2018 International Conference on Field-Programmable Logic and Applications

A DSL-Based FFT Hardware Generator in Scala

François Serre
Department of Computer Science
ETH Zurich
serref@inf.ethz.ch

Markus Püschel
Department of Computer Science
ETH Zurich
pueschel@inf.ethz.ch

***Abstract*—We present a generator for fast Fourier transforms (FFTs) on hardware. The input of the generator is a high-level description of an FFT algorithm; the output is a token-based, synchronized design in the form of RTL-Verilog. Building on prior work, the generator uses several layers of domain-specific languages (DSLs) to represent and optimize at different levels of abstraction to produce a RAM- and area-efficient hardware implementation. Two of these layers and DSLs are novel. The first one allows the use and domain-specific optimization of state-of-the-art streaming permutations. The second DSL enables the automatic pipelining of a streaming hardware dataflow and the synchronization of its data-independent control signals. The generator including the DSLs are implemented in Scala, leveraging its type system, and uses concepts from lightweight modular staging (LMS) to handle the constraints of streaming hardware. Particularly, these concepts offer genericity over hardware number representation, including seamlessly switching between fixed-point arithmetic and FloPoCo generated IEEE floating-point operators, while ensuring type-safety. We show benchmarks of generated FFTs that outperform prior FFT generators.**

***Index Terms*—Fast Fourier transform; IP core; Streaming datapaths; Hardware generation; Scala**

I. Introduction

Due to its ubiquity in signal processing and communications, much effort has been devoted to the efficient implementation of fast Fourier transforms (FFTs) in hardware [1]–[10]. In particular, [10] proposes a generator capable of producing implementations with different trade-offs in terms of performance and resource consumption. This generator is built as a back-end of *Spiral*, a generator of signal processing libraries tuned for a specific platform [11], and operates with different FFTs represented in a domain specific language (DSL) called SPL. It then exploits different symmetries (or regularities) of these algorithms to *fold* them temporally (*iterative reuse*, see Fig. 1(b)) and spatially (*streaming reuse*, see Fig 1(c)), to obtain a space of relevant designs. The desired design is then output as RTL-Verilog.

In the meantime, the state of the art of the different components needed in the FFT has improved. As examples, FloPoCo [12] provides an open-source generator for pipelined floating-point arithmetic with arbitrary precision, streaming implementations of linear permutations have reached optimality in terms of latency, routing complexity and RAM bank usage [13], [14], and a new architecture (Fig. 2) further reduces RAM usage in some cases by fusing permutations [15]. However, no generator to this date appears to combine these features with the flexibility offered by [7]. One possible cause is the difficulty of programming a generator capable of mapping a high-level design (as in Figs. 1 and 2) to a concrete RTL implementation. Some of the challenges are discussed next.

Mismatch of hardware and software datatypes. A first difficulty, common among HLS tools, comes from the wide diversity of datatypes that hardware design offers. The precision of (unsigned or signed) integers or fixed point numbers is arbitrary, in contrast to a small set of choices in software. The same applies to floating-point arithmetic, ranging from IEEE754 to the space covered by FloPoCo [12] internal representations, each with variable mantissa and exponent width.

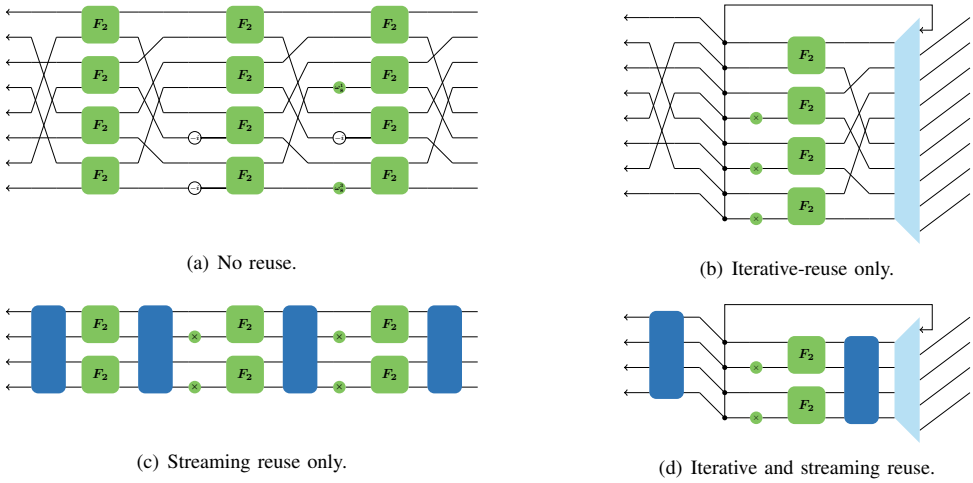
Two different evaluation times. A second issue is that a given function may need to be either evaluated during design generation or implemented in the resulting design, or even partially evaluated during generation and partially implemented.

