Towards LibraryX: A Framework for Cross-Library Call Optimization

Sanil Rao, Anant Prakash, Franz Franchetti
Department of Electrical and Computer Engineering
{sanilr,anantpra,franzf}@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

***Abstract*—In scientific computing, key algorithms are implemented using domain specific libraries. In many cases these algorithms cannot be expressed using just one domain library, instead having to use library calls from various domains. Cross library implementations are productive but suffer in performance because library developers cannot optimize library calls outside their domain. Additionally, automated solutions like compilers struggle to optimize these cases as they cannot cross the library call boundary. This leaves manually implementing optimized algorithms for idealized performance, hindering readability and productivity. We propose LibraryX as a framework to express key algorithms using domain libraries while automatically providing high-performance implementations done by hand. LibraryX uses a combination of library call semantic capture, abstraction lifting/code generation, and runtime compilation to provide optimized implementations without modifying the source application. We demonstrate LibraryX using the example of FFT convolution.**

I. Introduction

Domain-specific libraries have long been the standard for performance in scientific computing. Library developers focus on the performance of each of their operators while applications developers use those operators to clearly express their computations. This offers a clear separation of concerns for library developers and application developers. However, as we near the limits of Moore's Law, this this separation is fading. Large intermediate buffers and significant data movement between domain libraries has made relying solely on optimized kernels within a domain insufficient. Performance experts are required to optimize computations by fusing kernels across domains, stripping away library calls. This aids in performance at the cost of readability and programmer productivity.

This tradeoff between productivity and performance will only continue to get more challenging. The cambrian explosion of computer architectures further exacerbates application complexity. Each new architecture requires new language extensions and programming models to utilize effectively. This in turn requires complex control flow or specialized execution engines, adding additional build complexity. While portability layers aim to resolve many of these issues they struggle to achieve similar performance to hand written codes.

We propose LibraryX as a solution to this complicated search space of performance and productivity. LibraryX can optimize scientific applications by recognizing known computation patterns and replacing them with optimized variants. This is done by treating library calls as specifications rather than implementations. LibraryX utilizes a combination of lazy evaluation to capture the computation's semantics, abstraction lifting to recognize computations, code generation to produce an architecturally optimized kernel, and runtime compilation to replace the original library based implementation without modification to the source implementation.

II. Example: FFT Convolution

Convolution is an important kernel in scientific computing especially in the area of spectral methods, which have large inputs at scale. For large inputs the FFT convolution, with complexity $O(n \log n)$, is more attractive than direct convolution which has complexity $O(mn)$ where m is the filter matrix which is generally much smaller than n. Figure 1 shows the source code for an FFT convolution using various domain libraries. It consists of three function calls a forward real-to-complex FFT, a complex multiplication of the FFT output with the second input and an inverse complex-to-real FFT.

While easy to read and modify for different architectures this implementation can be significantly optimized. The temporary buffer for the complex multiply is not needed. Additionally the memory traffic overhead can be reduced by using the result of the FFT immediately as its being computed [3]. Unfortunately, these optimizations are difficult to perform without significantly modifying the source code and inspecting the implementation of each library call. We demonstrate how LibraryX is able to provide these optimizations without changing the source application.

Delayed Execution. To understand the computation we first need to capture its semantics. 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 function to expose its semantics details and generate a dataflow graph of the given computation. This side effect representation is the Operator Language (OL) formulation that will be used by the SPIRAL [1] code generation system for analysis and optimization. Once complete, the function `checkOutput` acts as the execution trigger, invoking abstraction lifting, code generation and runtime compilation of the optimized implementation. For convolution this means we generate an OL expression for each of its components, the FFT, pointwise multiply, and inverse FFT.

```cpp
#include <iostream>
#include <complex>
#include <vector>
#include <algorithm>
#include "fftw3.h"
#include "libraryX.hpp"
using namespace std;

int main() {
  int N = 1024;
  vector<double> input(N*N*N);
  vector<double> output(N*N*N);
  vector<complex<double>> input2(N*N*N);
  vector<complex<double>> temp(N*N*N);
  vector<complex<double>> out(N*N*N);
  buildInput(input);
  buildInput(input2);

  fftw_plan p = fftw_plan_dft_r2c_3d(N, N, N,
    input.data(), (fftw_complex*)out.data(),64);
  fftw_execute(p);

  auto complex_multiply = std::multiplies<>{};
  std::transform(out.begin(), out.end(), //range
    input2.begin(), //second input
    temp.begin(), //output
    complex_multiply); //operator

  fftw_plan p2 = fftw_plan_dft_c2r_3d(N, N, N,
    (fftw_complex*)temp.data(), output.data(),64);
  fftw_execute(p2);

  checkOutput(output);
}
```

