Bind the Gap: Compiling Real Software to Hardware FFT Accelerators

Jackson Woodruff Jordi Armengol-Estapé Sam Ainsworth Michael F.P. O'Boyle
University of Edinburgh
Edinburgh, UK
J.C.Woodruff@sms.ed.ac.uk,jordi.armengol.estape@ed.ac.uk,sam.ainsworth@ed.ac.uk,mob@inf.ed.ac.uk

Abstract

Specialized hardware accelerators continue to be a source of performance improvement. However, such specialization comes at a programming price. The fundamental issue is that of a *mismatch* between the diversity of user code and the functionality of fixed hardware, limiting its wider uptake.

Here we focus on a particular set of accelerators: those for Fast Fourier Transforms. We present FACC (Fourier ACcelerator Compiler), a novel approach to automatically map legacy code to Fourier Transform accelerators. It automatically generates drop-in replacement adapters using Input-Output (IO)-based program synthesis that bridge the gap between user code and accelerators. We apply FACC to unmodified GitHub C programs of varying complexity and compare against two existing approaches. We target FACC to a high-performance library, FFTW, and two hardware accelerators, the NXP PowerQuad and the Analog Devices FFTA, and demonstrate mean speedups of 9x, 17x and 27x respectively.

1 Introduction

Specialized accelerators deliver significant performance improvements [123]. However, specialization is in direct tension with programmability [43]. The more specialized the accelerator, the greater its potential performance [133], but the less likely it is to be used [100].

Fast Fourier Transform (FFT) acceleration is a good example of this. While there are hundreds of commercial accelerator designs [1, 3–6, 22], the API calls used to program them lack the portability and flexibility of software libraries [10, 53] making offloading the domain of experts [107]. Manually migrating to new software APIs is complex and time-consuming [46, 72], and made more challenging by the inability for APIs to hide the complex eccentricities exposed by real hardware [25, 26].

Ideally, we would like hardware to be as specialized and idiosyncratic as needed for performance. We also want existing code to automatically morph to new accelerators with no user involvement [29, 47]. Unfortunately, "*most applications require modifications to achieve high speedup on domain-specific accelerators*" [44]. Here we focus on FFT acceleration as a real-world example of this problem. We demonstrate that automatic modification *is* possible and achieve significant speedups on GitHub legacy C programs[1].

Attempts at replacing application code with accelerator library calls [59] are brittle and do not scale to real-world code or algorithms complex enough to justify acceleration [19]. The fundamental issue is *mismatch*. As the complexity of accelerator functionality increases, the likelihood that it exactly matches a user's application becomes vanishingly small [89, 136].

Mismatch occurs at a variety of levels. The most basic form is *code mismatch* where the number of different ways of writing the same algorithm defeats approaches based on code-shape. Significant mismatch also occurs at a data-representation level, *data mismatch*. Here the code and accelerator may have different types or values — for instance using a custom definition of a complex type. Further still, *domain mismatch* is common, with many accelerators only supporting powers-of-2-sized FFTs [8] or limiting the size of inputs [4]. Finally, there may be *behavioral mismatch*. For example, accelerator output values or user code may be bit reversed or un-normalized. We tackle this fundamental issue in targeting accelerators: the mismatch between user code and accelerator functionality using a novel input-output behavioral scheme using generate-and-test over fuzzing samples to find unique solutions.

1.1 Current Schemes

Programming accelerators typically involves rewriting code in an language or with a new API [131] but this is time-consuming and requires expert knowledge [46, 72]. Recently, work trying to automatically match and replace existing code

[1] All code available at [12]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523439

with accelerator libraries for simple operations has used constraint matching of code to an API description [27, 37, 45, 59]. However these schemes are brittle and fail with minor code variations, and constraints are challenging to write [58]. Exact matching techniques [96, 115] fail once the code scales beyond an order of magnitude of ten instructions, and FFTs scale up to thousands. The near duplicate codes these techniques require is highly unlikely, even when implementations are copy-and-pasted [135].

There is a different stream of research aimed at code-clone detection and algorithm classification [18, 42]. Rather than focusing on code structure via constraint solving, it uses machine-learning-based embeddings of code. Codes with similar behavior will have similar embeddings. These have been successful at *labelling* sections of code [34] and we leverage this as a novel filter to our IO program synthesis. While these code-embedding schemes can identify relevant sections of code, they cannot reason, transform code or compile to accelerators. We evaluate constraint and code embedding approaches in section 8.

1.2 Our Approach

We present FACC (Fourier ACcelerator Compiler), a compiler that maps user code to Fourier transform accelerators. FACC builds a neural classifier [18, 42] to isolate procedures within user code as candidates for potential acceleration. It then explores a space of possible bindings from user variables to accelerator parameters. Next, FACC uses input-output behavioral synthesis to generate accelerator wrappers that bridge the mismatch between user code and accelerator. This allows us to match user code as small as 12 lines of code scaling up to procedures with more than 2000.

We take two accelerators: the Analog Devices FFTA [8] and the NXP PowerQuad [7], and an optimized software library, FFTW [53], and automatically match them to unmodified GitHub code, showing large performance improvement: 27x, 19x and 9x over the original software respectively.

This paper makes the following contributions:

- We introduce four key mismatches that must be overcome for source-code to accelerator compilation.
- We implement a synthesis-baed IO-matching solution to overcome these micsmatches for FFT accelerators.
- We evaluate on real-world code and show significant speedups of automatically compiling to hardware accelerators and optimized libraries.

2 Motivation

Compiling software to specialized hardware accelerators faces the challenge of mismatch between user code and accelerator behavior. Fourier transforms are an excellent example of this problem: they are one of the most widely used transforms in DSP [120], and offer a number of performance/flexibility tradeoffs [126]. This results in a large amount of legacy C code implemented in drastically different styles and optimized for different input sizes. Current-generation hardware accelerators for FFT can out-preform even the most optimized software implementations [11] provided we can bridge the gap between software and hardware.

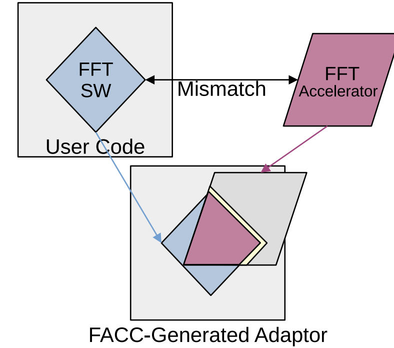

Figure 1. FACC takes user code and an accelerator interface as input and produces an adapter that appears identical to the user code, but uses the accelerator. FACC identifies target regions in the user code that implement FFTs (highlighted in blue), then automatically finds and replaces compatible parts of the FFT with their hardware equivalent where functionality overlaps, while falling back to the software for other operations. Code is synthesised to bridge the gap (in yellow) in implementation and data structures.

Consider Figure 1: there is a section of existing legacy C code that performs a Fourier transform. We would like to cut it out and replace it with a call to an accelerator API. Unfortunately, there is a mismatch between the user code and the accelerator API that prevents this. FACC automatically generates an adapter that acts as a mediator between user code and the accelerator API. User code is now replaced with a call to the adapter enabling acceleration.

2.1 Mismatch Example

Fourier transforms can be implemented in any number of different ways [49]. Figure 2 shows a number of Fourier transforms that are not trivially acceleratable due to mismatches between user codes and accelerators.

The first, left-most column shows a *code mismatch*. The user code is a recursive FFT implementation, but the optimized library provides an iterative implementation. This kind of difference is extremely common in real-world code, but cannot be handled by pattern-matching solutions, which struggle to match copy-pasted code [135]. Indeed, the core FFT in the code we evaluate on ranges from 12 to over 2000 lines, representing a huge diversity in code despite performing the same task.

The second column shows a *data mismatch*. The user code uses a custom complex type different from the complex representation used by the accelerator. To run this code on the

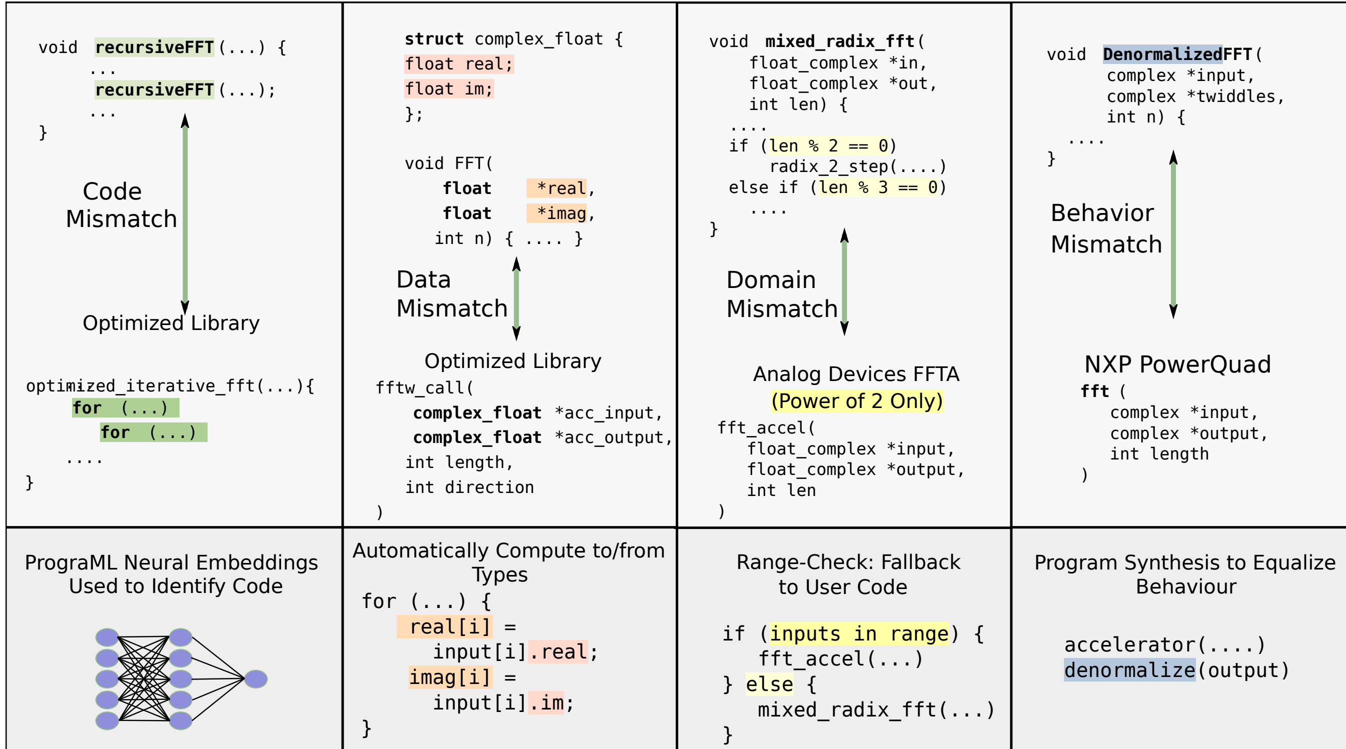

Figure 2. Examples of common mismatches between source code and accelerators, with FACC's resolution below them.

accelerator, the types must be de-constructed and mapped to the inputs to the accelerator.

