profound relationship between kernel size and performance variability. Figure 7.14 shows this variability when comparing the small Box graphs with the large Box graphs. Figure 7.15 goes one step further by fixing both the DAG configuration and the Box Size but varying the implementation between lighter kernels and heavier kernels, showing a trade-off between them depending on the number of available devices. To further understand the performance impact of varying kernel size, we ran five variants of the SPIRAL generated code with varying numbers of kernels, 19, 15, 11, 8, and 5 (portrayed in Fig. 5.3-d). We ran these kernels on the Zenith machine with two consumer grade GPUs, 1 Nvidia GPU, and 1 AMD GPU as this hardware will most clearly showcase variability given their relative computation power

compared to the previously tested machines. We see in Fig. 7.16 using `DAG Fusion`, the kernel variant that performs the best depends on which hardware device is being used. If the Nvidia GPU is used, 8 kernels perform the best, while when the AMD GPU is used, 11 kernels perform the best. The factors that influence which implementation is the best are the amount of computation per kernel, kernel launch time, and ability to leverage multi-stream execution as well as the runtime orchestration. IRISX is able to automatically discover which implementation provides the best performance.

Functionality Discovery. In addition to kernel-level concurrency, IRISX can discover functionality for applications or what the minimum amount of resources is needed to successfully execute and terminate. As seen in Figs. 7.14e and 7.14g, there are certain sizes for which the Euler Equations application does not execute due to insufficient memory requirements. IRISX leverages its internal memory prediction model, calculating the memory needed for the application and the SPIRAL generated temporary memory and comparing it against the amount of system memory available. This allows IRISX to significantly prune the search space for valid configurations to empirically test for performance.

These features add another dimension to IRISX capabilities. Varying the number of kernels suggests opportunities for the runtime system to gather kernel specific metrics and inform the code generation system to generate new variants on-the-fly that better utilize available hardware resources. Furthermore, the application developer no longer has to manually implement functionality testing, as that can be abstracted away through IRISX. IRISX can directly inform the user of the configurations that can be executed successfully.

7.5.3 Multi-vendor Scalability

So far, IRISX has shown performance portability across different systems with the same vendor architecture. However, IRISX is capable of utilizing and orchestrating different kinds of accelerators in the same system without source code modification, making it ready for future heterogeneous systems where different types of accelerators might exist in a single node. Figure 7.17 shows the speedup of the Euler Equations application on the Cades Cloud system. This system has a multi-vendor compute node with four Nvidia

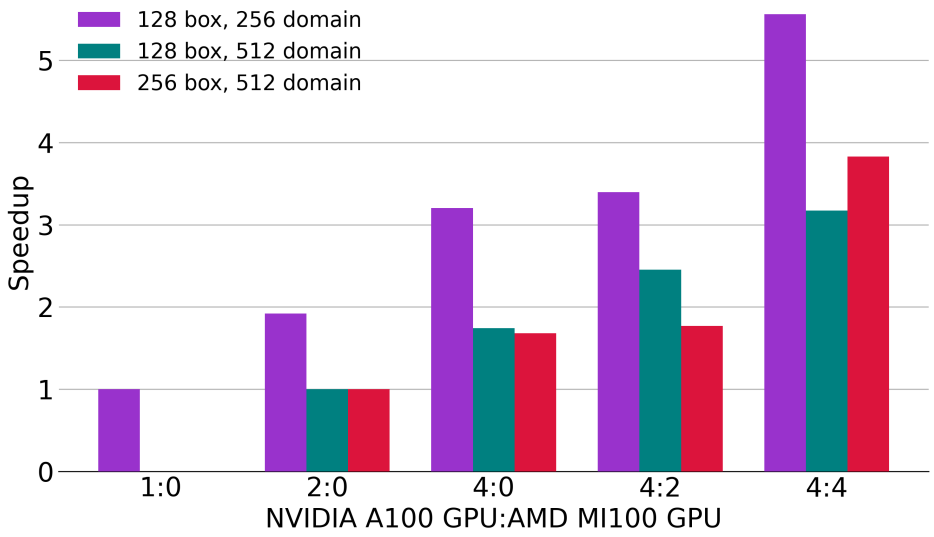


Figure 7.17: Speedup across the vendor boundary using the Cades Cloud Machine. For one, two, and four GPU cases Nvidia GPUs are used and for six and eight GPU cases two and four AMD GPUs are added.

