# Abstracting Vector Architectures in Library Generators: Case Study Convolution Filters

Alen Stojanov[†] Georg Ofenbeck[†] Tiark Rompf [*‡] Markus Püschel[†]

[†]Department of Computer Science, ETH Zurich: {astojanov, ofgeorg, pueschel}@inf.ethz.ch
[‡]Oracle Labs: {first.last}@oracle.com [*]EPFL: {first.last}@epfl.ch

## Abstract

We present FGen, a program generator for high performance convolution operations (finite-impulse-response filters). The generator uses an internal mathematical DSL to enable structural optimization at a high level of abstraction. We use FGen as a testbed to demonstrate how to provide modular and extensible support for modern SIMD vector architectures in a DSL-based generator. Specifically, we show how to combine staging and generic programming with type classes to abstract over both the data type (real or complex) and the target architecture (e.g., SSE or AVX) when mapping DSL expressions to C code with explicit vector intrinsics. Benchmarks shows that the generated code is highly competitive with commercial libraries.

***Categories and Subject Descriptors*** I.2.2 [*Automatic Programming*]: Program synthesis, Program transformation; D.3.3 [*Programming Languages*]: Language Constructs and Features – Abstract data types; D.3.4 [*Programming Languages*]: Processors – Code generation, Optimization, Run-time environments

***Keywords*** Synthesis; Abstraction over Staging; Selective Precomputation; Scalar Replacement; Data Representation

## 1. Introduction

Numerical libraries need to be highly tuned to the platform's architecture and microarchitecture to reach highest performance. This tuning requires expensive programming effort, and conflicts with portability, since it often has to be repeated for every new processor generation. Over the last decade, one solution that has emerged to solve this problem are program generators that use domain-specific languages (DSLs) to express mathematical algorithms at a high level of abstraction, which is then compiled into platform-specific high performance code [2, 5, 7, 8, 15, 16, 22, 23, 27]. In some of these generators, DSLs are also used internally to perform difficult optimizations such as loop fusion or vectorization at a high level of abstraction through rewriting to overcome compiler limitations. Examples of this idea include Spiral [14] (a generator for linear transforms) which uses a DSL called $\Sigma$-SPL [6], and LGen [22] which uses an extension called $\Sigma$-LL.

These languages can be viewed as a domain-specific extension of the array programming paradigm, augmented with explicit data access objects and higher level mathematical operators. Intuitively, this representation makes it possible to restructure the computation to achieve the above optimizations.

An orthogonal question, investigated for example in [4, 13] is how to efficiently build such generators in an effective and maintainable way using modern programming language features. In particular, the concept of staging [25] has been proposed to build generators within a host language [1, 5]. Modern staging frameworks such as LMS (Lightweight Modular Staging) [17] go beyond primitives for emitting code and have become popular for implementing generators based on one or multiple levels of DSLs [13, 18, 19, 24]. However, only few existing generators target SIMD vector architectures, i.e., emit code that uses the so-called intrinsics interface to directly and efficiently use vector instructions.

**Contributions.** The first main contribution of this paper is a new library generator called FGen for a very narrow but important operation: convolution, or, as it is called in media processing, finite-impulse-response (FIR) filters. The generator takes as input a mathematical convolution expression including the size of two arrays involved and outputs an optimized library function. Internally a variant of the above-mentioned DSL $\Sigma$-LL is used to structure the computation and to facilitate the mapping to a vector architecture.

The second main contribution is a generic support layer for targetting vector architectures from DSL-based program generators. Specifically, we combine staging and generic programming using type classes to abstract at the DSL level over both the data representation (e.g. real or complex numbers) and the vector architecture (e.g., SSE or AVX). Extensions to new data types and new vector architectures thus become completely modular in the backend translation engine. This SIMD support layer is not specific to convolution but designed to be applicable to a large set of possible generators built with suitable DSLs.

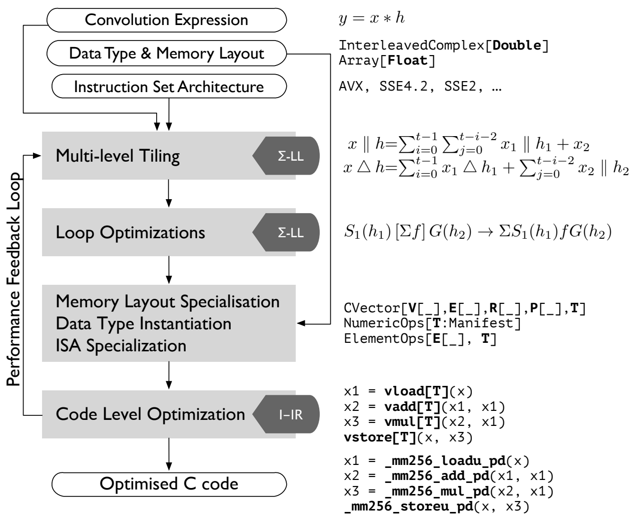
Finally we show benchmarks comparing our automatically generated convolution code with commercial high performance libraries to demonstrate the viability of our approach.