For instance, the FFT involves multiplications with a set of constants, called *twiddle factors*. A twiddle factor $t_{i,j}$ is a complex number that depends on two parameters: the index i of the element, and the number of the computation stage j. In the case of non-iterative designs (Figs. 1(a) and 1(c)), the parameter j is known at generation time, while in iterative scenarios (Figs. 1(b) and 1(d)), the design would need to implement a *counter* counting the number of datasets that were already processed by the stage. Similarly, the parameter i is known at generation-time for non-streaming designs (Figs. 1(a) and 1(b)) for each different multiplier, while in streaming designs (Figs. 1(c) and 1(d)), i depends on the multiplier position, and on a *timer* that counts the number of cycles elapsed since the dataset began to enter. As the computation of a twiddle factor would typically involve a ROM containing different possible values, it is essential to exploit during generation as much as possible the structure of i and j to reduce ROM consumption and DSP slices in case of trivial multiplications.

A typical solution for handling this problem consists of writing and maintaining different versions for each different scenario, which is error-prone and time consuming.

Synchronization issues. The design requires pipelining to handle the frequency required by the user. Keeping the example of twiddle factors, an inspection of different FFT algorithms shows that most constants are 1, i or $-$i, which results in a trivial multiplication that does not require pipelin-

978-1-5386-8517-4/18/$31.00 ©2018 IEEE
DOI 10.1109/FPL.2018.00060



(a) No reuse.

(b) Iterative-reuse only.

(c) Streaming reuse only.

(d) Iterative and streaming reuse.

Fig. 1. Radix-2 Pease-FFT datapaths (each from right to left) operating on $2^n = 8$ elements with different types of folding [7]. In (a), the design is not folded and consists of 3 stages (each comprising a perfect-shuffle permutation, an array of butterflies F_2 and an element-wise multiplication by constants), followed by a bit-reversal permutation. (b) is horizontally folded: it implements only one instance of this stage that processes the dataset iteratively. (c) is vertically folded: the dataset is input *streamed* in chunks of $2^k = 4$ elements (the *streaming width*) that enter during $2^t = 2$ consecutive cycles. (d) combines both types of folding.

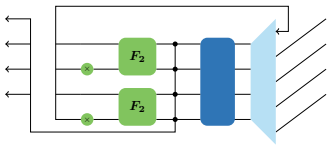


Fig. 2. Radix-2 Pease-FFT dataflow operating with fused permutations.

ing. However, it is necessary in this case to add supplementary registers if another non-trivial multiplication exists, to keep the whole dataset synchronized.

Additionally, if the twiddle factor computation is done in hardware, it may also require pipelining. As this computation is independent of the input to the FFT, it is possible to initiate it in advance to avoid impacting the global latency of the design. However, this requires a precise cycle tracking to trigger the counter and the timer at the appropriate time.

Handling the latency. As some of the designs produced use a loop (Figs.1(b), 1(d), and 2), special attention must be paid to guarantee that the latency of the inner structure is long enough to avoid collision between the tail and the head of a given dataset. Additionally, this inner latency determines the minimal time separating two datasets, which must be reported to the user.

Contributions. We address the above problems by significantly extending the FFT generator presented in [15] using a more principled design. We achieve superior results compared to prior work. Specifically:

- We present a hardware generator for a design space of FFTs. This generator is implemented in Scala [16] and leverages Scala's facilities for embedding DSLs, concepts from lightweight modular staging (LMS) [17] to perform optimization at the DSL levels, and Scala's type system to offer the flexibility discussed above.
- In the generator we use two novel DSLs to facilitate streaming optimizations.
- We benchmark against the prior FFT generator showing improvements in the performance/resource trade-off.

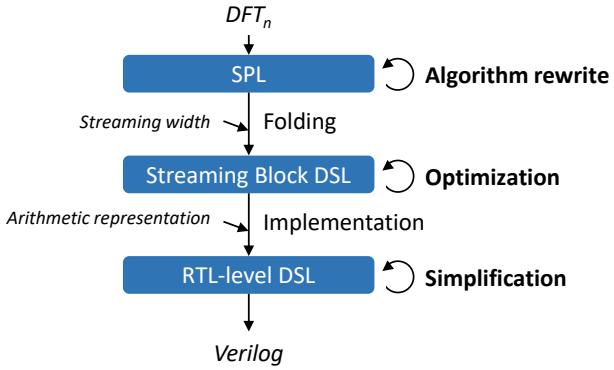


Fig. 3. The different layers of our generator.

