Optimizing Data Reshaping Operations in Functional IRs for High-Level Synthesis

Optimizing Data Reshaping Operations in Functional IRs for High-Level Synthesis

Christof Schlaak
University of Edinburgh
United Kingdom
christof.schlaak@ed.ac.uk

Tzung-Han Juang
McGill University
Canada
tzung-han.juang@mail.mcgill.ca

Christophe Dubach
McGill University
Canada
christophe.dubach@mcgill.ca

Abstract

FPGAs (Field Programmable Gate Arrays) have become the substrate of choice to implement accelerators. They deliver high performance with low power consumption, while offering the flexibility of being re-programmable. But they are notoriously hard to program directly using HDLs (Hardware Description Languages). Traditional HLS (High-Level Synthesis) methods are addressing some of these issues but are far from being perfect. Programmers are still required to write hardware-specific code and existing HLS tools often produce sub-optimal designs.

Modern approaches based on multi-level functional IR (Intermediate Representation) such as Aetherling, have demonstrated the advantages of generating high performance designs in a predictable way. A functional IR makes optimizations via rewrite rules simple to express, and abstract away the hardware details. However, as we will see in this paper, functional approaches bring their own set of challenges to produce high performance hardware. In particular, data reshaping operations, such as transposition, introduce overheads that hurt performance or even prevent the generation of synthesizable hardware designs altogether.

This paper presents an approach with rewrite rules to solve this fundamental issue and produce efficient FPGA designs from functional IRs. Using rewrites, it is possible to generate high performance designs for matrix multiplication and 2D convolution. The paper also evaluates the performance impact of the optimizations and shows that without them, low performance designs are produced, or even worse, it is impossible to synthesize the designs at all.

LCTES '22, June 14, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9266-2/22/06…$15.00
https://doi.org/10.1145/3519941.3535069

CCS Concepts: • **Hardware → Hardware accelerators**; • **Software and its engineering → Functional languages**; **Source code generation**.

Keywords: High-level synthesis, functional IR, rewrite rules, compilers

ACM Reference Format:
Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. 2022. Optimizing Data Reshaping Operations in Functional IRs for High-Level Synthesis. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3519941.3535069

1 Introduction

When implementing accelerators, FPGAs shine due to their high efficiency and flexibility. However, this flexibility comes with extra burden placed on the programmer. Traditional HLS tools try to simplify hardware development but, despite their name, they are still relatively low-level and require some hardware knowledge to achieve high performance. Furthermore, these toolchains are typically built using traditional software compiler passes that are not a good match for hardware optimization. This makes it hard to predict what the hardware will look like at the end of the compilation process [17], often leading to poor designs.

The compiler community has recently witnessed a shift towards multi-level IRs (*e.g.*, MLIR [14]). A multi-level IR simplifies the compiler design by decomposing compilation into a lowering process with multiple simple steps. Multi-level IRs are a great fit for HLS as recently demonstrated by Lift-hls [12] and Aetherling [5]. With such approaches, the IR is gradually lowered: high-level abstractions are replaced with hardware-specific primitives, until the desired hardware design is generated. Furthermore, the use of a high-level, functional IR as an entry point to the compiler ensures that programs can be written in a simple, hardware-agnostic way.

However, despite the success of prior work, several challenges remain with *functional approaches in general*. In particular, as we will see in this paper, they lack a systematic approach to deal with data reshaping operations efficiently. Data reshaping is required to express important workloads such as MxM (Matrix Multiplication) or convolution.

In such workloads, there is a need for *repeating*, *transposing* or *sliding* data. Aetherling [5], for instance, does not support these features and simply cannot express MxM, nor 2D convolution. Lift-hls [12] implements a work-around for MxM, but does not address the repetition issue generally, while transposing and sliding of data, as needed for tiling and convolution, is not supported at all.

This paper makes two major contributions. First, it identifies the challenges that functional data reshaping operations present during high-level synthesis. In particular, it discusses and evaluates the impact of not optimizing such reshaping operations. Most of the issues encountered are related to needs for large on-chip buffers or complex signal wiring, in order to perform the reshaping operations.

Secondly, the paper shows how each challenge is tackled using a set of simple rewrite rules. The key idea consists of moving the reshaping operations towards the source that produces the data in the first place. When this source is, for instance, a read operation, indexed by a stream of addresses, the data reshaping operations are implemented by manipulating the read addresses instead of the data itself. As we will see, manipulating addresses is much cheaper and faster, than reshaping the data produced by the read operations.

The performance results show that without the optimizations presented in this paper, a functional approach either produces designs that do not synthesize due to resource constraints, or produces designs with abysmal performance.

To summarize, the main contributions of this paper are:

- The identification of the main causes of hardware inefficiencies associated with data reshaping operations;
- A rewrite rule approach to optimize away reshaping operations by merging the reshaping operations when possible with other primitives;
- An evaluation of this optimization process on matrix multiplication and 2D convolution on a real FPGA.

The rest of the paper is structured as follows: Section 2 presents background information about functional languages for hardware generation and highlights the challenges associated with data reshaping operations. The main contributions of this paper are presented in section 3, which describes both the source of inefficiencies for common data reshaping operations as well as the solution to these problems. Section 4 evaluates the approach on matrix multiplication and 2D convolution, showing the effects of the optimizations presented. Finally, section 5 discussed related work, while section 6 concludes this paper.

2 Functional IRs for Hardware Generation

Lift-hls [12], Aetherling [5] and Spatial [11] have recently shown that hardware synthesis from high-level functional IRs can deliver performance for simple use-cases. This paper is based on similar principles and this section reviews such functional IRs and shows their unresolved challenges.

$$Join : [T]_M \rightarrow [T]_N \rightarrow [T]_{M+N} \quad (1)$$

$$Split\text{<}N\text{>} : [T]_M \rightarrow [[T]_N]_{M/N} \quad (2)$$

$$Zip : ([T]_N, [U]_N) \rightarrow [(T, U)]_N \quad (3)$$

$$Map : (T \rightarrow U) \rightarrow [T]_N \rightarrow [U]_N \quad (4)$$

$$Reduce : (U \rightarrow (T \rightarrow U)) \rightarrow [T]_N \rightarrow U \quad (5)$$

(a) Common functional hardware-agnostic primitives. $[T]_N$ denotes an array of N elements of type T.

$$MapStm : (T \rightarrow U) \rightarrow {}^{STM}[T]_N \rightarrow {}^{STM}[U]_N \quad (6)$$

$$MapVec : (T \rightarrow U) \rightarrow {}^{VEC}[T]_N \rightarrow {}^{VEC}[U]_N \quad (7)$$

(b) Hardware-specific `Map` over stream (STM) or vector (VEC) data, as in the IRs of Lift-hls [12] and Aetherling [5]. Definitions for `ZipStm`, `SplitVec`, etc. are similar to the hardware-agnostic ones, except that stream and vector data types are used instead.