The third column shows a *domain mismatch*. The user code implements a mixed-radix FFT, which allows for inputs of many different lengths, but the Analog Devices FFTA only supports inputs that are powers of two. As a result, only some inputs to the original user code can be accelerated and dynamic or static checks must be added.

Finally, the right-most column shows a *behavioral mismatch*. The user code implements an FFT but does not normalize the result. To undo the normalization the accelerator does perform, an adapter that denormalizes the output from the accelerator must be used.

Despite these mismatches, this code is acceleratable provided the code, data, domain and behavioral gaps are bridged.

2.2 The General Challenges of Generic FFT Support

There are potentially an unbounded number of differences between functionally equivalent FFTs. Mismatches of code, data domain and behavior can all be handled by FACC's combination of code detection, program synthesis, and generate-and-test IO equivalence.

2.2.1 Code Mismatch.

Different programmers use different strategies for solving the same problem. This results in incidental differences between implementations which defeat constraint-based approaches. FACC achieves independence of coding style by first using code neuro-embedding to find candidate regions in user code. Once it has synthesised candidate adapters for these regions, FACC uses IO examples to test whether the adapter and original candidate code are behavioraly equivalent.

2.2.2 Data Mismatch.

Different implementations of the same algorithm can use different representations of the same data. FACC explores the space of possible mappings between user code and accelerator API variable types via binding synthesis. It uses constraints on data types and variable ranges to reduce the space of possible mappings which are then later evaluated for input-output (IO) behavioral equivalence.

2.2.3 Domain Mismatch.

A valid input to user code may not be a valid input to an accelerator and vice-versa, and this causes complex constraints on functional equivalence between accelerator and code. For example, the Analog Devices FFTA [6] only supports inputs that are powers of two of size greater than 64 and less than 2048 in small mode, and 65536 in large mode. There are two issues to deal with here. The accelerators may not support the full range of inputs that the user code supports. This is a task that requires the generation of a static or dynamic range check. The user code may also not support the full range of *its own* inputs, either throwing errors or resulting in undefined or arbitrary behavior

when fed with unintended random inputs, which can make equivalence testing difficult. FACC uses value profiling [32] and range analysis [64] to address this problem.

2.2.4 Behavioral Mismatch.

Accelerators may not implement the same functionality as user code. To make code match, we either specialize or generalize.

Behavioral specialization is where accelerator input/-configuration parameters are assigned constant values. For example an accelerator may support both FFT and IFFT algorithms, but user code may only implement FFT and so the accelerator should be specialized to match the user code.

Behavioral generalization is where software performs some function that the accelerator does not. For example, the user code may compute un-normalized results, while a hardware accelerator may return normalized results. A software function should be used to generalize the accelerator to produce compatible results.

2.3 Correctness

Implementations of FFTs vary between tens of lines of code and thousands (see section 8.1) and handle arrays of floating-point numbers. Proving traditional correctness is impossible, as different implementations have different error properties [83, 92]. Correctness is further complicated by a lack of formal models available for commercial hardware accelerators, whose designs are often corporate secrets [2]. Even if these issues are overcome, modern floating-point theorem provers are not capable of proving equivalence of such large-scale floating-point algorithms.

Instead of relying on formal proofs of equivalence, we use a pragmatic approach based on fuzzing via input/output examples to determine behavioral equivalence in a number of test cases. Once FACC has confidence that the code can be replaced, it is the developer's role to sign off the source-code replacement via code output in the source language they understand (e.g. figure 3). False positives are very rare without malicious input designed to disguise itself as an FFT. During our evaluation, we encountered no false positives.

```
complex *FFT_accel(complex *x, int N) {
  // Check for valid inputs to accelerator
  if (is_power_of_two(N) && N <= 65536) {
    // Bind user inputs to accelerator
    int len = N;
    #pragma align 64
    complex_float output[len];
    complex_float input[len];
    #pragma end
    for (int i = 0; i < len; i++) {
      input[i].re = x[i].real;
      input[i].im = x[i].imag;
    }
    // Call accelerator
    accel_cfft(input, output, len);
    // Bind accelerator outputs
    for (int j = 0; j < N; j++) {
      x[j].imag = output[j].im;
      x[j].real = output[j].re;
    }
    // De-normalize outputs
    for (int k = 0; k < N; i ++)  {
      x[k].imag *= N;
      x[k].real *= N;
    }
  } else { // Not valid accelerator input
    // Fallback to user code.
    UserFFT(x, N);
  }
}
```

Figure 3. A drop-in replacement for user code generated by FACC. The Analog Devices FFTA used here requires that inputs are 64-byte aligned, and is out-of-place, while the user's code is in-place. Pre-binding is highlighted in gray, post-binding in pink, post-behavior in green and range in orange — pre-behavior is empty in this case.

3 System Overview

FACC uses Input-Output (IO)-based program synthesis to generate an adapter that is a drop-in replacement for the original user code, matching the output behavior for all inputs even though the implementation is different. Given some accelerator performing a function A and some user code performing a function U, FACC finds adapter functions g, h such that $\forall x. U(x) = g(A(h(x)))$, where x represents test input. Crucially, this test for IO equivalence is invariant of the exact structure of the code, which in FFTs can vary from tens to thousands of lines, and so can match any code which given the same inputs produces the same outputs. Adapters are created via a generate-and-test approach, by generating many plausible candidates, filtered first using known constraints and heuristics, before all but one option is eliminated using fuzzing. Finally, the synthesised adapter is presented to the user for verification.

3.1 A Generic Framework for Accelerator Support

Our key insight is that to support an accelerator performing function A, and use it to accelerate diverse user code U,

we must patch the difference using functions range (r), pre-binding (b), post-binding (b'), pre-behavioral (s) and post-behavioral (s') such that

$$U = r(s \circ b \circ A \circ b' \circ s')$$

where each function provides the following behavior:

- b, b' address the *data mismatch* problem by mapping between accelerator variables and user-code variables. b takes user-code inputs and produces conversions to accelerator inputs, while b' takes the accelerator outputs and converts them to user outputs (Section 5.1).
- r addresses the *domain mismatch* problem with *input range checking* to determine whether the inputs presented can be run on the accelerator. FACC does this with a mix of static and dynamic analysis, generating the minimal possible check with the static information available (Section 5.2).
- s, s' address the *behavioral mismatch* problem by adding or undoing accelerator functionality to match the user code. FACC sets s to the identity function, as many pre-behavioral FFT problems have a post-behavioral equivalent s' which can be used instead (Section 5.3).

An example output is shown in figure 3. In order to match the accelerator's data format (gray) the adapter converts the user code's input to a different datatype — aligned and changed to be out-of-place. After accelerator execution, the adapter restores the in-place representation (pink). The normalization performed by the accelerator but not the user code is undone (green). If the accelerator's constraints on size and being a power of two aren't met, the user code is run instead (orange).

Generic and Domain-Specific Components The framework described above is domain-agnostic. However, to make the synthesis problem tractable, some parts are domain-specific. In particular, our solution to behavior mismatch relies on sketch-based synthesis and is domain-specific to FFTs. We expect our sketches to be easily extendable to new domains. Our solutions to the data mismatch and domain mismatch problems are general and applicable to many types of accelerator.

3.2 Operation

FACC uses synthesis to generate an adapter that enables drop-in accelerator use. Multiple candidates are generated and tested against the user code to pick the correct one. Figure 4 shows the stages of the tool:

1. An API to compile to and limitations of the hardware are provided as input.
2. **Candidate detection** discovers potential targets using neural classification [42], and analyzes user code using static analysis to aid in generating a match (Section 4.3).

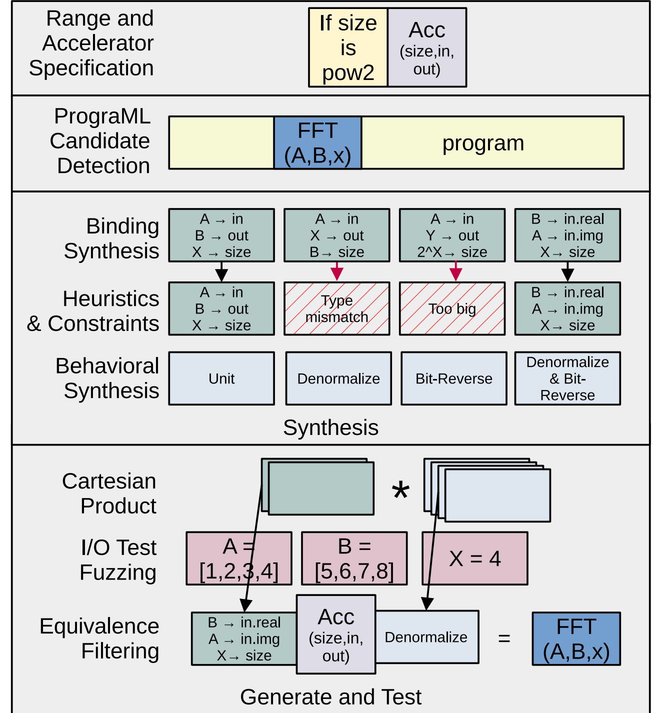

Figure 4. FACC takes a specification of an accelerator, and produces an equivalent version of the original program with acceleration. It uses neural embeddings to find plausible candidates for replacement, then creates a set of possible input and output bindings, filtered by constraints and heuristics. It then tries to patch the functionality of the accelerator to match that of the user code via behavioral synthesis. Finally, FACC generates all possible combinations of these mappings, and tests them for IO equivalence with the user code.

3. **Synthesis** generates candidates for the r, s, s', b, b' functions, discarding those made invalid via constraints and heuristics (Section 5).
4. **Generate and Test** filters the combination of all possible matches using IO tests to generate a drop-in replacement (Section 6).

4 Identifying Acceleratable Candidates

FACC bridges the gap between user code and accelerator behavior by generating adapters. Before it can do that, it employs an existing tool [42] to identify candidate acceleratable code regions. FACC then gathers information on how variables within code regions are used to drive adapter synthesis.