Fig. 1. FFT convolution example. This C++ code is transparently executed on an AMD GPU after it is dynamically translated to HIP.

Abstraction Lifting and Code Generation. After a sequence of operations is captured as an OL DAG its needs to be recognized. We leverage SPIRAL's extensive pattern matching engine to discover if the input OL DAG is a known pattern. This pattern matching engine takes information like input size and dataflow to find the best expression available. If successful, the input OL DAG will be lifted into a single OL expression encapsulating the computation.

The lifted OL convolution goes through SPIRAL's OL transformation hierarchy which enables optimizations at different levels of abstraction. This includes different breakdowns for algorithm selection, loop merging, and index simplification. For convolution this would be optimizations such as removing the temporary and pipelining computations. The optimized expression is then lowered to SPIRAL's intermediate representation where traditional compiler optimizations are performed, such as strength reduction and copy propagation. The final implementation can be generated for various hardware platforms and ISAs.

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 a hardware abstraction layer (HAL). HAL parses the metadata of the SPIRAL generated code to compile and execute on a given target platform such as CPUs or GPUs. This metadata includes external temporary memory buffers, thread blocks and grids, and kernel invocation order. Once the metadata is parsed HAL uses vendor specific runtime compilation to dynamically link and execute the kernel using the user provided input and output buffers as arguments.

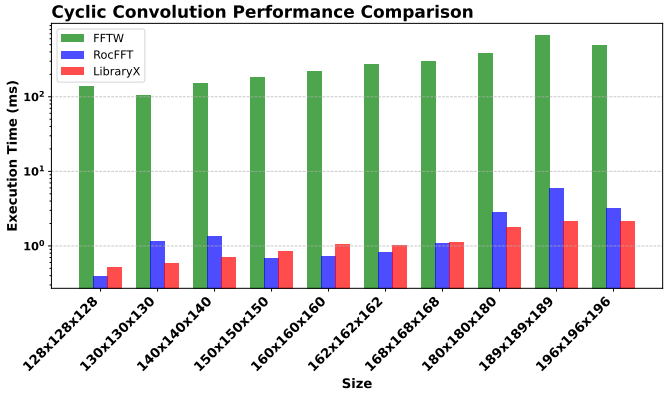

Fig. 2. Performance comparison between a baseline, vendor, and LibraryX implementation of cyclic convolution. LibraryX is competitive with the vendor implementation, outperforming them for sizes with larger prime factors.

III. Conclusion and Future Work

LibraryX is a framework for cross-library call optimization in scientific applications. Through a combination of semantic capture, abstraction lifting, code generation, and runtime compilation, LibraryX is able to optimize scientific applications written against libraries without changing the source application. LibraryX translates an application from a direct execution model to a lazy evaluation model in order to understand the computations' semantics. The derived computation graph is given to the SPIRAL code generation system for analysis and optimization, producing a high-performance kernel for a variety of hardware platforms. LibraryX's backend is then able to execute the application on a configured target architecture in place of the original execution, transparently populating the output buffer. LibraryX is extensible, supporting various vendor runtime and compilation systems but also other types of runtime systems such as the IRIS [2] runtime system. We plan to extend LibraryX to target other application frontends such as Python and Fortran as well as other domains like cryptography [5] and graph processing [4].

References

[1] 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.

[2] 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.

[3] D. T. Popovici, A. Canning, Z. Zhao, L.-W. Wang, and J. Shalf. A systematic approach to improving data locality across fourier transforms and linear algebra operations. In *Proceedings of the 35th ACM International Conference on Supercomputing*, ICS '21, page 329–341, New York, NY, USA, 2021. Association for Computing Machinery.

[4] 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.

[5] 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.