Figure 1. Examples of functional primitives. Type variables T and U are data types, M and N are natural numbers that mainly encode array length.

2.1 Background

Hardware-agnostic IR. Examples of typical high-level hardware-agnostic primitives, as they appear in many functional languages, are listed in fig. 1a. They focus on the *what*, rather than the *how*. This frees up the programmer from having to worry about the underlying hardware specifics to achieve performance. Instead, the compiler is responsible for providing performance, given a high-level input program.

Producing high-performance hardware from such high-level primitives is challenging. The compiler must make many choices regarding parallelization and buffering strategies. To address this challenge, it is typical to use another IR level to encode implementation choices.

Hardware-specific IR. At the lowest level, choices about whether computation is run in parallel or sequential, for instance, are encoded directly in the IR. The two low-level *Map* primitives in fig. 1b express this choice.

The two types of data collections, vectors and streams, and their corresponding `Map` processing operations, are illustrated in fig. 2. In the context of hardware generation, a `MapStm` produces a sequential process in *time*, consuming and producing a stream of data with potential for pipeline parallelism. In contrast, `MapVec` processes data in *space*, resulting in spatial parallelism with the function f duplicated in hardware for every element of the vector.

Lift-hls [12], Aetherling [5], Spatial [11] and Shir [21] are examples of HLS approaches that use such a type of functional hardware-specific IR. Further design choices, i.e. whether data buffered in on-chip or off-chip memory can be encoded in a similar style, as shown in Shir [21].

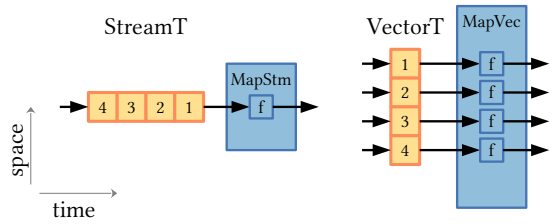


Figure 2. Streamed data (left) is passed element by element through function f of `MapStm`. `MapVec` creates multiple instances of f to process a vector (right) in parallel.

Memory access to on-chip or off-board RAM (Random Access Memory) is represented in the IR with `Read` and `Write` primitives. `Read`, for instance, receives an n-dimensional stream of addresses and returns an n-dimensional stream of data from memory.

Compilation to Optimized Hardware. While hardware-specific IRs are great at expressing hardware choices, it remains challenging to lower a high-level, hardware-agnostic IR into such a hardware-specific one. Spatial [11] gets around this problem by requiring programmers to express directly their program with the hardware-specific IR. In contrast, Lift-hls [12], Aetherling [5] and Shir [21] tackle this problem head on using a system of rewrite rules.

A rewrite rule could, for instance, parallelize a *Map*. Additional conditions may further restrict the application of a rule. The following example shows how a hardware-agnostic input program to the compiler, eq. (8), could be lowered step by step, using rewrites, into a hardware-specific expression eq. (10) encoding some parallelization optimization:

$$Map(Mul) \circ Zip \circ (inputA, inputB) \tag{8}$$

$$\downarrow \text{lowering} \downarrow$$

$$MapStm(Mul) \circ ZipStm \circ (inputA, inputB) \tag{9}$$

$$\downarrow \text{optimization} \downarrow$$

$$VecToStm \circ MapVec(Mul) \circ StmToVec \circ ZipStm \circ (inputA, inputB) \tag{10}$$

Since each rewrite is simple and type-preserving, any combination of rewrites will preserve program semantics. Each intermediate step (between two rewrites) results in an expression that is synthesizable. Furthermore, the rewrites are designed so that it is possible to apply them in any order until a fixed-point is achieved.

Once all rewrites have been applied, a simple process turns the final hardware-specific representation of the program, eq. (10), into a hardware description based on VHDL (VHSIC Hardware Description Language) code. This paper's methodology builds on top of prior work by Shir [21].

2.2 Data Reshaping Challenges

Despite all the advantages of a high-level functional approach to hardware design, there are some cases where these concepts are too far away from the hardware world. In particular data reshaping operations such as transposition or sliding of data, as used in 2D convolution for instance, might require large amounts of hardware if implemented naively.

Nested `Map` also cause problems when accessing different input data as in the case of MxM. In such cases, the input data must be repeated multiple times. A trivial task in pure software, since memory can be read multiple times. However, in the hardware world, data arrives in the form of streams from memories or data generators. Repeating such a data stream is a non-trivial business, since its state needs to be reset and some control mechanism needs to determine how many times the stream needs to be repeated. This might not be supported by the streaming protocol, or at best 'only' introduces long latency if the computation is deeply pipelined.

These issues are not specific to just the compiler presented in this paper, but to all functional based approaches for hardware generation. Direct compilation from such a representation to hardware sometimes results in slow performance or even non-synthesizable hardware designs, as we will see in the evaluation in section 4.

Table 1 provides a feature table that shows the limitations of state-of-the-art functional approaches for high-level synthesis. As can be seen, Lift-hls [12] provides none of the data reshaping optimizations but manages to implement MxM with a non-generic work-around for the data repetition. The challenges around transposition or sliding windows are left untouched, which is why convolution cannot be realised. Aetherling [5] has not support for repetition of streams, transposition and is therefore unable to encode matrix multiplication or 2D convolutions. Spatial [11] is able to express the above data reshaping operations, but since there are no hardware-agnostic primitives, it is up to the user to perform the optimizations manually. Finally, Spiral [22] is not very generic and focuses mostly on FFT (fast Fourier transform).

2.3 Summary

The goal of this paper is to show how to turn high-level, hardware-agnostic programs into high performance designs in the presence of data reshaping operations. The main contributions of this paper are two-fold. First, it identifies the problems created by *data reshaping* in the context of hardware synthesis. Secondly, it shows how to make these problems disappear — literally — by applying rewrite rules.

These contributions will benefit all functional-based IR used for high-level synthesis. Furthermore, although the approach is evaluated on two applications — matrix multiplication and convolution — that exercise the reshaping operations presented, it is applicable to any application expressed as a combination of dense array operations.

	Optimizations				Applications			
Framework	hw-agnostic IR	stream repetition	transpose & slide2D	stm-vec conversions	FPGA evaluation	matrix mult.	1D conv.	2D conv.
Lift-hls [12]	✓	✗	✗	✗	✓	✓	✗	✗
Aetherling [5]	✓	✗	✗	✗	✗	✗	✓	✗
Spatial [11]	✗	— manual —			✓	✓	✓	✓
Spiral [22]	✓	✗	✗	✗	✓	✗	✗	✗
This paper	✓	✓	✓	✓	✓	✓	✓	✓

Table 1. Features of related work compared to this paper.

