Enriching C-Based High-Level Synthesis with Parallel Pattern Templates

Lana Josipovic, Nithin George and Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences, 1015 Lausanne, Switzerland
{lana.josipovic, nithin.george, paolo.ienne}@epfl.ch

***Abstract*—Despite the popularity of C-based *High-Level Synthesis* (HLS) tools, their generic input programming languages make it challenging for the designer to find the expression that will result in adequate hardware quality and performance. Moreover, the syntactic variance of the input description often causes the inability of the HLS tool to fully identify and benefit from the properties of the computations. In this work, we propose extending standard C-based HLS tools with the concept of computational patterns. In particular, we present a template-based hardware generation strategy which enables complete exploitation of the pattern properties to produce high-quality hardware modules. The parametric templates allow us to automatically scale the implementation to the resource and data-bandwidth constraints of the target device, independent from the analysis abilities of the HLS tool. To demonstrate the benefits of our approach, we generated hardware implementations for six applications which we composed using a set of computational patterns (i.e. `map`, `zip` and `reduce`), achieving $1.3\times$ to $2.8\times$ speed-up over a state-of-the-art commercial HLS tool.**

I. Producing Quality Hardware is Hard

High-Level Synthesis (HLS) tools generate hardware designs from high-level programming languages, such as C, C++, SystemC, and OpenCL [10], [2], [1], and should liberate designers from the hardware details of *Register Transfer Level* (RTL) languages like VHDL and Verilog. Despite their progress, HLS tools tend to be criticized for the difficulty of extracting the desired level of performance without extensive experimentation with the tools. HLS designs require code refactoring and annotations, which are often platform- and tool-specific [3], [8]. A popular attempt to circumvent the difficulty of developing software for parallel platforms has been to rely on a limited number of frequently recurring parallel computational patterns which are common to different application domains. In this paper, we argue that computational patterns can and should be made first-class citizens of HLS tools based on imperative languages such as C. These tools have a large user community which could benefit from the pattern properties. Incrementally replacing C-code constructs with pattern representations would enable users to progressively reach the desired design performance, while keeping the code platform- and tool-independent. With that in mind, we explore whether extending C-based HLS tools with patterns using a template-based hardware generation strategy can lead to significantly better hardware implementations. We demonstrate how constructing and optimizing the patterns through parametric RTL templates leads to highly efficient hardware designs.

This research is partially supported by Intel Corp.

978-1-5090-5602-6/16/$31.00©2016 IEEE

II. Computational Patterns and Templates

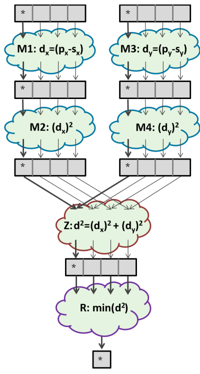
C-based HLS tools suffer from the following deficiencies: (a) Their input languages are generic programming languages that do not have sufficient expressiveness to capture the fine details of hardware designs. (b) The quality of the results depends on the syntactic variance in the input program and the analysis abilities of the tool. We argue that C-based HLS tools should be extended to support the notion of computational patterns, since they provide a powerful abstraction for programmers and are amenable to quality hardware architectures.

To illustrate, we consider the following examples: (1) Conveying associativity of a computation to a C-based HLS tool is not straightforward. The `reduce` pattern is by definition associative, which the HLS tool can exploit to achieve an efficient hardware implementation. (2) Unrolling is a standard HLS optimization. However, in loops with non-constant bounds, the tool often struggles to find the optimal level of parallelism. The computations within patterns such as `map`, `zip` and `reduce` are guaranteed to be side-effect free operations, which enables full exploitation of the available parallelism. The examples above demonstrate how patterns can be translated to good-quality hardware using a template-based design approach: (1) Defining a `reduce` as a binary reduction tree within a template, with all predetermined control logic and connections, removes the need for the HLS tool to search for the appropriate implementation. (2) Every `map` has a strict property of how data is consumed, with the ith input producing the ith output element. This information can be built into a template and directly exploited when parallelizing the computation.

III. Parallel Patterns for Hardware

A. Properties of Parallel Patterns

Computational patterns, such as `map`, `zip`, and `reduce`, operate on multi-element data structures. A `map` takes a collection of elements and applies a function (called *lambda*) to each element to produce a new collection. A `zip` operates on two data structures and uses a binary function to create a new structure. Since the lambda functions in `map` and `zip` are side-effect free, the individual function invocations for separate input elements are mutually independent and can perform in parallel. A `reduce` applies a binary function to a collection of elements and produces a single value. Since the function is



(a) Nearest neighbor.