## 2. Overview

We present FGen, a program generator for performance-optimized functions implementing convolutions, or FIR filters. FGen follows [13] in design and implementation but extends the work to include filters, and to support vector ISAs in a modular way. Figure 1 gives a high-level overview of the generator. The input to FGen specifies the desired function to be generated. It consists of

1. the desired convolution, written mathematically as $y = h * x$ with specified sizes for the vectors $x, h, y$;
2. the data type and memory layout (e.g., real, interleaved or split complex); and
3. the vector ISA (e.g., Intel SSE or Intel AVX).

We give a short overview on the inner workings next and follow up with more details in Section 3. In the first step, FGen formally tiles the computation for better locality using the mathematical language $\Sigma$-LL that slightly extends the aforementioned $\Sigma$-SPL [6] and $\Sigma$-LL [22]. In essence, the language consists of vectors, matrices, and data access objects as operands, as well as linear algebra operations including addition, multiplication, and different



$y = x * h$

```
InterleavedComplex[Double]
Array[Float]
```

```
AVX, SSE4.2, SSE2, …
```

$x \parallel h = \sum_{i=0}^{t-1} \sum_{j=0}^{t-i-2} x_1 \parallel h_1 + x_2$

$x \triangle h = \sum_{i=0}^{t-1} x_1 \triangle h_1 + \sum_{j=0}^{t-i-2} x_2 \parallel h_2$

$S_1(h_1)\,[\Sigma f]\,G(h_2) \rightarrow \Sigma S_1(h_1) f G(h_2)$

```
CVector[V[_],E[_],R[_],P[_],T]
NumericOps[T:Manifest]
ElementOps[E[_], T]
```

```
x1 = vload[T](x)
x2 = vadd[T](x1, x1)
x3 = vmul[T](x2, x1)
vstore[T](x, x3)
```

```
x1 = _mm256_loadu_pd(x)
x2 = _mm256_add_pd(x1, x1)
x3 = _mm256_mul_pd(x2, x1)
_mm256_storeu_pd(x, x3)
```

Figure 1: FGen Architecture

forms of convolution. The chosen tile size is a search parameter but is chosen to be a multiple of the desired vector length of the specified ISA. $\Sigma$-LL makes loops and access patterns explicit, thus enabling loop optimizations like merging in the next step.

After the proper mathematical structure is established, the data type and memory layout parameters are introduced. The code style (e.g., to what degree unrolled) is a search parameter that is also fixed at this time. These choices are implemented using abstractions as explained in [13]. Next, the desired vector ISA is introduced and code is generated in an intermediate representation (IR) for vector code. The choice of vector ISA and the data type and layout are mutually dependent. For instance, when dealing with interleaved complex arrays, we must specialize to suitable shuffles that interleave and de-interleave complex numbers. Handling the interaction of data layout and vector ISA in a modular fashion is very complex and a main contribution of this paper (see Section 3.4).

Finally, the code is unparsed to C including vector intrinsics for SSE or AVX. The performance is measured and informs a feedback-driven search to find the best choices for all the parameters and choices mentioned before. The best code found is returned as final result.

## 3. Implementation

In this section we explain FGen in Fig. 1 in greater detail, focusing on the most important components and ideas.

### 3.1 Input to FGen

The input to FGen is a filter $y = x * h$ (or one of the variants mentioned below) including the sizes of the arrays. The second input is the data type (single or double precision, real or complex) and data layout for complex numbers (interleaved or split). These are specified through higher-kinded types as explained in [13]. Finally, the desired ISA is specified through a simple identifier. The interaction of the ISA and the data types is explained in Section 3.4.

| MATLAB notation | $\Sigma$-LL notation |
|---|---|
| `y = x(a:b)` | $y = G(h_a^{b-a+1\rightarrow n}) \cdot x$ |
| `y(a:b) = x` | $y = S(h_a^{b-a+1\rightarrow N}) \cdot x$ |

Figure 2: Gathers and Scatters using MATLAB notation

### 3.2 Tiling

The convolution $y = x*h$ of two vectors $x = (x_0, x_1, \ldots, x_{n-1})^T$ and $h = (h_0, h_1, \ldots, h_{k-1})^T$ is defined as[1]

$$y_m = \sum_{i=0}^{k-1} x_{m+(k-1)-i} \cdot h_i, \quad m = 0, \ldots, n-1. \tag{1}$$