II. Generation Pipeline

Our proposed generator receives as input the desired transform size (a power of two), and some parameters (size, streaming width, iterative reuse applied or not, hardware arithmetic representation), and outputs the corresponding design in the form of RTL Verilog. The generation consists of three layers pictured in Fig. 3. Each of these layers employs a DSL to represent, manipulate, and optimize the FFT at different levels of abstraction. Each DSL is implemented as embedded DSL inside Scala, and staging is used to allow manipulation. We first give a brief overview and then discuss one layer in greater detail in a subsequent section.

A. SPL

The first step for implementing a DFT in hardware consists of choosing a suitable algorithm, i.e., FFT, and follows [7]. We represent FFTs as *breakdown rules* that decompose a DFT_{2^n} into smaller DFTs. These rules are represented in SPL, a mathematical language that provides basic matrices and operators on these matrices.

In our generator two such rules are used: the constant-geometry radix-2^r Pease FFT [18] used for iterative reuse, and the radix-2^r iterative Cooley-Tukey FFT used for designs with streaming reuse only, and for the generation of the base case 2^r-FFT. As an example, the SPL representation of the Pease FFT used in Fig. 1 is

$$\mathrm{DFT}_{2^n} = R_{2^n} \prod_{j=0}^{n-1} \left(T_{n-j-1} \cdot (I_{2^{n-1}} \otimes \mathrm{DFT}_2) \cdot L_{2^n}\right). \quad (1)$$

In this expression, L_{2^n} and R_{2^n} are permutations (the perfect-shuffle and the bit-reversal, respectively), T_j is a diagonal matrix that performs element-wise complex multiplications with the twiddle-factors, and $I_{2^{n-1}} \otimes \mathrm{DFT}_2$ represents 2^{n-1} parallel butterflies (each an addition and a subtraction).

Our implementation of this DSL in Scala is similar as in [19].

B. Streaming-block DSL

In the second step, the SPL expression is formally folded according to the *streaming width*, i.e., the number of elements of the dataset that the design would be able to handle in each cycle. This includes inserting the necessary datapaths for the streaming permutations from [13], [15]. The DSL used thus expands SPL to include the streaming width (similar to the so-called Hardware-SPL in [7]), but also the following needed streaming blocks:

- single array of switches,
- double array of switches,
- temporal streaming permutation, and
- array of multiplexers.

Fig. 4(a) illustrates the design of Fig. 2 expressed with this DSL.

During this stage, a set of rewriting rules is used to simplify the streaming blocks, particularly in the case of a fused permutation. Fig. 4(b) shows the result of these optimizations on our example.

C. Streaming-RTL DSL

In the final stage, the streaming blocks are transformed into a dependency graph where each node, called a *signal*, represents a hardware operator that outputs one value per cycle. A signal may have zero (constant signals, inputs, timers and counters), one (flip/flop registers used for pipelining), or more parent signals. In the case of streaming reuse, this graph may contain loops.

The graph is constructed and represented using a *Streaming-RTL* DSL. Hardware datatypes, pipelining decisions and synchronization issues are mostly abstracted from this language. As an example, the implementation of the streaming block for the twiddles T_j can be written within a few lines, and works for every folding scenario and hardware datatype:

```
def T(inputs: Vector[Sig[Complex[Double]]], j: Sig[Int])
  (implicit dt: HW[Complex[Double]]) = {
  // We first declare a timer
  // that ticks for the duration of a dataset
  val timer = Timer(1 << t)

  // we define a (non-staged) Vector containing
  // all 2^n th roots of unity
  val rootsOfUnity = Vector.tabulate(1 << n){i =>
    val angle = -2 * Math.Pi * i / (1 << n)
    Complex(Math.cos(angle), Math.sin(angle))}

  // For each input signal,
  inputs.zipWithIndex.map{case (input, p) =>
    // we construct a signal corresponding to the index
    // of a given element (concatenation of the t bits
    // of the timer, and the k bits of the current port p),
    val i = timer ++ p(Unsigned(k))

    // we compute the corresponding twiddle factor,
    val address = (i & 1) * ((i >>> (j + 1)) << j)
    val twiddle = rootsOfUnity(address)

    // and we return the product of the input signal
    // with this twiddle factor
    input * twiddle
} }
```

As can be seen, only a few elements in the body of this function (`Timer`, `Unsigned`) may indicate that this code represents a low-level hardware architecture. This improves its readability and therefore its maintainability. However, all signals implicitly carry an underlying hardware type (including the corresponding size in bits), and timing information. All operations are bit- and cycle-accurate, and software and hardware type-safety is ensured. This DSL and its implementation are detailed in the next section.