3 Rewriting Reshaping Operations

A rewrite rule system is used to generate high performance designs in the presence of reshaping data. As this section will show, rewrites make data reshaping operations more efficient on the underlying hardware, while preserving the program semantics. Instead of reshaping the data, as implied by the programmed algorithm, it is more efficient to reorganize the read addresses of that data in memory instead.

3.1 Repetition of Data

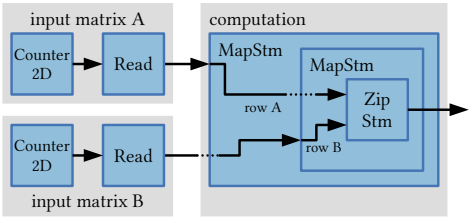
Generally, in an accelerator design the application's input data is read from memory and then streamed through the compute logic of the FPGA. However, some applications require the input data to be repeated.

One simple example for this is the creation of all possible pairs of the rows of two matrices, as in this functional code:

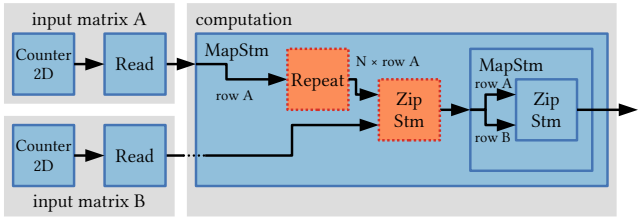
$$MapStm(\lambda\, rowA \Rightarrow MapStm(\lambda\, rowB \Rightarrow ZipStm \circ (rowA, rowB)) \circ inputMatrixB) \circ inputMatrixA \tag{11}$$

Here, the outer map operates on the input matrix A, while the inner map operates on input matrix B. For functional programs executed in software, this example works without any issue. But if hardware is generated in a naive way, by creating a hardware module for each primitive in the program, as depicted in fig. 3a, the result will be incorrect.

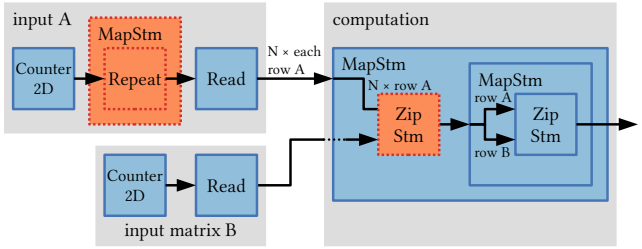
3.1.1 Problem Statement. A naive hardware implementation for `MapStm` would just extract the rows from the input matrix. Then, the `ZipStm` returns the pairs of the n-th row of matrix A and the n-th row of matrix B. However, due to the simultaneous use of parameters from two different, nested `MapStm` expressions, the original program specifies a different behaviour: The rows must be repeatedly read from memory. The compiler *must* introduce some extra logic in the generated hardware design to achieve the correct repetition of data.



(a) The result of the naively generated hardware design is wrong, because the rows of matrix A are not repeated correctly.



(b) A `Repeat` block is inserted to achieve the required repetition of one of the input rows. This repeat will back-propagate a signal to the producer to inform that the stream must be repeated.



(c) The `Repeat` has moved into the address generation, to remove the need to back-propagate repeat signals and to improve the performance. The `Read` block for input A transforms a 3D stream of addresses into a 3D stream of data, containing N times each row.

Figure 3. Hardware block diagrams to create all pairs of rows for two input matrices. Counters emit 2D streams of addresses, that, after reading, result in 2D streams of values. Newly inserted modules are orange and have a dashed border. A dotted signal entering a `MapStm` means that this signal is *not* an input to this `MapStm` block.

In general, this is required, whenever a lambda is using an *unbound parameter* in its body. In the above example, the inner lambda is accessing the unbound parameter `rowA`, which must therefore be repeated.

A more familiar example for such a memory access pattern is *Matrix Multiplication*, with the only difference being the calculation of the dotproducts from these pairs of rows. This is under the assumption that the second matrix has been transposed in memory already. For two $N \times N$ matrices A and B, where $A \times B = C$, each row of matrix A and the entire matrix B are read N times. Thus, the input data must be repeated to provide the correct data for the computation.

This data repetition problem is solved by breaking it down into two simple, automatic steps. First, the compiler detects unbound parameter accesses and inserts explicit `Repeat` primitives around each access. While this might not be the most optimal location, as we will see shortly, this produces functionally correct hardware. Secondly, rewrite rules move these new primitives in the IR to more optimal places to improved performance, while preserving the functional semantics of the given application.

3.1.2 Insertion of Explicit Repeat. To enable the compiler to explicitly encode repetitions in the high-level IR, the `Repeat` primitive is introduced first. It repeats the input stream N times to create a matrix of $N \times M$ elements T:

$$Repeat\langle N\rangle : \overset{STM}{}[T]_M \rightarrow \overset{STM}{}[\overset{STM}{}[T]_M]_N \tag{12}$$

Coming back to the above example, the parameter `rowA` is wrapped in such a `Repeat`. Furthermore, the unbound parameter from the inner lambda is removed entirely using a rewrite rule. This produces an expression where the repeated row of A and the matrix B are zipped together and fed into the map expression:

$$\begin{aligned} &MapStm(\lambda\, rowA \Rightarrow \\ &\quad MapStm(ZipStm) \circ ZipStm \\ &\qquad \circ \left(Repeat\langle N\rangle \circ rowA, inputMatrixB\right)) \\ &\quad \circ inputMatrixA \end{aligned} \tag{13}$$

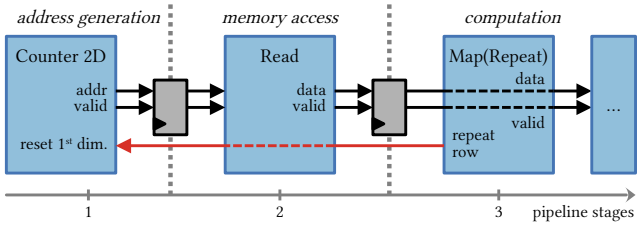
This will repeat one row of matrix A N times before proceeding to the next row. Thus, the generated hardware design will contain a new module, as shown in the block diagram in fig. 3b, and produce the correct result.