4.1 Identifying Acceleratable Regions

FACC is a binding tool, using a neural classifier based on ProGraML [42], to detect likely acceleratable FFT-based code.

Data We use the OJClone algorithm classification dataset introduced in Mou et al. [95] consisting of 105 classes, each composed of different implementations of the same task. We add FFT as an additional class, with the same FFT code snippets obtained from Github used in the rest of the article. We restrict all classes to 20 instances for a balanced dataset. Each instance is parsed and transformed into a data flow graph of its LLVM instructions with ProGraML [42]. Due to the reduced size of the dataset, we implement 10-fold cross validation, such that each train split contains 80% of the dataset and the remaining 20% is left as holdout.

Model We implement a Graph Convolutional Neural Network with two graph convolutional layers followed by max-pooling and a linear layer to perform the actual classification, using PyTorch [102] and DGL [130]. We do not perform any hyperparameter search (instead, set reasonable default values), and use the Adam optimizer [75] with weight decay as regularization. All models are trained for a maximum of 100 epochs using early stopping with a patience of 10, which led to convergence in all experiments.

Identifying Invalid Regions No code detection tool is perfect, and so ProGraML may misclassify algorithms. FACC evaluates all of these as potential generate-and-test targets, and if an invalid region (i.e. one not matching the accelerator interface) is identified, FACC will fail to generate valid bindings and leave the spuriously identified region unchanged. In this sense, the neural classifier is used to cut down the search space: rather than considering all instruction sequences of all programs as possible targets, it only tries to match those labelled by the neural classifier[2].

Code Mismatch Identifying code regions is only the first part of overcoming code mismatch. The second is that code itself is highly diverse; our evaluation set ranges from 12 lines to over 2000 for similar behavior. In section 6, input-output (IO) testing is used to test whether the adapter synthesized in section 5 matches the behavior of the identified code. IO-testing allows us to ignore the underlying code structure eliminating code mismatch by focusing only on the interface.

4.2 Identifying Input/Output Variables

FACC relies on existing liveness analysis to determine which variables are output variables and which are input variables. This allows for extraction of functions with side-effects, or extraction of sub-function regions of code. We use variable range analyses [64, 79, 88] points-to analyses [21, 63] and value-profiling [32] to reduce compilation time.

4.3 Type Inference

FACC expands types in two ways: by inferring the lengths of arrays, and by inferring more structured types over base types where they may be required by the accelerator.

This step takes a single type from the user code as input, and produces a number of plausible extended types to use for the remainder of the synthesis as an output. A pseudo-code type augmentation algorithm is shown in algorithm 1.

Algorithm 1 Type Augmentation Algorithm. Takes a type as input, and produces all plausible types that can replace it.

```
procedure AugmentTypes(Type_in)
    Types = ∅
    if IsArray(Type_in) then                    ▷ Infer Lengths
        for len ∈ Possible Length Variables do
            Add Type_in#len to Types
        end for
    else
        Add Type_in to Types          ▷ Not Array so No Length
    end if
    for T ∈ All Possible Types do              ▷ Infer Structure
        for T′ ∈ Types do
            if IsCompatible(T, T′) then
                Add T to Types
            end if
        end for
    end for
    return Types
end procedure
```

Length Inference Arguments passed as arrays to functions often have a variable number of values. For example, a type signature that takes a single integer as argument can only take a single input, but a function that takes an array can take `N` inputs, where `N` is the length of the array. In languages like C, array lengths are implicit, not directly specified by the programmer. Although best-effort compiler passes can assist with providing this information [87], FACC is able to infer array lengths using a generate-and-test approach. Each array is assigned a number of possible length parameters, and the correct one is determined during testing.

Structure Inference API designers are often encouraged to present APIs with the most syntactic information possible [66]. The user code faces no such restrictions. As a result, FACC needs to infer more syntactic information over base types. All plausible (dependent on the types in the accelerator API) inferred types are considered, and filtered via generate-and-test (see algorithm 1).

5 Synthesis

Here we describe the core accelerator support problem. We address three key mismatches: data mismatch using binding synthesis, domain mismatch using range-check generation, and behavioral mismatch using behavioral synthesis.

[2]Code is available at [13]

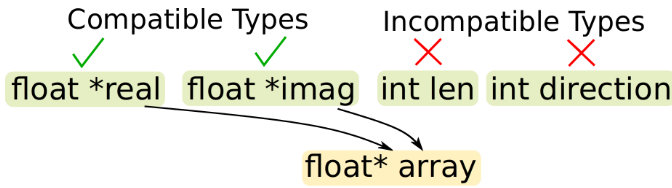

Figure 5. Type constraints reject two impossible bindings.

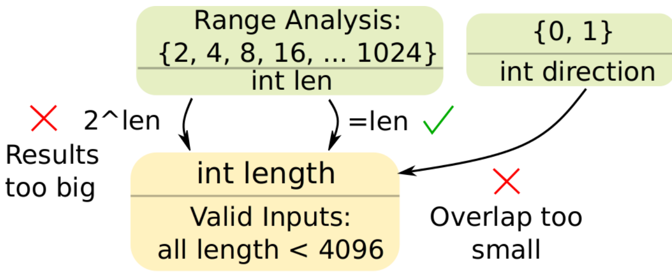

Figure 6. A non-trivial conversion (2^n) is considered, but ruled out due to range heuristics.

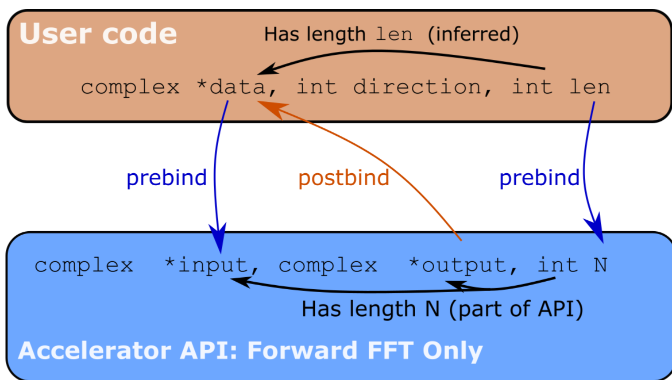

Figure 7. An example solution to the binding problem. To determine the correct binding, FACC tries all plausible bindings that cannot be statically determined impossible.

5.1 Data Mismatch: Binding Synthesis

In binding synthesis, we take a set of input variables and a set of output variables from the user code. We generate every mapping that Type Inference (section 4.3) does not allow us to eliminate either via constraint or heuristic, between these variables and the accelerator API variables, to be evaluated using generate-and-test. Figures 5 and 6 show an example creating possible bindings for a single variable while rejecting those statically known to be impossible. Figure 7 shows a full candidate mapping.

5.1.1 Non-trivial Conversions.

The vast majority of accelerator parameters can be copied directly from parameters existing in user code. However, frequently, the same information is encoded in indirectly compatible ways. A typical example is using N to directly encode array length, compared to using 2^N to represent array length. Another typical example is the many different ways that a flag can be represented in C: 0 and 1, -1 and 1, 1 and 0, etc. FACC generates conversions allowing compatibility between implementations with different flag values. Variable-range information is used to vastly reduce the search-space of conversions.

5.1.2 Constraints.

FACC applies constraints to generated bindings, limiting the search of impossible matches.

Type Conversions If a variable x is to be assigned to some variable y, then there must be a known conversion between the two types, including over distinct representations of complex numbers.

Array Assignments If any two array variables share a length variable, then the arrays that they are assigned to must also share a length variable — and those two length variables must be assigned to each other.

5.1.3 Heuristics.

FACC also applies a number of heuristics to the bindings generated.

Range Heuristics are applied to determine whether the accelerator is likely to be useful. For example, if a variable x may take any one of 100 values, and is assigned to an accelerator API variable y, which only supports one value, the odds of successful acceleration are extremely small, so the binding is not considered likely (figure 6).

Single-Read Heuristics FACC assumes that user-code variables should only be read from once when assigning to accelerator variables. This heuristic greatly reduces the synthesis space by assuming a lack of unneeded redundancy in the programmer's original code.

5.2 Domain Mismatch: Range-Check Generation

Fixed function accelerators are often extremely specialized — significant performance is possible by making the common case fast. However, legacy C code is more general in scope.

It is important that offloaded code only operates within the valid range of the accelerator. To ensure this, we synthesize range checks, which offload to the accelerator if the inputs are valid, and fall back to the user code otherwise.

We use two sets to determine the overlap region of an accelerator and user source code.

Accelerator Specification The accelerator API is expected to specify what set of inputs the API functions on. These inputs are used both to direct testing of compatibility, and to generate input-range checks.

User-Code Analysis Inter-procedural range analysis complements the accelerator specification by allowing FACC to reduce the quantity of input checking to the intersection of the accelerator's range and the user code's range, rather than all possible FFT inputs.

5.3 Behavior Mismatch: Behavioral Synthesis

Behavioral synthesis introduces adapters that make accelerators transparently compatible with more user code. For example, suppose we have a user FFT function that does not normalize the results, even though the FFT accelerator available does. We use post-behavioral synthesis to generate de-normalizing code and enable accelerator use while allowing the programmer to use de-normalized results.

We implement domain-specific post-behavioral synthesis program using sketch-based synthesis [116]. For FFT functions, there are a small number of behaviors that are often omitted: normalization/denormalization and bit-reversal.

We provide a number of sketches with holes, and a procedure to fill the holes and produce all options. No infinite sketches are allowed — all sketches must be finite once holes are filled, and there must be a finite number of ways to fill each hole. Generated candidates are tested against user code.

6 Generate and Test

FACC is an Input-Ouput (IO)-based synthesis tool. The candidate adapters generated by synthesis are compared to the original code using fuzzing to determine equivalence. The working adapter is output in the original source language (figure 3) and used as a drop-in replacement in the user's code.

6.1 Random-Input Generation

Tests are randomly generated with a bias towards smaller examples that run more quickly. Examples are constrained to be within the computed range analyses of user code, and the valid-input range of the accelerator. As discussed in Section 4.3, variable-length arrays have inferred length variables and so order of generation is important. We use a topology sort to ensure that that variables are assigned in a valid order. In addition to IO-equivalence, AddressSanitizer [114] is used to detect arrays with incorrect length parameters assigned by detecting out-of-bounds accesses.

