DSL-based modular IP core generators: Example FFT and related structures

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

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

I. Introduction

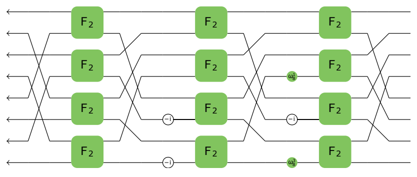
The algorithms for many important building blocks in embedded applications share a common network structure consisting of stages of small processing elements, separated by permutations. Examples include fast Fourier transforms (FFTs, see Fig 1a) and sorting networks (SNs, see Fig. 1c), built from butterflies and 2-input sorters, respectively. The symmetries in this structure can be used to *fold* these algorithms into compact designs that trade throughput performance for lower hardware resource usage [1] (see Fig. 1b). Extensive work has been performed on building generators that output an associated space of relevant designs in the form of RTL-Verilog [2], [3], [4], [5]. One key idea is the use of domain-specific languages (DSLs) to represent network structures [2], [3].

However, the flexibility needed in these generators remains a challenge. Besides different network variants and degrees of folding, various *hardware number representations* can be used, from fixed-point to floating-point arithmetic using FloPoCo [6] with variable precision width. Additionally, different methods exist to implement the streaming permutations needed in folded networks [7], [8], [9].

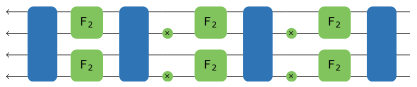
The generator we present here, originally proposed in [10], [11], uses a principled, modular design to offer this flexibility. It leverages features of the multi-paradigm language Scala and lightweight modular staging (LMS) [12] to implement different levels of DSLs and associated optimizations. Staging means delayed computation and is achieved through type constructors. If applied to a DSL, execution yields an expression tree that can be manipulated for optimization. Staging used in the implementation of the twiddle factors allows seamless integration of parts that are precomputed and parts that are computed in hardware. This division naturally occurs in various degrees due to the space of folding options.

II. Generation Pipeline

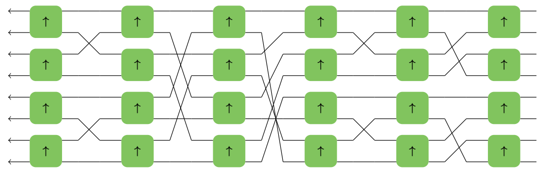
Our generator produces state-of-the-art hardware designs in a sequence of steps. It receives as input the desired computation and its size, along with parameters that define the desired type of folding, and outputs the corresponding design in the form of RTL Verilog. The generation consists of three layers pictured in Fig. 2. Each of these layers employs a DSL to represent and optimize the design at different levels of abstraction. Each DSL is implemented as embedded DSL inside Scala and staging enables optimizations as mentioned above. We discuss these layers next.



(a) FFT



(b) streaming FFT



(c) SN

Fig. 1: (a) Radix-2 Pease FFT [13], (b) the same FFT folded for *streaming* with 4 ports, and (c) Batcher bitonic sorting network [14]. All networks process from right to left.

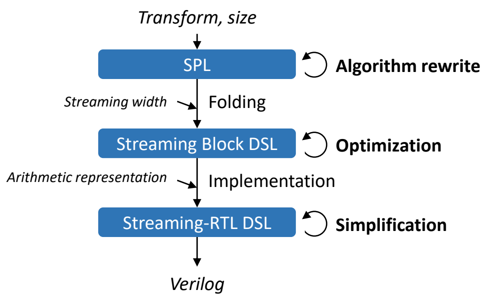


Fig. 2: The different layers of our generator.

III. SPL

The first step consists of choosing a suitable algorithm, i.e., network structure. The algorithm is represented in *SPL* as done in [2]. As an example, the radix-2 Pease FFT shown in Fig. 1a corresponds to the decomposition for $n = 3$.

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

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 inside Scala is similar as done in [15] for software.

IV. Streaming block DSL

In the second step of the generator, the SPL expression is formally folded (as in Fig. 1b). This includes inserting the necessary datapaths for the streaming permutations from [7], which uses stages of RAM banks, and stages of switching networks. The DSL used thus expands SPL to include the streaming width (similar to the so-called Hardware-SPL in [2]), but also the needed streaming permutation blocks (array of switches and memory block). During this stage, a set of rewriting rules is used to simplify the streaming blocks.

V. 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. The graph is constructed and represented using a *Streaming-RTL* DSL. This DSL offers the following features:

- The nodes 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.
- Signals that can be evaluated at generation time are automatically simplified during the generation (*staging*), thus sparing hardware resources.

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 number representation. The type constructor `Sig` controls the staging:

```
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, and timing information. In the case of a streaming design (Fig. 1b), a ROM is generated to store twiddle values, while a single constant would be used for unfolded designs (Fig. 1a). All operations are bit- and cycle-accurate, and software and hardware type-safety is ensured.

References

[1] 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.

[2] 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.

[3] M. Zuluaga, P. A. Milder, and M. Püschel, “Streaming sorting networks,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 21, no. 4, 2016.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] M. Garrido, “Multiplexer and memory-efficient circuits for parallel bit reversal,” *IEEE Transactions on Circuits and Systems II (TCAS-II)*, 2018.

[10] F. Serre and M. Püschel, “A DSL-based FFT hardware generator in Scala,” in *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pp. 315–322, 2018.

[11] F. Serre, “SGen - a streaming hardware generator.” https://acl.inf.ethz.ch/research/hardware/, 2018.

[12] 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.

[13] M. C. Pease, “The indirect binary n-cube microprocessor array,” *IEEE Transactions on Computers*, vol. 26, no. 5, pp. 458–473, 1977.

[14] K. E. Batcher, “Sorting networks and their applications,” in *Proc. Spring Joint Computer Conference*, vol. 32 of *AFIPS*, pp. 307–314, 1968.

[15] 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.