Nevertheless, this design is far from perfect as the repetition of the rows is dynamic and therefore determined during runtime of the hardware. The counter is unable to predict when addresses must be repeated. Whenever it emits the last memory address of a row, the counter has to wait until it receives a potential repeat signal to decide whether to repeat the current row or to advance to the next one. Figure 4a visualizes a pipeline with such a repeat signal. This pipeline between counter and repeat logic must stall to allow this communication. As the corresponding timing in fig. 4c shows, this stall causes a delay which depends on the number of registers in the data flow.

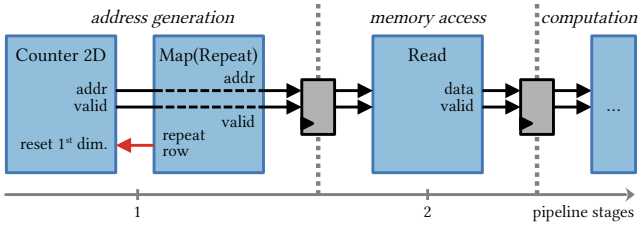
3.1.3 Repeat Optimization. This section now looks at how to avoid the delay of the repeat signal. To achieve this, the compiler applies rewrite rules to move the `Repeat` inserted in the previous step. The `Repeat` primitives in the IR are *moved* step by step towards the IR's leaves, which are address counters for reading the program's input data. Effectively, the *repetition of data* is transformed into a more efficient *repetition of addresses*, as depicted in fig. 3c.

The counter and repeat logic communicate within the same pipeline stage, as fig. 4b shows. Thus, the generated hardware is able to generate valid memory addresses each cycle, as indicated in fig. 4d.

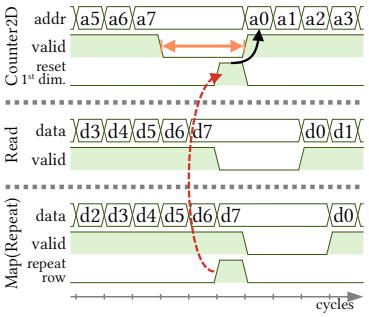
This optimization is implemented as a set of about 20 *simple* semantic-preserving rewrite rules which are applied as a fixed-point iteration over the IR.



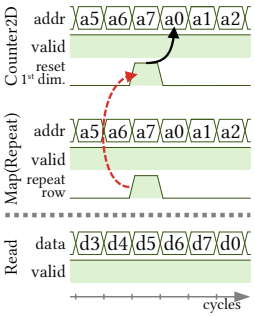
(a) Repetition of data: The address generation part must stall to allow the repeat logic to communicate with the counter over multiple pipeline stages.



(b) Repetition of addresses: The repeat logic is next to the counter. They communicate directly, no pipeline stall required.



(c) The counter must wait 3 cycles (orange) for a potential repeat signal before sending the next address row. Here, a repeat row signal is sent, causing the address to jump back to `a0`.



(d) The repeat signal is sent instantly when the row's last address arrives. No cycles are wasted due to pipeline stalls.

Figure 4. Block diagrams and waveforms before (a), (c), as well as after repeat optimization (b), (d). Here, the 2D counters emit an 8x8 matrix of addresses. The repeat logic, `Map(Repeat)`, sends a signal (red arrow) to the address counter to reset its inner dimension. Grey boxes are registers. Memory access takes one cycle in this example.

For instance, one rule splits a `MapStm` to isolate the `Repeat` primitive:

$$MapStm(f \circ Repeat\langle N\rangle) \circ input \Longrightarrow MapStm(f) \circ MapStm(Repeat\langle N\rangle) \circ input \quad (14)$$

While another rule moves the repetition across the `Read` primitive to cause a repetition of memory addresses:

$$MapStm(Repeat\langle N\rangle) \circ Read \circ input \Longrightarrow Read \circ MapStm(Repeat\langle N\rangle) \circ input \quad (15)$$

After applying such rewrites, the `Repeat` primitives will eventually reach an address counter. In the IR, counters are defined based on an initial value C_0, a step size C_S and a maximum value C_M:

$$Counter\langle C_0, C_S, C_M\rangle : \overset{STM}{}[T]_{(C_M - C_0)/C_S} \quad (16)$$

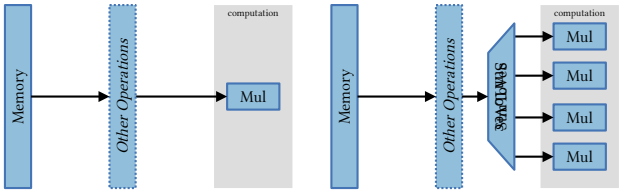
Given the example in eq. (13), the automatic rewriting process returns the following optimized expression. This time, the part for reading input matrix A is expanded to show the repetition next to the counter (last two lines):

$$\begin{aligned} &MapStm(\lambda\, rowA \Rightarrow \\ &\quad MapStm(\lambda\, rowB \Rightarrow ZipStm \circ (rowA, rowB)) \\ &\qquad \circ inputMatrixB) \\ &\quad \circ Read \\ &\quad \circ MapStm(Repeat\langle N\rangle) \\ &\quad \circ Split\langle N\rangle \circ Counter\langle 0, 1, N * N - 1\rangle \end{aligned} \quad (17)$$

This address generation no longer simply increments its value, but jumps back to the beginning of the current row, whenever a repetition is desired. The compiler is able to generate very efficient hardware for such a combination of counters and repeat modules. A valid address value is produced each clock cycle. This way, the repetition problem is solved statically, during compile-time, and the problem of run-time pipeline stalls, as mentioned above and depicted in fig. 4c, disappears. This will lead to improved overall performance, as evaluated in section 4.

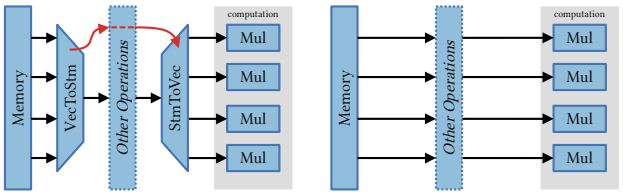
3.2 Stream and Vector Conversions

Stream and vector types allow to easily express designs with different area usage and performance characteristics on the FPGA. For design space exploration the compiler must be able to trade in available resources for increased throughput with a more parallelized design. For this, stream-based primitives are simply substituted with their vector-based counterpart. If, for example, a `MapStm` with a function f is replaced by a `MapVec`, many instances of f will be created to process all the elements in parallel. Mixed stream and vector designs are also possible by first splitting the input stream and then only parallelizing the inner stream part.



(a) Fully stream-based design with only one multiplier. The memory output width matches the input width of the computational part.

(b) `StmToVec` is inserted to connect the new 4x parallelized computation to the remaining part of the design. The memory port width is now the bottleneck.



