High-Level Synthesis of Functional Patterns with Lift

Martin Kristien
University of Edinburgh
United Kingdom
m.kristien@sms.ed.ac.uk

Bruno Bodin
Yale-NUS
Singapore
bruno.bodin@yale-nus.edu.sg

Michel Steuwer
University of Glasgow
United Kingdom
michel.steuwer@glasgow.ac.uk

Christophe Dubach
University of Edinburgh
United Kingdom
christophe.dubach@ed.ac.uk

Abstract

High-level languages are commonly seen as a good fit to tackle the problem of performance portability across parallel architectures. The Lift framework is a recent approach which combines high-level, array-based programming abstractions, with a system of rewrite-rules that express algorithmic as well as low-level hardware optimizations. Lift has successfully demonstrated its ability to address the challenge of performance portability across multiple types of CPU and GPU devices by automatically generating code that is on-par with highly optimized hand-written code.

This paper demonstrates the potential of Lift for targeting FPGA-based platforms. It presents the design of new Lift parallel patterns operating on data streams, and describes the implementation of a Lift VHDL backend. This approach is evaluated on a Xilinx XC7Z010 FPGA using matrix multiplication, leading to a 10x speed-up over highly optimized CPU code and a commercial HLS tool. Furthermore, by considering the potential of design space exploration enabled by Lift, this work is a stepping stone towards automatically generated competitive code for FPGAs.

CCS Concepts • **Hardware → Hardware accelerators**; **Reconfigurable logic applications**; *Hardware-software codesign*; *Emerging languages and compilers*; • **Software and its engineering** → *Functional languages*; • **Computer systems organization** → Data flow architectures.

Keywords Lift, FPGA, data flows, design automation

ARRAY '19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6717-2/19/06…$15.00
https://doi.org/10.1145/3315454.3329957

ACM Reference Format:
Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. 2019. High-Level Synthesis of Functional Patterns with Lift. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3315454.3329957

1 Introduction

The last decade has seen the widespread adoption of novel parallel architectures such as GPUs (Graphics Processing Units), which have enabled exciting new applications such as deep learning. However, programming these devices is challenging, requiring knowledge about the low level details of the hardware to extract maximal performance. Furthermore, the optimization process is often platform-specific leading to non-performance-portable code.

Field Programmable Gate Arrays (FPGA) have been around for over three decades but have largely remained a niche market for very specific domains. However, we are now witnessing a large move towards the massive deployment of FPGAs in data centers from major industrial players [20]. Compared to GPUs, FPGAs have a higher energy efficiency and the ability to specialize the hardware makes them ideal for applications such as machine-learning.

Programming and optimizing FPGAs is, however, even more demanding than the already challenging task of programming GPUs. The lack of higher-level abstractions in Hardware Description Languages (HDLs) requires FPGA developers to possess extensive knowledge of computer architecture and hardware design. Several projects try to close this gap, such as the high-level LiquidMetal [4] approach or high-level synthesis tools based on a dialect of C (*e.g.*, HLS) or OpenCL. However, these approaches are usually still very specific to FPGAs, requiring programmers to write code in a certain contrived style in order to extract high performance.

Solving this programmability, optimization and performance portability challenge is critical for bringing FPGAs to the masses. We argue that the key to solving this issue lies in

the development of a high-level platform-agnostic programming language which can be compiled into efficient code. In particular, it should be possible to express an application at a purely algorithmic level in a declarative way without any need to understand and/or choose the target platform. This simplifies the development of applications but at the same time it is also important for achieving performance portability, pushing the responsibility for achieving high performance onto the compiler.

In this work, we propose to extend the existing Lift [22] compiler framework, with a backend targeting FPGAs. Lift is a novel approach to achieving performance portability in heterogeneous systems. Developers describe their applications using parallel patterns, such as *map* and *reduce*, in a high-level and platform-agnostic functional language. The Lift compiler then generates high-performance code for a selected architecture via a design space exploration driven by a set of rewrite rules expressing optimization choices.

The functional nature of the Lift language makes it easier to be mapped into hardware design, as individual composable functions can be directly translated to composable hardware modules. Functional programming avoids many common problems such as tracking of global state and mutable data and instead allows for representing applications purely as data flows. The Lift language is platform-agnostic and prior work [23, 24] has already demonstrated promising performance when targeting different GPU classes by exploring the design space using Lift's rewrite rules.

This paper focuses on high-level synthesis for FPGAs from an intermediate representation of Lift programs. The generated code consists of hardware description files to be synthesized and used on an FPGA and C code executed by the CPU, functioning as both the user interface and the driver of the FPGA logic. To summarize, our contributions are

- A set of new low-level patterns for Lift operating on data streams;
- a new Lift VHDL backend targeting FPGA-based systems and an associated runtime for the host;
- an evaluation of our work using matrix-multiplication comparing against optimized CPU code and a commercial HLS tool for FPGAs.

