LibraryX: A Framework for Cross-Library-Call Optimization

Sanil Rao, Anant Prakash, Naifeng Zhang, Het Mankad, Franz Franchetti
Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA
{sanilr, anantpra, naifengz, hmankad, franzf}@andrew.cmu.edu

***Abstract*—Scientific applications utilize performance libraries as a software engineering concept: these libraries encapsulate important and well-understood (mathematical) operations, allow for reuse, and are implemented and tuned by experts. Domain scientists then implement complex algorithms based on these domain-specific libraries. While individual library calls are optimized, larger performance gains across sequences of calls—sometimes spanning multiple libraries—are often unrealized, forcing a trade-off between performance and implementation complexity.**

To overcome this issue, we propose LibraryX, an approach and a system that allows for cross-library-call optimization even when library calls stem from multiple performance libraries. LibraryX annotates library calls with semantic information and optimizes entire directed acyclic graphs (DAGs) of calls dynamically using the SPIRAL code generation system. We demonstrate its effectiveness across a range of memory bound workloads, achieving significant speedups on Nvidia, AMD, and Intel accelerators compared to code using native libraries without cross-call optimization.

I. Introduction

A recurring challenge in scientific computing is achieving high performance while maintaining developer productivity. Many fundamental problems in scientific computing—such as spectral methods, Partial Differential Equations (PDEs) solvers, and graph analytics—are memory bound with low arithmetic intensity, making them difficult to optimize. Achieving high performance requires carefully leveraging low-level hardware-specific primitives that are neither intuitive nor accessible to new developers, thereby reducing productivity. This problem is compounded when targeting multiple hardware platforms, as each platform provides specific primitives necessary to achieve peak performance, necessitating complete algorithm reimplementation.

To address the challenge of developer productivity while also maintaining good performance, the community uses domain-specific libraries. These libraries ease development by abstracting away hardware details and providing battle-tested implementations[1][2][3] as a suite of functions. Although libraries are considered the gold standard, unfortunately, library abstraction fundamentally limits optimization opportunities. Effective optimization of low arithmetic intensity algorithms, for example, requires a global view of the entire algorithm, including understanding data flow and reuse patterns, while carefully utilizing available hardware resources. Library interfaces obscure these details, treating each operation as an isolated black box, preventing cross-library-operation optimizations that are critical for such workloads. This leaves significant performance on the table, forcing developers to abandon libraries entirely.

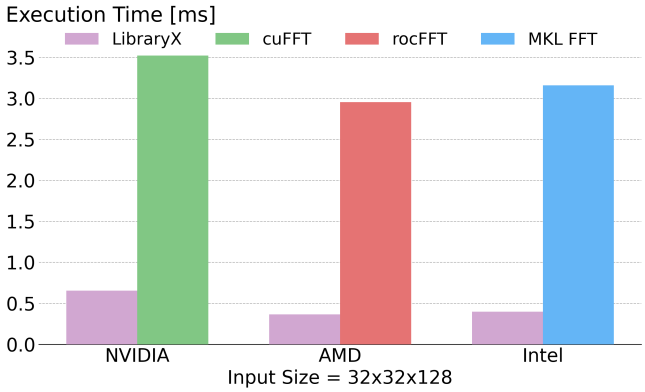


Fig. 1: Performance comparison between various vendors and LibraryX implementation of Hockney Free-Space convolution (lower is better). LibraryX outperforms all vendor implementations with speedups of 5x, 9x, and 8x respectively.

Existing solutions do not adequately address this library optimization problem. Libraries must be specialized, as no one library can capture the entire space of scientific computing. General-purpose compilers[4][5] treat library calls as opaque functions and cannot optimize across library boundaries, losing the global algorithmic view needed for effective optimization. Domain-specific compilers[6][7][8][9][10][11] can provide deeper optimization within their specific domains but are too restrictive to handle multi-domain algorithms that combine operations from different libraries and cannot be easily integrated into existing applications. None of these approaches can automatically capture and optimize the global data flow patterns that span multiple library calls in complex scientific kernels.

This paper presents LibraryX, an automated framework for optimizing algorithms written against standard library operators in scientific computing, with a focus on low arithmetic intensity algorithms. LibraryX's key insight is to treat library calls as specifications rather than opaque implementations. Through a library call interception process, LibraryX extracts the mathematical semantics of the operation, constructing a unified intermediate representation that captures the global algorithmic view, enabling cross-library optimizations impossible with isolated library calls.

LibraryX leverages SPIRAL, a program synthesis system

with extensive knowledge of transformations for scientific algorithms. SPIRAL analyzes the unified representation and generates optimized implementations for the target hardware, which LibraryX transparently executes in place of original library calls, providing library-based productivity with expert-level performance, all without source code modification. As shown in Fig. 1, LibraryX achieves up to 9× speedups over vendor libraries for Hockney Free-Space convolution across multiple platforms, freeing developers to focus on application features rather than performance engineering.

Contributions. This paper demonstrates the end-to-end system, LibraryX, across a number of applications and GPU-based accelerators. Specifically, this paper shows:

- Lifting of single and multi-domain library-based algorithms from implementations to specifications.
- Expression of mathematical library semantics in a cross-library domain-specific language, enabling the generation of optimized cross-library implementations.
- Automatic full-solver code synthesis and optimization across library boundaries.
- Performance portability of algorithms written against LibraryX across GPUs from different vendors, demonstrating significant speed-up over vendor library based implementations.

II. 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.

A. Main Example: Hockney Free-Space 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 a limiting factor, 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, an element-wise 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 Free-Space, the Hockney Free-Space convolution[12] is used. Unlike circular convolution with periodic boundaries, Hockney Free-Space 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. We illustrate the Hockney Free-Space Convolution calculation in Fig. 2 along with its mathematical specification, discussed in detail in subsequent sections.

B. Semantic Capturing and Runtime Optimization

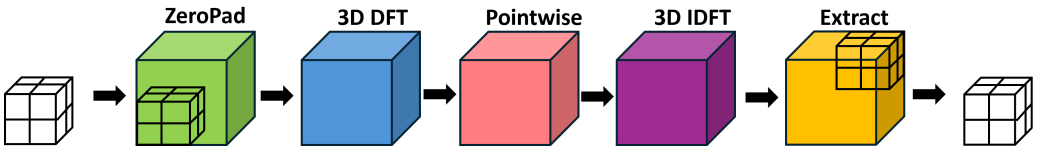
C Preprocessor and Library Interpositioning. The C preprocessor provides a mechanism to modify source programs through macros before invoking the compiler. Using these macros, programmers can perform textual replacement of operations, provide compile time branches, and turn on or off sections of code. These macros offer increased flexibility for programmers to support portability to different systems and environments. Preprocessor macros can be used at any point in the program, but only affect portions of the program following the macro definition. The C preprocessor can be used to perform library interpositioning where library function calls are replaced with a different implementation of the function without directly modifying the function.

Lazy Evaluation. Lazy evaluation is an execution strategy in which functions or expressions defer execution until a specific output is required. Programming languages that implement lazy evaluation collect expression sequences at runtime, evaluating them as needed for program correctness. Functional programming languages like Haskell[13] and OCaml[14] and some partitioned global address space (PGAS) languages, such as X10[15], have built-in primitives to support lazy evaluation. This strategy is in direct contrast to eager or strict evaluation, which immediately evaluates a function as it is executed in program order. Lazy evaluation shares many names, including delayed/deferred execution and futures.

