LibraryX-ASIC: A First Look

Sanil Rao, Larry Tang, Franz Franchetti
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA
{sanilr, lawrenct, franzf}@andrew.cmu.edu

***Abstract*—Domain-specific accelerators have become the new frontier for increasing computation performance in modern computing systems. Accelerators provide dedicated hardware support for key operations in their domain. However, these accelerators require significant effort to program, with each new accelerator having its own programming characteristics. This makes it difficult for programmers to take advantage of their performance. To overcome this difficulty, we introduce a new framework for accelerator programming development called LibraryX-ASIC. LibraryX-ASIC leverages domain specific software libraries like FFTW or C++ STL as a way to program accelerators, while offloading the actual computation to the new accelerator. We show the system design of LibraryX-ASIC using the example of an FFT library and an FFT accelerator.**

***Index Terms*—Code Generation, FFT, SPIRAL, ASIC, Accelerator**

I. Introduction

With the end of Dennard scaling, hardware developers have turned to building domain-specific accelerators to increase computing performance. These accelerators provide two key benefits over general purpose hardware: performance and energy efficiency. Accelerators have custom circuits for critical operations in their domain, enabling increased energy efficiency for their workloads.

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

To address these issues of programmability and portability of accelerator devices, we propose LibraryX-ASIC. LibraryX-ASIC is an automated framework designed to hide the complexity of accelerator offload behind domain-specific software libraries like FFTW [1] for FFTs, or C++ Standard Template Libraries (STL). Using the library's standard interface, LibraryX-ASIC recognizes the operation, generates an optimized accelerator implementation, and offloads it to the accelerator automatically, populating the output buffer upon completion. This relieves the programmer of having to worry about various accelerator offload paradigms. We show how LibraryX-ASIC can be utilized through an FFT benchmark and an FFT accelerator.

Contributions. This paper makes the following contributions:

- Introduces the LibraryX-ASIC framework for automatic accelerator offload.
- Provides a system walkthrough of the framework using an FFT example.
- Provides preliminary results compared to software implementations.

II. Background

A. SPIRAL

The SPIRAL [2] code generation system is a domain specific language and compiler for Fast Fourier Transform (FFT) algorithms. Its internal language, Signal Processing Language (SPL), uses linear transforms to express key operations in FFT computations. These operators are composed using a pointfree matrix vector representation where an implicit input vector x gets multiplied by an SPL operator matrix, producing an implicit output vector y. SPL then gets broken down using SPIRAL code transformation layers, producing optimized code for a variety of hardware platforms including, CPUs, GPUs, FPGAs, and ASICs. LibraryX-ASIC leverages SPIRAL to produce equivalent optimized accelerator code for FFT library calls.

B. FFT Accelerator

The FFT accelerator is designed to address two primary challenges — flexibility and programmability, in existing FFT hardware implementations by following the design principles outlined in [3]. The key observation is that software flexibility in libraries like FFTW [1] arises from recursion where the base cases are small sized FFTs known as *codelets*. Thus flexibility in hardware can be retained by designing highly configurable hardware codelets and a surrounding architecture that orchestrates their execution.

Similar to an FFTW plan, the FFT computation on the accelerator is defined by a sequence of descriptors containing the configuration parameters for the hardware codelet. Descriptors are fetched from a local instruction memory and then decoded to obtain configuration parameters including input/output base address and stride, batch size, FFT radix, and compute ordering. The codelet datapath can be reconfigured to compute different small sized FFTs and also reordered between element-wise multipliers and a transposer. Details of the accelerator microarchitecture are presented in Fig. 1. The