2 The Lift Framework

Figure 1 shows an overview of the Lift project [22]. Lift programs are written in a high-level language using a set of parallel patterns, such as *map* and *reduce*. This figure shows the dot product example in Lift: the two vectors x and y are pairwise combined using the *zip* pattern. The *map* pattern then applies a function (in this case multiplication) to each combined pair, effectively multiplying the two vectors. Finally, the result of the multiplication of x and y is summed up with the *reduce* pattern. The dot product is a small example program used to illustrate Lift. Much more complex and

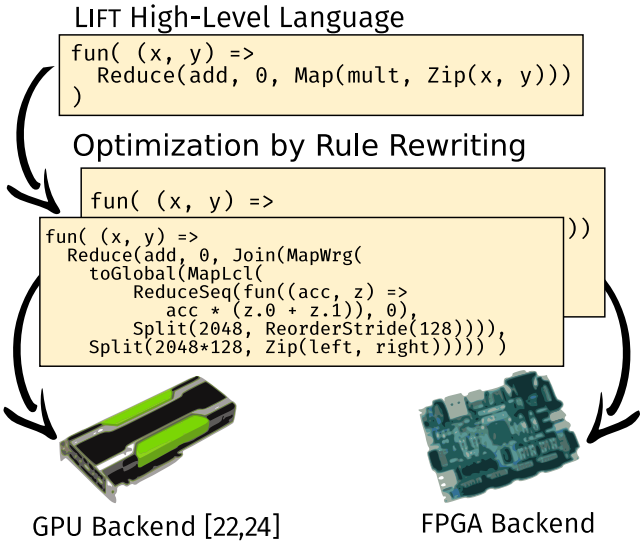


Figure 1. Lift compiler pipeline with dot product example optimized via rewrite rules and code generation targeting OpenCL. In this paper we extend Lift to target FPGAs.

interesting programs are expressible in Lift, including more complex linear algebra [23] and stencil computations [10, 24] which are both essential building blocks of convolutional neural networks.

This input program, which only describes *what* is computed but not *how*, is then transformed using a set of rewrite rules which encode implementation and optimization choices. The transformation process is either driven by heuristics or by an exploration process which applies rules more freely [23]. The result of the rewrite phase is a low-level Lift expression for which the decision of *how* to perform the computation has been made. Performance portability is achieved by applying a different sequence of rewrite rules targeting specific computer architectures.

In a final step OpenCL code (for CPUs and GPUs) or VHDL code (for FPGAs) is generated. This step is conceptually easy as all implementation decisions have been made, but still requires a careful compiler design [24].

In this paper, we introduce new low-level patterns in Lift which encode implementations and optimizations which are particularly good for FPGAs. We will introduce these patterns in the next section and describe how we generate VHDL code for them. In the following section 4 we discuss how we can introduce these FPGA-specific low-level patterns via a set of new rewrite rules.

Lift programs are described as `Lambda` function declarations. Since Lift is a functional language, the Lift compiler can decide on how to map semantics of programs to the target architecture. It does so by applying semantically consistent rewrite rules to high-level IR, in order to explore the design space. The rewrite rules can lower the abstraction of

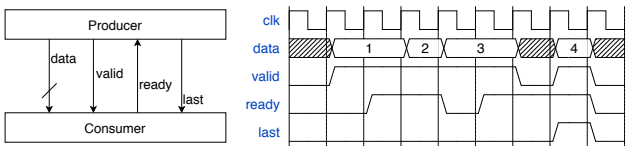


Figure 2. Streaming protocol used for data flow between producer and consumer. Timing diagram demonstrates transfer of a stream of four elements.

the IR, effectively deciding on what architectural primitives will participate in performing the computation described by the program.

In this paper we present new low-level patterns for the Lift framework and implement the last stage of the compiler pipeline, namely code generation, for these patterns targeting FPGA.

3 Lift Primitives for Hardware Synthesis

This section describes the FPGA-specific primitives, or patterns, that are introduced to Lift in order to generate FPGA-compatible code.

Two patterns operate at the hardware-software interface, namely *ToFPGA* and *ToHost*. These can be used to move data between the main memory and the FPGA. The data transfer at the FPGA boundary is viewed as a stream. Several new patterns are introduced that operate at the hardware level, this includes `MapStream`, `ZipStream`, `ReduceStream`, `LetStream` and `UserModule`.

Before introducing these patterns, we explain how we compose hardware modules with streams.

3.1 Hardware Composability with Streams

To enable composability of hardware modules produced by Lift, arrays from the high-level Lift IR are represented as data flows, i.e. `streams`. Using streams, hardware entities can be connected in a consumer-producer fashion. Each entity can hide its internal logic as long as the entity's boundary respects the streaming protocol described by Figure 2.

The streaming protocol used allows for bidirectional synchronization. Producer asserts the `valid` signal when the data bus contains a new valid element of the corresponding stream. Consumer can assert the `ready` signal when it is ready to consume the new data. The data transfer is considered successful for cycles when both `valid` and `ready` signals are asserted.

Producer can further indicate when the corresponding stream ends by asserting the `last` signal. This is useful for instance when dealing with multi-dimensional data structures such as matrices, represented as arrays of arrays at the Lift level. In such cases, the `last` signal will be composed of multiple bits, one for each dimension of the array (*e.g.*, one for the end of a row and one for the end of the matrix).

3.2 Types and Notation

Each pattern is described by giving its type signature and an informal description of its semantics. The types are important, as they determine which composition and nesting of the patterns are valid. For the type signature we are using the following notation:

- S_n - denotes a scalar type S of size n-bytes. Examples of scalar types are *int* and *float*.
- $[T]_n$ - denotes an array of n many elements of type T.
- (T, U) - denotes a two-elements tuple of type T and U.
- $T \rightarrow U$ - denotes a function mapping arguments of type T to a result of type U.

We use S exclusively for denoting scalar types, where T and U denote arbitrary data types.

When depicting hardware design schemes, entity boundaries are depicted using boxes with solid lines. Logic inside boxes with solid lines is encapsulated in terms of scope and can only access signals inside the box or at the box's interface. Global scope patterns are depicted using logical boundaries, represented by boxes with dashed lines. As opposed to boxes with solid lines, logic inside boxes with dashed lines can access signals outside of the box.