Runtime Compilation. Runtime compilation is a compilation strategy to dynamically compile and link source code into a running program. This process involves taking source code, usually in a high-level language, and calling a traditional compiler on that source code to generate a binary or dynamic library. This output is then accessed within the running program so that the binary's exported functions can be executed. This process is generally programmer managed. Runtime compilation resides within the general framework of Just-In-Time compilation. The key difference is that Just-In-Time compilation involves optimizing sections of code at the intermediate or bytecode level and is usually tightly coupled with the runtime system.

C. Analysis and Code Generation

SPIRAL[16] 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



$$y = \left(\mathrm{I}_{k\times K} \otimes \mathrm{I}_{m\times M} \otimes \mathrm{I}_{n\times N}\right)\left(\mathrm{iDFT}_{K\times M\times N}\right)\left(\mathrm{Diag}_f\right)\left(\mathrm{DFT}_{K\times M\times N}\right)\left(\mathrm{I}_{K\times k} \otimes \mathrm{I}_{M\times m} \otimes \mathrm{I}_{N\times n}\right)x \quad (1)$$

Fig. 2: Hockney Free-Space Convolution illustration and SPL derivation from library calls.

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 as

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

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

$$A\otimes B = [a_{i,j}B], \quad \text{for } A=[a_{i,j}], \quad (3)$$

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 the DFT yields two expressions, $I_n \otimes \mathrm{DFT}_n$ and $\mathrm{DFT}_n \otimes I_n$. In the former case, a $n\times n$ 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,

$$\mathrm{DFT}_n = \left(\mathrm{DFT}_m \otimes \mathrm{I}_k\right) T_k^n \left(\mathrm{I}_m \otimes \mathrm{DFT}_k\right) L_m^n, \quad n = mk, \quad (4)$$

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

$$\mathrm{DFT}_{m\times n} = \mathrm{DFT}_m \otimes \mathrm{DFT}_n, \quad (5)$$

$$\mathrm{DFT}_{k\times m\times n} = \mathrm{DFT}_k \otimes \mathrm{DFT}_m \otimes \mathrm{DFT}_n. \quad (6)$$

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

$$\mathrm{I}_{n\times N} = \left[\mathrm{I}_n \,|\, \mathbf{0}\right], \mathbf{0} \in \mathbb{R}^{n\times(N-n)} \quad \text{and} \quad \mathrm{I}_{N\times n} = \mathrm{I}_{n\times N}^{\mathsf{T}}, \quad (7)$$

where $N > n$. Using the rectangular identity, zero padding and extraction take the form

$$y = \mathrm{I}_{N\times n} x \quad \text{and} \quad y = \mathrm{I}_{N\times n}^{\mathsf{T}} x, \quad (8)$$

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 from 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.

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) [17], a superset of SPL. OL introduces multilinear operators into the SPIRAL system. These operations include scalar/dot product, convolution, and matrix multiplication. The OL representation of scalar/dot product of two vectors of length n with scalars α and β is,

$$y = ([\alpha,\beta]\otimes \mathrm{I}_n)x, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad (9)$$

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 [18], stencils and structured grids [19][20][21], and cryptography [22][23].

III. LibraryX Design

The focus of LibraryX is sequential single threaded applications using libraries without external parallelism constructs. We discuss in detail the key stages of LibraryX shown in Fig. 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.

A. Capturing the Library Frontend

To capture a computation written using libraries without modifying the source implementation, there must 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

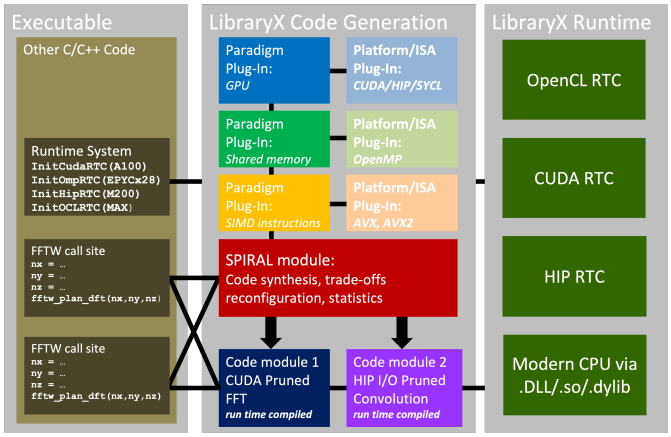


Fig. 3: The flow of LibraryX from library call capture, to code generation, and finally backend hardware execution.

in the `shim`. The `shim` contains LibraryX implementations of these operators, rather than the base library implementation. At the end of the header file are the preprocessor macros needed to perform library interpositioning, replacing the library calls with the shimmed calls.

At runtime, the LibraryX'd 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 equivalent representation in SPIRAL. 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 produced. If that function pointer has not been populated, LibraryX will generate and compile an optimized implementation of the computation, producing that pointer. This function pointer will then be used with the captured input and outputs to perform the optimized computation.

Figure 5 highlights the `shim`, library interpositioning, and backend object creation in LibraryX. Lines 8-48 show an abbreviated example of `shimmed` library calls. Lines 50-55 show how library interpositioning transforms the original library calls into their `shim` equivalents at compile time. Finally, lines 41-46 show how LibraryX instantiates the backend object to create a function pointer to the generated GPU function and kernels shown on lines 60-66. Although the `shim` mappings are written/managed by the LibraryX developer, they are reusable across libraries of the same type, such as FFTs, as the mappings are linked to the mathematics not library API.

B. DAG Unification and Abstraction Lifting

SPIRAL needs to transform the DAG representation, consisting of SPL expression nodes with input and output data edges, into a unified SPL expression. This transformation uses SPIRAL's robust pattern matching system. This system determines whether 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, the SPL expression. This SPL expression can then be manipulated with algebraic rules to optimize the expression. Some of these rules are shown in Table I. As part of the unification process, SPIRAL will configure which of these rules should be applied to the unified abstraction. 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 1 to 31. This process will be shown in detail in Section V.

If no match is found, LibraryX's graceful failure mode is triggered. SPIRAL will output that no expression was found as an *unsuccessful* token. LibraryX will then begin to execute the original library functions in the order in which they are captured. In this way, LibraryX does not impede the progress of large program runs if at any point there is a capture failure.

C. 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. We describe the abstract transformation stages in this section, with previous work discussing the architecture specific optimizations within the various domains SPIRAL supports. 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[24].

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 (icode), is 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 icode is then taken through a source code parser, which generates source code for the target platform.

In addition to the generated code, SPIRAL adds metadata about the generated code parameters. This metadata includes any intermediate memory, with their types and sizes, kernel names, with their specific thread geometry and signature, and 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.

D. Runtime Execution Environment (REE)

LibraryX abstracts hardware execution using an object-oriented design, instantiating the appropriate hardware backend based on flags passed at configure time. These backend implementations vary slightly for CPUs and GPUs. Generally, both 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 compilation overhead of REE is minimal, as SPIRAL produces lightweight source code without any complicated language features or abstractions.