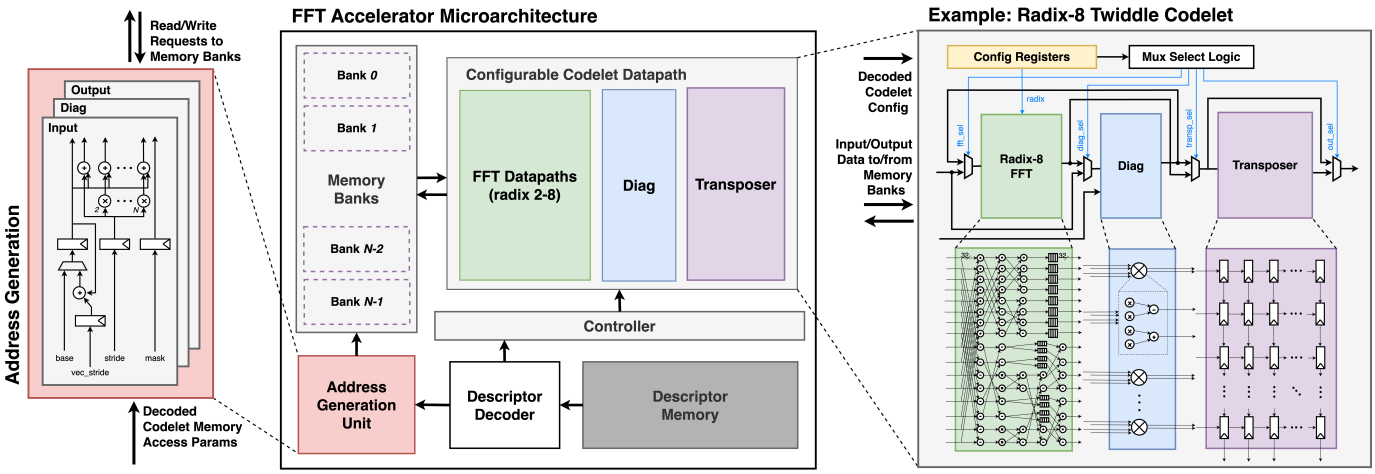

Fig. 1: Overview of the FFT accelerator microarchitecture.

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

  buildInput(input);

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

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

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

Fig. 2: Example FFT program. This FFT application written against FFTW will be executed on an FFT acclerator without user modification using LibraryX-ASIC. The output buffer will be populated as if nothing changed.

codelet datapath has been silicon-verified in an FFT accelerator [4] consisting of a radix-8 FFT core and eight element-wise multipliers to accelerate a radix-8 *twiddle codelet* of FFTW.

III. End-To-End Example: FFT

We describe the LibraryX-ASIC system design using a simple FFT program shown in Fig. 2. We discuss how FFT library calls are recognized and captured. We then describe the high-level process to generate equivalent accelerator code using SPIRAL. Finally, we show how that code is compiled and executed on the accelerator.

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

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

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

Fig. 3: SPIRAL generated code for the FFT accelerator. This code uses information from SPIRAL's FFT description include input ranges and input and output strides.

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

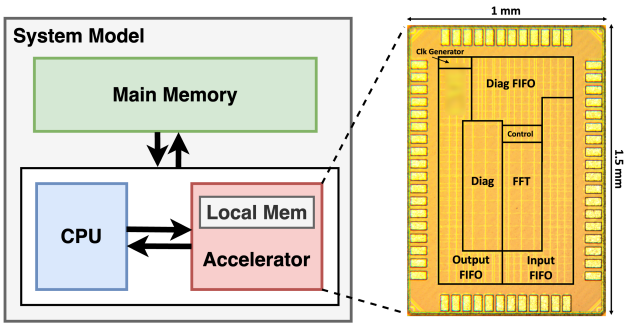

Fig. 4: High level abstract system model of the CPU-accelerator system. The model consists of a CPU controller coupled to the accelerator, fast on-chip local memory, and main memory. The micrograph shows a silicon-verified FFT accelerator implementing a part of the architecture in Fig. 1.

semantics. This SPL expression is then passed to the SPIRAL system for code generation.

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

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

These instructions are called internal code, an intermediate representation similar to other general-purpose compilers. The FFT accelerator exposes an intrinsic C library for computation offload, and SPIRAL produces the intrinsic code for the FFT computation. Along with the actual operation, SPIRAL also produces the setup code to move the pointers from the host device to the accelerator, and performs memory cleanup once the operation is complete. An example generated FFT implementation is shown in Fig. 3.