GPUs and four AMD GPUs. IRISX is capable of running seamlessly on one or all devices available on this platform by configuring a single environment variable. IRISX shows an increasing speedup as the number of processors increases for large box sizes even when the number of processors exceeds a single vendor. While the current state-of-the-art HPC systems do not have such multi-vendor accelerator environment, cloud and experimental computing facilities already have such deployment. Moreover, this demonstration shows that IRISX is not bound to any set/kind of architecture, being flexible for any future extremely heterogeneous systems.

7.5.4 Comparison with Proto

Figure 7.18 shows the performance difference between the 3D Euler equation implemented in Proto library without SPIRAL and the best configuration picked by IRISX on Milan, a two Nvidia A100 GPU node in ExCL. For this performance comparison, we time the computation time for Proto and IRISX considering that one-time code generation performed by IRISX was done previously. We see that on 1 GPU IRISX provides a significant

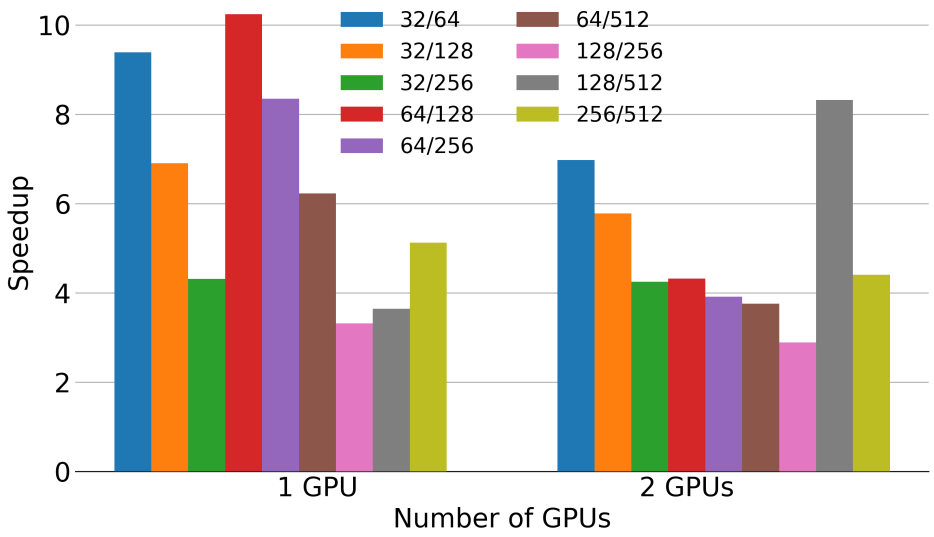


Figure 7.18: Speedup of the best implementation across all configurations in IRISX compared to the base Proto library on Milan.

speedup in the range of 3.3× all the way to 10× across the problem sizes. A similar trend can be seen for two GPUs with speedups in the range of 2.8× up to 8.8× across the problem sizes.

7.6 FortranX Results

We show the performance results of FortranX compared to a library baseline for the cyclic convolution kernel. Cyclic convolution is a critical operation in the areas of numerical methods and spectral methods. This kernel uses an FFT-based implementation rather than direct convolution due to algorithmic complexity benefits. The FFT-based cyclic convolution consists of a forward FFT, a pointwise multiplication, and an inverse FFT. The results are shown in Fig. 7.19. FortranX leverages SPIRAL to provide algorithmic optimizations to the cyclic convolution kernel, which includes library-call fusion along with kernel execution on the GPU through IRIS. We demonstrate that FortranX achieves up to ∼15.7x improvement over the serial Fortran implementation. Further tuning can be achieved to get even better performance.