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 GPUs 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.

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, enabling the movement of memory between distinct spaces transparently. The preprocessor macros 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>& in,
                  std::vector<int> dim ={1,2,3}) {
  BoxData<double, 1> output(Box(Point::Zeros(),
            Point({dim[0], dim[1], dim[2]})));
  in.copyTo(output);
  return output;
}
BoxData<double,1> extract(BoxData<double, 1> out,
                  std::vector<int> dim ={1,2,3}) {
  BoxData<double, 1> small(Box(Point::Zeros(),
            Point({dim[0], dim[1], dim[2]})));
  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;
  unsigned 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})));
  uint64_t size = N*M*(K/2+1);
  std::vector<std::complex<double>> temp(size);
  std::vector<std::complex<double>> fout(size);
  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(fout.begin(), //start location
            fout.end(), //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);
}
```

Fig. 4: Source code for Hockney Free-Space Convolution. This sequential C++ code is transparently executed on a GPU after it is dynamically translated to CUDA/HIP/OpenCL.

```
1  //libraryX.hpp
2  #ifndef LIBRARYX_HPP
3  #define LIBRARYX_HPP
4  BoxData<double,1> captured_in{};
5  double *captured_in2 = nullptr;
6  BoxData<double,1> captured_out{};
7  Graph DAG{};
8  void (*funcPtr)(double*,
9                  double*, double*) = nullptr;
10 BoxData<double, 1> libraryx_zeroPad(
11  BoxData<double, 1>& input,
12  std::vector<int> dim ={1,2,3}) {
13    captured_in = input;
14    DAG.insert("I(" + vec2str(dim) + ")",input);
15    return nullptr;
16 }
17 fftw_plan libraryx_fft_r2c_3d(int x, int y,
18   int z, fftw_complex *input, double *output,
19   unsigned f) {
20    DAG.insert("DFT(" + dims2str(x,y,z) +")",
21      input, output);
22    return nullptr;
23 }
24 template<typename InIt, typename OutIt,
25          typename UOp>
26 void libraryx_transform(InIt f1, InIt s1,
27   OutIt fo, UOp uop) {
28   captured_in2 = (double*)get_pointer(s1);
29   DAG.insert("Diag", s1);
30 }
31 fftw_plan libraryx_fft_c2r_3d(int x, int y,
32   int z, double *input, fftw_complex *output,
33   unsigned f) {
34    DAG.insert("DFT(" + dims2str(x,y,z) +")",
35      input, output);
36    return nullptr;
37 }
38 BoxData<double, 1> libraryx_extract(
39   BoxData<double, 1>& input,
40   std::vector<int> dim ={1,2,3}) {
41     captured_out = input;
42     DAG.insert("I(" + vec2str(dim) + ")",input);
43     //create LibraryX Obj and generate code
44     LibraryXObj lb;
45     if(!funcPtr) { funcPtr = lb.compile(DAG); }
46     funcPtr(captured_out.data(),
47       captured_in.data(), captured_in2);
48     return captured_out;
49 }
50 #define zeroPad libraryx_zeroPad
51 #define fftw_plan_dft_r2c_3d libraryx_fft_r2c_3d
52 #define fftw_plan_dft_c2r_3d libraryx_fft_c2r_3d
53 #define transform libraryx_transform
54 #define extract libraryx_extract
55 #endif //LIBRARYX_HPP
56
57 // spiral_generated_code.cu
58 // code generated by transform function
59 void generated_code(double* Y, double* X,
60     double* symbl) {
61   ker_hockney0<<<g1, b626>>>(X);
62   ker_hockney1<<<g2, b627>>>();
63   ker_hockney2<<<g3, b628>>>(symbl);
64   ker_hockney3<<<g4, b629>>>();
65   ker_hockney4<<<g6, b631>>>(Y);
66 }
```

Fig. 5: LibraryX header file (lines 1-57) showing LibraryX instantiation of standard library operators which executes SPIRAL generated kernel implementation (lines 59-68) after analysis and code generation. The SPL mapping is provided by the developer, and only needs to be provided once per unique operator, as the SPL can be reused across operations performing the same mathematics.

IV. End-to-End Example: Hockney Free-Space Convolution

We walk through how LibraryX optimizes Hockney Free-Space convolution. We show the following steps: (1) capturing library call semantics as a DAG specification, (2) recognizing and lifting DAG specification to high-level abstraction, (3) generating optimized implementations for the abstraction, and (4) 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. 4 to Fig. 5 during program execution.

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 via library interpositioning. 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 Free-Space 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 II.

Abstraction Lifting and Code Generation. After a sequence of operations is captured as an SPL DAG it 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 V. 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 implementation. 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 at the end of Fig. 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.

V. Hockney Free-Space Convolution Optimization Derivation

Having shown the end-to-end transformation of Hockney Free-Space 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 Free-Space convolution can be expressed as the point-free composition of five distinct operations shown in SPL as equation 1. This

TABLE I: 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 13, 16, and 17, $m_1 = n_3$ and $m_2 = n_4$.

Property	Operation	
Identity Expansion	$\mathrm{I}_{mn} = \mathrm{I}_m \otimes \mathrm{I}_n$	(11)
Identity Associativity	$A \otimes B = (A \otimes \mathrm{I}_{n_2})(\mathrm{I}_{m_1} \otimes B)$	(12)
Mixed-Product1	$(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$	(13)
Mixed-Product2	$(A \otimes \mathrm{I}_{n_2})(\mathrm{I}_{m_1} \otimes B) = (\mathrm{I}_{n_1} \otimes B)(A \otimes \mathrm{I}_{m_2})$	(14)
Decomposition	$A \otimes B = (A\,\mathrm{I}_{m_1} \otimes \mathrm{I}_{n_2} B) = (\mathrm{I}_{n_1} A \otimes B\,\mathrm{I}_{m_2})$	(15)
Left Identity Distribution	$\mathrm{I}_n \otimes (AC) = (\mathrm{I}_n \otimes A)(\mathrm{I}_n \otimes C)$	(16)
Right Identity Distribution	$(AC) \otimes \mathrm{I}_n = (A \otimes \mathrm{I}_n)(C \otimes \mathrm{I}_n)$	(17)
Permuted Commutativity	$A \otimes B = L^{n_1 n_2}_{n_1} (B \otimes A) L^{m_1 m_2}_{m_2}$	(18)
block-diagonal Kronecker form	$A^{m\times m}_i \otimes_i \mathrm{I}_n = L^{mn}_m \left(\bigoplus_{i=0}^{n-1} A^{m\times m}_i \right) L^{mn}_n$	(19)
Pruned DFT	$\mathrm{PrunedDFT}^n_N = \mathrm{DFT}_N \mathrm{I}_{N\times n}$	(20)
Inverse Pruned DFT	$\mathrm{iPrunedDFT}^n_N = \mathrm{I}_{n\times N} \mathrm{DFT}_N$	(21)

equation was semantically captured by LibraryX using Table II. 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, resulting in significant computational benefits. In equation 1, the composition of the expressions green and blue shows the zero expansion of the input with the 3D forward DFT. Using the definition of DFT decomposition 5 and rule 12 in Table I, this expression can be rewritten as