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

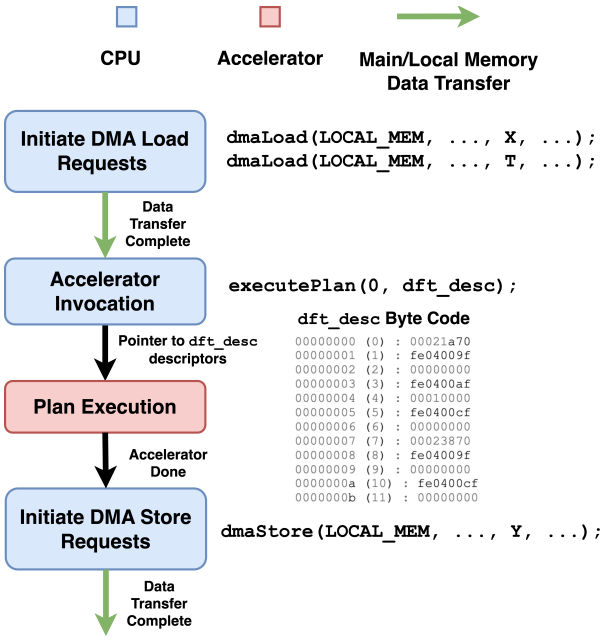

Fig. 5: Flow chart demonstrating the execution of the generated accelerator program.

in Fig. 4. The three main components are (1) main memory with a mechanism for data transfer, (2) the accelerator which interfaces to fast on-chip local memory, and (3) a controller CPU coupled to the accelerator.

Fig. 5 now walks through the execution of the generated program. The program first initiates input data transfer from main memory to the accelerator's local memory with the function call to `dmaLoad()`. Once data transfer is complete, the accelerator is invoked from the CPU and passes a memory pointer to the base address of the descriptor array, `dft_desc`. The accelerator program is defined by this array that constructs the byte code that programs the accelerator. The accelerator then fetches, decodes, and executes all descriptors in the program. Finally, the accelerator signals completion to the CPU, and a DMA store request is issued to transfer output data from local memory back to main memory. At function call exit the accelerator output now exists in the output buffer.

IV. Experimental Results

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

Experimental Setup. We show both CPU and GPU performance results as baselines to compare against LibraryX-ASIC. On CPU, we run FFTW on a 20-core Intel Xeon E5-2698v4 and for GPU evaluation, we run cuFFT on an Nvidia H100. Accelerator performance results are based on a cycle-accurate accelerator model that is calibrated against real silicon measurements from test chips [3], [4] taped on a TSMC 28nm process. The performance model simulates the abstract machine model shown in Fig. 5, where the controlling CPU is a single core of the Intel Xeon E5-2698 CPU, main memory consists of 256 GB of RDIMM DDR4, and the accelerator

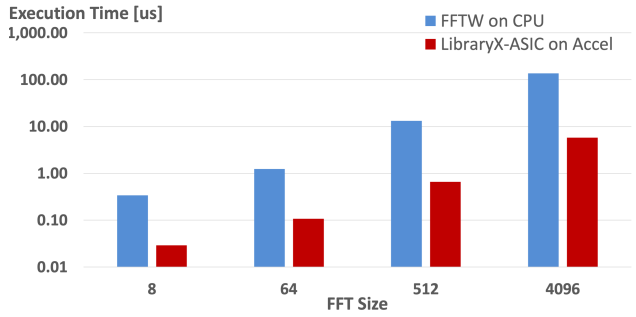

Fig. 6: FFTW on CPU vs. LibraryX-ASIC on accelerator system.