We use the language of media processing and call $h$ the filter, $k$ the number of taps, and the computation finite-impulse-response filter (FIR). The filter can also be viewed as a matrix vector product:

$$y = x * h \Longleftrightarrow y = M_n(h) \cdot x, \tag{2}$$

where $M_n(h)$ is the $n \times n$ matrix

$$\begin{bmatrix} h_{k-1} & h_{k-2} & \cdots & h_0 & & & \\ & \ddots & \ddots & \ddots & \ddots & & \\ & & h_{k-1} & h_{k-2} & \cdots & h_0 \\ & & & h_{k-1} & \cdots & h_1 \\ & & & & \ddots & \vdots \\ & & & & & h_{k-1} \end{bmatrix}. \tag{3}$$

To improve locality and thus performance, FGen tiles the initial filter into a sum of smaller FIR filters. To express this mathematical we use the language $\Sigma$-LL. It includes the convolutions introduced before, and gather and scatter matrices that represent data accesses. Gathers and scatters are parameterized by a data access function. For our purpose we only consider one type of function from domain $\{0, 1, \ldots, n-1\}$ to range $\{0, 1, \ldots, N-1\}$:

$$h_b^{n\rightarrow N} : i \mapsto b + i. \tag{4}$$

To describe the basic idea of the gathers and scatters, we use MATLAB-like notation. We assume two vectors $x$ and $y$, having sizes $n$ and $N$ respectively, and parameters $a$ and $b$. Figure 2 shows the MATLAB equivalents of the used gathers and scatters. Formally, a gather is a wide matrix that has one 1 in every row and is 0 elsewhere; a scatter is a transposed gather.

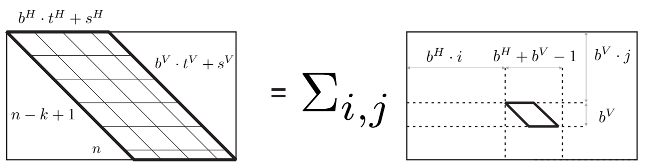
Using this notation we can now decompose the FIR filter $y = x*h$. First, we observe that in (3), the first $n - k + 1$ rows form a "parallelogram" shape matrix, and the last $k - 1$ rows form an upper triangular matrix. Correspondingly, we can decompose the filter into a concatenation ($\oplus$) of two specialized filters that we call parallelogram shape filter (PFIR) and upper triangular filter (UTFIR):

$$x * h = S(h_0^{n-k+1\rightarrow n}) \cdot x \parallel h + S(h_{n-k+1}^{k-1\rightarrow n}) \cdot x' \triangle h', \tag{5}$$

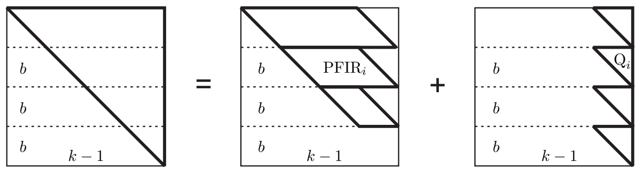
where $x' = G(h_{n-k+1}^{k-1\rightarrow n}) \cdot x$ and $h' = G(h_1^{k-1\rightarrow k}) \cdot h$. In the following these two filters are decomposed separately. Each decomposition has degrees of freedom which are searched over by FGen.

**PFIR Decomposition** We decompose PFIR by tiling in both the horizontal and vertical direction as shown in Fig. 3a. Let's assume

[1] We note that there are different variants of convolution depending on the handling of the boundaries.



(a) PFIR tiling



(b) UTFIR mixed tiling

Figure 3: FIR Tiling

that $n-k+1 = b^H \cdot t^H + s^H$ and $k = b^V \cdot t^V + s^V$; $t^H$ and $t^V$ define the loop bounds for each tile, and $b^H$ and $b^V$ define the size of the tile. The resulting decomposition is

$$\begin{array}{rcl} x \parallel h & = & \sum_{i=0}^{t^H-1} S(h_{b^H \cdot i}^{b^H \to n-k+1}) \sum_{j=0}^{t^V-1} (x_1 \parallel h_1) \\ & + & \sum_{i=0}^{t^H-1} S(h_{b^H \cdot i}^{b^H \to n-k+1})(x_2 \parallel h_2) \\ & + & S(h_{b^H t^H}^{s^H \to n-k+1}) \sum_{j=0}^{t^V-1} (x_3 \parallel h_1) \\ & + & S(h_{b^H t^H}^{s^H \to n-k+1})(x_4 \parallel h_2) \end{array} \quad (6)$$

such that:

$$\begin{array}{rclrcl} x_1 & = & G(h_{b^H \cdot i + b^V \cdot j}^{b^H + b^V - 1 \to n}) \cdot x & x_4 & = & G(h_{b^H t^H + b^V t^V}^{s^H + s^V - 1 \to n}) \cdot x \\ x_2 & = & G(h_{b^V t^V + b^H \cdot i}^{b^H + s^V - 1 \to n}) \cdot x & h_1 & = & G(h_{k - b^V \cdot (j+1)}^{b^V \to k}) \cdot h \\ x_3 & = & G(h_{b^H t^H + b^V \cdot j}^{s^H + b^V - 1 \to n}) \cdot x & h_2 & = & G(h_0^{s^V \to k}) \cdot h \end{array}$$

Note that the somewhat complicated appearance is due to the specification of domain and range sizes in the gathers and scatters.

**UTFIR Decomposition**. We decompose UTFIR by tiling as shown in Fig. 3b. If $k-1 = b \cdot t + s$, this means a decomposition into $t$ smaller UTFIRs and $t-1$ PFIRs. These can be further decomposed recursively.

In Σ-LL this decomposition reads as

$$\begin{array}{rcl} x \bigtriangleup h & = & \sum_{i=0}^{t-1} S(h_{b \cdot i}^{b \to k}) \sum_{j=0}^{t-i-2} (x_1 \parallel h_1) \\ & + & \sum_{i=0}^{t-1} S(h_{b \cdot i}^{b \to k})(x_2 \parallel h_2) \\ & + & \sum_{i=0}^{t-1} S(h_{b \cdot i}^{b \to k})(x_3 \bigtriangleup h_3) \\ & + & S(h_{b \cdot t,1}^{s \to k})(x_4 \bigtriangleup h_4) \end{array} \quad (7)$$

where:

$$\begin{array}{rclrcl} x_1 & = & G(h_{bi+bj}^{2b-1 \to k}) \cdot x & h_1 & = & G(h_{k-b(j+1)}^{b \to k}) \cdot h \\ x_2 & = & G(h_{b(t-1)}^{s+b-1 \to k}) \cdot x & h_2 & = & G(h_{b(i+1)}^{s \to k}) \cdot h \\ x_3 & = & G(h_{k-b}^{b \to k}) \cdot x & h_3 & = & G(h_{b \cdot i}^{b \to k}) \cdot h \\ x_4 & = & G(h_{k-s}^{s \to k}) \cdot x & h_4 & = & G(h_{k-s}^{s \to k}) \cdot h \end{array}$$

**ISA Tiling**. Equations (6) and (7) include three important parameters for the tiling, namely $b^H$, $b^V$, and $b$. If a vector ISA and hence an associated (ISA) vector length $\nu$ is specified, we make sure that the innermost tiling is a multiple of $\nu$ for efficient mapping to intrinsics.

### 3.3 Loop Optimizations

Loop optimizations such as loop merging, or loop unrolling are done at the Σ-LL level using a rewrite system that fuses, for example data accesses. As one example,

$$\sum_{i=0}^{p-1} S(h_b^{N_1 \to N}) \sum_{j=0}^{q-1} S(h_{b\prime}^{n \to N_1})$$

is rewritten into

$$\sum_{i=0}^{p-1} \sum_{j=0}^{q-1} S(h_{b+b\prime}^{n \to N_1}).$$

### 3.4 Abstraction over Data Representation, Code Style and Vectorization

Once an optimized Σ-LL program is reached it is converted to the I-IR / C-IR DSL. While Σ-LL deals with the representation of mathematical vectors, C-IR and I-IR facilitate the encoding of data representations of these vectors. For each data representation this includes all array operations composed by memory access functions and numerical computations in their scalar and vectorized versions respectively. We abstract all of those different possibilities into a single data abstraction, which provides an interface that is very similar to its mathematical and conceptual equivalent in Σ-LL. These abstractions are implemented using staging, type classes and higher kinded types. All DSLs used in FGen are implemented through LMS. Translation from one DSL to another is performed via a *staged interpreter* approach as described in [13, 18]. While the implementation details are not relevant for Σ-LL, they are a key enabler in efficient abstraction at the I-IR and C-IR level. We give a quick, non self-contained, overview over the main concepts in the next few paragraphs.