$$\big(\mathrm{DFT}_K \otimes \mathrm{I}_{MN}\big)\big(\mathrm{I}_K \otimes \mathrm{DFT}_{M\times N}\big)\big(\mathrm{I}_{K\times k} \otimes \mathrm{I}_{MN}\big)\big(\mathrm{I}_k \otimes \mathrm{I}_{M\times m} \otimes \mathrm{I}_{N\times n}\big). \quad (10)$$

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 14 is applied given a reordered expression

$$\big(\mathrm{DFT}_K \otimes \mathrm{I}_{MN}\big)\big(\mathrm{I}_{K\times k} \otimes \mathrm{I}_{MN}\big)\big(\mathrm{I}_k \otimes \mathrm{DFT}_{M\times N}\big)\big(\mathrm{I}_k \otimes \mathrm{I}_{M\times m} \otimes \mathrm{I}_{N\times n}\big). \quad (22)$$

This reordering exposes another key optimization through the application of the identity rules 16 and 17 along with the associativity of matrix multiplication, allowing simplification of the four terms into two terms shown below

$$\big(\mathrm{DFT}_K \mathrm{I}_{K\times k} \otimes \mathrm{I}_{MN}\big)\big(\mathrm{I}_k \otimes \big(\mathrm{DFT}_{M\times N}(\mathrm{I}_{M\times m} \otimes \mathrm{I}_{N\times n})\big)\big). \quad (23)$$

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 rules 11, 12, 13 and 16. Using 5 and then applying rule 12 gives

$$\big(\mathrm{DFT}_K \mathrm{I}_{K\times k} \otimes \mathrm{I}_{MN}\big)\big(\mathrm{I}_k \otimes \big((\mathrm{DFT}_M \otimes \mathrm{I}_N)(\mathrm{I}_M \otimes \mathrm{DFT}_N)(\mathrm{I}_{M\times m} \otimes \mathrm{I}_{N\times n})\big)\big), \quad (24)$$

TABLE II: SPL formulation of standard library calls used to calculate Hockney Free-Space 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})`	$y = \big(\mathrm{I}_{K\times k} \otimes \mathrm{I}_{M\times m}\big)x$
`y = zeroPad3D(x,{k,m,n},{K,M,N})`	$y = \big(\mathrm{I}_{K\times k} \otimes \mathrm{I}_{M\times m} \otimes \mathrm{I}_{N\times n}\big)x$
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})`	$y = \big(\mathrm{I}_{k\times K} \otimes \mathrm{I}_{m\times M}\big)x$
`y = extract3D(x,{K,M,N},{k,m,n})`	$y = \big(\mathrm{I}_{k\times K} \otimes \mathrm{I}_{m\times M} \otimes \mathrm{I}_{n\times N}\big)x$

and then applying 13, we get

$$\big(\mathrm{DFT}_K \mathrm{I}_{K\times k} \otimes \mathrm{I}_{MN}\big)\big(\mathrm{I}_k \otimes \big((\mathrm{DFT}_M \otimes \mathrm{I}_N)(\mathrm{I}_{M\times m} \otimes \mathrm{DFT}_N \mathrm{I}_{N\times n})\big)\big). \quad (25)$$

We then apply 13 one more time, which yields

$$\big(\mathrm{DFT}_K \mathrm{I}_{K\times k} \otimes \mathrm{I}_{MN}\big)\big(\mathrm{I}_k \otimes (\mathrm{DFT}_M \mathrm{I}_{M\times m} \otimes \mathrm{DFT}_N \mathrm{I}_{N\times n})\big). \quad (26)$$

Finally, we can apply equation 12 to the innermost Kronecker product of the right hand term followed by 16 and 11 to get the final term

$$\big(\mathrm{DFT}_K \mathrm{I}_{K\times k} \otimes \mathrm{I}_{MN}\big)\big((\mathrm{I}_k \otimes \mathrm{DFT}_M \mathrm{I}_{M\times m} \otimes \mathrm{I}_N)(\mathrm{I}_{km} \otimes \mathrm{DFT}_N \mathrm{I}_{N\times n})\big). \quad (27)$$

The final simplification shows that the zero expansion and DFT in each dimension can be performed independently as sequential 1D DFTs.

A similar expansion and simplification strategy can be applied to the expression composition in gold and purple (1), extraction and inverse DFT. By applying all the rules shown previously we get the expression,

$$(\mathrm{I}_{km} \otimes \mathrm{I}_{n\times N}\,\mathrm{iDFT}_N)(\mathrm{I}_k \otimes \mathrm{I}_{m\times M}\,\mathrm{iDFT}_M \otimes \mathrm{I}_N)\big(\mathrm{I}_{k\times K}\,\mathrm{iDFT}_K \otimes \mathrm{I}_{MN}\big). \quad (28)$$

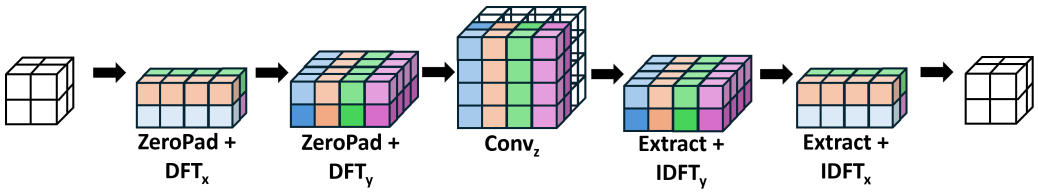
In this form, the 3D inverse DFT is decomposed into batches of 1D DFT in each dimension, and the writing of the result is done directly in the DFT rather than as a unique step after the DFT operation.

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 rules 20 and 21, which introduce the pruned DFT[25], consuming the identity matrices. This transforms equations 27 and 28 as follows

$$\big(\mathrm{PrunedDFT}^k_K \otimes \mathrm{I}_{MN}\big)\big(\mathrm{I}_k \otimes \mathrm{PrunedDFT}^m_M \otimes \mathrm{I}_N\big)\big(\mathrm{I}_{km} \otimes \mathrm{PrunedDFT}^n_N\big), \quad (29)$$

$$\big(\mathrm{I}_{km} \otimes \mathrm{iPrunedDFT}^n_N\big)\big(\mathrm{I}_k \otimes \mathrm{iPrunedDFT}^m_M \otimes \mathrm{I}_N\big)\big(\mathrm{iPrunedDFT}^k_K \otimes \mathrm{I}_{MN}\big). \quad (30)$$

Substituting the pruned equations 29 and 30 into the left and right sides of 1 the central three expressions become



$$y = \left(\mathrm{I}_{km} \otimes \mathrm{iPrunedDFT}_N^n\right)\left(\mathrm{I}_k \otimes \mathrm{iPrunedDFT}_M^m \otimes \mathrm{I}_N\right)\left(\mathrm{PrunedCConv}_k^{f_i} \otimes_i \mathrm{I}_{MN}\right)\left(\mathrm{I}_k \otimes \mathrm{PrunedDFT}_M^m \otimes \mathrm{I}_N\right)\left(\mathrm{I}_{km} \otimes \mathrm{PrunedDFT}_N^n\right)x \tag{31}$$

Fig. 6: Optimized Hockney Free-Space Convolution illustration and derivation. LibraryX automates what was done manually as part of a PhD thesis[26].

$$\left(\mathrm{iPrunedDFT}_K^k \otimes \mathrm{I}_{MN}\right)\left(\mathrm{Diag}_f\right)\left(\mathrm{PrunedDFT}_K^k \otimes \mathrm{I}_{MN}\right). \tag{32}$$

Through the application of the block-diagonal Kronecker form rule 19 the Diag expression can be rewritten as

$$\mathrm{Diag}_f = \mathrm{L}_K^{KMN}\left(\bigoplus_{i=0}^{MN-1} \mathrm{Diag}_{f_i}\right)\mathrm{L}_{MN}^{KMN}. \tag{33}$$

Inserting this expansion into equation 32 we get

$$\left(\mathrm{iPrunedDFT}_K^k \otimes \mathrm{I}_{MN}\right)\mathrm{L}_K^{KMN}\left(\bigoplus_{i=0}^{MN-1} \mathrm{Diag}_{f_i}\right)\mathrm{L}_{MN}^{KMN}\left(\mathrm{PrunedDFT}_K^k \otimes \mathrm{I}_{MN}\right). \tag{34}$$

To further simplify this expression we can reorder the permutation matrices using rule 18 along with multiplying by the inverse permutation matrix $\left(\mathrm{L}_K^{KMN}\right)^{-1} = \mathrm{L}_{MN}^{KMN}$ on each side

$$\mathrm{PrunedDFT}_K^k \otimes \mathrm{I}_{MN} = \mathrm{L}_K^{KMN}(\mathrm{I}_{MN} \otimes \mathrm{PrunedDFT}_K^k)\mathrm{L}_{MN}^{kMN}, \tag{35}$$

$$\mathrm{L}_{MN}^{KMN}\left(\mathrm{PrunedDFT}_K^k \otimes \mathrm{I}_{MN}\right) = \mathrm{L}_{MN}^{KMN}\mathrm{L}_K^{KMN}(\mathrm{I}_{MN} \otimes \mathrm{PrunedDFT}_K^k)\mathrm{L}_{MN}^{kMN}, \tag{36}$$

$$\mathrm{L}_{MN}^{KMN}\left(\mathrm{PrunedDFT}_K^k \otimes \mathrm{I}_{MN}\right) = \mathrm{I}_{KMN}(\mathrm{I}_{MN} \otimes \mathrm{PrunedDFT}_K^k)\mathrm{L}_{MN}^{kMN}, \tag{37}$$

$$\mathrm{L}_{MN}^{KMN}\left(\mathrm{PrunedDFT}_K^k \otimes \mathrm{I}_{MN}\right) = (\mathrm{I}_{MN} \otimes \mathrm{PrunedDFT}_K^k)\mathrm{L}_{MN}^{kMN}, \tag{38}$$

similarly for the inverse case,

$$\left(\mathrm{iPrunedDFT}_K^k \otimes \mathrm{I}_{MN}\right)\mathrm{L}_K^{KMN} = \mathrm{L}_k^{kMN}(\mathrm{I}_{MN} \otimes \mathrm{iPrunedDFT}_K^k). \tag{39}$$

Substituting the results from equations 38 and 39 into equation 34 we get

$$\mathrm{L}_k^{kMN}(\mathrm{I}_{MN} \otimes \mathrm{iPrunedDFT}_K^k)\left(\bigoplus_{i=0}^{MN-1} \mathrm{Diag}_{f_i}\right)(\mathrm{I}_{MN} \otimes \mathrm{PrunedDFT}_K^k)\mathrm{L}_{MN}^{kMN}. \tag{40}$$