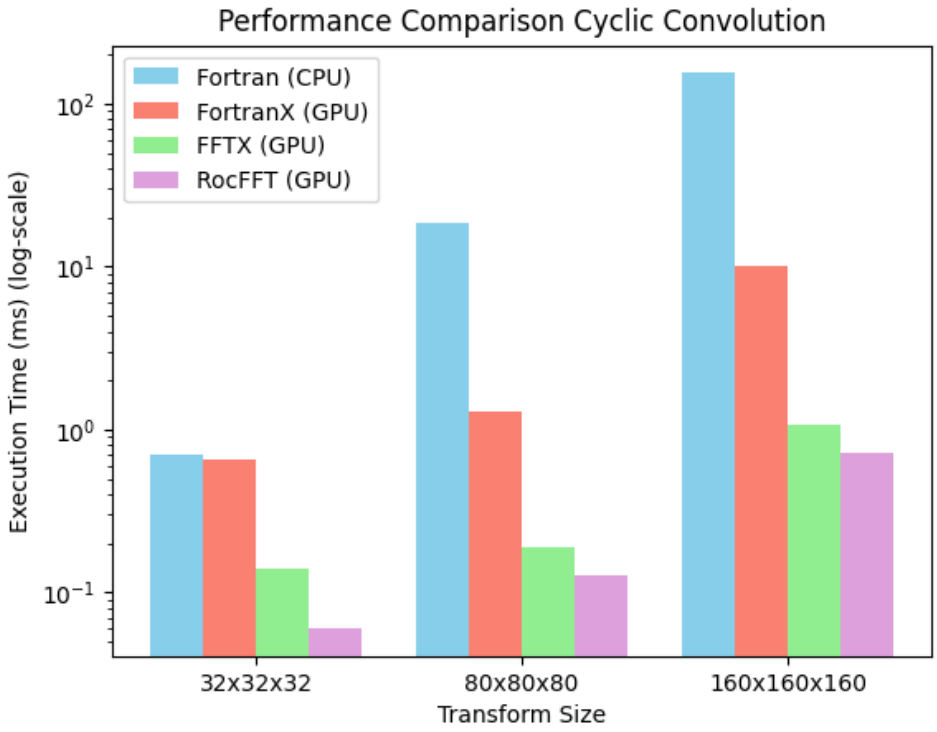


Figure 7.19: Comparison of Fortran, FortranX, SPIRAL, and vendor implementations of cyclic convolution. The y-axis is log scale.

7.7 LibraryX-ASIC

We show preliminary results of LibraryX-ASIC for FFTs of various sizes against software implementations.

7.7.1 Experimental Setup.

We show both CPU and GPU performance results as baselines to compare against LibraryX-ASIC. On CPU, we run FFTW on a 20-core Intel Xeon E5-2698v4 and for GPU evaluation, we run cuFFT on an Nvidia H100. Accelerator performance results are based on a cycle-accurate accelerator model that is calibrated against real silicon measurements from test chips [88, 89] taped on a TSMC 28nm process. The performance model simulates the abstract machine model shown in Fig. 6.6, where the controlling CPU is a single

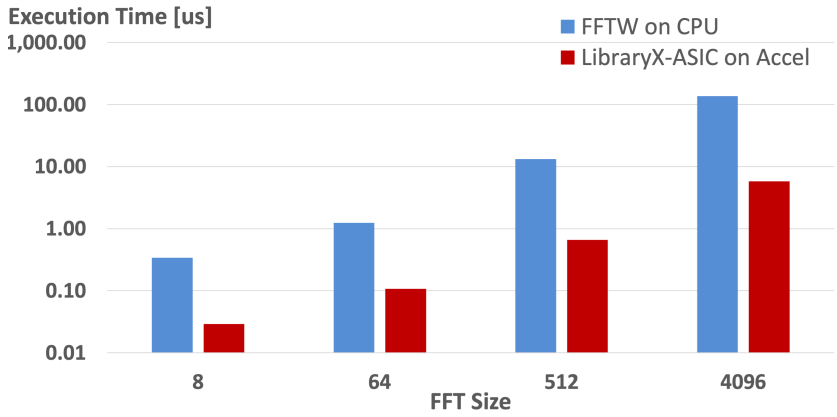


Figure 7.20: FFTW on CPU vs. LibraryX-ASIC on accelerator system.

core of the Intel Xeon E5-2698 CPU, main memory consists of 256 GB of RDIMM DDR4, and the accelerator interfaces to 256 kB of banked, SRAM-based local memory. The FFT accelerator core accelerates a radix-8 twiddle codelet. For GPU comparisons, we target off-chip HBM3e DRAM with the same accelerator configuration.