**Staging.** Multi-stage programming [25] is a technique that allows to interleave a program generator with parts of the code to be generated within the same program. Traditionally this is done through annotations within the code, which requires a specialized compiler such as MetaOCaml. Lightweight Modular Staging [17] is a framework with the same goal, but instead of annotations it uses types to distinguish generator code and code for the next stage. Within LMS the type that defines future-stage expressions is called `Rep[T]`. Loosely speaking LMS overloads all operations on standard types, such that for each operation on type `T`, there exists a staged version on type `Rep[T]`. In Pseudo-Code the overload for the Plus operator on Integers would take the form:

```
def + (a: Int, b: Int) = a + b
def + (a: Rep[Int], b: Rep[Int]) = {
        CreateASTNodePlus(a,b) }
```

Using this construct the following code

```
a: Int; b: Int, c: Int
c = a + b
> c: Int = ... // Result of a + b
```

performs a regular addition while the code

```
a: Rep[Int]; b: Rep[Int], c: Rep[Int]
c = a + b
> c: Rep[Int] = plus(a, b) // AST Node
```

is redirected to the overloaded version, which in turn creates an AST Node representing the computation.

**Abstraction over staging decisions.** Utilizing the fact that staging within LMS is controlled through types allows us to use them in the context of type polymorphic functions and classes. Given a function f, which is polymorphic in type `T`, we can instantiate two versions, by applying arguments of different types:

```
def f[T](lhs: T, rhs: T) = lhs + rhs
// regular computation
a: Int; b: Int, c: Int
```

```
type T = Double
add[Rep,Real,NoRep,SISD,T]
```

```
#define T double
#define N 4
void add(T* x, T* y, T* z) {
  int i = 0;
  for(; i < N; ++i) {
    T x1 = x[i];
    T y1 = y[i];
    T z1 = x1 + y1;
    z[i] = z1;
  }
}
```

(a) Staged SISD Array

```
type T = Double
add[NoRep,Real,Rep,SISD,T]
```

```
#define T double
void add(T* x, T* y, T* z) {
  T x1 = x[0]; T x2 = x[1];
  T x3 = x[2]; T x4 = x[3];
  T y1 = y[0]; T y2 = y[1];
  T y3 = y[2]; T y4 = y[3];
  T z1 = x1 + y1;
  T z2 = x2 + y2;
  T z3 = x3 + y3;
  T z4 = x4 + y4;
  z[0] = z1; z[1] = z2;
  z[2] = z3; z[3] = z4;
}
```

(b) SISD Array of Staged Doubles

```
type T = Double
add[Rep,Real,NoRep,SIMD,T]
```

```
#define T double
#define N 1
void add(T* x, T* y, T* z) {
  int i = 0;
  for(; i < N; ++i) {
    __m256d x1, y1, z1;
    x1 = _mm256_loadu_pd(x + i);
    y1 = _mm256_loadu_pd(y + i);
    z1 = _mm256_add_pd(x1, y1);
    _mm256_storeu_pd(z + i, y1);
  }
}
```

(c) Staged SIMD Array

```
type T = Double
add[NoRep,Real,Rep,SIMD,T]
```

```
#define T double
void add(T* x, T* y, T* z) {
  __m256d x1, y1, z1;
  x1 = _mm256_loadu_pd(x + 0);
  y1 = _mm256_loadu_pd(y + 0);
  z1 = _mm256_add_pd(x1, y1);
  _mm256_storeu_pd(z + 0, y1);
}
```

(d) SIMD Array of Staged Doubles

Figure 4: Different Data Type Instantiations result with different code style (assuming arrays of size 4 and AVX as an ISA)

```
c = f(a,b)
// staged computation
a: Rep[Int]; b: Rep[Int], c: Rep[Int]
c = f(a,b)
```

This also applies to Scala for comprehensions which Scala treats as regular functions, with a parameter of type Range that can be overloaded in a similar fashion:

```
for (range: Range     ) { body } // Regular loop
for (range: Rep[Range]) { body } // AST loop node
```

The first version executes the body expression, and the second version creates an AST node representing a loop that includes the body expression.

**Encapsulating staging decisions.** FGen heavily utilizes this mechanism to abstract over staging decisions as described in full detail in [13]. For this work, these abstraction have been extend further to also include vectorization. Within I-IR the operands of our DSL are highly polymorphic classes that take the shape:

```
class CVector[V[_], E[_], R[_], P[_], T](...) {
  type Element = E[R[P[T]]]
  def  size   (): V[Int]
  def  apply  (i: V[Int])  : Element
  def  update (i: V[Int], v: Element)
}
```

For simplicity the code above omits the type classes that are implicitly passed together with each type parameter. Within each of the type parameters provided to the CVector class we encode part of the abstraction, by providing abstract composable interfaces that are implemented through those type classes.