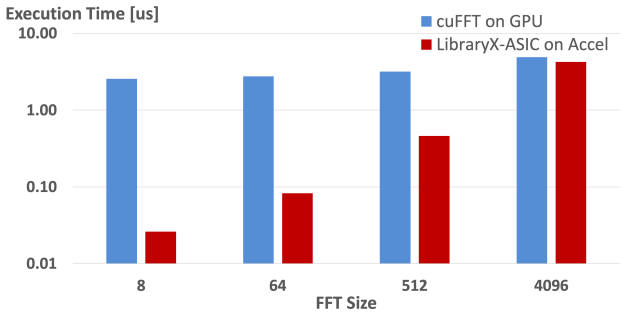

Fig. 7: cuFFT on GPU vs. LibraryX-ASIC on accelerator system.

interfaces to 256 kB of banked, SRAM-based local memory. The FFT accelerator core accelerates a radix-8 twiddle codelet. For GPU comparisons, we target off-chip HBM3e DRAM with the same accelerator configuration.

Results. The execution times in microseconds are shown in Fig. 6 and 7 for power-of-8 1D complex FFTs ranging from 8 to 4096. LibraryX-ASIC targeting the CPU-accelerator system achieves speedups of 11x – 23x as compared to running FFTW on CPU only. The performance results demonstrate an order of magnitude improvement in execution time for a real FFT program running on the custom FFT ASIC through the LibraryX-ASIC framework. Compared to cuFFT running on an H100 GPU, LibraryX-ASIC also provides up to an order of magnitude speedup at smaller size FFTs. Improved speedup against the GPU at larger FFT sizes can be achieved by scaling the number of hardened codelets in the FFT accelerator. These results demonstrate the LibraryX-ASIC framework automatically targeting a custom FFT accelerator through CPU and GPU FFT library calls.

V. Related Work

Accelerator Programming. The Fourier ACcelerator Compiler (FACC) [5] is a compiler that translates C FFT implementations to various FFT accelerators. FACC uses a constraint system to generate equivalent implementations of FFTs by modifying incorrect types, loops, and other code structures. It uses a generate and test system to create different implementations and verify correctness. LibraryX-ASIC shares the same idea as FACC to be a drop-in replacement for targeting accelerators. However, they differ in their respective approaches, as LibraryX-ASIC uses semantics of the library call to generate optimized FFT implementations. This provides increased flexibility in implementation compared to the adapter approach of FACC.

FFT Hardware. The significance of the FFT has naturally led to a vast amount of custom FFT hardware designs in domain-specific SoCs, communications ASICs, and DSP systems. Many hardware implementations [6] are designed and used as standalone accelerators or are later integrated into a larger system. FFT accelerator functionality is typically optimized for the target application and thus has limited support for various FFT sizes or type of transform. There also exists a large body of work on building FFT hardware generators [7] targeting both FPGAs and ASICs.

VI. Conclusion

LibraryX-ASIC is an automated framework for software portability on custom accelerators. LibraryX-ASIC uses the semantics of library calls to recognize computations, generates an optimized implementation using the SPIRAL system, and executes the generated code on the accelerator device. Using an example of an FFT program and an FFT accelerator, LibraryX-ASIC demonstrates significant performance improvements compared to the original software library implementation without software modification. We plan to extend LibraryX-ASIC to support other types of FFTs as well as spectral method operations such as circular convolution.

VII. Acknowledgment

This work is funded in part by DOE ASCR X-Stack Bluestone DE-FOA-0002460.

References

[1] M. Frigo and S. Johnson, "Fftw: an adaptive software architecture for the fft," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, vol. 3, 1998, pp. 1381–1384 vol.3.

[2] 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*, vol. 106, no. 11, pp. 1935–1968, 2018.

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

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

[5] J. Woodruff, J. Armengol-Estapé, S. Ainsworth, and M. F. P. O'Boyle, "Bind the gap: compiling real software to hardware fft accelerators," ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 687–702. [Online]. Available: https://doi.org/10.1145/3519939.3523439

[6] S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, *Handbook of signal processing systems*. Springer, 2013.

[7] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, no. 2, Apr. 2012. [Online]. Available: https://doi.org/10.1145/2159542.2159547