7.7.2 Results.

The execution times in microseconds are shown in Fig. 7.20 and 7.21 for power-of-8 1D complex FFTs ranging from 8 to 4096. LibraryX-ASIC targeting the CPU-accelerator system achieves speedups of 11x – 23x as compared to running FFTW on CPU only. The performance results demonstrate an order of magnitude improvement in execution time for a real FFT program running on the custom FFT ASIC through the LibraryX-ASIC framework.

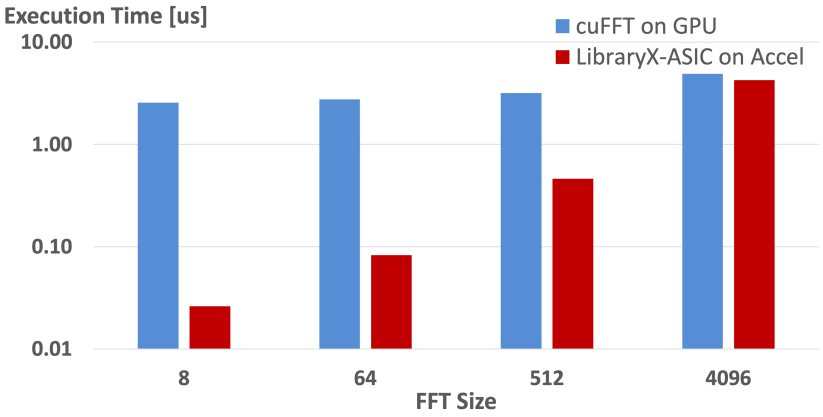


Figure 7.21: cuFFT on GPU vs. LibraryX-ASIC on accelerator system.

Compared to cuFFT running on an H100 GPU, LibraryX-ASIC also provides up to an order of magnitude speedup at smaller size FFTs. Improved speedup against the GPU at larger FFT sizes can be achieved by scaling the number of hardened codelets in the FFT accelerator. These results demonstrate the LibraryX-ASIC framework automatically targeting a custom FFT accelerator through CPU and GPU FFT library calls.

Chapter 8

Conclusion and Future Work

Although there has been a large body of work in optimizing scientific computing applications, there has been little work in bridging the gap between application productivity and performance. In most cases, applications developers have to make the decision between using clean interfaces, such as domain specific libraries, versus expert implementations. This is because libraries offer a balance of performance and productivity, but can leave significant performance on the table. A performance expert can unlock that performance at the cost of removing the library calls themselves. This results in unmaintainable applications that have to be constantly rewritten for each new hardware platform.

This works makes the following contributions. First, it introduces the LibraryX framework, a system that recognizes library call sequences and provides the optimizations done by performance experts in scientific computing. LibraryX uses a combination of library call capture, code generation, and runtime compilation to modify the implementation of a scientific computing application without source code modification. Second, it demonstrates the efficacy of LibraryX in key domains of scientific compupting, including spectral methods, graph analytics and sparse linear algebra, and structured grids. LibraryX demonstrates how its internal code generation system, SPIRAL, can be taught performance optimizations such as kernel fusion to significantly reduce a computation's data footprint and increase its arithmetic intensity. Finally, it shows how LibraryX can be

expanded to support external runtime systems such as IRIS, fixed-function ASICs, and other front-end languages such as Fortran. In particular, this thesis shows the following:

- Recognizes that scientific library primitives have semantics that can provide a high-level description of a computation independent of the primitive implementation.
- Develops an approach, LibraryX, that leverages the semantics of library calls by treating them as specifications rather than implementations. Using techniques such as preprocessor library interpositioning, Inspector/Executor, and operator overloading, library calls can produce their semantic meaning for analysis and optimization.
- Library semantics can be taught to the SPIRAL code generation system to discover optimizations through high-level analysis and automatically generate an implementation that has significant performance benefits. This enables whole computation (solver) optimization as opposed to single kernel/primitive optimization.
- Optimizes both single-library and multi-library call sequences, addressing the combinatorial explosion problem of providing optimized primitives for all combinations of library calls.
- Showcases how the interplay between code generation and an intelligent runtime system can finetune application performance on diverse multi-accelerator hardware platforms.
- Demonstrates the efficacy of this approach by applying it to multiple domains within scientific computing including spectral methods, graph analytics/sparse linear algebra, and structured grids, and showed that it can be expanded to support additional front-end languages and accelerators.

8.1 Limitations

Although extensible for a variety of scientific domains, LibraryX has some limitations in how it can be applied along with overhead through its runtime approach. LibraryX relies

on a library interface with function calls that it can replace. LibraryX is unable to generally parse core language features, such as loops and conditionals, meaning that any algorithm that has core logic outside a function call cannot be recognized. This is specifically an issue for computations that use data-dependent control flow, which LibraryX does not support. In addition, LibraryX operates under the assumption that intermediate dataholders are not referenced during the computation.

The overhead of LibraryX is generally negligible, with the exception of the code generation system. By using preprocessor directives library calls are replaced at compile time, with much simpler logic than the original operation. Additionally, DAG creation and generated kernel runtime compilation have little performance impact. The code generation time can be significant, depending on the analysis time. This is significantly reduced by caching both on disk and in program memory and can be amortized for large program runs.

Similarly, IRISX is a dynamic runtime system, resulting in overhead in different stages of execution. IRISX relies on SPIRAL to generate computation kernels at runtime, which depends on the number of architectures, the OL description, the size of the problem, and the number of kernel variants. However, this is a one-time cost as there is a layered caching system in both the memory of the running program and on disk. The IRIS runtime system also has task management and kernel launch overhead, which IRISX alleviates through various fusion and performing memory optimizations to reduce the footprint of temporary memory.

8.2 Future Work

In this section we discuss the possible directions for future work.

- Move the LibraryX framework from a combined compile-time and runtime approach to a purely compile time approach using standard compiler toolchains like LLVM or MLIR.

- Expand to other domains in scientific computing including Particle methods and Dense linear algebra/Machine Learning.
- Provide general API support such that other code generation approaches, runtime compilation schemes, and programming models can leverage the LibraryX framework. Particular targets that can reason about optimizations like FLAME [39] and Polly [38] are good candidates for extensions to LibraryX.
- Show more examples of frontend languages that utilize LibraryX such as Python, Rust, and Julia.
- Begin the process of generalizing from specific cases to entire domains within scientific computing.

Bibliography

[1] CuFFT. Available at https://docs.nvidia.com/cuda/cufft/index.html. 5

[2] ExCL: ORNL Experimental Computing Laboratory. Available at https://www.excl.ornl.gov/. 112

[3] Proto. Available at https://github.com/applied-numerical-algorithms-group-lbnl/proto. 5, 100, 113

[4] rocFFT. Available at https://rocm.docs.amd.com/projects/rocFFT/en/latest/. 5

[5] SPIRAL. https://spiral.net/. 77

[6] GraphBLAS Template Library (GBTL), Version 3.0. Available at https://github.com/cmu-sei/gbtl, June 2020. 5, 34, 98

[7] CUDA C++ Programming Guide. Available at https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, April 2022. 53

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association. 5

[9] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, Dec 2006. 113

[10] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011. 8, 75

[11] A. Azad and A. Buluc. A work-efficient parallel sparse matrix-sparse vector multiplication algorithm. *Proceedings - 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, 7 2017. 55

[12] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012. 8

[13] S. Beamer, K. Asanović, and D. Patterson. The gap benchmark suite, 2017. 34

[14] M. Blanco, T. M. Low, and K. Kim. Exploration of fine-grained parallelism for load balancing eager k-truss on gpu and cpu. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019. 35, 53, 102