6.2 Potential for Bounded Model Checking

Bounded model checking is an approach where a theorem-proving tool shows that a program cannot enter a specified error state, or provides a counter example. Given that the accelerators we support have bounded input sizes and for other sizes we call the original code, bounded model checking is sufficient. However, FFT algorithms are reliant on floating-point analyses and fall into a significantly harder category of model checking. The input to a floating-point model checker can be phrased as:

```
float *u = user_fft(...);
float *a = accel_fft(...);
float e = error(u, a);
assert (e < threshold);
```

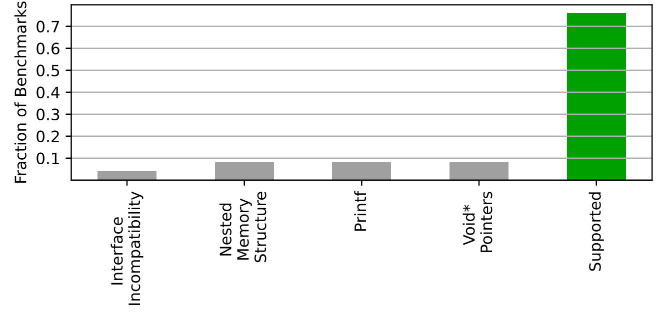

Figure 8. FACC success and failure classification.

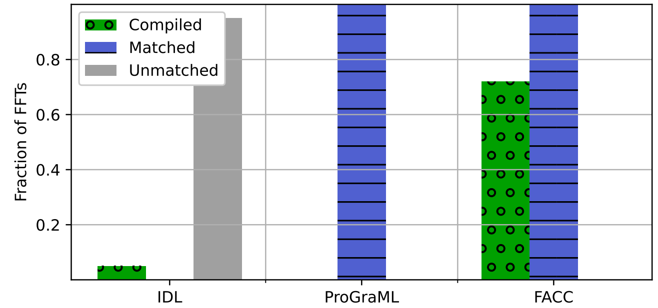

Figure 9. Performance of different strategies: constraint matching, neural embeddings and FACC.

Despite the portability of IEEE 754 floating point [9], it is designed for small-step operations, rather than full algorithms such as FFTs. Floating-point tools such as XSat [54] or Klee [31] can accept bounded model checking problems that could theoretically prove equivalence between functions within accuracy bounds. Existing techniques fall far short from being computationally efficient enough to prove the correctness of complex floating-point functions. FACC requires programmer sign-off due to imprecision of hardware and software implementations, as well as the IO testing mechanism.

7 Setup

We search GitHub for "FFT", and restrict the results to C. Of the first 100 results, we have identified 24 distinct complex floating-point FFT implementations after excluding buggy code[3], code with missing dependencies, clones and implementations in different languages. We have added the FFT in MiBench [61]. We have placed these 25 implementations into a benchmark suite, and used FACC to compile from each. Where required, we have constructed a value-profiling environment, to enable FACC to compile the benchmark to the accelerator.

[3]Buggy should be interpreted as "the authors were unable to make the code produce correct results to the Fourier transformation."

Project	Lines of Code	Lengths Supported	Algorithm	Twiddle Factors	Imaginary Numbers	Pointer Arithmetic	Loop Structure	Optimizations
0	83	Only 64	Radix-2 FFT	Constant	Custom	No	While-True-Break	Minimal
1	278	Powers of 2 (≤ 256)	Radix-2 FFT	Constant	Custom	No	Do-While/For	Minimal
2	65	Powers of 2	Radix-2 FFT	Computed in FFT	Custom	No	For/Recursive	Minimal
3	107	Powers of 2	Radix-2 FFT	Computed in FFT	Custom	No	For	Minimal
4	934	All	Mixed-Radix FFT	Computed in FFT	Custom	No	For/Recursive	Extensive Unrolling
5	2159	All	Mixed-Radix FFT	Pre-Computed	Custom	Yes	For	Hand-Vectorized/Unrolled
6	77	Powers of 2	Radix-2 FFT	Computed in FFT	Custom	No	For	Minimal
7	237	Powers of 2	Radix-2 FFT	Pre-Computed	Custom	Yes	For	Minimal
8	101	Powers of 2	Radix-2 FFT (DIF)	Computed in FFT	C99 Complex	No	For	Minimal
9	1627	All	Mixed-Radix FFT	Pre-Computed	Custom	Yes	For/While/Recursive	Extensive Unrolling
10	75	Powers of 2	Radix-2 FFT	Pre-Computed	Custom	No	For	Minimal
11	538	All	Mixed-Radix FFT	Pre-Computed	Custom	Yes	Do-While/For	Twiddle-Factor Memoization
12	367	All	Mixed-Radix + Bluestein	Computed in FFT	Custom	No	For/Recursive	Unrolling
13	101	Powers of 2	Radix-2 FFT (DIT)	Computed in FFT	C99 Complex	No	For	Minimal
14	314	Powers of 2	Radix-2 FFT	Computed in FFT	None	No	For	Minimal
15	215	All	Recursive FFT	Computed in FFT	C99 Complex	No	Recursive	Minimal
16	20	All	DFT	Unneeded	C99 Complex	No	For	None
17	12	All	DFT	Unneeded	C99 Complex	No	For	None

Table 1. Features of each benchmark used, representative of a wide range of implementation styles, from highly-optimized several-thousand line implementations to short, simple Discrete Fourier Transforms (DFTs).

Implementation FACC is implemented using OCaml, with behavioral synthesis libraries implemented in C. FACC currently has a C backend which is compatible with toolchains for the various backend targets. In total our implementation is 13,000 lines of OCaml, with 1,000 lines for range check generation, 1,000 lines for behavioural synthesis, 3,000 lines for binding, and 4,000 lines for backend-specific generation and the remaining 4,000 used for various utilities. All compiler and benchmark code is available at [12].

Experimental Setup Codes were placed in a benchmark suite that tests them on inputs that could be accelerated by the accelerator in question. We evaluate on three platforms:

FFTW: A desktop environment running Windows Subsystem for Linux and using an Intel i9-10900X processor and the FFTW optimized library. Code is available at [12].

ADSP board (SC589/FFTA): A multicore embedded environment using the Analog Devices ADSP-SC589 Development board with an Arm Cortex A5 as a primary core, an *SC589* SHARC DSP core and an *FFTA* Fourier transform hardware accelerator. Code is available at [14].

NXP Board (Powerquad): A single core embedded environment using the NXP LPC55S69 Development board with an Arm M33 as a primary core and an NXP *PowerQuad* accelerator capable of accelerating Fourier transforms. Code is available at [15].

Competitive Approaches We evaluate IDL [59], an existing constraint based approach to identifying code sections for acceleration. We evaluate our ProGraML-based classifier's [42] speedup by offloading FFTs to an SC589 DSP core. FFTs can be offloaded to the SC589 DSP core simply by identifying them, but the semantic information required to offload to the FFTA is not inferred. Rather, we use ProGraML as a hint that the code is likely to perform better on the DSP than the CPU.

8 Results

We evaluate FACC along several dimensions, comparing against success rates of IDL and ProGraML (section 8.2), performance of IDL and ProGraML (section 8.3), performance across multiple platforms (section 8.4) and properties of the compilation (section 8.5).

8.1 Which Benchmarks does FACC Support?

FACC compiles 18 of the 25 implementations as shown in Figure 8. Table 1 shows a summary of the code features used in the projects FACC is able to compile. We can see that implementations vary both at the level of functionality they support, with different implementations supporting different lengths of input, and in the way they implement the Fourier transform. Approaches vary between 12 and 2,159 lines of code, using iterative and recursive approaches. A number of implementations unroll loops and base cases by hand to achieve better performance, while others introduce memoization between calls and others still use hand-vectorized instructions. It is very common to use custom-defined complex types, rather than the standard C99 type.

Figure 8 shows why FACC cannot compile some cases. Printfs during execution results in observably different behavior than can be supported on an accelerator that does not print to stdout. Void* pointers and Integer FFTs both require more implementation work to support the appropriate type conversions required. Support for nested memory structures requires implementation of support for nested calls to malloc. The features to support these are work in progress.

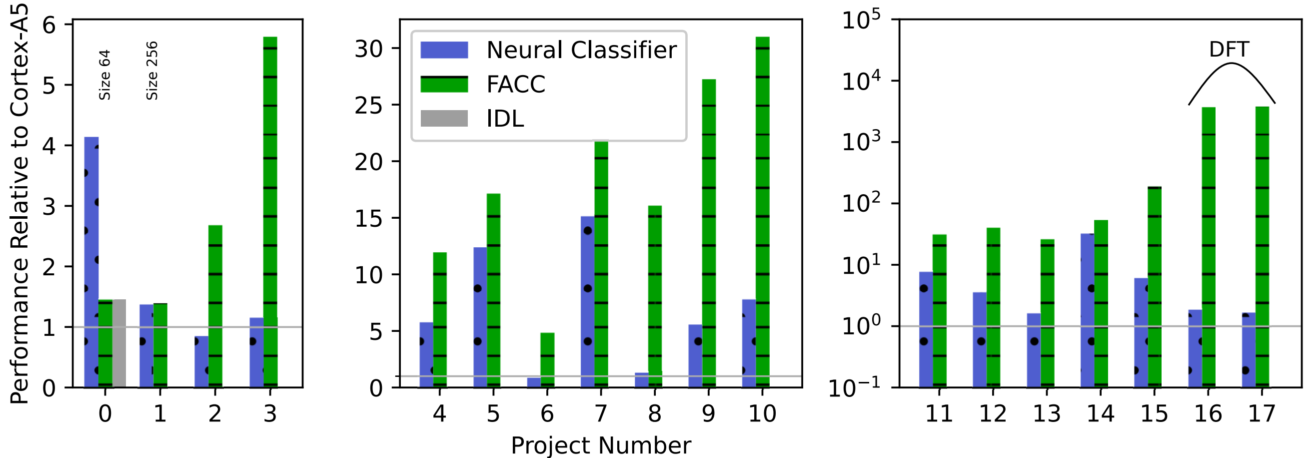

Figure 10. Comparing offloading techniques between on the Analog Devices ADSP-SC589 Development board. Inputs of size 1024 are used unless otherwise noted. An Arm Cortex-A5 is the master core, and can offload either to the SC-589 DSP or to the FFTA accelerator. A neural embedding is used to offload to the DSP core and achieves geometric mean speedup of 3.5x. FACC offloads to the FFTA, and achieves corresponding speedup of 27x.

8.2 Which Benchmarks do IDL and ProGraML Support?

Figure 9 shows the performance of three different compilation techniques on our benchmark suite.

IDL For IDL [59], we design a pattern for project 0 (in Table 1). We can see that IDL can compile the single benchmark we hand-crafted a pattern for, but cannot generalize. Figure 12 shows why: from our workload set, no pattern becomes similar enough to any other past 50 lines, and most diverge much more quickly. While simple function prologue snippets are sometimes similar enough to allow us to match them between functions for a few lines, the level of code mismatch in the core FFT algorithm makes this strategy ineffective. Even if we charitably try to match the two simplest codes, 16 and 17 at 20 and 12 lines respectively, we immediately fail; they use different library functions for complex arithmetic.