(c) Memory width scaled by 4. `VecToStm` is inserted to connect it to the rest of the design. To remove such conversions, the `VecToStm` is moved (red arrow) until it reaches the `StmToVec`.

(d) Final design after rewriting: The high memory bit-width is exploited by the parallelized computation. The performance is no longer impaired by the conversions.

Figure 5. Rewriting process from initial design (a) through intermediate steps (b) and (c) to fully parallelized design (d). All wires (black lines) have the same bit-width.

3.2.1 Parallelization.

Given an initial design, as seen in fig. 5a, it is possible to parallelize it by first converting the stream into a vector, using `StmToVec`, and then by performing the computation in parallel as shown in fig. 5b.

With the computation parallelized, the bandwidth of the data source —usually memory— must be increased, as in fig. 5c, to feed enough data to the computational part. Otherwise, the generated hardware would not perform well due to waiting times. Again, a `VecToStm` conversion must be used to maintain interoperability with the remaining part of the accelerator design.

After increasing the memory's bandwidth, the communication between this source of data and computation can still contain some 'other operations', as in fig. 5c. These operations may consist of further computations or even data type conversions, because the memory's hardware interface may differ from the required input data type of the computation. The overall performance is still impaired by the communication bottleneck between memory and computation.

3.2.2 Conversion Optimization.

In order to speed up the communication between memory and computation, a design as shown in fig. 5c must be transformed into a design as in fig. 5d.

The compiler achieves this optimization automatically by moving the `VecToStm` primitive away from its data source, through the 'other operations', towards a potential `StmToVec` primitive. Once these two conversions meet in the IR, the communication bottleneck is fixed, by removing both conversions with the following rewrite rule:

$$StmToVec \circ VecToStm \circ input \\ \Longrightarrow input \tag{18}$$

The `VecToStm` primitive is moved through the IR by applying a set of about 20 rewrite rules, some of which are described below. If, for example, a `VecToStm` in combination with another function f is found in a `MapStm`, it is isolated with the following map-fission rule:

$$MapStm(VecToStm \circ f) \circ input \\ \Longrightarrow MapStm(VecToStm) \circ MapStm(f) \circ input \tag{19}$$

The rewriting system now continues to move the combination of `MapStm` and `VecToStm` further away from the data source and towards a potential `StmToVec`. A map-fusion rule combines two `MapStm` into a single one, enabling further rewrites to then move the `VecToStm` through function f:

$$MapStm(f) \circ MapStm(VecToStm) \circ input \\ \Longrightarrow MapStm(f \circ VecToStm) \circ input \tag{20}$$

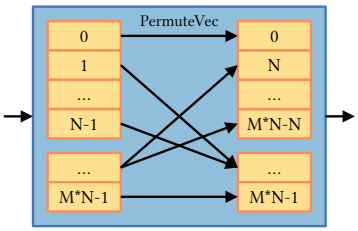
If a `VecToStm` meets a `JoinStm` primitive during its relocation in the IR, it is stuck and cannot skip this `JoinStm`. It is impossible to replace a $JoinStm \circ VecToStm$ combination with another expression, where the `VecToStm` occurs on the left-hand side, while leaving the overall semantics untouched. Therefore, the `JoinStm` also needs to be pushed through the IR as well, to make room for a more parallelized design. One of the rewrite rules responsible for this works as follows, where `AnyExpr` is a placeholder for any expression:

$$AnyExpr \circ JoinStm \circ input \\ \Longrightarrow JoinStm \circ MapStm(AnyExpr) \circ input \tag{21}$$

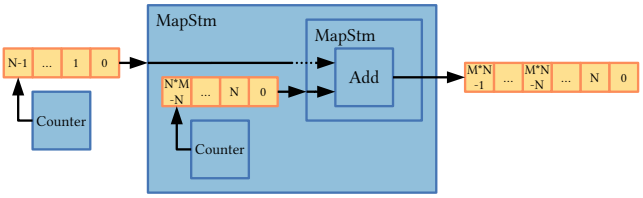
In summary, memory and computation related parallelization optimizations easily cause a demand for additional conversions to be inserted into the IR to preserve a feasible design. These conversions, namely `VecToStm` and `StmToVec`, are a performance bottleneck for the hardware design. The rules presented exemplify how `VecToStm` is moved in the IR to eventually find a matching `StmToVec` and then annihilate both conversions. Once the fixed-point iteration terminates, the communication wires between a wide memory and a parallelized computation are also parallelized, as in fig. 5d.

3.3 Transposition

If the input data for MxM or convolutional layers is too large for the FPGA's on-chip buffers, *tiling* is applied. The large input matrix is partitioned into several small tiles, which are then processed one by one.



(a) Naive transposition of 2D vectors based on the `PermuteVec` primitive and the permutation function $\lambda\, i \Rightarrow i/N+(i \bmod N)*M$.



(b) Optimized hardware to generate transposed 2D addresses. The complex wire mesh, as in fig. 6a, is no longer required.

Figure 6. Block diagrams for transposition of 2D data.

Using the `Split` primitive, horizontal slices are easily extracted from matrices. Full tiling, however, also requires these slices to be split into vertical chunks. For this, the `Split` primitive is applied on the *transposed* matrix. With this simple trick, tiling with M by N tiles is expressed as:

$$Transpose \circ Map(Split\langle M\rangle) \\ \circ Transpose \circ Map(Split\langle N\rangle) \tag{22}$$

To implement transposition, which is a special case of permutation, the `PermuteVec` primitive is first introduced:

$$PermuteVec\langle(N \to N)\rangle : \overset{VEC}{[T]}_N \to \overset{VEC}{[T]}_N \tag{23}$$

It operates on a vector of N elements by applying a statically known permutation function $(N \to N)$ to manipulate the order of elements based on their indices.

Now, with this new primitive, transposition of 2D vectors is expressed by first flattening the M by N input matrix using `JoinVec`, then permuting it with a more complex function, and finally splitting it again:

$$TransposeVec = SplitVec\langle M\rangle \\ \circ PermuteVec\langle(\lambda\, i \Rightarrow i/N + (i \bmod N) * M)\rangle \\ \circ JoinVec \tag{24}$$

Stream-based transposition builds on this but requires the 2D input data to be converted to vectors first:

$$TransposeStm = MapStm(VecToStm) \circ VecToStm \\ \circ TransposeVec \circ StmToVec \circ MapStm(StmToVec) \tag{25}$$

3.3.1 Problem Statement. As demonstrated, all the required functionality for transposition is expressible based on `PermuteVec`. The generated hardware for this primitive results in several wire assignments, as depicted in fig. 6a. For large data structures, the wiring quickly becomes too complex and requires more area than feasible for the FPGA.