Now notice that the middle three terms are block diagonal with dimension MN, which means that we can simplify equation 40 to

$$\mathrm{L}_k^{kMN}\left(\bigoplus_{i=0}^{MN-1} \mathrm{iPrunedDFT}_K^k \mathrm{Diag}_{f_i} \mathrm{PrunedDFT}_K^k\right)\mathrm{L}_{MN}^{kMN}. \tag{41}$$

Using rule 19, we can rewrite the formulation for clarity as follows,

$$(\mathrm{iPrunedDFT}_K^k \mathrm{Diag}_{f_i} \mathrm{PrunedDFT}_K^k) \otimes_i \mathrm{I}_{MN}. \tag{42}$$

This exposes the key operation that a DFT followed by an element-wise multiplication (using the block-diagonal matrix) followed by an inverse DFT is actually a circular convolution, pruned in this case. Therefore, we can finally denote this term as

$$\left(\mathrm{PrunedCConv}_k^{f_i} \otimes_i \mathrm{I}_{MN}\right). \tag{43}$$

Having completed all the transformations, the final expression for Hockney Free-Space convolution is shown in equation 31.

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[26] on this computation, which requires significant manual implementation to achieve good performance.

TABLE III: 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

VI. Evaluation

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

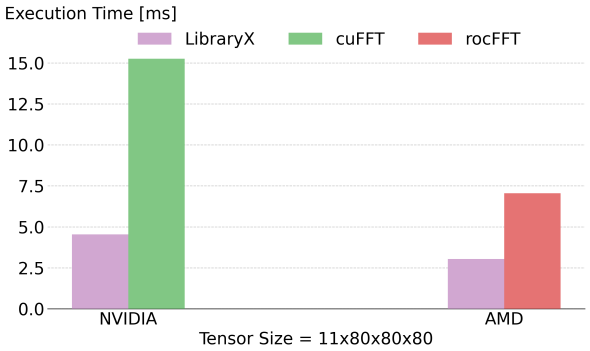


Fig. 7: PSATD Performance between LibraryX and various vendors. LibraryX achieves speedups of 3.4 times and 2.3 times across different vendor platforms.

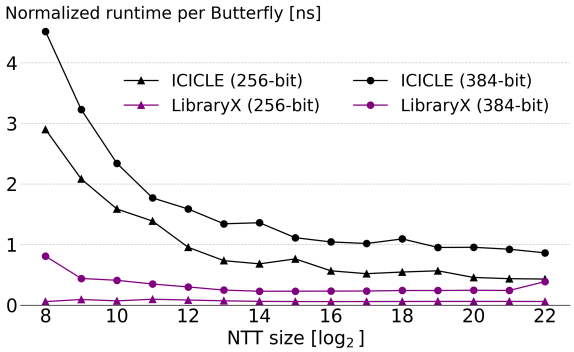


Fig. 8: Performance comparison of NTTs between ICICLE and LibraryX on the Nvidia H100. For different bit-widths and NTT sizes LibraryX significantly outperforms ICICLE.

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 III.

A. Applications

PSATD. The Pseudo-spectral-analytical time-domain method[27] is a computational solver in WarpX[28], an advanced electrostatic and electromagnetic 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 point, is extracted and multiplied with a unique sparse matrix generated at each index point. This result is then inverse FFT'd 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 cryptographic 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 compressible Euler equations are used in the study of gas dynamics and take the form shown below

$$\frac{\partial U}{\partial t} + \nabla \cdot \vec{F}(U) = 0. \tag{44}$$

A fourth-order finite volume method [29] can be used to solve equation 44 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 $\mathscr{G}$. A mathematical specification for counting these triangles is given as

$$\Delta = \frac{1}{6}\Gamma(A^3), \tag{45}$$

where A is a symmetric adjacency matrix representing the input graph $\mathscr{G}$[30] and Γ is the trace operation, the sum of the elements along the main diagonal. As A is a sparse matrix, developers can use sparse linear algebra libraries like GraphBLAS[31][32][33] to implement this and other graph analytics algorithms.

VII. Results

We show the results of the LibraryX implementation against vendor libraries or state-of-the-art tools for each application. We measure kernel 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 against the vendor library implementation in Fig. 7. We see that LibraryX outperforms the vendor implementation across both vendor platforms achieving speedups of 3.4 times, and 2.3 times respectively. There 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

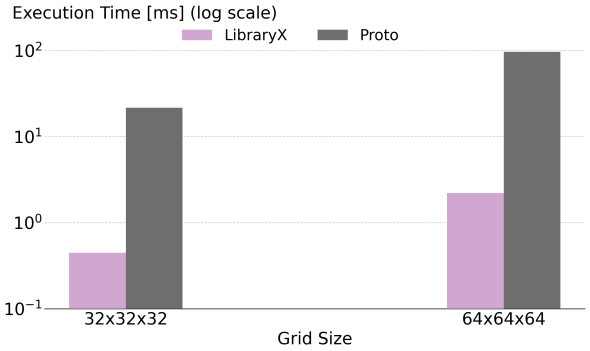


Fig. 9: Performance comparison of the spatial discretization of the 3D Euler Equations between LibraryX and Proto library on the Nvidia Titan V. LibraryX significantly outperforms Proto.

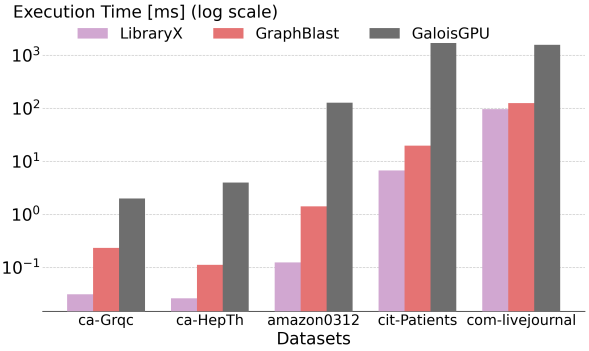


Fig. 10: Performance comparison between state-of-the-art graph processing frameworks and LibraryX for Triangle Counting on an Nvidia Titan V. LibraryX outperforms both tools for a range of datasets.

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 [22][23]. 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-the-art library for high performance cryptographic acceleration called ICICLE [34]. ICICLE has a very good generalization of NTTs of varying bit widths. Figure 8 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 13 times for 256-bit and 4.8 times for 384-bit. LibraryX enables developers to use the general ICICLE interface 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 [35]. 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 computational components, stencils, and pointwise operations [21]. Figure 9 shows the performance of LibraryX and Proto for Euler equations for two different 3D input grids of sizes 32 and 64 on a Nvidia Titan V. LibraryX outperforms Proto by 48 times and 43 times 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[36] and Galois[37]. The GraphBLAST framework is an extension of GraphBLAS[32] 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 10 shows the performance across each framework on a variety of datasets. These datasets range from small graphs of $\sim$5 thousand 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 5 times faster than GraphBLAST and around 200 times faster than Galois. LibraryX uses SPIRAL's triangle counting implementation [18], which is based on optimizations found in previous work [38][39]. 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.

VIII. Limitations and Overhead

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 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, acting only as inputs/outputs to library functions.

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 as it's primarily performing data serialization. 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.

IX. Related Work

Automated Library Optimization There have been many efforts to automate library-based program optimization. Many modern software libraries use a design strategy called generative programming[40]. Generative programming treats software components as generic specifications that are heavily parameterized. During program execution, these library calls are instantiated with a highly optimized implementation determined by the input parameters. Active Libraries[41] leverages generative programming for all library components, providing flexibility in optimizations. Many software libraries [42][43][44][45][46][47] use the techniques described in Active Libraries. Telescoping languages[48] created a system to precompute and store a large number of optimizations for various domain-specific scripting libraries. In this way, when a programmer used various high-level library calls, the database of optimizations could be queried to find the best-performing implementation of that library call sequence. This separates the design of the library implementation from the optimization of the computation. The Broadway compiler[49] is a compiler infrastructure used to optimize library-based programs by exploiting the semantics of the library domain. Using domain-expert annotations, Broadway is able to find opportunities to optimize library call sequences through dataflow analysis.