Once constructed and optimized, the resulting graph is translated to a Verilog file.

III. A DSL FOR "STREAMING-RTL"

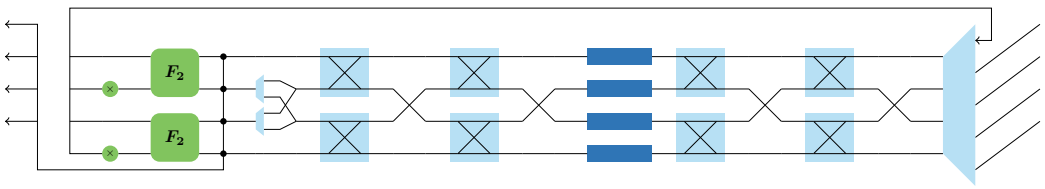
The streaming-RTL DSL is used to construct from a streaming-block level representation of an FFT algorithm a dependency graph that represents the final circuit. It offers the following features:

- The nodes (*signals*) of the graph are manipulated exactly as the values they would represent in a regular Scala program. Only their type changes.
- The language provides genericity over the actual hardware datatype and precision. However, the datatype can be made explicit, offering bit-accurate control.
- Pipelining and synchronization of data-independent control is performed implicitly, but timing information and manual pipelining remains available.

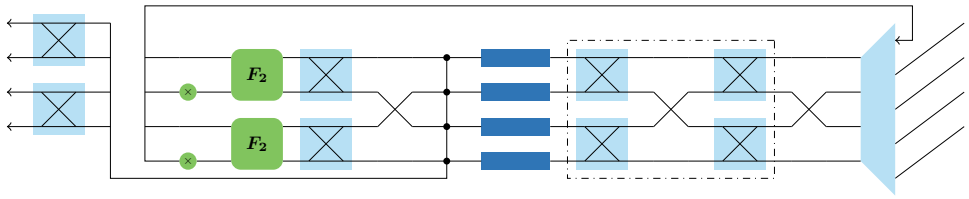
We discuss next the implementation of these abstractions, using the features offered by the Scala type system.

A. Staging and LMS

The implementation of our DSL uses the concept of *staging*, in particular as done in LMS [17], but using our own implementation. Staging allows to distinguish those parts of the computation to be evaluated at generation time and those



(a) Before optimization.



(b) After optimization.

Fig. 4. Design of Fig. 2 expressed using the streaming block DSL. The necessary streaming permutation is expanded into switches and RAM banks (blue rectangles). The optimization here "unrolls" some parts of the permutation to remove an array of multiplexer. Additionally, two arrays of switches were grouped for later mapping to 4-to-1 multiplexers. These optimizations increase the throughput and reduce the area of the final design.

that will be implemented in hardware via a type annotation. Specifically, this is done by changing a type `T` to the type `Sig[T]`; the latter means that computations on this type will be delayed, and may become part of the hardware implementation.

For example, in the following code, evaluating `f1` yields the sum of its parameters augmented by 18. `f2`, however, returns an expression tree representing the computation on symbolic inputs.

```
def f1(x: Double, y: Double) = x + y + 18
def f2(x: Sig[Double], y: Sig[Double]) = x + y + 18
```

This tree can then be translated (*unparsed*) to RTL-Verilog, yielding an implementation of two adders (adding two signals and an immediate).

This behavior is obtained through the following class hierarchy (truncated for brevity):

```
abstract class Sig[T:HW]{
  val dt = implicitly[HW[T]]
  val delay: Option[Delay]}
case class Plus[T](lhs: Sig[T], rhs: Sig[T]) extends Sig[T]
case class Const[T:HW](value: T) extends Sig[T]
case class Register[T](input: Sig[T]) extends Sig[T]
...
```

The parent class `Sig[T]` takes as a type parameter the type `T` of the expression it represents, and offers (*lifts*) the same operators as a regular instance of `T` would. These lifted operators return the corresponding node in the hierarchy of `Sig[T]`.