[15] M. Bolten, F. Franchetti, P. H. J. Kelly, C. Lengauer, and M. Mohr. Algebraic description and automatic generation of multigrid methods in spiral. *Concurrency and Computation: Practice and Experience*, 29(17):e4105, 2017. e4105 cpe.4105. 63

[16] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, et al. Distibuted dense numerical linear algebra algorithms on massively parallel architectures: Dplasma. 2010. 8

[17] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013. 8

[18] A. Brahmakshatriya, Y. Zhang, C. Hong, S. Kamil, J. Shun, and S. Amarasinghe. Compiling graph applications for gpu s with graphit. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 248–261, 2021. 108

[19] A. Buluç and J. R. Gilbert. The combinatorial blas: design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, Nov. 2011. 16

[20] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang. Design of the graphblas api for c. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 643–652, 2017. 34, 98, 101

[21] P. Burkhardt. Graphing trillions of triangles. *Information Visualization*, 16, 09 2016. 47, 98

[22] A. M. Cabrera, S. Hitefield, J. Kim, S. Lee, N. R. Miniskar, and J. S. Vetter. Toward performance portable programming for heterogeneous systems on a chip: A case study with qualcomm snapdragon soc. In *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*, pages 1–7. IEEE, 2021. 77

[23] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, Oct. 2018. USENIX Association. 5

[24] GCC, the GNU Compiler Collection. Available at `https://gcc.gnu.org`. 89

[25] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, page 25–39, Berlin, Heidelberg, 1998. Springer-Verlag. 3

[26] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011. xvii, 105

[27] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, Mar. 1988. 16

[28] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011. 8

[29] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. 8, 75

[30] L. Fedeli, A. Huebl, F. Boillod-Cerneux, T. Clark, K. Gott, C. Hillairet, S. Jaure, A. Leblanc, R. Lehe, A. Myers, C. Piechurski, M. Sato, N. Zaïm, W. Zhang, J.-L. Vay, and H. Vincenti. Pushing the frontier in the design of laser-based electron accelerators with groundbreaking mesh-refined particle-in-cell simulations on exascale-class supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022. 97

[31] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel. Operator language: A program generation framework for fast kernels. In W. M. Taha, editor, *Domain-Specific Languages*, pages 385–409, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. 6, 12, 77, 89

[32] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura. Spiral: Extreme performance portability. *Proceedings of the IEEE*, 106(11):1935–1968, 2018. xvi, 5, 6, 9, 35, 54, 55, 75, 88

[33] F. Franchetti and M. Puschel. Generating high performance pruned fft implementations. In *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 549–552, 2009. 31

[34] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura. Discrete Fourier transform on multicores: Algorithms and automatic implementation. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, 26(6):90–102, 2009. 51

[35] F. Franchetti, D. G. Spampinato, A. Kulkarni, D. T. Popovici, T. M. Low, M. Franusich, A. Canning, P. McCorquodale, B. Van Straalen, and P. Colella. Fftx and spectralpack: A first look. In *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, pages 18–27. IEEE, 2018. xiii, 46, 59, 79

[36] F. Franchetti, Y. Voronenko, and M. Püschel. Formal loop merging for signal transforms. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 315–326, New York, NY, USA, 2005. Association for Computing Machinery. xvi, 49, 54, 55, 59, 63, 84

[37] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation". 5, 91

[38] T. Grosser, A. Groesslinger, and C. Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012. 132

[39] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, Dec. 2001. 132

[40] S. Guyer and C. Lin. Broadway: A compiler for exploiting the domain-specific semantics of software libraries. *Proceedings of the IEEE*, 93(2):342–357, 2005. 6

[41] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 100–112, New York, NY, USA, 2018. Association for Computing Machinery. 5

[42] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector SIMD architectures. In *ACM International Conference on Supercomputing*, pages 13–24, 2013. 13

[43] R. W. Hockney and J. W. Eastwood. *Computer simulation using particles*. Taylor & Francis, Inc., USA, 1988. 19

[44] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, sep 1989. 7

[45] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 703–718, New York, NY, USA, 2022. Association for Computing Machinery. 5, 17

[46] K. Inbasekar, Y. Shekel, and M. Asa. ICICLE v2: Polynomial API for coding ZK provers to run on specialized hardware. Cryptology ePrint Archive, Paper 2024/973, 2024. 100