3.3 Low-level Patterns for FPGA

This section presents the low-level patterns that are added to Lift to support FPGA hardware synthesis. At the abstract level, all these patterns operate on arrays which are synthesized as streams, as explained earlier.

3.3.1 ToFPGA

The *ToFPGA* pattern moves data into the FPGA without modifying it. The pattern operates on an array of values of the same type:

$$ToFPGA : [S]_n \rightarrow [S]_n$$

When code generation occurs, this pattern triggers the generation of host code to manage the data transfer to the FPGA as well as FPGA hardware to manage receiving this data and turn it into a stream.

3.3.2 ToHost

The *ToHost* pattern performs the inverse operation and moves the data from the FPGA back into the host's main memory. It has, therefore, the same straightforward type:

$$ToHost : [S]_n \rightarrow [S]_n$$

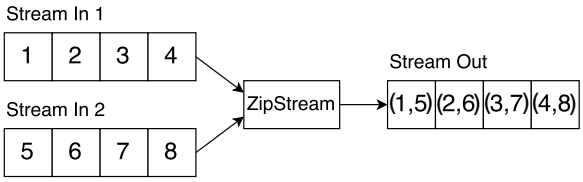
Similarly to the *ToFPGA* primitive, this pattern triggers the generation of host code to receive the data produced by the FPGA and generates FPGA hardware to handle the communication with the host. Furthermore, the responsibility of generating the input stream of this pattern is handed over to an FPGA generator.

3.3.3 ZipStream

The *ZipStream* pattern takes two arrays of the same length and base type as arguments and creates an array of pairs. The type of the pattern is:

$$ZipStream : ([T]_n,\ [T]_n) \rightarrow [(T, T)]_n$$

ZipStream does not only reshape the input data into tuples, it also synchronizes the consumption of the two inputs. This is achieved using the `ready` and `valid` valid signals from the two input components connected to the *ZipStream* module. The hardware design scheme of the pattern is shown below.

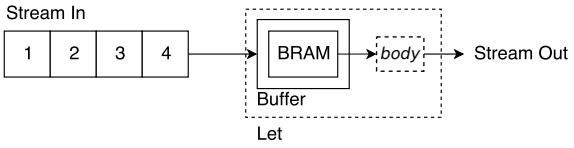


3.3.4 LetStream

The *LetStream* pattern takes a function as an argument which represents the *body* of the LetStream. LetStream maps its data input to the input parameter of the function, which should consume it. The type of *LetStream* is:

$$LetStream : ((T \rightarrow U),\ T) \rightarrow U$$

Crucially, the *LetStream* pattern allows the data input to be consumed multiple times by the body by caching it in a block RAM and streaming it whenever the data is requested until the cache is released. The cache can be released after the body of the *LetStream* pattern has produced the last result, as indicated by the generation of a `last` signal at the output from the *LetStream* pattern's body. This is the main mechanism used to achieve data reuse. LetStream usage will be demonstrated later in section 3.4.

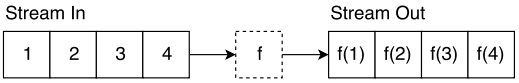


LetStream is implemented in global scope, as indicated by the dashed boundary. This is necessary, as the body is an arbitrary function which could potentially access other data in global scope.

3.3.5 MapStream

The *MapStream* pattern accepts a function declaration `f` as the first and an input array as the second parameter. Using the *MapStream* pattern results in application of the function `f` to all elements of the input array. The type signature of *MapStream* is:

$$MapStream : ((T \rightarrow U),\ [T]_n) \rightarrow [U]_n$$

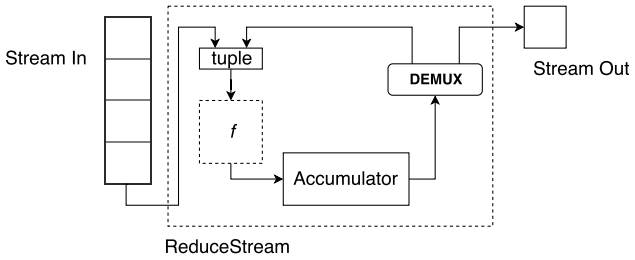


Note, like *LetStream* the *MapStream* pattern has only logical boundaries (i.e. it is not enclosed in a separate entity) to allow function `f` to be arbitrary, giving it access to signals in global scope.

3.3.6 ReduceStream

The *ReduceStream* pattern reduces a sequence of multiple elements into a sequence of one element. Reduction is performed by repeated application of a binary function f to all elements in the input sequence, storing the output in an accumulator, which is fed back into the function f as one of its two inputs. When the whole input sequence is consumed, *ReduceStream* outputs the value of accumulator. The type of *ReduceStream* is:

$$ReduceStream : (((T,\ U) \rightarrow U),\ U,\ [T]_n) \rightarrow [U]_1$$



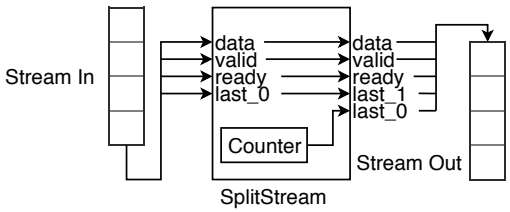
Similarly to *LetStream* and *MapStream*, *ReduceStream* has only logical boundaries to allow an arbitrary function `f` to be used.