B. Abstraction over hardware datatypes

Following the concept of *abstraction over data representation* from [19], instances of `Sig[T]` (*signals* of `T`) carry their underlying hardware datatype in the form of a typeclass `HW[T]`. Concrete hardware datatypes are instances of classes derived from `HW[T]`, and contain the size in bits of this datatype:

```
abstract sealed class HW[T](val size: Int)
case class Signed(_size: Int) extends HW[Int](_size)
case class Unsigned(_size: Int) extends HW[Int](_size)
case class FxP(integral: Int, fractional: Int)
  extends HW[Double](integral + fractional)
case class IEEE(wE: Int, wF: Int)
  extends HW[Double](wE + wF + 1)
case class FloPoCo(wE: Int, wF: Int)
  extends HW[Double](wF + wE + 3)
...
```

A Scala `Int` could therefore be represented as a signed or unsigned integer of a given size, and a Scala `Double` can be represented using a fixed-point representation, a FloPoCo[1] or an IEEE floating-point representation. This information is passed to children nodes, and is used for the implementation of the lifted operators and for the representation of constants in the generated code.

As an example, depending on the underlying hardware type of its parameters, the previous example `f2` would seamlessly

- use fixed-point adders and represent 18 as a fixed-point immediate, or
- use FloPoCo generated floating-point adders and represent 18 with the corresponding FloPoCo binary representation, or
- implement a conversion from an IEEE floating-point signal to a FloPoCo representation, implement the FloPoCo adder, and implement the conversion back to an IEEE representation.

C. Synchronization

Each signal has a *delay* field that represents the time needed for this signal to output a valid value. It is used to check if two operands are synchronized, and, if it is not the case, to suitably delay one of them using registers.

[1]The FloPoCo generator is called upon instantiation of the corresponding datatype class to generate the different arithmetic operators. The result of this generation is then parsed to extract the latency of these operators.

A delay consists of an integer representing a number of cycles, and a *timeline* indicating to which "reference frame" this delay belongs. This timeline can be

- the *primary* timeline, referring to the number of cycles elapsed since the inputs arrived in the module,
- a *loop* timeline, referring to the number of cycles elapsed since a dataset entered within a loop, or
- a *floating* timeline, used by data-independent signals awaiting to be "synchronized" with another timeline.

As an example, a `Register` would have the same delay as its input with a cycle number incremented by one, while an input signal would have a delay of 0 on the primary timeline.

Loop timelines. In the case of iterative reuse, the *streaming product* (the streaming block that creates the loop and the multiplexer in Figs. 1(b) and 1(d)) creates a new loop timeline, and implements its inner expression using this timeline. The corresponding latency is then measured using the maximal delay of the signals that are returned. This information is then used during a second implementation, where a contingent lack of latency is compensated by a FIFO, or an increase of latency of a potential inner temporal permutation. The streaming product then presents its outputs using the same timeline as its inputs, delayed accordingly.

Floating timelines. In our generator, all data-independent control signals rely on counters (that count the number of datasets that have passed) and on timers (that count the number of cycles elapsed since the beginning of the current dataset). To ensure that such control signals become available at the correct instant, each time a new counter or timer is declared, a corresponding floating timeline is created. All data-independent operations performed are then pipelined using this timeline. However, when a signal with a floating timeline and a signal with an external timeline need to be synchronized, a new *floating delay* node is inserted with the expected delay.

As an example, we consider the following function `f3`:

```
def f3(x: Sig[Int]) = {
  val t = Timer(8) + 3
  x ^ t}
```

This function creates a 3-bit timer, and adds the constant 3 to it. This operation implicitly adds a pipelining register, yielding a signal `t` with a delay of 1 on the floating timeline associated with the timer. The input signal `x` is then xored with `t`. As these two signals are associated with different timelines, a floating delay signal depending on `t` is created with the same `delay` member as `x`, and `f3` finally returns a signal representing a XOR of `x` and the floating delay signal.

After the graph construction, the floating timeline is synchronized with the other timeline such that all floating delays can be implemented using the minimal number of registers. In particular, this ensures that data-dependent signals never have to be uselessly delayed. In our example, the floating timeline is synchronized such that the floating delay is implemented with a direct assignment. Thus, a delay of one cycle on the floating timeline corresponds to the delay of `x`.

To prevent nodes of a floating timeline from being synchronized with different incompatible timelines, and to avoid circular dependencies between floating timelines, the first time a node of a floating timeline is synchronized with a node from another timeline, the floating timeline is marked as "being in translation" with this other timeline, and an error is thrown if a node is later synchronized with a third timeline. With this relation, when the graph is built, timelines form a set of trees, rooted by the primary and loop timelines. Floating timelines are then synchronized starting from the roots.