ProGraML By contrast, the modified ProGraML [42] classifier is effective at detecting FFTs: Figure 11 shows a cross validation. Top-1 refers to classifying a code region solely by the highest predicted probability class. Top-3 refers to considering those 3 classes with the highest probability. The FFT top-3 recall reaches 100% with as few as 11 examples. Using top-3, we also find precision converges rapidly to 1. This means FACC will try binding on all code regions labelled FFT by Top-3, discarding those where there is no legal binding, to avoid false-positive code outputs — it is better to have a classifier that identifies too many regions than too few regions.

Although we use a top-3 scheme, a top-1 scheme for FFTs provides a different performance point with an F1 score of 0.8. Such classification schemes can be tuned to obtain suitable performance/coverage characteristic for the compute power available.

We also show the overall performance for predicting all classes - not just FFT. We observe that with around 8 examples per class, top-3 accuracy is consistently above 50%. Overall, the model does not overfit to the train split, and reaches useful performance with relatively few examples. This is due to the effectiveness of the ProGraML representation, the convolutional graph inductive bias, and the class separability of the dataset, especially in the case of FFT, whose data-flow graph shows clearly distinguishable patterns. Generally, we can see that neural embeddings are effective at detecting FFTs, and also have applicability for similar acceleration-identification tasks in other domains.

8.3 How Do FACC's Adapters Preform?

Figure 10 shows the performance of FACC on the ADSP board compared with the ProGraML Neural Classifier and IDL. FACC takes advantage of the algorithm-specific accelerator, achieving geometric mean speedups of 27x. ProGraML cannot exploit the accelerator since it is just a classifier, but achieves a 3.5x speedup by moving FFT code to the DSP. Interestingly, in one case the DSP is actually faster than the FFTA due to the small data size. IDL only detects one acceleration opportunity, achieving the same performance as FACC on benchmark 0 only.

8.4 How Do FACC's Adapters Perform on Different Platforms?

Figure 13 shows the performance improvement obtained by each implementation on the ADSP board, the NXP board and

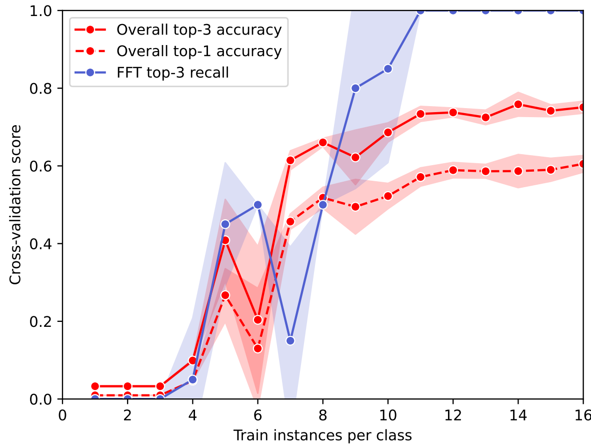

Figure 11. Cross-validation accuracy (mean and standard deviation) of our ProGraML-based neural classifier in terms of the number examples per class when trained using a reduced version of the OJClone dataset with FFT examples injected.

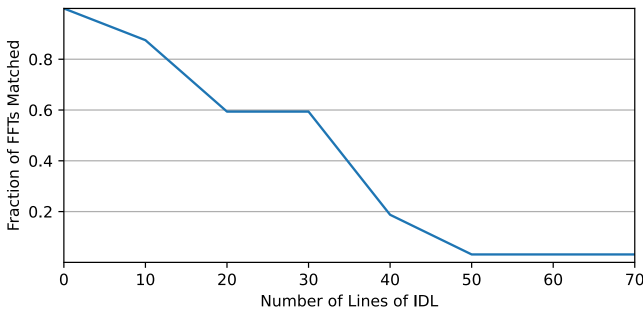

Figure 12. How the number of patterns matched changes with the length of the IDL pattern used. IDL patterns to match entire FFTs are thousands of lines long and do not generalize. By 50 lines we have only a single remaining match and still only cover the prologue of a single FFT function.

the FFTW optimized software library. Relative performance is dictated by the performance of the accelerator and the performance of the compiler used to compile the original implementation.

These differences can be seen on benchmark 8, where the original implementation is poorly optimized for the hardware on the Arm Cortex-M33, but runs much better on the Arm Cortex-A5 and Intel i9-10900X. We can also see significant differences between styles of implementations, with projects 16 and 17, which are DFTs yielding particularly large speedups (10,000x on the PowerQuad). The geometric mean speedup for each accelerator relative to their baseline is 9x for FFTW, 17x for the PowerQuad and 27x for the FFTA.

Performance for varying sizes of input for projects 1–7 is shown in figure 14. Speedups increase with data size as expected for an offloading-based accelerator model [19]. Speedups are possible using optimized software libraries, although the opportunities are more limited and may require profiling to determine viability.

8.5 Compilation Time

Figure 15 shows the compilation time taken by FACC for each benchmark. Results are gathered on a 6 core Intel i7-8700K CPU running at 3.70 GHz with 32 GB. We anticipate a number of simple parallelism-based optimizations could significantly reduce compilation time.

Figure 16 shows how the number of binding examples generated for each target. FFTW exposes more functionality in it's interface, so requires more examples to be generated. FACC uses the same interface to access the ADSP board's FFTA and the NXP board's PowerQuad, so the number of examples is identical. The difference in compile time is due to different supported input lengths: the PowerQuad supports smaller input sizes, which are faster to test. None of these programs result in excessively large search spaces. If the search space were to grow, standard synthesis pruning techniques could be applied [28].

9 Related Work

9.1 Algorithm Identification

A number of techniques have been developed that enable algorithm identification within extensive user codebases. Vector-embedding techniques such as code2vec [18] can be used to identify and label algorithms in code. There are numerous techniques that use larger, code-clone specific datasets to achieve quantifiable results. Embeddings such as ProGraML [42] achieve upwards of 95% accuracy in clone detection, and a number of other machine-learning approaches using static information exist [30, 51, 55, 95, 101, 132, 134, 139]. Dynamic runtime information can also be used for this task [129] and numerous approaches developed without machine learning exist [69, 70, 74, 110]. API-recommendation tools [65, 68] can also be used for algorithm identification. Finally, NLP has been applied to code comments to identify the algorithm [76]. Algorithmic mismatch has been explored on a number of dimensions in relation to AI accelerators [128], and with relation to different FFT API calls [122].

9.2 Existing Compilation Techniques

DSP optimizations [52, 73] can aid FFT performance, but do not come close to accelerator performance [11]. DSL approaches get closer [108, 109, 118, 127], and work to extract such DSLs from source code has been developed [16, 20, 71, 91], but these approaches rely on programmable small-step accelerators and do not generalize to big-step accelerators such as the FFT accelerators we explore.

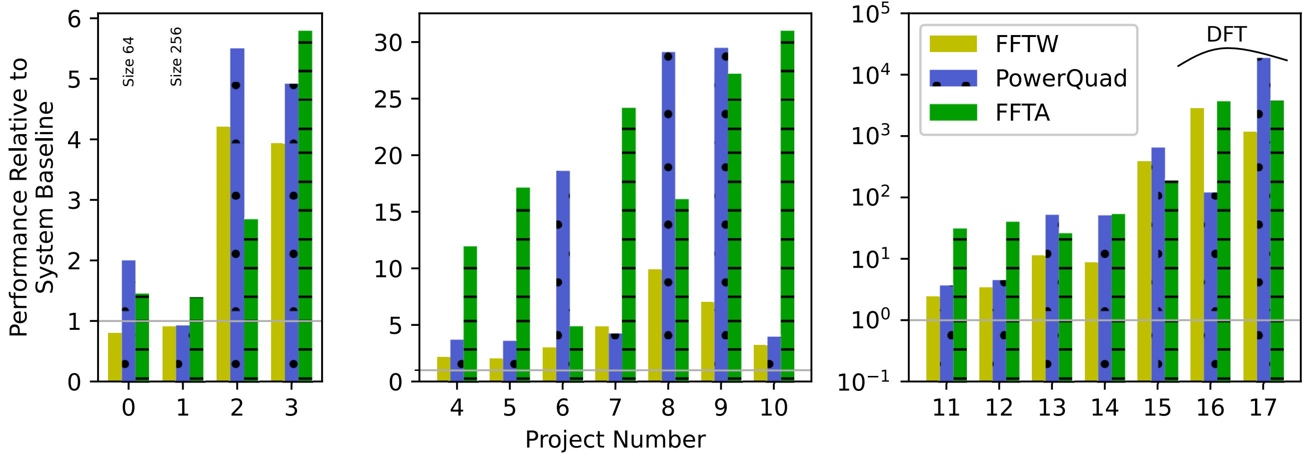

Figure 13. Relative performance for different FFT implementations on GitHub, comparing original software and FACC's generated accelerator call for FFTs of length 1024. FFTA results from the ADSP board are compared to software running on the Arm Cortex-A5. PowerQuad results from the NXP board are compared to software on an Arm Cortex-M33. FFTW results are compared to software on an i9-10900X desktop CPU. Geometric mean speedup is 9x for FFTW, 17x for the PowerQuad and 27x for the FFTA.

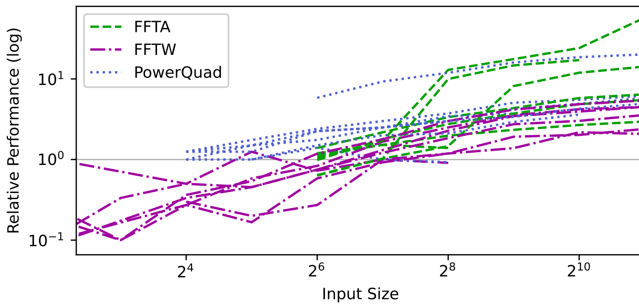

Figure 14. Speedup for accelerating benchmarks 1–7 on different sizes of input. Different accelerators and benchmarks have different overlap ranges, but in general, as problem size increases relative speedup increases.

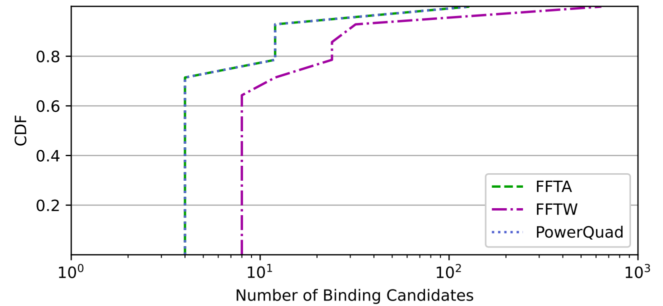

Figure 16. CDF of the number of candidates generated by FACC for each benchmark. One distribution per target. FFTA and PowerQuad overlap due to similarity of restrictions exposed via API from the hardware, unlike the software FFTW.