3.3.7 SplitStream

The *SplitStream* pattern introduces new inner dimensions to a stream of data. This is necessary to represent multi-dimensional streams. Given the size of the inner dimension n and a stream, *SplitStream* generates a two-dimensional stream. The new stream has inner dimension of n and the total number of elements equal to the original one-dimensional stream. In case m does not divide n, the Lift compiler rejects the program during type checking. This can be done by multi-bit `last` signal, each indicating the end of stream for the corresponding dimension.

$$SplitStream : (n,\ [T]_m) \rightarrow [[T]_n]_{m/n}$$

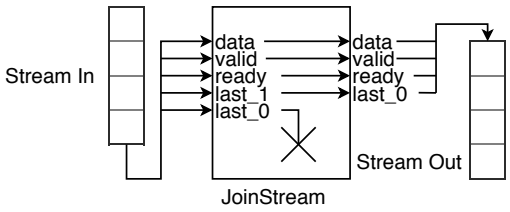
As can be seen in the figure below, the inner bit of the last signal is generated by a counter. This last inner signal is emitted when the counter reaches the length of the inner dimension of the output. This information is directly extracted from the pattern input n.



3.3.8 JoinStream

The *JoinStream* pattern is the opposite to the *SplitStream* pattern. It removes one dimension from a multi-dimensional stream by flattening (*i.e.*, merging) the two outermost dimensions. This is easily done by discarding the `last` signal from the inner dimension.

$$JoinStream : ([[T]_n]_m) \rightarrow [T]_{n*m}$$



3.3.9 UserModule

A *UserModule* has a user-defined signature and functionality.

$$UserModule : T \rightarrow U$$

It can only be implemented as a separate entity, encapsulating the logic of the user module. This pattern encapsulates custom simple modification of data that do not require any global context. Potential examples are arithmetic operations operating on integers or pairs of integers which can be defined by creating an `Adder` *UserModule*.

Addition **An addition** *UserModule* is defined as an entity accepting a pair of integers and producing their sum as a single integer.

Multiplication The multiplication of two 64-bits integers can incur a large delay. For the investigated designs, the direct multiplication of two integers could not be achieved for clock frequencies greater than 95MHz. This frequency limit results in sub-optimal performance. To resolve this issue, a new *UserModule* was developed which performs integer multiplication in two pipeline stages.

The multiplication is split into 4 partial multiplications and their results are added together for the final result, as shown in Figure 3.

Since the multiplication *UserModule* accepts two 64-bit integers and produces a 64-bit integer, the upper bits of the multiplication can be discarded. Therefore, the partial multiplication of `a*c` is not computed at all. The other three partial multiplications (i.e. `a*d`, `b*c`, and `b*d`) are computed in the first stage of the pipeline. In one clock cycle, these partial results are summed up to produce the final output. Using a pipelined version of integer multiplication produced correct results even for high clock frequencies of 175MHz on our hardware.

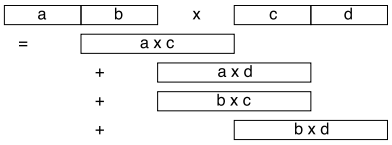


Figure 3. Multiplication expressed through partial results

3.4 Dot Product with Low-Level Patterns

Lift uses rewrite rules to describe the step-by-step process of transforming the original high-level expression of the dot product from Figure 1 into an FPGA-compatible low-level expression. The most basic way of performing this transformation is to wrap the expression by a `ToHost` operator and to wrap each input by a `ToFPGA` while replacing every operator by FPGA-compatible ones (i.e. `Map` into `MapStream`). The Dot product in Lift after the transformation to low-level FPGA-specific patterns is depicted in Listing 1.

Listing 1. Dot product in Lift Low-level for FPGA

```
1  add = fun(x => UserModule.Addition(x))
2  mul = fun(x => UserModule.Multiplication(x))
3  program = fun( (x, y) =>
4    ToHost(
5      LetStream(fun (z =>
6        ReduceStream(add, 0,
7          MapStream(mul, ZipStream(z, ToFPGA(y)))
8        ))
9        , ToFPGA(x)
10 ) ) )
```

As this program is a sequence of nested function calls it is read from right to left. The program starts by sending argument `x` to the FPGA (line 9), where it is stored using the *LetStream* pattern in line 5. The stored argument can be referred to in the body of `LetStream` as `z`. Then, second argument is sent to the FPGA (line 7), where it is zipped with the stored first argument `z`. Zipped arguments are pairwise multiplied and the resultant sequence is reduced using addition operation in line 6. At the end, the program sends the result back to the host using `ToHost` pattern.

The block diagram corresponding to the program in Listing 1 is depicted in Figure 4. In the example above, addition and multiplication operations are expressed using the *UserModule* pattern which wrap the custom modules described in the section above.