Synchronization tokens. When the graph is unparsed, token synchronization signals are generated to trigger the different counters and timers. Tokens for loop timelines are generated by "ORing" tokens of the primary timeline. As the maximal throughput of the design is known at this time, tokens of the primary timeline can be generated using consecutive resettable timers instead of a resettable shift-register.

In our previous example, the timer declared within `f3` receives its token one cycle before `x` becomes available, ensuring that `t` is computed at the right time.

D. Smart constructors

Lifted operators are provided using *implicit classes*, which make it possible to add a posteriori methods and operators to existing objects.

For instance, the following class provides a $+$ operator to any `Sig[T]`, when `T` is a numeric type:

```
implicit class NumericSig[T:Numeric](lhs: Sig[T]){
  implicit val dt = lhs.dt
  def +(rhs: T): Sig[T] = lhs + Const(rhs)
  def +(rhs: Sig[T]): Sig[T] = {
    ensure(rhs.dt == dt)
    (lhs, rhs).synch match {
      case (Const(x), Const(y)) => Const(x + y)
      case (Zero(), _) => rhs
      case (_, Zero()) => lhs
      case (lhs, rhs) => dt match {
        case _: FloPoCo => PlusFPC(lhs, rhs).register
        case _: IEEE => (lhs.toFPC + rhs.toFPC).toIEEE
        case _ => Plus(lhs, rhs).register
} } } }
```

Here, the operator first checks that the two operands have the same hardware type (ensuring type-safety). It then synchronizes them, and handles particular cases (if the two operands are constants, or if one of them is the constant zero). Finally, it creates a new Plus signal, according to the hardware datatype, and adds pipelining registers.

These *smart constructors* are responsible for major optimizations. As an example, the constructor of ROM signals (implemented by adding a new `apply` method on indexed sequences of `T`) checks every bit of the control signal, and returns a smaller ROM in the case where one of them is constant. Particularly, it would return a constant if the control signal is constant, thus guaranteeing an efficient implementation of the twiddle stage T_j, even in non-streaming or non-iterative cases.

IV. RESULTS

To validate the designs produced by our generator, we benchmarked them against the equivalent circuits generated with [7]. All designs operate on 32bits IEEE754 floating-points, and were synthesized using Vivado 2018.1, targeting a Virtex7 xc7vx1140 FPGA. The floating-point operators used

TABLE I
RESOURCES USED BY NON-ITERATIVE RADIX-2 FFTS WITH A STREAMING WIDTH OF 4.

Size	Frequency (MHz)		DSPs		Area (Slices)			Memory (BRAM tiles)		
	[7]	Proposed	[7]	Proposed	[7]	Proposed	Ratio	[7]	Proposed	Ratio
8	326	345	16	16	4136	4954	1.19×	12	12	1.00×
16	326	345	32	32	6226	7105	1.14×	18	16	0.88×
32	326	345	48	48	8134	9131	1.12×	24	20	0.83×
64	326	345	64	64	10188	11181	1.09×	30	24	0.80×
128	326	310	80	80	12206	13188	1.08×	36	30	0.83×
256	326	310	96	96	13814	14869	1.07×	42	36	0.85×
512	326	310	112	112	15511	17023	1.09×	48	42	0.87×
1024	326	310	128	128	17204	19083	1.10×	54	48	0.88×
2048	326	310	144	144	19088	21111	1.10×	72	51	0.70×
4096	326	310	160	160	21184	23561	1.11×	108	66	0.61×
8192	326	310	176	176	23327	24551	1.05×	174	114	0.65×
16384	326	310	192	192	25513	26575	1.04×	322	194	0.60×
32768	317	308	208	208	27792	29752	1.07×	610	367.5	0.60×

TABLE II
RESOURCES USED BY ITERATIVE RADIX-2 FFTS WITH A STREAMING WIDTH OF 4.

Size	Frequency (MHz)		DSPs		Area (Slices)			Memory (BRAM tiles)		
	[7]	Proposed	[7]	Proposed	[7]	Proposed	Ratio	[7]	Proposed	Ratio
8	326	346	16	16	2250	2351	1.04×	10	4	0.40×
16	326	346	16	16	2296	2387	1.04×	10	4	0.40×
32	326	346	16	16	2397	2511	1.04×	10	4	0.40×
64	326	310	16	16	2375	2327	0.98×	10	6	0.60×
128	326	310	16	16	2355	2355	1.00×	10	6	0.60×
256	326	346	16	16	2372	2561	1.08×	10	6	0.60×
512	326	346	16	16	2332	2523	1.08×	10	6	0.60×
1024	326	343	16	16	2356	2333	0.99×	10	9	0.90×
2048	326	346	16	16	2380	2533	1.06×	20	8	0.40×
4096	326	346	16	16	2746	2529	0.92×	40	14	0.35×
8192	326	346	16	16	2516	2620	1.04×	76	26	0.34×
16384	326	334	16	16	2682	2770	1.03×	160	51	0.31×
32768	326	346	16	16	3752	3112	0.82×	315	102	0.32×

TABLE III
RESOURCES USED BY ITERATIVE RADIX-4 FFTS WITH A STREAMING WIDTH OF 8.