DSL Compilers. There are several domain-specific compilers [5][7][8][6][16][11] and frameworks that are used to optimize implementations for a variety of different domains. These frameworks expose domain-specific operators that can be composed to create complex computations. This expression can then be optimized through proper scheduling and implementation decisions to provide high-performance on various hardware platforms. In the domain of machine learning and deep learning, these compilers[50][51] are able to recognize and optimize computational DAGs at compile time[52]. Additionally, these compilers serve as backends to popular machine learning frameworks[53][54], automatically transforming operator sequences into optimized implementations that can be executed on various hardware platforms.

Discussion. LibraryX shares many of the goals described in Active Libraries and Telescoping Languages. LibraryX differs from these approaches by providing a hybrid compile time and runtime approach, using preprocessor directives to gather library call semantics and runtime code generation/compilation to generate optimized variants. While preprocessor directives serve a similar purpose to annotations as used by Broadway, LibraryX is able to hide these in header files without modifying user code directly. This provides a separation of concerns for application developers using LibraryX. Furthermore, the mapping of library calls to domain-specific operators is not done by other domain-specific compilers, a burden to leverage their performance improvements. In the machine learning space, this mapping does exist, but their optimizations are generally done at compile-time and are tightly coupled to the high-level framework implementing the computation. The key benefit of LibraryX is through its ability to optimize applications across not only the library call boundary but also the domain library boundary. As far as we know, LibraryX is the only system that can automatically optimize applications written using multiple domain libraries concurrently.

X. Conclusion

This paper presents LibraryX, a framework for optimizing scientific applications written against multiple domain-specific libraries, with a focus on low arithmetic intensity, memory-bound applications. LibraryX achieves runtime full-program optimization by recognizing and replacing sequences (DAGs) of library calls with a unified representation that captures higher-level computational semantics, enabling cross-library optimizations impossible with isolated library calls. LibraryX utilizes SPIRAL to generate an architecture-optimized implementation which is dynamically compiled and executed in place of the original implementation. We demonstrate LibraryX's effectiveness across spectral algorithms/FFTs, stencils/PDEs, and sparse linear algebra/graph algorithms, achieving significant speedups over vendor libraries on modern accelerators without source code modification, enabling write-once, run-anywhere performance portability.

LibraryX's approach of treating library calls as specifications rather than opaque implementations opens new opportunities for automated optimization in scientific computing. By leveraging SPIRAL's domain-specific transformation knowledge, LibraryX shows that developers need not sacrifice performance for productivity. Future work includes extending domain coverage to additional mathematical libraries and exploring automatic performance tuning based on application-specific usage patterns.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration under Contract No. 7421006. This research was also supported by Bluestone, an X-Stack project in the DOE Advanced Scientific Computing Office under Contract No. DE-SC0022275 as well as Durban and Fairbanks, "Advancements in Artificial Intelligence for Science" projects under Contract Nos. DE-SC0025645 and CW62109. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, p. 308–323, Sep. 1979. [Online]. Available: https://doi.org/10.1145/355841.355847

[2] 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.*, vol. 14, no. 1, p. 1–17, Mar. 1988. [Online]. Available: https://doi.org/10.1145/42288.42291

[3] A. Buluç and J. R. Gilbert, "The combinatorial blas: design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, p. 496–509, Nov. 2011. [Online]. Available: https://doi.org/10.1177/1094342011403516

[4] 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*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[5] 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*, ser. CGO '21. IEEE Press, 2021, p. 2–14. [Online]. Available: https://doi.org/10.1109/CGO51591.2021.9370308

[6] 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," vol. 48, no. 6, 2013.

[7] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: a high-performance graph dsl," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276491

[8] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133901

[9] 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*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 286–300. [Online]. Available: https://doi.org/10.1145/3519939.3523437

[10] ——, "Spdistal: Compiling distributed sparse tensor computations," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[11] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: A functional data-parallel ir for high-performance gpu code generation," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 74–85.

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

[13] S. Marlow *et al.*, "Haskell 2010 language report," *Available online http://www. haskell. org/(May 2011)*, 2010.

[14] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, "The ocaml system: Documentation and user's manual," *INRIA*, vol. 3, p. 42.

[15] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 519–538. [Online]. Available: https://doi.org/10.1145/1094811.1094852

[16] F. Franchetti, T.-M. Low, T. Popovici, R. Veras, D. G. Spampinato, J. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, "SPIRAL: Extreme performance portability," *Proceedings of the IEEE, special issue on "From High Level Specification to High Performance Code"*, vol. 106, no. 11, 2018.

[17] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel, "Operator language: A program generation framework for fast kernels," in *Domain-Specific Languages*, W. M. Taha, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 385–409.

[18] S. Rao, A. Kutuluru, P. Brouwer, S. McMillan, and F. Franchetti, "GBTLX: A First Look," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–7.

[19] 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*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 127–138. [Online]. Available: https://doi.org/10.1145/2491956.2462187

[20] 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*, 2013, pp. 13–24.