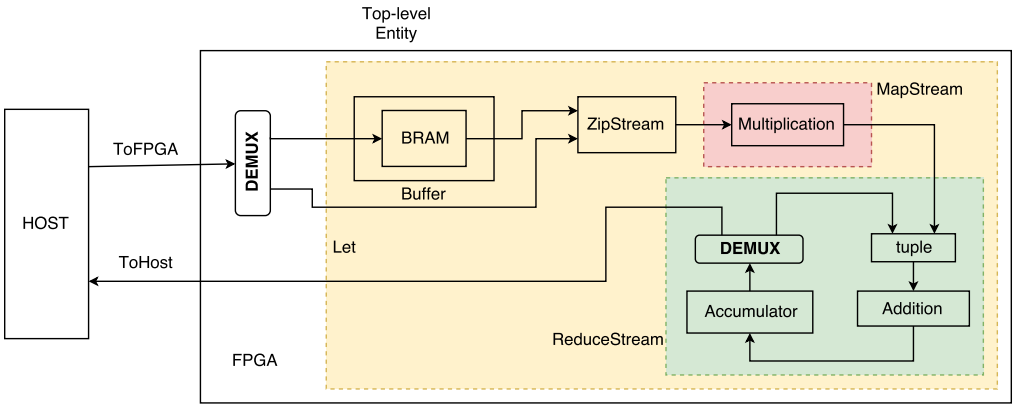


Figure 4. Hardware design of dot product.

Figure 4 demonstrates the composability of Lift patterns in hardware design. Aligned with the paradigm of functional programming, Lift makes use of a strong type system to ensures that the composition of patterns remains valid throughout the rewrite process inside the compiler. Figure 4 shows that the hardware design closely follows the data flow through the composition of patterns, resulting in a valid composition at the hardware design level as well. There is no need for an explicit control unit to orchestrate the execution of individual modules. All control is performed by individual entities through controlling the streaming protocol signals. This design results in fully distributed control based on interactions of producers and consumers in the data flow. Furthermore, since Lift patterns are mostly self-contained, they can be directly mapped to VHDL entities and composition of entities can be directly derived from composition of Lift patterns. This results in almost one-to-one mapping between Lift functions and structural VHDL code.

4 Expressing Optimizations in Lift

The previously defined low-level patterns are already sufficient to generate a valid program for FPGA. However, as the Lift framework is fully based on rule rewriting for applying performance optimization, we defined a set of rewrite rules beneficial for FPGAs. To reach the performance achieved in this paper, there are only three optimizations encoded as rewrite rules required: *tiling*, *vectorization*, and *coarse-grained parallelism*. These optimizations are beneficial for applications with a high level of data reuse. Since the dot product accesses each data element only once, it is not suitable for demonstration of these optimization rules. Therefore, we will consider the much more interesting matrix multiplication as the case study discussing optimizations.

4.1 Baseline

Listing 2 shows a naive implementation of matrix multiplication in Lift which essentially applies the dot product (in line 4) to each combination of row and column of the twoinput matrices.

Listing 2. Matrix Multiplication in Lift high-level patterns

```
1 fun( (A, B) =>
2   Map(fun(aRow =>
3     Map(fun(bCol =>
4       Reduce(add, 0, Map(mul, Zip(aRow, bCol)))
5     ), Transpose(B))
6   ), A))
```

The naive implementation of matrix multiplication can be directly compiled to hardware design depicted in Figure 5. The computation starts by sending one input matrix to the FPGA and storing it in a block ram module (1). Then, another input matrix is sent to the FPGA. The second matrix is split into rows and each row is stored in another block ram module (2) for reuse.

When one full matrix and one row of another matrix are stored on the FPGA, matrix-vector multiplication (`Let` pattern 3) can be performed. This is done by splitting the matrix into columns and mapping dot product operation (4) over the resultant stream of columns.

Since one matrix and one row of another matrix have to be stored on the FPGA, the amount of block memory available limits the size of the matrices that can be multiplied. This issue is addressed by our first optimization, *tiling*.

4.2 Tiling

Tiling is a common loop transformation that improves cache utilization in CPUs and GPUs. The principle of tiling is to subdivide the data to be processed inside a loop in tiles, and to individually process those tiles. This reduces the amount of data computed at once, and thus reduces the memory required on the FPGA. Such techniques are required with FPGAs to support matrix multiplication of large matrices.

In Lift, tiling is performed by splitting the input matrices along both dimensions, so that they are decomposed into

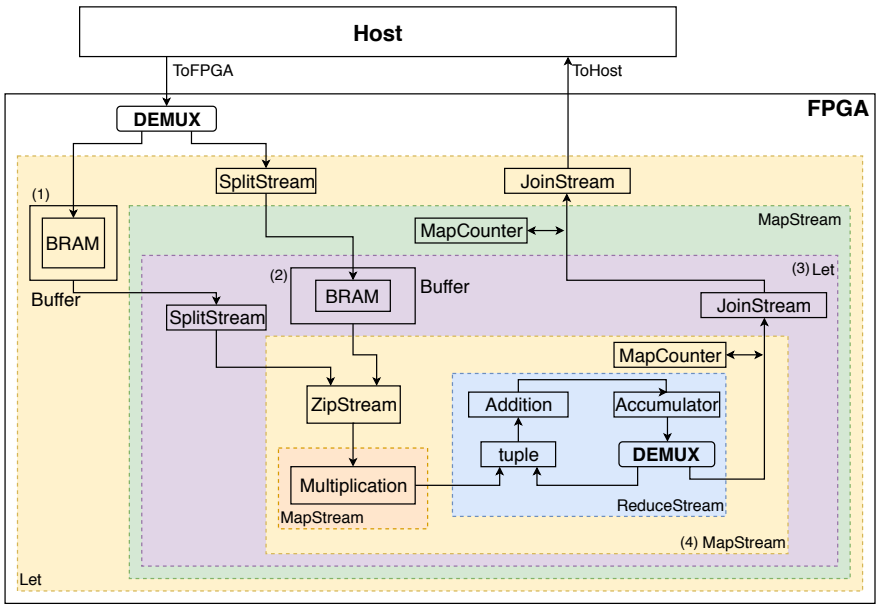


Figure 5. Hardware design of baseline matrix multiplication.

multiple memory blocks. Tiling is achieved by reusing the *Split* pattern using the following combinations:

```
1  Tile(m, n, matrix) =
2  Map(splitRows =>
3      Map(splitColumns =>
4          Transpose(splitColumns)
5      ,Split(n,Transpose(splitRows))
6  ,Split(m,matrix))
```

UnTile is built in a similar fashion using a series of *Map*, *Join* and *Transpose*.