- T describes the underlying array primitive (double, float, etc).
- P[_] describes whether we deal with SIMD or SISD instructions. It is accompanied by a type class that abstracts SIMD specific operators such as shuffle, hadd, vadd etc., and SISD specific operations such as addition, multiplication etc.
- R[_] describes whether we stage the elements of the array. The accompanying type class abstracts numerical operations for both staged and non-staged version.
- E[_] describes the the structure of one array element. E.g. it can be complex numbers consisting of two primitves or directly primitves. E[_]. Abstractions for numerical operations relative to the element such as addition, multiplications, complex number interleaving are implemented in the accompanied type class.
- V[_] encodes whether array accesses will be visible in the target code element operations.

A concrete choice of a data layout, code style and the vectorization is done through instantiating the CVector class accordingly, e.g.,

```
class Scalar_SISD_Double_Vecor extends
 CVector[NoRep, Real, Rep, SISD, Double] ...
```

where the type NoRep is a higher order type defined as

```
type NoRep[T] = T
```

In the translation process from Σ-LL to I-IR, this enables us to define the target translations in terms of the type polymorphic base class CVector as e.g.

```
def add[V[_], E[_], R[_], P[_], T] (
  (x, y, z) : Tuple3[CVector[V,E,R,P,T]]
) = for ( i <- 0 until x.size() ) {
  z(i) = x(i) + y(i)
}
```

Concrete implementations can be picked by passing corresponding instantiations of the base class to the function. Figure 4 illustrates this for four variants that could be passed to the add functions.

Staging decisions are of crucial importance to FGen. When tiling is performed to fit the SIMD vector length, SIMD instantiations are performed to generate the vectorized part of the code; the leftover computation is instantiated as unvectorized SISD code. When tiling for registers, data structures are properly instantiated to generate the unrolled version of the code. Data abstraction and staging decisions are used to provide a single codebase that will fit all generated code versions.

## 3.5 ISA Abstraction

Once data types are fixed, the next step is to replace each Σ-LL expression with one or several I-IR expressions. The Intrinsics IR is ISA independent and it only specializes to a particular ISA, once this argument is fixed in the generator. The specialization to a particular ISA is inter-dependent on the data abstraction. We observe this in the case of Interleaved Complex Array. Real and imaginary parts must be interleaved when data is loaded or stored. To achieve this, the ISA abstraction calls the corresponding shuffle, unpackhi and unpacklo instructions, to perform the desired interleaving.

**FIR intrinsics.** Once the ISA is fixed, we perform the conversion of Σ-LL expressions to I-IR. FGen implements translation of Σ-LL to I-IR in a single codebase and uses staging decisions to generate different code versions. The actual SIMD conversion of the PFIR filter is given below:

```
class SigmaLL2CIRTrasnlator[E[_], R[_], P[_], T] {
  type Element = E[R[P[T]]]
  def sum (in: List[Element]) =
    if (in.length == 1) in(0) else {
      val (m, e) = (in.length / 2, in.length)
      sum(in.slice(0, m)) + sum(in.slice(m, e))
    }
  def translate(stm: Stm) = stm match {
    case TP(y, PFIR(x, h)) =>
      val xV = List.tabulate(k)(i => x.apply(i))
      val hV = List.tabulate(k)
        (i => h.vset1(h.apply(k-i-1)))
      val tV = (xV, hV).zipped map (_*_)
      y.update(y(0), sum(tV))
  }
}
```

Note that if ISA is not specified, the code snippet will result with a construction of SISD C-IR code.

### 3.6 Code Level Optimizations

Code level optimizations are done on the C-IR DSL. Most of these are already provided by LMS. Those include common subexpression elimination, dead code removal and code motion.

## 4. Experimental Results

In this section we show performance benchmarks with FGen generated code against current commercial libraries.

**Experimental setup.** We benchmarked on two machines, Intel(R) Xeon(R) CPU E5-2643 3.3 GHz supporting AVX, running Ubuntu 13.10, kernel v3.11.0-12-generic, and Intel(R) Core(TM)2 Duo CPU L7500 1.6GHz supporting SSSE3, running Debian 7, kernel v3.2.0-4-686-pae. Intel's Hyper-Threading, Turbo Boost (Xeon) and Intel Dynamic Acceleration (Core2) were disabled on both machines during the tests. We compare against convolutions from Intel IPP v8.0.1 and Intel MKL v11.1.1. Note that in both, the vector lengths are parameters in contrast to our generated specialized code. As base line we also include a straightforward implementation of convolution: a double loop corresponding to (1) with fixed array sizes.

All code is compiled using the Intel C++ Composer 2013.SP1.1.106, with flags `-std=c99 -O3 -xHost`.