3.3.2 Transpose Optimization. In order to address this issue, a fixed-point iteration over the IR applies rewrite rules to express the transposition more efficiently. Similar to the repeat optimization in section 3.1, the transposition of data is rewritten as a transposition of memory read addresses.

Initially, the program given to the compiler may contain the following expression for transposition on data:

$$TransposeStm \circ SplitStm\langle N\rangle \circ Read \circ Counter\langle 0, 1, M * N\rangle \tag{26}$$

Given this expression, the following rewrite rule moves the `TransposeStm` further towards the `Read`'s input — the counters that generate the addresses:

$$TransposeStm \circ SplitStm\langle N\rangle \circ Read \circ input \Longrightarrow Read \circ TransposeStm \circ SplitStm\langle N\rangle \circ input \tag{27}$$

As a result of this rewriting, the complex data reshaping operations are placed next to the address counter:

$$Read \circ TransposeStm \circ SplitStm\langle N\rangle \circ Counter\langle 0, 1, M * N\rangle \tag{28}$$

Nevertheless, this expression still generates a complex wiring on the FPGA. The following rewrite rule replaces the transposition of addresses by a more efficient expression, based on two simple counters:

$$TransposeStm \circ SplitStm\langle N\rangle \circ Counter\langle C_0, C_S, M * N\rangle \Longrightarrow MapStm(\lambda\, p_1 \Rightarrow MapStm(\lambda\, p_2 \Rightarrow Add \circ (p_1, p_2)) \circ Counter\langle 0, N, M\rangle) \circ Counter\langle C_0, C_S, N\rangle \tag{29}$$

The corresponding hardware design is shown in fig. 6b and has no more complex wiring left. After rewriting, this final, optimized expression is returned:

$$Read \circ MapStm(\lambda\, p_1 \Rightarrow MapStm(\lambda\, p_2 \Rightarrow Add \circ (p_1, p_2)) \circ Counter\langle 0, N, M\rangle) \circ Counter\langle 0, 1, N\rangle \tag{30}$$

3.4 Slide

In order to implement convolution, the hardware-agnostic IR is extended by a primitive for sliding windows, similar to the stencil operation in prior works of Lift [7]:

$$Slide\langle W, S\rangle : [T]_N \rightarrow [[T]_W]_{(N-W)/S+1} \tag{31}$$

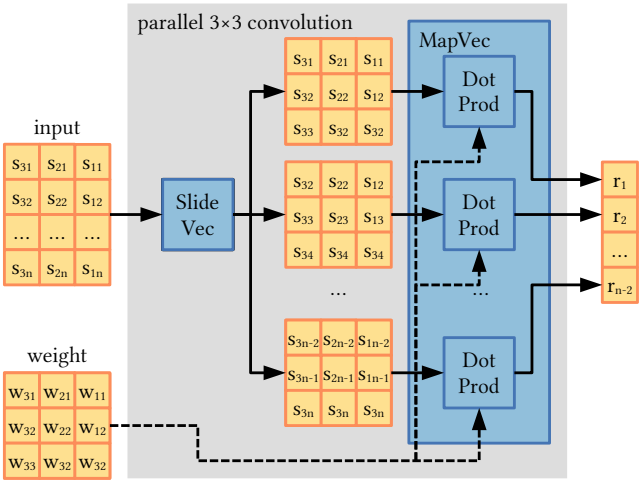


Figure 7. 3×3 convolution with slide parallelization.

For each input element, this operator produces the corresponding 'window' of W elements. Depending on the step size S, some of these outputs are skipped.

For 2D convolution, the following combination of primitives expresses a 2D sliding window operation:

$$Map(Slide\langle W_2, S_2\rangle \circ Transpose) \circ Slide\langle W_1, S_1\rangle \tag{32}$$

On the hardware-specific IR level, the slide primitive is refined to operate on streams and vectors:

$$SlideStm\langle W, S\rangle : {}^{STM}[T]_N \rightarrow {}^{STM}[{}^{VEC}[T]_W]_{(N-W)/S+1} \tag{33}$$

$$SlideVec\langle W, S\rangle : {}^{VEC}[T]_N \rightarrow {}^{VEC}[{}^{VEC}[T]_W]_{(N-W)/S+1} \tag{34}$$

With these primitives an efficient parallel convolution is achieved, as depicted in fig. 7.

3.4.1 Problem Statement. The hardware implementation of `SlideStm` is based on a shift register that emits its entire contents, the window vector, whenever a new input is received. The parallel variant `SlideVec` creates a wiring mesh in hardware, similar to fig. 6a, but even more complex, because every input signal is connected to W/S outputs. For large window sizes, either an enormous shift register or complex wiring is inferred, which may not fit onto the FPGA.

3.4.2 Slide Optimization. An optimization is required to implement the elaborate slide operation on real hardware. Similar to the optimizations for repetition and transposition, in sections 3.1 and 3.3, instead of reshaping data, the addresses of that data in memory are reorganized, which is more efficient. This is achieved by another rewrite-based fixed-point iteration over the IR. To describe this process on an example, the following initial expression is considered:

$$MapStm(VecToStm) \circ SlideStm\langle W, S\rangle \circ Read \circ Counter\langle start, step, N\rangle \tag{35}$$

First, a rewrite rule moves the `SlideStm` primitive to the input of `Read`, so that it operates on the addresses:

$$\begin{aligned} &MapStm(VecToStm) \circ SlideStm\langle W, S\rangle \circ Read \\ &\Longrightarrow Read \circ MapStm(VecToStm) \\ &\quad \circ SlideStm\langle W, S\rangle \end{aligned} \tag{36}$$

In the next step, a rewrite rule replaces the slide over addresses by a more efficient combination of counters that produces the same values:

$$\begin{aligned} &MapStm(VecToStm) \circ SlideStm\langle W, S\rangle \\ &\circ Counter\langle start, step, N\rangle \\ &\Longrightarrow MapStm(\lambda\, p_1 \Rightarrow MapStm(\lambda\, p_2 \Rightarrow Add \circ (p_1, p_2)) \\ &\quad \circ Counter\langle 0, step, W\rangle) \\ &\quad \circ Counter\langle start, S, (N - W + step)/S + 1\rangle \end{aligned} \tag{37}$$

After these rewrite optimizations, the slide expression is removed entirely. The generated hardware for the counters looks similar to the optimized transposition in fig. 6b, only the counters' configurations are different. The large shift registers or wire meshes are in effect replaced by counters, reducing area while preserving functionality.