Representation Using Computational Patterns	Sematically Equivalent Computation in C

`map(in, off,`
`    N, lmda, out);`
	`for (int i=0; i<N; i++){`
	`  int idx = off(i);`
	`  out[i] = lmda(in[idx]); }`

`zip(in1, off1,`
`    in2, off2,`
`    N, lmda, out);`
	`for (int i=0; i<N; i++){`
	`  int idx1 = off1(i);`
	`  int idx2 = off2(i);`
	`  out[i] = lmda(in1[idx1],`
	`                in2[idx2]); }`

`reduce(in, off, N,`
`       lmda, init, out);`
	`int accum=init;`
	`for (int i=0; i<N; i++){`
	`  int idx = off(i);`
	`  accum = lmda(accum, in[idx]);}`
	`out = accum;`

- Input Array: `in, in1, in2`
- Output Element/Array: `out`
- Element Count: N
- Lambda Function: `lmda()`
- Offset Computation Function: `off()`, `off1()` and `off2()`

(b) Parallel patterns in C.

```
#define N (10000)         // Size of points in set S
#define MAX_D (100000000) // Max distance (a very large value)
/** Defintion of the nearest neighbor function **/
float nearest_neighbor(float* s_x, float* s_y, float ptx, float pty) {
   float m1[N], m2[N], m3[N], m4[N], z[N], r;
   /** Define all lambda functions used in the core-computation **/
   // Function to compute the real offset index of each element
   auto off = [=](int index) -> int { return index; }
   // Lambda functions used inside the patterns
   float lmda_m1  [=](float x) -> float { return (x - ptx); }
   float lmda_m2  [=](float y) -> float { return (y - pty); }
   float lmda_m34 [=](float x) -> float { return (x * x); }
   float lmda_z [=](float x, float y) -> float { return (x+y); }
   float lmda_r [=](float x, float y) -> float { return ((x>y)?y:x); }
   /** Core-computation of the nearest neighbor **/
   // Compute squared distances along x and y axes using map
   map (s_x, off, N, lmda_m1,  m1);
   map (s_y, off, N, lmda_m2,  m2);
   map (m1,  off, N, lmda_m34, m3);
   map (m2,  off, N, lmda_m34, m4);
   // Compute the squared distances using zip
   zip (m3, off, m4, off, N, lmda_z, z);
   // Find the minimum squared distance using reduce
   reduce (z, off, N, lmda_r, MAX_D, r);
   // Return the minimum squared distance
   return r; }
```

(c) C-code with computational patterns.

Fig. 1: (a) The nearest neighbor application expressed as parallel patterns. (b) Computational patterns and their corresponding semantically equivalent computations in C. (c) The nearest neighbor application in C-code extended with the patterns.

side-effect free and associative, the operations can execute in parallel and form a reduction tree. To illustrate the use of the patterns, we consider the *Nearest neighbor* computation, which determines the distance of a point p to its nearest neighbor among the points in set S in a two-dimensional space:

$$d^2{}_{min} = \min_{\forall s \in S} \left((p_x - s_x)^2 + (p_y - s_y)^2 \right). \qquad (1)$$

We decompose the equation into four `map` operations to compute the squared distances along the axes, a `zip` which adds the separate results to compute d^2, and a `reduce` for finding the minimum squared distance, d^2_{min} (Figure 1a).

B. Templates for Parallel Patterns

In this work, we implement patterns as templates, which offers the following advantages: (1) Parametric RTL templates provide sufficient flexibility to deterministically leverage the structural properties of the patterns for optimizations. (2) A template-based construction allows us to automatically scale the implementation to the resource and data-bandwidth constraints of the target device, independently of the analysis abilities of the HLS tool. (3) Pattern-specific control within the templates can be specialized to an application by integrating specific computational instances into the predefined template.

C. Embedding Parallel Patterns into the C Language

Extending C-based HLS tools with computational patterns has three significant advantages: (1) C-based HLS tools have a large user base that can directly leverage this feature. (2) The pattern properties enable the user to efficiently express the nature of the computations. (3) The user can incrementally optimize the code by progressively replacing loop constructs with pattern representations, without compromising code portability. A natural way to integrate computational patterns into C-based HLS tools is to define a new library with declarations for patterns as regular functions (Figure 1b). The C++11 standard [5] already provides support for expressing lambda operations. Although not addressed in this work, we believe that future C-based HLS tools will enable users to write applications using the patterns (Figure 1c). Hence, we explore the remaining key question: whether extending HLS tools with computational patterns and using a template-based hardware generation strategy leads to significantly better hardware designs.