We only consider double precision code (4-way on AVX and 2-way on SSSE3). The input sizes, related to the input vector of the convolution expression, are powers of two in the form of $n = 512 \cdot 2^i$ for $i = 1, \dots, 16$ to ensure a sampling of all cache levels for both machines. For each machine we perform two types of tests:

(a) All vectors are arrays of real numbers, and the filter size is 8 or 20;

(b) All vectors are arrays of interleaved complex numbers, and the filter size is 8 or 20 (complex numbers).

Time is measured under warm-cache conditions, using a two loops measuring strategy. The inner loop measures time as the mean of sufficently many iteration; the outer loop returns the median of several such runs.

Figure 5 gives an overview of the results. All plots show the size of the input vector on the x-axis and the performance in flops per cycle (f/c) on the y-axis. The theoretical peak performance of the platform is represented with a horizontal line in each plot. We discuss real (left four plots) and complex (right four plots) convolutions separately.

**Real convolution.**. FGen-generated code outperforms the other implementations, except IPP for small sizes and 20 taps. The reason is not clear as the code is distributed as binary, which prevents inspection. In some cases MKL performs worse than the base implementation. Apparently, `icc` can efficiently optimize and vectorize the simple double loop with fixed bounds.

**Complex convolution.**. For large sizes FGen-generated code is faster (AVX) or roughly competitive (SSSE3) with the next best IPP. Again, MKL performs worse than the straightline code with a similar possible explanation as above. We note that in FGen there is further room for improvement in the shuffling needed to work on interleaved data. We believe that the gains for larger sizes on AVX are due to a more thorough exploration of the possible tiling strategies in FGen.

**Remarks.**. We note for longer sizes of the filter $h$, both IPP and MKL outperform FGen due to the use of FFTs, which reduces the asymptotic runtime from $O(nk)$ to $O(n \log k)$. FGen does not support FFT-based convolution at this time.

## 5. Related Work

The array programming paradigm favors computing on collections of data as a whole over element-at-a-time processing. Besides a higher-level programming style, the key benefit of going from scalar values to vectors of data as core computational units is that array computation is implicitly parallel, and easy to map to SIMD instructions by a compiler. Starting with a stylized mathematical array notation that lead to APL [9] and its successors, J [3] or K [11], array languages have focused on user-facing constructs that enable programmers to abstract over the rank, dimension or in general shape of the data. The same holds for Sisal [12] or, more recently, SAC [20], which has also inspired embedded DSLs [10, 26]. Compilers for all these languages attempt to generate SIMD intrinsics to varying degrees, e.g., by using type inference in the case of SAC [21], but in general do not expose this fact to the programmer in any way. Our focus in this paper has been on using an array-style language, $\Sigma$-LL, as an intermediate language in a program generator stack. As opposed to other array languages, $\Sigma$-LL models only single-dimensional arrays, i.e., does not provide shape polymorphism in the usual sense. The translation from $\Sigma$-LL to I-IR/C-IR and optimized C code, however, is highly parametric in data layout, vector ISA, and other parameters. This is again in contrast to user-facing languages, where the low-level part of the compilation is hidden from the programmers. We believe that our $\Sigma$-LL / I-IR combination offers a sweet spot in the design space between fully opaque array languages (which do not offer fine grained control about vectorization) and ISA specific intrinsics as provided by C compilers (which provide full control but are too low-level and cumbersome to use).

## References

[1] B. Aktemur, Y. Kameyama, O. Kiselyov, and C.-c. Shan. Shonan challenge for generative programming: short position paper. In *Proc. Partial evaluation and program manipulation (PEPM)*, pages 147–154, 2013.

[2] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. Quintana-Orti, and R. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. on Mathematical Software*, 31(1):1–26, 2005.

[3] C. Burke and R. Hui. J for the APL programmer. *SIGAPL APL Quote Quad*, 27(1):11–17, Sept. 1996.

[4] A. Cohen, S. Donadio, M. jesus Garzaran, C. Herrmann, and D. Padua. In search of a program generator to implement generic transformations