[47] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 6. ACM, 2014. 8

[48] L. V. Kale and S. Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993. 8

[49] K. Kennedy. Telescoping languages: a compiler strategy for implementation of high-level domain-specific programming systems. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 297–304, 2000. 4

[50] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, et al. Mathematical foundations of the

graphblas. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016. 34

[51] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, USA, 2011. 34, 98

[52] J. Kim, S. Lee, B. Johnston, and J. S. Vetter. IRIS: A portable runtime system exploiting multiple heterogeneous programming systems. In *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*, pages 1–8. IEEE, 2021. 8, 75, 77

[53] J. Kim, S. Lee, B. Johnston, and J. S. Vetter. Iris: A portable runtime system exploiting multiple heterogeneous programming systems. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, 2021. 88

[54] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. 5, 17

[55] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. S. Müller. Chameleon: reactive load balancing for hybrid mpi+ openmp task-parallel applications. *Journal of Parallel and Distributed Computing*, 138:55–64, 2020. 8

[56] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 127–138, New York, NY, USA, 2013. Association for Computing Machinery. 13

[57] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society. 5, 17

[58] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. Mlir: scaling compiler infrastructure

for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, page 2–14. IEEE Press, 2021. 5, 17

[59] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 144–154, New York, NY, USA, 1993. Association for Computing Machinery. 7

[60] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, Sept. 1979. 16

[61] M. Lee and T. M. Low. A family of provably correct algorithms for exact triangle counting. In *Proceedings of the First International Workshop on Software Correctness for HPC Applications*, Correctness'17, page 14–20, New York, NY, USA, 2017. Association for Computing Machinery. 102

[62] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014. xvii, 105

[63] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002. 114

[64] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, Mar. 2021. 5

[65] T. M. Low, V. N. Rao, M. Lee, D. Popovici, F. Franchetti, and S. McMillan. First look: Linear algebra-based triangle counting without matrix multiplication. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2017. xii, 35, 39, 40, 47, 49

[66] T. M. Low, D. G. Spampinato, A. Kutuluru, U. Sridhar, D. T. Popovici, F. Franchetti, and S. McMillan. Linear algebraic formulation of edge-centric k-truss algorithms

with adjacency matrices. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, 2018. 47

[67] H. Mankad, M. A. H. Monil, S. Rao, P. Colella, B. Van Straalen, F. Franchetti, and J. Vetter. A Performance-Portable MultiGPU Implementation of 3D Euler Equations using ProtoX and IRIS. In *SC '24 workshop, ScalAH '24: The 15th Workshop on Latest Advances in Scalable Agorithms for Large Scale Heterogeneous Systems*, 2024. Accepted for publication. 9

[68] H. Mankad, S. Rao, P. Colella, B. Van Straalen, and F. Franchetti. Protox: A first look. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2023. 13, 77, 100, 114

[69] P. McCorquodale and P. Colella. A high-order finite-volume method for conservation laws on locally refined grids. *Communications in Applied Mathematics and Computational Science*, 6(1):1–25, 2011. 98

[70] P. McCorquodale and P. Colella. A high-order finite-volume method for conservation laws on locally refined grids. *Communications in Applied Mathematics and Computational Science*, 6(1):1 – 25, 2011. 114

[71] N. R. Miniskar, M. A. H. Monil, P. Valero-Lara, F. Y. Liu, and J. S. Vetter. Iris-dmem: Efficient memory management for heterogeneous computing. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2023. 76, 84

[72] M. A. H. Monil, N. R. Miniskar, F. Y. Liu, J. S. Vetter, and P. Valero-Lara. Laris: Targeting portability and productivity for LAPACK codes on extreme heterogeneous systems by using IRIS. In *IEEE/ACM Redefining Scalability for Diversely Heterogeneous Architectures Workshop, RSDHA@SC 2022, Dallas, TX, USA, November 13-18, 2022*, pages 12–21. IEEE, 2022. 8