IV. Building and Optimizing Pattern Templates

A. Constructing the Templates

We can represent the iteration over a collection of elements as a loop construct containing three blocks: the input block reads the data for the computation from the memory, the computation block performs the operations, and the output block writes the results back to the memory. In the case of `map`, `zip` and `reduce`, the input elements are not modified by the computation or by the outputs generated within the loop. We can therefore transform the loop construct into a dataflow pipeline where the input block feeds the computation block which in turn feeds the output block.

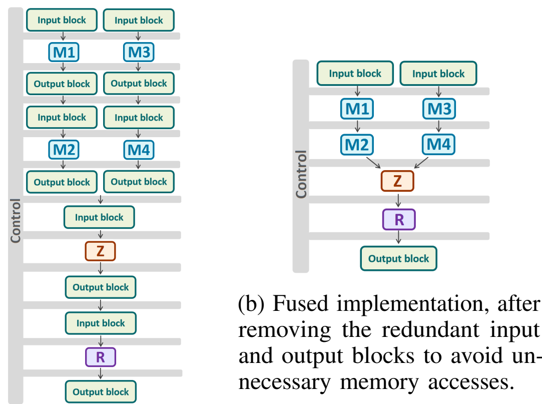
We implement the input and output blocks in RTL, since they can then be easily modified to accommodate different memory interface protocols. We use the ordinary capabilities of an HLS tool to produce the logic for computing the memory addresses of the input and output elements. The computational blocks need to be designed specifically for each application. Therefore, we use the core HLS tool to generate the hardware circuit for a single lambda function and integrate it into our template. Our templates contain predefined, pattern-specific control logic. The blocks use pairs of ready/valid signals to realize an elastic communication protocol, which enables them to function correctly even if their throughputs vary.

B. Optimizing the Templates

We leverage the properties of the patterns for optimizations:

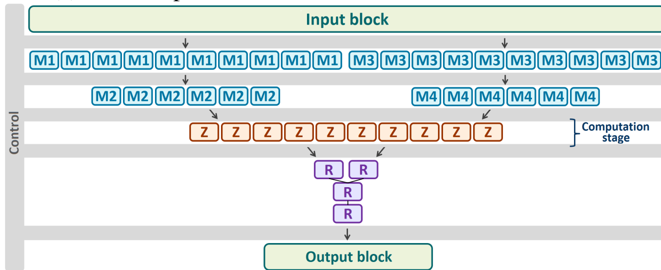
1) Loop Fusion: Loop fusion merges consecutive patterns when there is no need to store intermediate data into memory. To apply this optimization, we simply remove the output block and the adjacent input block. Since our templates use an elastic protocol, we do not need to change anything else. In the nearest neighbor application (Figure 2a), we can fuse the computations M1 to M4 into just two `map` patterns, which are then fused with the `zip` and the `reduce` (Figure 2b).

2) Parallelizing Computations (Unrolling): Our templates express the computation as a data flow streaming from memory to memory through a series of computing units; it is therefore easy to equalize the bandwidth at any section of this data flow. We use a simple concept of throughput-matching across the stages: (1) Determining the effective throughput based on the



(a) Basic implementation.

(b) Fused implementation, after removing the redundant input and output blocks to avoid unnecessary memory accesses.



(c) Unrolled implementation, after parallelizing each of the patterns to maintain the required data throughput in every computation stage.

Fig. 2: Optimizing the nearest neighbor application.

input and output data rates. (2) Determining the throughput of an individual computational block in every stage. (3) Increasing the parallelism in each stage separately to match the effective throughput. (4) Connecting across differently parallelized stages with parameterizable connection units in RTL. The parallelized nearest neighbor application is shown in Figure 2c.

3) Pipelined Parallelism: Our approach creates a design that is pipelined across the different computational units, conceptually similar to what Prabhakar et al. [7] achieve by means they refer to as metapipelining. The individual computational blocks we use are not pipelined (i.e. we use the naive implementations produced by the HLS tool, without applying the pipelining optimization). In future work, we plan to introduce pipelined blocks to lower the number of parallel instances needed to achieve the same throughput.

V. Results and Discussion

We compare our designs with those produced using Vivado HLS [10]. To find the best design points that Vivado HLS can produce, we applied different optimizations, such as loop pipelining, loop unrolling and function inlining. We manually tuned and refactored the C-code to maximize performance.