3.5 Summary

As seen in this section, data reshaping operations must be optimized to generate efficient hardware. The key idea is to *move* these operations across the other primitives. This process will either annihilate them altogether, as seen for the conversions between streams and vectors, or it might bring the reshaping operations close to a counter, where the operation is optimized away.

4 Evaluation

In normal compiler operation, all the presented optimizations are enabled by default. To evaluate the data reshaping optimizations and show how much they improve performance, some of the rewrite rules are explicitly disabled in the following experiments.

The compiler is implemented in Scala and the input programs are written in a DSL (Domain Specific Language) embedded in Scala. The compiler generates VHDL files from these high-level specifications in just a few seconds. Quartus Prime 19.2 synthesizes these for the Intel Arria 10 GX FPGA which is connected to an Intel Xeon machine via PCIe Gen 3 x8. All synthesized designs meet the timing requirements with a clock frequency of 200 Mhz.

Tiled MxM and 2D convolution serve as the main benchmarks, operating on randomly generated input data sets. The results computed by the real FPGA are verified against a reference CPU implementation. A cycle counter on the FPGA measures the end-to-end runtime, which includes the initial input data transfer to the FPGA, as well as the transfer of the results back to host RAM.

Table 2. Generated tiled matrix multiplication designs with a different rewrite optimizations enabled. The input matrix consists of 4096×4096 8-bit integers. The tile size is 512×2048. The generated hardware designs use 2048 multipliers.

	Rewrites			Performance			Resources		
experiment no.	Transpose	Repeat	Conversion	GOPS	OPC*	DSP efficiency**	Logic blocks	On-chip RAM	DSP blocks
1	not synthesizable!								
2	✓			0.3	1	<1%	38%	53%	67%
3	✓	✓		81.9	409	20%	32%	53%	67%
4	✓	✓	✓	290.9	1455	71%	31%	53%	67%

* OPC represents parallel mul-add operations per cycle.
** DSP efficiency is the proportion of the overall runtime in which the DSPs are used, i.e., OPC / DSPs.

4.1 Tiled Matrix Multiplication

In tiled MxM, the input tiles are processed separately and accumulated directly on the FPGA to form a tile of the resulting matrix. Table 2 lists the tiled matrix multiplication experiments, with different rewrite rules enabled. They are based on the following input program code, which is all the user has to provide to the compiler. The optimizations are applied automatically, if not explicitly disabled.

```
Map(λ row1 =>
  Map(λ row2 =>
    Reduce(λ a => Add(a),
      Map(λ m => Mul(m), Zip(row1, row2))),
    matrix2),
  matrix1)
```

The first experiment shows, that with all the rewrite rules disabled, the generated design cannot be synthesized for the FPGA. The stream-based or vector-based transpositions still remaining in the IR either produce too large on-chip buffers, or too complex wiring. The Transpose rewrite rule must be enabled to generate feasible designs, as the following experiments 2–4 show.

The performance of the second experiment suffers from the repeat problem, as described in section 3.1. The efficiency of the DSPs (Digital Signal Processors) is low. Less than 1% of the overall runtime is actually used to perform useful operations on a DSP. The remaining time is spent waiting for input data to arrive due to pipeline stalls, as seen in fig. 4c, due to the need to back propagate the repeat signals.

In experiment 3 all the rewrite rules for the Repeat optimization are enabled. This leads to a few hundred times increase in performance, with a DSP efficiency of 20%.

Table 3. Generated 2D convolutional layer designs with different rewrite rules. The input image has 1024×1024 8-bit integers. The tile size is 128×128. The kernel weights are 3×3 with 3 input and 64 output channels.

	Rewrites				Performance		Resources		
experiment no.	Slide	Transpose	Repeat	Conversion	Throughput*	DSP efficiency	Logic blocks	On-chip RAM	DSP blocks
5					not synthesizable!				
6	✓				not synthesizable!				
7	✓	✓			3%	3%	17%	6%	28%
8	✓	✓	✓		3%	3%	17%	6%	28%
9	✓	✓	✓	✓	76%	80%	14%	6%	28%

* Throughput is the ratio of the experiment's read and write speeds compared to that of memcopy.

When all optimizations are enabled, the DSP efficiency of experiment 4 increases by 3.5× compared to experiment 3, so that the DSPs are used in 71% of the overall runtime. Future work will look at increasing efficiency further, by using double-buffering to overlap tile loading with computation.

4.2 2D Convolution

In this section, the rewrite rules are evaluated on tiled 2D convolution. The high-level Scala code given to the compiler specifies a full convolution as found in typical convolutional neural networks:

```
Map(λ weight =>
  Map(λ rowGroup =>
    Map(λ window =>
      Reduce(λ a => Add(a),
        Map(λ m => Mul(m), Zip(window,weight))),
      Slide(rowGroup)),
    Slide(input)),
  weightGroup)
```

The results are shown in table 3. The performance is evaluated using memory throughput, since this benchmark is memory bound. The throughput in table 3 is specified as a percentage in comparison to the maximum throughput achieved by memcopy. This value has been measured with a separate memcopy benchmark and is 6.5 GB/s, which is in line with the maximum theoretical PCIe speed.

The first two convolution experiments show that the Slide and Transpose optimization must be enabled to generate synthesizable designs. Again, without these rules the design would require too much on-chip memory or cause too complex wire routing than feasible for the FPGA.

Once the rules are applied, as in experiments 7–9, the generated hardware design is able to run on the FPGA. With more rewrite rule optimizations enabled, the throughput increases from 3% up to 76% of its maximum possible value.

The introduction of the repeat rewrite rules do not have a noticeable effect on the performance of a parallelized convolution, because only vectors are repeated here, which does not induce much overhead. However, enabling the conversion optimization does increase performance by 25×.

5 Related Work

Hardware Design Languages. Formerly "very high level" hardware design languages like VHDL, Verilog, and SystemVerilog are now considered low level. Fleet [29], Esterel [6], and Bluespec [18] raise the abstraction level but still require hardware expertise. This contrasts with hardware-agnostic approaches, such as the one presented in this paper.

High-Level Synthesis. Many approaches, *e.g.*, LegUp [4] and SOFF [9], take code in a C-like language to generate hardware. The FPGA vendors Intel and Xilinx have their own languages: Intel OpenCL SDK, Vivado HLS and SDAccel. The user provides program code annotations to instruct their compilers about hardware optimizations. To reduce this manual effort, HeteroCL [13] offers a more abstract program specification. Since these are based on C-like code, they inherit all the problems of such approaches: vendor-specific optimizations complicate performance portability [19]. Software-like representations are not well suited to describe hardware and may generate unpredictable results [17].