[21] 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)*, 2023, pp. 1–6.

[22] 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)*, 2023, pp. 1–7.

[23] 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.

[24] 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*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 315–326. [Online]. Available: https://doi.org/10.1145/1065010.1065048

[25] F. Franchetti and M. Puschel, "Generating high performance pruned fft implementations," in *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2009, pp. 549–552.

[26] B. Van Straalen, "Method of local corrections solver for manycore architectures," Ph.D. dissertation, EECS Department, University of California, Berkeley, Aug 2018. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-122.html

[27] 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*, vol. 279, p. 108457, Oct. 2022. [Online]. Available: http://dx.doi.org/10.1016/j.cpc.2022.108457

[28] 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*, ser. SC '22. IEEE Press, 2022.

[29] 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*, vol. 6, no. 1, pp. 1–25, 2011.

[30] P. Burkhardt, "Graphing trillions of triangles," *Information Visualization*, vol. 16, 09 2016.

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

[32] T. G. Mattson, C. Yang, S. McMillan, A. Buluç, and J. E. Moreira, "Graphblas c api: Ideas for future versions of the specification," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–6.

[33] S. McMillan, "GraphBLAS Template Library(GBTL)," https://github.com/cmu-sei/gbtl.

[34] 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. [Online]. Available: https://eprint.iacr.org/2024/973

[35] "Proto Github," https://github.com/applied-numerical-algorithms-group-lbnl/proto.

[36] C. Yang, A. Buluç, and J. D. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu," *ACM Trans. Math. Softw.*, vol. 48, no. 1, feb 2022. [Online]. Available: https://doi.org/10.1145/3466795

[37] 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*, 2013, pp. 456–471.

[38] 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*, ser. Correctness'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 14–20. [Online]. Available: https://doi.org/10.1145/3145344.3145484

[39] M. P. 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)*, 2019, pp. 1–7.

[40] 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*. Berlin, Heidelberg: Springer-Verlag, 1998, p. 25–39.

[41] T. L. Veldhuizen and D. Gannon, "Active libraries: Rethinking the roles of compilers and libraries," 1998. [Online]. Available: https://arxiv.org/abs/math/9810022

[42] 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*, vol. 74, no. 12, pp. 3202–3216, 2014.

[43] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

[44] M. Frigo and S. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[45] NVIDIA, "cuBLAS, the CUDA Fast Fourier Transform library," 2024, [Online; accessed 31-December-2024]. [Online]. Available: https://docs.nvidia.com/cuda/cufft/

[46] AMD, "rocFFT, the AMD Fast Fourier Transform library," 2024, [Online; accessed 31-December-2024]. [Online]. Available: https://rocm.docs.amd.com/projects/rocFFT/en/latest/

[47] Intel, "The Intel oneAPI Math Kernel Library," 2024, [Online; accessed 31-December-2024]. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html?s=Newest

[48] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey, "Telescoping languages: A system for automatic generation of domain languages," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 387–408, 2005.

[49] S. Guyer and C. Lin, "Broadway: A compiler for exploiting the domain-specific semantics of software libraries," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 342–357, 2005.

[50] A. Sabne, "Xla : Compiling machine learning for peak performance," 2020.

[51] 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)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[52] 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*, vol. 32, no. 3, p. 708–727, Mar. 2021. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2020.3030548

[53] 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*, ser. OSDI'16. USA: USENIX Association, 2016, p. 265–283.

[54] 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*. Red Hook, NY, USA: Curran Associates Inc., 2019.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

XI. Overview of Contributions and Artifacts

A. Paper's Main Contributions

This paper discusses the following contributions:

- C_1 Lifting of single and multi-domain library-based algorithms from implementations to specifications.
- C_2 Expression of mathematical library semantics in a cross-library domain-specific language, enabling the generation of optimized cross-library implementations.
- C_3 Automatic full-solver code synthesis and optimization across library boundaries.
- C_4 Performance portability of algorithms written against LibraryX across GPUs from different vendors, demonstrating significant speed-up over vendor library based implementations.

B. Computational Artifacts

The main contribution of this research is LibraryX software framework.

- A_1 LibraryX: https://github.com/sr7cb/LibraryX
- A_2 SPIRAL: https://github.com/spiral-software/spiral-software
- A_3 FFTW: https://www.fftw.org/
- A_4 cuFFT: https://developer.nvidia.com/cufft
- A_5 rocFFT https://rocm.docs.amd.com/projects/rocFFT/en/latest/
- A_6 Intel MKL FFT: https://tinyurl.com/inteloneapimkl
- A_7 Proto: https://github.com/applied-numerical-algorithms-group-lbnl/proto
- A_8 ProtoX: https://doi.org/10.1109/HPEC58863.2023.10363547
- A_9 ICICLE: https://dev.ingonyama.com/
- A_{10} NTTX: https://www.spiral.net/software/nttx.html
- A_{11} GraphBlast https://github.com/gunrock/graphblast
- A_{12} GaloisGPU https://github.com/IntelligentSoftwareSystems/GaloisGPU
- A_{13} FFTX: https://github.com/spiral-software/fftx

Artifact ID	Contributions Supported	Related Paper Elements
A_1, A_2, A_4, A_5, A_6	C_4	Figure 1
A_1, A_2	C_1, C_2	Figure 2
A_1, A_2, A_{13}	C_3	Figure 3
A_3, A_7	C_1, C_4	Figure 4
A_1, A_2	C_2, C_4	Figure 5
A_2, A_{13}	C_2, C_3	Figure 6
A_1, A_2, A_4, A_5	C_4	Figure 7
A_1, A_9, A_{10}	C_4	Figure 8
A_1, A_7, A_8	C_4	Figure 9
A_1, A_{11}, A_{12}	C_4	Figure 10

XII. Artifact Identification

A_1: Artifact 1 represents the LibraryX package which is used to gather many computational results for this work. LibraryX provides the mechanisms to shim libraries and convert the library implementation to the SPIRAL SPL formulation. The time to reproduce the artifact is 5 minutes to build, and an hour to run all the experiments and 5 minutes to analyze the output performance. We currently support the GPUs from all vendors NVIDIA, AMD, and Intel. The inputs are generated for each experiment in the program. The general flow is to download the package and build the specific experiment and run the experiment to gather the results. The data points are configured based on real world sizes of interest. While LibraryX is a proof-of-concept many of its ideas were integrated into the open-source FFTX library.