Our template-based hardware generation strategy consists of the following steps: (1) Creating computational blocks corresponding to lambda functions using the HLS tool. (2) Determining pattern fusion based on pattern interconnections and dependencies. (3) Determining the unroll factor for each pattern based on the input/output data rates. (4) Creating the parameterized RTL template with the input and output blocks, connecting logic and appropriate elastic control. (5) Integrating computational blocks into the templates to obtain the complete hardware module. In the current implementation, we give the lambda functions manually as inputs to our experimental tool. This is a simple parsing problem which can be trivially added to the tools features. Connections between the patterns and input/output data rates need to be defined by the user. This information can be easily obtained from the intermediate representation of the HLS tool and we plan to include its automatic extraction in future work. We automatically scale the templates to match the throughput requirement in every computational stage and integrate the computational blocks from Vivado HLS to create the complete hardware module.

We study applications operating on single-precision, floating point data. We use RAMs as a memory interface and apply our template-based construction approach to the inner-most level of nested-pattern designs. Placing and routing in Vivado gives us the resource usage. We obtain the execution time from the design simulation and the post-routing timing analysis.

4) Pearson correlation coefficient: A `reduce` calculates the mean values. Subtraction and multiplication (`map`) results are reduced into the coefficient numerator and denominator.

5) Nearest neighbor: We have used this application as an example throughout the paper.

6) Unity-based normalization: A `reduce` finds the minimum and maximum of the set. A `map` normalizes the values.

7) Circular discrete convolution: An addition (`reduce`) on a point-wise multiplication (`zip`) of the input datasets.

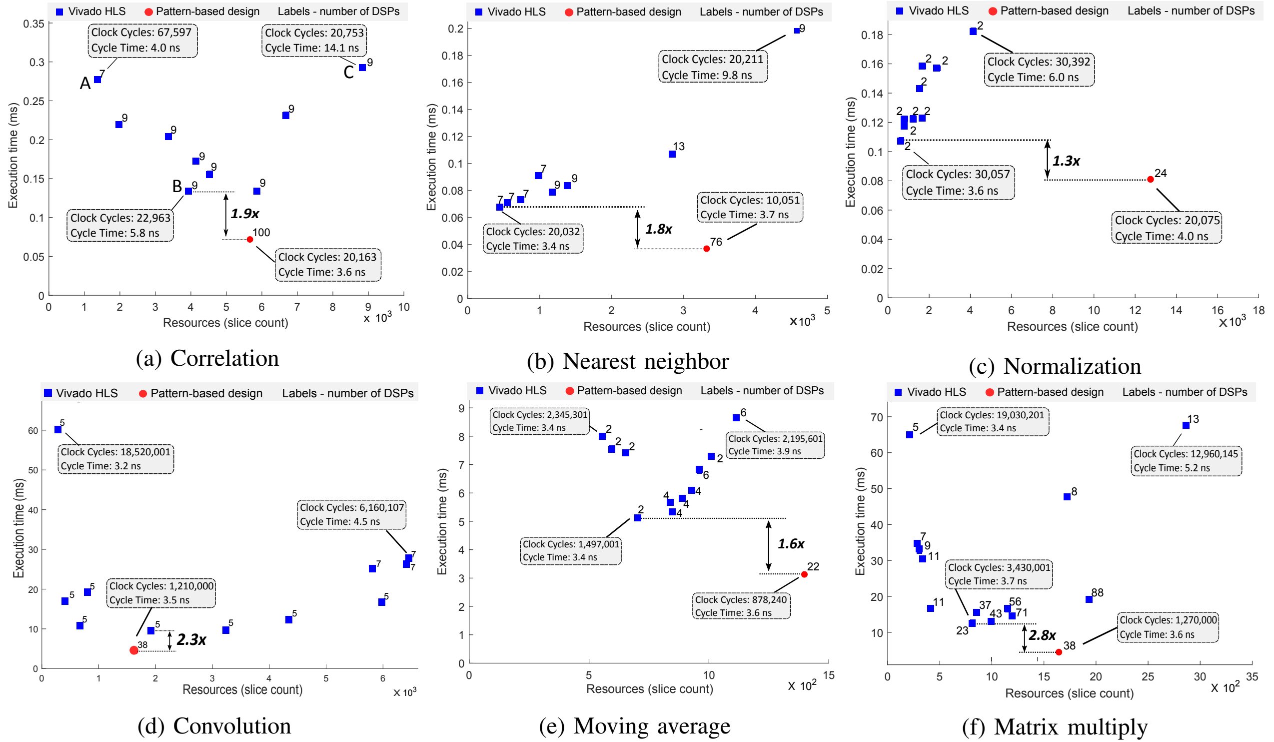
8) Simple moving average: Calculating a series of averages of different subsets of the input dataset (`reduce`).

9) Matrix multiply: Multiplication of the row and column elements (`zip`) and their reduction into one value (`reduce`).