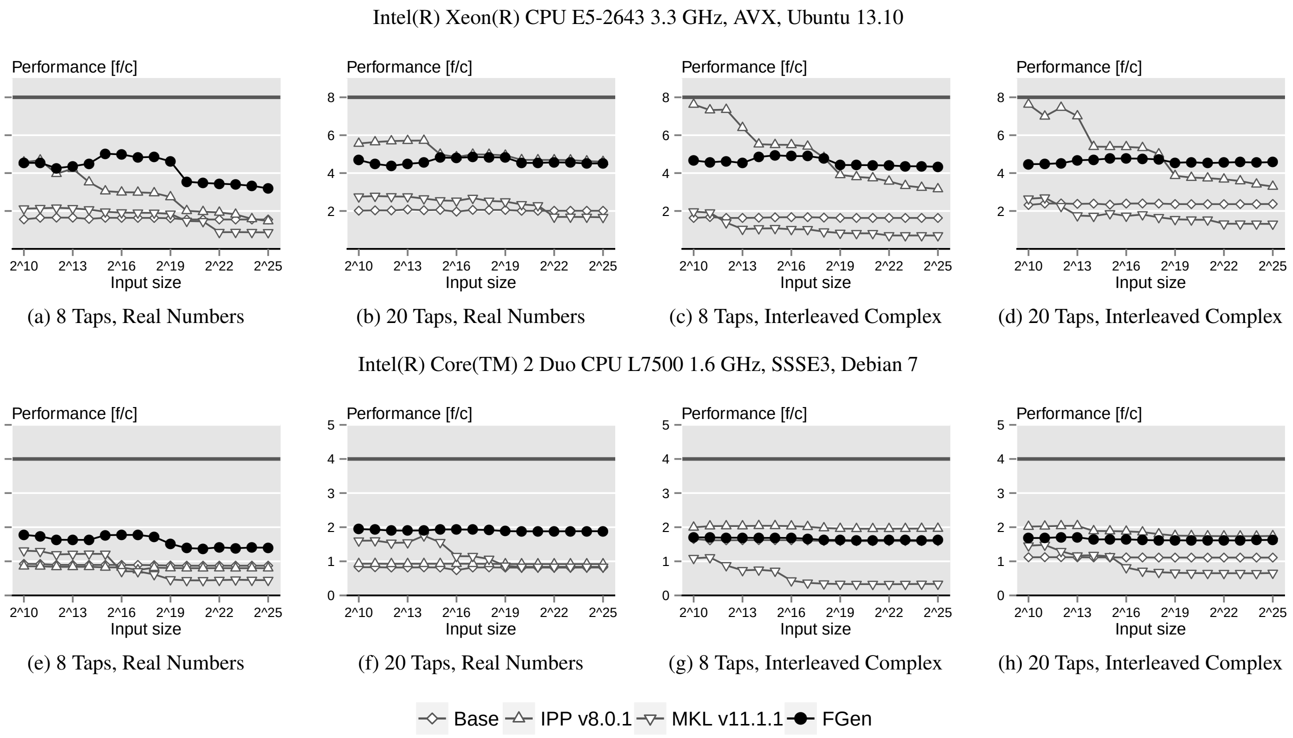


Figure 5: `FGen` Performance compared to IPP, MKL and Base implementation

for high-performance computing. In *1 st MetaOCaml Workshop (associated with GPCE*, pages 166771–7, 2004.

[5] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A multi-stage language for high-performance computing. *SIGPLAN Not.*, 48(6):105–116, June 2013.

[6] F. Franchetti, Y. Voronenko, and M. Püschel. Formal loop merging for signal transforms. In *Programming Languages Design and Implementation (PLDI)*, pages 315–326, 2005.

[7] M. Frigo. A fast Fourier transform compiler. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 169–180, 1999.

[8] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. on Mathematical Software*, 27(4):422–455, 2001.

[9] K. E. Iverson. *Programming Language*. John Wiley & Sons Inc, 1962.

[10] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. volume 45, pages 261–272, New York, NY, USA, Sept. 2010. ACM.

[11] Kx Systems. K reference manual version 2.0. 1998.

[12] J. McGraw, S. Skedzielewski, S. Allan, and R. Oldehoeft. Sisal: Streams and iteration in a single assignment language: Reference manual version 1.2. 1998.

[13] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in Scala: Towards the systematic construction of generators for performance libraries. In *International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pages 125–134, 2013.

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

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

[16] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In H.-J. Boehm and C. Flanagan, editors, *PLDI*, pages 519–530. ACM, 2013.

[17] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[18] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. In *Proc. Principles of programming languages (POPL)*, pages 497–510, 2013.

[19] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented DSLs. DSL, 2011.

[20] S.-B. Scholz. Single assignment C – efficient support for high-level array operations in a functional setting, 2003.

[21] A. Sinkarovs and S.-B. Scholz. Data layout inference for code vectorisation. In *HPCS*, pages 527–534. IEEE, 2013.

[22] D. G. Spampinato and M. Püschel. A basic linear algebra compiler. In *International Symposium on Code Generation and Optimization (CGO)*, 2014.

[23] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.

[24] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and reuse with compiled domain-specific languages. In G. Castagna, editor, *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2013.

[25] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[26] V. Ureche, T. Rompf, A. K. Sujeeth, H. Chafi, and M. Odersky. StagedSAC: a case study in performance-oriented dsl development. In O. Kiselyov and S. Thompson, editors, *PEPM*, pages 73–82. ACM, 2012.

[27] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.