A sequence of rewrite rules transforms the program in Listing 2 to express the tiling optimization [23]. We can reuse these existing rules which produce this tiled version of the matrix multiplication:

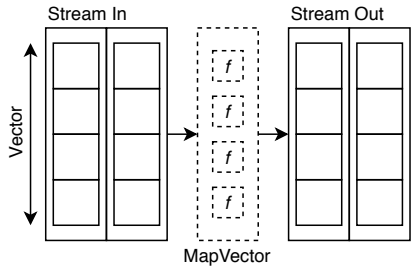
Listing 3. Matrix Multiplication after Tiling

```
1  fun( (A, B) =>
2   UnTile(Map(fun(rowOfTilesA =>
3    Map(fun(colOfTilesB =>
4     Reduce(fun(tileAcc, (tileA, tileB) =>
5      Map(Map(add), Zip(tileAcc,
6       Map(fun(aRow =>
7        Map(fun(bCol =>
8         Reduce(add, 0, Map(mul, Zip(aRow, bCol))))
9        , tileB)) , tileA)))
10    ), 0, Zip(rowOfTilesA, colOfTilesB))
11   ), Tile(m, k, Transpose(B)))
12  ), Tile(n, k, A))))
```

Here one can still see how the same matrix multiplication primitives are used in lines 6–9 to be applied to the tiles which have been created using the *Tile* primitive. The computed tile is then added to an accumulation tile using the outer *Reduce* primitive (line 4).

At a later stage of compilation this tiled matrix multiplication is transformed into a low-level implementation where only the matrix multiplication operating on the tiles is sent to the FPGA. The accumulation of computed tiles is done on the host.

4.3 Vectorization



Vectorization is a form of data parallelism. Vectorized data streams access multiple elements at the same clock cycle, whereas elements in sequential streams are only accessed one at a time. Multi-element access can speed up memory throughput as well as computational throughput. These two combined result in speedups almost linearly proportional to the vectorization factor.

In Lift, we added specific patterns that manipulate vectorized data streams. *StreamToVector* and *VectorToStream* convert data streams between scalar values and streams of vectors with a particular size. The *MapVector* and *ReduceVector*

patterns operate on vectors and behave similar to their counterparts operating on streams. Note that vectorization is limited by the available hardware as it requires more resources but results in better resource utilization.

The figure above demonstrates application of *MapVector* pattern to a stream of vectorized data with vectorization factor of 4. All elements of a vector are processed at the same clock cycle. To achieve parallel processing of vector elements, function f has to be duplicated.

4.4 Coarse-Grained Parallelism

In hardware, coarse-grained parallelism is expressed by replicating the same data-flows multiple times. In matrix multiplication when mapping the dot product function over the rows of the input matrix, we can exploit coarse-grained parallelism by performing the dot product for multiple rows in parallel. As opposed to vectorization, this form of parallelism requires duplication of streams.

In Lift, this optimization is applicable by rewriting the `MapStream` pattern to the `MapStreamPar` variant for which the backend generates several data flows in parallel. Since the input and output are sequential, the backend creates a demultiplexer at the beginning of the parallel logic block and a multiplexer at the end.

5 Evaluation

We start by presenting the target system and our evaluation methodology. The Lift FPGA backend performance is compared against existing solutions and we evaluate the impact of our optimizations.

5.1 System Setup and Implementation Details

The target system is a Xilinx's ZYBO board [28] which is the smallest board of the Zynq-7000 family. This board features a dual-core Cortex-A9 processor running at a clock frequency of 650 MHz and runs Linux. The core has access to 512MB of DDR3 memory with a bandwidth of 1050Mbps and 8 DMA (Direct Memory Access) ports. The FPGA comprises 240 KB Block RAM (i.e. PL memory) and 28,000 logic cells.

Applications are evaluated by measuring throughput (i.e. number of operations divided by execution time). The runtime is measured across the call to the driver's API specific to the application. The average of five measurements is reported. Standard deviation of the measurements was less than 5% of the measured value for all experiments, which is negligible. All the C programs are compiled using gcc 4.9 with -O3 optimizations.

The Lift compiler produces VHDL code for the FPGA side of the program. This also include an AXI DMA used to communicate with the FPGA. The Lift framework generates a C host program performing the computation that is not offloaded, and that uses a DMA driver to communicate with the DMA device on the FPGA.

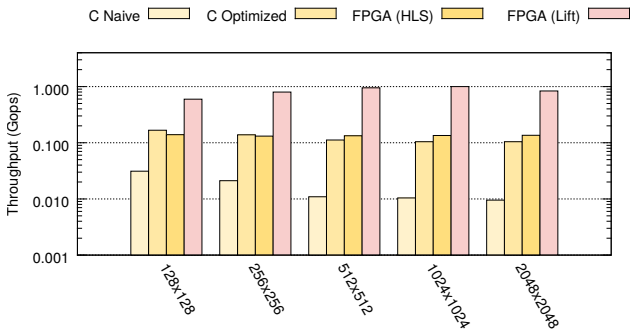


Figure 6. Throughput (in Giga operations per seconds) of integer 64 bits matrix multiplication for different matrix sizes.