A_2: Artifact 2 represents the SPIRAL code generation system. SPIRAL is used for performance portability and utilization by generating different computation kernels and is demonstrated by showing performance improvements over library baselines. The time to reproduce the artifact is 5 minutes to build, 30 minutes to run all experiments and 5 minutes for artifact analysis. We currently support GPUs from all major platforms, NVIDIA, AMD, and Intel. The general flow is to download the package and build it, putting it in the environment path. The inputs are generated for each experiment in the program.

A_3: Artifact 3 represents the FFTW package. This is general purpose FFT library available on most systems containing a number of FFT implementations. The time to reproduce this artifact is 5 minutes to download it using the package manager of your choosing or downloading online. The purpose of FFTW is to leverage its easy to use interface to get accessed to other accelerator systems and is not part of the computational results.

A_4: Artifact 4 represents the cuFFT package. This is the NVIDIA provided library to perform FFTs on NVIDIA GPUs supporting a range of sizes, types, and precisions. CuFFT is used as a comparison to LibraryX, and to demonstrate performance improvements when integrated in LibraryX. The

time to reproduce this artifact is the 10 minutes to download the CUDA toolkit which contains cuFFT. CuFFT comes prebuilt so you only need to link against it when building the experiments before running them. The inputs are generated for each experiment in the program. The general flow is to download the package and put it in environment path during compilation and execution.

A_5: Artifact 5 represents the rocFFT package. This is the AMD provided library to perform FFTs on AMD GPUs supporting a range of sizes, types, and precisions. RocFFT is used as a comparison to LibraryX, and to demonstrate performance improvements when integrated in LibraryX. The time to reproduce this artifact is the 10 minutes to download the rocm package which contains rocFFT. RocFFT comes prebuilt so you only need to link against it when building the experiments before running them. The inputs are generated for each experiment in the program. The general flow is to download the package and put it in environment path during compilation and execution.

A_6: Artifact 6 represents the Intel MKL FFT package. This is the Intel provided library to perform FFTs on Intel GPUs supporting a range of sizes, types, and precisions. MKL FFT is used as a comparison to LibraryX, and to demonstrate performance improvements when integrated in LibraryX. The time to reproduce this artifact is the 10 minutes to download the oneAPI MKL package which contains MKL FFT. MKL FFT comes prebuilt so you only need to link against it when building the experiments before running them. The inputs are generated for each experiment in the program. The general flow is to download the package and put it in environment path during compilation and execution.

A_7: Artifact 7 represents the Proto library. Proto provides an easy to use library for writing PDE solvers. Proto is used as a comparison to LibraryX, and to demonstrate performance improvements when integrated in LibraryX. The time to reproduce the artifact is 5 minutes to build, 30 minutes to run all experiments and 5 minutes for artifact analysis. Proto currently support, CPUs, and GPUs from NVIDIA and AMD. The inputs are generated for each experiment in the program. The general flow is to download the package and build it then run the provided Euler example.

A_8: Artifact 8 represents ProtoX a package for SPIRAL to generate high-performance kernels. ProtoX is used for performance portability and utilization by generating analogous computation kernels and is demonstrated by showing performance improvements over library baselines. The time to reproduce the artifact is 5 minutes to download. We currently support, CPUs, and GPUs from all major platforms, NVIDIA, AMD, and Intel. The inputs are generated for each experiment in the program. The general flow is to download the package within the SPIRAL source tree.

A_9: Artifact 9 represents the ICICLE package. ICICLE provides library abstractions for cryptography along with fast implementations of the NTT on NVIDIA GPUs. ICICLE is used as a comparison to LibraryX, and to demonstrate performance improvements when integrated in LibraryX. The time to reproduce this artifact is the 10 minutes to download and build the ICICLE package. The inputs are generated for each experiment in the program. The general flow is to download the package, build it, and run the provided NTT examples in the package.

A_{10}: Artifact 10 represents NTTX a package for SPIRAL to generate high-performance kernels. NTTX is used for performance portability and utilization by generating analogous computation kernels and is demonstrated by showing performance improvements over library baselines. The time to reproduce the artifact is 5 minutes to download. We currently support NVIDIA GPUs. The inputs are generated for each experiment in the program. The general flow is to download the package within the SPIRAL source tree.

A_{11}: Artifact 11 represents the GraphBLAST package. GraphBLAST is a high performance GPU implementation of the GraphBLAS specification for graph algorithms. GraphBLAST is used as a comparison to LibraryX, and to demonstrate performance improvements when integrated with LibraryX. The time to reproduce this artifact is the 20 minutes to download and build the GraphBLAST package. The inputs come from the SNAP real-world graph dataset. The general flow is to download and build the package and run the provided triangle counting script.

A_{12}: Artifact 12 represents the GaloisGPU package. GaloisGPU is a high performance graph analytics package that runs on NVIDIA GPUs. GaloisGPU is used as a comparison to LibraryX, and to demonstrate performance improvements when integrated with LibraryX. The time to reproduce this artifact is the 20 minutes to download and build the GaloisGPU package. The inputs come from the SNAP real-world graph dataset. The general flow is to download and build the package and run the provided triangle counting application.

A_{13}: Artifact 13 represents the FFTX package for SPIRAL to generate high-performance kernels. FFTX is used for performance portability and utilization by generating different computation kernels and is demonstrated by showing performance improvements over library baselines. The time to reproduce the artifact is 5 minutes to download. We currently support, CPUs, and GPUs from all major platforms, NVIDIA, AMD, and Intel. The inputs are generated for each experiment in the program. The general flow is to download the package within the SPIRAL source tree.