When code refactoring and optimization directives improve the parallelism in the computation, the execution time of the Vivado designs reduces (Figure 3a, point A to B). As the designs become more parallel and larger, the input/output data rates constrain the performance by increasing the clock period (Figure 3a, point B to C). For every application, our template-based approach achieves higher performance than any of the Vivado HLS design points. This is not due to the implementation quality of the computation units themselves—the units within our templates are generated using Vivado HLS as well. Instead, we better leverage the properties of the computational patterns to appropriately parallelize and interconnect these units. The HLS tool fails to identify some of the pattern properties—e.g., the data type prevents it from exploiting the associativity which the `reduce` allows us to assume. This prevents it from fully parallelizing the computations. Our approach enables us to fully exploit the parallelism to achieve higher performance. Although this increases the resource usage, we uncover Pareto design points that were not possible using the HLS tool and achieve performance improvements ranging between $1.3\times$ and $2.8\times$. As noted in Section IV-B3, our resource usage could be further reduced by supporting pipelined computational blocks.

VI. Related Work

Tools such as FCUDA [6], OpenCL-to-FPGA [1] and SDAccel [9] use parallel variants of C, however, they depend on extensive compiler analysis and the quality of the generated



(a) Correlation (b) Nearest neighbor (c) Normalization

(d) Convolution (e) Moving average (f) Matrix multiply

Fig. 3: Comparison of the execution time and resource usage of the design space exploration using Vivado HLS with the design obtained with our approach. Note that all pattern-based designs are Pareto optimal in time and number of slices. The larger use of DSP units, if significant in relative terms, has in most cases a very limited impact on total silicon real-estate used.

hardware design will depend on how the input program is expressed syntactically [8], [3], [7]. Some parallel programming frameworks, such as OpenMP, enable the users to explicitly specify computations that are similar to `map`, `zip` and `reduce`. OpenMP uses pragmas which can enable HLS tools to correctly identify some patterns in the application. Once identified, these HLS tools can leverage the template-based approach proposed in this work. Prior efforts have explored computational patterns to generate hardware. George et al. [3] efficiently implement high-level applications on FPGAs, but used existing HLS tools (i.e., Vivado HLS) to implement their patterns. We show that the resulting designs can be significantly improved using our strategy. Prabhakar et al. [7] showed that metapipelining and tiling optimizations can improve the quality of the hardware designs generated from patterns. They use a functional programming language as input to the HLS tool to sidestep the shortcomings of C-based languages in expressing hardware. Our goal is to include computational patterns into C-based HLS tools using a template-based design strategy. Matai et al. [4] have studied the idea of using composable parameterized templates. However, their templates are complete algorithmic building blocks such as sorters and histogram primitives. Our technique explores a different grain of decomposition such as `map` and `reduce`.

VII. Conclusion

In this work, we show that extending the capabilities of C-based HLS tools with computational patterns and using a template-based hardware construction strategy results in highly efficient hardware designs. We construct and optimize the designs through parametric RTL templates to deterministically and effortlessly leverage the structural properties of the patterns. We achieve high-quality, high-performance hardware implementations which indicate that our technique is a promising improvement of C-based HLS tools.

References

[1] Altera. Implementing FPGA design with the OpenCL standard. Technical report, Altera Corporation, Nov. 2013.

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *Trans. on Embedded Computing Systems*, 13(2):24:1–24:27, Sept. 2013.

[3] N. George, H. Lee, D. Novo, T. Rompf, K. Brown, A. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In *Proc. of the 24th Int. Conference on Field-Programmable Logic and Appl.*, pages 1–8, Munich, Sept. 2014.

[4] J. Matai, D. Lee, A. Althoff, and R. Kastner. Composable, parameterizable templates for high level synthesis. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition*, Dresden, Mar. 2016.

[5] Microsoft. C/C++ Language and Standard Libraries, 2015. Online, accessed: 15-Jul-16.

[6] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. M. W. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Proc. of the 7th IEEE Symposium on Application Specific Processors*, pages 35–42, San Francisco, Calif., July 2009.

[7] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. D. Sa, C. Kozyrakis, and K. Olukotun. Generating configurable hardware from parallel patterns. In *Proc. of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, Atlanta, Apr. 2016.

[8] K. Rupnow, Y. Liang, Y. Li, and D. Chen. A study of high-level synthesis: Promises and challenges. In *Proceedings of the 9th IEEE International Conference on ASIC*, pages 1102–5, Oct. 2011.

[9] Xilinx. SDAccel Development Enviroment. Online, accessed: 31-Jan-16.

[10] Xilinx Inc. *Vivado High-Level Synthesis.*