5.2 Comparison with High-Level Synthesis

To evaluate the performance of the VHDL code generated with Lift, we implemented an integer 64bits matrix multiplication using the optimizations described in Section 4. We compared the Lift generated program against a naive implementation in C, a hand-tuned implementation in C (using tiling and OpenMP), and an FPGA design produced using a commercial High-level Synthesis tool (Vivado).

Figure 6 summarizes the results of our experiments for matrices of different sizes from 128x128 to 2048x2048. The same tile size is used for all input sizes. Tile size is selected to be the largest possible that would still fit onto the FPGA logic. In our experiments, tile size is selected to be 128x128. Note, for the smallest input size, tile size is equal to the input size, thus no tiling is performed by the host.

The baseline of our experiments, a naive implementation of the matrix multiplication in C, performs poorly. It has no particular optimization in term of cache reuse, and it is single threaded. In order to have a fair comparison between CPU and FPGA, we also implemented an optimized C program using tiling, and thread parallelism. This version performs an order of magnitude faster than the naive C.

The Lift generated FPGA designs are up to ten times faster than the hand-tuned C CPU version and one hundred times faster than the naive C implementation. Generally, the performance of the Lift-generated FPGA designs are ten times faster than those generated with the HLS tool.

The HLS tool we used automatically generates the communication between the host and the FPGA during compilation. The implementation of matrix multiplication we used was tiled, the tile size was up to 2KB. The HLS tool takes approximately 15 minutes to generate a bitstream for matrix multiplication of 2048x2048 matrices (Table 1). In contrast the synthesis of Lift programs was ten times faster. This is mostly because most the Lift design reuses a set of common IP blocks. In terms of FPGA utilization, a bitstream obtained using Lift is comparable with HLS design except that the

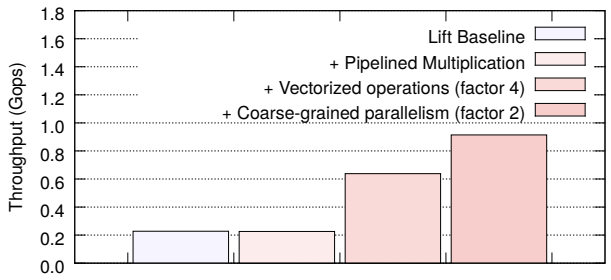


Figure 7. Impact of optimizations on Lift generated FPGA design for 128x128 matrix multiplication.

number of used DSP was five time bigger. This higher usage of DSP explains the difference in runtime performance since Lift makes efficient use of the available hardware resource.

5.3 Study of Optimization Benefits

To better understand the effect of the optimizations implemented in Lift on performance, we generate multiple implementations for the same matrix multiplication program with different optimizations written by hand in Lift. We executed all designs at the same frequency (100Mhz). Figure 7 shows the results. A first naive implementation of matrix multiplication for matrices of 128x128 elements achieves only 0.22 Gops. In the case of small matrices there is no need for tiling as the FPGA's BRAM is big enough to hold the entire matrix. Using pipelined multiplication instead of standard multiplication operator (as described in Section 3) will have no impact on the execution time, however the maximum clock frequency to be used with the design could be increased. The optimization with the biggest impact is vectorization (described in Section 4). The maximum we were able to generate was using a factor of 4, leading to a throughput of 0.63 Gops. Furthermore, with coarse-grained parallelism of a factor of 2, we reached the final performance for this frequency of 0.91 Gops.

6 Related Work

To improve the programmability of hardware, design tools provide optimized version of primitive operations, such as addition or multiplication. However, the programmer must still express the application and connection of library functions manually at low level. Instead of directly using Hardware Description Languages, several projects provide abstractions. For example, SystemVerilog [1] is a major improvement compared to Verilog or VHDL. It introduces object-oriented programming, strong type system, and parametrization of hardware modules. Bluespec [17] includes even higher-order functions and polymorphism. Esterel [7] creates abstraction of Verilog control flow elements, reducing code size and producing optimized circuits. Functional languages such as μFP [21], HML [14], or CλaSH [2] provide similar features. Lava [5] and Kansas Lava [9] are embedded DSLs in Haskell to describe circuits aiming to provide an elegant interface between design implementations and verification tools. Similarly, Chisel [3] is an embedded DSL in Scala. Even though these approaches take advantage of modern programming language features, they require the programmer to understand hardware design concepts. Moreover, designed application is directly mapped to hardware without much compiler optimization. Achieving good performance is hard for any application of moderate size.

High-Level Synthesis (HLS) frameworks compile C-based programs into hardware [16]. The advantage of HLS is that software programmers can design hardware, which is a great step in programmability of FPGAs. C-based approaches, however, have limited power to express coarse and fine grained parallelism. Furthermore, C and C++ have software specific constructs, such as dynamic memory management, pointers, and recursion, which complicate designing hardware. Other approaches rely on heterogeneous programming frameworks such as CUDA [18] or OpenCL [6]. While expressing coarse and fine grained parallelism of application, they remain low-level, harder to program than C or C++.

A different approach is HLS from functional languages [12, 15, 27], and dataflow languages [11, 13, 25]. For example Townsend *et al.* [27] uses a subset of Haskell and perform a syntax-directed translation to a internal dataflow representation which ultimately translated into a hardware design. Those works are closer to ours, as applications are expressed using high-level platform-agnostic patterns. However, with Lift, we want to take advantage of its rewrite rule engine that enables automatic design space exploration and can lead to competitive implementation for FPGA.