Size	Frequency (MHz)		DSPs		Area (Slices)			Memory (BRAM tiles)		
	[7]	Proposed	[7]	Proposed	[7]	Proposed	Ratio	[7]	Proposed	Ratio
16	326	346	48	56	7093	7964	1.12×	22	8	0.36×
64	326	345	48	56	7167	8491	1.18×	22	8	0.36×
256	326	310	48	56	7498	8355	1.11×	22	14	0.63×
1024	326	345	48	56	7794	8712	1.11×	24	14	0.58×
4096	326	345	48	56	7778	8722	1.12×	64	27	0.42×
16384	326	345	48	56	8382	8535	1.01×	248	86.5	0.34×

in our designs were generated using FloPoCo 4.1.2, targeting a 700MHz Virtex6 platform.

Tables I, II and III show results after place-and-route for a variety of designs. Each line presents, for a given transform size, the maximal frequency obtained for our design and the corresponding design from [7], as well as resources used in terms of DSPs, logic slices and memory. A third column shows the ratio between the two designs for the last two measurements.

We observe that the number of BRAM tiles required drops significantly with our generator. This is a direct consequence of the streaming permutations being implemented with [13] and, in the case of iterative designs, with the technics described in [15]. Each of these methods theoretically allows to halve the memory used for the streaming permutations, but does not affect the number of RAM slices used as ROMs to store the twiddle factors.

On the other side, the logic area consumption slightly increases. The gain obtained using FloPoCo for the arithmetic part and the use of 4-input multiplexers is counter-balanced by the additional logic needed to implement memory conflict avoidance as described in [13].

The number of DSPs used by the two designs is the same, except for Table III because of the choice of another algorithm

for the base 8-FFT.

V. Limitations and related work

Hardware DSLs implemented in Scala. The DSL we propose is specifically crafted for the generation of streaming Fourier transforms on FPGA, and provides only the primitives and the amount of abstraction needed for this purpose. This differentiates it from lower level hardware description languages written in Scala. For instance, Chisel [20] can represent a much wider variety of hardware designs, but requires the pipelining registers to be manually added. Targeting dataflow hardware, DFiant [21] proposes a dependency-driven automatic pipelining similar to ours, but does not seem to support automatic synchronization of data-independent controls. It uses literal types to expose the hardware datatype and precision to the user, thus enforcing type safety at compile-time. In our case, the hardware datatype is abstracted (provided via a type class), and hardware type safety is only ensured at generation time. On the other hand, high-level synthesis tools [22], [23] would offer even higher abstractions, up to the dataset level, but would not allow the user to program at the port-level, thus making the implementation of our permutation streaming blocks difficult.

LMS [17] itself does not only provide staging, but offers a whole toolchain to implement and compile DSLs. Particularly, it grants automatic common subexpression elimination during the construction of the dependency graph. However, in our case, floating timelines reduce the efficiency of such an optimization during the graph construction, and our tests have shown that synthesis softwares such as Vivado already provide it, thus limiting the use of implementing it. LMS provides as well a facility to manipulate the generated graph, but as ours already includes timing information, these manipulations are limited to timing invariant ones (fusing ROMs that contain identical values for instance), for which a direct implementation is possible. The main optimizations in our graph are made during its generation, using smart constructors.

The pipeline proposed in [24] to generate matrix operations illustrates the capability of LMS to target hardware. It shares many similarities with ours, particularly its use of LMS and FloPoCo. However, a significant part of the final RTL design is outsourced to the external back-end LegUp [25].

Hardware generator for FFTs. Our generator only handles the generation of power of two sized-FFTs. This represents only a subset of the architectures that [7] can produce. Particularly, their generator is capable of handling non-power of two sizes, and a wider range of signal processing transforms.

SPL and Spiral have as well been implemented and enhanced in Haskell [26] to produce efficient FFT implementations in C. A VHDL back-end for this compiler is being developed.