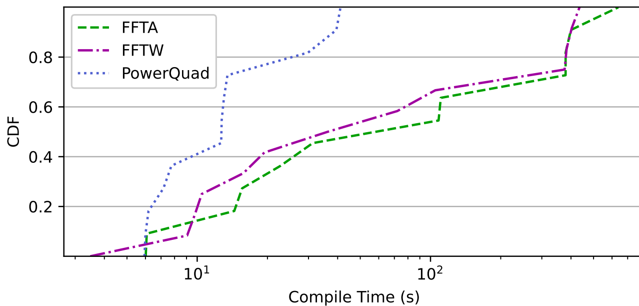

Figure 15. CDF of the compilation times taken by FACC for each benchmark. One distribution per target.

Constraint matching [27, 45, 57] provides a way of matching and extracting interfaces from high-level code. Unfortunately, these approaches are brittle [45] — they do not scale beyond a single implementation/accelerator pair, and constraint patterns are extremely hard to write [39, 40, 58]. Rewrite-rule based compilers can be used to target accelerators [77], but these still rely on initial matching using constraints or similar. For affine algorithms, approaches using polyhedral analysis have also been attempted [24, 82, 119, 121] — but these are inapplicable to non-affine or highly-optimized implementations. Other authors focus on ensuring that the presented API retains easy programmability [86], aiming to help programmers program accelerators directly.

A large amount of work has been done on API migration [38, 97–99, 105],the task of migrating code using one API

to use a new API. Likewise, a number of API-recommendation tools [68, 137] have been developed, although these do not tell the programmer how to integrate the API. Another common approach is a backend-independent API [93, 122] allowing for migrations to happen under the hood. Samak et al. [111, 112] approach a similar problem in the object-oriented space, using embeddings and synthesis to generate adapter classes for drop-in replacement classes in Java. The tools, Mask and ClassFinder, use symbolic execution to prove equivalence, a technique which does not scale to FFT-sized algorithms. Work applying program synthesis to take advantage of it's syntax-independence has been applied for software optimization [41, 106, 113].

9.3 FFT Accelerators

Hundreds of research implementations [22, 56] and commercial implementations [1, 3–6, 22] of FFT accelerators exist intended both as stand-alone accelerators, and to be integrated in larger accelerators [125]. Work on supporting FFT acceleration exists for FPGAs [94], GPUs [84] and specialized architectures from linear algebra cores [104] to CGRAs [67, 85], machine-learning accelerators [50, 80, 117], optical computers [78] and sonic computers [103].

FFT accelerator performance is largely memory-bandwidth limited [48, 124], a problem exacerbated by access patterns that make poor use of DRAM buffers [17, 23]. Much work has been focused on reducing memory demands. Computing twiddle factors on-chip has been explored [33, 35, 62] and applied in industry [90]. In-memory FFT accelerators have also been proposed to reduce this communication overhead [36, 81, 138] along with 3D-stacked memory accelerators [60].

10 Conclusion

This paper describes FACC, a tool for compiling user-code to Fourier-transform accelerators and optimized libraries. FACC uses IO matching and program synthesis to address the problems of code, data, domain and behavioral mismatch, allowing for easy accelerator integration into existing source code. Using FACC and real-world optimized libraries and hardware accelerators, we are able to achieve speedups averaging 9x for FFTW, 17x for the PowerQuad and 27x for the Analog Devices FFTA.

While FACC focusses on matching user code to acceleration APIs, it can also be used to match optimized libraries to emerging hardware e.g matching FFTW to FFTA. This would allow users, who have already restructured their application to use libraries, to continue to benefit from hardware evolution, while automatically handling the unusual constraints that fixed-function hardware poses.

Although, this paper focusses on Fourier-transforms, this approach is readily applicable to other fixed-function accelerators. Fixed-function need not be the enemy of programmability and automatic targeting. Rather, we can automatically rearchitect software to build adapters that bend the accelerator to the user's will rather than vice versa.

Acknowledgements The material is partially based on research by Michael O'Boyle which was sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7864. The views and conclusions contained herein are those of the authors and do not represent the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] [n.d.]. B4860 QorIQ Qonverge Multi-Accelerator Platform Engine Baseband 4 (MAPLE-B3) Reference Manual. ([n. d.]). Available at https://www.nxp.com/files-static/training_pdf/vFTF09_AN149.pdf.

[2] 2005. Defense Science Board Task Force on High Performance Microchip Supply. (2005).

[3] 2010. *ADSP-214xx SHARC Processor Hardware Reference*. techreport 82-000469-01. Analog Devices. Available at https://www.analog.com/media/en/dsp-documentation/processor-manuals/ADSP-214xx_HRM_rev0.3.pdf.

[4] 2011. *KeyStone Architecture Fast Fourier Transform Coprocessor (FFTC)*. techreport SPRUGS2C. Texas Instruments. Available at https://www.ti.com/lit/ug/sprugs2c/sprugs2c.pdf.

[5] 2015. *Keystone II Architecture Fast Fourier Transform Coprocessor (FFTC)*. techreport SPRUHE0A. Texas Instruments. Available at https://www.ti.com/lit/ug/spruhe0a/spruhe0a.pdf.

[6] 2018. Analog Devices SHARC+ Dual-Core DSP with Arm Cortex-A5: ADSP-SC582/SC583/SC584/SC589/ADSP21583/21584/21587. (2018). Available at https://www.analog.com/media/en/technical-documentation/data-sheets/ADSP-SC582_583_584_587_589_ADSP-21583_584_587.pdf.

[7] 2019. AN12282: Digital Signal Processing for NXP LPC5500 Using PowerQuad. AN12282 (January 2019). Available at https://www.nxp.com/docs/en/application-note/AN12282.pdf.

[8] 2019. CrossCore Embedded Studio 2.9.0: C/C++ Library Manual for SHARC Processors. 82-100118-01 (2019). Avalable at https://www.analog.com/media/en/dsp-documentation/software-manuals/cces-sharclibrary-manual.pdf.

[9] 2019. *IEEE Standard for Floating-Point Arithmetic*. techreport 754-2019. Microprocessor Standards Committee.

[10] 2020. *Intel oneAPI Math Kernel Library — Data Parallel C++ Developer Reference*. techreport. Intel. Available at https://docs.oneapi.com/versions/latest/onemkl/index.html.

[11] 2021. *ADSP-SC58x FFTA Benchmarks*. techreport. Analog Devices. Available at https://ez.analog.com/dsp/sharc-processors/w/documents/5017/adsp-sc58x-ffta-benchmarks.

[12] 2021. FACC Souce Code. Available at https://github.com/FourierACceleratorCompiler/FACC.

[13] 2021. FFT Classification Environment. Available at https://github.com/FourierACceleratorCompiler/FFTClassification.

[14] 2021. FFTA Evaluation Environment. Available at https://github.com/FourierACceleratorCompiler/FFTAEnvironment.

[15] 2021. NXP PowerQuad Evaluatoin Environment. Available at https://github.com/FourierACceleratorCompiler/NXPEnvironment.

[16] M B S Ahmad, J Ragan-Kelley, A Cheung, and S Kamil. 2019. Automatically translating image processing libraries to halide. *ACM Transactions on Graphics* 38 (11 2019), 1–13. Issue 6. https://doi.org/

10.1145/3355089.3356549

[17] B Akin, Franz F, and J C. Hoe. 2014. Understanding the design space of DRAM-optimized hardware FFT accelerators. *ASAP*. https://doi.org/10.1109/asap.2014.6868669

[18] U Alon, M Zilberstein, O Levy, and E Yahav. 2019. code2vec: learning distributed representations of code. *POPL* 3 (1 2019), 1–29. https://doi.org/10.1145/3290353

[19] M S B Altaf and D A Wood. 2017. LogCA: A High-Level Performance Model for Hardware Accelrators. *ICSA* (2017).

[20] K Angstadt, J B Jeannin, and W Weimer. 2020. Accelerating Legacy String Kernels via Bounded Automata Learning. *ASPLOS*. https://doi.org/10.1145/3373376.3378503

[21] M Barbar, Y Sui, and S Chen. 2021. Object Versioning for Flow-Sensitive Pointer Analysis. *CGO*. https://doi.org/10.1109/cgo51591.2021.9370334

[22] G. Bergland. 1969. Fast Fourier transform hardware implementations — A survey. *IEEE Transactions on Audio and Electroacoustics* 17 (6 1969), 109–119. Issue 2. https://doi.org/10.1109/tau.1969.1162048

[23] L Bertaccini, L Benini, and F Conti. 2021. To Buffer, or Not to Buffer? A Case Study on FFT Accelerators for Ultra-Low-Power Multicore Clusters. *ASAP* (2021).

[24] S G Bhaskaracharya, J Demouth, and V Grover. 2020. Automatic Kernel Generation for Volta Tensor Cores. (2020). Available at https://arxiv.org/pdf/2006.12645.pdf.

[25] Francesco Biletta. 2021. *Automotive radar processing optimization by exploiting the hardware accelerator of radar sensor chip*. techreport. Politecnico di Torino.

[26] D Bittman, R Soule, E L Miller, V Shrivastav, P Mehra, M Boisvert, A Silberschatz, and P Alvaro. 2021. Don't let RPCs Constrain Your API. *HotNets* (2021).

[27] Gabriel Hjort Blindell. 2018. *Universal Instruction Selection*. phdthesis. KTH Royal Institute of Technology.

[28] J Bornholt, E Torlak, D Grossman, and L Ceze. 2016. Optimizing Synthesis with Metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 775–788. https://doi.org/10.1145/2837614.2837666

[29] E Brunvand, D Kline, and A K Jones. 2018. Dark Silicon Considered Harmful: A Case for Truly Green Computing. *International Green and Sustainable Computing Conference* (2018).

[30] L Buch and A Andrzejak. 2019. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. *SANER*. https://doi.org/10.1109/saner.2019.8668039

[31] C Cadar, D Dunbar, and D Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *OSDI* (2008).

[32] B Calder, P Feller, and A Eustae. 1997. Value Profiling. *Micro* (1997), 259–269.

[33] J Chen, Y Lei, Y Peng, T He, and Z Deng. 2016. Configurable Floating-Point FFT Accelerator on FPGA Based Multiple-Rotation CORDIC. *Chinese Journal of Electronics* 25, 6 (2016).