In that regard Delite [26] is the closest work to Lift. Delite also provides a FPGA backend [8, 19]. However, their solution differs in two important ways. First, the Lift framework adopts a different philosophy compared to Delite which uses generic (e.g. dead-code elimination) and domain-specific (e.g. tiling) transformations in a traditional fashion. Pattern-based transformations are thus part of domain-specific transformations and are not platform-specific. In contrast Lift is solely based on rule rewriting which allows a bigger exploration space, with more opportunities for optimization. Second, the Delite backend generates either C or JMax, while our proposed approach directly produces VHDL.

7 Conclusion and Future Work

In this paper, we describe a new backend for the Lift framework enabling efficient FPGA code generation. Demonstrating with matrix multiplication, we show that our approach achieves significantly higher performance than optimized C code as well as the FPGA design by a commercial HLS

Table 1. FPGA synthesis data for 64 bits matrix multiplication of 2048x2048 matrices using Lift and a commercial HLS tool.

	LUT pairs	LUTs	Slices	Registers	DSP	BRAMs	Frequency	Synthesis Time	Execution Time
Resources	17600	17600	4400	35200	80	60	-	-	-
Lift	12%	30%	50%	18%	60%	70%	131 Mhz	2 min	20 sec
HLS	19%	32%	66%	29%	13%	80%	100 Mhz	15 min	126 sec

tool. Optimizations such as tiling, vectorization, and coarse-grained parallelism are expressed as rewrite rules. This enables sophisticated design space optimizations using Lift's existing rewrite engine.

We believe that this work paves the way for a fully automated and efficient HW/SW code generation of heterogeneous systems where CPU, GPU, and FPGA can be used altogether. Nevertheless, this work is still in progress and further steps are necessary for achieving our ultimate goal of a high-level platform-agnostic framework for programming of heterogeneous systems. We plan to extend our approach with more low-level patterns specialized for FPGAs as well as rewrite rules implementing transformations commonly used in HLS [16].

Future work will also focus on convolutions, which is a key component of neural networks. We are confident that the approach presented in this paper will be suitable for producing efficient convolution, given that they can be represented easily as matrix operations.

References

[1] Accellera. 2002. SystemVerilog 3.0 AccelleraâĂŹs Extensions to Verilog. (2002).

[2] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. 2010. CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *EUROMICRO*. https://doi.org/10.1109/DSD.2010.21

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *DAC*. ACM.

[4] David F Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA programming for the masses. *Commun. ACM* 56, 4 (2013), 56–63.

[5] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: hardware design in Haskell. In *ICFP*. ACM.

[6] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. 2012. From OpenCL to high-performance hardware on FPGAs. In *FPL*. IEEE.

[7] Stephen A Edwards. 2002. High-Level Synthesis from the Synchronous Language Esterel. In *IWLS*.

[8] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. 2014. Hardware system synthesis from Domain-Specific Languages. In *FPL*. IEEE.

[9] Andy Gill. 2011. Declarative FPGA circuit synthesis using Kansas Lava. In *ERSA*.

[10] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. ACM, New York, NY, USA, 100–112. https://doi.org/10.1145/3168824

[11] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. 2008. Optimus: Efficient Realization of Streaming Applications on FPGAs. In *CASES*.

[12] Shan Shan Huang, Amir Hormati, David F Bacon, and Rodric M Rabbah. 2008. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary.. In *ECOOP*. Springer.

[13] Hyunuk Jung, Hoeseok Yang, and Soonhoi Ha. 2008. Optimized RTL code generation from coarse-grain dataflow specification for fast HW/SW cosynthesis. *Journal of Signal Processing Systems* 52, 1 (2008), 13–34.

[14] Yanbing Li and M. Leeser. 1995. HML: an innovative hardware description language and its translation to VHDL. In *ASP-DAC*. https://doi.org/10.1109/ASPDAC.1995.486388

[15] Alan Mycroft and Richard Sharp. 2000. A Statically Allocated Parallel Functional Language. In *ICALP*.

[16] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604. https://doi.org/10.1109/TCAD.2015.2513673

[17] Rishiyur S. Nikhil. 2008. *Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions*. Springer Netherlands, Dordrecht. https://doi.org/10.1007/978-1-4020-8588-8_8

[18] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. 2009. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *SASP*.

[19] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating Configurable Hardware from Parallel Patterns. In *ASPLOS*. ACM.

[20] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ISCA*. IEEE.

[21] Mary Sheeran. 1984. muFP, a Language for VLSI Design. In *LFP*. ACM.

[22] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. In *ICFP*. ACM.

[23] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2016. Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation. In *CASES*. ACM.

[24] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *CGO*. ACM.

[25] Robert Stewart, Deepayan Bhowmik, Andrew Wallace, and Greg Michaelson. 2017. Profile Guided Dataflow Transformation for FPGAs and CPUs. *Journal of Signal Processing Systems* 87, 1 (01 Apr 2017), 3–20. https://doi.org/10.1007/s11265-015-1044-y

[26] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article

134 (April 2014), 25 pages. https://doi.org/10.1145/2584665

[27] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. 2017. From Functional Programs to Pipelined Dataflow Circuits. In *CC*. ACM.

[28] Xilinx. 2017. Zybo Zynq-7000 ARM/FPGA SoC Trainer Board. https://www.xilinx.com/products/boards-and-kits/1-4azfte.html.