[73] M. A. H. Monil, N. R. Miniskar, K. Teranishi, J. S. Vetter, and P. Valero-Lara. Matris: Multi-level math library abstraction for heterogeneity and performance portability

using iris runtime. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 1081–1092, 2023. 8

[74] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471, 2013. 101

[75] OpenACC. OpenACC: Directives for accelerators, 2015. 8

[76] OpenMP. OpenMP reference, 1999. 8

[77] M. Paleczny, C. Vick, and C. Click. The java HotSpot™ server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, Monterey, CA, Apr. 2001. USENIX Association. 7

[78] S. Parimalarangan, G. M. Slota, and K. Madduri. Fast parallel graph triad census and triangle counting on shared-memory platforms. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1500–1509. IEEE, 2017. 35

[79] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019. 5

[80] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011. 35

[81] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. ryan Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005. 35, 59

[82] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 519–530, New York, NY, USA, 2013. Association for Computing Machinery. 5, 17

[83] S. Rao, A. Kutuluru, P. Brouwer, S. McMillan, and F. Franchetti. Gbtlx: A first look. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020. 13, 59, 77, 102

[84] S. Rao, M. A. H. Monil, H. Mankad, J. Vetter, and F. Franchetti. FFTX-IRIS: Towards Performance Portability and Heterogeneity for SPIRAL Generated Code. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 1635–1641, New York, NY, USA, 2023. Association for Computing Machinery. 9, 88

[85] A. Sabne. Xla : Compiling machine learning for peak performance, 2020. 5

[86] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3):66–73, 2010. 7

[87] M. M. Strout, M. Hall, and C. Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018. 6

[88] L. Tang, S. Chen, K. Harisrikanth, G. Xu, F. Franchetti, and K. Mai. A 1.19ghz 9.52gsamples/sec radix-8 fft hardware accelerator in 28nm. In *2024 IEEE Hot Chips 36 Symposium (HCS)*, pages 1–1, 2024. 91, 126

[89] L. Tang, S. Chen, K. Harisrikanth, G. Xu, K. Mai, and F. Franchetti. A high throughput hardware accelerator for fftw codelets: A first look. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2022. 91, 126

[90] B. Van Straalen. *Method of Local Corrections Solver for Manycore Architectures*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2018. xi, 28, 32

[91] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries, 1998. 3

[92] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. V. Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. P. Jr., T. Peterka, M. Strout, and J. Wilke. Extreme heterogeneity 2018 - productive computational science in the era of extreme heterogeneity: Report for DOE ASCR workshop on extreme heterogeneity. Technical report, USDOE Office of Science (SC) (United States), 2018. 75

[93] J. Woodruff, J. Armengol-Estapé, S. Ainsworth, and M. F. P. O'Boyle. Bind the gap: compiling real software to hardware fft accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 687–702, New York, NY, USA, 2022. Association for Computing Machinery. 5

[94] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 298–308, 2001. 77

[95] R. Yadav, A. Aiken, and F. Kjolstad. Distal: The distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 286–300, New York, NY, USA, 2022. Association for Computing Machinery. 8, 17

[96] R. Yadav, A. Aiken, and F. Kjolstad. Spdistal: Compiling distributed sparse tensor computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022. 8, 17

[97] C. Yang, A. Buluç, and J. D. Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. *ACM Trans. Math. Softw.*, 48(1), feb 2022. 101

[98] N. Zhang, A. Ebel, N. Neda, P. Brinich, B. Reynwar, A. G. Schmidt, M. Franusich, J. Johnson, B. Reagen, and F. Franchetti. Generating high-performance number theoretic transform implementations for vector architectures. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2023. 13, 77, 99

[99] N. Zhang and F. Franchetti. Code generation for cryptographic kernels using multi-word modular arithmetic on gpu. In *2025 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2025. 99

[100] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. Graphit: a high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. 5, 17

[101] E. Zoni, R. Lehe, O. Shapoval, D. Belkin, N. Zaïm, L. Fedeli, H. Vincenti, and J.-L. Vay. A hybrid nodal-staggered pseudo-spectral electromagnetic particle-in-cell method with finite-order centering. *Computer Physics Communications*, 279:108457, Oct. 2022. 97