[34] X Chen, R Bajaj, Y Chen, J He, B He, W F Wong, and D Chen. 2019. On-The-Fly Parallel Data Shuffling for Graph Processing on OpenCL-based FPGAs. (2019).

[35] X Chen, Y Lei, Z Lu, and S Chen. 2018. A Variable-Size FFT Hardware Accelerator Based on Matrix Transposition. *VLSI* 26 (10 2018), 1953–1966. Issue 10. https://doi.org/10.1109/tvlsi.2018.2846688

[36] H Cilasun, S Resch, Z I Chowdhury, E Olson, M Zabihi, Z Zhao, T Peterson, J P Wang, S S. Sapatnekar, and U Karpuzcu. 2020. CRAFFT: High Resolution FFT Accelerator In Spintronic Computational RAM. *DAC*. https://doi.org/10.1109/dac18072.2020.9218673

[37] B Collie, P Ginsbach, and M F.P. O'Boyle. 2019. Type-Directed Program Synthesis and Constraint Generation for Library Portability. *PACT* (2019).

[38] B Collie, P Ginsbach, J Woodruff, A Rajan, and M F.P. O'Boyle. 2020. M3: Semantic API Migration. *ASE* (2020).

[39] B Collie and M FP O'Boyle. 2021. Program Lifting using Gray-Box Behavior. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 60–74.

[40] B Collie, J Woodruff, and M FP O'Boyle. 2020. Modeling black-box components with probabilistic synthesis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 1–14.

[41] M Cowan, T Moreau, T Chen, J Bornholt, and L Ceze. 2020. Automatic generaiton of high-performance quantized machine learning kernels. *CGO* (2020).

[42] C Cummins, Z V. Fisches, T Ben-Nun, T Hoefler, and H Leather. 2021. PROGRAML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. *ICML* (2021).

[43] V Dadu, J Weng, S Liu, and T Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. *MICRO*. https://doi.org/10.1145/3352460.3358276

[44] W J. Dally, Y Turakhia, and S Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63 (6 2020), 48–57. Issue 7. https://doi.org/10.1145/3361682

[45] J P L De Carvalho, B Kuzma, I Korostelev, J N Amaral, C Barton, J Moreira, and G Araujo. 2021. KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls. *TACO* (2021).

[46] Danny Dig and Ralph Johnson. 2006. How do APIs evolve? A story of refactoring. *SMR* 18 (2006), 83–107.

[47] J Domke, E Vatai, A Drozd, P Chen, Y Oyama, L Zhang, S Salaria, D Mukunoki, A Podobas, M Wahib, and S Matsuoka. 2021. Matrix Engines for High Performance Computing: A Paragon of Performance or Grasping at Straws? *IEEE PDPS* (2021).

[48] B Duan, X Wang, Wand Li, C Zhang, P Zhang, and N Sun. 2011. Floating-point mixed-radix FFT core generation for FPGA and comparison with GPU and CPU. *FPT*. https://doi.org/10.1109/fpt.2011.6132672

[49] P Duhamel and M Vetterli. 1990. Fast Fourier Transforms: A Tutorial Review and a State of the Art. *Signal Processing* (1990), 259–299.

[50] S Durrani, M S Chughtai, M Hidayetoglu, R Tahir, A Dakkak, L Rauchwerger, F Zaffar, and W Hwu. 2021. Accelerating Fourier and Number Theoretic Transforms using Tensor Cores and Warp Shuffles. *PACT* (2021).

[51] C Fang, Z Liu, Y Shi, J Huang, and Q Shi. 2020. Functional Code Clone Detection with Syntax ans Semantics Fusion Learning. *ISSTA* (2020).

[52] Björn Franke. 2010. C Compilers and Code Optimization for DSPs. *Handbook of Signal Processing Systems* (7 2010), 575–601. https://doi.org/10.1007/978-1-4419-6345-1_21

[53] Matteo Frigo. 1999. A fast Fourier transform compiler. *ACM SIGPLAN Notices* 34 (5 1999), 169–180. Issue 5. https://doi.org/10.1145/301631.301661

[54] Z Fu and Z Su. 2016. XSat: A Fast Floating-Point Satisfiability Solver. *Computer Aided Verification* (2016), 187–209.

[55] Y Gao, Z Wang, S Liu, L Yang, W Sang, and Y Cai. 2019. TECCD: A Tree Embedding Approach for Code Clone Detection. *ICSME* (2019).

[56] Mario Garrido. 2021. A Survey on Pipelined FFT Hardware Architectures. *Journal of Signal Processing Systems* (2021).

[57] Philip Ginsbach. 2019. *From Constraint Programming to Heterogeneous Parallelism*. phdthesis. University of Edinburgh.

[58] P Ginsbach, B Collie, and M F. P. O'Boyle. 2020. Automatically harnessing sparse acceleration. *CC*. https://doi.org/10.1145/3377555.3377893

[59] P Ginsbach, T Remmelg, M Steuwer, B Bodin, C Dubach, and M F. P. O'Boyle. 2018. Automatic Matching of Legacy Code to Heterogeneous APIs. *ASPLOS*. https://doi.org/10.1145/3173162.3173182

- [60] Q Guo, N Alachiotis, B Akin, F Sadi, G Xu, T M Low, L Pileggi, J C Hoe, and F Franchetti. 2014. 3D-stacked memory-side acceleration: Accelerator and system design. *Workshop on Near-Data Processing* (2014).
- [61] M R Guthaus, J S Ringenberg, and D Ernst. 2001. MiBench: A free, comercially representative embedded benchmark suite. *Workshop on Workload Characterization* (2001), 3–14.
- [62] X Han, J Chen, B Qin, and S Rahardja. 2020. A Novel Area-Power Efficient Design for Approximated Small-Point FFT Architecture. *CADICS* 39 (12 2020), 4816–4827. Issue 12. https://doi.org/10.1109/tcad.2020.2978839
- [63] B Hardekopf and C Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. *CGO*. https://doi.org/10.1109/cgo.2011.5764696
- [64] W.H. Harrison. 1977. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering* SE-3 (5 1977), 243–250. Issue 3. https://doi.org/10.1109/tse.1977.231133
- [65] X He, L Xu, X Zhang, Y Feng, and B Xu. 2021. PyART: Python API Recommendation in Real-Time. *ICSE* (2021).
- [66] Michi Henning. 2009. API design matters. *Commun. ACM* 52, 5 (2009), 46–56.
- [67] R K. Hiware and D Padole. 2015. Configuration Memory Based Dynamic Coarse Grained Reconfigurable Multicore Architecture for 8 Point FFT. *ICETET*. https://doi.org/10.1109/icetet.2015.37
- [68] Q Huang, X Xia, Z Xing, D Lo, and X Wang. 2018. API method recommendation without worrying about the task-API knowledge gap. *ASE*. https://doi.org/10.1145/3238147.3238191
- [69] L Jiang, G Misherghi, Z Su, and S Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. *ICSE*. https://doi.org/10.1109/icse.2007.30
- [70] R Jiang, Z Chen, Z Zhang, Y Pei, M Pan, and T Zhang. 2018. [Research Paper] Semantics-Based Code Search Using Input/Output Examples. *SCAM*. https://doi.org/10.1109/scam.2018.00018
- [71] S Kamil, A Cheung, S Itzhaky, and A Solar-Lezama. 2016. Verified lifting of stencil computations. *PLDI* (2016).
- [72] P Kapur, B Cossette, and R J Walker. 2010. Refactoring References for Library Migration. *OOPSLA* (2010).
- [73] Christoph W. Kessler. 2019. Compiling for VLIW DSPs. *Handbook of Signal Processing Systems* (10 2019), 979–1020. https://doi.org/10.1007/978-3-319-91734-4_27
- [74] H Kim, Y Jung, S Kim, and K Yi. 2011. MeCC. *ICSE11*. https://doi.org/10.1145/1985793.1985835
- [75] D Kingma and J Ba. 2015. Adam: A method for stochastic optimization. *ICLR* (2015).
- [76] J Klainongsuang, Y S Nugroho, H Hata, B Manaskasemsak, A Rungsawang, P Leelaprute, and K Matsumoto. 2019. Identifying Algorithm Names in Code Comments. *CoRR* (2019). Available at https://arxiv.org/abs/1907.04557.
- [77] M Kristien, B Bodin, M Steuwer, and C Dubach. 2019. High-level synthesis of functional patterns with Lift. *ARRAY*. https://doi.org/10.1145/3315454.3329957
- [78] I Kundu, E Cottle, F Michel, J Wilson, and N New. 2021. The Dawn of Energy Efficient Computing: Optically Accelerating the Fast Fourier Transform Core. *OSA* (2021).
- [79] C Lattner, A Lenharth, and V Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. *PLDI*. https://doi.org/10.1145/1250734.1250766
- [80] B Li, S Cheng, and J Lin. 2021. tcFFT: Accelerating Half-Precision FFT through Tensor Cores. *CoRR* (2021). Available at https://arxiv.org/pdf/2104.11471.pdf.
- [81] D Li, A Pakala, and K Yang. 2022. MeNTT: A Compact and Efficient Processing-in-Memory Number Theoretic Transform (NTT) Accelerator. *VLSI* (2022).
- [82] Nian Liu. 2021. Characterizing Deprecated Deep Learning Python APIs: An Empirical Study on TensorFlow. (2021).
- [83] W Liu, Q Liao, F Qiao, W Xia, C Wang, and F Lombardi. 2019. Approximate Designs for Fast Fourier Transform (FFT) With Application to Speech Recognition. *IEEE Transactions on Circuits and Systems I: Regular Papers* 66 (12 2019), 4727–4739. Issue 12. https://doi.org/10.1109/tcsi.2019.2933321
- [84] D. B Lloyd, C Boyd, and N Govindaraju. 2008. Fast computation of general Fourier Transforms on GPUs. *ICME*. https://doi.org/10.1109/icme.2008.4607357
- [85] Joao D Lopes and Jose T de Sousa. 2016. Fast Fourier Transform on the Versat CGRA. *Silicon Errors Logic-System Effects* (2016), 174–187.
- [86] T Louw and S McIntosh-Smith. 2021. Using the Graphcore IPU for traditional HPC applications. *3rd Workshop on Accelerated Machine Learning (AccML)* (2021).
- [87] A J. Maas, H Nazaré, and B Liblit. 2016. Array length inference for C library bindings. *ASE*. https://doi.org/10.1145/2970276.2970310
- [88] Andrew MacLeod. 2018. Ranger: An on-demand range generate for GCC. (2018). Available at https://gcc.gnu.org/wiki/AndrewMacLeod/Ranger.
- [89] G Mathew, C Parnin, and K T Stolee. 2020. SLACC. *ICSE*. https://doi.org/10.1145/3377811.3380407
- [90] Mark McKeown. 2013. *FFT Implementation on the TMS320VC5505, TMS320C5505, and TMS320C5515 DSPs.* techreport SPRABB6B. Texas Instruments. Available at https://www.ti.com/lit/an/sprabb6b/sprabb6b.pdf.
- [91] C Mendis, J Bosboom, K Wu, S Kamil, J Ragan-Kelley, S Paris, Q Zhao, and S Amarasinghe. 2015. Helium: lifting high-performance stencil kernels from stripped x86 binaries to halide DSL code. *PLDI*. https://doi.org/10.1145/2737924.2737974
- [92] R. Meyer. 1989. Error analysis and comparison of FFT implementation structures. *International Conference on Acoustics, Speech, and Signal Processing*. https://doi.org/10.1109/icassp.1989.266571
- [93] A V Mohanan, C Bonamy, and P Augier. 2019. FluidFFT: Common API (C++ and Python) for Fast Fourier Transform HPC Libraries. *Journal of Open Research Software* 7 (4 2019). https://doi.org/10.5334/jors.238
- [94] S Mookherjee, L DeBrunner, and V DeBrunner. 2015. A low power radix-2 FFT accelerator for FPGA. *2015 49th Asilomar Conference on Signals, Systems and Computers*. https://doi.org/10.1109/acssc.2015.7421167
- [95] L Mou, G Li, L Zhang, T Wang, and Z Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. (2016).
- [96] Alastair Colin Murray. 2011. *Customising Compilers for Customisable Processors.* phdthesis. University of Edinburgh.
- [97] T D Nguyen, A T Nguyen, H D Phan, and T N. Nguyen. 2017. Exploring API Embedding for API Usages and Applications. *ICSE*. https://doi.org/10.1109/icse.2017.47
- [98] A Ni, D Ramos, A Yang, I Lynce, V Manquinho, R Martins, and C Le Goues. 2021. SOAR: A Synthesis Approach for Data Science API Refactoring. *ICSE* (2021). https://arxiv.org/pdf/2102.06726.pdf
- [99] B B Nielsen, M T Torp, and A Moller. 2021. Semantic Patches for Adaption of JavaScript Programs to Evolving Libraries. *ICSE* (2021).
- [100] T Nowatzki, V Gangadhar, K Sankaralingam, and G Wright. 2017. Domain Specialization Is Generally Unnecessary for Accelerators. *IEEE Micro* 37 (2017), 40–50. Issue 3. https://doi.org/10.1109/mm.2017.60
- [101] S Numata, N Yoshida, E Choi, and K Inoue. 2016. On the Effectiveness of Vector-Based Approach for Supporting Simultaneous Editing of Software Clones. *Product-Focused Software Process Improvement* (11 2016), 560–567. https://doi.org/10.1007/978-3-319-49094-6_41
- [102] A Paszke, S Gross, F Massa, A Lerer, J Bradbury, G Chanan, T Killeen, Z Lin, N Gimelshein, L Antiga, A Kopf, E Yang, Z DeVito, M Raison, A Tejani, S Chilamkurthy, B Steiner, L Fang, J Bai, and S Chintala. 2019.