VI. Conclusion

The overall theme in our work is the principled design of domain-specific hardware generators using state-of-the-art languages and language features. This paper followed this theme with the design and implementation of a generator for streaming FFTs inside Scala, using embedded DSLs and the concept of staging. Specifically, our generator employed a pipeline of three abstraction levels, corresponding to three levels of DSLs. Two of them, the streaming-block DSL and the streaming-RTL DSL are novel and were specifically designed to include state-of-the-art components and enable the transformations and optimizations needed in FFTs. The produced FFTs improve over prior work. The generator should be easily extendable to other DSP components related to FFTs. A web version of our generator is available at [27].

References

[1] D. Cohen, "Simplified control of FFT hardware," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, no. 6, pp. 577–579, 1976.

[2] P. Kumhom, J. R. Johnson, and P. Nagvajara, "Design, optimization, and implementation of a universal FFT processor," in *Proc. International ASIC/SOC Conference (ASIC)*, pp. 182–186, 2000.

[3] A. Cortés, I. Vélez, and J. F. Sevillano, "Radix r^kFFTs: Matricial representation and SDC/SDF pipeline implementation," *IEEE Transactions on Signal Processing*, vol. 57, no. 7, pp. 2824–2839, 2009.

[4] S. He and M. Torkelson, "A new approach to pipeline FFT processor," in *Proc. Parallel Processing Symposium (IPPS)*, pp. 766–770, 1996.

[5] B. Akin, F. Franchetti, and J. C. Hoe, "FFTs with near-optimal memory access through block data layouts: Algorithm, architecture and design automation," *Journal of Signal Processing Systems*, vol. 85, no. 1, pp. 67–82, 2015.

[6] B. G. Jo and M. H. Sunwoo, "New continuous-flow mixed-radix (CFMR) FFT processor using novel in-place strategy," *IEEE Transactions on Circuits and Systems I*, vol. 52, no. 5, pp. 911–919, 2005.

[7] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 2, pp. 15:1–15:33, 2012.

[8] M. Garrido, M. Á. Sánchez, M. L. López-Vallejo, and J. Grajal, "A 4096-point radix-4 memory-based FFT using DSP slices," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 25, no. 1, pp. 375–379, 2017.

[9] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Linear transforms: From math to efficient hardware," in *Workshop on High-Level Synthesis colocated with DAC*, 2008.

[10] G. Nordin, P. A. Milder, J. C. Hoe, and M. Püschel, "Automatic generation of customized discrete Fourier transform IPs," in *Proc. Design Automation Conference (DAC)*, pp. 471–474, 2005.

[11] 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"*, vol. 93, no. 2, pp. 232– 275, 2005.

[12] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[13] F. Serre, T. Holenstein, and M. Püschel, "Optimal circuits for streamed linear permutations using RAM," in *Proc. International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 215–223, 2016.

[14] T. Koehn and P. Athanas, "Arbitrary streaming permutations with minimum memory and latency," in *Proc. International Conference on Computer-Aided Design (ICCAD)*, pp. 1–6, 2016.

[15] F. Serre and M. Püschel, "Memory-efficient fast Fourier transform on streaming data by fusing permutations," in *Proc. International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 219–228, 2018.

[16] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima Inc, 2008.

[17] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs," *Commun. ACM*, vol. 55, pp. 121–130, June 2012.

[18] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *Journal of the ACM*, vol. 15, no. 2, pp. 252–264, 1968.

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

[20] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a Scala embedded language," in *Proc. Design Automation Conference (DAC)*, pp. 1216–1225, 2012.

[21] O. Port and Y. Etsion, "DFiant: A dataflow hardware description language," in *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, 2017.

[22] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun, "OptiML: an implicitly parallel domain-specific language for machine learning," in *Proc. International Conference on Machine Learning (ICML)*, pp. 609–616, 2011.

[23] N. George, H. Lee, D. Novo, M. Owaida, D. Andrews, K. Olukotun, and P. Ienne, "Automatic support for multi-module parallelism from computational patterns," in *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, 2015.

[24] N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne, "Making domain-specific hardware synthesis tools cost-efficient," in *Proc. International Conference on Field-Programmable Technology (FPT)*, pp. 120–127, Dec 2013.

[25] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proc. International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 33–36, 2011.

[26] G. Mainland and J. Johnson, "A Haskell compiler for signal transforms," in *Proc. International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pp. 219–232, 2017.

[27] F. Serre, "DFT and streamed linear permutation generator for hardware." https://github.com/fserre/sgen, 2018.