Besides C code, functional languages are also used for HLS, as in Cλash [1], Chisel [2], Delite [20, 28], AnyHSL [19] and others [16, 30]. There are also languages for HLS that model dataflow, like [8, 10, 27] and LiquidMetal [3]. T2S-Tensor [25] presents a framework for optimizing and generating systolic arrays for FPGA. The Spatial [11] language and compiler enable the user to provide a high-level algorithmic specification for hardware generation. However, these specifications contain hardware details after all. While these proposed approaches are a step in the right direction, they still require hardware expertise.

Other projects like DNNWeaver [23] and [15] employ large-scale templates to generate hardware. Although the templates can be configured by adjusting their parameters, these approaches are not as flexible and generic as IR-based ones and are restricted to certain application types.

Optimizations Using Rewrite Rules. Rewrite rules offer a systematic way to modify programs. This is used for design space exploration and design optimization. Similar to this work, Lift [26], Lift-hls [12], Shir [21], Aetherling [5], and Spiral [22] leverage the mechanism of rewrite rules. As seen in the motivation of this paper, all these approaches lack optimizations for data reshaping operations and are unable to express efficient tiled MxM and 2D convolution.

Shir [21] is a framework to synthesize hardware designs from a functional IR. Its focus lies on the ability to encode arbitrary memory usage in the IR. Similar to this paper, Shir uses rewrite rules to optimize their memory design and increase the design's performance. This paper builds on top of Shir but, instead, focuses on data reshaping optimizations using rewrite rules and introduces tiled designs for matrix multiplication and 2D convolution.

Glenside [24] presents an IR with access patterns to circumvent the issue of implicit repetitions by avoiding name binding (lambdas) and higher order functions altogether. This, however, prevents the programmer from using well-known functional patterns, *e.g.*, `Map`, and puts the additional task of choosing the right access patterns on the programmer's shoulders. Moreover, this requires the definition of specialized operators like `reduceSum` and `dotProd` in the Glenside IR. In contrast, the high-level IR presented in this paper supports lambdas and higher order functions, which enable the programmer to specify arbitrary programs on any dimensional data in a familiar way with only a few common functional primitives. Furthermore, Glenside does not generate hardware designs but rewrites the program to match a predefined implementation, not exploiting the flexibility of the FPGA. They neither discuss how expensive operations like transposition could be achieved in hardware, nor show any performance results.

6 Conclusion

As seen in this paper, a functional approach for HLS is challenging in the presence of data reshaping operations. If not handled carefully, such operations can lead to incorrect designs (repetition), non-synthesizable design (transpose, slide) or poor performance (repetition, stream/vector conversion).

This paper has shown how all these challenges are solved through the introduction of an explicit repeat primitive and through the application of simple rewrite rules. These rewrite rules optimize the design by moving the data reshaping operations in the IR closer to the counters, or even by removing these operations altogether.

The evaluation on a real Intel Arria 10 FPGA shows that performance is increased by up to 25× for matrix multiplication and convolution applications, when applying the optimizations presented. In addition, we have seen that without these optimizations, the designs require too many resources to be synthesized for the FPGA.

Acknowledgments

This work was supported by Microsoft Research through its PhD Scholarship Programme, and by a Facebook Research Award. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program.

References

[1] C.P.R. Baaij. 2015. *Digital circuit in CλaSH: functional specifications and type-directed synthesis*. Ph.D. Dissertation. University of Twente, Netherlands. https://doi.org/10.3990/1.9789036538039 eemcs-eprint-23939.

[2] 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 *Proceedings of the 49th Annual Design Automation Conference (DAC)*.

[3] David F. Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses. *Commun. ACM* 56, 4 (April 2013), 56–63. https://doi.org/10.1145/2436256.2436271

[4] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA)*. ACM, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[5] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-Directed Scheduling of Streaming Accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[6] Stephen A Edwards. 2002. High-Level Synthesis from the Synchronous Language Esterel. In *IWLS*. 401–406.

[7] 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* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 100–112. https://doi.org/10.1145/3168824

[8] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. 2008. Optimus: Efficient Realization of Streaming Applications on FPGAs. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Atlanta, GA, USA) *(CASES)*. 41–50. https://doi.org/10.1145/1450095.1450105

[9] G. Jo, H. Kim, J. Lee, and J. Lee. 2020. SOFF: An OpenCL High-Level Synthesis Framework for FPGAs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 295–308.

[10] 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.

[11] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[12] 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)*.

[13] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 242–251. https://doi.org/10.1145/3289602.3293910

[14] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[15] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A Hardware–Software Blueprint for Flexible Deep Learning Specialization. *IEEE Micro* 39, 5 (2019), 8–16. https://doi.org/10.1109/MM.2019.2928962

[16] Alan Mycroft and Richard Sharp. 2000. A Statically Allocated Parallel Functional Language. In *ICALP*. 37–48.

[17] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable Accelerator Design with Time-Sensitive Affine Types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*.

[18] Rishiyur S. Nikhil. 2008. *Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions*. Springer Netherlands, Dordrecht, 129–146. https://doi.org/10.1007/978-1-4020-8588-8_8

[19] M Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, Sebastian Hack, Jürgen Teich, and Frank Hannig. 2020. AnyHLS: High-Level Synthesis With Partial Evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3202–3214.

[20] 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 *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS)*. 651–665. https://doi.org/10.1145/2872362.2872415

[21] Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. 2022. Memory-Aware Functional IR for Higher-Level Synthesis of Accelerators. *ACM Trans. Archit. Code Optim.* 19, 2, Article 16 (jan 2022), 26 pages. https://doi.org/10.1145/3501768

[22] François Serre and Markus Püschel. 2019. DSL-Based Hardware Generation with Scala: Example Fast Fourier Transforms and Sorting Networks. 13, 1, Article 1 (Dec. 2019), 23 pages. https://doi.org/10.1145/3359754

[23] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (10 2016). https://doi.org/10.1109/micro.2016.7783720

[24] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure Tensor Program Rewriting via Access Patterns (Representation Pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 21–31. https://doi.org/10.1145/3460945.3464953

[25] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. Albonesi, V. Sarkar, W. Chen, P. Petersen, G. Lowney, A. Herr, C. Hughes, T. Mattson, and P. Dubey. 2019. T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 181–189. https://doi.org/10.1109/FCCM.2019.00033

[26] 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 *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

[27] 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

[28] 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

[29] James Thomas, Pat Hanrahan, and Matei Zaharia. 2020. Fleet: A framework for massively parallel streaming on FPGAs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 639–651.

[30] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. 2017. From Functional Programs to Pipelined Dataflow Circuits. In *Proceedings of the 26th International Conference on Compiler Construction* (Austin, TX, USA) *(CC)*. 11 pages.