PyTorch: An Imperative Style, High-Performance Deep Learning Library. *NeurIPs* (2019).

[103] D A Patel, V P Bui, K T C Chai, A Lal, and M M S Aly. 2022. SonicFFT: A system architecture of ultrasonic-based FFT acceleration. *ASP-DAC* (2022).

[104] A Pedram, J McCalpin, and A Gerstlauer. 2013. Transforming a linear algebra core to an FFT accelerator. *ASAP*. https://doi.org/10.1109/asap.2013.6567572

[105] H D Phan, A T Nguyen, T D Nguyen, and T N. Nguyen. 2017. Statistical Migration of API Usages. *ICSE-C*. https://doi.org/10.1109/icse-c.2017.17

[106] P M Phothilimthana, T Jelvis, R Shah, N Totla, S Chasins, and R Bodik. 2014. Chlorophyll: Synthesis-Aided Compiler for Low-Power Spatial Architectures. *PLDI* (2014).

[107] S Pophale and D Oryspayev. 2021. Outcomes of OpenMP Hackathon: OpenMP Application Experiences with the Offloading Mode. *IWOMP* (September 2021), 68–80.

[108] M Puschel, J M Moura, J Johnson, D Padua, Manuela M V, Bryan W S, J Xiong, F Franchetti, A Gacic, Y Voronenko, K Chen, R W Johnson, and N Rizzolo. 2004. SPRIAL: Code Generation for DSP Transforms. *Proc. IEEE* 93 (2004).

[109] J Ragan-Kelley, A Adams, D Sharlet, C Barnes, S Paris, M Levoy, S Amarasinghe, and F Durand. 2017. Halide. *Commun. ACM* 61 (12 2017), 106–115. Issue 1. https://doi.org/10.1145/3150211

[110] H Sajnani, V Saini, J Svajlenko, C K. Roy, and C V. Lopes. 2016. SourcererCC. *ICSE*. https://doi.org/10.1145/2884781.2884877

[111] M Samak, J P Cambronero, and M C Rinard. 2021. Searching for Replacement Classes. https://arxiv.org/pdf/2110.05638.pdf. *CoRR* (2021).

[112] M Samak, D Kim, and M C Rinard. 2020. Synthesizing Replacement Classes. *POPL* (2020).

[113] E Schkufza, R Sharma, and A Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. *PLDI* (2014).

[114] K Serebryany, D Bruening, A Potapenko, and D Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. *Usenix ATC* (2012).

[115] Amin Shafiee S, E Hansson, and C Kessler. 2013. Extensible Recognition of Algorithmic Patterns in DSP Programs for Automatic Parallelization. *IJPP* 41 (12 2013), 806–824. Issue 6. https://doi.org/10.1007/s10766-012-0229-2

[116] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. phdthesis.

[117] A Sorna, X Cheng, E D'Azevedo, K Won, and S Tomov. 2018. Optimizing the Fast Fourier Transform Using Mixed Precision on Tensor Core Hardware. *HiPCW*. https://doi.org/10.1109/hipcw.2018.8634417

[118] G Stewart, M Gowda, G Mainland, B Radunovic, D Vytiniotis, and Cristina L Agullo. 2015. Ziria. *ASPLOS*. https://doi.org/10.1145/2694344.2694368

[119] W Sun, S Sioutas, S Stuijk, A Nelson, and H Corporaal. 2021. Efficient Tensor Cores support in TVM for Low-Latency Deep learning. *DATE 2021* (2021).

[120] D Sundararajan. 2001. *The Discrete Fourier Transform*. World Scientific Publishing Co. Pte. Ltd.

[121] W J Tan, W T Tang, R S M Goh, S J Turner, and W F Wong. 2015. A Code Generation Framework for Targeting Optimized Library Calls for Multiple Platforms. *PDS* 26, 7 (July 2015), 1789–1799.

[122] Ping Tak Peter Tang. 2005. DFTI—a new interface for Fast Fourier Transform libraries. *ACM Trans. Math. Software* 31 (12 2005), 475–507. Issue 4. https://doi.org/10.1145/1114268.1114271

[123] Michael B. Taylor. 2012. Is dark silicon useful? *DAC*. https://doi.org/10.1145/2228360.2228567

[124] T Thanh-Hoang, A Shambayati, C Deutschbein, H Hoffmann, and A A. Chien. 2014. Performance and energy limits of a processor-integrated FFT accelerator, In 2014 IEEE High Performance Extreme Computing Conference (HPEC). *HPEC*. https://doi.org/10.1109/hpec.2014.7040951

[125] TH Tsai and HC Liu. 2021. Design and implementation of filterbank for MPEG-2/4 AAC system. *Integration* (2021).

[126] M D van de Burgwal, P T Wolkotte, and G J M Smit. 2009. Non-Power-of-Two FFTs: Exploring the Flexibility of the Montium TP. *International Journal of Reconfigurable Computing* (2009).

[127] A VanHattum, R Nigam, V T. Lee, J Bornholt, and A Sampson. 2020. A Synthesis-Aided Compiler for DSP Architectures (WiP Paper), In LCTES '20: 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. *LCTES*. https://doi.org/10.1145/3372799.3394358

[128] S I Venieris, I Panopoulos, I Leontiadis, and I Venieris. 2021. How to Readh Real-TIme AI on Consumer Devices? Solutions for Programmable and Custom Architectures. *ASAP* (2021).

[129] K Wang and Z Su. 2018. Learning Blended, Precise Semantic Program Embeddings. *CoRR* (2018). Available at https://arxiv.org/abs/1907.02136.

[130] M Wang, D Zheng, Z Ye, Q Gan, M Li, X Song, J Zhou, C Ma, L Yu, Y Gai, T Xiao, T He, G Karypis, J Li, and Z Zhang. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *CoRR* (2019). Avaialable at https://arxiv.org/pdf/1909.01315.

[131] R Weber, A Gothandaraman, R J Hinde, and G D Peterson. 2011. Comparing Hardware Accelerators in Scientific Applications: A Case Study. *PDS* (2011).

[132] H Wei and M Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code, In Twenty-Sixth International Joint Conference on Artificial Intelligence. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. https://doi.org/10.24963/ijcai.2017/423

[133] J Weng, A Jian, J Wang, L Wang, Y Wang, and T Nowatzki. 2021. UNIT: Unifying Tensorized Instruction Compilation. *CGO* (2021).

[134] M White, M Tufano, C Vendome, and D Poshyvanyk. 2016. Deep learning code fragments for code clone detection. *ASE* (2016).

[135] W Woo, S Park, S Kim, H Lee, and H Oh. 2021. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. *ICSE* (2021).

[136] J Woodruff and M F P O'Boyle. 2021. New Regular Expressions on Old Accelerators. *DAC 2021* (2021).

[137] C Xu, X Sun, B Li, X Lu, and H Guo. 2018. MULAPI: Improving API method recommendation with API usage location. *Journal of Systems and Software* 142 (8 2018), 195–205. https://doi.org/10.1016/j.jss.2018.04.060

[138] H E Yantir, W Guo, A M. Eltawil, F J. Kurdahi, and K N Salama. 2019. An Ultra-Area-Efficient 1024-Point In-Memory FFT Processor. *Micromachines* 10 (7 2019), 509. Issue 8. https://doi.org/10.3390/mi10080509

[139] J Zhang, X Wang, H Zhang, H Sun, K Wang, and X Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. *ICSE*. https://doi.org/10.1109/icse.2019.00086